```
/*
 * CvProcessor.h
 *
 *  Created on: 21 fÃ©vr. 2012
 *      Author: davidroussel
 */

#ifndef CVPROCESSOR_H_
#define CVPROCESSOR_H_

#include <string>
#include <map>
#include <ctime>    // for clock
using namespace std;

#include <opencv2/core/core.hpp>    // for Mat
using namespace cv;

#include "CvProcessorException.h"

/**
 * Class to process a source image with OpenCV 2+
 */
class CvProcessor
{
    public:

        /**
         * Verbose level for error / warnings / notification messages
         */
        typedef enum
        {
            VERBOSE_NONE = 0,   //!< no messages are displayed
            VERBOSE_ERRORS,   //!< only error messages are displayed
            VERBOSE_WARNINGS,   //!< error & warning messages are displayed
            VERBOSE_NOTIFICATIONS, //!< error, warning and notifications messages are displayed
            VERBOSE_ACTIVITY, //!< all previouses + log messages
            NBVERBOSELEVEL
        } VerboseLevel;


        /**
         * Index of channels in OpenCV BGR or Gray images
         */
        typedef enum
        {
            BLUE = 0,//!< Blue component is first in BGR images
            GRAY = 0,//!< Gray component is first in gray images
            GREEN,   //!< Green component is second in BGR images
            RED,     //!< Red component is last in BGR images
            NBCHANNELS
        } Channels;

    protected:
        /**
         * The source image: CV_8UC<nbChannels>
         */
        Mat * sourceImage;

        /**
         * Source image number of channels (generally 1 or 3)
         */
        int nbChannels;

        /**
         * Source image size (cols, rows)
         */
        Size size;

        /**
         * The source image type (generally CV_8UC<nbChannels>)
         */
        int type;

        /**
         * Map to store aditionnal images pointers by name
         */
        map<string, Mat*> images;

        /**
         * The verbose level for printed messages
         */
```



```
        VerboseLevel verboseLevel;

        /**
         * Process time in ticks (~1e6 ticks/second)
         * @see clock_t for details on ticks
         */
        clock_t processTime;

        /**
         * Indicates if processing time is absolute or measured in ticks/feature
         * processed by this processor.
         * A feature can be any kind of things the processor has to detect or
         * create while processing an image.
         */
        bool timePerFeature;

    public:
        /**
         * OpenCV image processor constructor
         * @param sourceImage the source image
         * @param verbose level for printed messages
         * @pre source image is not NULL
         */
        CvProcessor(Mat * sourceImage,
                    const VerboseLevel level = VERBOSE_NONE);

        /**
         * OpenCV image Processor destructor
         */
        virtual ~CvProcessor();

        /**
         * OpenCV image Processor abstract Update
         * @note this method should be implemented in sub classes
         */
        virtual void update() = 0;

        // ------------------------------------------------------------------
        // Images accessors
        // ------------------------------------------------------------------
        /**
         * Changes source image
         * @param sourceImage the new source image
         * @throw CvProcessorException#NULL_IMAGE when new source image is NULL
         * @note this method should NOT be directly reimplemented in sub classes
         * unless it is transformed into a QT slot
         */
        virtual void setSourceImage(Mat * sourceImage)
            throw (CvProcessorException);

        /**
         * Adds a named image to additionnal images
         * @param name the name of the image
         * @param image the image reference
         * @return true if image has been added to additionnal images map, false
         * if image key (the name) already exists in the addtitionnal images map.
         */
        bool addImage(const char * name, Mat * image);

        /**
         * Adds a named image to additionnal images
         * @param name the name of the image
         * @param image the image reference
         * @return true if image has been added to additionnal images map, false
         * if image key (the name) already exists in the addtitionnal images map.
         */
        bool addImage(const string & name, Mat * image);

//      /*
//       * Update named image in additionnal images.
//       * @param name the name of the image
//       * @param image the image reference
//       * @post the image located at key name is updated.
//       */
//      virtual void updateImage(const char * name, const Mat & image);
//
//      /*
//       * Update named image in additionnal images.
//       * @param name the name of the image
//       * @param image the image reference
//       * @post the image located at key name is updated.
//       */
```

```
//        virtual void updateImage(const string & name, const Mat & image);

        /**
         * Get image by name
         * @param name the name of the image we're looking for
         * @return the image registered by this name in the additionnal images
         * map
         * @throw CvProcessorException#INVALID_NAME is used name is not already
         * registerd in the images
         */
        const Mat & getImage(const char * name) const
            throw (CvProcessorException);

        /**
         * Get image by name
         * @param name the name of the image we're looking for
         * @return the image registered by this name in the additionnal images
         * map
         * @throw CvProcessorException#INVALID_NAME is used name is not already
         * registerd in the images
         */
        const Mat & getImage(const string & name) const
            throw (CvProcessorException);

        /**
         * Get image pointer by name
         * @param name the name of the image we're looking for
         * @return the image pointer registered by this name in the additionnal
         * images map
         * @throw CvProcessorException#INVALID_NAME is used name is not already
         * registerd in the images
         */
        Mat * getImagePtr(const char * name)
            throw (CvProcessorException);

        /**
         * Get image pointer by name
         * @param name the name of the image we're looking for
         * @return the image registered by this name in the additionnal images
         * map
         * @throw CvProcessorException#INVALID_NAME is used name is not already
         * registerd in the images
         */
        Mat * getImagePtr(const string & name)
            throw (CvProcessorException);
        // -------------------------------------------------------------------
        // Options settings and gettings
        // -------------------------------------------------------------------
        /**
         * Number of channels in source image
         * @return the number of channels of source image
         */
        int getNbChannels() const;

        /**
         * Type of the source image
         * @return the openCV type of the source image
         */
        int getType() const;

        /**
         * Get the current verbose level
         * @return the current verbose level
         */
        VerboseLevel getVerboseLevel() const;

        /**
         * Set new verbose level
         * @param level the new verobse level
         */
        virtual void setVerboseLevel(const VerboseLevel level);

        /**
         * Return processor processing time of step index [default implementation
         * returning only processTime, should be reimplemented in subclasses]
         * @param index index of the step which processing time is required,
         * 0 indicates all steps, and values above 0 indicates step #. If
         * required index is bigger than number of steps than all steps value
         * should be returned.
         * @return the processing time of step index.
         * @note should be reimplemented in subclasses in order to define
         * time/feature behaviour
```



```
         */
        virtual double getProcessTime(const size_t index = 0) const;

        /**
         * Indicates if processing time is per feature processed in the current
         * image or absolute
         * @return
         */
        bool isTimePerFeature() const;

        /**
         * Sets Time per feature processing time unit
         * @param value the time per feature value (true or false)
         */
        virtual void setTimePerFeature(const bool value);

    protected:
        // -------------------------------------------------------------------
        // Setup and cleanup attributes
        // -------------------------------------------------------------------
        /**
         * Setup internal attributes according to source image
         * @param sourceImage a new source image
         * @param fullSetup full setup is needed when source image is changed
         * @pre sourceimage is not NULL
         * @note this method should be reimplemented in sub classes
         */
        virtual void setup(Mat * sourceImage, const bool fullSetup = true);

        /**
         * Clean up internal attributes before changing source image or
         * cleaning up class before destruction
         * @note this method should be reimplemented in sub classes
         */
        virtual void cleanup();
};

#endif /* CVPROCESSOR_H_ */
```

```
/*
 * CvProcessor.cpp
 *
 *  Created on: 21 fÃ©vr. 2012
 *      Author: davidroussel
 */


#include "CvProcessor.h"

/*
 * OpenCV image processor constructor
 * @param sourceImage the source image
 * @pre source image is not NULL
 */
CvProcessor::CvProcessor(Mat *sourceImage, const VerboseLevel level) :
    sourceImage(sourceImage),
    nbChannels(sourceImage→channels()),
    size(sourceImage→size()),
    type(sourceImage→type()),
    verboseLevel(level),
    processTime(0),
    timePerFeature(false)
{
    // No dynamic links in constructors, so this setup will always be
    // CvProcessor::setup
    setup(sourceImage, false);
}

/*
 * OpenCV image Processor destructor
 */
CvProcessor::~CvProcessor()
{
    // No Dynamic link in destructors ?
    cleanup();

    map<string, Mat*>::const_iterator cit;
    for (cit = images.begin(); cit ≠ images.end(); ++cit)
    {
        // Release handle to evt deallocate data
        /*
         * Since this is a pointer it should be necessary to release data
         */
        cit→second→release();
    }
    // Calls destructors on all elements
    images.clear();
}

/*
 * Setup internal attributes according to source image
 * @param sourceImage a new source image
 * @param fullSetup full setup is needed when source image is changed
 * @pre sourceimage is not NULL
 * @note this method should be reimplemented in sub classes
 */
void CvProcessor::setup(Mat *sourceImage, const bool fullSetup)
{
    if (verboseLevel ≥ VERBOSE_ACTIVITY)
    {
        clog << "CvProcessor::"<< (fullSetup ? "full " : "") <<"setup" << endl;
    }

    // Full setup starting point (==> previous cleanup)
    if (fullSetup)
    {
        this→sourceImage = sourceImage;
        nbChannels = sourceImage→channels();
        size = sourceImage→size();
        type = sourceImage→type();
    }

    // Partial setup starting point  (==> in any cases)
    processTime = (clock_t) 0;
    addImage("source", this→sourceImage);
}

/*
 * Clean up internal atrtibutes before changing source image or
 * cleaning up class before destruction
 * @note this method should be reimplemented in sub classes
```



```
 */
void CvProcessor::cleanup()
{
    if (verboseLevel ≥ VERBOSE_ACTIVITY)
    {
        clog << "CvProcessor::cleanup()" << endl;
    }

    // remove source pointer
    map<string, Mat*>::iterator it;
    for (it = images.begin(); it ≠ images.end(); ++it)
    {
        if (it→first ≡ "source")
        {
            images.erase(it);
            break;
        }
    }
}

/*
 * Changes source image
 * @param sourceImage the new source image
 * @throw CvProcessorException#NULL_IMAGE when new source image is NULL
 */
void CvProcessor::setSourceImage(Mat *sourceImage)
    throw (CvProcessorException)
{
    // clean up current attributes
    cleanup();

    if (sourceImage ≡ NULL)
    {
        clog << "CvProcessor::setSourceImage NULL sourceImage" << endl;
        throw CvProcessorException(CvProcessorException::NULL_IMAGE);
    }

    // setup attributes again
    setup(sourceImage);
}

/*
 * Adds a named image to additionnal images
 * @param name the name of the image
 * @param image the image reference
 * @return true if image has been added to additionnal images map, false
 * if image key (the name) already exists in the addtitionnal images map.
 */
bool CvProcessor::addImage(const char *name, Mat * image)
{
    string sname(name);

    return addImage(sname, image);
}

/*
 * Adds a named image to additionnal images
 * @param name the name of the image
 * @param image the image reference
 * @return true if image has been added to additionnal images map, false
 * if image key (the name) already exists in the addtitionnal images map.
 */
bool CvProcessor::addImage(const string & name, Mat * image)
{
    if (verboseLevel ≥ VERBOSE_ACTIVITY)
    {
        clog << "Adding image " << name << "@[" << (long)(image) << "] in" << endl;
        // Show map content before adding image
        map<string, Mat*>::const_iterator cit;
        for (cit = images.begin(); cit ≠ images.end(); ++cit)
        {
            clog  << "\t" << cit→first << "@["<< (long)(cit→second) << "]" << endl;
        }
    }

    pair<map<string,Mat*>::iterator,bool> ret;
    bool retValue;
    ret = images.insert(pair<string, Mat*>(name, image));

    if (ret.second ≡ false)
    {
        if (verboseLevel ≥ VERBOSE_WARNINGS)
```

```
165         {
166             cerr << "CvProcessor::addImage(\"" << name
167                  << "\",...) : already added" << endl;
168         }
169 
170         retValue = false;
171     }
172     else
173     {
174         retValue = true;
175     }
176 
177     return retValue;
178 }
179 /*
180  * Update named image in additionnal images.
181  * @param name the name of the image
182  * @param image the image reference
183  * @post the image located at key name is updated.
184  */
185 //void CvProcessor::updateImage(const char * name, Mat * image)
186 //{
187 //  // Search for this name in the map
188 //  map<string, Mat*>::iterator it;
189 //  for (it = images.begin(); it != images.end(); ++it)
190 //  {
191 //      if (it->first == name)
192 //      {
193 //          (it->second->release();
194 //          images.erase(it);
195 //      }
196 //  }
197 //
198 //  string sname(name);
199 //
200 //  updateImage(sname, image);
201 //}
202 
203 /*
204  * Update named image in additionnal images.
205  * @param name the name of the image
206  * @param image the image reference
207  * @post the image located at key name is updated.
208  */
209 //void CvProcessor::updateImage(const string & name, const Mat & image)
210 //{
211 //  clog << "update image " << name << " with " << (long) &image << endl;
212 //  images.erase(name);
213 //
214 //  addImage(name, image);
215 //}
216 
217 /*
218  * Get image by name
219  * @param name the name of the image we're looking for
220  * @return the image registered by this name in the additionnal images
221  * map
222  * @throw CvProcessorException#INVALID_NAME is used name is not already
223  * registerd in the images
224  */
225 const Mat & CvProcessor::getImage(const char *name) const
226     throw (CvProcessorException)
227 {
228     string sname(name);
229 
230     return getImage(sname);
231 }
232 
233 /*
234  * Get image pointer by name
235  * @param name the name of the image we're looking for
236  * @return the image pointer registered by this name in the additionnal
237  * images map
238  * @throw CvProcessorException#INVALID_NAME is used name is not already
239  * registerd in the images
240  */
241 const Mat & CvProcessor::getImage(const string & name) const
242     throw (CvProcessorException)
243 {
244     // Search for this name
245     map<string, Mat*>::const_iterator cit;
246     for (cit = images.begin(); cit ≠ images.end(); ++cit)
```



```
247     {
248         if (cit→first ≡ name)
249         {
250             if (cit→second→data ≡ NULL)
251             {
252                 // image contains no data
253                 throw CvProcessorException(CvProcessorException::NULL_DATA,
254                                            name.c_str());
255             }
256             return *(cit→second);
257         }
258     }
259 
260     // not found : throw exception
261     throw CvProcessorException(CvProcessorException::INVALID_NAME,
262                                name.c_str());
263 }
264 
265 /*
266  * Get image pointer by name
267  * @param name the name of the image we're looking for
268  * @return the image pointer registered by this name in the additionnal
269  * images map
270  * @throw CvProcessorException#INVALID_NAME is used name is not already
271  * registerd in the images
272  */
273 Mat * CvProcessor::getImagePtr(const char *name)
274     throw (CvProcessorException)
275 {
276     string sname(name);
277 
278     return getImagePtr(sname);
279 }
280 
281 /*
282  * Get image pointer by name
283  * @param name the name of the image we're looking for
284  * @return the image registered by this name in the additionnal images
285  * map
286  * @throw CvProcessorException#INVALID_NAME is used name is not already
287  * registerd in the images
288  */
289 Mat * CvProcessor::getImagePtr(const string & name)
290     throw (CvProcessorException)
291 {
292     // Search for this name
293     map<string, Mat*>::const_iterator cit;
294     for (cit = images.begin(); cit ≠ images.end(); ++cit)
295     {
296         if (cit→first ≡ name)
297         {
298             if (verboseLevel ≥ VERBOSE_ACTIVITY)
299             {
300                 clog << "getImagePtr(" << name << "): returning : "
301                      << (long) (cit→second) << endl;
302             }
303             return cit→second;
304         }
305     }
306 
307     // not found : throw exception
308     throw CvProcessorException(CvProcessorException::INVALID_NAME, name.c_str());
309 }
310 
311 /*
312  * Number of channels in source image
313  * @return the number of channels of source image
314  */
315 int CvProcessor::getNbChannels() const
316 {
317     return nbChannels;
318 }
319 
320 /*
321  * Type of the source image
322  * @return the openCV type of the source image
323  */
324 int CvProcessor::getType() const
325 {
326     return type;
327 }
328 
```

```
/*
 * Get the current verbose level
 * @return the current verbose level
 */
CvProcessor::VerboseLevel CvProcessor::getVerboseLevel() const
{
    return verboseLevel;
}

/*
 * Set new verbose level
 * @param level the new verobse level
 */
void CvProcessor::setVerboseLevel(const VerboseLevel level)
{
    if ((level ≥ VERBOSE_NONE) ∧ (level < NBVERBOSELEVEL))
    {
        verboseLevel = level;
    }

    cout << "Verbose level set to: ";
    switch (verboseLevel)
    {
        case VERBOSE_NONE:
            cout << "no messages";
            break;
        case VERBOSE_ERRORS:
            cout << "unrecoverable errors only";
            break;
        case VERBOSE_WARNINGS:
            cout << "errors and warnings";
            break;
        case VERBOSE_NOTIFICATIONS:
            cout << "errors, warnings and notifications";
            break;
        case VERBOSE_ACTIVITY:
            cout << "All messages";
            break;
        case NBVERBOSELEVEL:
        default:
            cout << "Unknown verobse mode (unchanged)";
            break;
    }
    cout << endl;
}

/*
 * Return processor processing time of step index [default implementation
 * returning only processTime, should be reimplemented in subclasses]
 * @param index index of the step which processing time is required,
 * 0 indicates all steps, and values above 0 indicates step #. If
 * required index is bigger than number of steps than all steps value
 * should be returned.
 * @return the processing time of step index.
 * @note should be reimplemented in subclasses in order to define
 * time/feature behaviour
 */
double CvProcessor::getProcessTime(const size_t) const
{
    return processTime;
}


/*
 * Indicates if processing time is per feature processed in the current
 * image or absolute
 * @return
 */
bool CvProcessor::isTimePerFeature() const
{
    return timePerFeature;
}

/*
 * Sets Time per feature processing time unit
 * @param value the time per feature value (true or false)
 */
void CvProcessor::setTimePerFeature(const bool value)
{
    timePerFeature = value;
}

