```cpp
/*
 * 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
         */
```

```cpp
        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_ */
```

```cpp
/*
 * 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
```

```cpp
 */
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)
        {
            cerr << "CvProcessor::addImage(\"" << name
                << "\",...) : already added" << endl;
        }

        retValue = false;
    }
    else
    {
        retValue = true;
    }

    return retValue;
}
/*
 * 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.
 */
//void CvProcessor::updateImage(const char * name, Mat * image)
//{
//  // Search for this name in the map
//  map<string, Mat*>::iterator it;
//  for (it = images.begin(); it != images.end(); ++it)
//  {
//      if (it->first == name)
//      {
//          (it->second->release();
//          images.erase(it);
//      }
//  }
//
//  string sname(name);
//
//  updateImage(sname, 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.
 */
//void CvProcessor::updateImage(const string & name, const Mat & image)
//{
//  clog << "update image " << name << " with " << (long) &image << endl;
//  images.erase(name);
//
//  addImage(name, 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 & CvProcessor::getImage(const char *name) const
    throw (CvProcessorException)
{
    string sname(name);

    return getImage(sname);
}

/*
 * 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
 */
const Mat & CvProcessor::getImage(const string & name) const
    throw (CvProcessorException)
{
    // Search for this name
    map<string, Mat*>::const_iterator cit;
    for (cit = images.begin(); cit ≠ images.end(); ++cit)
```

```
    {
        if (cit→first ≡ name)
        {
            if (cit→second→data ≡ NULL)
            {
                // image contains no data
                throw CvProcessorException(CvProcessorException::NULL_DATA,
                                           name.c_str());
            }
            return *(cit→second);
        }
    }

    // not found : throw exception
    throw CvProcessorException(CvProcessorException::INVALID_NAME,
                               name.c_str());
}

/*
 * 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 * CvProcessor::getImagePtr(const char *name)
    throw (CvProcessorException)
{
    string sname(name);

    return getImagePtr(sname);
}

/*
 * 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 * CvProcessor::getImagePtr(const string & name)
    throw (CvProcessorException)
{
    // Search for this name
    map<string, Mat*>::const_iterator cit;
    for (cit = images.begin(); cit ≠ images.end(); ++cit)
    {
        if (cit→first ≡ name)
        {
            if (verboseLevel ≥ VERBOSE_ACTIVITY)
            {
                clog << "getImagePtr(" << name << "): returning : "
                     << (long) (cit→second) << endl;
            }
            return cit→second;
        }
    }

    // not found : throw exception
    throw CvProcessorException(CvProcessorException::INVALID_NAME, name.c_str());
}

/*
 * Number of channels in source image
 * @return the number of channels of source image
 */
int CvProcessor::getNbChannels() const
{
    return nbChannels;
}

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

```

```cpp
/*
 * 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;
}

```

```
#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;
}
```

```cpp
/*
 * 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();
```

```cpp

        /**
         * 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);

        /**
         * 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 = defaultTimeOut);

};

#endif /* QCVPROCESSOR_H_ */
```

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

#include <QRegExpValidator>
#include <QDebug>
#include <cstring>      // for strcpy
#include "QcvProcessor.h"

/*
 * Default timeout to show messages
 */
int QcvProcessor::defaultTimeOut = 5000;

/*
 * Number format used to format numbers into QStrings
 */
char QcvProcessor::numberFormat[10] = {"%8.1f ms"};

/*
 * The regular expression used to validate new number formats
 * @see #setNumberFormat
 */
QRegExp QcvProcessor::numberRegExp("%[+- 0#]*[0-9]*([.][0-9]+)?[efEF]");

/*
 * 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::QcvProcessor(Mat * image,
                           QMutex * imageLock,
                           QThread * updateThread,
                           QObject * parent) :
    CvProcessor(image), // <-- virtual base class constructor first
    QObject(parent),
    sourceLock(imageLock),
    updateThread(updateThread),
    message(),
    processTimeString()
{
    if (updateThread ≠ NULL)
    {
        this→moveToThread(updateThread);

        connect(this, SIGNAL(finished()), updateThread, SLOT(quit()),
                Qt::DirectConnection);

        updateThread→start();
    }
}

/*
 * QcvProcessor destructor
 */
QcvProcessor::~QcvProcessor()
{
    // Lock might be already destroyed in source object so don’t try to unlock

    message.clear();
    processTimeString.clear();

    emit finished();

    if (updateThread ≠ NULL)
    {
        // Wait until update thread has received the "finished" signal through
        // "quit" slot
        updateThread→wait();
    }
}

/*
 * Sets new number format
 * @param format the new number format
 */
```

```
void QcvProcessor::setNumberFormat(const char * format)
{
    /*
     * The format string should validate the following regex
     * %[+− 0#]*[0−9]*([.][0−9]+)?[efEF]
     */
    QRegExpValidator validator(numberRegExp, NULL);

    QString qFormat(format);
    int pos = 0;
    if ((validator.validate(qFormat,pos) ≡ QValidator::Acceptable) ∧
        (strlen(format) ≤ 10))
    {
        strcpy(numberFormat, format);
    }
    else
    {
        qWarning("QcvProcessor::setNumberFormat(%s) : invalid format", format);
    }
}


/*
 * Update computed images slot and sends updated signal
 * required
 */
void QcvProcessor::update()
{
    /*
     * Important note : CvProcessor::update() should NOT be called here
     * since it should be called in QcvXXXprocessor subclasses such that
     * QcvXXXProcessor::update method should contain :
     *  − call to CvXXXProcessor::update() (not QCvXXXProcessor)
     *  − emit signals from QcvXXXProcessor
     *  − call to QcvProcessor::update() (this method)
     */
    emit updated();
    processTimeString.sprintf(numberFormat, getProcessTime(0) / 1000.0);
    emit processTimeUpdated(processTimeString);
}

/*
 * Changes source image slot.
 * Attributes needs to be cleaned up then set up again
 * @param image the new source Image
 * @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
 */
void QcvProcessor::setSourceImage(Mat *image)
    throw (CvProcessorException)
{
    if (verboseLevel ≥ VERBOSE_NOTIFICATIONS)
    {
        clog << "QcvProcessor::setSourceImage(" << (ulong) image << ")" << endl;
    }

    Size previousSize(sourceImage→size());
    int previousNbChannels(nbChannels);

    if (sourceLock ≠ NULL)
    {
        sourceLock→lock();
        // qDebug() << "QcvProcessor::setSourceImage: lock";
    }

    CvProcessor::setSourceImage(image);

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

    emit imageChanged(sourceImage);

    emit imageChanged();

    if ((previousSize.width ≠ image→cols) ∨
        (previousSize.height ≠ image→rows))
```

```
    {
        emit imageSizeChanged();
    }

    if (previousNbChannels ≠ nbChannels)
    {
        emit imageColorsChanged();
    }

    // Force update
    update();
}

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

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

#ifndef CVDFT_H_
#define CVDFT_H_

#include <vector>
using namespace std;

#include <cv.h>
using namespace cv;

#include "CvProcessor.h"

/**
 * Class to compute DFT on input image
 */
class CvSimpleDFT : virtual public CvProcessor
{
    public:
        /**
         * Minimum log scale factor.
         * Default value is 5.
         */
        static const double minLogScaleFactor;

        /**
         * Maximum log scale factor.
         * Default value is 20 or 30.
         */
        static const double maxLogScaleFactor;

    protected:
        /**
         * Minimum of source image rows & cols for cropping source
         */
        int minSize;

        /**
         * Maximum of source image rows & cols for cropping source
         */
        int maxSize;

        /**
         * Border size to crop on source image
         */
        int borderSize;

        /**
         * DFT optimal size
         */
        int optimalDFTSize;

        /**
         * Optimal Fourier size
         */
        Size dftSize;

        /**
         * Input frame cropped to square size for FFT: CV_8UC<nbChannels>
         */
        Mat inFrameSquare;

        /**
         * Input frame cropped color channels: CV_8UC1 x <nbChannels>
         */
        vector<Mat> channels;

        /**
         * Input frame square channels converted to doubles: CV_64FC1 x <nbChannels>
         */
        vector<Mat> channelsDouble;

        /**
         * Input frame square channels complex channels:
         * CV_64FC1 x 2 x <nbChannels>
         */
        vector<vector<Mat> > channelsDoubleComplexComponents;
```

```

        /**
         * Input frame square complex image: CV_64FC2 x <nbChannels>
         */
        vector<Mat> channelsComplexImages;

        /**
         * Complex spectrum images: CV_64FC2 x <nbChannels>
         */
        vector<Mat> channelsComplexSpectrums;

        /**
         * Complex spectrum channels: CV_64FC1 x 2 x <nbChannels>
         */
        vector<vector<Mat> > channelsComplexSpectrumComponents;

        /**
         * Spectrum magnitude: CV_64FC1 x <nbChannels>
         */
        vector<Mat> channelsSpectrumMagnitude;

        /**
         * LogScale factor to apply on log magnitude to show spectrum.
         */
        double logScaleFactor;

        /**
         * log spectrum magnitude: CV_64FC1 x <nbChannels>
         */
        vector<Mat> channelsSpectrumLogMagnitude;

        /**
         * [Log] spectrum magnitude channels converted for display:
         * CV_8UC1 x <nbChannels>
         */
        vector<Mat> channelsSpectrumLogMagnitudeDisplay;

        /**
         * [Log] spectrum magnitude image converted for display:
         * CV_8UC<nbChannels>
         */
        Mat spectrumMagnitudeImage;

    public:
        /**
         * DFT processor constructor
         * @param sourceImage the source image
         * @pre source image is not NULL
         */
        CvSimpleDFT(Mat * sourceImage);

        /**
         * DFT Processor destructor
         */
        virtual ~CvSimpleDFT();

