ObjectDetector interface C++ library

v1.7.0

Table of contents

- How to create a custom implementation

Overview

ObjectDetector C++ library provides standard interface, but also defines data structures and rules for different object detectors (motion detectors, events detectors, neural networks etc.). **ObjectDetector** interface class does not do anything, just provides interface, defines data structures and provides methods to encode/decode commands and encode/decode (serialize/deserialize) parameters. Different object detector classes inherit interface form **ObjectDetector** C++ class. **ObjectDetector.h** file contains ObjectDetectorCommand enum (provides enum of action commands), ObjectDetectorParam enum (provides enum of params), ObjectDetectorParams class (contains list of params) and includes ObjectDetector class declaration. All object detectors should include params and commands listed in **ObjectDetector.h** file. **ObjectDetector**class depends on two external libraries (included as submodules): Frame (video frame data structure) and ConfigReader (provides methods to read/write JSON config files).

Versions

Table 1 - Library versions.

Version	Release date	What's new
1.0.0	17.07.2023	First version.
1.0.1	17.07.2023	- 3rdparty variable name mistake fixed.
1.1.0	18.07.2023	- Added frame ID field for detection results.
1.2.0	22.08.2023	- Added new setMask method.
1.3.0	12.09.2023	- Changed params serialization method.
1.4.0	24.09.2023	- Updated encode(...) and decode(...) methods of ObjectDetectorParams. - Added decodeAndExecuteCommand(...) method. - Added example of object detector implementation.
1.5.0	26.09.2023	- Signature of getParams(...) method changed.
1.5.1	13.11.2023	- Frame class updated.
1.6.1	14.12.2023	- Virtual destructor added. - Frame class updated.
1.7.0	08.01.2024	- List of objects class names added to ObjectDetectorParams. - Data structures updated. - Added automatic linux build check based on GitHub Actions.

Library files

The **ObjectDetector** library is a CMake project. Library files:

```
CMakeLists.txt ------------------- Main CMake file of the library.
3rdparty ------------------------- Folder with third-party libraries.
    CMakeLists.txt ---------------- CMake file to include third-party libraries.
    ConfigReader ------------------ Source code of the ConfigReader library.
    Frame ------------------------- Source code of the Frame library.
example -------------------------- Folder with custom (example) object detector class.
    CMakeLists.txt ---------------- CMake file for custom object detector class.
    CustomObjectDetector.cpp ------ Source code file of example object detector class.
    CustomObjectDetector.h -------- Header with object detector class declaration.
    CustomObjectDetectorVersion.h - Header file which includes class version.
    CustomObjectDetectorVersion.h.in - CMake service file to generate version file.
test ----------------------------- Folder with test of data structures.
    CMakeLists.txt ---------------- CMake file for test application.
    main.cpp ---------------------- Source code file of test application.
src ------------------------------ Folder with source code of the library.
    CMakeLists.txt ---------------- CMake file of the library.
    ObjectDetector.cpp ------------ Source code file of the library.
    ObjectDetector.h -------------- Header file with ObjectDetector class declaration.
    ObjectDetectorVersion.h ------- Header file with class version.
    ObjectDetectorVersion.h.in ---- CMake service file to generate version file.
```