```



```
411
```

```
#ifndef CVPROCESSOREXCEPTION_H_
#define CVPROCESSOREXCEPTION_H_

#include <iostream>     // for ostream
#include <string>       // for string
#include <exception>    // for std::exception base class
using namespace std;

/**
 * Exception class for CvProcessor.
 * Contains mainly exception reasons why an CvProcessor operation could not be
 * performed.
 */
class CvProcessorException : public exception
{
    public:
        /**
         * Matrices operation exception cases
         */
        typedef enum
        {
            /**
             * Null image.
             * Used when trying to add null image as source image of the
             * processor
             */
            NULL_IMAGE,
            /**
             * Null image data.
             * Used when trying to use image with NULL data
             */
            NULL_DATA,
            /**
             * Invalid name in image acces by name.
             * Used when searching for images by name which is not contained
             * in the already registered names
             */
            INVALID_NAME,
            /**
             * Invalid image type.
             * Some Processors needs specific images types
             */
            INVALID_IMAGE_TYPE,
            /**
             * Illegal data access (i.e. read/write access on read only data)
             */
            ILLEGAL_ACCESS,
            /**
             * Allocation failure on dynamically allocated elements
             */
            ALLOC_FAILURE,
            /**
             * Unable to read a file
             */
            FILE_READ_FAIL,
            /**
             * File parse error
             */
            FILE_PARSE_FAIL,
            /**
             * Unable to write file
             */
            FILE_WRITE_FAIL,
            /**
             * OpenCV exception
             */
            OPENCV_EXCEPTION
        } ExceptionCause;

        /**
         * CvProcessor exception constructor
         * @param e the chosen error case for this error
         * @see ExceptionCause
         */
        CvProcessorException(const CvProcessorException::ExceptionCause e);

        /**
         * CvProcessor exception constructor with exception message descriptor
         * @param e the chosen error case for this error
         * @param descr character string describing the message
         * @see ExceptionCause
         */
```



```
        CvProcessorException(const CvProcessorException::ExceptionCause e,
                             const char * descr);

        /**
         * CvProcessor exception from regular (typically OpenCV) exception
         * @param e the exception to relay
         */
        CvProcessorException(const exception & e, const char * descr = "");

        /**
         * CvProcessor exception destructor
         * @post message cleared
         */
        virtual ~CvProcessorException() throw ();

        /**
         * Explanation message of the exception
         * @return a C-style character string describing the general cause
         * of the current error.
         */
        virtual const char* what() const throw();

        /**
         * CvProcessorException cause
         * @return the cause enum of the exception
         */
        CvProcessorException::ExceptionCause getCause();

        /**
         * Source message of the exception
         * @return the message string of the exception
         */
        string getMessage();

        /*
         * Note output operators are not necessary since what() method is used
         * to explain the reason of the exception.
         * Example :
         * try
         * {
         *  ... do something which throws an std::exception
         * }
         * catch (exception & e)
         * {
         *  cerr << e.what() << endl;
         * }
         */

    protected:
        /**
         * The current error case
         */
        CvProcessorException::ExceptionCause cause;

        /**
         * description message of the exception
         */
        string message;
};

#endif /*CVPROCESSOREXCEPTION_H_*/
```

```cpp
#include "CvProcessorException.h"
#include <iostream>     // for cerr et endl;
#include <string>       // for string
#include <sstream>      // for ostringstream
using namespace std;

/*
 * CvProcessor exception constructor
 * @param e the chosen error case for this error
 * @see ExceptionCause
 */
CvProcessorException::CvProcessorException(
    const CvProcessorException::ExceptionCause e) :
    exception(),
    cause(e),
    message("")
{
}

/*
 * CvProcessor exception constructor with message descriptor
 * @param e the chosen error case for this error
 * @param descr character string describing the message
 * @see ExceptionCause
 */
CvProcessorException::CvProcessorException(
    const CvProcessorException::ExceptionCause e, const char * descr) :
    exception(),
    cause(e),
    message(descr)
{
}

/*
 * CvProcessor exception from regular (typically OpenCV) exception
 * @param e the exception to relay
 */
CvProcessorException::CvProcessorException(const exception & e, const char * descr) :
    exception(e),
    cause(OPENCV_EXCEPTION),
    message(descr)
{
}


/*
 * CvProcessor exception destructor
 * @post message cleared
 */
CvProcessorException::~CvProcessorException() throw ()
{
    message.clear();
}

/*
 * Explanation message of the exception
 * @return a C-style character string describing the general cause
 * of the current error.
 */
const char * CvProcessorException::what() const throw()
{
    const char * initialWhat = exception::what();

    ostringstream output;

    output << initialWhat << ":";

    output << "CvProcessorException : ";

    if (message.length() > 0)
    {
        output << message << ":";
    }

    switch (cause) {
        case CvProcessorException::NULL_IMAGE:
            output << "NULL image" << endl ;
            break;
        case CvProcessorException::NULL_DATA:
            output << "NULL image data" << endl ;
            break;
        case CvProcessorException::INVALID_NAME:
```



```cpp
            output << "Invalid name" << endl ;
            break;
        case CvProcessorException::INVALID_IMAGE_TYPE:
            output << "Invalid image type" << endl;
            break;
        case CvProcessorException::ILLEGAL_ACCESS:
            output << "Illegal access" << endl;
            break;
        case CvProcessorException::ALLOC_FAILURE:
            output << "New element allocation failure" << endl;
            break;
        case CvProcessorException::FILE_READ_FAIL:
            output << "Unable to read file" << endl;
            break;
        case CvProcessorException::FILE_PARSE_FAIL:
            output << "File parse error" << endl;
            break;
        case CvProcessorException::FILE_WRITE_FAIL:
            output << "Unable to write file" << endl;
            break;
        default:
            output << "Unknown exception" << endl;
            break;
    }

    return output.str().c_str();
}


/*
 * CvProcessorException cause
 * @return the cause enum of the exception
 */
CvProcessorException::ExceptionCause CvProcessorException::getCause()
{
    return cause;
}

/*
 * Source message of the exception
 * @return the message string of the exception
 */
string CvProcessorException::getMessage()
{
    return message;
}
```

```
/*
 * QcvProcessor.h
 *
 *  Created on: 19 fÃ©vr. 2012
 *      Author: davidroussel
 */

#ifndef QCVPROCESSOR_H_
#define QCVPROCESSOR_H_

#include <QObject>
#include <QString>
#include <QRegExp>
#include <QMutex>
#include <QThread>
#include "CvProcessor.h"

/**
 * Qt flavored class to process a source image with OpenCV 2+
 */
class QcvProcessor : public QObject, public virtual CvProcessor
{
    Q_OBJECT

    protected:

        /**
         * Default timeout to show messages
         */
        static int defaultTimeOut;

        /**
         * Number format used to format numbers into QStrings
         */
        static char numberFormat[10];

        /**
         * The regular expression used to validate new number formats
         * @see #setNumberFormat
         */
        static QRegExp numberRegExp;

        /**
         * The Source image mutex in order to avoid concurrent access to
         * the source image (typically the source image may be modified
         */
        QMutex * sourceLock;

        /**
         * the thread in which this processor should run
         */
        QThread * updateThread;

        /**
         * Message to send when something changes
         */
        QString message;

        /**
         * String used to store formatted process time value
         */
        QString processTimeString;

    public:

        /**
         * QcvProcessor constructor
         * @param image the source image
         * @param imageLock the mutex for concurrent access to the source image.
         * In order to avoid concurrent access to the same image
         * @param updateThread the thread in which this processor should run
         * @param parent parent QObject
         */
        QcvProcessor(Mat * image,
                     QMutex * imageLock = NULL,
                     QThread * updateThread = NULL,
                     QObject * parent = NULL);

        /**
         * QcvProcessor destructor
         */
        virtual ~QcvProcessor();
```



```

        /**
         * Sets new number format
         * @param format the new number format
         * @pre format string should look like "%8.1f" or at least not be longer
         * than 10 chars since format is a 10 chars array.
         * @post id format string is valid and shorter than 10 chars
         * it has been applied as the new format string.
         */
        static void setNumberFormat(const char * format);

    public slots:
        /**
         * Update computed images slot and sends updated signal
         */
        virtual void update();

        /**
         * Changes source image slot.
         * Attributes needs to be cleaned up then set up again
         * @param image the new source Image
         * @throw CvProcessorException#NULL_IMAGE when new source image is NULL
         * @post Various signals are emitted:
         *  - imageChanged(sourceImage)
         *  - imageCchanged()
         *  - if image size changed then imageSizeChanged() is emitted
         *  - if image color space changed then imageColorsChanged() is emitted
         */
        virtual void setSourceImage(Mat * image) throw (CvProcessorException);

        /**
         * Sets Time per feature processing time unit slot.
         * @param value the time per feature value (true or false)
         */
        virtual void setTimePerFeature(const bool value);

    signals:
        /**
         * Signal emitted when update is complete
         */
        void updated();

        /**
         * Signal emitted when processor has finished.
         * Used to tell helper threads to quit
         */
        void finished();

        /**
         * Signal emitted when source image is reallocated
         */
        void imageChanged();

        /**
         * Signal emitted when source image is reallocated
         * @param image the new source image pointer or none if just
         * image changed notification is required
         */
        void imageChanged(Mat * image);

        /**
         * Signal emitted when source image colors changes from color to gray
         * or from gray to color
         */
        void imageColorsChanged();

        /**
         * Signal emitted when source image size changes
         */
        void imageSizeChanged();

        /**
         * Signal emited when processing time has channged
         * @param value the new value of the processing time
         */
        void processTimeUpdated(const QString & formattedValue);

        /**
         * Signal to set text somewhere
         * @param message the message
         */
        void sendText(const QString & message);
```

```
165
166         /**
167          * Signal to send update message when something changes
168          * @param message the message
169          * @param timeout number of ms the message should be displayed
170          */
171         void sendMessage(const QString & message, int timeout = defaultTimeOut);
172
173 };
174
175 #endif /* QCVPROCESSOR_H_ */
```



```
1   /*
2    * QcvProcessor.cpp
3    *
4    *  Created on: 19 fÃ©vr. 2012
5    *      Author: davidroussel
6    */
7
8   #include <QRegExpValidator>
9   #include <QDebug>
10  #include <cstring>      // for strcpy
11  #include "QcvProcessor.h"
12
13  /*
14   * Default timeout to show messages
15   */
16  int QcvProcessor::defaultTimeOut = 5000;
17
18  /*
19   * Number format used to format numbers into QStrings
20   */
21  char QcvProcessor::numberFormat[10] = {"%8.1f ms"};
22
23  /*
24   * The regular expression used to validate new number formats
25   * @see #setNumberFormat
26   */
27  QRegExp QcvProcessor::numberRegExp("%[+- 0#]*[0-9]*([.][0-9]+)?[efEF]");
28
29  /*
30   * QcvProcessor constructor
31   * @param image the source image
32   * @param imageLock the mutex for concurrent access to the source image
33   * In order to avoid concurrent access to the same image
34   * @param updateThread the thread in which this processor should run
35   * @param parent parent QObject
36   */
37  QcvProcessor::QcvProcessor(Mat * image,
38                             QMutex * imageLock,
39                             QThread * updateThread,
40                             QObject * parent) :
41      CvProcessor(image), // <-- virtual base class constructor first
42      QObject(parent),
43      sourceLock(imageLock),
44      updateThread(updateThread),
45      message(),
46      processTimeString()
47  {
48      if (updateThread != NULL)
49      {
50          this->moveToThread(updateThread);
51
52          connect(this, SIGNAL(finished()), updateThread, SLOT(quit()),
53                  Qt::DirectConnection);
54
55          updateThread->start();
56      }
57  }
58
59  /*
60   * QcvProcessor destructor
61   */
62  QcvProcessor::~QcvProcessor()
63  {
64      // Lock might be already destroyed in source object so don't try to unlock
65
66      message.clear();
67      processTimeString.clear();
68
69      emit finished();
70
71      if (updateThread != NULL)
72      {
73          // Wait until update thread has received the "finished" signal through
74          // "quit" slot
75          updateThread->wait();
76      }
77  }
78
79  /*
80   * Sets new number format
81   * @param format the new number format
82   */
```

```
83  void QcvProcessor::setNumberFormat(const char * format)
84  {
85      /*
86       * The format string should validate the following regex
87       * %[+- 0#]*[0-9]*([.][0-9]+)?[efEF]
88       */
89      QRegExpValidator validator(numberRegExp, NULL);
90
91      QString qFormat(format);
92      int pos = 0;
93      if ((validator.validate(qFormat,pos) ≡ QValidator::Acceptable) ∧
94          (strlen(format) ≤ 10))
95      {
96          strcpy(numberFormat, format);
97      }
98      else
99      {
100         qWarning("QcvProcessor::setNumberFormat(%s) : invalid format", format);
101     }
102 }
103
104
105 /*
106  * Update computed images slot and sends updated signal
107  * required
108  */
109 void QcvProcessor::update()
110 {
111     /*
112      * Important note : CvProcessor::update() should NOT be called here
113      * since it should be called in QcvXXXprocessor subclasses such that
114      * QcvXXXProcessor::update method should contain :
115      *  - call to CvXXXProcessor::update() (not QCvXXXProcessor)
116      *  - emit signals from QcvXXXProcessor
117      *  - call to QcvProcessor::update() (this method)
118      */
119     emit updated();
120     processTimeString.sprintf(numberFormat, getProcessTime(0) / 1000.0);
121     emit processTimeUpdated(processTimeString);
122 }
123
124 /*
125  * Changes source image slot.
126  * Attributes needs to be cleaned up then set up again
127  * @param image the new source Image
128  * @post Various signals are emitted:
129  *  - imageChanged(sourceImage)
130  *  - imageCchanged()
131  *  - if image size changed then imageSizeChanged() is emitted
132  *  - if image color space changed then imageColorsChanged() is emitted
133  */
134 void QcvProcessor::setSourceImage(Mat *image)
135     throw (CvProcessorException)
136 {
137     if (verboseLevel ≥ VERBOSE_NOTIFICATIONS)
138     {
139         clog << "QcvProcessor::setSourceImage(" << (ulong) image << ")" << endl;
140     }
141
142     Size previousSize(sourceImage→size());
143     int previousNbChannels(nbChannels);
144
145     if (sourceLock ≠ NULL)
146     {
147         sourceLock→lock();
148         // qDebug() << "QcvProcessor::setSourceImage: lock";
149     }
150
151     CvProcessor::setSourceImage(image);
152
153     if (sourceLock ≠ NULL)
154     {
155         // qDebug() << "QcvProcessor::setSourceImage: unlock";
156         sourceLock→unlock();
157     }
158
159     emit imageChanged(sourceImage);
160
161     emit imageChanged();
162
163     if ((previousSize.width ≠ image→cols) ∨
164         (previousSize.height ≠ image→rows))
```



```
165     {
166         emit imageSizeChanged();
167     }
168
169     if (previousNbChannels ≠ nbChannels)
170     {
171         emit imageColorsChanged();
172     }
173
174     // Force update
175     update();
176 }
177
178 /*
179  * Sets Time per feature processing time unit slot
180  * @param value the time per feature value (true or false)
181  */
182 void QcvProcessor::setTimePerFeature(const bool value)
183 {
184     CvProcessor::setTimePerFeature(value);
185 }
```

```
/*
 * CvHistograms.h
 *
 *  Created on: 22 fÃ©vr. 2012
 *      Author: David Roussel
 */

#ifndef CVHISTOGRAMS_H_
#define CVHISTOGRAMS_H_

#include <cv.h> // For Mat and Scalar
using namespace cv;

#include <vector>
using namespace std;

#include "CvProcessor.h"

/*
 * Forward declaration of Histograms output operator
 */
template <typename T, size_t channels> class CvHistograms;
template <typename T, size_t channels>
ostream & operator << (ostream & out, const CvHistograms<T, channels> & h);

/**
 * OpenCV Multiple histograms of an image.
 * @param T the data type in the image. Usually, unsigned char (default is uchar)
 * @param channels the number of channels in the image (default is 1)
 * If image has only one channel, no other histogram are computed.
 * But if image has several channels, each layer has an histogram and an
 * additional histogram corresponding to gray level equivalent image is
 * computed by linear combination of the previously computed histograms.
 * Eventually, linear combination coefficients are used :
 *  - for RGB images linear combination coefficients are
 *      - C_red = 0.30
 *      - C_green = 0.59
 *      - C_blue = 0.11
 *  - for YUV images linear combination coefficients are not necessary since
 *  the V component is already a grayscale component
 */
template <typename T = uchar, size_t channels = 1>
class CvHistograms : virtual public CvProcessor
{
    public:
        /**
         * Color Histogram indices
         */
        typedef enum
        {
            HIST_BLUE = 0,      //!< HIST_BLUE
            HIST_GREEN = 1,     //!< HIST_GREEN
            HIST_RED = 2,       //!< HIST_RED
            HIST_GRAY = 3       //!< HIST_GRAY
        } ColorHistIndex;

        /**
         * Transfert function to apply on the image.
         * Transfert function (also called LUT : standing for Look Up Table)
         * are applied on the image with OpenCV function :
         * @code
         * void LUT(const Mat& src, const Mat& lut, Mat& dst)
         * @endcode
         * with
         *  - src - Source array of 8-bit elements
         *  - lut - Look-up table of 256 elements. In the case of multi-channel
         *  source array, the table should either have a single channel
         *  (in this case the same table is used for all channels) or the same
         *  number of channels as in the source array
         *  - dst - Destination array; will have the same size and the same number
         *  of channels as src , and the same depth as lut
         */
        typedef enum
        {
            /**
             * No transfert function should be applied on the image
             */
            NONE=0,
            /**
             * Image threshold on all channels should be applied on the image
             * @see CvHistograms<T,channels>::computeGrayThresholdLUT
             */
```



```
            THRESHOLD_GRAY,
            /**
             * Optimal image dynamic should be applied on the image
             * @see CvHistograms<T,channels>::computeGrayOptimalLUT
             */
            DYNAMIC_GRAY,
            /**
             * Levels equalization should be applied on the images
             * @see CvHistograms<T,channels>::computeGrayEqualizeLUT
             */
            EQUALIZE_GRAY,
            /**
             * Image threshold with different threshold on each channel should be
             * applied on the image
             * @see CvHistograms<T,channels>::computeColorThresholdLUT
             */
            THRESHOLD_COLOR,
            /**
             * Optimal image dynamic should be applied on the image using
             * different dynamic on each channel
             * @see CvHistograms<T,channels>::computeColorOptimalLUT
             */
            DYNAMIC_COLOR,
            /**
             * Levels equalization should be applied on the images using different
             * equalization on each channel
             * @see CvHistograms<T,channels>::computeColorEqualizeLUT
             */
            EQUALIZE_COLOR,
            /**
             * Gamma transfert function
             * @see CvHistograms<T,channels>::computeGammaLUT
             */
            GAMMA,
            /**
             * Negative transfert function
             * @see CvHistograms<T,channels>::computeNegativeLUT
             */
            NEGATIVE,
            /**
             * Defines the number of available transfert functions.
             * Used to toggle between LUTs to apply by using
             * @code currentTransfertFunc % NBTRANS @endcode
             */
            NBTRANS
        } TransfertType;

        /**
         * Processing indices for getProcessTime method
         * @see #getProcessTime
         */
        typedef enum
        {
            ALL = 0,                    //!< ALL
            UPDATE_HISTOGRAM,           //!< UPDATE_HISTOGRAM
            COMPUTE_LUT,                //!< COMPUTE_LUT
            DRAW_LUT,                   //!< DRAW_LUT
            APPLY_LUT,                  //!< APPLY_LUT
            UPDATE_HISTOGRAM_AFTER_LUT, //!< UPDATE_HISTOGRAM_AFTER_LUT
            DRAW_HISTOGRAM,             //!< DRAW_HISTOGRAM
            NB_PROC_INDEX               //!< Number of processing time indices
        } ProcessTimeIndex;

    protected:
        // ----------------------------------------------------------------------
        // Histograms attributes
        // ----------------------------------------------------------------------
        /**
         * 3 coefficients for additionnal grayscale histogram from RGB image :
         *  - \f$Coef_{red} = 0.30\f$
         *  - \f$Coef_{green} = 0.59\f$
         *  - \f$Coef_{blue} = 0.11\f$
         * @note Be aware that OpenCV Color images are ususally encoded in BGR
         * format instead of RGB.
         */
        static const float BGR2Gray[];

        /**
         * Number of bins in the histogram.
         * All histogram populations ranges from 0 to bins-1
         */
        static const size_t bins;
```

```
165
166         /**
167          * Checks whether to compute additionnal gray level histogram
168          * from RGB components.
169          * @note has no impact if number of channels in the image is not 3
170          */
171         bool computeGray;
172
173         /**
174          * Number of computed histograms.
175          * @note could be bigger than the number of channels in the image
176          * if an additional gray level histogram is computed.
177          */
178         size_t nbHistograms;
179
180         /**
181          * The histogram values (an array containing "bins" elements).
182          * if image has 3 channels (BGR), a fourth histogram is computed
183          * according to the computeGray attribute in order to compute the
184          * equivalent gray level histogram.
185          * @see #BGR2Gray
186          */
187         vector<float *> histograms;
188
189         /**
190          * Maximum value found in all histograms
191          */
192         float maxValue;
193
194         /**
195          * The cumulative histogram computed by cumulatively sum "hist".
196          * (an array containing "bins" elements)
197          */
198         vector<float *> cumulHistograms;
199
200         /**
201          * Maximum value found in all cumulative histogram.
202          * @note cumulative maximum should be the number of pixels
203          * in the image but when histogram is time cumulative it is
204          * a multiple of number of pixels.
205          */
206         float cMaxValue;
207
208         /**
209          * checks whether histograms are time cumulative or not.
210          * if time cumulative histogram value are not cleared before
211          * updating the histogram values.
212          */
213         bool timeCumulative;
214
215         // ------------------------------------------------------------------
216         // LUT attributes
217         // ------------------------------------------------------------------
218         /**
219          * Gray level transfert function
220          */
221         Mat monoTransfertFunc;
222
223         /**
224          * Colors transfert functions
225          */
226         Mat colorTransferFunc;
227
228         /**
229          * Current LUT to apply.
230          * Alternatively receives monoTransfertFunc or colorTransferFunc address
231          * depending on the transfert function to apply
232          */
233         Mat * lut;
234
235         /**
236          * Current LUT type
237          */
238         TransfertType lutType;
239
240         /**
241          * Previous LUT type. Used to avoid recomputing LUTs that does not
242          * depend on image histogram such as NONE, GAMMA and NEGATIVE
243          */
244         TransfertType previousLutType;
245
246         /**
```



```
247          * Current percentage for LUTs that requires such a parameter
248          */
249         float lutParam;
250
251         /**
252          * previous percentage for LUTs that requires such a parameter.
253          * Needed to know if LUT not depending on image histogram should be
254          * refreshed when param changes, such as Gamma
255          */
256         float previousLutParam;
257
258         /**
259          * Maximum percentage for LUTs that requires such a parameter
260          */
261         static const float maxParam;
262
263         /**
264          * Minimum percentage for LUTs that requires such a parameter
265          */
266         static const float minParam;
267
268         /**
269          * Indicates if LUT has been updated
270          */
271         bool lutUpdated;
272
273         // ------------------------------------------------------------------
274         // Drawing attributes
275         // ------------------------------------------------------------------
276         /**
277          * checks whether to show cumulative histograms in the drawing or
278          * regular histograms
279          */
280         bool showCumulative;
281
282         /**
283          * components to show in the drawing
284          */
285         vector<bool> showComponent;
286
287         /**
288          * image width of the histogram drawing frame
289          */
290         size_t histWidth;
291
292         /**
293          * image height of the histogram drawing frame
294          */
295         size_t histHeight;
296
297         /**
298          * image width of the LUT drawing frame
299          */
300         size_t lutWidth;
301
302         /**
303          * image height of the LUT drawing frame
304          */
305         size_t lutHeight;
306
307         /**
308          * drawing color for the histograms
309          */
310         vector<Scalar> displayColors;
311
312         /**
313          * The color Matrices to draw each histogram
314          */
315         vector<Mat> histComponents;
316
317         /**
318          * The Frame to draw all histograms in
319          */
320         Mat histDisplayFrame;
321
322         /**
323          * The color Matrices to draw each LUT
324          */
325         vector<Mat> lutComponents;
326
327         /**
328          * The Frame to draw all LUTs in
```

```
329             */
330         Mat lutDisplayFrame;
331 
332         /**
333          * The frame to draw transformed image when LUT is applied
334          */
335         Mat outDisplayFrame;
336 
337         // ------------------------------------------------------------------
338         // Time measurement attributes
339         // ------------------------------------------------------------------
340 
341         /**
342          * Update histogram time when new frames appends
343          */
344         clock_t updateHistogramTime1;
345 
346         /**
347          * LUT computing time
348          */
349         clock_t computeLUTTime;
350 
351         /**
352          * LUT drawing time
353          */
354         clock_t drawLUTTime;
355 
356         /**
357          * LUT apply time on image
358          */
359         clock_t applyLUTTime;
360 
361         /**
362          * Update histogram time after LUT is applied (when needed)
363          */
364         clock_t updateHistogramTime2;
365 
366         /**
367          * Histogram drawing time
368          */
369         clock_t drawHistogramTime;
370 
371     public:
372 
373         /**
374          * Histogram constructor
375          * @param image the image to use for computing histograms
376          * @param drawColors the drawing colors of the histogram
377          * @param computeGray checks whether to compute 4th gray level
378          * histogram on BGR image or not
379          * @param drawHeight the drawing height of the histogram window
380          * @param drawWidth the drawing width of the histogram window
381          * @param timeCumulation checks whether to compute time cumulative
382          * histograms or not.
383          */
384         CvHistograms(Mat * image,
385                      const bool computeGray = true,
386                      const size_t drawHeight = 256,
387                      const size_t drawWidth = 512,
388                      const bool timeCumulation = false);
389 
390         /**
391          * Histogram destructor.
392          * clears histogram values and release display frame
393          */
394         virtual ~CvHistograms();
395 
396         /**
397          * Update histogram, LUT and resulting images
398          */
399         virtual void update(void);
400 
401         /**
402          * Update histograms values.
403          */
404         virtual void updateHistogram(void);
405 
406         /**
407          * Value reading access operator
408          * @param i the ith histogram to access. if i is invalid, 0 is returned
409          * @param j the jth bin value of the ith histogram to access. if j is
410          * invalid, 0 is returned.
```



```
411          * @param cumulative checks whether to return regular histogram value
412          * or cumulative histogram value
413          * @return the value in the jth bin of the ith histogram
414          * @par usage :
415          * @code
416          *  float jthValue = myHist(i,j);
417          *  float jthCumulativeValue = myHist(i,j,true);
418          * @endcode
419          */
420         float operator()(size_t i,
421                          size_t j,
422                          bool cumulative = false) const;
423 
424         /**
425          * Value reading/writing access operator
426          * @param i the ith histogram to access. if i is invalid, 0 is returned
427          * @param j the jth bin value of the ith histogram to access. if j is
428          * invalid, 0 is returned.
429          * @param cumulative checks whether to return regular histogram value
430          * or cumulative histogram value
431          * @return the value in the jth bin of the ith histogram
432          * @par usage :
433          * @code
434          *  float myHist(i,j) = jthValue;
435          *  float myHist(i,j,true) = jthCumulativeValue;
436          * @endcode
437          */
438         float & operator()(size_t i,
439                            size_t j,
440                            bool cumulative = false);
441 
442 //       /**
443 //        * Gets Histogram display frame
444 //        * @return the image histogram is drawn in
445 //        */
446 //       const Mat & getHistogramImage(void) const;
447 //
448 //       /**
449 //        * Gets Histogram display frame pointer
450 //        * @return the image histogram is drawn in
451 //        */
452 //       Mat * getHistogramImagePtr(void);
453 
454 
455 //       /**
456 //        * Gets Transfert Func display frame
457 //        * @return the image transfert func is drawn in
458 //        */
459 //       const Mat & getTransfertFuncImage(void) const;
460 //
461 //       /**
462 //        * Gets Transfert Func display frame pointer
463 //        * @return the image transfert func is drawn in
464 //        */
465 //       Mat * getTransfertFuncImagePtr(void);
466 
467         /**
468          * Number of bins in all histograms
469          * @return the Number of bins in all histograms
470          */
471         static size_t getBins();
472 
473         /**
474          * Get the number of histograms computed
475          * @return the current number of histograms computed by this class
476          */
477         size_t getNbHistograms() const;
478 
479         /**
480          * Gets the additionnal gray histogram status
481          * @return true if additional gray level histogram is computed,
482          * false otherwise
483          */
484         bool isComputeGray() const;
485 
486         /**
487          * Maximum histograms value;
488          * @return the maximum value in all histograms
489          */
490         float getMaxValue() const;
491 
492         /**
```

```
493             * Maximum cumulative histograms value;
494             * @return the maximum value in all histograms
495             * @note regular cumulative maximum value is the number of pixels in
496             * the image, but when timecumulative is activated it can be bigger.
497             */
498         float getCMaxValue() const;
499
500 //      /**
501 //       * Histograms values accessor
502 //       * @return the histograms values
503 //       * @see #histograms
504 //       */
505 //      const vector<float *> & getHistogramValues() const;
506
507 //      /**
508 //       * Cumulative histograms values accessor
509 //       * @return the cumulative histograms values
510 //       * @see #cumulHistograms
511 //       */
512 //      const vector<float *> & getCumulativeHistogramValues() const;
513
514 //      /**
515 //       * Mono Channel (gray) transfert function accessor
516 //       * @return reference to the monoTransfertFunc
517 //       * @see #monoTransfertFunc
518 //       */
519 //      const Mat & getMonoTransfertFunc() const;
520
521 //      /**
522 //       * Color (3 channels) transfert function accessor
523 //       * @return reference to the color transfert function
524 //       * @see #colorTransfertFunc
525 //       */
526 //      const Mat & getColorTransfertFunc() const;
527
528         /**
529             * Time cumulative histogram status read access
530             * @return the time cumulative histogram status
531             */
532         bool isTimeCumulative() const;
533
534         /**
535             * Time cumulative histogram status read access
536             * @param value the value to set for time cumulative status
537             */
538         virtual void setTimeCumulative(bool value);
539
540         /**
541             * Cumulative histogram status read access
542             * @return the cumulative histogram status
543             */
544         bool isCumulative() const;
545
546         /**
547             * Cumulative histogram status read access
548             * @param value the value to set for cumulative status
549             */
550         virtual void setCumulative(bool value);
551
552         /**
553             * Ith histogram component shown status read access
554             * @param i the ith histogram component
555             * @return true if this component show status is true
556             */
557         bool isShowComponent(const size_t i) const;
558
559         /**
560             * Ith histogram component shown status write access
561             * @param i the ith histogram component
562             * @param value the value to set for this component show status
563             */
564         virtual void setShowComponent(const size_t i,
565                                       const bool value);
566
567         /**
568             * Indicates if LUT has been updated or if it has not changed
569             * @return true if LUT has been updated
570             */
571         bool isLUTUpdated() const;
572
573         /**
574             * Gets the current LUT type
```



```
575             * @return the current LUT type
576             */
577         TransfertType getLutType() const;
578
579         /**
580             * Sets the current LUT type
581             * @param lutType the new LUT type
582             */
583         virtual void setLutType(const TransfertType lutType);
584
585         /**
586             * Gets the current parameter value for LUTs using a percentage parameter
587             * @return the current LUT parameter
588             */
589         float getLUTParam() const;
590
591         /**
592             * Sets the current LUT % parameter
593             * @param lutParam the new LUT parameter
594             */
595         virtual void setLUTParam(float currentParam);
596
597 //      /**
598 //       * Gets the transformed image after drawTransformedImage
599 //       * @return the out display frame
600 //       */
601 //      const Mat & getTransformedImage() const;
602 //
603 //      /**
604 //       * Gets the transformed image pointer after drawTransformedImage
605 //       * @return the out display frame
606 //       */
607 //      Mat * getTransformedImagePtr();
608
609         /**
610             * Return processor processing time of step index [default implementation
611             * returning only processTime, should be reimplemented in subclasses]
612             * @param index index of the step which processing time is required,
613             * 0 indicates all steps, and values above 0 indicates step #. If
614             * required index is bigger than number of steps than all steps value
615             * should be returned.
616             * @return the processing time of step index.
617             * @note should be reimplemented in subclasses in order to define
618             * time/feature behaviour
619             */
620         virtual double getProcessTime(const size_t index) const;
621
622         /**
623             * output operator for Histograms
624             * @param out the output stream
625             * @param h the histograms to print on the stream
626             * @return a reference to the output stream so it can be cumulated
627             */
628         friend ostream & operator <<<> (ostream & out,
629                                         const CvHistograms<T,channels> & h);
630     protected:
631         /**
632             * Setup attributes when source image is changed
633             * @param image source Image
634             * @param completeSetup
635             * @param computeGray checks if additionnal gray level histogram should
636             * be computed
637             * @param drawHeight histogram draw height
638             * @param drawWidth histogram draw width
639             * @param timeCumulation cheks time cumulation status
640             */
641         virtual void setup(Mat * image,
642                            const bool completeSetup = false);
643         /**
644             * Cleanup attributes before changing source image or cleaning class
645             * before destruction
646             */
647         virtual void cleanup();
648
649         /**
650             * Draws selected histogram(s) in drawing frame and returns the drawing
651             * frame
652             * @return the updated drawing frame.
653             * @post depending on several attributes one or several histograms
654             * have been drawn in the drawing frame wich is returned
655             *   - if #showCumulative is true then cumulative histograms are drawn
656             *   otherwise regular histograms are drawn
```

```
657          *  - each histogram is drawn only if its showComponent[i] is true.
658          */
659         virtual void drawHistograms(void);
660
661         /**
662          * Draws selected transfert function in drawing frame and returns the
663          * drawing frame
664          * @param lut the LUT to draw : the LUT may contains 1 or several
665          * channels
666          * @return the updated drawing frame
667          */
668         virtual void drawTransfertFunc(const Mat * lut);
669
670         /**
671          * Compute linear transfert function (LUT) : no change in image levels
672          * @return the LUT containing the corresponding transfert function,
673          * the returned matrix contains only one channel corresponding to
674          * the graylevel LUT which should be applied to all color channels of
675          * the image
676          * @post the result is stored in monoTransfertFunc
677          * @note It's useless to compute a color Linear LUT since all channels
678          * would contain the exact same values.
679          */
680         Mat * computeLinearGrayLUT(void);
681
682         /**
683          * Compute linear transfert function (LUT) : no change in image levels
684          * @return the LUT containing the corresponding transfert function,
685          * the returned matrix contains 3 channels corresponding to
686          * the color LUT which should be applied to all color channels of
687          * the image
688          * @post the result is stored in colorTransfertFunc
689          * @note It's useless to compute a color Linear LUT since all channels
690          * would contain the exact same values.
691          */
692         Mat * computeLinearColorLUT(void);
693
694         /**
695          * Compute the optimal dynamic LUT for preserving "percentDynamic"
696          * percent of the whole image ligthness range.
697          * @param percentDynamic the gray level percentage to spread on the
698          * whole (100%) gray level range in the image
699          * @return the LUT containing the corresponding transfert function,
700          * the returned matrix contains only one channel corresponding to
701          * the graylevel LUT which should be applied to all color channels of
702          * the image
703          * @post the result is stored in monoTransfertFunc
704          */
705         Mat * computeGrayOptimalLUT(unsigned int percentDynamic);
706
707         /**
708          * Compute the optimal dynamic LUTs (one for each channel) for preserving
709          * "percentDynamic" percent of the whole image color ranges.
710          * @param percentDynamic the colors level percentage to spread on the
711          * whole (100%) colors level range in the image
712          * @return the LUT containing the corresponding transfert functions,
713          * the returned matrix contains as much channels as the image and
714          * corresponding to the color level LUT which should be applied to
715          * each color channels of the image
716          * @post the result is stored in colorTransfertFunc
717          */
718         Mat * computeColorOptimalLUT(unsigned int percentDynamic);
719
720         /**
721          * Computes the transfert function corresponding to gray level
722          * equalization
723          * @return the matrix containing the gray level equalization LUT to
724          * apply on the image
725          * @post the result is stored in monoTransfertFunc
726          */
727         Mat * computeGrayEqualizeLUT(void);
728
729         /**
730          * Computes the transfert functions corresponding to each channel
731          * level equalization
732          * @return the matrix contaning each channel level equalization LUT to
733          * apply on the image
734          * @post the result is stored in colorTransferFunc
735          */
736         Mat * computeColorEqualizeLUT(void);
737
738         /**
```



```
739          * Compute the LUT corresponding to thresholded image with tPercent
740          * of the pixel population on each side of the threshold according
741          * to the cumulative gray level histogram
742          * @param tPercent percent of the population on each side of the
743          * threshold
744          * @return the LUT containing the corresponding transfert function,
745          * the returned matrix contains only one channel corresponding to
746          * the graylevel LUT which should be applied to all color channels of
747          * the image
748          * @post the result is stored in monoTransfertFunc
749          */
750         Mat * computeGrayThresholdLUT(float tPercent);
751
752         /**
753          * Compute the LUT corresponding to thresholded image with tPercent
754          * of the pixel components population on each side of the
755          * thresholds according to the cumulative color histograms
756          * @param tPercent percent of the population on each side of the
757          * thresholds
758          * @return the matrix containing each channel level equalization LUT to
759          * apply on the image
760          * @post the result is stored in colorTransferFunc
761          */
762         Mat * computeColorThresholdLUT(float tPercent);
763
764         /**
765          * Compute gamma LUT.
766          * \f$y(k) = x(k)^{\gamma}\f$
767          * @param tPercent
768          * @return the matrix containing the gamma LUT (mono)
769          */
770         Mat * computeGammaLUT(float tPercent);
771
772         /**
773          * Compute the LUT corresponding to negative image
774          * @return the matrix containing the negative LUT (mono)
775          */
776         Mat * computeNegativeLUT(void);
777
778         /**
779          * Compute and returns the current transfert function to be applied
780          * on the image, eventually with the current LUT parameter
781          * @return the mono or color LUT matrix to apply on the image depending
782          * on the lutType
783          * @see TransfertType
784          */
785         Mat * computeLUT();
786
787         /**
788          * Apply current LUT (if != NULL) to the source image to produce the
789          * outFrame
790          * @return true if LUT has been applied, false if lut is NULL or
791          * lutType is NONE
792          */
793         virtual bool drawTransformedImage(void);
794 };
795
796 #endif /* CVHISTOGRAMS_H_ */
```

```cpp
/*
 * CvHistograms.cpp
 *
 *  Created on: 22 fÃ©vr. 2012
 *      Author: David Roussel
 */
#include <cmath>        // for powf function
#include <iostream>     // for input / output streams
#include <limits>       // for numeric limits (max value of type T)
using namespace std;

#include "CvHistograms.h"

/*
 * Number of bins in the histogram.
 * All histogram populations ranges from 0 to bins-1
 */
template<typename T, size_t channels>
const size_t CvHistograms<T,channels>::bins = (size_t)powf(2,sizeof(T)*8);

/*
 * 3 coefficients for additionnal grayscale histogram from RGB image :
 *  - \f$Coef_{red} = 0.30\f$
 *  - \f$Coef_{green} = 0.59\f$
 *  - \f$Coef_{blue} = 0.11\f$
 * @note Be aware that OpenCV Color images are ususally encoded in BGR
 * format instead of RGB.
 */
template<typename T, size_t channels>
const float CvHistograms<T,channels>::BGR2Gray[] = {0.11, 0.59, 0.30};

/*
 * Maximum percentage for LUTs that requires such a parameter
 */
template<typename T, size_t channels>
const float CvHistograms<T,channels>::maxParam = 100.0;

/*
 * Minimum percentage for LUTs that requires such a parameter
 */
template<typename T, size_t channels>
const float CvHistograms<T,channels>::minParam = 0.0;


/*
 * Histogram constructor
 * @param image the image to use for computing histograms
 * @param drawColors the drawing colors of the histogram
 * @param computeGray checks whether to compute 4th gray level
 * histogram on BGR image or not
 * @param drawHeight the drawing height of the histogram window
 * @param drawWidth the drawing width of the histogram window
 * @param timeCumulation checks whether to compute time cumulative
 * histograms or not.
 */
template<typename T, size_t channels>
CvHistograms<T,channels>::CvHistograms(Mat * image,
                                       const bool computeGray,
                                       const size_t drawHeight,
                                       const size_t drawWidth,
                                       const bool timeCumulation) :
    CvProcessor(image),
    computeGray(computeGray),
    timeCumulative(timeCumulation),
    monoTransfertFunc(1,bins,CV_8UC1),
    colorTransferFunc(1,bins,CV_8UC(channels)),
    lut(NULL),
    lutType(NONE),
    previousLutType(NBTRANS),
    lutParam(80.0),
    previousLutParam(80.0),
    showCumulative(false),
    histWidth(drawWidth),
    histHeight(drawHeight),
    lutWidth(bins),
    lutHeight(bins),
    histDisplayFrame(drawHeight, drawWidth, CV_8UC(channels)),
    lutDisplayFrame(bins, bins, CV_8UC(channels)),
    outDisplayFrame(image→size(), image→type())

{
    // Partial setup since lots has been done in initialisation list above
```



```cpp
    setup(image, false);

    addImage("histogram", &histDisplayFrame);
    addImage("lut", &lutDisplayFrame);
    addImage("out", &outDisplayFrame);
}

/*
 * Setup attributes when source image is changed
 * @param image source Image
 * @param computeGray checks if additionnal gray level histogram should
 * be computed
 * @param drawHeight histogram draw height
 * @param drawWidth histogram draw width
 * @param timeCumulation cheks time cumulation status
 */
template<typename T, size_t channels>
void CvHistograms<T,channels>::setup(Mat * image,
                                     const bool completeSetup)
{
    CvProcessor::setup(image, completeSetup);

    // Complete setup starting point (==> previous cleanup)
    if (completeSetup)
    {
        monoTransfertFunc = Mat(1,bins,CV_8UC1);
        colorTransferFunc = Mat(1,bins,CV_8UC(channels));
        lut = NULL;
        lutType = NONE;
        previousLutType = NBTRANS;
        lutParam = 80.0;
        showCumulative = false;
        lutWidth = bins;
        lutHeight = bins;
        histDisplayFrame = Mat(histHeight, histWidth, CV_8UC(channels));
        lutDisplayFrame = Mat(bins, bins, CV_8UC(channels));
        outDisplayFrame = Mat(image→size(), image→type());
    }
    else //
    {
        // Creates colors to draw histogram components
        displayColors.push_back(Scalar(0xFF,0x00,0x00)); // Blue
        displayColors.push_back(Scalar(0x00,0xFF,0x00)); // Green
        displayColors.push_back(Scalar(0x00,0x00,0xFF)); // Red
        displayColors.push_back(Scalar(0xCC,0xCC,0xCC)); // Gray
    }

    // Partial setup starting point (==> no previous cleanup but contructor)


    if (sourceImage→data ≠ NULL)
    {
        maxValue = 0.0;
        cMaxValue = 0.0;

        nbHistograms = channels;
        if (this→computeGray ∧ (nbHistograms ≡ 3))
        {
            nbHistograms++;
        }

        for (size_t i=0; i < nbHistograms; i++)
        {
            // creates ith histogram
            histograms.push_back(new float[bins]);
            // creates ith cumulative histogram
            cumulHistograms.push_back(new float[bins]);
            // defines if ith component should be drawn
            showComponent.push_back(true);
            // creates ith drawing color histogram frame
            histComponents.push_back(Mat(histHeight, histWidth, CV_8UC3));
            lutComponents.push_back(Mat(lutHeight, lutWidth, CV_8UC3));

            /*
             * Initialize Histogram and cumiulative histograms values to 0.0
             * Avoid calling [] on vectors multiple times by using local
             * variables to store vector content (in this case float arrays)
             */
            float * h = histograms[i];
            float * ch = cumulHistograms[i];
            // initialize histograms values
            for (size_t j=0; j < bins; j++)
```

```
            {
                h[j] = 0.0;
                ch[j] = 0.0;
            }
        }

        if (this→computeGray ∧ (nbHistograms ≡ 4))
        {
            showComponent[HIST_GRAY] = false; // don't show gray hist. yet
        }
    }
    else // sourceImage->data is NULL
    {
        cerr << "CvHistograms::Setup : NULL source image" << endl;
        exit(EXIT_FAILURE);
    }
}

/*
 * Histogram destructor.
 * clears histogram values and release display frame
 */
template<typename T, size_t channels>
CvHistograms<T,channels>::~CvHistograms()
{
    cleanup();
}

/*
 * Cleanup attributes before changing source image or cleaning class
 * before destruction
 */
template<typename T, size_t channels>
void CvHistograms<T,channels>::cleanup()
{
    for (size_t i=0; i < histograms.size(); i++)
    {
        delete(histograms[i]);
        delete(cumulHistograms[i]);
        histComponents[i].release();
        lutComponents[i].release();
    }

    outDisplayFrame.release();
    lutDisplayFrame.release();
    lutComponents.clear();
    histDisplayFrame.release();
    histComponents.clear();
    displayColors.clear();
    showComponent.clear();
    colorTransferFunc.release();
    monoTransfertFunc.release();
    cumulHistograms.clear();
    histograms.clear();

    // Super cleanup
    CvProcessor::cleanup();
}

/*
 * Number of bins in all histograms
 * @return the Number of bins in all histograms
 */
template<typename T, size_t channels>
size_t CvHistograms<T,channels>::getBins()
{
    return bins;
}

/*
 * Get the number of histograms computed
 * @return the current number of histograms computed by this class
 */
template<typename T, size_t channels>
size_t CvHistograms<T,channels>::getNbHistograms() const
{
    return nbHistograms;
}

/*
 * Gets the additionnal gray histogram status
 * @return true if additional gray level histogram is computed,
```



```
 * false otherwise
 */
template<typename T, size_t channels>
bool CvHistograms<T,channels>::isComputeGray() const
{
    return computeGray;
}

/*
 * Maximum histograms value;
 * @return the maximum value in all histograms
 */
template<typename T, size_t channels>
float CvHistograms<T,channels>::getMaxValue() const
{
    return maxValue;
}

/*
 * Maximum cumulative histograms value;
 * @return the maximum value in all histograms
 * @note regular cumulative maximum value is the number of pixels in
 * the image, but when timecumulative is activated it can be bigger.
 */
template<typename T, size_t channels>
float CvHistograms<T,channels>::getCMaxValue() const
{
    return cMaxValue;
}

//template<typename T, size_t channels>
//const vector<float *> & CvHistograms<T,channels>::getHistogramValues() const
//{
//  return histograms;
//}

//template<typename T, size_t channels>
//const vector<float *> & CvHistograms<T,channels>::getCumulativeHistogramValues() const
//{
//  return cumulHistograms;
//}


//template<typename T, size_t channels>
//const Mat & CvHistograms<T,channels>::getMonoTransfertFunc() const
//{
//  return monoTransfertFunc;
//}

//template<typename T, size_t channels>
//const Mat & CvHistograms<T,channels>::getColorTransfertFunc() const
//{
//  return colorTransferFunc;
//}

/*
 * Value reading access operator
 * @param i the ith histogram to access. if i is invalid, 0 is returned
 * @param j the jth bin value of the ith histogram to access. if j is
 * invalid, 0 is returned.
 * @param cumulative checks whether to return regular histogram value
 * or cumulative histogram value
 * @return the value in the jth bin of the ith histogram
 * @par usage :
 * @code
 *  float jthValue = myHist(i,j);
 *  float jthCumulativeValue = myHist(i,j,true);
 * @endcode
 */
template<typename T, size_t channels>
float CvHistograms<T,channels>::operator ()(size_t i, size_t j,
                                            bool cumulative) const
{
    if (i < nbHistograms)
    {
        if (j < bins)
        {
            if (¬cumulative)
            {
                return (const float) histograms[i][j];
            }
            else
```

```
329             {
330                 return (const float) cumulHistograms[i][j];
331             }
332         }
333         else
334         {
335             cerr << "CvHistograms::operator() const invalid second index "
336                  << "j = " << j << endl;
337             return operator()(i,bins-1);
338         }
339     }
340     else
341     {
342         cerr << "CvHistograms::operator() const invalid first index i = "
343              << i << endl;
344         return operator()(nbHistograms-1,j);
345     }
346 }
347
348 /*
349  * Value reading/writing access operator
350  * @param i the ith histogram to access. if i is invalid, 0 is returned
351  * @param j the jth bin value of the ith histogram to access. if j is
352  * invalid, 0 is returned.
353  * @param cumulative checks whether to return regular histogram value
354  * or cumulative histogram value
355  * @return the value in the jth bin of the ith histogram
356  * @par usage :
357  * @code
358  *  float myHist(i,j) = jthValue;
359  *  float myHist(i,j,true) = jthCumulativeValue;
360  * @endcode
361  */
362 template<typename T, size_t channels>
363 float & CvHistograms<T,channels>::operator ()(size_t i, size_t j,
364                                               bool cumulative)
365 {
366     if (i < nbHistograms)
367     {
368         if (j < bins)
369         {
370             if (¬cumulative)
371             {
372                 return histograms[i][j];
373             }
374             else
375             {
376                 return cumulHistograms[i][j];
377             }
378         }
379         else
380         {
381             cerr << "CvHistograms::operator() invalid second index j = "
382                  << j << endl;
383             return operator()(i,bins-1);
384         }
385     }
386     else
387     {
388         cerr << "CvHistograms::operator() invalid first index i = " << i
389              << endl;
390         return operator()(nbHistograms-1,j);
391     }
392 }
393
394 /*
395  * Time cumulative histogram status read access
396  * @return the time cumulative histogram status
397  */
398 template<typename T, size_t channels>
399 bool CvHistograms<T,channels>::isTimeCumulative() const
400 {
401     return timeCumulative;
402 }
403
404 /*
405  * Time cumulative histogram status read access
406  * @param value the value to set for time cumulative status
407  */
408 template<typename T, size_t channels>
409 void CvHistograms<T,channels>::setTimeCumulative(bool value)
410 {
```



```
411     timeCumulative = value;
412 }
413
414 /*
415  * Toggles time cumulation value
416  */
417 //template<typename T, size_t channels>
418 //void CvHistograms<T,channels>::togglesTimeCumulative()
419 //{
420 //  timeCumulative = !timeCumulative;
421 //}
422
423 /*
424  * Cumulative histogram status read access
425  * @return the cumulative histogram status
426  */
427 template<typename T, size_t channels>
428 bool CvHistograms<T,channels>::isCumulative() const
429 {
430     return showCumulative;
431 }
432
433 /*
434  * Cumulative histogram status read access
435  * @param value the value to set for cumulative status
436  */
437 template<typename T, size_t channels>
438 void CvHistograms<T,channels>::setCumulative(bool value)
439 {
440     showCumulative = value;
441 }
442
443 /*
444  * Toggles if cumulative or regular histograms should be shown
445  */
446 //template<typename T, size_t channels>
447 //void CvHistograms<T,channels>::togglesCumulative()
448 //{
449 //  showCumulative = !showCumulative;
450 //}
451
452 /*
453  * Ith histogram component shown status read access
454  * @param i the ith histogram component
455  * @return true if this component show status is true
456  */
457 template<typename T, size_t channels>
458 bool CvHistograms<T,channels>::isShowComponent(const size_t i) const
459 {
460     if (i < nbHistograms)
461     {
462         return showComponent[i];
463     }
464     else
465     {
466         return false;
467     }
468 }
469
470 /*
471  * Ith histogram component shown status write access
472  * @param i the ith histogram component
473  * @param value the value to set for this component show status
474  */
475 template<typename T, size_t channels>
476 void CvHistograms<T,channels>::setShowComponent(const size_t i,
477                                                 const bool value)
478 {
479 //  clog << "Set Showcomponent nÂ° " << i << (value ? "true" : "false") << endl;
480     if (i < nbHistograms)
481     {
482         showComponent[i] = value;
483     }
484 }
485
486 /*
487  * Toggles if ith histogram component should be drawn
488  * @param i the if component to show or hide
489  * @return true if the ith component has been switched, or false
490  * if it could not be switched (because of invalid index for instance).
491  */
492 //template<typename T, size_t channels>
```

```
493 //bool CvHistograms<T,channels>::togglesComponent(size_t i)
494 //{
495 //  if (i < nbHistograms)
496 //  {
497 //      showComponent[i] = !showComponent[i];
498 //      return true;
499 //  }
500 //  else
501 //  {
502 //      return false;
503 //  }
504 //}
505
506 /**
507  * Update histogram, LUT and resulting images
508  */
509 template<typename T, size_t channels>
510 void CvHistograms<T,channels>::update(void)
511 {
512     clock_t start;
513     processTime = 0;
514
515     // Compute histogram
516     start = clock();
517
518     updateHistogram();
519
520     updateHistogramTime1 = clock() - start;
521     processTime += updateHistogramTime1;
522
523     // Compute requested LUT
524     start = clock();
525
526     lut = computeLUT();
527
528     computeLUTTime = clock() - start;
529     processTime += computeLUTTime;
530
531     if (isLUTUpdated())
532     {
533         // draw TransfertFunction to lutDisplayFrame
534         start = clock();
535
536         drawTransfertFunc(lut);
537
538         drawLUTTime = clock() - start;
539         processTime += drawLUTTime;
540     }
541
542     // Try to apply LUT
543     start = clock();
544
545     bool lutApplied = drawTransformedImage();
546
547     applyLUTTime = clock() - start;
548     processTime += applyLUTTime;
549
550     if (lutApplied)
551     {
552         // if LUT has been applied histogram should be updated
553         start = clock();
554
555         updateHistogram();
556
557         updateHistogramTime2 = clock() - start;
558         processTime += updateHistogramTime2;
559     }
560     else
561     {
562         updateHistogramTime2 = 0;
563     }
564
565     // Finally draw Histogram
566     start = clock();
567
568     drawHistograms();
569
570     drawHistogramTime = clock() - start;
571     processTime += drawHistogramTime;
572 }
573
574 /*
```



```
575  * Update histograms values.
576  */
577 template<typename T, size_t channels>
578 void CvHistograms<T,channels>::updateHistogram(void)
579 {
580     maxValue = 0.0;
581     cMaxValue = 0.0;
582
583     // reset histograms values if necessary
584     if (¬timeCumulative)
585     {
586         // reset histograms values (including evt gray level histogram)
587         for (size_t i=0; i < nbHistograms; i++)
588         {
589             float * h = histograms[i];
590             for (size_t j=0; j < bins; j++)
591             {
592                 h[j] = 0.0;
593             }
594         }
595     }
596
597     // creating iterators over image
598     MatConstIterator_<Vec<T, channels> > iterator =
599         sourceImage→begin<Vec<T, channels> >();
600     MatConstIterator_<Vec<T, channels> > end =
601         sourceImage→end<Vec<T, channels> >();
602
603     // updateHistogram histograms values
604     for (; iterator ≠ end; ++iterator)
605     {
606         Vec<T,channels> pixel = *iterator;
607
608         for (size_t i=0; i < channels; i++)
609         {
610             // updateHistogram corresponding histogram bin
611             float histValue = ++histograms[i][(size_t)pixel[i]];
612
613             // updateHistogram max value if needed
614             if (histValue > maxValue)
615             {
616                 maxValue = histValue;
617             }
618         }
619     }
620
621     // eventually updates gray level histogram
622     if (computeGray ∧ (channels ≡ 3))
623     {
624         for (size_t l=0; l < channels; l++)
625         {
626             for (size_t i=0; i < bins; i++)
627             {
628                 histograms[HIST_GRAY][i] += BGR2Gray[l] * histograms[l][i];
629             }
630         }
631     }
632
633     // update cumulative histograms
634     for (size_t h=0; h < nbHistograms; h++)
635     {
636         float * regularHistogram = histograms[h];
637         float * cumulativeHistogram = cumulHistograms[h];
638
639         size_t b;
640         cumulativeHistogram[0] = regularHistogram[0];
641         for (b=1; b < bins; b++)
642         {
643             cumulativeHistogram[b] = cumulativeHistogram[b-1]
644                                    + regularHistogram[b];
645         }
646
647         // b == bins now, so checks if last is greater than max value
648         if (cumulativeHistogram[b-1]> cMaxValue)
649         {
650             cMaxValue = cumulativeHistogram[b-1];
651         }
652     }
653 }
654
655 /*
656  * Gets Histogram display frame
```

```
 * @return the image histogram is drawn in
 */
//template<typename T, size_t channels>
//const Mat & CvHistograms<T,channels>::getHistogramImage(void) const
//{
//  return histDisplayFrame;
//}

/*
 * Gets Histogram display frame pointer
 * @return the image histogram is drawn in
 */
//template<typename T, size_t channels>
//Mat * CvHistograms<T,channels>::getHistogramImagePtr(void)
//{
//  return &histDisplayFrame;
//}

/*
 * Draws selected histogram(s) in drawing frame and returns the drawing
 * frame
 * @return the updated drawing frame.
 * @post depending on several attributes one or several histograms
 * have bee drawn in the drawing frame wich is returned
 *  - if #showCumulative is true then cumulative histograms are drawn
 *  otherwise regular histograms are drawn
 *  - each histogram is drawn only if its showComponent[i] is true.
 */
template<typename T, size_t channels>
void CvHistograms<T,channels>::drawHistograms(void)
{
    float curveStep = (float)histWidth / (float)bins;
    vector<float *> * valuesPtr;
    float max;
    if (showCumulative)
    {
        valuesPtr = &cumulHistograms;
        max = cMaxValue;
    }
    else
    {
        valuesPtr = &histograms;
        max = maxValue;
    }

    // Fill the drawing frame with black
    rectangle(histDisplayFrame,
              Point(0,0),
              Point(histWidth-1, histHeight-1),
              Scalar(0x00,0x00,0x00,0x00),
              CV_FILLED);

    // Draw the bins (reversed)
    for (size_t h=0; h < nbHistograms; h++)
    {
        // fills this color histogram frame with black
        rectangle(histComponents[h],
                  Point(0,0),
                  Point(histWidth-1, histHeight-1),
                  Scalar(0x00,0x00,0x00,0x00),
                  CV_FILLED);

        // if this color histogram should be drawn
        if (showComponent[h])
        {
            for (size_t i=0; i < bins; i++)
            {
                // draws each bin (reversed) in this color hist. frame
                rectangle(histComponents[h], // the image to draw in
                          Point(i*curveStep,histHeight-1), // first corner of this bin
                          Point((i+1)*curveStep, // second corner of this bin
                                histHeight-1-cvRound(((*valuesPtr)[h][i]/max) *
                                                     histHeight)),
                          displayColors[h], // current color
                          CV_FILLED, // filled rectangle
                          CV_AA); // antialiased line
            }
            // adds this color histogram frame to the drawing frame
            add(histDisplayFrame, histComponents[h], histDisplayFrame);
        }
    }
}
```



```

/*
 * Gets Transfert Func display frame
 * @return the image transfert func is drawn in
 */
//template<typename T, size_t channels>
//const Mat & CvHistograms<T,channels>::getTransfertFuncImage(void) const
//{
//  return lutDisplayFrame;
//}

/*
 * Gets Transfert Func display frame
 * @return the image transfert func is drawn in
 */
//template<typename T, size_t channels>
//Mat * CvHistograms<T,channels>::getTransfertFuncImagePtr(void)
//{
//  return &lutDisplayFrame;
//}

/*
 * Draws selected transfert function in drawing frame and returns the
 * drawing frame
 * @param lut the LUT to draw : the LUT may contains 1 or several
 * channels
 * @return the updated drawing frame
 */
template<typename T, size_t channels>
void CvHistograms<T,channels>::drawTransfertFunc(const Mat * lut)
{
    float curveStep = (float)lutWidth / (float)bins;

    const Mat * currentLUT;

    if (lut ≠ NULL)
    {
        currentLUT = lut;
    }
    else // identity LUT should be computed
    {
        currentLUT = computeLinearGrayLUT();
    }

    size_t lutChannels = (size_t) currentLUT→channels();

    // Fill the drawing frame with black
    rectangle(lutDisplayFrame,
              Point(0,0),
              Point(lutWidth-1, lutHeight-1),
              Scalar(0x00,0x00,0x00,0x00),
              CV_FILLED);

    // Draw the bins (reversed)
    if (lutChannels ≡ 1)
    {
        // draws directly in histDisplayFrame with white color
        for (size_t i = 0; i < bins; i++)
        {
            rectangle(
                lutDisplayFrame, // the image to draw in
                Point(i * curveStep, lutHeight - 1), // first corner of this bin
                Point(
                    (i + 1) * curveStep, // second corner of this bin
                    lutHeight - 1 - cvRound(
                            ((float) currentLUT→at<T>(0, i) / bins)
                                * lutHeight)), displayColors[3], // current color
                CV_FILLED, // filled rectangle
                CV_AA); // antialiased line

        }
    }
    else // lutChannels == 3 or others
    {
        // draws in each colorLUTFrams and adds it to histDisplayFrame
        for (size_t c=0; c < lutChannels; c++)
        {
            if (showComponent[c])
            {
                // Fill the color drawing frame with black
                rectangle(lutComponents[c],
                          Point(0,0),
```

```
                            Point(lutWidth-1, lutHeight-1),
                            Scalar(0x00,0x00,0x00,0x00),
                            CV_FILLED);

                    for (size_t i = 0; i < bins; i++)
                    {
                        rectangle(
                            lutComponents[c], // the image to draw in
                            Point(i * curveStep, lutHeight - 1), // first corner of this bin
                            Point(
                                (i + 1) * curveStep, // second corner of this bin
                                lutHeight - 1 - cvRound(
                                        ((float) currentLUT->at<Vec<T, channels> >(
                                                0, i)[c] / bins) * lutHeight)),
                            displayColors[c], // current color
                            CV_FILLED, // filled rectangle
                            CV_AA); // antialiased line
                    }
                    add(lutDisplayFrame, lutComponents[c], lutDisplayFrame);
                }
            }
        }
}

/*
 * Indicates if LUT has been updated or if it has not changed
 * @return true if LUT has been updated
 */
template<typename T, size_t channels>
bool CvHistograms<T,channels>::isLUTUpdated() const
{
    return lutUpdated;
}

/*
 * Gets the current LUT type
 * @return the current LUT type
 */
template<typename T, size_t channels>
typename CvHistograms<T, channels>::TransfertType
CvHistograms<T,channels>::getLutType() const
{
    return lutType;
}

/*
 * Sets the current LUT type
 * @param lutType the new LUT type
 */
template<typename T, size_t channels>
void CvHistograms<T,channels>::setLutType(const TransfertType lutType)
{
    previousLutType = this->lutType;

    computeLUTTime = 0;
    drawLUTTime = 0;
    applyLUTTime = 0;
    updateHistogramTime2 = 0;

    if (lutType < NBTRANS)
    {
        this->lutType = lutType;
    }
    else
    {
        this->lutType = NONE;
    }
}

/*
 * Gets the current parameter value for LUTs using a percentage parameter
 * @return the current LUT parameter
 */
template<typename T, size_t channels>
float CvHistograms<T,channels>::getLUTParam() const
{
    return lutParam;
}

/*
 * Sets the current LUT % parameter
 * @param lutParam the new LUT parameter
```



```
 */
template<typename T, size_t channels>
void CvHistograms<T,channels>::setLUTParam(float currentParam)
{
    previousLutParam = lutParam;

    if (currentParam > maxParam)
    {
        this->lutParam = maxParam;
    }
    else if (currentParam < minParam)
    {
        this->lutParam = minParam;
    }
    else
    {
        this->lutParam = currentParam;
    }
}

/*
 * Gets the transformed image after drawTransformedImage
 * @return
 */
//template<typename T, size_t channels>
//const Mat & CvHistograms<T,channels>::getTransformedImage() const
//{
//  return outDisplayFrame;
//}

/*
 * Gets the transformed image pointer after drawTransformedImage
 * @return
 */
//template<typename T, size_t channels>
//Mat * CvHistograms<T,channels>::getTransformedImagePtr()
//{
//  return &outDisplayFrame;
//}

/*
 * Return processor processing time of step index [default implementation
 * returning only processTime, should be reimplemented in subclasses]
 * @param index index of the step which processing time is required,
 * 0 indicates all steps, and values above 0 indicates step #. If
 * required index is bigger than number of steps than all steps value
 * should be returned.
 * @return the processing time of step index.
 * @note should be reimplemented in subclasses in order to define
 * time/feature behaviour
 */
template<typename T, size_t channels>
double CvHistograms<T,channels>::getProcessTime(const size_t index) const
{
    switch(index)
    {
        case (CvHistograms<T,channels>::UPDATE_HISTOGRAM):
            return (double) updateHistogramTime1;
        case (CvHistograms<T,channels>::COMPUTE_LUT):
            return (double) computeLUTTime;
        case (CvHistograms<T,channels>::DRAW_LUT):
            return (double) drawLUTTime;
        case (CvHistograms<T,channels>::APPLY_LUT):
            return (double) applyLUTTime;
        case (CvHistograms<T,channels>::UPDATE_HISTOGRAM_AFTER_LUT):
                return (double) updateHistogramTime2;
        case (CvHistograms<T,channels>::DRAW_HISTOGRAM):
                return (double) drawHistogramTime;
        default:
            return (double) processTime;
    }
}


/*
 * Compute linear transfert function (LUT) : no change in image levels
 * @return the LUT containing the corresponding transfert function,
 * the returned matrix contains only one channel corresponding to
 * the graylevel LUT which should be applied to all color channels of
 * the image
 * @post the result is stored in monoTransfertFunc
 * @note It's useless to compute a color Linear LUT since all channels
```

```
 * would contain the exact same values.
 */
template<typename T, size_t channels>
Mat * CvHistograms<T,channels>::computeLinearGrayLUT(void)
{
    for (size_t i=0; i < bins; i++)
    {
        monoTransfertFunc.at<T>(0,i) = i;
    }

    return &monoTransfertFunc;
}

/*
 * Compute linear transfert function (LUT) : no change in image levels
 * @return the LUT containing the corresponding transfert function,
 * the returned matrix contains 3 channels corresponding to
 * the color LUT which should be applied to all color channels of
 * the image
 * @post the result is stored in colorTransfertFunc
 * @note It's useless to compute a color Linear LUT since all channels
 * would contain the exact same values.
 */
template<typename T, size_t channels>
Mat * CvHistograms<T,channels>::computeLinearColorLUT(void)
{
    for (size_t c=0; c < channels; c++)
    {
        for (size_t i=0; i < bins; i++)
        {
            colorTransferFunc.at<Vec<T,channels> >(0,i)[c] = i;
        }
    }

    return &colorTransferFunc;
}


/*
 * Compute the optimal dynamic LUT for preserving "percentDynamic"
 * percent of the whole image ligthness range.
 * @param percentDynamic the gray level percentage to spread on the
 * whole (100%) gray level range in the image
 * @return the LUT containing the corresponding transfert function,
 * the returned matrix contains only one channel corresponding to
 * the graylevel LUT which should be applied to all color channels of
 * the image
 * @post the result is stored in monoTransfertFunc
 *          ^
 * maxVal   |              ________
 *          |          /
 *          |         /
 *          |        /          dy/dx = (maxVal-minVal) / (maxThresIndex - minthresIndex - 1)
 *          |       /
 *          |      /
 * minVal   |_____/     maxThresIndex
 *          +----+------+-------------->
 *    minThresIndex
 */
template<typename T, size_t channels>
Mat * CvHistograms<T,channels>::computeGrayOptimalLUT(unsigned int percentDynamic)
{
    if (computeGray ∧ nbHistograms ≡ 4)
    {
        float threshold = (100 - percentDynamic)/200.0;
        float imageSize = sourceImage→rows * sourceImage→cols;
        float minThres = imageSize * threshold;
        float maxThres = imageSize - minThres;

        size_t minThresIndex = 0;
        size_t maxThresIndex = bins;

        T minVal = 0;
        T maxVal = numeric_limits<T>::max(); // 255 for uchar

        // finds minThresIndex in cumulHistograms[HIST_GRAY][i=0..bins]
        // TODO Ã  complÃ©ter ...

        // finds maxThresIndex in cumulHistograms[HIST_GRAY][i=0..bins]
        // TODO Ã  complÃ©ter ...

        // fill monoTransfertFunc before minThresIndex with minVal
```



```
        // TODO Ã  complÃ©ter ...

        // fill monoTransfertFunc between minThresIndex and maxThesIndex with Dy/Dx Values
        float slope = (float)(maxVal - minVal) /
                      (float)(maxThresIndex-1-minThresIndex);

        // TODO Ã  complÃ©ter ...

        // fill monoTransfertFunc after maxThresIndex with maxVal
        // TODO Ã  complÃ©ter ...
    }
    else
    {
        cerr << "CvHistograms<T,channels>::computeGrayOptimalLUT : "
             << "There is no gray histogram !" << endl;
    }

    return &monoTransfertFunc;
}

/*
 * Compute the optimal dynamic LUTs (one for each channel) for preserving
 * "percentDynamic" percent of the whole image color ranges.
 * @param percentDynamic the colors level percentage to spread on the
 * whole (100%) colors level range in the image
 * @return the LUT containing the corresponding transfert functions,
 * the returned matrix contains as much channels as the image and
 * corresponding to the color level LUT which should be applied to
 * each color channels of the image
 * @post the result is stored in colorTransfertFunc
 */
template<typename T, size_t channels>
Mat * CvHistograms<T,channels>::computeColorOptimalLUT(unsigned int percentDynamic)
{
    float threshold = (1 - (percentDynamic/100.0))/2.0;
    float imageSize = sourceImage→rows * sourceImage→cols;
    float minThres = imageSize * threshold;
    float maxThres = imageSize - minThres;

    size_t minThresIndex[channels];
    size_t maxThresIndex[channels];
    T minVal = 0;
    T maxVal = numeric_limits<T>::max(); // 255 for uchar;
    float slope[channels];

    for (size_t c=0; c < channels; c++)
    {
        minThresIndex[c] = 0;
        maxThresIndex[c] = bins;

        // finds minThresIndex in cumulHistograms[c][...] for this channel
        // TODO Ã  complÃ©ter ...

        // finds maxThresIndex in cumulHistograms[c][...] for this channel
        // TODO Ã  complÃ©ter ...

        // fill colorTransferFunc before minThresIndex with minVal
        // TODO Ã  complÃ©ter ...

        // ramp slope for this channel = Dy/Dx
        slope[c] = (float)(maxVal - minVal) /
                   (float)(maxThresIndex[c]-1-minThresIndex[c]);

        // fill colorTransferFunc between minThresIndex and maxThesIndex with regular ramp
        // TODO Ã  complÃ©ter ...

        // fill colorTransferFunc after maxThresIndex with maxVal
        // TODO Ã  complÃ©ter ...
    }

    return &colorTransferFunc;
}


/*
 * Computes the transfert function corresponding to gray level
 * equalization
 * @return the matrix containing the gray level equalization LUT to
 * apply on the image
 * @post the result is stored in monoTransfertFunc
 */
template<typename T, size_t channels>
```

```
Mat * CvHistograms<T,channels>::computeGrayEqualizeLUT(void)
{
    T maxVal = numeric_limits<T>::max();
    if (computeGray ∧ nbHistograms ≡ 4)
    {
        /*
         * Equalisation consists in applying the corresponding cumulative
         * histogram (cumulHistograms[HIST_GRAY][i=0..bins] normalized to maxVal)
         * as a mono transfert function
         */
        // TODO Ã  complÃ©ter ...
    }
    else
    {
        cerr << "CvHistograms<T,channels>::computeGrayEqualizeLUT : "
             << "There is no gray level histogram" << endl;
    }

    return &monoTransfertFunc;
}


/*
 * Computes the transfert functions corresponding to each channel
 * level equalization
 * @return the matrix contaning each channel level equalization LUT to
 * apply on the image
 * @post the result is stored in colorTransferFunc
 */
template<typename T, size_t channels>
Mat * CvHistograms<T,channels>::computeColorEqualizeLUT(void)
{
    T maxVal = numeric_limits<T>::max(); // 255 for uchar;
    /*
     * Color equalisation consists in applying the corresponding cumulative
     * histogram (cumulHistograms[c=0..channels][i=0..bins] normalized to maxVal)
     * as a color transfert function
     */
    for (size_t c=0; c < channels; c++)
    {
        // TODO Ã  complÃ©ter ...
    }

    return &colorTransferFunc;
}

/*
 * Compute the LUT corresponding to thresholded image with tPercent
 * of the pixel population on each side of the threshold according
 * to the cumulative gray level histogram
 * @param tPercent percent of the population on each side of the
 * threshold
 * @return the LUT containing the corresponding transfert function,
 * the returned matrix contains only one channel corresponding to
 * the graylevel LUT which should be applied to all color channels of
 * the image
 * @post the result is stored in monoTransfertFunc
 */
template<typename T, size_t channels>
Mat * CvHistograms<T,channels>::computeGrayThresholdLUT(float tPercent)
{
    T minVal = 0;
    T maxVal = numeric_limits<T>::max(); // 255 for uchar;

    if (computeGray ∧ nbHistograms ≡ 4)
    {
        if (tPercent > 0.0 ∧ tPercent < 100.0)
        {
            // determine threshold population count
            float thresLevel = (float)cMaxValue * (tPercent/100);

            // initialize thresIndex at any possible value;
            size_t thresIndex = bins/2;

            // search for thresLevel in cumulHistograms[HIST_GRAY][i=0..bins]
            // TODO Ã  complÃ©ter ...

            // apply minVal in monoTransfertFunc to population below threshold index
            // TODO Ã  complÃ©ter ...

            // apply maxVal in monoTransfertFunc to population above threshold index
            // TODO Ã  complÃ©ter ...
```



```

        }
        else
        {
            cerr << "CvHistograms<T,channels>::computeGrayThresholdLUT : "
                 << "percentage should be between 0 and 100 : " << tPercent
                 << endl;
        }
    }
    else
    {
        cerr << "CvHistograms<T,channels>::computeGrayThresholdLUT : "
             << "There is no gray level histogram" << endl;
    }

    return &monoTransfertFunc;
}

/*
 * Compute the LUT corresponding to thresholded image with tPercent
 * of the pixel components population on each side of the
 * thresholds according to the cumulative color histograms
 * @param tPercent percent of the population on each side of the
 * thresholds
 * @return the matrix contaning each channel level equalization LUT to
 * apply on the image
 * @post the result is stored in colorTransferFunc
 */
template<typename T, size_t channels>
Mat * CvHistograms<T,channels>::computeColorThresholdLUT(float tPercent)
{
    T minVal = 0;
    T maxVal = numeric_limits<T>::max(); // 255 for uchar;
    size_t mThresIndex[channels];

    if (tPercent > 0.0 ∧ tPercent < 100.0)
    {
        // determine threshold population count
        float thresLevel = (float)cMaxValue * (tPercent/100);

        for (size_t c=0; c < channels; c++)
        {
            // initialize thresIndex at any possible value;
            size_t thresIndex = bins/2;

            // search for thresLevel in cumulHistograms[c][i=0..bins]
            // TODO Ã  complÃ©ter ...

            mThresIndex[c] = thresIndex;

            // apply minVal in colorTransferFunc to population below threshold index
            // TODO Ã  complÃ©ter ...

            // apply maxVal in colorTransferFunc to population above threshold index
            // TODO Ã  complÃ©ter ...
        }
    }
    else
    {
        cerr << "CvHistograms<T,channels>::computeGrayThresholdLUT : "
             << "percentage should be between 0 and 100 : " << tPercent
             << endl;
    }

    return &colorTransferFunc;
}

/*
 * Compute gamma LUT.
 * \f$y(k) = x(k)^{\gamma}\f$
 * @param tPercent
 * @return the matrix containing the gamma LUT
 */
template<typename T, size_t channels>
Mat * CvHistograms<T,channels>::computeGammaLUT(float tPercent)
{
    /*
     * Gamma varies approximately from
     * 0.25 when tPercent==0% to 4 when tPercent ==100%
     */
    double gamma = 0.4101 * exp(2.3186 * ((double)tPercent/100.0)) - 0.2506;