        /**
         * DFT Update.
         * Steps in update
         *  - crop source image to a square according to optima FFT size
         *  - split in frame square into color channels
         *  - converts these color channels to double
         *  - apply frequency shift on double channels to
         *      - produce the shifted real component of source channels
         *      - produce later a spectrum with low frequencies at image center
         *  - merge real/image channels into complex image per channel
         *  - compute dft on each channel
         *  - split channels complex spectrum in to real/imag components
         *  - compute channels spectrum magnitude from real/imag components
         *  - log scale channels spectrum magnitude
         *  - converts channels log magnitude for display
         */
        virtual void update();

        // --------------------------------------------------------------------
        // Options settings and gettings
        // --------------------------------------------------------------------

        /**
         * Optimal dft size for current source image
         * @return the current optimal dft size
         */
        int getOptimalDftSize() const;

        /**
         * Get current log scale factor
         * @return the current log scale factor
         */
        double getLogScaleFactor() const;

        /**
         * Setting the log scale factor
         * @param logScaleFactor the new log scale factor
         */
        virtual void setLogScaleFactor(double logScaleFactor);

    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
         */
        void setup(Mat * sourceImage, bool fullSetup = true);

        /**
         * Clean up internal atrtibutes before changing source image or
         * cleaning up class before destruction
         */
        void cleanup();

        // ----------------------------------------------------------------------
        // Utility methods
        // ----------------------------------------------------------------------
        /**
         * Modify image to obtain reverse frequencies on the Fourier transform
         * (low frequencies at the center of the image and high frequencies on
         * the border), or modify image obtained from reverse Fourier transform
         * with reversed frequencies.
         * @param imgIn source image
         * @param imgOut destination image
         * @par Algorithm:
         * This is based on the following property of the Z transform :
         * \f[
         *  TZ\left\{a^{k} x_{k}\right\} = X\left(\frac{z}{a}\right)
         * \f]
         * if \f$y_{k} = (-1)^{k} x_{k}\f$ then \f$Y(z) = X(-z)\f$
         * which can be explained in Fourier space by replacing
         * \f$z\f$ by \f$e^{j 2 \pi F}\f$:
         * \f[
         * Y\left[e^{j 2 \pi F}\right] = X\left[-e^{j2\pi F}\right] =
         * X\left[e^{j\pi}e^{j2\pi F}\right] =
         * X\left[e^{j2\pi\left(F + \frac{1}{2}\right)}\right]
         * \f]
         * hence
         * \f[
         * Y(F) = X\left(F + \frac{1}{2}\right)
         * \f]
         * or
         * \f[
         * Y(f) = X\left(f + \frac{f_{e}}{2}\right)
         * \f]
         * where \f$f_{e}\f$ is the sampling frequency, which means the
         * resulting Fourier transform will present an \f$\frac{f_{e}}{2}\f$
         * frequency offset. And since the sampling frequency lies in the middle
         * of the spectrum in the DFT. Low frequencies will appear centered
         * around the middle of the spectrum.
         *
         * In 2D the algorithm is the following:
         * \f[
         * imgOut(i,j) = (-1)^{i+j} \cdot imgIn(i,j)
         * \f]
         * \f$f_{e}\f$ is at the center of the spectrum image in 2D, which
         * means, low frequencies will be located at the center of the image.
         */
        template <typename T>
        void frequencyShift(Mat & imgIn, Mat & imgOut);

        /**
         * Log scale T valued image
```

```
         * @param imgIn input image
         * @param imgOut output image
         * @param scaleFactor such as
         * \f$ imgOut = scaleFactor \times \log(1 + imgIn)\f$
         */
        template <typename T>
        void logScaleImg(const Mat & imgIn, Mat & imgOut, const T scaleFactor);
};