ObjectDetector interface class description

ObjectDetector class declaration

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

```
class ObjectDetector
{
public:

    /// Class destructor.
    virtual ~ObjectDetector();

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

    /// Init object detector.
    virtual bool initObjectDetector(ObjectDetectorParams& params) = 0;

    /// Set object detector param.
    virtual bool setParam(ObjectDetectorParam id, float value) = 0;

    /// Get object detector param value.
    virtual float getParam(ObjectDetectorParam id) = 0;
```

```
    /// Get object detector params structure.
    virtual void getParams(ObjectDetectorParams& params) = 0;

    /// Get list of detected objects.
    virtual std::vector<Object> getObjects() = 0;

    /// Execute command.
    virtual bool executeCommand(ObjectDetectorCommand id) = 0;

    /// Perform detection.
    virtual bool detect(cr::video::Frame& frame) = 0;

    /// Set detection mask.
    virtual bool setMask(cr::video::Frame mask) = 0;

    /// Encode set param command.
    static void encodeSetParamCommand(
            uint8_t* data, int& size, ObjectDetectorParam id, float value);

    /// Encode action command.
    static void encodeCommand(
            uint8_t* data, int& size, ObjectDetectorCommand id);

    /// Decode command.
    static int decodeCommand(uint8_t* data, int size,
                             ObjectDetectorParam& paramId,
                             ObjectDetectorCommand& commandId,
                             float& value);

    /// Decode and execute command.
    virtual bool decodeAndExecuteCommand(uint8_t* data, int size) = 0;
};
```

getVersion method

getVersion() method returns string of current version of **ObjectDetector** class. Particular object detector class can have it's own **getVersion()** method. Method declaration:

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

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

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

Console output:

```
ObjectDetector class version: 1.7.0
```

initObjectDetector method

initObjectDetector(...) method initializes object detector. Particular object detector class should initialize only supported parameters from **ObjectDetectorParams** class. Method declaration:

```
virtual bool initObjectDetector(ObjectDetectorParams& params) = 0;
```

Parameter	Value
params	Object detector parameters class. Particular object detector class may not support all params listed in ObjectDetectorParams class. If object detector doesn't support particular params this params should have default values.

Returns: TRUE if the object detector initialized or FALSE if not.

setParam method

setParam(...) method sets new object detector param value. Particular object detector class must provide thread-safe **setParam(...)** method call. This means that the **setParam(...)** method can be safely called from any thread. Method declaration:

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

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

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

getParam method

getParam(...) method returns object detector parameter value. Particular object detector class must provide thread-safe **getParam(...)** method call. This means that the **getParam(...)** method can be safely called from any thread. Method declaration:

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

Parameter	Description
id	Parameter ID according to ObjectDetectorParam enum.

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

getParams method

getParams(...) method returns object detector params class object as well a list of detected objects. Particular object detector class must provide thread-safe **getParams(...)** method call. This means that the **getParams(...)** method can be safely called from any thread. Method declaration:

```
virtual void getParams(ObjectDetectorParams& params) = 0;
```

Parameter	Description
params	Output ObjectDetectorParams class object.

getObjects method

getObjects() method returns list of detected objects. User can object list of detected objects via **getParams(...)** method as well. Particular object detector class must provide thread-safe **getObjects(...)** method call. This means that the **getObjects(...)** method can be safely called from any thread. Method declaration:

```
virtual std::vector<Object> getObjects() = 0;
```

Returns: list of detected objects (see **Object** class description). If no detected object the list will be empty.

executeCommand method

executeCommand(...) method executes object detector command. Particular object detector class must provide thread-safe **executeCommand(...)** method call. This means that the **executeCommand(...)** method can be safely called from any thread. Method declaration:

```
virtual bool executeCommand(ObjectDetectorCommand id) = 0;
```

Parameter	Description
id	Command ID according to ObjectDetectorCommand enum.

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

detect method

detect(...) method performs detection algorithm. Method declaration:

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

Parameter	Description
frame	Video frame for processing. Object detector processes only RAW pixel formats (BGR24, RGB24, GRAY, YUYV24, YUYV, UYVY, NV12, NV21, YV12, YU12, see Frame class description). Required pixel format depends on implementation. It is recommended to support all possible RAW pixel formats supported by Frame object.

Returns: TRUE if the video frame was processed or FALSE if not. If object detector disabled (see ObjectDetectorParam enum description) the method should return TRUE.

setMask method

setMask(...) method designed to set detection mask. Method declaration:

```
virtual bool setMask(cr::video::Frame mask) = 0;
```

Parameter	Description
mask	Detection mask is see **Frame** object with GRAY pixel format. Detector omits image segments, where detection mask pixel values equal 0.