```

```
1313      // Apply (x^gamma)*bins where x=i/bins in monoTransfertFunc
1314      // TODO Ã  complÃ©ter ...
1315
1316      return &monoTransfertFunc;
1317  }
1318
1319  /*
1320   * Compute the LUT corresponding to negative image
1321   * @return the matrix containing the negative LUT (mono)
1322   */
1323  template<typename T, size_t channels>
1324  Mat * CvHistograms<T,channels>::computeNegativeLUT(void)
1325  {
1326      // Apply (bins - 1 -i) in  monoTransfertFunc
1327      // TODO Ã  complÃ©ter ...
1328
1329      return &monoTransfertFunc;
1330  }
1331
1332  /*
1333   * Compute and returns the current transfert function to be applied
1334   * on the image, eventually with the current LUT parameter
1335   * @return the mono or color LUT matrix to apply on the image depending
1336   * on the lutType
1337   * @see TransfertType
1338   */
1339  template<typename T, size_t channels>
1340  Mat * CvHistograms<T,channels>::computeLUT()
1341  {
1342      Mat * lut = NULL;
1343
1344      lutUpdated = true;
1345
1346      switch(lutType)
1347      {
1348          case NONE :
1349              /*
1350               * Identity LUT
1351               * Linear LUT does not depend on histogram so if previous
1352               * LUT was already Linear then don't compute it again, just
1353               * return the last LUT
1354               */
1355              if (previousLutType ≠ lutType)
1356              {
1357                  lut = computeLinearGrayLUT();
1358              }
1359              else
1360              {
1361                  lut = &monoTransfertFunc;
1362                  lutUpdated = false;
1363              }
1364              break;
1365          case THRESHOLD_GRAY :
1366              /*
1367               * LUT to split pixels below param % to black and pixels over
1368               * param % to white based on graylevel cumulative histogram
1369               */
1370              lut = computeGrayThresholdLUT(lutParam);
1371              break;
1372          case THRESHOLD_COLOR :
1373              /*
1374               * LUT to split param% of the pixel components to black and
1375               * 100-param% to full B, G or R based on cumulative histograms
1376               * components
1377               */
1378              lut = computeColorThresholdLUT(lutParam);
1379              break;
1380          case DYNAMIC_GRAY :
1381              /*
1382               * LUT to spread param% of the pixel levels over 100% of the dynamic
1383               * based on cumulative gray level histogram
1384               */
1385              lut = computeGrayOptimalLUT(lutParam);
1386              break;
1387          case DYNAMIC_COLOR :
1388              /*
1389               * LUT to spread param% of the pixel components levels over 100% of
1390               * the dynamic based on cumulative color histograms
1391               */
1392              lut = computeColorOptimalLUT(lutParam);
1393              break;
1394          case EQUALIZE_GRAY :
```



```
1395              /*
1396               * Gray level histogram equalization LUT
1397               */
1398              lut = computeGrayEqualizeLUT();
1399              break;
1400          case EQUALIZE_COLOR :
1401              /*
1402               * Color components histograms equalization LUTs
1403               */
1404              lut = computeColorEqualizeLUT();
1405              break;
1406          case GAMMA :
1407              /*
1408               * Gamma LUT does not depend on histogram so if previous
1409               * LUT was already Gamma then don't compute it again, just
1410               * return the last LUT
1411               */
1412              if ((previousLutType ≠ lutType) ∨ (previousLutParam ≠ lutParam))
1413              {
1414                  lut = computeGammaLUT(lutParam);
1415              }
1416              else
1417              {
1418                  lut = &monoTransfertFunc;
1419                  lutUpdated = false;
1420              }
1421              break;
1422          case NEGATIVE :
1423              /*
1424               * Negative LUT does not depend on histogram so if previous
1425               * LUT was already Negative then don't compute it again, just
1426               * return the last LUT
1427               */
1428              if (previousLutType ≠ lutType)
1429              {
1430                  lut = computeNegativeLUT();
1431              }
1432              else
1433              {
1434                  lut = &monoTransfertFunc;
1435                  lutUpdated = false;
1436              }
1437              break;
1438          default :
1439              cerr << "CvHistograms<T,channels>::applyLUT : unknown LUT"
1440                   << endl;
1441              break;
1442      }
1443
1444      previousLutType = lutType;
1445      previousLutParam = lutParam;
1446
1447      return lut;
1448  }
1449
1450  /*
1451   * Apply current LUT (if != NULL) to the source image to produce the
1452   * outFrame
1453   * @return true if LUT has been applied, false if lut is NULL or
1454   * lutType is NONE
1455   */
1456  template<typename T, size_t channels>
1457  bool CvHistograms<T,channels>::drawTransformedImage(void)
1458  {
1459      if ((lut ≠ NULL) ∧ (lutType ≠ NONE))
1460      {
1461          LUT(*sourceImage, *lut, outDisplayFrame);
1462          return true;
1463      }
1464      else
1465      {
1466          outDisplayFrame = *sourceImage;
1467          return false;
1468      }
1469  }
1470
1471  /*
1472   * output operator for Histograms
1473   * @param out the output stream
1474   * @param h the histograms to print on the stream
1475   * @return a reference to the output stream so it can be cumulated
1476   */
```

```cpp
template<typename T, size_t channels>
ostream & operator <<(ostream & out, const CvHistograms<T,channels> & h)
{
    for (size_t i = 0; i < h.bins; i++)
    {
        out << i << ":";

        for (size_t j=0; j < h.nbHistograms; j++)
        {
            out << h.histograms[i][j] << " ";
        }

        out << endl;
    }

    return out;
}

// ================================================================================
// Templates proto instanciations
// ================================================================================

// template class instanciation
// for gray level images
template class CvHistograms<uchar, 1>;
template ostream & operator << (ostream &, const CvHistograms<uchar,1> &);

// for BGR or YUV images
template class CvHistograms<uchar, 3>;
template ostream & operator << (ostream &, const CvHistograms<uchar,3> &);
```



```cpp
/*
 * QcvHistograms.h
 *
 *  Created on: 14 fÃ©vr. 2012
 *      Author: davidroussel
 */

#ifndef QCVEHISTOGRAMS_H_
#define QCVEHISTOGRAMS_H_

#include "QcvProcessor.h"
#include "CvHistograms.h"

/**
 * Defines type for Histograms of 8 bits and 3 channels images.
 * @note this is because QObjects subclasses can NOT be templates,
 * so QcvHistograms should inherit CvHistograms<uchar,3> rather than
 * CvHistograms<T,channels>
 */
typedef CvHistograms<uchar,3> CvHistograms8UC3;

/**
 * OpenCV Color Image Histogram processing class with QT flavor
 */
class QcvHistograms: public QcvProcessor, public CvHistograms8UC3
{
    Q_OBJECT

    protected:

        /**
         * String containing update histogram formatted time
         */
        QString updateHistogramTime1String;

        /**
         * String containing formatted LUT computing time
         */
        QString computeLUTTimeString;

        /**
         * String containing formatted LUT drawng time
         */
        QString drawLUTTimeString;

        /**
         * String containing formatted LUT apply time
         */
        QString applyLUTTimeString;

        /**
         * String containing formatted histogram update time after
         * LUT applied
         */
        QString updateHistogramTime2String;

        /**
         * String containing formatted histogram drawing time
         */
        QString drawHistogramTimeString;

    public:

        /**
         * QcvHistograms constructor
         * @param image the source image
         * @param computeGray indicates if an aditionnal gray level histogram
         * should be computed
         * @param drawHeight histogram drawing height
         * @param drawWidth histogram drawing width
         * @param timeCumulation indicates if timecumulation is on for histogram
         * @param imageLock the mutex for concurrent access to the source image.
         * In order to avoid concurrent access to the same image
         * @param updateThread the thread in which this processor should run
         * @param parent parent QObject
         * computation
         */
        QcvHistograms(Mat * image,
                      QMutex * imageLock = NULL,
                      QThread * updateThread = NULL,
                      const bool computeGray = true,
                      const size_t drawHeight = 256,
```

```
                        const size_t drawWidth = 512,
                        const bool timeCumulation = false,
                        QObject * parent = NULL);

        /**
         * QImageHistogram destructor
         */
        virtual ~QcvHistograms();

        /**
         * Time cumulative histogram setting with notification
         * @param value the value to set for time cumulative status
         */
        void setTimeCumulative(bool value);

        /**
         * Cumulative histogram status setting with notification
         * @param value the value to set for cumulative status
         */
        void setCumulative(bool value);

        /**
         * Ith histogram component show setting with notifications
         * @param i the ith histogram component
         * @param value the value to set for this component show status
         */
        void setShowComponent(size_t i,
                              bool value);
        /**
         * Current LUT setting with notification
         * @param lutType the new LUT type
         */
        void setLutType(const TransfertType lutType);

//      /**
//       * Sets the current LUT % parameter with notification
//       * @param lutParam the new LUT parameter
//       */
//      void setLUTParam(float currentParam);

    protected:

        /**
         * Draws selected histogram(s) in drawing frame and notifies the drawing
         * frame
         * @return the updated drawing frame.
         */
        void drawHistograms(void);

        /**
         * Draws selected transfert function in drawing frame and notifies the
         * drawing frame
         * @param lut the LUT to draw : the LUT may contains 1 or several
         * channels
         * @return the updated drawing frame
         */
        void drawTransfertFunc(const Mat * lut);

        /**
         * Apply current LUT (if != NULL) to the source image to produce the
         * outFrame and notifies the drawing frame
         * @return true if LUT has been applied, false if lut is NULL or
         * lutType is NONE
         */
        bool drawTransformedImage(void);

    public slots:
        /**
         * Update computed images slot and sends displayImageChanged signal if
         * required
         */
        void update();

        /**
         * Changes source image slot.
         * Attributes needs to be cleaned up then set up again
         * @param image the new source Image
         */
        void setSourceImage(Mat * image) throw (CvProcessorException);


    signals:
```



```
        /**
         * Signal sent when update is completed AND transformed image is updated
         */
        void outImageUpdated();

        /**
         * Signal sent when transformed image has been reallocated
         * @param image the new transformed image
         */
        void outImageChanged(Mat * image);

        /**
         * Signal sent when update is completed AND histogram image changes
         */
        void histogramImageUpdated();

        /**
         * Signal sent when histogram image has been reallocated
         * @param image the new histogram image
         */
        void histogramImageChanged(Mat * image);

        /**
         * Signal sent when update is completed AND LUT image changes
         */
        void lutImageUpdated();

        /**
         * Signal sent when lut image has been reallocated;
         * @param image the new LUT image
         */
        void lutImageChanged(Mat * image);

        /**
         * Signal emitted when histogram is updated with a new image
         * @param formattedValue string containing the formatted time value
         */
        void histogramTime1Updated(const QString & formattedValue);

        /**
         * Signal emitted when LUT is computed
         * @param formattedValue string containing the formatted time value
         */
        void computeLUTTimeUpdated(const QString & formattedValue);

        /**
         * Signal emitted when LUT is drawn
         * @param formattedValue string containing the formatted time value
         */
        void drawLUTTimeUpdated(const QString & formattedValue);

        /**
         * Signal emitted when LUT is applied on image
         * @param formattedValue string containing the formatted time value
         */
        void applyLUTTimeUpdated(const QString & formattedValue);

        /**
         * Signal emitted when histogram is updated after LUT has been applied
         * @param formattedValue string containing the formatted time value
         */
        void histogramTime2Updated(const QString & formattedValue);

        /**
         * Signal emitted when histogram is drawn
         * @param formattedValue string containing the formatted time value
         */
        void drawHistogramTimeUpdated(const QString & formattedValue);
};

#endif /* QCVEHISTOGRAMS_H_ */
```

```
/*
 * QcvHistograms.cpp
 *
 *  Created on: 14 fÃ©vr. 2012
 *      Author: davidroussel
 */

#include <QDebug>
#include "QcvHistograms.h"

/*
 * QcvHistograms constructor
 * @param image the source image
 * @param computeGray indicates if an aditionnal gray level histogram
 * should be computed
 * @param drawHeight histogram drawing height
 * @param drawWidth histogram drawing width
 * @param timeCumulation indicates if timecumulation is on for histogram
 * @param imageLock the mutex for concurrent access to the source image.
 * In order to avoid concurrent access to the same image
 * @param updateThread the thread in which this processor should run
 * @param parent parent QObject
 * computation
 */
QcvHistograms::QcvHistograms(Mat * image,
                             QMutex * imageLock,
                             QThread * updateThread,
                             const bool computeGray,
                             const size_t drawHeight,
                             const size_t drawWidth,
                             const bool timeCumulation,
                             QObject * parent) :
    CvProcessor(image),
    QcvProcessor(image, imageLock, updateThread, parent),
    CvHistograms8UC3(image, computeGray, drawHeight, drawWidth, timeCumulation)
{
    QcvProcessor::setNumberFormat("%7.0f");
}

/*
 * QImageHistogram destructor
 */
QcvHistograms::~QcvHistograms()
{
    updateHistogramTime1String.clear();
    computeLUTTimeString.clear();
    drawLUTTimeString.clear();
    applyLUTTimeString.clear();
    updateHistogramTime2String.clear();
    drawHistogramTimeString.clear();
}

/*
 * Update computed images and sends displayImageChanged signal if
 * required
 */
void QcvHistograms::update()
{
    if (sourceLock ≠ NULL)
    {
        sourceLock→lock();
        // qDebug() << "QcvHistograms::update : lock";
    }

    /*
     * Update Histogram images
     */
    CvHistograms8UC3::update();

    if (sourceLock ≠ NULL)
    {
        sourceLock→unlock();
        // qDebug() << "QcvHistograms::update : unlock";
    }

    /*
     * emit time measurement signals
     */
    updateHistogramTime1String.sprintf(numberFormat,
                                       getProcessTime(UPDATE_HISTOGRAM));
    emit(histogramTime1Updated(updateHistogramTime1String));