#endif /* CVDFT_H_ */
```

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

#include <limits>
#include <cmath>

//#include <iostream>
//using namespace std;

#include "CvSimpleDFT.h"

/*
 * Minimum log scale factor.
 * Default value is 5.
 */
const double CvSimpleDFT::minLogScaleFactor = 5.0;

/*
 * Maximum log scale factor.
 * Default value is 20.
 */
const double CvSimpleDFT::maxLogScaleFactor = 30.0;

/*
 * DFT processor constructor
 * @param sourceImage the source image
 */
CvSimpleDFT::CvSimpleDFT(Mat * sourceImage) :
    CvProcessor(sourceImage),
    minSize(MIN(sourceImage→rows, sourceImage→cols)),
    maxSize(MAX(sourceImage→rows, sourceImage→cols)),
    borderSize((maxSize−minSize)/2),
    optimalDFTSize(getOptimalDFTSize(minSize)),
    dftSize(optimalDFTSize, optimalDFTSize),
    inFrameSquare(dftSize, type),
    logScaleFactor(10.0),
    spectrumMagnitudeImage(dftSize, type)
{
    setup(sourceImage, false);

    addImage("square", &inFrameSquare);
    addImage("spectrum", &spectrumMagnitudeImage);
}

/*
 * DFT Processor destructor
 */
CvSimpleDFT::~CvSimpleDFT()
{
    cleanup();
}

/*
 * Setup internal attributes according to source image
 * @param sourceImage a new source image
 * @param fullSetup full setup is needed when source image is changed
 */
void CvSimpleDFT::setup(Mat *sourceImage, bool fullSetup)
{
    // Full setup starting point (already performed in constructor)
    if (fullSetup)
    {
        CvProcessor::setup(sourceImage, fullSetup);
        minSize = MIN(sourceImage→rows, sourceImage→cols);
        maxSize = MAX(sourceImage→rows, sourceImage→cols);
        borderSize = (maxSize−minSize)/2;
        optimalDFTSize = getOptimalDFTSize(minSize);
        dftSize.height = optimalDFTSize;
        dftSize.width = optimalDFTSize;
        inFrameSquare = Mat(dftSize, type);
//      logScaleFactor = 10.0;
        spectrumMagnitudeImage = Mat(dftSize, type);
    }

    // Partial setup starting point
    for (int i=0; i < nbChannels; i++)
    {
        channels.push_back(Mat(dftSize, CV_8UC1));
```

```cpp
        channelsDouble.push_back(Mat(dftSize, CV_64FC1));
        channelsDoubleComplexComponents.push_back(vector<Mat>());
        channelsComplexImages.push_back(Mat(dftSize, CV_64FC2));
        channelsComplexSpectrums.push_back(Mat(dftSize, CV_64FC2));
        channelsComplexSpectrumComponents.push_back(vector<Mat>());
        channelsSpectrumMagnitude.push_back(Mat(dftSize, CV_64FC1));
        channelsSpectrumLogMagnitude.push_back(Mat(dftSize, CV_64FC1));
        channelsSpectrumLogMagnitudeDisplay.push_back(Mat(dftSize, CV_8UC1));

        // complex channels
        for (int j=0; j < 2; j++)
        {
            channelsDoubleComplexComponents[i].push_back(Mat(dftSize, CV_64FC1));
            channelsComplexSpectrumComponents[i].push_back(Mat(dftSize, CV_64FC1));
        }

        // fill complex channels of channelsDoubleComplexComponents with 0
        channelsDoubleComplexComponents[i][1] = Scalar(0.0);
    }
}

void CvSimpleDFT::cleanup()
{
    for (int i=0; i < nbChannels; i++)
    {
        // complex channels
        for (int j=0; j < 2; j++)
        {
            channelsComplexSpectrumComponents[i][j].release();
            channelsDoubleComplexComponents[i][j].release();
        }

        channelsSpectrumLogMagnitudeDisplay[i].release();
        channelsSpectrumLogMagnitude[i].release();
        channelsSpectrumMagnitude[i].release();
        channelsComplexSpectrumComponents[i].clear();
        channelsComplexSpectrums[i].release();
        channelsComplexImages[i].release();
        channelsDoubleComplexComponents[i].clear();
        channelsDouble[i].release();
        channels[i].release();
    }

    channelsSpectrumLogMagnitudeDisplay.clear();
    channelsSpectrumLogMagnitude.clear();
    channelsSpectrumMagnitude.clear();
    channelsComplexSpectrumComponents.clear();
    channelsComplexSpectrums.clear();
    channelsComplexImages.clear();
    channelsDoubleComplexComponents.clear();
    channelsDouble.clear();
    channels.clear();

    spectrumMagnitudeImage.release();
    inFrameSquare.release();

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

/*
 * Update
 */
void CvSimpleDFT::update()
{
//  clog << "CvSimpleDFT::update()" << endl;

    /*
     * Crop source image to center square and resize it to nearest
     * DFT optimal size
     * *sourceImage -> inFrameSquare
     */
    if (sourceImage→cols > sourceImage→rows)
    {
        // wider than high : resize a colRange(borderSize, borderSize + minSize)
        // of sourceImage to dftSize in inFrameSquare
        resize(sourceImage→colRange(borderSize, borderSize + minSize),
               inFrameSquare,
               dftSize,
               0,
               0,
               INTER_AREA);
    }
    else
    {
        // higher than wide : resize a rowRange(borderSize, borderSize + minSize)
        // of sourceImage to dftSize in inFrameSquare
        resize(sourceImage→rowRange(borderSize, borderSize + minSize),
               inFrameSquare,
               dftSize,
               0,
               0,
               INTER_AREA);
    }

    /*
     * Split input frame square to individual channels
     * inFrameSquare -> channels
     */
    // TODO Ã  complÃ©ter ...

    // Process each component (1 for gray images, 3 for color images)
    for (int i=0; i < nbChannels; i++)
    {
        /*
         * Fourier transform processing
         *  - Convert uchar center square image to CV_64F real component
         *  - perform frequency shift on real image to obtain low frequencies
         *      in the middle of the DFT image rather than in the corners
         *  - merge real & imag component to complexImage before DFT
         *      imag component could be filled with 0
         *  - compute DFT
         *  - split DFT channels
         *  - compute DFT magnitude from DFT channels
         *  - logScale magnitude with factor (5 to 20)
         *  - convertScaleAbs logMagnitude to CV_8UC1 to display image
         *
         */

        // convert component to double
        // channels[] -> channelsDouble
        // TODO Ã  complÃ©ter ...

        // Frequency shift channelsDouble to real complex component with
        // frequencyShift<double>(...)
        // Frequency shift allow to prepare spatial image components to
        // produce frequency image later with low frequencies in the center
        // of frequency image
        // channelsDouble[] -> channelsDoubleComplexComponents[][0]
        // TODO Ã  complÃ©ter ...
        // channelsDoubleComplexComponents[i][1] is already filled with 0 in
        // setup method so frequency shift is not necessary on imaginary part

        // Merge Real and Imaginary into a complex component image
        // channelsDoubleComplexComponents[] -> channelsComplexImages[]
        // TODO Ã  complÃ©ter ...

        // Perform Fourier transform (dft) on Complex component image
        // channelsComplexImages[] -> channelsComplexSpectrums[] with
        // DFT_COMPLEX_OUTPUT
        // TODO Ã  complÃ©ter ...

        // Split component Complex spectrum to real/imag channels
        // channelsComplexSpectrums[] -> channelsComplexSpectrumComponents[]
        // TODO Ã  complÃ©ter ...

        // Compute component spectrum magnitude
        // channelsComplexSpectrumComponents[][0 & 1] -> channelsSpectrumMagnitude[]
        // TODO Ã  complÃ©ter ...

        // Log scale magnitude with logScaleImg<double>(...) and logScaleFactor
        // channelsSpectrumMagnitude[] -> channelsSpectrumLogMagnitude[]
        // TODO Ã  complÃ©ter ...

        // Convert Log scale channels Spectrum to display channels
        // channelsSpectrumLogMagnitude[] -> channelsSpectrumLogMagnitudeDisplay[]
        // TODO Ã  complÃ©ter ...
    }

    // Merge channels spectrum Log magnitude to color spectrum image
    // channelsSpectrumLogMagnitudeDisplay -> spectrumMagnitudeImage
    // TODO Ã  complÃ©ter ...

}
```

```

/*
 * Optimal dft size for current source image
 * @return the current optimal dft size
 */
int CvSimpleDFT::getOptimalDftSize() const
{
    return optimalDFTSize;
}

/*
 * Get current log scale factor
 * @return the current log scale factor
 */
double CvSimpleDFT::getLogScaleFactor() const
{
    return logScaleFactor;
}

/*
 * Setting the log scale factor
 * @param logScaleFactor the new log scale factor
 */
void CvSimpleDFT::setLogScaleFactor(double logScaleFactor)
{
    if (logScaleFactor > maxLogScaleFactor)
    {
        this→logScaleFactor = maxLogScaleFactor;
    }
    else if (logScaleFactor < minLogScaleFactor)
    {
        this→logScaleFactor = minLogScaleFactor;
    }
    else
    {
        this→logScaleFactor = logScaleFactor;
    }
}

// ----------------------------------------------------------------------
// Utility methods
// ----------------------------------------------------------------------
/*
 * Modify image to obtain reverse frequencies on the Fourier transform
 * (low frequencies at the center of the image and high frequencies on
 * the border), or modify image obtained from reverse Fourier transform
 * with reversed frequencies.
 * @param imgIn source image
 * @param imgOut destination image
 */
template <typename T>
void CvSimpleDFT::frequencyShift(Mat & imgIn, Mat & imgOut)
{
    int i, j;

    for (i = 0; i < imgIn.rows; i++)
    {
        for (j = 0; j < imgIn.cols; j++)
        {
            imgOut.at<T> (i, j) = imgIn.at<T> (i, j) * (T)pow(−1.0, i + j);
        }
    }
}

/*
 * Log scale T valued image
 * @param imgIn input image
 * @param imgOut output image
 * @param scaleFactor such as
 * \f$ imgOut = scaleFactor \times \log(1 + imgIn)\f$
 */
template <typename T>
void CvSimpleDFT::logScaleImg(const Mat & imgIn, Mat & imgOut,
    const T scaleFactor)
{
    MatConstIterator_<T> inIt = imgIn.begin<T>();
    MatConstIterator_<T> inItEnd = imgIn.end<T>();
    MatIterator_<T> outIt = imgOut.begin<T>();
    MatIterator_<T> outItEnd = imgOut.end<T>();
    for (; inIt ≠ inItEnd ∧ outIt ≠ outItEnd; ++inIt, ++outIt)
    {
        (*outIt) = scaleFactor * (T)log(1.0 + (*inIt));
    }
}
```

```
/*
 * QcvSimpleDFT.h
 *
 *  Created on: 22 fÃ©vr. 2012
 *      Author: davidroussel
 */

#ifndef QCVDFT_H_
#define QCVDFT_H_

#include "QcvProcessor.h"
#include "CvSimpleDFT.h"

/**
 * Qt flavored Simple Fourier transform
 */
class QcvSimpleDFT: public QcvProcessor, public CvSimpleDFT
{
    Q_OBJECT

    public:

        /**
         * QcvSimpleDFT constructor
         * @param image the source image
         * @param imageLock the mutex on source image
         * @param updateThread the thread in which this processor runs
         * @param parent parent QObject
         */
        QcvSimpleDFT(Mat * image,
                     QMutex * imageLock = NULL,
                     QThread * updateThread = NULL,
                     QObject * parent = NULL);

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

        // --------------------------------------------------------------------
        // Options settings with message notification
        // --------------------------------------------------------------------

    public slots:
        /**
         * Update computed images slot and sends updated signal
         * 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 source image changes to adjust max filter sizes
//       */
//      void dftSizeChanged();

        /**
         * Signal sent when input dftSize square image has been reallocated
         * @param image the new in square image
         */
        void squareImageChanged(Mat * image);

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

        /**
         * Signal sent when inverse image has been reallocated
         * @param image the new inverse image
         */
        void inverseImageChanged(Mat * image);
```

```cpp
};

#endif /* QCVDFT_H_ */
```

```cpp
/*
 * QcvSimpleDFT.cpp
 *
 *  Created on: 22 fÃ©vr. 2012
 *      Author: davidroussel
 */

#include "QcvSimpleDFT.h"


/*
 * QcvSimpleDFT constructor
 * @param image the source image
 * @param imageLock the mutex on source image
 * @param updateThread the thread in which this processor runs
 * @param parent parent QObject
 */
QcvSimpleDFT::QcvSimpleDFT(Mat * image,
                           QMutex * imageLock,
                           QThread * updateThread,
                           QObject * parent) :
    CvProcessor(image), // <-- virtual base class constructor first
    QcvProcessor(image, imageLock, updateThread, parent),
    CvSimpleDFT(image)
{
}

/*
 * QcvSimpleDFT destructor
 */
QcvSimpleDFT::~QcvSimpleDFT()
{
    message.clear();
}

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

    /*
     * Update DFT images
     */
    CvSimpleDFT::update();

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

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

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

    QcvProcessor::setSourceImage(image);

    emit squareImageChanged(&inFrameSquare);

    emit spectrumImageChanged(&spectrumMagnitudeImage);

    if ((previousDftSize.width ≠ dftSize.width) ∨
        (previousDftSize.height ≠ dftSize.height))
    {
        emit imageSizeChanged();
        emit sendText(QString::number(optimalDFTSize));
    }

    // Force update
    update();
}
```

```
/*
 * 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
         */
```

```cpp
        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  /**
```

```cpp
//^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);
```

```cpp

    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)
```

```cpp
                    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();
```

```cpp
        bottom = p2.y();
    }
    else
    {
        top = p2.y();
        bottom = p1.y();
    }

    selectionRect.setLeft(left);
    selectionRect.setRight(right);
    selectionRect.setTop(top);
    selectionRect.setBottom(bottom);
}



/*
 * Widget minimum size is set to the contained image size
 * @return le size of the image within
 */
//QSize QcvMatWidget::minimumSize() const
//{
//  return sizeHint();
//}


/*
 * Size hint (because size depends on sourceImage properties)
 * @return size obtained from sourceImage
 */
QSize QcvMatWidget::sizeHint() const
{
    if (sourceImage ≠ NULL)
    {
        return QSize(sourceImage->cols, sourceImage->rows);
    }
    else
    {
        return defaultSize;
    }
}

/*
 * Gets Mat widget mouse clickable status
 * @return true if widget is sensitive to mouse click
 */
bool QcvMatWidget::isMouseClickable() const
{
    return (mouseSense & MOUSE_CLICK);
}

/*
 * Gets Mat widget mouse dragable status
 * @return true if widget is sensitive to mouse drag
 */
bool QcvMatWidget::isMouseDragable() const
{
    return (mouseSense & MOUSE_DRAG);
}

/*
 * Update slot customized to include convertImage before actually
 * updating
 */
void QcvMatWidget::update()
{
//  count++;
//  qDebug() << "QcvMatWidget::update " << count;
//  std::cerr << "{o";
    convertImage();
    QWidget::update();
//  std::cerr << "}";
}

// -----------------------------------------------------------------------------
// convertImage old algorithm
// -----------------------------------------------------------------------------
//  int cvIndex, cvLineStart;
//  // switch between bit depths
//  switch (displayImage.depth())
//  {
//      case CV_8U:
```

```
//          switch (displayImage.channels())
//          {
//              case 1: // Gray level images
//                  if ( (displayImage.cols != image.width()) ||
//                       (displayImage.rows != image.height()) )
//                  {
//                      QImage temp(displayImage.cols, displayImage.rows,
//                              QImage::Format_RGB32);
//                      image = temp;
//                  }
//                  cvIndex = 0;
//                  cvLineStart = 0;
//                  for (int y = 0; y < displayImage.rows; y++)
//                  {
//                      unsigned char red, green, blue;
//                      cvIndex = cvLineStart;
//                      for (int x = 0; x < displayImage.cols; x++)
//                      {
//                          // DO it
//                          red   = displayImage.data[cvIndex];
//                          green = displayImage.data[cvIndex];
//                          blue  = displayImage.data[cvIndex];
//
//                          image.setPixel(x, y, qRgb(red, green, blue));
//                          cvIndex++;
//                      }
//                      cvLineStart += displayImage.step;
//                  }
//                  break;
//              case 3: // BGR images (Regular OpenCV Color Capture)
//                  if ( (displayImage.cols != image.width()) ||
//                       (displayImage.rows != image.height()) )
//                  {
//                      QImage temp(displayImage.cols, displayImage.rows,
//                              QImage::Format_RGB32);
//                      image = temp;
//                  }
//                  cvIndex = 0;
//                  cvLineStart = 0;
//                  for (int y = 0; y < displayImage.rows; y++)
//                  {
//                      unsigned char red, green, blue;
//                      cvIndex = cvLineStart;
//                      for (int x = 0; x < displayImage.cols; x++)
//                      {
//                          // DO it
//                          red   = displayImage.data[cvIndex + 2];
//                          green = displayImage.data[cvIndex + 1];
//                          blue  = displayImage.data[cvIndex + 0];
//
//                          image.setPixel(x, y, qRgb(red, green, blue));
//                          cvIndex += 3;
//                      }
//                      cvLineStart += displayImage.step;
//                  }
//                  break;
//              default:
//                  printf("This number of channels is not supported\n");
//                  break;
//          }
//          break;
//      default:
//          printf("This type of Image is not implemented in QcvMatWidget\n");
//          break;
//  }