Returns: TRUE if the detection mask was set or FALSE if not.

encodeSetParamCommand method

encodeSetParamCommand(...) static method designed to encode command to change any parameter for remote object detector. To control object detector 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 **ObjectDetector** class contains static methods for encoding the control command. The **ObjectDetector** class provides set param commands and an action command. **encodeSetParamCommand(...)** encodes set param commands. Method declaration:

```
static void encodeSetParamCommand(uint8_t* data, int& size, ObjectDetectorParam 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 ObjectDetectorParam enum.
value	Parameter value. Value depends on parameter ID.

SET_PARAM command format (11 bytes):

Byte	Value	Description
0	0x01	SET_PARAM command header value.
1	Major	Major version of ObjectDetector class.
2	Minor	Minor version of ObjectDetector 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 **ObjectDetector** class instance. This method used on client side (control system). Command encoding example:

```
// Buffer for encoded data.
uint8_t data[11];
// Size of encoded data.
int size = 0;
// Random parameter value.
float outValue = (float)(rand() % 20);
// Encode command.
ObjectDetector::encodeSetParamCommand(data, size, ObjectDetectorParam::MIN_OBJECT_WIDTH,
outValue);
```

encodeCommand method

encodeCommand(...) static method designed to encode command for remote object detector. To control object detector 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 **ObjectDetector** class contains static methods for encoding the control command. The **ObjectDetector** class provides set param commands and an action command. **encodeCommand(...)** encodes action commands. Method declaration:

```
static void encodeCommand(uint8_t* data, int& size, ObjectDetectorCommand id);
```

Parameter	Description
data	Pointer to data buffer for encoded command. Must have size >= 7.
size	Size of encoded data. Will be 7 bytes.

Parameter	Description
id	Command ID according to ObjectDetectorCommand enum.

COMMAND format (7 bytes):

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

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

```
// Buffer for encoded data.
uint8_t data[7];
// Size of encoded data.
int size = 0;
// Encode command.
ObjectDetector::encodeCommand(data, size, ObjectDetectorCommand::RESET);
```

decodeCommand method

decodeCommand(...) static method decodes command on object detector side (edge device). Method declaration:

```
static int decodeCommand(uint8_t* data, int size, ObjectDetectorParam& paramId,
ObjectDetectorCommand& commandId, float& value);
```

Parameter	Description
data	Pointer to input command.
size	Size of command. 11 bytes for set param command and 7 bytes for action command.
paramId	Output (decoded) parameter ID according to ObjectDetectorParam enum. After decoding set param command the method will return parameter ID.
commandId	Output (decoded) command ID according to ObjectDetectorCommand enum. After decoding COMMAND the method will return command ID.

Parameter	Description
value	Parameter value (after decoding set param command).

Returns: 0 - in case decoding action command, **1** - in case decoding set param command or **-1** in case error.

decodeAndExecuteCommand method

decodeAndExecuteCommand(...) method decodes and executes command on object detector side (edge device). Method decodes commands which encoded which prepared with encodeSetParamCommand(...) and encodeCommand(...) methods and call setParam(...) or executeCommand(...) methods. The particular implementation of the object detector must provide thread-safe **decodeAndExecuteCommand(...)** method call. This means that the **decodeAndExecuteCommand(...)** method can be safely called from any thread. Method declaration:

```
virtual bool decodeAndExecuteCommand(uint8_t* data, int size) = 0;
```

Parameter	Description
data	Pointer to input command.
size	Size of command. Must be 11 bytes for set param command or 7 bytes for action command.

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

Data structures

ObjectDetectorCommand enum

ObjectDetectorCommand enum represents action commands. Enum declaration:

```
enum class ObjectDetectorCommand
{
    /// Reset.
    RESET = 1,
    /// Enable.
    ON,
    /// Disable.
    OFF
};
```

Table 2 - Object detector action commands description. Some commands maybe unsupported by particular object detector class.

Command	Description
RESET	Reset object detector. After reset the list of detected object should be empty until new objects will be detected.
ON	Enable object detector.
OFF	Disable object detector. If object detector disabled the list of detected objects must be empty.

ObjectDetectorParam enum

ObjectDetectorParam enum represents detector parameters. 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_PROBABILITY.
    MIN_DETECTION_PROBABILITY,
    /// 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. Defines where log information will be printed in. Values: 0 - Disable, 1 - Only file, 2 - Only terminal (console), 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.

Parameter	Access	Description
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.
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_PROBABILITY	read / write	Probability threshold from 0 to 1. To be detected object detection probability must be >= MIN_DETECTION_PROBABILITY. For example: neural networks for each detection result calculates detection probability from 0 to 1. MIN_DETECTION_PROBABILITY 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.

Parameter	Access	Description
PROCESSING_TIME_MKS	read only	Processing time for last frame, microseconds.
TYPE	read / write	Algorithm / backend type. Depends on implementation.
MODE	read / write	Mode. Default: 0 - Off, 1 - On.
CUSTOM_1	read / write	Custom parameter. Depends on implementation.
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:

```
struct Object
{
    /// Object ID. Must be unique for particular object.
    int id{ 0 };
    /// Frame ID. Must be the same as frame ID of processed video frame.
    int frameId{ 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 };
};
```

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.
frameId	int	Frame ID. Must be the same as frame ID of processed video 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).
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 >= MIN_DETECTION_PROBABILITY (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** provides structure to write/read params from JSON files (**JSON_READABLE** macro, see **ConfigReader** class description) and provide methos to encode (serialize) and decode (deserialize) 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 / backend 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;
    // A list of object class names used in detectors that recognize different
    // object classes. Detected objects have an attribute called 'type.'
    // If a detector doesn't support object class recognition or can't determine
    // the object type, the 'type' field must be set to 0. Otherwise, the 'type'
    // should correspond to the ordinal number of the class name from the
    // 'classNames' list (if the list was set in params), starting from 1
    // (where the first element in the list has 'type == 1').
    std::vector<std::string> classNames{ "" };

    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,
                  classNames);

    /// Copy operator.
    ObjectDetectorParams& operator= (const ObjectDetectorParams& src);

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

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

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.

Field	Type	Description
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.
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.

Field	Type	Description
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.
classNames	std::vector	A list of object class names used in detectors that recognize different object classes. Detected objects have an attribute called 'type.' If a detector doesn't support object class recognition or can't determine the object type, the 'type' field must be set to 0. Otherwise, the 'type' should correspond to the ordinal number of the class name from the 'classNames' list (if the list was set in params), starting from 1 (where the first element in the list has 'type == 1').

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 (4 bytes) where each bit represents particular parameter and **decode(...)** method recognizes it. Method declaration:

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

Parameter	Value
data	Pointer to data buffer. Buffer size should enough size.
dataBufferSize	Size of data buffer. If the data buffer size is not large enough to serialize all detected objects (40 bytes per object), not all objects will be included in the data. If buffer has not enough size to encode all parameters the method will return FALSE.
size	Size of encoded data.
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.

Returns: TRUE is params serialized or FALSE if not.

ObjectDetectorParamsMask structure declaration:

```
struct ObjectDetectorParamsMask
{
    bool initString{ true };
    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 };
    bool classNames{ true };
};
```

Example without parameters mask:

```
// Prepare random params.
ObjectDetectorParams in;
in.logMode = rand() % 255;
in.classNames = { "apple", "banana", "orange", "pineapple", "strawberry",
                  "watermelon", "lemon", "peach", "pear", "plum" };
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[2048];
int size = 0;
in.encode(data, 2048, size);
cout << "Encoded data size: " << size << " bytes" << endl;
```

Example with parameters mask:

```
// Prepare random params.
ObjectDetectorParams in;
in.logMode = rand() % 255;
in.classNames = { "apple", "banana", "orange", "pineapple", "strawberry",
                  "watermelon", "lemon", "peach", "pear", "plum" };
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; // Exclude parameter from serialization.