```



```
    computeLUTTimeString.sprintf(numberFormat, getProcessTime(COMPUTE_LUT));
    emit(computeLUTTimeUpdated(computeLUTTimeString));
    if (isLUTUpdated())
    {
        drawLUTTimeString.sprintf(numberFormat, getProcessTime(DRAW_LUT));
        emit(drawLUTTimeUpdated(drawLUTTimeString));
    }

    applyLUTTimeString.sprintf(numberFormat, getProcessTime(APPLY_LUT));
    emit(applyLUTTimeUpdated(applyLUTTimeString));

    if ((lut ≠ NULL) ∧ (lutType ≠ NONE))
    {
        updateHistogramTime2String.sprintf(numberFormat,
                                           getProcessTime(UPDATE_HISTOGRAM_AFTER_LUT));
        emit(histogramTime2Updated(updateHistogramTime2String));
    }

    drawHistogramTimeString.sprintf(numberFormat, getProcessTime(DRAW_HISTOGRAM));
    emit(drawHistogramTimeUpdated(drawHistogramTimeString));

    /*
     * emit updated signal
     */
    QcvProcessor::update(); // emits updated signal
}

/*
 * Changes source image slot.
 * Attributes needs to be cleaned up then set up again
 * @param image the new source Image
 */
void QcvHistograms::setSourceImage(Mat * image) throw (CvProcessorException)
{
    QcvProcessor::setSourceImage(image);

    // notifies any connected component to change source images
    emit outImageChanged(&outDisplayFrame);
    emit histogramImageChanged(&histDisplayFrame);
    emit lutImageChanged(&lutDisplayFrame);
}


/*
 * Time cumulative histogram status read access
 * @param value the value to set for time cumulative status
 */
void QcvHistograms::setTimeCumulative(bool value)
{
    CvHistograms8UC3::setTimeCumulative(value);
    message.clear();
    message.append(tr("Time Cumulative Histogram is "));
    if (value)
    {
        message.append(tr("on"));
    }
    else
    {
        message.append(tr("off"));
    }
    emit sendMessage(message, defaultTimeOut);
}

/*
 * Cumulative histogram status read access
 * @param value the value to set for cumulative status
 */
void QcvHistograms::setCumulative(bool value)
{
    CvHistograms8UC3::setCumulative(value);
    message.clear();
    message.append(tr("Cumulative Histogram is "));
    if (value)
    {
        message.append(tr("on"));
    }
    else
    {
        message.append(tr("off"));
    }

    emit sendMessage(message, defaultTimeOut);
```

```
}

/*
 * Ith histogram component shown status write access
 * @param i the ith histogram component
 * @param value the value to set for this component show status
 */
void QcvHistograms::setShowComponent(size_t i, bool value)
{
    CvHistograms8UC3::setShowComponent(i, value);
    message.clear();
    switch (i)
    {
        case 0:
            message.append(tr("Red"));
            break;
        case 1:
            message.append(tr("Green"));
            break;
        case 2:
            message.append(tr("Blue"));
            break;
        case 3:
            message.append(tr("Gray"));
            break;
        default:
            message.append(tr("Unkown"));
            break;
    }
    message.append(tr(" histogram Component is "));

    if (value)
    {
        message.append(tr("on"));
    }
    else
    {
        message.append(tr("off"));
    }

    emit sendMessage(message, defaultTimeOut);
}

/*
 * Sets the current LUT type
 * @param lutType the new LUT type
 */
void QcvHistograms::setLutType(const TransfertType lutType)
{
    CvHistograms8UC3::setLutType(lutType);
    message.clear();
    message.append(tr("Current transfert function is "));
    switch (lutType)
    {
        case NONE:
            message.append(tr("Identity"));
            break;
        case THRESHOLD_GRAY:
            message.append(tr("Threshold based on gray histogram"));
            break;
        case DYNAMIC_GRAY:
            message.append(tr("Optimal dynamic based on gray histogram"));
            break;
        case EQUALIZE_GRAY:
            message.append(tr("Equalize based on gray histogram"));
            break;
        case THRESHOLD_COLOR:
            message.append(tr("Threshold based on color histograms"));
            break;
        case DYNAMIC_COLOR:
            message.append(tr("Optimal dynamic based on color histograms"));
            break;
        case EQUALIZE_COLOR:
            message.append(tr("Equalize based on color histograms"));
            break;
        case GAMMA:
            message.append(tr("Gamma"));
            break;
        case NEGATIVE:
            message.append(tr("Inverse"));
            break;
        default:
```



```
            message.append(tr("unknown"));
            break;
    }

    emit sendMessage(message, defaultTimeOut);
}

/*
 * Sets the current LUT % parameter
 * @param lutParam the new LUT parameter
 */
//void QcvHistograms::setLUTParam(float currentParam)
//{
//
//}

/*
 * Draws selected histogram(s) in drawing frame and returns the drawing
 * frame
 * @return the updated drawing frame.
 */
void QcvHistograms::drawHistograms(void)
{
    CvHistograms8UC3::drawHistograms();
    emit histogramImageUpdated();
}

/*
 * Draws selected transfert function in drawing frame and returns the
 * drawing frame
 * @param lut the LUT to draw : the LUT may contains 1 or several
 * channels
 * @return the updated drawing frame
 */
void QcvHistograms::drawTransfertFunc(const Mat *lut)
{
    CvHistograms8UC3::drawTransfertFunc(lut);
    emit lutImageUpdated();
}

/*
 * Apply current LUT (if != NULL) to the source image to produce the
 * outFrame
 * @return true if LUT has been applied, false if lut is NULL or
 * lutType is NONE
 */
bool QcvHistograms::drawTransformedImage(void)
{
    bool result = CvHistograms8UC3::drawTransformedImage();
    emit outImageUpdated();
    return result;
}
```

```
/*
 * QcvMatWidget.h
 *
 *  Created on: 28 fÃ©vr. 2011
 *^H      Author: davidroussel
 */

#ifndef QCVMATWIDGET_H_
#define QCVMATWIDGET_H_

#include <QWidget>
#include <QHBoxLayout>
#include <QMouseEvent>
#include <QPoint>

#include <cv.h>
using namespace cv;

/**
 * Abstract widget to show OpenCV Mat image into QT.
 * Should be refined in
 *  - QcvMatWidgetLabel
 *  - QcvMatWidgetImage
 *  - QcvMatWidgetGL
 */
class QcvMatWidget : public QWidget
{
    Q_OBJECT

    public:
        /**
         * Mouse sensivity of the image widget
         */
        typedef enum
        {
            /**
             * Sensitive to no mouse click or drag
             */
            MOUSE_NONE = 0,
            /**
             * Sensitive to mouse clicks
             */
            MOUSE_CLICK = 1,
            /**
             * Sensitive to mouse drag
             */
            MOUSE_DRAG = 2,
            /**
             * Sensitive to mouse click and drag
             */
            MOUSE_CLICK_AND_DRAG = 3
        } MouseSense;

    protected:
        /**
         * The widget layout
         */
        QHBoxLayout * layout;

        /**
         * The OpenCV BGR or gray image
         */
        Mat * sourceImage;

        /**
         * The OpenCV RGB image converted from gray or BGR OpenCV image
         */
        Mat displayImage;

        /**
         * Default size when no image has been set
         */
        static QSize defaultSize;

        /**
         * the aspect ratio ofthe image to draw
         */
        double aspectRatio;

        /**
         * Default aspect ratio when image is not set yet
         */
```



```
        static double defaultAspectRatio;

        /**
         * Indicate a mouse button is currently pressed within the widget
         */
        bool mousePressed;

        /**
         * Indicate a mouse is moved after a button has been pressed
         */
        bool mouseMoved;

        /**
         * Mouse sensivity
         */
        MouseSense mouseSense;

        /**
         * mouse pressed location
         */
        QPoint pressedPoint;

        /**
         * Mouse pressed button
         */
        Qt::MouseButton pressedButton;

        /**
         * mouse drag location
         */
        QPoint draggedPoint;

        /**
         * mouse release location
         */
        QPoint releasedPoint;

        /**
         * Selection rectangle
         */
        QRect selectionRect;

        /**
         * Drawing color
         */
        static const Scalar drawingColor;

        /**
         * Drawing width
         */
        static const int drawingWidth;

//      size_t count;

    public:

        /**
         * OpenCV QT Widget default constructor
         * @param parent parent widget
         * @param mouseSense mouse sensivity
         */
        QcvMatWidget(QWidget *parent = NULL,
                     MouseSense mouseSense = MOUSE_NONE);

        /**
         * OpenCV QT Widget constructor
         * @param sourceImage the source image
         * @param parent parent widget
         * @param mouseSense mouse sensivity
         * @pre sourceImage is not NULL
         */
        QcvMatWidget(Mat * sourceImage,
                     QWidget *parent = NULL,
                     MouseSense mouseSense = MOUSE_NONE);

        /**
         * OpenCV Widget destructor.
         * Releases displayImage.
         */
        virtual ~QcvMatWidget(void);

//^H    ^H  /**
```

```
//^H     ^H   * Widget minimum size is set to the contained image size
//^H     ^H   * @return le size of the image within
//^H     ^H   */
//^H     ^H  QSize minimumSize() const;

        /**
         * Size hint (because size depends on sourceImage properties)
         * @return size obtained from sourceImage or defaultSize if sourceImage
         * is not set yet
         */
        QSize sizeHint() const;

        /**
         * Gets Mat widget mouse clickable status
         * @return true if widget is sensitive to mouse click
         */
        bool isMouseClickable() const;

        /**
         * Gets Mat widget mouse dragable status
         * @return true if widget is sensitive to mouse drag
         */
        bool isMouseDragable() const;

    protected:

        /**
         * paint event reimplemented to draw content (in this case only
         * draw in display image since final rendering method is not yet available)
         * @param event the paint event
         */
        virtual void paintEvent(QPaintEvent * event);

        /**
         * Widget setup
         * @post new Layout has been created and set for this widget
         */
        void setup();

        /**
         * Converts BGR or Gray source image to RGB display image
         * @pre sourceImage is not NULL
         * @post BGR or Gray source image has been converted to RGB displayimage
         * @see #sourceImage
         * @see #displayImage
         */
        void convertImage();

        /**
         * Callback called when mouse button pressed event occurs.
         * reimplemented to send pressPoint signal when left mouse button is
         * pressed
         * @param event mouse event
         */
        void mousePressEvent(QMouseEvent *event);

        /**
         * Callback called when mouse move event occurs.
         * reimplemented to send dragPoint signal when mouse is dragged
         * (after left mouse button has been pressed)
         * @param event mouse event
         */
        void mouseMoveEvent(QMouseEvent *event);

        /**
         * Callback called when mouse button released event occurs.
         * reimplemented to send releasePoint signal when left mouse button is
         * released
         * @param event mouse event
         */
        void mouseReleaseEvent(QMouseEvent *event);

        /**
         * Draw Cross
         * @param p the cross center
         */
        virtual void drawCross(const QPoint & p);

        /**
         * Draw rectangle
         * @param r the rectangle to draw
         */
```



```
        virtual void drawRectangle(const QRect & r);

//      /*
//       * paint event reimplemented to draw content
//       * @param event the paint event
//       */
//      virtual void paintEvent(QPaintEvent * event) = 0;

        /**
         * Modifiy selectionRect using two points
         * @param p1 first point
         * @param p2 second point
         */
        void selectionRectFromPoints(const QPoint & p1, const QPoint & p2);

    public slots:
        /**
         * Sets new source image
         * @param sourceImage the new source image
         * @pre sourceimage is not NULL
         * @post new sourceImage has been set and aspectRatio has been updated
         */
        virtual void setSourceImage(Mat * sourceImage);

        /**
         * Update slot customized to include convertImage before actually
         * updating
         * @post sourceImage have been converted to RGB and widget updated
         */
        virtual void update();

    signals:

        /**
         * Signal sent to transmit the point in the widget where a mouse
         * button has been pressed
         * @param p the point where any mouse button has been pressed
         * @param button the button pressed
         */
        void pressPoint(const QPoint & p, const Qt::MouseButton & button);

        /**
         * Signal sent to transmit the point in the widget where mouse cursor is
         * currently dragged to (which suppose a mouse button has been
         * previously pressed)
         * @param p the point where the mouse cursor is dragged to
         */
        void dragPoint(const QPoint & p);

        /**
         * Signal sent to transmit the point in the widget where a mouse
         * button has been released
         * @param p the point where left mouse button has been released
         * @param button the button pressed
         */
        void releasePoint(const QPoint & p, const Qt::MouseButton & button);

        /**
         * Signal sent to transmit the rectangle selection when mouse button
         * has been clicked, dragged and released
         * @param r the rectangle selection
         * @param button the button pressed during dragging
         */
        void releaseSelection(const QRect & r, const Qt::MouseButton & button);
};

#endif /* QCVMATWIDGET_H_ */
```

```
/*
 * QcvMatWidget.cpp
 *
 *  Created on: 28 fÃ©vr. 2011
 *      Author: davidroussel
 */
#include <QtDebug>
#include "QcvMatWidget.h"

/*
 * Default size when no image has been set
 */
QSize QcvMatWidget::defaultSize(640, 480);

/*
 * Default aspect ratio when image is not set yet
 */
double QcvMatWidget::defaultAspectRatio = 4.0/3.0;

/*
 * Drawing color
 */
const Scalar QcvMatWidget::drawingColor(0xFF,0xCC,0x00,0x88);

/*
 * Drawing width
 */
const int QcvMatWidget::drawingWidth(3);

/*
 * OpenCV QT Widget default constructor
 * @param parent parent widget
 * @param mouseSense mouse sensivity
 */
QcvMatWidget::QcvMatWidget(QWidget *parent,
                           MouseSense mouseSense) :
    QWidget(parent),
    sourceImage(NULL),
    aspectRatio(defaultAspectRatio),
    mousePressed(false),
    mouseSense(mouseSense)
//  count(0)
{
    setup();
}

/*
 * OpenCV QT Widget constructor
 * @param the source image
 * @param parent parent widget
 * @param mouseSense mouse sensivity
 */
QcvMatWidget::QcvMatWidget(Mat * sourceImage,
                           QWidget *parent,
                           MouseSense mouseSense) :
    QWidget(parent),
    sourceImage(sourceImage),
    aspectRatio((double)sourceImage->cols / (double)sourceImage->rows),
    mousePressed(false),
    mouseSense(mouseSense)
//  count(0)
{
    setup();
}

/*
 * OpenCV Widget destructor.
 * Releases displayImage.
 */
QcvMatWidget::~QcvMatWidget()
{
    displayImage.release();
}

/*
 * paint event reimplemented to draw content (in this case only
 * draw in display image since final rendering method is not yet available)
 * @param event the paint event
 */
void QcvMatWidget::paintEvent(QPaintEvent * event)
{
    Q_UNUSED(event);
```



```

    if (displayImage.data ≠ NULL)
    {
        // evt draw in image
        if (mousePressed)
        {
            // if MOUSE_CLICK only draws a cross
            if (mouseSense > MOUSE_NONE)
            {
                if (¬(mouseSense & MOUSE_DRAG))
                {
                    if (mouseMoved)
                    {
                        drawCross(draggedPoint);
                    }
                    else
                    {
                        drawCross(pressedPoint);
                    }
                }
                else  // else if MOUSE_DRAG starts drawing a rectangle
                {
                    drawRectangle(selectionRect);
                }
            }
        }
    }
    else
    {
        qWarning("QcvMatWidget::paintEvent : image.data is NULL");
    }
}


/*
 * Widget setup
 */
void QcvMatWidget::setup()
{
    layout = new QHBoxLayout();
    layout→setContentsMargins(0,0,0,0);
    setLayout(layout);
}

/*
 * Sets new source image
 * @param sourceImage the new source image
 */
void QcvMatWidget::setSourceImage(Mat * sourceImage)
{
    // qDebug("QcvMatWidget::setSourceImage");

    this→sourceImage = sourceImage;

    // re-setup geometry since height x width may have changed
    aspectRatio = (double)sourceImage→cols / (double)sourceImage→rows;
    // qDebug ("aspect ratio changed to %4.2f", aspectRatio);

}

/*
 * Converts BGR or Gray source image to RGB display image
 * @see #sourceImage
 * @see #displayImage
 */
void QcvMatWidget::convertImage()
{
//  qDebug("Convert image");

    int depth = sourceImage→depth();
    int channels = sourceImage→channels();

    // Converts any image type to RGB format
    switch (depth)
    {
        case CV_8U:
            switch (channels)
            {
                case 1: // gray level image
                    cvtColor(*sourceImage, displayImage,CV_GRAY2RGB);
                    break;
                case 3: // Color image (OpenCV produces BGR images)
```

```
                    cvtColor(*sourceImage, displayImage, CV_BGR2RGB);
                    break;
                default:
                    qFatal("This number of channels (%d) is not supported",
                           channels);
                    break;
            }
            break;
        default:
            qFatal("This image depth (%d) is not implemented in QOpenCVWidget",
                   depth);
            break;
    }
}

/*
 * Callback called when mouse button pressed event occurs.
 * reimplemented to send pressPoint signal when left mouse button is
 * pressed
 * @param event mouse event
 */
void QcvMatWidget::mousePressEvent(QMouseEvent *event)
{
    if (mouseSense > MOUSE_NONE)
    {
//      qDebug("mousePressEvent(%d, %d) with button %d",
//             event->pos().x(), event->pos().y(), event->button());
        mousePressed = true;
        pressedPoint = event→pos();
        pressedButton = event→button();

        if((event→button() ≡ Qt::LeftButton) ∧ (mouseSense & MOUSE_DRAG))
        {
            // initialise selection rect
            selectionRect.setTopLeft(pressedPoint);
            selectionRect.setBottomRight(pressedPoint);
        }

        emit pressPoint(pressedPoint, pressedButton);
    }
}

/*
 * Callback called when mouse move event occurs.
 * reimplemented to send dragPoint signal when mouse is dragged
 * (after left mouse button has been pressed)
 * @param event mouse event
 */
void QcvMatWidget::mouseMoveEvent(QMouseEvent *event)
{
    mouseMoved = true;
    draggedPoint = event→pos();

    if ((mouseSense & MOUSE_DRAG) ∧ mousePressed)
    {
//      qDebug("mouseMoveEvent(%d, %d) with button %d",
//             event->pos().x(), event->pos().y(), event->button());

        selectionRectFromPoints(pressedPoint, draggedPoint);

        emit dragPoint(draggedPoint);
    }
}

/*
 * Callback called when mouse button released event occurs.
 * reimplemented to send releasePoint signal when left mouse button is
 * released
 * @param event mouse event
 */
void QcvMatWidget::mouseReleaseEvent(QMouseEvent *event)
{
    if ((mouseSense > MOUSE_NONE) ∧ mousePressed)
    {
//      qDebug("mouseReleaseEvent(%d, %d) with button %d",
//             event->pos().x(), event->pos().y(), event->button());
        mousePressed = false;
        mouseMoved = false;
        releasedPoint = event→pos();
        emit releasePoint(releasedPoint, pressedButton);

        if ((event→button() ≡ Qt::LeftButton) ∧ (mouseSense & MOUSE_DRAG))
```



```
        {
            selectionRectFromPoints(pressedPoint, releasedPoint);
            emit releaseSelection(selectionRect, event→button());
        }
    }
}

/*
 * Draw Cross
 * @param p the cross center
 */
void QcvMatWidget::drawCross(const QPoint & p)
{
    int x0 = p.x();
    int y0 = p.y();
    int x1, x2, x3, x4;
    int y1, y2, y3, y4;
    int offset = 10;

    x1 = x0 - 2*offset;
    x2 = x0 - offset;
    x3 = x0 + offset;
    x4 = x0 + 2*offset;
    y1 = y0 - 2*offset;
    y2 = y0 - offset;
    y3 = y0 + offset;
    y4 = y0 + 2*offset;

    Point p1a(x1, y0);
    Point p1b(x2, y0);
    Point p2a(x3, y0);
    Point p2b(x4, y0);
    Point p3a(x0, y1);
    Point p3b(x0, y2);
    Point p4a(x0, y3);
    Point p4b(x0, y4);

    line(displayImage, p1a, p1b, drawingColor, drawingWidth, CV_AA);
    line(displayImage, p2a, p2b, drawingColor, drawingWidth, CV_AA);
    line(displayImage, p3a, p3b, drawingColor, drawingWidth, CV_AA);
    line(displayImage, p4a, p4b, drawingColor, drawingWidth, CV_AA);
}

/*
 * Draw rectangle
 * @param r the rectangle to draw
 */
void QcvMatWidget::drawRectangle(const QRect & r)
{
    int x1 = r.left();
    int x2 = r.right();
    int y1 = r.top();
    int y2 = r.bottom();

    Point p1(x1, y1);
    Point p2(x2, y2);

    rectangle(displayImage, p1, p2, drawingColor, drawingWidth, CV_AA);
}

/**
 * Modifiy selectionRect using two points
 * @param p1 first point
 * @param p2 second point
 */
void QcvMatWidget::selectionRectFromPoints(const QPoint & p1, const QPoint & p2)
{
    int left, right, top, bottom;
    if (p1.x() < p2.x())
    {
        left = p1.x();
        right = p2.x();
    }
    else
    {
        left = p2.x();
        right = p1.x();
    }

    if (p1.y() < p2.y())
    {
        top = p1.y();
```

```
329             bottom = p2.y();
330         }
331         else
332         {
333             top = p2.y();
334             bottom = p1.y();
335         }
336 
337         selectionRect.setLeft(left);
338         selectionRect.setRight(right);
339         selectionRect.setTop(top);
340         selectionRect.setBottom(bottom);
341     }
342 
343 
344 
345     /*
346      * Widget minimum size is set to the contained image size
347      * @return le size of the image within
348      */
349     //QSize QcvMatWidget::minimumSize() const
350     //{
351     //  return sizeHint();
352     //}
353 
354 
355     /*
356      * Size hint (because size depends on sourceImage properties)
357      * @return size obtained from sourceImage
358      */
359     QSize QcvMatWidget::sizeHint() const
360     {
361         if (sourceImage ≠ NULL)
362         {
363             return QSize(sourceImage→cols, sourceImage→rows);
364         }
365         else
366         {
367             return defaultSize;
368         }
369     }
370 
371     /*
372      * Gets Mat widget mouse clickable status
373      * @return true if widget is sensitive to mouse click
374      */
375     bool QcvMatWidget::isMouseClickable() const
376     {
377         return (mouseSense & MOUSE_CLICK);
378     }
379 
380     /*
381      * Gets Mat widget mouse dragable status
382      * @return true if widget is sensitive to mouse drag
383      */
384     bool QcvMatWidget::isMouseDragable() const
385     {
386         return (mouseSense & MOUSE_DRAG);
387     }
388 
389     /*
390      * Update slot customized to include convertImage before actually
391      * updating
392      */
393     void QcvMatWidget::update()
394     {
395     //  count++;
396     //  qDebug() << "QcvMatWidget::update " << count;
397     //  std::cerr << "{o";
398         convertImage();
399         QWidget::update();
400     //  std::cerr << "}";
401     }
402 
403     // ----------------------------------------------------------------------
404     // convertImage old algorithm
405     // ----------------------------------------------------------------------
406     //  int cvIndex, cvLineStart;
407     //  // switch between bit depths
408     //  switch (displayImage.depth())
409     //  {
410     //      case CV_8U:
```



```
411     //          switch (displayImage.channels())
412     //          {
413     //              case 1: // Gray level images
414     //                  if ( (displayImage.cols != image.width()) ||
415     //                       (displayImage.rows != image.height()) )
416     //                  {
417     //                      QImage temp(displayImage.cols, displayImage.rows,
418     //                                  QImage::Format_RGB32);
419     //                      image = temp;
420     //                  }
421     //                  cvIndex = 0;
422     //                  cvLineStart = 0;
423     //                  for (int y = 0; y < displayImage.rows; y++)
424     //                  {
425     //                      unsigned char red, green, blue;
426     //                      cvIndex = cvLineStart;
427     //                      for (int x = 0; x < displayImage.cols; x++)
428     //                      {
429     //                          // DO it
430     //                          red   = displayImage.data[cvIndex];
431     //                          green = displayImage.data[cvIndex];
432     //                          blue  = displayImage.data[cvIndex];
433     //
434     //                          image.setPixel(x, y, qRgb(red, green, blue));
435     //                          cvIndex++;
436     //                      }
437     //                      cvLineStart += displayImage.step;
438     //                  }
439     //                  break;
440     //              case 3: // BGR images (Regular OpenCV Color Capture)
441     //                  if ( (displayImage.cols != image.width()) ||
442     //                       (displayImage.rows != image.height()) )
443     //                  {
444     //                      QImage temp(displayImage.cols, displayImage.rows,
445     //                                  QImage::Format_RGB32);
446     //                      image = temp;
447     //                  }
448     //                  cvIndex = 0;
449     //                  cvLineStart = 0;
450     //                  for (int y = 0; y < displayImage.rows; y++)
451     //                  {
452     //                      unsigned char red, green, blue;
453     //                      cvIndex = cvLineStart;
454     //                      for (int x = 0; x < displayImage.cols; x++)
455     //                      {
456     //                          // DO it
457     //                          red   = displayImage.data[cvIndex + 2];
458     //                          green = displayImage.data[cvIndex + 1];
459     //                          blue  = displayImage.data[cvIndex + 0];
460     //
461     //                          image.setPixel(x, y, qRgb(red, green, blue));
462     //                          cvIndex += 3;
463     //                      }
464     //                      cvLineStart += displayImage.step;
465     //                  }
466     //                  break;
467     //              default:
468     //                  printf("This number of channels is not supported\n");
469     //                  break;
470     //          }
471     //          break;
472     //      default:
473     //          printf("This type of Image is not implemented in QcvMatWidget\n");
474     //          break;
475     //  }
476 
```