```

```

#ifndef QCVMATWIDGETLABEL_H
#define QCVMATWIDGETLABEL_H

#include <cv.h>
#include <QLabel>

using namespace cv;

#include "QcvMatWidget.h"

/**
 * OpenCV Widget for QT with QImage display
 */
class QcvMatWidgetLabel : public QcvMatWidget
{
    private:
        /**
         * The Image Label
         */
        QLabel * imageLabel;

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

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

        /**
         * OpenCV Widget destructor.
         */
        virtual ~QcvMatWidgetLabel(void);

    protected:
        /**
         * Widget setup
         * @pre imageLabel has been allocated
         * @post imageLabel has been added to the layout
         */
        void setup();

        /**
         * paint event reimplemented to draw content
         * @param event the paint event
         * @pre imageLabel has been allocated
         * @post displayImage has been set as pixmap of the imageLabel
         */
        void paintEvent(QPaintEvent * event);

};

#endif //QCVMATWIDGETLABEL_H
```

```cpp
//#include <iostream>
#include <QtDebug>
#include "QcvMatWidgetLabel.h"

using namespace std;

/*
 * OpenCV QT Widget default constructor
 * @param parent parent widget
 */
QcvMatWidgetLabel::QcvMatWidgetLabel(QWidget *parent,
                                     MouseSense mouseSense) :
    QcvMatWidget(parent, mouseSense),
    imageLabel(new QLabel())
{
    setup();
}

/*
 * OpenCV QT Widget constructor
 * @param the source OpenCV qImage
 * @param parent parent widget
 */
QcvMatWidgetLabel::QcvMatWidgetLabel(Mat * sourceImage,
                                     QWidget *parent,
                                     MouseSense mouseSense) :
    QcvMatWidget(sourceImage, parent, mouseSense),
    imageLabel(new QLabel())
{
    setup();
}

/*
 * Widget setup
 * @pre imageLabel has been allocated
 */
void QcvMatWidgetLabel::setup()
{
    layout→addWidget(imageLabel,0,Qt::AlignCenter);
}

/*
 * OpenCV Widget destructor.
 */
QcvMatWidgetLabel::~QcvMatWidgetLabel(void)
{
    delete imageLabel;
}

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

    QcvMatWidget::paintEvent(event);

    if (displayImage.data ≠ NULL)
    {
        // Builds Qimage from RGB image data
        // and sets image as Label pixmap
        imageLabel→setPixmap(QPixmap::fromImage(QImage((uchar *) displayImage.data,
                                                       displayImage.cols,
                                                       displayImage.rows,
                                                       displayImage.step,
                                                       QImage::Format_RGB888)));
    }
    else
    {
        qWarning("QcvMatWidgetLabel::paintEvent : image.data is NULL");
    }
}
```

```cpp
/*
 * 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:
        /**
```

```cpp
	 * 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();
}
```

```
/*
 * 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_ */
```

```
/*
 * 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);
}
```

```cpp
/*
 * 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;

        /**
```

```cpp
         * Image Matrix to obtain from capture
         */
        Mat image;

        /**
         * image resized (if required)
         */
        Mat imageResized;

        /**
         * [resized] image flipped (if required)
         */
        Mat imageFlipped;

        /**
         * Image converted for display:
         *  - scaled
         *  - flipped horizontally
         *  - converted to gray
         */
        Mat imageDisplay;

        /**
         * Live video indication (from cam)
         */
        bool liveVideo;

        /**
         * flipVideo to mirror image
         */
        bool flipVideo;

        /**
         * scale image to preferred width and height
         */
        bool resize;

        /**
         * scaling is performed into capture rather than through cv::resize
         * function
         */
        bool directResize;

        /**
         * image converted to gray
         */
        bool gray;

        /**
         * Allow capture to skip an image capture when lock can’t be acquired
         * before grabbing a new image. Otherwise we’ll wait until the lock
         * is acquired before grabbing an new image. The lock might be acquired
         * by another lenghty thread/processor during image processing.
         */
        bool skip;

        /**
         * Current Image size (might be different from natural capture image
         * size)
         */
        QSize size;

        /**
         * Capture natural image size (without resizing)
         */
        QSize originalSize;

        /**
         * Capture frame rate obtained either by getting the CV_CAP_PROP_FPS
         * VideoCapture property or by computing capture time on several images
         * @see #grabInterval
         */
        double frameRate;

        /**
         * default time interval between refresh
         */
        static int defaultFrameDelay;

        /**
         * Number of frames to test frame rate
         */
```

```
		static size_t defaultFrameNumberTest;

		/**
		 * Status message to send when something changes
		 */
		QString statusMessage;

		/**
		 * Default message showing time (at least 2000 ms)
		 */
		static int messageDelay;

	public:
		/**
		 * QcvVideoCapture constructor.
		 * Opens the default camera (0)
		 * @param flipVideo mirror image status
		 * @param gray convert image to gray status
		 * @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
		 * otherwise capture is updated in the current thread.
		 * @param updateThread the thread used to run this capture
		 * @param parent the parent QObject
		 */
		QcvVideoCapture(const bool flipVideo = false,
						const bool gray = false,
						const bool skip = true,
						const unsigned int  width = 0,
						const unsigned int height = 0,
						QThread * updateThread = NULL,
						QObject * parent = NULL);

		/**
		 * QcvVideoCapture constructor with device Id
		 * @param deviceId the id of the camera 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(const int deviceId,
						const bool flipVideo = false,
						const bool gray = false,
						const bool skip = true,
						const unsigned int  width = 0,
						const unsigned int height = 0,
						QThread * updateThread = NULL,
						QObject * parent = NULL);

		/**
		 * 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(const QString & fileName,
						const bool flipVideo = false,
						const bool gray = false,
						const bool skip = true,
						const unsigned int  width = 0,
						const unsigned int height = 0,
						QThread * updateThread = NULL,
						QObject * parent = NULL);

```

```
		/**
		 * QcvVideoCapture destructor.
		 * releases video capture and image
		 */
		virtual ~QcvVideoCapture();

		/**
		 * Size accessor
		 * @return the image size
		 */
		const QSize & getSize() const;

		/**
		 * Gets resize state.
		 * @return true if imageDisplay have been resized to preferred width and
		 * height, false otherwise
		 */
		bool isResized() const;

		/**
		 * Gets direct resize state.
		 * @return true if image can be resized directly into capture.
		 * @note direct resize capabilities are tested into #grabTest which is
		 * called in all constructors. So #isDirectResizeable should not be
		 * called before #grabTest
		 */
		bool isDirectResizeable() const;

		/**
		 * Gets video flipping status
		 * @return flipped video status
		 */
		bool isFlipVideo() const;

		/**
		 * Gets video gray converted status
		 * @return the converted to gray status
		 */
		bool isGray() const;

		/**
		 * Gets the image skipping policy
		 * @return true if new image can be skipped when previous one has not
		 * been processed yet, false otherwise.
		 */
		bool isSkippable() const;

		/**
		 * Gets the current frame rate
		 * @return the current frame rate
		 */
		double getFrameRate() const;

		/**
		 * Image accessor
		 * @return the image to display
		 */
		Mat * getImage();

		/**
		 * The source image mutex
		 * @return  the mutex used on image access
		 */
		QMutex * getMutex();

	public slots:
		/**
		 * Open new device Id
		 * @param deviceId device number to open
		 * @param width desired width or 0 to keep capture width
		 * @param height desired height or 0 to keep capture height
		 * @return true if device has been opened and checked and timer launched
		 */
		bool open(const int deviceId,
				  const unsigned int width = 0,
				  const unsigned int height = 0);

		/**
		 * Open new video file
		 * @param fileName video file to open
		 * @param width desired width or 0 to keep capture width
		 * @param height desired height or 0 to keep capture height
```

```
         * @return true if video has been opened and timer launched
         */
        bool open(const QString & fileName,
                  const unsigned int width = 0,
                  const unsigned int height = 0);
        /**
         * Sets video flipping
         * @param flipVideo flipped video or not
         */
        void setFlipVideo(const bool flipVideo);

        /**
         * Sets video conversion to gray
         * @param grayConversion the gray conversion status
         */
        void setGray(const bool grayConversion);

        /**
         * Sets #imageDisplay size according to preferred width and height
         * @param size new desired size to set
         * @param alreadyLocked mutex lock has already been aquired so setSize does not have
         * to acquire the lock
         * @pre a first image have been grabbed
         */
        void setSize(const QSize & size);

    protected:
        /**
         * Performs a grab test to fill #image.
         * if capture is opened then tries to grab and if grab succeeds then
         * tries to retrieve image from grab and sets image size.
         * @return true if capture is opened and successfully grabbed a first
         * frame into #image, false otherwise
         * @post Moreover this method determines if direct resizing is allowed
         * on this capture instance by trying to set
         * CV_CAP_PROP_FRAME_WIDTH and CV_CAP_PROP_FRAME_HEIGHT.
         */
        bool grabTest();

        /**
         * Get or compute interval between two frames in ms and sets the
         * frameRate attribute.
         * Tries to get CV_CAP_PROP_FPS from capture and if not available
         * computes times between frames by grabbing defaultNumberTest images
         * @return interval between two frames
         * @param message message passed to grabInterval and display ahead of
         * the framerate computed during grabInterval
         * @pre capture is already instanciated
         * @post message indicating frame rate has been emitted and interval
         * between two frames has been returned
         */
        int grabInterval(const QString & message);

        /**
         * Sets #imageDisplay size according to preferred width and height
         * @param width desired width
         * @param height desired height
         * @pre a first image have been grabbed
         */
        void setSize(const unsigned int width,
                     const unsigned int height);

        /**
         * Tries to set capture size directly on capture by setting properties.
         *  − CV_CAP_PROP_FRAME_WIDTH to set frame width
         *  − CV_CAP_PROP_FRAME_HEIGHT to set frame height
         * @param width the width property to set on capture
         * @param height the height property to set on capture
         * @return true if capture is opened and if width and height have been
         * set successfully through @code capture.set(...) @endcode. Returns
         * false otherwise.
         * @post if at least width or height have been set successfully, capture
         * image is released then updated again so it will have the right
         * dimensions.
         * @warning if mutex lock can’t be obtained to ensure atomic access to
         * capture object, then we start recursing until we obtain that lock,
         * which is gross and should be fixed !!!
         */
        bool setDirectSize(const unsigned int width, const unsigned int height);

    protected slots:
        /**
```

```
         * update slot trigerred by timer : Grabs a new image and sends updated()
         * signal iff new image has been grabbed, otherwise there is no more
         * images to grab so kills timer.
         * @note If lock on OpenCV capture object can not be obtained then
         * capture is skipped. This is not critical since update is called
         * regularly by the #timer, so we’ll try updating image next time.
         */
        void update();

    signals:
        /**
         * Signal emitted when a new image has been grabbed
         */
        void updated();

        /**
         * Signal emitted when capture is released
         */
        void finished();

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

        /**
         * Signal to send when image has changed after opening new device or
         * setting new display size
         * @param image the new image to send
         */
        void imageChanged(Mat * image);
};

#endif /* QCVVIDEOCAPTURE_H_ */
```