// Encode data.
uint8_t data[2048];
int size = 0;
in.encode(data, 2048, 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 declaration:

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

Parameter	Value
data	Pointer to encode data buffer.
dataSize	Data size.

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

Example:

```
// Prepare random params.
ObjectDetectorParams in;
in.logMode = rand() % 255;
in.classNames = { "apple", "banana", "orange", "pineapple", "strawberry",
                  "watermelon", "lemon", "peach", "pear", "plum" };
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[2048];
int size = 0;
in.encode(data, 2048, size);
cout << "Encoded data size: " << size << " bytes" << endl;

// Decode data.
ObjectDetectorParams out;
if (!out.decode(data, size))
{
    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.classNames = { "apple", "banana", "orange", "pineapple", "strawberry",
                  "watermelon", "lemon", "peach", "pear", "plum" };
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": {
        "classNames": [
            "apple",
            "banana",
            "orange",
            "pineapple",
            "strawberry",
            "watermelon",
            "lemon",
            "peach",
            "pear",
            "plum"
        ],
        "custom1": 72.0,
        "custom2": 162.0,
        "custom3": 178.0,
        "enable": false,
        "frameBufferSize": 63,
        "initString": "sfsfsfsfsf",
        "logMode": 33,
        "maxObjectHeight": 68,
        "maxObjectWidth": 57,
```

```
        "maxXSpeed": 210.0,
        "maxYSpeed": 243.0,
        "minDetectionProbability": 52.0,
        "minObjectHeight": 244,
        "minObjectWidth": 178,
        "minXSpeed": 53.0,
        "minYSpeed": 198.0,
        "numThreads": 143,
        "resetCriteria": 8,
        "scaleFactor": 209,
        "sensitivity": 236.0,
        "type": 127,
        "xDetectionCriteria": 180,
        "yDetectionCriteria": 161
    }
}
```

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 the folder **3rdparty/ObjectDetector** will be created 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)
    SET(${PARENT}_OBJECT_DETECTOR_EXAMPLE               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 and example (ObjectDetector class test applications and example) 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!

How to create a custom implementation

The **ObjectDetector** class provides only an interface, data structures, and methods for encoding and decoding commands and params. To create your own implementation of the object detector, you must include the ObjectDetector repository in your project (see **Build and connect to your project** section). The catalogue **example** (see **Library files** section) includes an example of the design of the custom object detector. You must implement all the methods of the ObjectDetector interface class. Custom object detector class declaration:

```
class CustomObjectDetector: public ObjectDetector
{
public:

    /// Class constructor.
    CustomObjectDetector();

    /// Class destructor.
    ~CustomObjectDetector();

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

    /// Init object detector.
    bool initObjectDetector(ObjectDetectorParams& params);

    /// Set object detector param.
    bool setParam(ObjectDetectorParam id, float value);

    /// Get object detector param value.
    float getParam(ObjectDetectorParam id);
```

```
    /// Get object detector params structure.
    void getParams(ObjectDetectorParams& params);

    /// Get list of objects.
    std::vector<Object> getObjects();

    /// Execute command.
    bool executeCommand(ObjectDetectorCommand id);

    /// Perform detection.
    bool detect(cr::video::Frame& frame);

    /// Set detection mask.
    bool setMask(cr::video::Frame mask);

    /// Decode command.
    bool decodeAndExecuteCommand(uint8_t* data, int size);

private:

    /// Object detector params (default params).
    ObjectDetectorParams m_params;
};
```