04 nov 12 3:07 **QcvMatWidgetLabel.hpp** Page 1/1

```
1
2   #ifndef QCVMATWIDGETLABEL_H
3   #define QCVMATWIDGETLABEL_H
4
5   #include <cv.h>
6   #include <QLabel>
7
8   using namespace cv;
9
10  #include "QcvMatWidget.h"
11
12  /**
13   * OpenCV Widget for QT with QImage display
14   */
15  class QcvMatWidgetLabel : public QcvMatWidget
16  {
17      private:
18          /**
19           * The Image Label
20           */
21          QLabel * imageLabel;
22
23      public:
24          /**
25           * OpenCV QT Widget default constructor
26           * @param parent parent widget
27           * @param mouseSense mouse sensivity
28           */
29          QcvMatWidgetLabel(QWidget *parent = NULL,
30                            MouseSense mouseSense = MOUSE_NONE);
31
32          /**
33           * OpenCV QT Widget constructor
34           * @param sourceImage the source OpenCV qImage
35           * @param parent parent widget
36           * @param mouseSense mouse sensivity
37           */
38          QcvMatWidgetLabel(Mat * sourceImage,
39                            QWidget *parent = NULL,
40                            MouseSense mouseSense = MOUSE_NONE);
41
42          /**
43           * OpenCV Widget destructor.
44           */
45          virtual ~QcvMatWidgetLabel(void);
46
47      protected:
48          /**
49           * Widget setup
50           * @pre imageLabel has been allocated
51           * @post imageLabel has been added to the layout
52           */
53          void setup();
54
55          /**
56           * paint event reimplemented to draw content
57           * @param event the paint event
58           * @pre imageLabel has been allocated
59           * @post displayImage has been set as pixmap of the imageLabel
60           */
61          void paintEvent(QPaintEvent * event);
62
63  };
64
65  #endif //QCVMATWIDGETLABEL_H
```

09 mar 15 19:05 **QcvMatWidgetLabel.cpp** Page 1/1

```
1   //#include <iostream>
2   #include <QtDebug>
3   #include "QcvMatWidgetLabel.h"
4
5   using namespace std;
6
7   /*
8    * OpenCV QT Widget default constructor
9    * @param parent parent widget
10   */
11  QcvMatWidgetLabel::QcvMatWidgetLabel(QWidget *parent,
12                                       MouseSense mouseSense) :
13      QcvMatWidget(parent, mouseSense),
14      imageLabel(new QLabel())
15  {
16      setup();
17  }
18
19  /*
20   * OpenCV QT Widget constructor
21   * @param the source OpenCV qImage
22   * @param parent parent widget
23   */
24  QcvMatWidgetLabel::QcvMatWidgetLabel(Mat * sourceImage,
25                                       QWidget *parent,
26                                       MouseSense mouseSense) :
27      QcvMatWidget(sourceImage, parent, mouseSense),
28      imageLabel(new QLabel())
29  {
30      setup();
31  }
32
33  /*
34   * Widget setup
35   * @pre imageLabel has been allocated
36   */
37  void QcvMatWidgetLabel::setup()
38  {
39      layout→addWidget(imageLabel,0,Qt::AlignCenter);
40  }
41
42  /*
43   * OpenCV Widget destructor.
44   */
45  QcvMatWidgetLabel::~QcvMatWidgetLabel(void)
46  {
47      delete imageLabel;
48  }
49
50  /*
51   * paint event reimplemented to draw content
52   * @param event the paint event
53   */
54  void QcvMatWidgetLabel::paintEvent(QPaintEvent * event)
55  {
56  //  qDebug("QcvMatWidgetLabel::paintEvent");
57
58      QcvMatWidget::paintEvent(event);
59
60      if (displayImage.data ≠ NULL)
61      {
62          // Builds QImage from RGB image data
63          // and sets image as Label pixmap
64          imageLabel→setPixmap(QPixmap::fromImage(QImage((uchar *) displayImage.data,
65                                                         displayImage.cols,
66                                                         displayImage.rows,
67                                                         displayImage.step,
68                                                         QImage::Format_RGB888)));
69      }
70      else
71      {
72          qWarning("QcvMatWidgetLabel::paintEvent : image.data is NULL");
73      }
74  }
```

```
/*
 * QcvMatWidgetImage.h
 *
 *  Created on: 31 janv. 2012
 *    Author: davidroussel
 */

#ifndef QCVMATWIDGETIMAGE_H_
#define QCVMATWIDGETIMAGE_H_

#include <QImage>
#include <QPainter>

#include "QcvMatWidget.h"

/**
 * OpenCV Widget for QT with a QPainter to draw image
 */
class QcvMatWidgetImage: public QcvMatWidget
{
    protected:
        /**
         * the Qimage to display in the widget with a QPainter
         */
        QImage * qImage;

//      /**
//       * Size Policy returned by
//       */
//      QSizePolicy policy;

    public:
        /**
         * Default Constructor
         * @param parent parent widget
         * @param mouseSense mouse sensivity
         */
        QcvMatWidgetImage(QWidget *parent = NULL,
                          MouseSense mouseSense = MOUSE_NONE);

        /**
         * Constructor
         * @param sourceImage source image
         * @param parent parent widget
         * @param mouseSense mouse sensivity
         */
        QcvMatWidgetImage(Mat * sourceImage,
                          QWidget *parent = NULL,
                          MouseSense mouseSense = MOUSE_NONE);

        /**
         * Destructor.
         */
        virtual ~QcvMatWidgetImage();

//      /**
//       * Minimum size hint according to aspect ratio and min height of 100
//       * @return minimum size hint
//       */
//      QSize minimumSizeHint() const;

        /**
         * aspect ratio method
         * @param w width
         * @return the required height fo r this width
         */
        int heightForWidth ( int w ) const;

//      /**
//       * Size policy to keep aspect ratio right
//       * @return
//       */
//      QSizePolicy sizePolicy () const;

        /**
         * Sets new source image
         * @param sourceImage the new source image
         */
        virtual void setSourceImage(Mat * sourceImage);

    protected:
        /**
```



```
         * Setup widget (defines size policy)
         */
        void setup();

        /**
         * paint event reimplemented to draw content
         * @param event the paint event
         */
        void paintEvent(QPaintEvent * event);

};

#endif /* QCVMATWIDGETIMAGE_H_ */
```

```cpp
/*
 * QcvMatWidgetImage.cpp
 *
 *  Created on: 31 janv. 2012
 *     Author: davidroussel
 */

#include "QcvMatWidgetImage.h"
#include <QPaintEvent>
#include <QSizePolicy>
#include <QDebug>

/*
 * Default Constructor
 * @param parent parent widget
 */
QcvMatWidgetImage::QcvMatWidgetImage(QWidget *parent,
                                     MouseSense mouseSense) :
    QcvMatWidget(parent, mouseSense),
    qImage(NULL)
{
    setup();
}

/*
 * Constructor
 * @param sourceImage source image
 * @param parent parent widget
 */
QcvMatWidgetImage::QcvMatWidgetImage(Mat * sourceImage,
                                     QWidget *parent,
                                     MouseSense mouseSense) :
    QcvMatWidget(sourceImage, parent, mouseSense),
    qImage(NULL)
{
    setSourceImage(sourceImage);

    setup();
}

/*
 * Setup widget (defines size policy)
 */
void QcvMatWidgetImage::setup()
{
//  qDebug("QcvMatWidgetImage::Setup");

    /*
     * Customize size policy
     */
    QSizePolicy qsp(QSizePolicy::Fixed, QSizePolicy::Fixed);
    // sets height depends on width (also need to reimplement heightForWidth())
    qsp.setHeightForWidth(true);
    setSizePolicy(qsp);

    /*
     * Customize layout
     */

    // size policy has changed to call updateGeometry
    updateGeometry();
}

/*
 * Destructor.
 */
QcvMatWidgetImage::~QcvMatWidgetImage()
{
    if (qImage ≠ NULL)
    {
        delete qImage;
    }
}

/*
 * Sets new source image
 * @param sourceImage the new source image
 */
void QcvMatWidgetImage::setSourceImage(Mat * sourceImage)
{
    if (qImage ≠ NULL)
    {
        delete qImage;
    }
    // setup and convert image
    QcvMatWidget::setSourceImage(sourceImage);
    convertImage();
    qImage = new QImage((uchar *) displayImage.data, displayImage.cols,
        displayImage.rows, displayImage.step,
        QImage::Format_RGB888);

    // re-setup geometry since height x width may have changed
    updateGeometry();
}

/*
 * Size policy to keep aspect ratio right
 * @return
 */
//QSizePolicy QcvMatWidgetImage::sizePolicy () const
//{
//  return policy;
//}

/*
 * aspect ratio method
 * @param w width
 * @return the required height fo r this width
 */
int QcvMatWidgetImage::heightForWidth(int w) const
{
//  qDebug ("height  = %d for width  = %d called", (int)((double)w/aspectRatio), w);
    return (int)((double)w/aspectRatio);
}

/*
 * Minimum size hint according to aspect ratio and min height of 100
 * @return minimum size hint
 */
//QSize QcvMatWidgetImage::minimumSizeHint () const
//{
//  // qDebug("min size called");
//  // return QSize((int)(100.0*aspectRatio), 100);
//  return sizeHint();
//}


/*
 * paint event reimplemented to draw content
 * @param event the paint event
 */
void QcvMatWidgetImage::paintEvent(QPaintEvent *event)
{
//  qDebug("QcvMatWidgetImage::paintEvent");

    // evt draws in image directly
    QcvMatWidget::paintEvent(event);

    if (displayImage.data ≠ NULL)
    {
        // then draw image
        QPainter painter(this);
        painter.setRenderHint(QPainter::SmoothPixmapTransform, true);
        if (event ≡ NULL)
        {
            painter.drawImage(0, 0, *qImage);
        }
        else // partial repaint
        {
            painter.drawImage(event→rect(), *qImage);
        }
    }
    else
    {
        qWarning("QcvMatWidgetImage::paintEvent : image.data is NULL");
    }
}
```

```
/*
 * QcvMatWidgetGL.h
 *
 *  Created on: 28 fÃ©vr. 2011
 *    Author: davidroussel
 */

#ifndef QOPENCVWIDGETQGL_H_
#define QOPENCVWIDGETQGL_H_

#include <QGLWidget>

#include "QcvMatWidget.h"
#include "QGLImageRender.h"

/**
 * OpenCV Widget for QT with QGLWidget display
 */
class QcvMatWidgetGL: public QcvMatWidget
{
    private:
        /**
         * QGLWidget to draw in
         */
        QGLImageRender * gl;

//      size_t glCount;

    public:

        /**
         * OpenCV QT Widget default constructor
         * @param parent parent widget
         * @param mouseSense mouse sensivity
         */
        QcvMatWidgetGL(QWidget *parent = NULL,
                       MouseSense mouseSense = MOUSE_NONE);

        /**
         * OpenCV QT Widget constructor
         * @param sourceImage the source image
         * @param parent parent widget
         * @param mouseSense mouse sensivity
         */
        QcvMatWidgetGL(Mat * sourceImage,
                       QWidget *parent = NULL,
                       MouseSense mouseSense = MOUSE_NONE);

        /**
         * Sets new source image
         * @param sourceImage the new source image
         */
        void setSourceImage(Mat * sourceImage);

        /**
         * OpenCV Widget destructor.
         */
        virtual ~QcvMatWidgetGL();

    protected:
        /*
         * paint event reimplemented to draw content
         * @param event the paint event
         */
        void paintEvent(QPaintEvent * event);
};

#endif /* QOPENCVWIDGETQGL_H_ */
```



```
/*
 * QcvMatWidgetGL.cpp
 *
 *  Created on: 28 fÃ©vr. 2011
 *    Author: davidroussel
 */
#include <QDebug>

#include "QcvMatWidgetGL.h"

/*
 * OpenCV QT Widget default constructor
 * @param parent parent widget
 */
QcvMatWidgetGL::QcvMatWidgetGL(QWidget *parent,
                               MouseSense mouseSense) :
    QcvMatWidget(parent, mouseSense),
    gl(NULL)
//  glCount(0)
{
}

/*
 * OpenCV QT Widget constructor
 * @param parent parent widget
 */
QcvMatWidgetGL::QcvMatWidgetGL(Mat * sourceImage,
                               QWidget *parent,
                               MouseSense mouseSense) :
    QcvMatWidget(sourceImage, parent, mouseSense),
    gl(NULL)
//  glCount(0)
{
    setSourceImage(sourceImage);
}

/*
 * OpenCV Widget destructor.
 */
QcvMatWidgetGL::~QcvMatWidgetGL()
{
    if (gl ≠ NULL)
    {
        layout→removeWidget(gl);
        delete gl;
    }
}

/*
 * Sets new source image
 * @param sourceImage the new source image
 */
void QcvMatWidgetGL::setSourceImage(Mat *sourceImage)
{
    QcvMatWidget::setSourceImage(sourceImage);

    if (gl ≠ NULL)
    {
        layout→removeWidget(gl);
        delete gl;
    }

    convertImage();

    gl = new QGLImageRender(displayImage, this);

    layout→addWidget(gl, 0, Qt::AlignCenter);
}

/*
 * paint event reimplemented to draw content
 * @param event the paint event
 */
void QcvMatWidgetGL::paintEvent(QPaintEvent * event)
{
    QcvMatWidget::paintEvent(event);
//  qDebug() << "Paint event # " << glCount++;
    gl→update();
}
```

09 mar 15 18:43 **QGLImageRender.hpp** Page 1/1

```
/*
 * QGLImageRender.h
 *
 *  Created on: 28 fÃ©vr. 2011
 *      Author: davidroussel
 */

#ifndef QGLIMAGERENDER_H_
#define QGLIMAGERENDER_H_

#include <QGLWidget>
#include <QSize>
#include <QSizePolicy>
#include <cv.h>

using namespace cv;

/**
 * A Class allowing to draw OpenCV Mat images using OpenGL
 */
class QGLImageRender: public QGLWidget
{
    private:
        /**
         * The RGB image to draw
         */
        Mat image;

//      size_t fCount;

    public:
        /**
         * QGLImageRender Constructor
         * @param image the RGB image to draw in the pixel buffer
         * @param parent the parent widget
         */
        QGLImageRender(const Mat & image, QWidget *parent = NULL);

        /**
         * QGLImageRender destructor
         */
        virtual ~QGLImageRender();

        /**
         * Size hint
         * @return Qsize containing size hint
         */
        QSize sizeHint () const;

        /**
         * Minimum Size hint
         * @return QSize containing the minimum size hint
         */
        QSize minimumSizeHint() const;

        /**
         * Size Policy for this widget
         * @return A No resize at all policy
         */
        QSizePolicy sizePolicy () const;

    protected :
        /**
         * Initialise GL drawing (called once on each QGLContext)
         */
        void initializeGL();
        /**
         * Paint GL : called whenever the widget needs to be painted
         */
        void paintGL();
        /**
         * Resize GL : called whenever the widget has been resized
         */
        void resizeGL(int width, int height);
};

#endif /* QGLIMAGERENDER_H_ */
```