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

#include <QElapsedTimer>
#include <QMutexLocker>
#include <QDebug>

#include "QcvVideoCapture.h"

#include <opencv2/imgproc/imgproc.hpp>

/*
 * default time interval between refresh
 */
int QcvVideoCapture::defaultFrameDelay = 33;

/*
 * Number of frames to test frame rate
 */
size_t QcvVideoCapture::defaultFrameNumberTest = 5;

/*
 * Default message showing time (at least 2000 ms)
 */
int QcvVideoCapture::messageDelay = 5000;

/*
 * QcvVideoCapture constructor.
 * Opens the default camera (0)
 * @param flipVideo mirror image status
 * @param gray convert image to gray status
 * @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
 * otherwise capture is updated in the current thread.
 * @param updateThread the thread used to run this capture
 * @param parent the parent QObject
 */
QcvVideoCapture::QcvVideoCapture(const bool flipVideo,
                                 const bool gray,
                                 const bool skip,
                                 const unsigned int width,
                                 const unsigned int height,
                                 QThread * updateThread,
                                 QObject * parent) :
    QcvVideoCapture(0, flipVideo, gray, skip, width, height, updateThread,
                    parent)
{
}

/*
 * QcvVideoCapture constructor with device Id
 * @param deviceId the id of the camera 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 int deviceId,
                                 const bool flipVideo,
                                 const bool gray,
                                 const bool skip,
                                 const unsigned int width,
                                 const unsigned int height,
                                 QThread * updateThread,
                                 QObject * parent) :
    QObject(parent),
    filename(),
    capture(deviceId),
```

```cpp
    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
    }

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

```cpp
    imageResized.release();
    image.release();
}

/*
 * Open new device Id
 * @param deviceId device number to open
 * @param width desired width or 0 to keep capture width
 * @param height desired height or 0 to keep capture height
 * @return true if device has been opened and checked and timer launched
 */
bool QcvVideoCapture::open(const int deviceId,
                           const unsigned int width,
                           const unsigned int height)
{
    if (updateThread ≠ NULL)
    {
        if (lockLevel ≡ 0)
        {
            mutex.lock();
            // qDebug() << "QcvVideoCapture::open(" << deviceId << "...): lock";
        }
        lockLevel++;
    }

    filename.clear();
    if (timer→isActive())
    {
        timer→stop();
        qDebug("timer stopped");
    }

    if (capture.isOpened())
    {
        capture.release();
    }

    if (¬image.empty())
    {
        image.release();
    }

    capture.open(deviceId);

    bool grabbed = grabTest();

    if (grabbed)
    {
        setSize(width, height);

        statusMessage.clear();
        statusMessage.append("Camera ");
        statusMessage.append(QString::number(deviceId));
        statusMessage.append(" ");
        int delay = grabInterval(statusMessage);
        timer→start(delay);
        liveVideo = true;
        qDebug("timer started with %d ms delay", delay);

        // emit
        // message changed already emitted by grabInterval()
        emit imageChanged(&imageDisplay);

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

    return grabbed;
}

/*
 * Open new video file
 * @param fileName video file to open
 * @param width desired width or 0 to keep capture width
 * @param height desired height or 0 to keep capture height
 * @return true if video has been opened and timer launched
 */
bool QcvVideoCapture::open(const QString & fileName,
                           const unsigned int width,
                           const unsigned int height)
{
    filename = fileName;

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

    if (updateThread ≠ NULL)
    {
        if (lockLevel ≡ 0)
        {
            mutex.lock();
            // qDebug() << "QcvVideoCapture::open(" << fileName << "...): lock";
        }
        lockLevel++;
    }

    if (capture.isOpened())
    {
        capture.release();
    }

    if (¬image.empty())
    {
        image.release();
    }

    capture.open(fileName.toStdString());

    bool grabbed = grabTest();

    if (grabbed)
    {
        setSize(width, height);
        qDebug() << "open setSize done";
        statusMessage.clear();
        statusMessage.append("file ");
        statusMessage.append(fileName);
        statusMessage.append(" opened");

        int delay = grabInterval(statusMessage);
        timer→start(delay);
        liveVideo = false;
        qDebug("timer started with %d ms delay", delay);

        // emit changes
        // messageChanged already emitted by grabInterval
        emit imageChanged(&imageDisplay);

    }

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

    return grabbed;
}

/*
 * Size accessor
 * @return the image size
 */
const QSize & QcvVideoCapture::getSize() const
{
    return size;
}

/*
 * Sets #imageDisplay size according to preferred width and height
```

```cpp
 * @param width desired width
 * @param height desired height
 * @pre a first image have been grabbed
 */
void QcvVideoCapture::setSize(const unsigned int width,
                              const unsigned int height)
{
    if ((updateThread ≠ NULL))
    {
        if (lockLevel ≡ 0)
        {
            mutex.lock();
            // qDebug("QcvVideoCapture::setSize(%d, %d) locked", width, height);
        }
        lockLevel++;
    }

    unsigned int preferredWidth;
    unsigned int preferredHeight;

    // qDebug("QcvVideoCapture::setSize(%d, %d)", width, height);

    // if not empty then release it
    if (¬imageResized.empty())
    {
        imageResized.release();
    }

    if ((width ≡ 0) ∧ (height ≡ 0)) // reset to original size
    {
        if (directResize) // direct set size to original size
        {
            setDirectSize((unsigned int)originalSize.width(),
                          (unsigned int)originalSize.height());
            // image is updated into setDirectSize
        }
        preferredWidth = image.cols;
        preferredHeight = image.rows;

        resize = false;
        imageResized = image;
    }
    else // width != 0 or height != 0
    {
        if ((width ≡ (unsigned int)image.cols) ∧
            (height ≡ (unsigned int)image.rows)) // unchanged
        {
            preferredWidth = image.cols;
            preferredHeight = image.rows;
            imageResized = image;

            if (((int)preferredWidth ≡ originalSize.width()) ∧
                ((int)preferredHeight ≡ originalSize.height()))
            {
                resize = false;
            }
            else
            {
                resize = true;
            }
        }
        else // width or height have changed
        {
            /*
             * Resize needed
             */
            preferredWidth = width;
            preferredHeight = height;

            resize = true;

            if (directResize)
            {
                setDirectSize(preferredWidth, preferredHeight);
                imageResized = image;
            }
            else
            {
                imageResized = Mat(preferredHeight, preferredWidth, image.type());
            }
        }
    }
    if (updateThread ≠ NULL)
    {
        lockLevel−−;
        if (lockLevel ≡ 0)
        {
            // qDebug("QcvVideoCapture::setSize unlocked");
            mutex.unlock();
        }
    }

    qDebug("QcvVideoCapture resize is %s [%s]",
           (resize ? "ON" : "OFF"),
           (directResize ? "direct" : "soft"));

    size.setWidth(preferredWidth);
    size.setHeight(preferredHeight);
    statusMessage.clear();
    statusMessage.sprintf("Size set to %dx%d", preferredWidth, preferredHeight);
    emit messageChanged(statusMessage, messageDelay);


    /*
     * imageChanged signal is delayed until setGray is called into
     * setFlipVideo
     */
    // Refresh image chain
    setFlipVideo(flipVideo);
}

/*
 * Sets #imageDisplay size according to preferred width and height
 * @param size new desired size to set
 * @pre a first image have been grabbed
 */
void QcvVideoCapture::setSize(const QSize & size)
{
    setSize(size.width(), size.height());
}

/*
 * Sets video flipping
 * @param flipVideo flipped video or not
 */
void QcvVideoCapture::setFlipVideo(const bool flipVideo)
{
    bool previousFlip = this→flipVideo;
    this→flipVideo = flipVideo;

    if (updateThread ≠ NULL)
    {
        if (lockLevel ≡ 0)
        {
            mutex.lock();
            // qDebug() << "QcvVideoCapture::setFlipVideo(): lock";
        }
        lockLevel++;
    }

    if (¬imageFlipped.empty())
    {
        imageFlipped.release();
    }

    if (flipVideo)
    {
        imageFlipped = Mat(imageResized.size(), imageResized.type());
    }
    else
    {
        imageFlipped = imageResized;
    }

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