31 mar 15 15:57 **QGLImageRender.cpp** Page 1/2

```
/*
 * QGLImageRender.cpp
 *
 *  Created on: 28 fÃ©vr. 2011
 *      Author: davidroussel
 */
#include <QDebug>
#ifdef __APPLE__
    #include <gl.h>
    #include <glu.h>
#else
    #include <GL/gl.h>
    #include <GL/glu.h>
#endif
#include "QGLImageRender.h"

QGLImageRender::QGLImageRender(const Mat & image, QWidget *parent) :
    QGLWidget(parent),
    image(image)
//  fCount(0)
{
    if (¬doubleBuffer())
    {
        qWarning("QGLImageRender::QGLImageRender caution : no double buffer");
    }
    if (this→image.data ≡ NULL)
    {
        qWarning("QGLImageRender::QGLImageRender caution : image data is null");
    }
}

QGLImageRender::~QGLImageRender()
{
    image.release();
}

void QGLImageRender::initializeGL()
{
    qDebug("GL init ...");
    glClearColor(0.0, 0.0, 0.0, 0.0);
//  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
}

void QGLImageRender::paintGL()
{
//  qDebug("GL drawing pixels ...");

    glClear(GL_COLOR_BUFFER_BIT);

    if (image.data ≠ NULL)
    {
        glDrawPixels(image.cols, image.rows, GL_RGB,
                     GL_UNSIGNED_BYTE, image.data);
        // In any circumstance you should NOT use glFlush or swapBuffers() here
    }
    else
    {
        qWarning("Nothing to draw");
    }

}

void QGLImageRender::resizeGL(int width, int height)
{
//  qDebug("GL resizeGL ...");
//  glViewport(0, 0, width, height);
//  glMatrixMode(GL_PROJECTION);
//  glLoadIdentity();
//  gluOrtho2D(0.0, 0.0, (GLdouble)width, (GLdouble)height);

    qDebug("GL Resize (%d, %d)",width, height);

//  GLfloat zoom, xZoom, yZoom;
//
//  xZoom = (GLfloat)width/(GLfloat)image.cols;
//  yZoom = (GLfloat)height/(GLfloat)image.rows;
//
//  if (xZoom < yZoom)
//  {
//      zoom = xZoom;
//  }
//  else
```

```
//  {
//      zoom = yZoom;
//  }

    glViewport(0, 0, (GLsizei) width, (GLsizei) height);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (image.data ≠ NULL)
    {
//      gluOrtho2D(0, (GLdouble) image.cols, 0, (GLdouble) image.rows);
        glOrtho(0, (GLdouble) image.cols, 0, (GLdouble) image.rows, 1.0, -1.0);
    }

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

    /* apply the right translate so the image drawing starts top left */
    if (image.data ≠ NULL)
    {
        /*
         * For some reason we should not start drawing exactly at the limit
         * of the drawing plane so we start drawing at image.rows - something
         * which could be very tiny
         */
        glRasterPos2i(0,image.rows);
    }
    else
    {
        qWarning("QGLImageRender::resizeGL(...) : image.data is NULL");
    }

    /* apply the right zoom factor so image are displayed top 2 bottom */
    glPixelZoom(1.0, -1.0);
}


QSize QGLImageRender::sizeHint () const
{
    return minimumSizeHint();
}

QSize QGLImageRender::minimumSizeHint() const
{
    if (image.data ≠ NULL)
    {
        return QSize(image.cols, image.rows);
    }
    else
    {
        qWarning("QGLImageRender::minimumSizeHint : probably invalid sizeHint");
        return QSize(320,240);
    }
}

QSizePolicy QGLImageRender::sizePolicy () const
{
    return QSizePolicy(QSizePolicy::Fixed, QSizePolicy::Fixed);
}
```



```
/*
 * QcvVideoCapture.h
 *
 *  Created on: 29 janv. 2012
 *      Author: davidroussel
 */

#ifndef QCVVIDEOCAPTURE_H_
#define QCVVIDEOCAPTURE_H_

#include <QObject>
#include <QSize>
#include <QTimer>
#include <QThread>
#include <QMutex>

#include <opencv2/highgui/highgui.hpp>
using namespace cv;

/**
 * Qt Class for capturing videos from cameras of files with OpenCV.
 * QcvVideoCapture opens streams and refresh itself automatically.
 * When frame has been refreshed a signal is emitted.
 */
class QcvVideoCapture: public QObject
{
    Q_OBJECT

    protected:

        /**
         * file name used to open video file.
         * Used to reopen video file when video is finished.
         */
        QString filename;

        /**
         * Video capture instance
         * @warning capture is regularly updated by a timer, but can also be
         * manipulated by other methods (such as #setDirectSize). So capture
         * access for new images should be protected by a mutex to ensure
         * atomic access to capture object at a time.
         */
        VideoCapture capture;

        /**
         * refresh timer
         */
        QTimer * timer;

        /**
         * Independant thread to update capture.
         * If independant thread is required, then update method is called
         * from within this thread. Otherwise, update method is called from
         * main thread.
         */
        QThread * updateThread;

        /**
         * Mutex lock to ensure atomic access capture grabbing new image.
         * @warning if QcvVideoCapture object is not updated in the
         * #updateThread, then trying to lock mutex multiple times with
         * mutex.lock() will lead to a deadlock, so if this object has no
         * #updateThread (if #updateThread == NULL) we should use
         * mutex.tryLock() instead and give up when lock can't be obtained with
         * tryLock(). For instance when tryLock into #update method fails, this
         * means that capture object is locked in some other method, so we don't
         * grab any new image this time and hope, we'll be able to do it next
         * time #update will be called.
         */
        QMutex mutex;

        /**
         * Mutex lock state memory to avoid locking the mutex multiple times
         * across multiple methods. When a mutex.lock() is performed locked
         * should be set to true until mutex.unlock(). Hence, if a method
         * requiring lock is performed, a second lock is avoided by checking
         * this attribute.
         */
        size_t lockLevel;

        /**
```

```
83          * Image Matrix to obtain from capture
84          */
85         Mat image;
86
87         /**
88          * image resized (if required)
89          */
90         Mat imageResized;
91
92         /**
93          * [resized] image flipped (if required)
94          */
95         Mat imageFlipped;
96
97         /**
98          * Image converted for display:
99          *  - scaled
100         *  - flipped horizontally
101         *  - converted to gray
102         */
103        Mat imageDisplay;
104
105        /**
106         * Live video indication (from cam)
107         */
108        bool liveVideo;
109
110        /**
111         * flipVideo to mirror image
112         */
113        bool flipVideo;
114
115        /**
116         * scale image to preferred width and height
117         */
118        bool resize;
119
120        /**
121         * scaling is performed into capture rather than through cv::resize
122         * function
123         */
124        bool directResize;
125
126        /**
127         * image converted to gray
128         */
129        bool gray;
130
131        /**
132         * Allow capture to skip an image capture when lock can't be acquired
133         * before grabbing a new image. Otherwise we'll wait until the lock
134         * is acquired before grabbing an new image. The lock might be acquired
135         * by another lenghty thread/processor during image processing.
136         */
137        bool skip;
138
139        /**
140         * Current Image size (might be different from natural capture image
141         * size)
142         */
143        QSize size;
144
145        /**
146         * Capture natural image size (without resizing)
147         */
148        QSize originalSize;
149
150        /**
151         * Capture frame rate obtained either by getting the CV_CAP_PROP_FPS
152         * VideoCapture property or by computing capture time on several images
153         * @see #grabInterval
154         */
155        double frameRate;
156
157        /**
158         * default time interval between refresh
159         */
160        static int defaultFrameDelay;
161
162        /**
163         * Number of frames to test frame rate
164         */
```



```
165        static size_t defaultFrameNumberTest;
166
167        /**
168         * Status message to send when something changes
169         */
170        QString statusMessage;
171
172        /**
173         * Default message showing time (at least 2000 ms)
174         */
175        static int messageDelay;
176
177    public:
178        /**
179         * QcvVideoCapture constructor.
180         * Opens the default camera (0)
181         * @param flipVideo mirror image status
182         * @param gray convert image to gray status
183         * @param skip indicates capture can skip an image. When the capture
184         * result has not been processed yet, or when false that capture should
185         * wait for the result to be processed before grabbing a new image.
186         * This only applies when #updateThread is not NULL.
187         * @param width desired width or 0 to keep capture width
188         * @param height desired height or 0 to keep capture height
189         * otherwise capture is updated in the current thread.
190         * @param updateThread the thread used to run this capture
191         * @param parent the parent QObject
192         */
193        QcvVideoCapture(const bool flipVideo = false,
194                        const bool gray = false,
195                        const bool skip = true,
196                        const unsigned int  width = 0,
197                        const unsigned int height = 0,
198                        QThread * updateThread = NULL,
199                        QObject * parent = NULL);
200
201        /**
202         * QcvVideoCapture constructor with device Id
203         * @param deviceId the id of the camera to open
204         * @param flipVideo mirror image
205         * @param gray convert image to gray
206         * @param skip indicates capture can skip an image. When the capture
207         * result has not been processed yet, or when false that capture should
208         * wait for the result to be processed before grabbing a new image.
209         * This only applies when #updateThread is not NULL.
210         * @param width desired width or 0 to keep capture width
211         * @param height desired height or 0 to keep capture height
212         * @param updateThread the thread used to run this capture
213         * @param parent the parent QObject
214         */
215        QcvVideoCapture(const int deviceId,
216                        const bool flipVideo = false,
217                        const bool gray = false,
218                        const bool skip = true,
219                        const unsigned int  width = 0,
220                        const unsigned int height = 0,
221                        QThread * updateThread = NULL,
222                        QObject * parent = NULL);
223
224        /**
225         * QcvVideoCapture constructor from file name
226         * @param fileName video file to open
227         * @param flipVideo mirror image
228         * @param gray convert image to gray
229         * @param skip indicates capture can skip an image. When the capture
230         * result has not been processed yet, or when false that capture should
231         * wait for the result to be processed before grabbing a new image.
232         * This only applies when #updateThread is not NULL.
233         * @param width desired width or 0 to keep capture width
234         * @param height desired height or 0 to keep capture height
235         * @param updateThread the thread used to run this capture
236         * @param parent the parent QObject
237         */
238        QcvVideoCapture(const QString & fileName,
239                        const bool flipVideo = false,
240                        const bool gray = false,
241                        const bool skip = true,
242                        const unsigned int  width = 0,
243                        const unsigned int height = 0,
244                        QThread * updateThread = NULL,
245                        QObject * parent = NULL);
246
```

```
247         /**
248          * QcvVideoCapture destructor.
249          * releases video capture and image
250          */
251         virtual ~QcvVideoCapture();
252 
253         /**
254          * Size accessor
255          * @return the image size
256          */
257         const QSize & getSize() const;
258 
259         /**
260          * Gets resize state.
261          * @return true if imageDisplay have been resized to preferred width and
262          * height, false otherwise
263          */
264         bool isResized() const;
265 
266         /**
267          * Gets direct resize state.
268          * @return true if image can be resized directly into capture.
269          * @note direct resize capabilities are tested into #grabTest which is
270          * called in all constructors. So #isDirectResizeable should not be
271          * called before #grabTest
272          */
273         bool isDirectResizeable() const;
274 
275         /**
276          * Gets video flipping status
277          * @return flipped video status
278          */
279         bool isFlipVideo() const;
280 
281         /**
282          * Gets video gray converted status
283          * @return the converted to gray status
284          */
285         bool isGray() const;
286 
287         /**
288          * Gets the image skipping policy
289          * @return true if new image can be skipped when previous one has not
290          * been processed yet, false otherwise.
291          */
292         bool isSkippable() const;
293 
294         /**
295          * Gets the current frame rate
296          * @return the current frame rate
297          */
298         double getFrameRate() const;
299 
300         /**
301          * Image accessor
302          * @return the image to display
303          */
304         Mat * getImage();
305 
306         /**
307          * The source image mutex
308          * @return  the mutex used on image access
309          */
310         QMutex * getMutex();
311 
312     public slots:
313         /**
314          * Open new device Id
315          * @param deviceId device number to open
316          * @param width desired width or 0 to keep capture width
317          * @param height desired height or 0 to keep capture height
318          * @return true if device has been opened and checked and timer launched
319          */
320         bool open(const int deviceId,
321                   const unsigned int width = 0,
322                   const unsigned int height = 0);
323 
324         /**
325          * Open new video file
326          * @param fileName video file to open
327          * @param width desired width or 0 to keep capture width
328          * @param height desired height or 0 to keep capture height
```



```
329          * @return true if video has been opened and timer launched
330          */
331         bool open(const QString & fileName,
332                   const unsigned int width = 0,
333                   const unsigned int height = 0);
334         /**
335          * Sets video flipping
336          * @param flipVideo flipped video or not
337          */
338         void setFlipVideo(const bool flipVideo);
339 
340         /**
341          * Sets video conversion to gray
342          * @param grayConversion the gray conversion status
343          */
344         void setGray(const bool grayConversion);
345 
346         /**
347          * Sets #imageDisplay size according to preferred width and height
348          * @param size new desired size to set
349          * @param alreadyLocked mutex lock has already been aquired so setSize does not have
350          * to acquire the lock
351          * @pre a first image have been grabbed
352          */
353         void setSize(const QSize & size);
354 
355     protected:
356         /**
357          * Performs a grab test to fill #image.
358          * if capture is opened then tries to grab and if grab succeeds then
359          * tries to retrieve image from grab and sets image size.
360          * @return true if capture is opened and successfully grabbed a first
361          * frame into #image, false otherwise
362          * @post Moreover this method determines if direct resizing is allowed
363          * on this capture instance by trying to set
364          * CV_CAP_PROP_FRAME_WIDTH and CV_CAP_PROP_FRAME_HEIGHT.
365          */
366         bool grabTest();
367 
368         /**
369          * Get or compute interval between two frames in ms and sets the
370          * frameRate attribute.
371          * Tries to get CV_CAP_PROP_FPS from capture and if not available
372          * computes times between frames by grabbing defaultNumberTest images
373          * @return interval between two frames
374          * @param message message passed to grabInterval and display ahead of
375          * the framerate computed during grabInterval
376          * @pre capture is already instanciated
377          * @post message indicating frame rate has been emitted and interval
378          * between two frames has been returned
379          */
380         int grabInterval(const QString & message);
381 
382         /**
383          * Sets #imageDisplay size according to preferred width and height
384          * @param width desired width
385          * @param height desired height
386          * @pre a first image have been grabbed
387          */
388         void setSize(const unsigned int width,
389                      const unsigned int height);
390 
391         /**
392          * Tries to set capture size directly on capture by setting properties.
393          *  - CV_CAP_PROP_FRAME_WIDTH to set frame width
394          *  - CV_CAP_PROP_FRAME_HEIGHT to set frame height
395          * @param width the width property to set on capture
396          * @param height the height property to set on capture
397          * @return true if capture is opened and if width and height have been
398          * set successfully through @code capture.set(...) @endcode. Returns
399          * false otherwise.
400          * @post if at least width or height have been set successfully, capture
401          * image is released then updated again so it will have the right
402          * dimensions.
403          * @warning if mutex lock can't be obtained to ensure atomic access to
404          * capture object, then we start recursing until we obtain that lock,
405          * which is gross and should be fixed !!!
406          */
407         bool setDirectSize(const unsigned int width, const unsigned int height);
408 
409     protected slots:
410         /**
```

```
411          * update slot trigerred by timer : Grabs a new image and sends updated()
412          * signal iff new image has been grabbed, otherwise there is no more
413          * images to grab so kills timer.
414          * @note If lock on OpenCV capture object can not be obtained then
415          * capture is skipped. This is not critical since update is called
416          * regularly by the #timer, so we'll try updating image next time.
417          */
418         void update();
419
420     signals:
421         /**
422          * Signal emitted when a new image has been grabbed
423          */
424         void updated();
425
426         /**
427          * Signal emitted when capture is released
428          */
429         void finished();
430
431         /**
432          * Signal to send update message when something changes
433          * @param message the message
434          * @param timeout number of ms the message should be displayed
435          */
436         void messageChanged(const QString & message, int timeout = 0);
437
438         /**
439          * Signal to send when image has changed after opening new device or
440          * setting new display size
441          * @param image the new image to send
442          */
443         void imageChanged(Mat * image);
444 };
445
446 #endif /* QCVVIDEOCAPTURE_H_ */
447
```



```
1  /*
2   * QcvVideoCapture.cpp
3   *
4   *  Created on: 29 janv. 2012
5   *      Author: davidroussel
6   */
7
8  #include <QElapsedTimer>
9  #include <QMutexLocker>
10 #include <QDebug>
11
12 #include "QcvVideoCapture.h"
13
14 #include <opencv2/imgproc/imgproc.hpp>
15
16 /*
17  * default time interval between refresh
18  */
19 int QcvVideoCapture::defaultFrameDelay = 33;
20
21 /*
22  * Number of frames to test frame rate
23  */
24 size_t QcvVideoCapture::defaultFrameNumberTest = 5;
25
26 /*
27  * Default message showing time (at least 2000 ms)
28  */
29 int QcvVideoCapture::messageDelay = 5000;
30
31 /*
32  * QcvVideoCapture constructor.
33  * Opens the default camera (0)
34  * @param flipVideo mirror image status
35  * @param gray convert image to gray status
36  * @param skip indicates capture can skip an image. When the capture
37  * result has not been processed yet, or when false that capture should
38  * wait for the result to be processed before grabbing a new image.
39  * This only applies when #updateThread is not NULL.
40  * @param width desired width or 0 to keep capture width
41  * @param height desired height or 0 to keep capture height
42  * otherwise capture is updated in the current thread.
43  * @param updateThread the thread used to run this capture
44  * @param parent the parent QObject
45  */
46 QcvVideoCapture::QcvVideoCapture(const bool flipVideo,
47                                  const bool gray,
48                                  const bool skip,
49                                  const unsigned int width,
50                                  const unsigned int height,
51                                  QThread * updateThread,
52                                  QObject * parent) :
53     QcvVideoCapture(0, flipVideo, gray, skip, width, height, updateThread,
54                     parent)
55 {
56 }
57
58 /*
59  * QcvVideoCapture constructor with device Id
60  * @param deviceId the id of the camera to open
61  * @param flipVideo mirror image
62  * @param gray convert image to gray
63  * @param skip indicates capture can skip an image. When the capture
64  * result has not been processed yet, or when false that capture should
65  * wait for the result to be processed before grabbing a new image.
66  * This only applies when #updateThread is not NULL.
67  * @param width desired width or 0 to keep capture width
68  * @param height desired height or 0 to keep capture height
69  * @param updateThread the thread used to run this capture
70  * @param parent the parent QObject
71  */
72 QcvVideoCapture::QcvVideoCapture(const int deviceId,
73                                  const bool flipVideo,
74                                  const bool gray,
75                                  const bool skip,
76                                  const unsigned int width,
77                                  const unsigned int height,
78                                  QThread * updateThread,
79                                  QObject * parent) :
80     QObject(parent),
81     filename(),
82     capture(deviceId),
```

```
    timer(new QTimer(updateThread ≡ NULL ? this : NULL)),
    updateThread(updateThread),
    mutex(QMutex::NonRecursive),
    lockLevel(0),
    liveVideo(true),
    flipVideo(flipVideo),
    resize(false),
    directResize(false),
    gray(gray),
    skip(skip),
    size(0, 0),
    originalSize(0, 0),
    frameRate(0.0),
    statusMessage()
{
    if (updateThread ≠ NULL)
    {
        moveToThread(this→updateThread);
        connect(this, SIGNAL(finished()), updateThread, SLOT(quit()),
                Qt::DirectConnection);
    }

    timer→setSingleShot(false);
    connect(timer, SIGNAL(timeout()), SLOT(update()));

    if (grabTest())
    {
        setSize(width, height);
        QString message("Camera ");
        message.append(QString::number(deviceId));
        message.append(" ");
        int delay = grabInterval(message);
        if (updateThread ≠ NULL)
        {
            updateThread→start();
        }
        timer→start(delay);
        qDebug("timer started with %d ms delay", delay);
    }
    else
    {
        qDebug() << "QcvVideoCapture::QcvVideoCapture(" << deviceId
                 << "): grab test failed";
    }
}

/*
 * QcvVideoCapture constructor from file name
 * @param fileName video file to open
 * @param flipVideo mirror image
 * @param gray convert image to gray
 * @param skip indicates capture can skip an image. When the capture
 * result has not been processed yet, or when false that capture should
 * wait for the result to be processed before grabbing a new image.
 * This only applies when #updateThread is not NULL.
 * @param width desired width or 0 to keep capture width
 * @param height desired height or 0 to keep capture height
 * @param updateThread the thread used to run this capture
 * @param parent the parent QObject
 */
QcvVideoCapture::QcvVideoCapture(const QString & fileName,
                                 const bool flipVideo,
                                 const bool gray,
                                 const bool skip,
                                 const unsigned int width,
                                 const unsigned int height,
                                 QThread * updateThread,
                                 QObject * parent) :
    QObject(parent),
    filename(fileName),
    capture(fileName.toStdString()),
    timer(new QTimer(updateThread ≡ NULL ? this : NULL)),
    updateThread(updateThread),
    mutex(QMutex::NonRecursive),
    lockLevel(0),
    liveVideo(false),
    flipVideo(flipVideo),
    resize(false),
    directResize(false),
    gray(gray),
    skip(skip),
    size(0, 0),
```



```
    originalSize(0, 0),
    frameRate(0.0),
    statusMessage()
{
    if (updateThread ≠ NULL)
    {
        moveToThread(this→updateThread);
        connect(this, SIGNAL(finished()), updateThread, SLOT(quit()),
                Qt::DirectConnection);
    }

    timer→setSingleShot(false);
    connect(timer, SIGNAL(timeout()), SLOT(update()));

    if (grabTest())
    {
        setSize(width, height);
        QString message("File ");
        message.append(fileName);
        message.append(" ");

        int delay = grabInterval(message);
        if (updateThread ≠ NULL)
        {
            updateThread→start();
        }
        timer→start(delay);
        qDebug("timer started with %d ms delay", delay);
    }
}

/*
 * QcvVideoCapture destructor.
 * releases video capture and image
 */
QcvVideoCapture::~QcvVideoCapture()
{
    // wait for the end of an update
    if (updateThread ≠ NULL)
    {
        if (lockLevel ≡ 0)
        {
            mutex.lock();
            // qDebug() << "QcvVideoCapture::~QcvVideoCapture: lock";
        }
        lockLevel++;
    }

    if (timer ≠ NULL)
    {
        if (timer→isActive())
        {
            timer→stop();
            qDebug("timer stopped");
        }

        timer→disconnect(SIGNAL(timeout()), this, SLOT(update()));
    }

    if (updateThread ≠ NULL)
    {
        lockLevel--;
        if (lockLevel ≡ 0)
        {
            // qDebug() << "QcvVideoCapture::~QcvVideoCapture: unlock";
            mutex.unlock();
        }

        emit finished();

        // Wait until the updateThread receives the "finished" signal through
        // "quit" slot
        updateThread→wait();

        delete timer; // delete unparented timer
    }

    // release OpenCV ressources
    filename.clear();
    capture.release();
    imageDisplay.release();
    imageFlipped.release();
```

```
247     imageResized.release();
248     image.release();
249 }
250
251 /*
252  * Open new device Id
253  * @param deviceId device number to open
254  * @param width desired width or 0 to keep capture width
255  * @param height desired height or 0 to keep capture height
256  * @return true if device has been opened and checked and timer launched
257  */
258 bool QcvVideoCapture::open(const int deviceId,
259                            const unsigned int width,
260                            const unsigned int height)
261 {
262     if (updateThread ≠ NULL)
263     {
264         if (lockLevel ≡ 0)
265         {
266             mutex.lock();
267             // qDebug() << "QcvVideoCapture::open(" << deviceId << "...): lock";
268         }
269         lockLevel++;
270     }
271
272     filename.clear();
273     if (timer→isActive())
274     {
275         timer→stop();
276         qDebug("timer stopped");
277     }
278
279     if (capture.isOpened())
280     {
281         capture.release();
282     }
283
284     if (¬image.empty())
285     {
286         image.release();
287     }
288
289     capture.open(deviceId);
290
291     bool grabbed = grabTest();
292
293     if (grabbed)
294     {
295         setSize(width, height);
296
297         statusMessage.clear();
298         statusMessage.append("Camera ");
299         statusMessage.append(QString::number(deviceId));
300         statusMessage.append(" ");
301         int delay = grabInterval(statusMessage);
302         timer→start(delay);
303         liveVideo = true;
304         qDebug("timer started with %d ms delay", delay);
305
306         // emit
307         // message changed already emitted by grabInterval()
308         emit imageChanged(&imageDisplay);
309
310     }
311     if (updateThread ≠ NULL)
312     {
313         lockLevel−−;
314         if (lockLevel ≡ 0)
315         {
316             // qDebug() << "QcvVideoCapture::open(" << deviceId << "...): unlock";
317             mutex.unlock();
318         }
319     }
320
321     return grabbed;
322 }
323
324 /*
325  * Open new video file
326  * @param fileName video file to open
327  * @param width desired width or 0 to keep capture width
328  * @param height desired height or 0 to keep capture height
```



```
329  * @return true if video has been opened and timer launched
330  */
331 bool QcvVideoCapture::open(const QString & fileName,
332                            const unsigned int width,
333                            const unsigned int height)
334 {
335     filename = fileName;
336
337     if (timer→isActive())
338     {
339         timer→stop();
340         qDebug("timer stopped");
341     }
342
343     if (updateThread ≠ NULL)
344     {
345         if (lockLevel ≡ 0)
346         {
347             mutex.lock();
348             // qDebug() << "QcvVideoCapture::open(" << fileName << "...): lock";
349         }
350         lockLevel++;
351     }
352
353     if (capture.isOpened())
354     {
355         capture.release();
356     }
357
358     if (¬image.empty())
359     {
360         image.release();
361     }
362
363     capture.open(fileName.toStdString());
364
365     bool grabbed = grabTest();
366
367     if (grabbed)
368     {
369         setSize(width, height);
370         qDebug() << "open setSize done";
371         statusMessage.clear();
372         statusMessage.append("file ");
373         statusMessage.append(fileName);
374         statusMessage.append(" opened");
375
376         int delay = grabInterval(statusMessage);
377         timer→start(delay);
378         liveVideo = false;
379         qDebug("timer started with %d ms delay", delay);
380
381         // emit changes
382         // messageChanged already emitted by grabInterval
383         emit imageChanged(&imageDisplay);
384
385     }
386
387     if (updateThread ≠ NULL)
388     {
389         lockLevel−−;
390         if(lockLevel ≡ 0)
391         {
392             // qDebug() << "QcvVideoCapture::open(" << filename << "...): unlock";
393             mutex.unlock();
394         }
395     }
396
397     return grabbed;
398 }
399
400 /*
401  * Size accessor
402  * @return the image size
403  */
404 const QSize & QcvVideoCapture::getSize() const
405 {
406     return size;
407 }
408
409 /*
410  * Sets #imageDisplay size according to preferred width and height
```

```
411  * @param width desired width
412  * @param height desired height
413  * @pre a first image have been grabbed
414  */
415 void QcvVideoCapture::setSize(const unsigned int width,
416                               const unsigned int height)
417 {
418     if ((updateThread ≠ NULL))
419     {
420         if (lockLevel ≡ 0)
421         {
422             mutex.lock();
423             // qDebug("QcvVideoCapture::setSize(%d, %d) locked", width, height);
424         }
425         lockLevel++;
426     }
427
428     unsigned int preferredWidth;
429     unsigned int preferredHeight;
430
431     // qDebug("QcvVideoCapture::setSize(%d, %d)", width, height);
432
433     // if not empty then release it
434     if (¬imageResized.empty())
435     {
436         imageResized.release();
437     }
438
439     if ((width ≡ 0) ∧ (height ≡ 0)) // reset to original size
440     {
441         if (directResize) // direct set size to original size
442         {
443             setDirectSize((unsigned int)originalSize.width(),
444                           (unsigned int)originalSize.height());
445             // image is updated into setDirectSize
446         }
447         preferredWidth = image.cols;
448         preferredHeight = image.rows;
449
450         resize = false;
451         imageResized = image;
452     }
453     else // width != 0 or height != 0
454     {
455         if ((width ≡ (unsigned int)image.cols) ∧
456             (height ≡ (unsigned int)image.rows)) // unchanged
457         {
458             preferredWidth = image.cols;
459             preferredHeight = image.rows;
460             imageResized = image;
461
462             if (((int)preferredWidth ≡ originalSize.width()) ∧
463                 ((int)preferredHeight ≡ originalSize.height()))
464             {
465                 resize = false;
466             }
467             else
468             {
469                 resize = true;
470             }
471         }
472         else // width or height have changed
473         {
474             /*
475              * Resize needed
476              */
477             preferredWidth = width;
478             preferredHeight = height;
479
480             resize = true;
481
482             if (directResize)
483             {
484                 setDirectSize(preferredWidth, preferredHeight);
485                 imageResized = image;
486             }
487             else
488             {
489                 imageResized = Mat(preferredHeight, preferredWidth, image.type());
490             }
491         }
492     }
```



```
493
494     if (updateThread ≠ NULL)
495     {
496         lockLevel--;
497         if (lockLevel ≡ 0)
498         {
499             // qDebug("QcvVideoCapture::setSize unlocked");
500             mutex.unlock();
501         }
502     }
503
504     qDebug("QcvVideoCapture resize is %s [%s]",
505            (resize ? "ON" : "OFF"),
506            (directResize ? "direct" : "soft"));
507
508     size.setWidth(preferredWidth);
509     size.setHeight(preferredHeight);
510     statusMessage.clear();
511     statusMessage.sprintf("Size set to %dx%d", preferredWidth, preferredHeight);
512     emit messageChanged(statusMessage, messageDelay);
513
514
515     /*
516      * imageChanged signal is delayed until setGray is called into
517      * setFlipVideo
518      */
519     // Refresh image chain
520     setFlipVideo(flipVideo);
521 }
522
523 /*
524  * Sets #imageDisplay size according to preferred width and height
525  * @param size new desired size to set
526  * @pre a first image have been grabbed
527  */
528 void QcvVideoCapture::setSize(const QSize & size)
529 {
530     setSize(size.width(), size.height());
531 }
532
533 /*
534  * Sets video flipping
535  * @param flipVideo flipped video or not
536  */
537 void QcvVideoCapture::setFlipVideo(const bool flipVideo)
538 {
539     bool previousFlip = this→flipVideo;
540     this→flipVideo = flipVideo;
541
542     if (updateThread ≠ NULL)
543     {
544         if (lockLevel ≡ 0)
545         {
546             mutex.lock();
547             // qDebug() << "QcvVideoCapture::setFlipVideo(): lock";
548         }
549         lockLevel++;
550     }
551
552     if (¬imageFlipped.empty())
553     {
554         imageFlipped.release();
555     }
556
557     if (flipVideo)
558     {
559         imageFlipped = Mat(imageResized.size(), imageResized.type());
560     }
561     else
562     {
563         imageFlipped = imageResized;
564     }
565
566     if (updateThread ≠ NULL)
567     {
568         lockLevel--;
569         if (lockLevel ≡ 0)
570         {
571             // qDebug() << "QcvVideoCapture::setFlipVideo(): unlock";
572             mutex.unlock();
573         }
574     }
```

```
575
576     if (previousFlip ≠ flipVideo)
577     {
578         statusMessage.clear();
579         statusMessage.sprintf("flip video is %s", (flipVideo ? "on" : "off"));
580         emit messageChanged(statusMessage, messageDelay);
581         emit imageChanged(&imageDisplay);
582     }
583
584     /*
585      * imageChanged signal is delayed until setGray is called
586      */
587     // refresh image chain
588     setGray(gray);
589 }
590
591 /*
592  * Sets video conversion to gray
593  * @param grayConversion the gray conversion status
594  */
595 void QcvVideoCapture::setGray(const bool grayConversion)
596 {
597     bool previousGray = gray;
598
599     gray = grayConversion;
600
601     if (updateThread ≠ NULL)
602     {
603         if (lockLevel ≡ 0)
604         {
605             mutex.lock();
606             // qDebug() << "QcvVideoCapture::setGray(): lock";
607         }
608         lockLevel++;
609     }
610
611     if (¬imageDisplay.empty())
612     {
613         imageDisplay.release();
614     }
615
616     if (gray)
617     {
618         imageDisplay = Mat(imageFlipped.size(), CV_8UC1);
619     }
620     else
621     {
622         imageDisplay = imageFlipped;
623     }
624
625     if (updateThread ≠ NULL)
626     {
627         lockLevel−−;
628         if (lockLevel ≡ 0)
629         {
630             mutex.unlock();
631             // qDebug() << "QcvVideoCapture::setGray(): unlock";
632         }
633     }
634
635     if (previousGray ≠ grayConversion)
636     {
637         statusMessage.clear();
638         statusMessage.sprintf("gray video is %s", (gray ? "on" : "off"));
639         emit messageChanged(statusMessage, messageDelay);
640     }
641
642     /*
643      * In any cases emit image changed since
644      *  − setSize may have been called
645      *  − setFlipVideo may have been called
646      */
647     emit imageChanged(&imageDisplay);
648 }
649
650 /*
651  * Gets resize state.
652  * @return true if imageDisplay have been resized to preferred width and
653  * height, false otherwise
654  */
655 bool QcvVideoCapture::isResized() const
656 {
```



```
657     return resize;
658 }
659
660 /*
661  * Gets direct resize state.
662  * @return true if image can be resized directly into capture.
663  * @note direct resize capabilities are tested into #grabTest which is
664  * called in all constructors. So #isDirectResizeable should not be
665  * called before #grabTest
666  */
667 bool QcvVideoCapture::isDirectResizeable() const
668 {
669     return directResize;
670 }
671
672 /*
673  * Gets video flipping status
674  * @return flipped video status
675  */
676 bool QcvVideoCapture::isFlipVideo() const
677 {
678     return flipVideo;
679 }
680
681 /*
682  * Gets video gray converted status
683  * @return the converted to gray status
684  */
685 bool QcvVideoCapture::isGray() const
686 {
687     return gray;
688 }
689
690 /*
691  * Gets the image skipping policy
692  * @return true if new image can be skipped when previous one has not
693  * been processed yet, false otherwise.
694  */
695 bool QcvVideoCapture::isSkippable() const
696 {
697     return skip;
698 }
699
700
701 /*
702  * Gets the current frame rate
703  * @return the current frame rate
704  */
705 double QcvVideoCapture::getFrameRate() const
706 {
707     return frameRate;
708 }
709
710
711 /*
712  * Image accessor
713  * @return the image
714  */
715 Mat * QcvVideoCapture::getImage()
716 {
717     return &imageDisplay;
718 }
719
720 /*
721  * The source image mutex
722  * @return  the mutex used on image access
723  */
724 QMutex * QcvVideoCapture::getMutex()
725 {
726     return &mutex;
727 }
728
729
730 /*
731  * Performs a grab test to fill #image
732  * @return true if capture is opened and successfully grabs a first
733  * frame into #image, false otherwise
734  */
735 bool QcvVideoCapture::grabTest()
736 {
737 //  qDebug("Grab test");
738     bool result = false;
```

```
739
740     if (capture.isOpened())
741     {
742 #ifndef Q_OS_LINUX // V4L does not support these queries
743         int capWidth = capture.get(CV_CAP_PROP_FRAME_WIDTH);
744         int capHeight = capture.get(CV_CAP_PROP_FRAME_HEIGHT);
745
746         qDebug("Capture grab test with %d x %d image", capWidth, capHeight);
747 #endif
748         // grabs first frame
749         if (capture.grab())
750         {
751             bool retrieved = capture.retrieve(image);
752             if (retrieved)
753             {
754                 size.setWidth(image.cols);
755                 size.setHeight(image.rows);
756                 originalSize.setWidth(image.cols);
757                 originalSize.setHeight(image.rows);
758
759                 /*
760                  * Tries to determine if direct resizing in capture is possible
761                  * by setting original size through properties
762                  * Typically :
763                  *  - camera capture might be resizable
764                  *  - video file capture may not be resizable
765                  */
766                 directResize = setDirectSize(image.cols, image.rows);
767
768                 qDebug("Capture direct resizing is %s",
769                        (directResize ? "on" : "off"));
770
771                 result = true;
772             }
773             else
774             {
775                 qFatal("Video Capture unable to retreive image");
776             }
777         }
778         else
779         {
780             qFatal("Video Capture can not grab");
781         }
782     }
783     else
784     {
785         qFatal("Video Capture is not opened");
786     }
787
788     return result;
789 }
790
791 /*
792  * Get or compute interval between two frames
793  * @return interval between two frames
794  * @pre capture is already instanciated
795  */
796 int QcvVideoCapture::grabInterval(const QString & message)
797 {
798     int frameDelay = defaultFrameDelay;
799
800     // Tries to get framerate from capture
801     // ------------------------------------------------------------------
802     // Caution : on some systems getting video parameters is forbidden !
803     // For instance it does not work wirh linuxes equipped with V4L
804     // ------------------------------------------------------------------
805 #ifndef Q_OS_LINUX
806     frameRate = capture.get(CV_CAP_PROP_FPS);
807 #else
808     frameRate = -1.0;
809 #endif
810
811 //  qDebug("framerate direct query = %f", frameRate);
812
813     /*
814      * if capture obtained frameRate is inconsistent, then we'll try to find out
815      * by ourselves
816      */
817     if (frameRate ≤ 0.0)
818     {
819         /*
820          * If live Video : grab a few images and measure elapsed time
```



```
821          */
822         if (liveVideo)
823         {
824             QElapsedTimer localTimer;
825             localTimer.start();
826
827             for (size_t i=0; i < defaultFrameNumberTest; i++)
828             {
829                 capture >> image;
830             }
831
832             frameDelay = (int)(localTimer.elapsed() / defaultFrameNumberTest);
833             frameRate = 1.0/((double)frameDelay/1000.0);
834             qDebug("Measured capture frame rate is  %4.2f images/s", frameRate);
835         }
836         /*
837          * FIXME else ???
838          * video files read through capture should provide framerate with
839          * capture.get(CV_CAP_PROP_FPS) but what happens if they don't ???
840          */
841     }
842     else
843     {
844         qDebug("%s Capture frame rate = %4.2f", message.toStdString().c_str(),
845                                                 frameRate);
846         frameDelay = 1000/frameRate;
847     }
848
849     statusMessage.sprintf("%s frame rate = %4.2f images/s",
850                           message.toStdString().c_str(), frameRate);
851     emit messageChanged(statusMessage, messageDelay);
852
853     return frameDelay;
854 }
855
856 /*
857  * Tries to set capture size directly on capture by using properties.
858  *  - CV_CAP_PROP_FRAME_WIDTH to set frame width
859  *  - CV_CAP_PROP_FRAME_HEIGHT to set frame height
860  * @param width the width property to set on capture
861  * @param height the height property to set on capture
862  * @return true if capture is opened and if width and height have been
863  * set successfully through @code capture.set(...) @endcode. Returns
864  * false otherwise.
865  * @post if at least width or height have been set successfully, capture
866  * image is released then updated again so it will have the right
867  * dimensions.
868  */
869 bool QcvVideoCapture::setDirectSize(const unsigned int width,
870                                     const unsigned int height)
871 {
872 #ifdef Q_OS_LINUX
873     Q_UNUSED(width);
874     Q_UNUSED(height);
875 #endif
876     bool done = false;
877
878     /*
879      * We absolutely need this lock in order to safely set width and
880      * height directly into the capture, so if mutex is already locked
881      * we should wait for it to be unlocked before continuing. Moreover,
882      * if mutex is NON-recursive and already locked, the call to lock() could
883      * lead to a DEADlock, so mutex HAS to be recursive !
884      */
885
886 #ifndef Q_OS_LINUX
887     if (capture.isOpened())
888     {
889         bool setWidth = capture.set(CV_CAP_PROP_FRAME_WIDTH, (double)width);
890         bool setHeight = capture.set(CV_CAP_PROP_FRAME_HEIGHT, (double)height);
891         if (setWidth ∨ setHeight)
892         {
893             // release old capture image
894             image.release();
895
896             // force image update to get the right size
897             capture >> image;
898
899             done = true;
900         }
901     }
902 #endif
```

```
903
904         return done;
905     }
906
907     /*
908      * update slot trigerred by timer : Grabs a new image and sends updated()
909      * signal iff new image has been grabbed, otherwise there is no more
910      * images to grab so kills timer
911      */
912     void QcvVideoCapture::update()
913     {
914         bool locked = true;
915         bool image_updated = false;
916
917         if (updateThread ≠ NULL)
918         {
919             if (skip)
920             {
921                 locked = mutex.tryLock();
922                 // qDebug() << "QcvVideoCapture::update trylock"
923                 //          << (locked ? "granted" : "failed");
924                 if (locked)
925                 {
926                     lockLevel++;
927                 }
928             }
929             else
930             {
931                 if (lockLevel ≡ 0)
932                 {
933                     mutex.lock();
934                     // qDebug() << "QcvVideoCapture::update lock";
935                 }
936                 lockLevel++;
937             }
938         }
939
940         if (capture.isOpened() ∧ locked)
941         {
942             capture >> image;
943
944             if (¬image.data) // captured image has no data
945             {
946                 statusMessage.clear();
947
948                 if (liveVideo)
949                 {
950                     if (timer→isActive())
951                     {
952                         timer→stop();
953                         qDebug("timer stopped");
954                     }
955
956                     capture.release();
957
958                     statusMessage.sprintf("No more frames to capture ...");
959                     emit messageChanged(statusMessage, 0);
960                     qDebug("%s", statusMessage.toStdString().c_str());
961                 }
962                 else // not live video ==> video file
963                 {
964                     // We'll try to rewinds the file back to frame 0
965                     bool restart = capture.set(CV_CAP_PROP_POS_FRAMES, 0.0);
966
967                     if (restart)
968                     {
969                         statusMessage.sprintf("Capture restarted");
970                         emit messageChanged(statusMessage,
971                                             QcvVideoCapture::messageDelay);
972                         qDebug("%s", statusMessage.toStdString().c_str());
973
974                         // Refresh image chain resized -> flipped -> gray
975                         setSize(size);
976                     }
977                     else
978                     {
979                         capture.release();
980
981                         statusMessage.sprintf("Failed to restart capture ...");
982                         emit messageChanged(statusMessage, 0);
983                         emit(finished());
984                         qDebug("%s", statusMessage.toStdString().c_str());
```



```
985                 }
986             }
987         }
988         else // capture image has data
989         {
990             /*
991              * CAUTION
992              * image->imageResized->imageFlipped->imageDisplay
993              * constitute an image chain, so when size is changed with
994              * setSize it should call setFlipVideo which should call
995              * setGray
996              */
997
998             // resize image
999             if (resize ∧ ¬directResize)
1000            {
1001                cv::resize(image, imageResized, imageResized.size(), 0, 0,
1002                    INTER_AREA);
1003            }
1004            /*
1005             * else imageResized.data is already == image.data
1006             */
1007
1008            // flip image horizontally if required
1009            if (flipVideo)
1010            {
1011                flip(imageResized, imageFlipped, 1);
1012            }
1013            /*
1014             * else imageFlipped.data is already == imageResized.data
1015             */
1016
1017            // convert image to gray if required
1018            if (gray)
1019            {
1020                cvtColor(imageFlipped, imageDisplay, CV_BGR2GRAY);
1021            }
1022            /*
1023             * else imageDisplay.data is already == imageFlipped.data
1024             */
1025            image_updated = true;
1026        }
1027
1028        if (updateThread ≠ NULL)
1029        {
1030            lockLevel--;
1031            if (lockLevel ≡ 0)
1032            {
1033                // qDebug() << "QcvVideoCapture::update unlock";
1034                mutex.unlock();
1035            }
1036        }
1037
1038        if (image_updated)
1039        {
1040            emit updated();
1041        }
1042    }
1043    else
1044    {
1045        // mutex hasn't been locked, so we skipped one capture
1046        // qDebug() << "Capture skipped an image";
1047    }
1048 }
```

```
/*
 * CaptureFactory.h
 *
 *  Created on: 11 fÃ©vr. 2012
 *      Author: davidroussel
 */

#ifndef CAPTUREFACTORY_H_
#define CAPTUREFACTORY_H_

#include <QString>
#include <QStringList>
#include <QThread>
#include "QcvVideoCapture.h"

/**
 * Capture Factory creates QcvVideoCapture from arguments list
 */
class CaptureFactory
{
    protected:
        /**
         * The capture instance to create
         */
        QcvVideoCapture *capture;

        /**
         * Device number to open. Generally :
         *  - 0 is internal or fisrt camera
         *  - 1 is external or second camera
         */
        int deviceNumber;

        /**
         * Indicates capture opens camera or file.
         * Default value is true
         */
        bool liveVideo;

        /**
         * Video should be flipped horizontally for mirror effect
         * Default value is false
         */
        bool flippedVideo;

        /**
         * Video should be converted to gray during capture.
         * Default value is false
         */
        bool grayVideo;

        /**
         * Capture can skip capturing new image when previous image has not
         * been processed yet, or can wait for the previous image to be
         * processed before grabbing a new image.
         */
        bool skipImages;

        /**
         * Video preferred width (evt resize video)
         * Default value is 0 which means no preferred width
         */
        int preferredWidth;

        /**
         * Video preferred height (evt resize video)
         * Default value is 0 which means no preferred height
         */
        int preferredHeight;

        /**
         * Path to video file
         */
        QString videoPath;

    public:
        /**
         * Capture Factory constructor.
         * Arguments can be
         *  - [-d | --device] <device number> : camera number
         *  - [-f | --file] <filename> : video file name
         *  - [-m | --mirror] : flip image horizontally
```



```
         *  - [-g | --gray] : convert to gray level
         *  - [-s | --size] <width>x<height>: preferred width and height
         * @param argList program the argument list provided as a list of
         * strings
         */
        CaptureFactory(const QStringList & argList);

        /**
         * Capture factory destructor
         */
        virtual ~CaptureFactory();

        /**
         * Set the capture to live (webcam) or file source
         * @param live the video source
         */
        void setLiveVideo(const bool live);

        /**
         * Set device number to use when instanciating the capture with
         * live video.
         * @param deviceNumber the device number to use
         */
        void setDeviceNumber(const int deviceNumber);

        /**
         * Set path to video file when #liveVideo is false
         * @param path the path to the video file source
         */
        void setFile(const QString & path);

        /**
         * Set video horizontal flip state (useful for selfies)
         * @param flipped the horizontal flip state
         */
        void setFlipped(const bool flipped);

        /**
         * Set gray conversion
         * @param gray the gray conversion state
         */
        void setGray(const bool gray);

        /**
         * Set video grabbing skippable. When true, grabbing is skipped when
         * previously grabbed image has not been processed yet. Otherwise,
         * grabbing new image wait for the previous image to be processed.
         * This only applies if capture is run in a separate thread.
         * @param skip the video grabbing skippable state
         */
        void setSkippable(const bool skip);

        /**
         * Set video size (independently of video source actual size)
         * @param width the desired image width
         * @param height the desired image height
         */
        void setSize(const size_t width, const size_t height);

        /**
         * Set video size (independently of video source actual size)
         * @param size the desired video size
         */
        void setSize(const QSize & size);

        /**
         * Provide capture instanciated according to values
         * extracted from argument lists
         * @param updateThread the thread to run this capture or NULL if this
         * capture run in the current thread
         * @return the new capture instance
         */
        QcvVideoCapture * getCaptureInstance(QThread * updatethread = NULL);
};

#endif /* CAPTUREFACTORY_H_ */
```

```
/*
 * CaptureFactory.cpp
 *
 *  Created on: 11 fÃ©vr. 2012
 *      Author: davidroussel
 */

#include <cstdlib>  // for NULL
#include <QDebug>
#include <QFile>
#include <QtGlobal>
#include <QStringListIterator>
#include "CaptureFactory.h"

/*
 * Capture Factory constructor.
 * Arguments can be
 *  - [-d | --device] <device number> : camera number
 *  - [-f | --file] <filename> : video file name
 *  - [-m | --mirror] : flip image horizontally
 *  - [-g | --gray] : convert to gray level
 *  - [-s | --size] <width>x<height>: preferred width and height
 * @param argList program the argument list provided as a list of
 * strings
 */
CaptureFactory::CaptureFactory(const QStringList & argList) :
    capture(NULL),
    deviceNumber(0),
    liveVideo(true),
    flippedVideo(false),
    grayVideo(false),
    skipImages(false),
    preferredWidth(0),
    preferredHeight(0),
    videoPath()
{
    // C++ Like iterator
    // for (QStringList::const_iterator it = argList.begin(); it != argList.end(); ++it)
    // Java like iterator (because we use hasNext multiple times)
    for (QListIterator<QString> it(argList); it.hasNext(); )
    {
        QString currentArg(it.next());

        if (currentArg ≡ "-d" ∨ currentArg ≡"--device")
        {
            // Next argument should be device number integer
            if (it.hasNext())
            {
                QString deviceString(it.next());
                bool convertOk;
                deviceNumber = deviceString.toInt(&convertOk,10);
                if (¬convertOk ∨ deviceNumber < 0)
                {
                    qWarning("Warning: Invalid device number %d",deviceNumber);
                    deviceNumber = 0;
                }
                liveVideo = true;
            }
            else
            {
                qWarning("Warning: device tag found with no following device number");
            }
        }
        else if (currentArg ≡ "-v" ∨ currentArg ≡ "--video")
        {
            // Next argument should be a path name to video file or URL
            if (it.hasNext())
            {
                videoPath = it.next();
                liveVideo = false;
            }
            else
            {
                qWarning("file tag found with no following filename");
            }
        }
        else if (currentArg ≡ "-m" ∨ currentArg ≡ "--mirror")
        {
            flippedVideo = true;
        }
        else if (currentArg ≡ "-g" ∨ currentArg ≡ "--gray")
        {
```



```
            grayVideo = true;
        }
        else if (currentArg ≡ "-k" ∨ currentArg ≡ "--skip")
        {
            skipImages = true;
        }
        else if (currentArg ≡ "-s" ∨ currentArg ≡ "--size")
        {
            if (it.hasNext())
            {
                // search for <width>x<height>
                QString sizeString = it.next();
                int xIndex = sizeString.indexOf(QChar('x'), 0,
                    Qt::CaseInsensitive);
                if (xIndex ≠ -1)
                {
                    QString widthString = sizeString.left(xIndex);
                    preferredWidth = widthString.toUInt();
                    qDebug("preferred width is %d", preferredWidth);

                    QString heightString = sizeString.remove(0, xIndex+1);
                    preferredHeight = heightString.toUInt();
                    qDebug("preferred height is %d", preferredHeight);
                }
                else
                {
                    qWarning("invalid <width>x<height>");
                }
            }
            else
            {
                qWarning("size not found after --size");
            }
        }
    }
}

/*
 * Capture factory destructor
 */
CaptureFactory::~CaptureFactory()
{
}

/*
 * Set the capture to live (webcam) or file source
 * @param live the video source
 */
void CaptureFactory::setLiveVideo(const bool live)
{
    liveVideo = live;
}

/*
 * Set device number to use when instanciating the capture with
 * live video.
 * @param deviceNumber the device number to use
 */
void CaptureFactory::setDeviceNumber(const int deviceNumber)
{
    if (deviceNumber ≥ 0)
    {
        this→deviceNumber = deviceNumber;
    }
    else
    {
        qWarning("CaptureFactory::setDeviceNumber: invalid number %d", deviceNumber);
    }
}

/*
 * Set path to video file when #liveVideo is false
 * @param path the path to the video file source
 */
void CaptureFactory::setFile(const QString & path)
{
    if (QFile::exists(path))
    {
        videoPath = path;
    }
    else
    {
```

```
165         qWarning() << QObject::tr("CaptureFactory::setFile: path") << path
166                    << QObject::tr(" does not exist");
167     }
168 }
169
170 /*
171  * Set video horizontal flip state (useful for selfies)
172  * @param flipped the horizontal flip state
173  */
174 void CaptureFactory::setFlipped(const bool flipped)
175 {
176     flippedVideo = flipped;
177 }
178
179 /*
180  * Set gray conversion
181  * @param gray the gray conversion state
182  */
183 void CaptureFactory::setGray(const bool gray)
184 {
185     grayVideo = gray;
186 }
187
188 /*
189  * Set video grabbing skippable. When true, grabbing is skipped when
190  * previously grabbed image has not been processed yet. Otherwise,
191  * grabbing new image wait for the previous image to be processed.
192  * This only applies if capture is run in a separate thread.
193  * @param skip the video grabbing skippable state
194  */
195 void CaptureFactory::setSkippable(const bool skip)
196 {
197     skipImages = skip;
198 }
199
200 /*
201  * Set video size (independently of video source actual size)
202  * @param width the desired image width
203  * @param height the desired image height
204  */
205 void CaptureFactory::setSize(const size_t width, const size_t height)
206 {
207     preferredWidth = (int)width;
208     preferredHeight = (int)height;
209 }
210
211 /*
212  * Set video size (independently of video source actual size)
213  * @param size the desired video size
214  */
215 void CaptureFactory::setSize(const QSize & size)
216 {
217     preferredWidth = size.width();
218     preferredHeight = size.height();
219 }
220
221 /*
222  * Provide capture instanciated according to values
223  * extracted from argument lists
224  * @param updateThread the thread to run this capture or NULL if this
225  * capture run in the current thread
226  * @return the new capture instance
227  */
228 QcvVideoCapture * CaptureFactory::getCaptureInstance(QThread * updateThread)
229 {
230     // ------------------------------------------------------------------------
231     // Opening Video Capture
232     // ------------------------------------------------------------------------
233     if (liveVideo)
234     {
235         qDebug() << "opening device # " << deviceNumber;
236     }
237     else
238     {
239         qDebug() << "opening video file " << videoPath;
240     }
241
242     qDebug() << "Opening ";
243     if (liveVideo)
244     {
245         // Live video feed
246         qDebug() << "Live Video ... from camera # " << deviceNumber;
```



```
247         capture = new QcvVideoCapture(deviceNumber,
248                                       flippedVideo,
249                                       grayVideo,
250                                       skipImages,
251                                       preferredWidth,
252                                       preferredHeight,
253                                       updateThread);
254     }
255     else
256     {
257         // Video file or stream
258         qDebug() << videoPath << "…";
259         capture = new QcvVideoCapture(videoPath,
260                                       flippedVideo,
261                                       grayVideo,
262                                       skipImages,
263                                       preferredWidth,
264                                       preferredHeight,
265                                       updateThread);
266     }
267
268     return capture;
269 }
270
```

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include "QcvVideoCapture.h"
#include "QcvHistograms.h"

/**
 * Namespace for generated UI
 */
namespace Ui {
    class MainWindow;
}

/**
 * Rendering mode for main image
 */
typedef enum
{
    RENDER_IMAGE = 0,//!< QImage rendering mode
    RENDER_PIXMAP,   //!< QPixmap in a QLabel rendering mode
    RENDER_GL        //!< OpenGL in a QGLWidget rendering mode
} RenderMode;

/**
 * OpenCV/Qt Histograms and LUT main window
 */
class MainWindow : public QMainWindow
{
    Q_OBJECT

    public:
        /**
         * MainWindow constructor.
         * @param capture the capture QObject to capture frames from devices
         * or video files
         * @param processor processor and LUT processing class
         * @param parent parent widget
         */
        explicit MainWindow(QcvVideoCapture * capture,
                            QcvHistograms * histograms,
                            QWidget *parent = NULL);

        /**
         * MainWindow destructor
         */
        virtual ~MainWindow();

    signals:
        /**
         * Signal to send update message when something changes
         * @param message the message
         * @param timeout number of ms the message should be displayed
         */
        void sendMessage(const QString & message, int timeout = 0);

        /**
         * Signal to send when video size is changed
         * @param size the new video size
         */
        void sizeChanged(const QSize & size);

        /**
         * Signal to send when requesting opening a device (camera)
         * @param deviceId the device ID
         * @param width the requested video width
         * @param height the requested video height
         */
        void openDevice(const int deviceId,
                        const unsigned int width,
                        const unsigned int height);
        /**
         * Signal to send when requesting opening a file
         * @param deviceId the device ID
         * @param width the requested video width
         * @param height the requested video height
         */
        void openFile(const QString & fileName,
                      const unsigned int width,
                      const unsigned int height);

        /**
```



```
         * Signal to send when requesting video flip
         * @param flip video flip
         */
        void flipVideo(const bool flip);

    private:
        /**
         * The UI built in QtDesigner or QtCreator
         */
        Ui::MainWindow *ui;

        /**
         * The Capture object grabs frame using OpenCV HiGui
         */
        QcvVideoCapture * capture;

        /**
         * The Hist and LUT object compute histograms and performs LUT
         * on capture source image
         */
        QcvHistograms * processor;

        /**
         * Image preferred width
         */
        int preferredWidth;

        /**
         * Image preferred height
         */
        int preferredHeight;

        /**
         * Message to send to statusBar
         */
        QString message;

        /**
         * Changes widgetImage nature according to desired rendering mode.
         * Possible values for mode are:
         *  - IMAGE: widgetImage is assigned to a QcvMatWidgetImage instance
         *  - PIXMAP: widgetImage is assigned to a QcvMatWidgetLabel instance
         *  - GL: widgetImage is assigned to a QcvMatWidgetGL instance
         * @param mode
         */
        void setupImageWidget(const RenderMode mode);

        /**
         * Setup UI from capture settings when launching application
         */
        void setupUIfromCapture();

        /**
         * Setup UI from processor settings when launching application
         */
        void setupUIfromProcessor();

    private slots:

        /**
         * Re setup processor from UI settings when source image changes
         */
        void setupProcessorFromUI();

        /**
         * Menu action when Sources->camera 0 is selected
         * Sets capture to open device 0. If device is not available
         * menu item is set to inactive.
         */
        void on_actionCamera_0_triggered();

        /**
         * Menu action when Sources->camera 1 is selected
         * Sets capture to open device 0. If device is not available
         * menu item is set to inactive
         */
        void on_actionCamera_1_triggered();

        /**
         * Menu action when Sources->file is selected.
         * Opens file dialog and tries to open selected file (is not empty),
         * then sets capture to open the selected file
```

```
165             */
166         void on_actionFile_triggered();
167 
168         /**
169          * Menu action to quit application.
170          */
171         void on_actionQuit_triggered();
172 
173         /**
174          * Menu action when flip image is selected.
175          * Sets capture to change flip status which leads to reverse
176          * image horizontally
177          */
178         void on_actionFlip_triggered();
179 
180         /**
181          * Menu action when original image size is selected.
182          * Sets capture not to resize image
183          */
184         void on_actionOriginalSize_triggered();
185 
186         /**
187          * Menu action when constrained image size is selected.
188          * Sets capture resize to preferred width and height
189          */
190         void on_actionConstrainedSize_triggered();
191 
192         /**
193          * Menu action to replace current image rendering widget by a
194          * QcvMatWidgetImage instance.
195          */
196         void on_actionRenderImage_triggered();
197 
198         /**
199          * Menu action to replace current image rendering widget by a
200          * QcvMatWidgetLabel with pixmap instance.
201          */
202         void on_actionRenderPixmap_triggered();
203 
204         /**
205          * Menu action to replace current image rendering widget by a
206          * QcvMatWidgetGL instance.
207          */
208         void on_actionRenderOpenGL_triggered();
209 
210 
211         /**
212          * Original size radioButton action.
213          * Sets capture resize to off
214          */
215         void on_radioButtonOrigSize_clicked();
216 
217         /**
218          * Custom size radioButton action.
219          * Sets capture resize to preferred width and height
220          */
221         void on_radioButtonCustomSize_clicked();
222 
223         /**
224          * Width spinbox value change.
225          * Changes the preferred width and if custom size is selected apply
226          * this custom width
227          * @param value the desired width
228          */
229         void on_spinBoxWidth_valueChanged(int value);
230 
231         /**
232          * Height spinbox value change.
233          * Changes the preferred height and if custom size is selected apply
234          * this custom height
235          * @param value the desired height
236          */
237         void on_spinBoxHeight_valueChanged(int value);
238 
239         /**
240          * Flip capture image horizontally.
241          * changes capture flip status
242          */
243         void on_checkBoxFlip_clicked();
244 
245         /**
246          * Set transfert function to identity
```



```
247          */
248         void on_radioButtonIdentity_clicked();
249 
250         /**
251          * Set transfert function to inverse
252          */
253         void on_radioButtonInverse_clicked();
254 
255         /**
256          * Set transfert function to gamma
257          */
258         void on_radioButtonGamma_clicked();
259 
260         /**
261          * Set transfert function to threshold
262          */
263         void on_radioButtonThreshold_clicked();
264 
265         /**
266          * Set transfert function to optimal dynamic
267          */
268         void on_radioButtonDynamic_clicked();
269 
270         /**
271          * Set transfert function to equalization
272          */
273         void on_radioButtonEqualize_clicked();
274 
275         /**
276          * Set transfert function depending on processor to use colors
277          * components of the histogram generating 1 transfert function per image
278          * channels
279          */
280         void on_radioButtonChColor_clicked();
281 
282         /**
283          * Set transfert function depending on processor to use gray level
284          * histogram component generating 1 transfert function per image
285          * channels
286          */
287         void on_radioButtonChGray_clicked();
288 
289         /**
290          * Modify lut parameter applied to transfert function depending on
291          * histogram
292          * @param value the new value of lutParam
293          */
294         void on_spinBoxlutParam_valueChanged(int value);
295 
296         /**
297          * Set histogram mode to normal
298          */
299         void on_radioButtonHMNormal_clicked();
300 
301         /**
302          * Set Histogram mode to cumulative
303          */
304         void on_radioButtonHMCumulative_clicked();
305 
306         /**
307          * set Histogram mode to time cumulative
308          */
309         void on_radioButtonHMTime_clicked();
310 
311         /**
312          * Show/Hides histogram red component
313          */
314         void on_checkBoxHistRed_clicked();
315 
316         /**
317          * Show/Hides histogram green component
318          */
319         void on_checkBoxHistGreen_clicked();
320 
321         /**
322          * Show/Hides histogram Blue component
323          */
324         void on_checkBoxHistBlue_clicked();
325 
326         /**
327          * Show/Hides histogram gray component
328          */
```

```
329         void on_checkBoxHistGray_clicked();
330 };
331
332 #endif // MAINWINDOW_H
```



```
1  #include "mainwindow.h"
2  #include "ui_mainwindow.h"
3
4  #include <QObject>
5  #include <QFileDialog>
6  #include <QDebug>
7  #include <assert.h>
8
9  #include "QcvMatWidgetImage.h"
10 #include "QcvMatWidgetLabel.h"
11 #include "QcvMatWidgetGL.h"
12
13 /*
14  * MainWindow constructor
15  * @param capture the capture QObject to capture frames from devices
16  * or video files
17  * @param parent parent widget
18  */
19 MainWindow::MainWindow(QcvVideoCapture * capture,
20                        QcvHistograms * processor,
21                        QWidget *parent) :
22     QMainWindow(parent),
23     ui(new Ui::MainWindow),
24     capture(capture),
25     processor(processor),
26     preferredWidth(320),
27     preferredHeight(240)
28 {
29     ui→setupUi(this);
30     ui→scrollArea→setBackgroundRole(QPalette::Mid);
31
32     // ------------------------------------------------------------------------
33     // Assertions
34     // ------------------------------------------------------------------------
35     assert(capture ≠ NULL);
36
37     assert(processor ≠ NULL);
38
39     // ------------------------------------------------------------------------
40     // Special widgets initialisation
41     // ------------------------------------------------------------------------
42     ui→widgetImage→setSourceImage(processor→getImagePtr("out"));
43     ui→widgetHistogram→setSourceImage(processor→getImagePtr("histogram"));
44     ui→widgetLUT→setSourceImage(processor→getImagePtr("lut"));
45
46     // Replace widgetImage QcvMatWidget instance with QcvMatWidgetImage
47     // Sets Source image for widgetImage
48     // Connects processor->updated to widgetImage->update
49     // Connects processor->outImageChanged to widgetImage->setSourceImage
50     setupImageWidget(RENDER_IMAGE);
51
52     // ------------------------------------------------------------------------
53     // Signal/Slot connections
54     // ------------------------------------------------------------------------
55
56     // Histogram updates to various image widget updates
57     connect(processor, SIGNAL(histogramImageUpdated()),
58             ui→widgetHistogram, SLOT(update()));
59
60     connect(processor, SIGNAL(lutImageUpdated()),
61             ui→widgetLUT, SLOT(update()));
62
63     // Histogram source image changed to various image widget set sources
64
65     connect(processor, SIGNAL(histogramImageChanged(Mat*)),
66             ui→widgetHistogram, SLOT(setSourceImage(Mat*)));
67
68     connect(processor, SIGNAL(lutImageChanged(Mat*)),
69             ui→widgetLUT, SLOT(setSourceImage(Mat*)));
70
71     // Capture, histogram and this messages to status bar
72     connect(capture, SIGNAL(messageChanged(QString,int)),
73             ui→statusBar, SLOT(showMessage(QString,int)));
74
75     connect(processor, SIGNAL(sendMessage(QString,int)),
76             ui→statusBar, SLOT(showMessage(QString,int)));
77
78     connect(this, SIGNAL(sendMessage(QString,int)),
79             ui→statusBar, SLOT(showMessage(QString,int)));
80
81     // Connect UI signals to Capture slots
82     connect(this, SIGNAL(sizeChanged(const QSize &)),
```