```

    if (previousFlip ≠ flipVideo)
    {
        statusMessage.clear();
        statusMessage.sprintf("flip video is %s", (flipVideo ? "on" : "off"));
        emit messageChanged(statusMessage, messageDelay);
        emit imageChanged(&imageDisplay);
    }

    /*
     * imageChanged signal is delayed until setGray is called
     */
    // refresh image chain
    setGray(gray);
}

/*
 * Sets video conversion to gray
 * @param grayConversion the gray conversion status
 */
void QcvVideoCapture::setGray(const bool grayConversion)
{
    bool previousGray = gray;

    gray = grayConversion;

    if (updateThread ≠ NULL)
    {
        if (lockLevel ≡ 0)
        {
            mutex.lock();
            // qDebug() << "QcvVideoCapture::setGray(): lock";
        }
        lockLevel++;
    }

    if (¬imageDisplay.empty())
    {
        imageDisplay.release();
    }

    if (gray)
    {
        imageDisplay = Mat(imageFlipped.size(), CV_8UC1);
    }
    else
    {
        imageDisplay = imageFlipped;
    }

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

    if (previousGray ≠ grayConversion)
    {
        statusMessage.clear();
        statusMessage.sprintf("gray video is %s", (gray ? "on" : "off"));
        emit messageChanged(statusMessage, messageDelay);
    }

    /*
     * In any cases emit image changed since
     *  - setSize may have been called
     *  - setFlipVideo may have been called
     */
    emit imageChanged(&imageDisplay);
}

/*
 * Gets resize state.
 * @return true if imageDisplay have been resized to preferred width and
 * height, false otherwise
 */
bool QcvVideoCapture::isResized() const
{
```

```cpp
    return resize;
}

/*
 * Gets direct resize state.
 * @return true if image can be resized directly into capture.
 * @note direct resize capabilities are tested into #grabTest which is
 * called in all constructors. So #isDirectResizeable should not be
 * called before #grabTest
 */
bool QcvVideoCapture::isDirectResizeable() const
{
    return directResize;
}

/*
 * Gets video flipping status
 * @return flipped video status
 */
bool QcvVideoCapture::isFlipVideo() const
{
    return flipVideo;
}

/*
 * Gets video gray converted status
 * @return the converted to gray status
 */
bool QcvVideoCapture::isGray() const
{
    return gray;
}

/*
 * Gets the image skipping policy
 * @return true if new image can be skipped when previous one has not
 * been processed yet, false otherwise.
 */
bool QcvVideoCapture::isSkippable() const
{
    return skip;
}


/*
 * Gets the current frame rate
 * @return the current frame rate
 */
double QcvVideoCapture::getFrameRate() const
{
    return frameRate;
}


/*
 * Image accessor
 * @return the image
 */
Mat * QcvVideoCapture::getImage()
{
    return &imageDisplay;
}

/*
 * The source image mutex
 * @return  the mutex used on image access
 */
QMutex * QcvVideoCapture::getMutex()
{
    return &mutex;
}


/*
 * Performs a grab test to fill #image
 * @return true if capture is opened and successfully grabs a first
 * frame into #image, false otherwise
 */
bool QcvVideoCapture::grabTest()
{
//  qDebug("Grab test");
    bool result = false;
```

```cpp

    if (capture.isOpened())
    {
#ifndef Q_OS_LINUX // V4L does not support these queries
        int capWidth = capture.get(CV_CAP_PROP_FRAME_WIDTH);
        int capHeight = capture.get(CV_CAP_PROP_FRAME_HEIGHT);

        qDebug("Capture grab test with %d x %d image", capWidth, capHeight);
#endif
        // grabs first frame
        if (capture.grab())
        {
            bool retrieved = capture.retrieve(image);
            if (retrieved)
            {
                size.setWidth(image.cols);
                size.setHeight(image.rows);
                originalSize.setWidth(image.cols);
                originalSize.setHeight(image.rows);

                /*
                 * Tries to determine if direct resizing in capture is possible
                 * by setting original size through properties
                 * Typically :
                 *  - camera capture might be resizable
                 *  - video file capture may not be resizable
                 */
                directResize = setDirectSize(image.cols, image.rows);

                qDebug("Capture direct resizing is %s",
                       (directResize ? "on" : "off"));

                result = true;
            }
            else
            {
                qFatal("Video Capture unable to retreive image");
            }
        }
        else
        {
            qFatal("Video Capture can not grab");
        }
    }
    else
    {
        qFatal("Video Capture is not opened");
    }

    return result;
}

/*
 * Get or compute interval between two frames
 * @return interval between two frames
 * @pre capture is already instanciated
 */
int QcvVideoCapture::grabInterval(const QString & message)
{
    int frameDelay = defaultFrameDelay;

    // Tries to get framerate from capture
    // ------------------------------------------------------------------
    // Caution : on some systems getting video parameters is forbidden !
    // For instance it does not work wirh linuxes equipped with V4L
    // ------------------------------------------------------------------
#ifndef Q_OS_LINUX
    frameRate = capture.get(CV_CAP_PROP_FPS);
#else
    frameRate = −1.0;
#endif

//  qDebug("framerate direct query = %f", frameRate);

    /*
     * if capture obtained frameRate is inconsistent, then we’ll try to find out
     * by ourselves
     */
    if (frameRate ≤ 0.0)
    {
        /*
         * If live Video : grab a few images and measure elapsed time
         */
        if (liveVideo)
        {
            QElapsedTimer localTimer;
            localTimer.start();

            for (size_t i=0; i < defaultFrameNumberTest; i++)
            {
                capture >> image;
            }

            frameDelay = (int)(localTimer.elapsed() / defaultFrameNumberTest);
            frameRate = 1.0/((double)frameDelay/1000.0);
            qDebug("Measured capture frame rate is  %4.2f images/s", frameRate);
        }
        /*
         * FIXME else ???
         * video files read through capture should provide framerate with
         * capture.get(CV_CAP_PROP_FPS) but what happens if they don’t ???
         */
    }
    else
    {
        qDebug("%s Capture frame rate = %4.2f", message.toStdString().c_str(),
                                                frameRate);
        frameDelay = 1000/frameRate;
    }

    statusMessage.sprintf("%s frame rate = %4.2f images/s",
                           message.toStdString().c_str(), frameRate);
    emit messageChanged(statusMessage, messageDelay);

    return frameDelay;
}

/*
 * Tries to set capture size directly on capture by using properties.
 *  − CV_CAP_PROP_FRAME_WIDTH to set frame width
 *  − CV_CAP_PROP_FRAME_HEIGHT to set frame height
 * @param width the width property to set on capture
 * @param height the height property to set on capture
 * @return true if capture is opened and if width and height have been
 * set successfully through @code capture.set(...) @endcode. Returns
 * false otherwise.
 * @post if at least width or height have been set successfully, capture
 * image is released then updated again so it will have the right
 * dimensions.
 */
bool QcvVideoCapture::setDirectSize(const unsigned int width,
                                    const unsigned int height)
{
#ifdef Q_OS_LINUX
    Q_UNUSED(width);
    Q_UNUSED(height);
#endif
    bool done = false;

    /*
     * We absolutely need this lock in order to safely set width and
     * height directly into the capture, so if mutex is already locked
     * we should wait for it to be unlocked before continuing. Moreover,
     * if mutex is NON−recursive and already locked, the call to lock() could
     * lead to a DEADlock, so mutex HAS to be recursive !
     */

#ifndef Q_OS_LINUX
    if (capture.isOpened())
    {
        bool setWidth = capture.set(CV_CAP_PROP_FRAME_WIDTH, (double)width);
        bool setHeight = capture.set(CV_CAP_PROP_FRAME_HEIGHT, (double)height);
        if (setWidth ∨ setHeight)
        {
            // release old capture image
            image.release();

            // force image update to get the right size
            capture >> image;

            done = true;
        }
    }
#endif
```

```

    return done;
}

/*
 * update slot trigerred by timer : Grabs a new image and sends updated()
 * signal iff new image has been grabbed, otherwise there is no more
 * images to grab so kills timer
 */
void QcvVideoCapture::update()
{
    bool locked = true;
    bool image_updated = false;

    if (updateThread ≠ NULL)
    {
        if (skip)
        {
            locked = mutex.tryLock();
            // qDebug() << "QcvVideoCapture::update trylock"
            //       << (locked ? "granted" : "failed");
            if (locked)
            {
                lockLevel++;
            }
        }
        else
        {
            if (lockLevel ≡ 0)
            {
                mutex.lock();
                // qDebug() << "QcvVideoCapture::update lock";
            }
            lockLevel++;
        }
    }

    if (capture.isOpened() ∧ locked)
    {
        capture >> image;

        if (¬image.data) // captured image has no data
        {
            statusMessage.clear();

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

                capture.release();

                statusMessage.sprintf("No more frames to capture ...");
                emit messageChanged(statusMessage, 0);
                qDebug("%s", statusMessage.toStdString().c_str());
            }
            else // not live video ==> video file
            {
                // We’ll try to rewinds the file back to frame 0
                bool restart = capture.set(CV_CAP_PROP_POS_FRAMES, 0.0);

                if (restart)
                {
                    statusMessage.sprintf("Capture restarted");
                    emit messageChanged(statusMessage,
                                        QcvVideoCapture::messageDelay);
                    qDebug("%s", statusMessage.toStdString().c_str());