```
            capture, SLOT(setSize(const QSize &)));
    connect(this, SIGNAL(openDevice(int,uint,uint)),
            capture, SLOT(open(int,uint,uint)));
    connect(this, SIGNAL(openFile(QString,uint,uint)),
            capture, SLOT(open(QString,uint,uint)));
    connect(this, SIGNAL(flipVideo(bool)), capture, SLOT(setFlipVideo(bool)));

    // When Processor source image changes, some attributes are reinitialised
    // So we have to set them up again according to current UI values
    connect(processor, SIGNAL(imageChanged()),
            this, SLOT(setupProcessorFromUI()));

    // Time measurement strings connections
    connect(processor, SIGNAL(processTimeUpdated(QString)),
            ui->labelAllTime, SLOT(setText(QString)));
    connect(processor, SIGNAL(histogramTime1Updated(QString)),
            ui->labelUH1Time, SLOT(setText(QString)));
    connect(processor, SIGNAL(histogramTime2Updated(QString)),
            ui->labelUH2Time, SLOT(setText(QString)));
    connect(processor, SIGNAL(drawHistogramTimeUpdated(QString)),
            ui->labelDHTime, SLOT(setText(QString)));
    connect(processor, SIGNAL(computeLUTTimeUpdated(QString)),
            ui->labelCLTime, SLOT(setText(QString)));
    connect(processor, SIGNAL(drawLUTTimeUpdated(QString)),
            ui->labelDLTime, SLOT(setText(QString)));
    connect(processor, SIGNAL(applyLUTTimeUpdated(QString)),
            ui->labelALTime, SLOT(setText(QString)));

    // ------------------------------------------------------------------------
    // UI setup accroding to capture ahd histogram settings
    // ------------------------------------------------------------------------
    setupUIfromCapture();

    setupUIfromProcessor();
}

/*
 * MainWindow destructor
 */
MainWindow::~MainWindow()
{
    delete ui;
}

/*
 * Menu action when Sources->camera 0 is selected
 * Sets capture to open device 0. If device is not available
 * menu item is set to inactive.
 */
void MainWindow::on_actionCamera_0_triggered()
{
    int width = 0;
    int height = 0;

    if (ui->radioButtonCustomSize->isChecked())
    {
        width = preferredWidth;
        height = preferredHeight;
    }

    qDebug("Opening device 0 ...");
//  if (!capture->open(0, width, height))
//  {
//      qWarning("Unable to open device 0");
//      // disable menu item if camera 0 does not exist
//      ui->actionCamera_0->setDisabled(true);
//  }

    emit openDevice(0, width, height);
}

/*
 * Menu action when Sources->camera 1 is selected
 * Sets capture to open device 0. If device is not available
 * menu item is set to inactive
 */
void MainWindow::on_actionCamera_1_triggered()
{
    int width = 0;
    int height = 0;

    if (ui->radioButtonCustomSize->isChecked())
```



```
    {
        width = preferredWidth;
        height = preferredHeight;
    }

    qDebug("Opening device 1 ...");
//  if (!capture->open(1, width, height))
//  {
//      qWarning("Unable to open device 1");
//      // disable menu item if camera 1 does not exist
//      ui->actionCamera_1->setDisabled(true);
//  }

    emit openDevice(1, width, height);
}

/*
 * Menu action when Sources->file is selected.
 * Opens file dialog and tries to open selected file (is not empty),
 * then sets capture to open the selected file
 */
void MainWindow::on_actionFile_triggered()
{
    int width = 0;
    int height = 0;

    if (ui->radioButtonCustomSize->isChecked())
    {
        width = preferredWidth;
        height = preferredHeight;
    }

    QString fileName =
    QFileDialog::getOpenFileName(this,
                                 tr("Open Video"),
                                 "./",
                                 tr("Video Files (*.avi *.mkv *.mp4 *.m4v)"),
                                 NULL,
                                 QFileDialog::ReadOnly);

    qDebug("Opening file %s ...", fileName.toStdString().c_str());

    if (fileName.length() > 0)
    {
//      if (!capture->open(fileName, width, height))
//      {
//          qWarning("Unable to open device file : %s",
//                   fileName.toStdString().c_str());
//      }
        emit openFile(fileName, width, height);
    }
    else
    {
        qWarning("empty file name");
    }
}

/*
 * Menu action to qui application
 */
void MainWindow::on_actionQuit_triggered()
{
    this->close();
}

/*
 * Menu action when flip image is selected.
 * Sets capture to change flip status which leads to reverse
 * image horizontally
 */
void MainWindow::on_actionFlip_triggered()
{
    // capture->setFlipVideo(!capture->isFlipVideo());
    emit flipVideo(!capture->isFlipVideo());
    /*
     * There is no need to update ui->checkBoxFlip since it is connected
     * to ui->actionFlip through signals/slots
     */
}

/*
 * Menu action when original image size is selected.
```

```
 * Sets capture not to resize image
 */
void MainWindow::on_actionOriginalSize_triggered()
{
    ui→actionConstrainedSize→setChecked(false);

    emit sizeChanged(QSize(0, 0));
}

/*
 * Menu action when constrained image size is selected.
 * Sets capture resize to preferred width and height
 */
void MainWindow::on_actionConstrainedSize_triggered()
{
    ui→actionOriginalSize→setChecked(false);

    emit sizeChanged(QSize(preferredWidth, preferredHeight));
}

/*
 * Changes widgetImage nature according to desired rendering mode.
 * Possible values for mode are:
 *  - IMAGE: widgetImage is assigned to a QcvMatWidgetImage instance
 *  - PIXMAP: widgetImage is assigned to a QcvMatWidgetLabel instance
 *  - GL: widgetImage is assigned to a QcvMatWidgetGL instance
 * @param mode
 */
void MainWindow::setupImageWidget(const RenderMode mode)
{
    // Disconnect first
    disconnect(processor, SIGNAL(outImageUpdated()),
               ui→widgetImage, SLOT(update()));

    disconnect(processor, SIGNAL(outImageChanged(Mat*)),
               ui→widgetImage, SLOT(setSourceImage(Mat*)));

    // remove widget in scroll area
    QWidget * w = ui→scrollArea→takeWidget();

    if (w ≡ ui→widgetImage)
    {
        // delete removed widget
        delete ui→widgetImage;

        // create new widget
        Mat * image = processor→getImagePtr("out");
        if (image ≡ NULL)
        {
            qFatal("Null image out");
        }
        if (image→data ≡ NULL)
        {
            qFatal("image out NULL data");
        }
        switch (mode)
        {
            case RENDER_PIXMAP:
                ui→widgetImage = new QcvMatWidgetLabel(image);
                break;
            case RENDER_GL:
                ui→widgetImage = new QcvMatWidgetGL(image);
                break;
            case RENDER_IMAGE:
            default:
                ui→widgetImage = new QcvMatWidgetImage(image);
                break;
        }

        if (ui→widgetImage ≠ NULL)
        {
            ui→widgetImage→setObjectName(QString::fromUtf8("widgetImage"));

            // add it to the scroll area
            ui→scrollArea→setWidget(ui→widgetImage);

            connect(processor, SIGNAL(outImageUpdated()),
                    ui→widgetImage, SLOT(update()));

            connect(processor, SIGNAL(outImageChanged(Mat*)),
                    ui→widgetImage, SLOT(setSourceImage(Mat*)));

```



```
            // Sends message to status bar and sets menu checks
            message.clear();
            message.append(tr("Render more set to "));
            switch (mode)
            {
                case RENDER_IMAGE:
                    ui→actionRenderPixmap→setChecked(false);
                    ui→actionRenderOpenGL→setChecked(false);
                    message.append(tr("QImage"));
                    break;
                case RENDER_PIXMAP:
                    ui→actionRenderImage→setChecked(false);
                    ui→actionRenderOpenGL→setChecked(false);
                    message.append(tr("QPixmap in QLabel"));
                    break;
                case RENDER_GL:
                    ui→actionRenderImage→setChecked(false);
                    ui→actionRenderPixmap→setChecked(false);
                    message.append("QGLWidget");
                    break;
                default:
                break;
            }
            emit sendMessage(message, 5000);
        }
        else
        {
            qDebug("MainWindow::on_actionRenderXXX new widget is null");
        }
    }
    else
    {
        qDebug("MainWindow::on_actionRenderXXX removed widget is not imageWidget");
    }
}

/**
 * Setup UI from capture settings when launching application
 */
void MainWindow::setupUIfromCapture()
{
    // ------------------------------------------------------------------------
    // UI setup according to capture options
    // ------------------------------------------------------------------------
    // Sets size radioButton states
    if (capture→isResized())
    {
        /*
         * Initial Size radio buttons configuration
         */
        ui→radioButtonOrigSize→setChecked(false);
        ui→radioButtonCustomSize→setChecked(true);
        /*
         * Initial Size menu items configuration
         */
        ui→actionOriginalSize→setChecked(false);
        ui→actionConstrainedSize→setChecked(true);

        QSize size = capture→getSize();
        qDebug("Capture->size is %dx%d", size.width(), size.height());
        preferredWidth = size.width();
        preferredHeight = size.height();
    }
    else
    {
        /*
         * Initial Size radio buttons configuration
         */
        ui→radioButtonCustomSize→setChecked(false);
        ui→radioButtonOrigSize→setChecked(true);

        /*
         * Initial Size menu items configuration
         */
        ui→actionConstrainedSize→setChecked(false);
        ui→actionOriginalSize→setChecked(true);
    }

    // Sets spinboxes preferred size
    ui→spinBoxWidth→setValue(preferredWidth);
    ui→spinBoxHeight→setValue(preferredHeight);

```

```
411     // Sets flipCheckbox and menu item states
412     bool flipped = capture→isFlipVideo();
413     ui→actionFlip→setChecked(flipped);
414     ui→checkBoxFlip→setChecked(flipped);
415 }
416
417
418 /**
419  * Setup UI from processor settings when launching application
420  */
421 void MainWindow::setupUIfromProcessor()
422 {
423     qDebug("Setting up UI from processor");
424
425
426     // ----------------------------------------------------------------------
427     // UI setup according to Histograms options
428     // ----------------------------------------------------------------------
429     // Histogram channel visibility
430     QCheckBox * checkBoxesChannels[4] =
431     {
432         ui→checkBoxHistRed,
433         ui→checkBoxHistGreen,
434         ui→checkBoxHistBlue,
435         ui→checkBoxHistGray
436     };
437
438     size_t nbHistograms = processor→getNbHistograms();
439
440     for (size_t i = 0; i < nbHistograms; i++)
441     {
442         checkBoxesChannels[i]→setChecked(processor→isShowComponent(i));
443     }
444
445     if (nbHistograms < 4)
446     {
447         for (size_t i = nbHistograms; i < 4; i++)
448         {
449             checkBoxesChannels[i]→setEnabled(false);
450         }
451     }
452
453     // Histogram mode
454     if (processor→isCumulative())
455     {
456         ui→radioButtonHMCumulative→setChecked(true);
457     }
458     else
459     {
460         ui→radioButtonHMNormal→setChecked(true);
461     }
462
463     if (processor→isTimeCumulative())
464     {
465         ui→radioButtonHMTime→setChecked(true);
466     }
467     else
468     {
469         ui→radioButtonHMNormal→setChecked(true);
470     }
471
472     // Current LUT
473     CvHistograms8UC3::TransfertType lutMode = processor→getLutType();
474
475     switch (lutMode)
476     {
477         case CvHistograms8UC3::THRESHOLD_GRAY:
478         case CvHistograms8UC3::THRESHOLD_COLOR:
479             ui→radioButtonThreshold→setChecked(true);
480             break;
481         case CvHistograms8UC3::DYNAMIC_GRAY:
482         case CvHistograms8UC3::DYNAMIC_COLOR:
483             ui→radioButtonDynamic→setChecked(true);
484             break;
485         case CvHistograms8UC3::EQUALIZE_GRAY:
486         case CvHistograms8UC3::EQUALIZE_COLOR:
487             ui→radioButtonEqualize→setChecked(true);
488             break;
489         case CvHistograms8UC3::GAMMA:
490             ui→radioButtonGamma→setChecked(true);
491             break;
492         case CvHistograms8UC3::NEGATIVE:
```



```
493             ui→radioButtonInverse→setChecked(true);
494             break;
495         case CvHistograms8UC3::NONE:
496         default:
497             ui→radioButtonIdentity→setChecked(true);
498             break;
499     }
500
501     // LUT mode : color/gray
502     switch (lutMode)
503     {
504         case CvHistograms8UC3::THRESHOLD_COLOR:
505         case CvHistograms8UC3::DYNAMIC_COLOR:
506         case CvHistograms8UC3::EQUALIZE_COLOR:
507             ui→radioButtonChColor→setChecked(true);
508             break;
509         case CvHistograms8UC3::THRESHOLD_GRAY:
510         case CvHistograms8UC3::DYNAMIC_GRAY:
511         case CvHistograms8UC3::EQUALIZE_GRAY:
512         case CvHistograms8UC3::GAMMA:
513         case CvHistograms8UC3::NEGATIVE:
514         case CvHistograms8UC3::NONE:
515         default:
516             ui→radioButtonChGray→setChecked(true);
517             break;
518     }
519
520     // If there is no additionnal gray level histogram we might change
521     // the channels radio buttons accordingly
522     if (¬processor→isComputeGray())
523     {
524 //      ui->radioButtonChGray->setChecked(false);
525         ui→radioButtonChColor→setChecked(true);
526         ui→radioButtonChGray→setEnabled(false);
527     }
528
529     // LUT param
530     ui→spinBoxlutParam→setValue((int)processor→getLUTParam());
531 }
532
533
534 /**
535  * Re setup processor from UI settings when source image changes
536  */
537 void MainWindow::setupProcessorFromUI()
538 {
539 //  qDebug("Setting up processor from UI");
540
541     // Sets histogram channel visibility
542     processor→setShowComponent(CvHistograms8UC3::HIST_RED,
543                                ui→checkBoxHistRed→isChecked());
544     processor→setShowComponent(CvHistograms8UC3::HIST_GREEN,
545                                ui→checkBoxHistGreen→isChecked());
546     processor→setShowComponent(CvHistograms8UC3::HIST_BLUE,
547                                ui→checkBoxHistBlue→isChecked());
548     if (processor→getNbHistograms() ≥ CvHistograms8UC3::HIST_GRAY)
549     {
550         processor→setShowComponent(CvHistograms8UC3::HIST_GRAY,
551                                    ui→checkBoxHistGray→isChecked());
552     }
553
554     // Sets Histogram mode
555     if (ui→radioButtonHMNormal→isChecked())
556     {
557         processor→setCumulative(false);
558         processor→setTimeCumulative(false);
559     }
560     else if (ui→radioButtonHMCumulative→isChecked())
561     {
562         processor→setCumulative(true);
563         processor→setTimeCumulative(false);
564     }
565     else
566     {
567         processor→setCumulative(false);
568         processor→setTimeCumulative(true);
569     }
570
571     processor→setLUTParam((float)ui→spinBoxlutParam→value());
572
573     if (ui→radioButtonIdentity→isChecked())
574     {
```

```
575         processor→setLutType(CvHistograms8UC3::NONE);
576     }
577     if (ui→radioButtonInverse→isChecked())
578     {
579         processor→setLutType(CvHistograms8UC3::NEGATIVE);
580     }
581     if (ui→radioButtonGamma→isChecked())
582     {
583         processor→setLutType(CvHistograms8UC3::GAMMA);
584     }
585     if (ui→radioButtonThreshold→isChecked())
586     {
587         if (ui→radioButtonChGray→isChecked())
588         {
589             processor→setLutType(CvHistograms8UC3::THRESHOLD_GRAY);
590         }
591         else
592         {
593             processor→setLutType(CvHistograms8UC3::THRESHOLD_COLOR);
594         }
595     }
596     if (ui→radioButtonDynamic→isChecked())
597     {
598         if (ui→radioButtonChGray→isChecked())
599         {
600             processor→setLutType(CvHistograms8UC3::DYNAMIC_GRAY);
601         }
602         else
603         {
604             processor→setLutType(CvHistograms8UC3::DYNAMIC_COLOR);
605         }
606     }
607     if (ui→radioButtonEqualize→isChecked())
608     {
609         if (ui→radioButtonChGray→isChecked())
610         {
611             processor→setLutType(CvHistograms8UC3::EQUALIZE_GRAY);
612         }
613         else
614         {
615             processor→setLutType(CvHistograms8UC3::EQUALIZE_COLOR);
616         }
617     }
618 }
619
620
621 /*
622  * Menu action to replace current image rendering widget by a
623  * QcvMatWidgetImage instance.
624  */
625 void MainWindow::on_actionRenderImage_triggered()
626 {
627     setupImageWidget(RENDER_IMAGE);
628 }
629
630 /*
631  * Menu action to replace current image rendering widget by a
632  * QcvMatWidgetLabel with pixmap instance.
633  */
634 void MainWindow::on_actionRenderPixmap_triggered()
635 {
636     setupImageWidget(RENDER_PIXMAP);
637 }
638
639 /*
640  * Menu action to replace current image rendering widget by a
641  * QcvMatWidgetGL instance.
642  */
643 void MainWindow::on_actionRenderOpenGL_triggered()
644 {
645     setupImageWidget(RENDER_GL);
646 }
647
648 /*
649  * Original size radioButton action.
650  * Sets capture resize to off
651  */
652 void MainWindow::on_radioButtonOrigSize_clicked()
653 {
654     ui→actionConstrainedSize→setChecked(false);
655     emit sizeChanged(QSize(0, 0));
656 }
```



```
657
658 /*
659  * Custom size radioButton action.
660  * Sets capture resize to preferred width and height
661  */
662 void MainWindow::on_radioButtonCustomSize_clicked()
663 {
664     ui→actionOriginalSize→setChecked(false);
665     emit sizeChanged(QSize(preferredWidth, preferredHeight));
666 }
667
668 /*
669  * Width spinbox value change.
670  * Changes the preferred width and if custom size is selected apply
671  * this custom width
672  * @param value the desired width
673  */
674 void MainWindow::on_spinBoxWidth_valueChanged(int value)
675 {
676     preferredWidth = value;
677     if (ui→radioButtonCustomSize→isChecked())
678     {
679         emit sizeChanged(QSize(preferredWidth, preferredHeight));
680     }
681 }
682
683 /*
684  * Height spinbox value change.
685  * Changes the preferred height and if custom size is selected apply
686  * this custom height
687  * @param value the desired height
688  */
689 void MainWindow::on_spinBoxHeight_valueChanged(int value)
690 {
691     preferredHeight = value;
692     if (ui→radioButtonCustomSize→isChecked())
693     {
694         emit sizeChanged(QSize(preferredWidth, preferredHeight));
695     }
696 }
697
698 /*
699  * Flip capture image horizontally.
700  * changes capture flip status
701  */
702 void MainWindow::on_checkBoxFlip_clicked()
703 {
704     /*
705      * There is no need to update ui->actionFlip since it is connected
706      * to ui->checkBoxFlip through signals/slots
707      */
708     // capture->setFlipVideo(ui->checkBoxFlip->isChecked());
709     emit flipVideo(ui→checkBoxFlip→isChecked());
710 }
711
712 /*
713  * Set transfert function to identity
714  */
715 void MainWindow::on_radioButtonIdentity_clicked()
716 {
717     processor→setLutType(CvHistograms8UC3::NONE);
718 }
719
720 /*
721  * Set transfert function to inverse
722  */
723 void MainWindow::on_radioButtonInverse_clicked()
724 {
725     processor→setLutType(CvHistograms8UC3::NEGATIVE);
726 }
727
728 /*
729  * Set transfert function to gamma
730  */
731 void MainWindow::on_radioButtonGamma_clicked()
732 {
733     processor→setLutType(CvHistograms8UC3::GAMMA);
734 }
735
736 /*
737  * Set transfert function to threshold
738  */
```

```
739 void MainWindow::on_radioButtonThreshold_clicked()
740 {
741     if (ui→radioButtonChGray→isChecked())
742     {
743         processor→setLutType(CvHistograms8UC3::THRESHOLD_GRAY);
744     }
745     else
746     {
747         processor→setLutType(CvHistograms8UC3::THRESHOLD_COLOR);
748     }
749 }
750
751 /*
752  * Set transfert function to optimal dynamic
753  */
754 void MainWindow::on_radioButtonDynamic_clicked()
755 {
756     if (ui→radioButtonChGray→isChecked())
757     {
758         processor→setLutType(CvHistograms8UC3::DYNAMIC_GRAY);
759     }
760     else
761     {
762         processor→setLutType(CvHistograms8UC3::DYNAMIC_COLOR);
763     }
764 }
765
766 /*
767  * Set transfert function to equalization
768  */
769 void MainWindow::on_radioButtonEqualize_clicked()
770 {
771     if (ui→radioButtonChGray→isChecked())
772     {
773         processor→setLutType(CvHistograms8UC3::EQUALIZE_GRAY);
774     }
775     else
776     {
777         processor→setLutType(CvHistograms8UC3::EQUALIZE_COLOR);
778     }
779 }
780
781 /*
782  * Set transfert function depending on processor to use colors
783  * components of the histogram generating 1 transfert function per image
784  * channels
785  */
786 void MainWindow::on_radioButtonChColor_clicked()
787 {
788     CvHistograms8UC3::TransfertType type = processor→getLutType();
789     switch (type)
790     {
791         case CvHistograms8UC3::THRESHOLD_GRAY:
792             processor→setLutType(CvHistograms8UC3::THRESHOLD_COLOR);
793             break;
794         case CvHistograms8UC3::DYNAMIC_GRAY:
795             processor→setLutType(CvHistograms8UC3::DYNAMIC_COLOR);
796             break;
797         case CvHistograms8UC3::EQUALIZE_GRAY:
798             processor→setLutType(CvHistograms8UC3::EQUALIZE_COLOR);
799             break;
800         // in all other cases do nothing
801         case CvHistograms8UC3::NONE:
802         case CvHistograms8UC3::GAMMA:
803         case CvHistograms8UC3::NEGATIVE:
804         default:
805             // Nothing
806             break;
807     }
808 }
809
810 /*
811  * Set transfert function depending on processor to use gray level
812  * histogram component generating 1 transfert function per image
813  * channels
814  */
815 void MainWindow::on_radioButtonChGray_clicked()
816 {
817     CvHistograms8UC3::TransfertType type = processor→getLutType();
818     switch (type)
819     {
820         case CvHistograms8UC3::THRESHOLD_COLOR:
```



```
821             processor→setLutType(CvHistograms8UC3::THRESHOLD_GRAY);
822             break;
823         case CvHistograms8UC3::DYNAMIC_COLOR:
824             processor→setLutType(CvHistograms8UC3::DYNAMIC_GRAY);
825             break;
826         case CvHistograms8UC3::EQUALIZE_COLOR:
827             processor→setLutType(CvHistograms8UC3::EQUALIZE_GRAY);
828             break;
829         // in all other cases do nothing
830         case CvHistograms8UC3::NONE:
831         case CvHistograms8UC3::GAMMA:
832         case CvHistograms8UC3::NEGATIVE:
833         default:
834             // Nothing
835             break;
836     }
837 }
838
839 /*
840  * Modify lut parameter applied to transfert function depending on
841  * histogram
842  * @param value the new value of lutParam
843  */
844 void MainWindow::on_spinBoxlutParam_valueChanged(int value)
845 {
846     processor→setLUTParam((float)value);
847 }
848
849 /*
850  * Set histogram mode to normal
851  */
852 void MainWindow::on_radioButtonHMNormal_clicked()
853 {
854     processor→setTimeCumulative(false);
855     processor→setCumulative(false);
856 }
857
858 /*
859  * Set Histogram mode to cumulative
860  */
861 void MainWindow::on_radioButtonHMCumulative_clicked()
862 {
863     processor→setTimeCumulative(false);
864     processor→setCumulative(true);
865 }
866
867 /*
868  * set Histogram mode to time cumulative
869  */
870 void MainWindow::on_radioButtonHMTime_clicked()
871 {
872     processor→setCumulative(false);
873     processor→setTimeCumulative(true);
874 }
875
876 /*
877  * Show/Hides histogram red component
878  */
879 void MainWindow::on_checkBoxHistRed_clicked()
880 {
881     processor→setShowComponent((size_t)CvHistograms8UC3::HIST_RED,
882                                ui→checkBoxHistRed→isChecked());
883 }
884
885 /*
886  * Show/Hides histogram green component
887  */
888 void MainWindow::on_checkBoxHistGreen_clicked()
889 {
890     processor→setShowComponent((size_t)CvHistograms8UC3::HIST_GREEN,
891                                ui→checkBoxHistGreen→isChecked());
892 }
893
894 /*
895  * Show/Hides histogram blue component
896  */
897 void MainWindow::on_checkBoxHistBlue_clicked()
898 {
899     processor→setShowComponent((size_t)CvHistograms8UC3::HIST_BLUE,
900                                ui→checkBoxHistBlue→isChecked());
901 }
902
```

```
903 /*
904  * Show/Hides histogram gray component
905  */
906 void MainWindow::on_checkBoxHistGray_clicked()
907 {
908     processor→setShowComponent((size_t)CvHistograms8UC3::HIST_GRAY,
909                                ui→checkBoxHistGray→isChecked());
910 }
```



```
1  #include <QApplication>
2  #include <QThread>
3  #include <QDebug>
4  #include <libgen.h>     // for basename
5  #include <iostream>     // for cout
6
7  #include "QcvVideoCapture.h"
8  #include "CaptureFactory.h"
9  #include "QcvHistograms.h"
10 #include "mainwindow.h"
11
12 /**
13  * Usage function shown just before launching QApp
14  * @param name the name of the program (argv[0])
15  */
16 void usage(char * name);
17
18 /**
19  * Test program OpenCV2 + QT5
20  * @param argc argument count
21  * @param argv argument values
22  * @return QTApp return value
23  * @par usage : <Progname> [--device | -d] <#> | [--file | -f ] <filename>
24  * [--mirror | -m] [--gray | -g] [--size | -s] <width>x<height>
25  *  - device : [--device | -d] <device #> (0, 1, ...) Opens capture device #
26  *  - filename : [--file | -f ] <filename> Opens a video file or URL (including rtsp)
27  *  - mirror : mirrors image horizontally before display
28  *  - gray : turns on source image gray conversion
29  *  - size : [--size | -s] <width>x<height> resize capture to fit desired <width>
30  *  and <height>
31  */
32 int main(int argc, char *argv[])
33 {
34     // ----------------------------------------------------------------------
35     // Instanciate QApplication to receive special QT args
36     // ----------------------------------------------------------------------
37     QApplication app(argc, argv);
38
39     // Gets arguments after QT specials removed
40     QStringList argList = QCoreApplication::arguments();
41
42     int threadNumber = 3;
43     // parse arguments for --threads tag
44     for (QListIterator<QString> it(argList); it.hasNext(); )
45     {
46         QString currentArg(it.next());
47
48         if (currentArg ≡ "-t" ∨ currentArg ≡"--threads")
49         {
50             // Next argument should be thread number integer
51             if (it.hasNext())
52             {
53                 QString threadString(it.next());
54                 bool convertOk;
55                 threadNumber = threadString.toInt(&convertOk,10);
56                 if (¬convertOk ∨ threadNumber < 1 ∨ threadNumber > 3)
57                 {
58                     qWarning("Warning: Invalid thread number %d",threadNumber);
59                     threadNumber = 3;
60                 }
61             }
62             else
63             {
64                 qWarning("Warning: thread tag found with no following thread number");
65             }
66         }
67     }
68     // ----------------------------------------------------------------------
69     // Create Capture factory using program arguments and
70     // open Video Capture
71     // ----------------------------------------------------------------------
72     CaptureFactory factory(argList);
73     factory.setSkippable(true);
74
75     // Helper thread for capture
76     QThread * capThread = NULL;
77     if (threadNumber > 1)
78     {
79         capThread = new QThread();
80     }
81
82     // Capture
```

```
83      QcvVideoCapture * capture = factory.getCaptureInstance(capThread);
84
85      // ------------------------------------------------------------------------
86      // Create QHistandLUT
87      // ------------------------------------------------------------------------
88      // Helper thread for processor
89      QThread * procThread = NULL;
90      if (threadNumber > 2)
91      {
92          procThread = new QThread();
93      }
94      else
95      {
96          if (threadNumber > 1)
97          {
98              procThread = capThread;
99          }
100     }
101
102     // Processsor
103     QcvHistograms * histograms = NULL;
104     if (procThread ≡ NULL)
105     {
106         histograms = new QcvHistograms(capture→getImage());
107     }
108     else
109     {
110         if (procThread ≠ capThread)
111         {
112             histograms = new QcvHistograms(capture→getImage(),
113                                            capture→getMutex(),
114                                            procThread);
115         }
116         else // procThread == capThread
117         {
118             histograms = new QcvHistograms(capture→getImage(),
119                                            NULL,
120                                            procThread);
121         }
122     }
123
124     // ------------------------------------------------------------------------
125     // Connects capture to Histograms
126     // ------------------------------------------------------------------------
127     // Connects capture update to QHistandLUT update
128     QObject::connect(capture, SIGNAL(updated()),
129                      histograms, SLOT(update()));
130
131     // connect capture changed image to QHistandLUT set input
132     QObject::connect(capture, SIGNAL(imageChanged(Mat*)),
133                      histograms, SLOT(setSourceImage(Mat*)));
134     // ------------------------------------------------------------------------
135     // Now that Capture & QHistandLUT are on then
136     // add our MainWindow as toplevel
137     // and launches app
138     // ------------------------------------------------------------------------
139     MainWindow w(capture, histograms);
140     w.show();
141
142     usage(argv[0]);
143
144     int retVal = app.exec();
145
146     // ------------------------------------------------------------------------
147     // Cleanup & return
148     // ------------------------------------------------------------------------
149     delete histograms;
150     delete capture;
151     qDebug() << "Processor and Capture deleted";
152     bool sameThread = capThread ≡ procThread;
153
154     if (capThread ≠ NULL)
155     {
156         delete capThread;
157         qDebug() << "Capture Thread deleted";
158     }
159
160     if (procThread ≠ NULL ∧ ¬sameThread)
161     {
162         delete procThread;
163         qDebug() << "Processor Thread deleted";
164     }
```



```
165
166     return retVal;
167 }
168
169 /*
170  * Usage function shown just before launching QApp
171  * @param name the name of the program (argv[0])
172  */
173 void usage(char * name)
174 {
175     cout << "usage : " << basename(name) << " "
176          << "[−d | −−device] <device number> "
177          << "[−v | −−video] <video file> "
178          << "[−s | −−size] <width>x<height> "
179          << "[−m | −−mirror]"
180          << "[−t | −−threads] <number of threads [1..3]>"
181          << endl;
182 }
```