                    // Refresh image chain resized −> flipped −> gray
                    setSize(size);
                }
                else
                {
                    capture.release();

                    statusMessage.sprintf("Failed to restart capture ...");
                    emit messageChanged(statusMessage, 0);
                    emit(finished());
                    qDebug("%s", statusMessage.toStdString().c_str());
                }
            }
        }
        else // capture image has data
        {
            /*
             * CAUTION
             * image->imageResized->imageFlipped->imageDisplay
             * constitute an image chain, so when size is changed with
             * setSize it should call setFlipVideo which should call
             * setGray
             */

            // resize image
            if (resize ∧ ¬directResize)
            {
                cv::resize(image, imageResized, imageResized.size(), 0, 0,
                    INTER_AREA);
            }
            /*
             * else imageResized.data is already == image.data
             */

            // flip image horizontally if required
            if (flipVideo)
            {
                flip(imageResized, imageFlipped, 1);
            }
            /*
             * else imageFlipped.data is already == imageResized.data
             */

            // convert image to gray if required
            if (gray)
            {
                cvtColor(imageFlipped, imageDisplay, CV_BGR2GRAY);
            }
            /*
             * else imageDisplay.data is already == imageFlipped.data
             */
            image_updated = true;
        }

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

        if (image_updated)
        {
            emit updated();
        }
    }
    else
    {
        // mutex hasn’t been locked, so we skipped one capture
        // qDebug() << "Capture skipped an image";
    }
}
```

```
/*
 * 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
```

```cpp
         *  - [-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_ */
```

```cpp
/*
 * 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
    {
```

```
        qWarning() << QObject::tr("CaptureFactory::setFile: path") << path
                   << QObject::tr(" does not exist");
    }
}

/*
 * Set video horizontal flip state (useful for selfies)
 * @param flipped the horizontal flip state
 */
void CaptureFactory::setFlipped(const bool flipped)
{
    flippedVideo = flipped;
}

/*
 * Set gray conversion
 * @param gray the gray conversion state
 */
void CaptureFactory::setGray(const bool gray)
{
    grayVideo = 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 CaptureFactory::setSkippable(const bool skip)
{
    skipImages = skip;
}

/*
 * Set video size (independently of video source actual size)
 * @param width the desired image width
 * @param height the desired image height
 */
void CaptureFactory::setSize(const size_t width, const size_t height)
{
    preferredWidth = (int)width;
    preferredHeight = (int)height;
}

/*
 * Set video size (independently of video source actual size)
 * @param size the desired video size
 */
void CaptureFactory::setSize(const QSize & size)
{
    preferredWidth = size.width();
    preferredHeight = size.height();
}

/*
 * 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 * CaptureFactory::getCaptureInstance(QThread * updateThread)
{
    // −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
    // Opening Video Capture
    // −−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
    if (liveVideo)
    {
        qDebug() << "opening device # " << deviceNumber;
    }
    else
    {
        qDebug() << "opening video file " << videoPath;
    }

    qDebug() << "Opening ";
    if (liveVideo)
    {
        // Live video feed
        qDebug() << "Live Video ... from camera # " << deviceNumber;
        capture = new QcvVideoCapture(deviceNumber,
                                      flippedVideo,
                                      grayVideo,
                                      skipImages,
                                      preferredWidth,
                                      preferredHeight,
                                      updateThread);
    }
    else
    {
        // Video file or stream
        qDebug() << videoPath << "…";
        capture = new QcvVideoCapture(videoPath,
                                      flippedVideo,
                                      grayVideo,
                                      skipImages,
                                      preferredWidth,
                                      preferredHeight,
                                      updateThread);
    }

    return capture;
}

```

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include "QcvVideoCapture.h"
#include "QcvSimpleDFT.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;

/**
 * Channels index 2 Widget index conversion
 */
static const CvProcessor::Channels RGB[3] = {CvProcessor::RED,
                                             CvProcessor::GREEN,
                                             CvProcessor::BLUE};

/**
 * 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 Fourier transform and filter processor
         * @param parent parent widget
         */
        explicit MainWindow(QcvVideoCapture * capture,
                            QcvSimpleDFT * processor,
                            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 change is requested
         * @param size the new video size
         */
        void sizeChanged(const QSize & size);

        /**
         * Signal to send for opening a device (camera) with the capture
         * @param deviceId device number to open
         * @param width desired width or 0 to keep capture width
         * @param height desired height or 0 to keep capture height
         * @return true if device has been opened and checked and timer launched
         */
        void openDevice(const int deviceId,
                        const unsigned int width,
                        const unsigned int height);

        /**
         * Signal to send for opening a video file in the capture
         * @param fileName video file to open
         * @param width desired width or 0 to keep capture width
         * @param height desired height or 0 to keep capture height
         * @return true if video has been opened and timer launched
         */
        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);

        /**
         * Signal to send when requesting gray image
         * @param gray gray image status
         */
        void grayImage(const bool gray);

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

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

        /**
         * The Fourier Transform and filter processor
         */
        QcvSimpleDFT * 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 setRenderingMode(const RenderMode mode);

        /**
         * Set filters spinBoxes and sliders link state
         * @param linked the link status
         * @post When link is on all sliders/spinboxes of low pass and high pass
         * filters are linked together, moving/changing one changes the others.
         * When link os off sliders/spinboxes are not linked together
         */
        void setLinkedFilterSizes(bool linked);

    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
         */
        void on_actionFile_triggered();

        /**
         * Menu action to quit application.
         */
        void on_actionQuit_triggered();

        /**
         * Menu action when flip image is selected.
         * Sets capture to change flip status which leads to reverse
         * image horizontally
         */
        void on_actionFlip_triggered();

        /**
         * Menu action when gray image is selected.
         * Sets capture to change gray status which leads convert captured image
         * to gray or not
         */
        void on_actionGray_triggered();

        /**
         * Menu action when original image size is selected.
         * Sets capture not to resize image
         */
        void on_actionOriginalSize_triggered();

        /**
         * Menu action when constrained image size is selected.
         * Sets capture resize to preferred width and height
         */
        void on_actionConstrainedSize_triggered();

        /**
         * Menu action to replace current image rendering widget by a
         * QcvMatWidgetImage instance.
         */
        void on_actionRenderImage_triggered();

        /**
         * Menu action to replace current image rendering widget by a
         * QcvMatWidgetLabel with pixmap instance.
         */
        void on_actionRenderPixmap_triggered();

        /**
         * Menu action to replace current image rendering widget by a
         * QcvMatWidgetGL instance.
         */
        void on_actionRenderOpenGL_triggered();

        /**
         * Original size radioButton action.
         * Sets capture resize to off
         */
        void on_radioButtonOrigSize_clicked();

        /**
         * Custom size radioButton action.
         * Sets capture resize to preferred width and height
         */
        void on_radioButtonCustomSize_clicked();

        /**
         * Width spinbox value change.
         * Changes the preferred width and if custom size is selected apply
         * this custom width
         * @param value the desired width
         */
        void on_spinBoxWidth_valueChanged(int value);

        /**
         * Height spinbox value change.
         * Changes the preferred height and if custom size is selected apply
         * this custom height
         * @param value the desired height
         */
        void on_spinBoxHeight_valueChanged(int value);

        /**
         * Flip capture image horizontally.
         * changes capture flip status
         */
        void on_checkBoxFlip_clicked();

        /**
         * convert capture image to gray level.
         * changes cpature gray conversion status
         */
        void on_checkBoxGray_clicked();

        /**
         * Changes logscale factor for spectrum
         * @param value the new logscale factor
         */
        void on_spinBoxMag_valueChanged(int value);

};

#endif // MAINWINDOW_H
```

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

#include <QObject>
#include <QFileDialog>
#include <QDebug>
#include <assert.h>

#include "QcvMatWidgetImage.h"
#include "QcvMatWidgetLabel.h"
#include "QcvMatWidgetGL.h"

/*
 * MainWindow constructor.
 * @param capture the capture QObject to capture frames from devices
 * or video files
 * @param processor Fourier transform and filter processor
 * @param parent parent widget
 */
MainWindow::MainWindow(QcvVideoCapture * capture,
                       QcvSimpleDFT * processor,
                       QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow),
    capture(capture),
    processor(processor),
    preferredWidth(341),
    preferredHeight(256)
{
    ui→setupUi(this);
    ui→scrollAreaSource→setBackgroundRole(QPalette::Mid);
    ui→scrollAreaSpectrum→setBackgroundRole(QPalette::Mid);

    // ------------------------------------------------------------------------
    // Assertions
    // ------------------------------------------------------------------------
    assert(capture ≠ NULL);

    assert(processor ≠ NULL);

    // ------------------------------------------------------------------------
    // Special widgets initialisation
    // ------------------------------------------------------------------------
    // Replace QcvMatWidget instances with QcvMatWidgetImage instances
    // sets image widget sources for the first time
    // connects processor->update to image Widgets->updated
    // connects processor->image changed to image widgets->setSourceImage
    setRenderingMode(RENDER_IMAGE);

    ui→labelFFTSizeValue→setText(QString::number(processor→getOptimalDftSize()));

    // ------------------------------------------------------------------------
    // rest of Signal/Slot connections
    // ------------------------------------------------------------------------
    // processor->sendText --> labelFFTSizeValue->setText when source image
    // changes, fft size might also change

    connect(processor, SIGNAL(sendText(QString)),
            ui→labelFFTSizeValue, SLOT(setText(QString)));

    // Capture, processor and this messages to status bar
    connect(capture, SIGNAL(messageChanged(QString,int)),
            ui→statusBar, SLOT(showMessage(QString,int)));

    connect(processor, SIGNAL(sendMessage(QString,int)),
            ui→statusBar, SLOT(showMessage(QString,int)));

    connect(this, SIGNAL(sendMessage(QString,int)),
            ui→statusBar, SLOT(showMessage(QString,int)));

    // 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()));

    // Connects UI requests to capture
    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)));
```

```cpp
    connect(this, SIGNAL(flipVideo(bool)), capture, SLOT(setFlipVideo(bool)));
    connect(this, SIGNAL(grayImage(bool)), capture, SLOT(setGray(bool)));

    // ------------------------------------------------------------------------

    // ------------------------------------------------------------------------
    // 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);

    // Sets flipCheckbox and menu item states
    bool flipped = capture→isFlipVideo();
    ui→actionFlip→setChecked(flipped);
    ui→checkBoxFlip→setChecked(flipped);

    // Sets grayCheckbox and menu item states
    bool gray = capture→isGray();
    ui→actionGray→setChecked(gray);
    ui→checkBoxGray→setChecked(gray);

    // ------------------------------------------------------------------------
    // UI setup according to DFTProcessor options
    // ------------------------------------------------------------------------
    // Setting up log scale spinbox value and boundaries
    ui→spinBoxMag→setValue((int)processor→getLogScaleFactor());
    ui→spinBoxMag→setMinimum((int)processor→minLogScaleFactor);
    ui→spinBoxMag→setMaximum((int)processor→maxLogScaleFactor);
}

/*
 * 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;
```

```cpp

    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());
//      }

//      setupProcessorFromUI(); // Should already be called by imageChanged signal

        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 gray image is selected.
 * Sets capture to change gray status which leads convert captured image
 * to gray or not
 */
void MainWindow::on_actionGray_triggered()
{
    bool isGray = ¬capture→isGray();

    // capture->setGray(isGray);
    emit grayImage(isGray);
}

/*
 * Menu action when original image size is selected.
 * Sets capture not to resize image
 */
void MainWindow::on_actionOriginalSize_triggered()
{

    ui→actionConstrainedSize→setChecked(false);

    // capture->setSize(0, 0);
    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);

    // capture->setSize(preferredWidth, preferredHeight);
    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::setRenderingMode(const RenderMode mode)
{
    // Disconnect signals from slots first
    disconnect(processor, SIGNAL(updated()),
               ui→sourceImage, SLOT(update()));
    disconnect(processor, SIGNAL(updated()),
```

```cpp
               ui→spectrumImage, SLOT(update()));

    disconnect(processor, SIGNAL(squareImageChanged(Mat*)),
               ui→sourceImage, SLOT(setSourceImage(Mat*)));
    disconnect(processor, SIGNAL(spectrumImageChanged(Mat*)),
               ui→spectrumImage, SLOT(setSourceImage(Mat*)));

    // remove widgets in scroll areas
    QWidget * wSource = ui→scrollAreaSource→takeWidget();
    QWidget * wSpectrum = ui→scrollAreaSpectrum→takeWidget();

    if ((wSource ≡ ui→sourceImage) ∧
        (wSpectrum ≡ ui→spectrumImage))
    {
        // delete removed widgets
        delete ui→sourceImage;
        delete ui→spectrumImage;

        // create new widget
        Mat * sourceMat = processor→getImagePtr("square");
        Mat * spectrumMat = processor→getImagePtr("spectrum");

        switch (mode)
        {
            case RENDER_PIXMAP:
                ui→sourceImage = new QcvMatWidgetLabel(sourceMat);
                ui→spectrumImage = new QcvMatWidgetLabel(spectrumMat);
                break;
            case RENDER_GL:
                ui→sourceImage = new QcvMatWidgetGL(sourceMat);
                ui→spectrumImage = new QcvMatWidgetGL(spectrumMat);
                break;
            case RENDER_IMAGE:
            default:
                ui→sourceImage = new QcvMatWidgetImage(sourceMat);
                ui→spectrumImage = new QcvMatWidgetImage(spectrumMat);
                break;
        }

        if ((ui→sourceImage ≠ NULL) ∧
            (ui→spectrumImage ≠ NULL))
        {
            // Name the new images widgets with same name as in UI files
             ui→sourceImage→setObjectName(QString::fromUtf8("sourceImage"));
             ui→spectrumImage→setObjectName(QString::fromUtf8("spectrumImage"));

            // add to scroll areas
            ui→scrollAreaSource→setWidget(ui→sourceImage);
            ui→scrollAreaSpectrum→setWidget(ui→spectrumImage);

            // Reconnect signals to slots
            connect(processor, SIGNAL(updated()),
                    ui→sourceImage, SLOT(update()));
            connect(processor, SIGNAL(updated()),
                    ui→spectrumImage, SLOT(update()));

            connect(processor, SIGNAL(squareImageChanged(Mat*)),
                    ui→sourceImage, SLOT(setSourceImage(Mat*)));
            connect(processor, SIGNAL(spectrumImageChanged(Mat*)),
                    ui→spectrumImage, 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(tr("QGLWidget"));
                    break;
                default:
```

```
                break;
            }
            emit sendMessage(message, 5000);
        }
        else
        {
            qDebug("MainWindow::on_actionRenderXXX some new widget is null");
        }
    }
    else
    {
        qDebug("MainWindow::on_actionRenderXXX removed widget is not in ui->");
    }
}

/*
 * Re setup processor from UI settings when source changes
 */
void MainWindow::setupProcessorFromUI()
{
    processor→setLogScaleFactor((double)ui→spinBoxMag→value());
}

/*
 * Menu action to replace current image rendering widget by a
 * QcvMatWidgetImage instance.
 */
void MainWindow::on_actionRenderImage_triggered()
{
    qDebug("Setting image rendering to: images");
    setRenderingMode(RENDER_IMAGE);
}

/*
 * Menu action to replace current image rendering widget by a
 * QcvMatWidgetLabel with pixmap instance.
 */
void MainWindow::on_actionRenderPixmap_triggered()
{
    qDebug("Setting image rendering to: pixmaps");
    setRenderingMode(RENDER_PIXMAP);
}

/*
 * Menu action to replace current image rendering widget by a
 * QcvMatWidgetGL instance.
 */
void MainWindow::on_actionRenderOpenGL_triggered()
{
    qDebug("Setting image rendering to: opengl");
    setRenderingMode(RENDER_GL);
}

/*
 * Original size radioButton action.
 * Sets capture resize to off
 */
void MainWindow::on_radioButtonOrigSize_clicked()
{
    ui→actionConstrainedSize→setChecked(false);
    // capture->setSize(0, 0);
    emit sizeChanged(QSize(0, 0));
}

/*
 * Custom size radioButton action.
 * Sets capture resize to preferred width and height
 */
void MainWindow::on_radioButtonCustomSize_clicked()
{
    ui→actionOriginalSize→setChecked(false);
    // capture->setSize(preferredWidth, preferredHeight);
    emit sizeChanged(QSize(preferredWidth, preferredHeight));
}

/*
 * Width spinbox value change.
 * Changes the preferred width and if custom size is selected apply
 * this custom width
 * @param value the desired width
 */
void MainWindow::on_spinBoxWidth_valueChanged(int value)
```

```
{
    preferredWidth = value;
    if (ui→radioButtonCustomSize→isChecked())
    {
        // capture->setSize(preferredWidth, preferredHeight);
        emit sizeChanged(QSize(preferredWidth, preferredHeight));
    }
}

/*
 * Height spinbox value change.
 * Changes the preferred height and if custom size is selected apply
 * this custom height
 * @param value the desired height
 */
void MainWindow::on_spinBoxHeight_valueChanged(int value)
{
    preferredHeight = value;
    if (ui→radioButtonCustomSize→isChecked())
    {
        // capture->setSize(preferredWidth, preferredHeight);
        emit sizeChanged(QSize(preferredWidth, preferredHeight));
    }
}

/*
 * Flip capture image horizontally.
 * changes capture flip status
 */
void MainWindow::on_checkBoxFlip_clicked()
{
    /*
     * There is no need to update ui->actionFlip since it is connected
     * to ui->checkBoxFlip through signals/slots
     */
    // capture->setFlipVideo(ui->checkBoxFlip->isChecked());
    emit flipVideo(ui→checkBoxFlip→isChecked());
}

/*
 * convert capture image to gray level.
 * changes cpature gray conversion status
 */
void MainWindow::on_checkBoxGray_clicked()
{
    bool isGray = ui→checkBoxGray→isChecked();
    // capture->setGray(isGray);
    emit grayImage(isGray);
}

/*
 * Changes logscale factor for spectrum
 * @param value the new logscale factor
 */
void MainWindow::on_spinBoxMag_valueChanged(int value)
{
    processor→setLogScaleFactor((double)value);

    double realScale = processor→getLogScaleFactor();

    ui→spinBoxMag→setValue((int)realScale);
}
```

```cpp
#include <QApplication>
#include <QThread>
#include <libgen.h>     // for basename
#include <iostream>     // for cout

#include "QcvVideoCapture.h"
#include "CaptureFactory.h"
#include "QcvSimpleDFT.h"
#include "mainwindow.h"

/**
 * Usage function shown just before launching QApp
 * @param name the name of the program (argv[0])
 */
void usage(char * name);

/**
 * Test program OpenCV2 + QT4
 * @param argc argument count
 * @param argv argument values
 * @return QTApp return value
 * @par usage : <Progname> [--device | -d] <#> | [--file | -f ] <filename>
 * [--mirror | -m] [--size | -s] <width>x<height>
 *  - device : [--device | -d] <device #> (0, 1, ...) Opens capture device #
 *  - filename : [--file | -f ] <filename> Opens a video file or URL (including rtsp)
 *  - mirror : mirrors image horizontally before display
 *  - size : [--size | -s] <width>x<height> resize capture to fit desired <width>
 *  and <height>
 */
int main(int argc, char *argv[])
{
    // -------------------------------------------------------------------------
    // Instanciate QApplication to receive special QT args
    // -------------------------------------------------------------------------
    QApplication app(argc, argv);

    // Gets arguments after QT specials removed
    QStringList argList = QCoreApplication::arguments();

    int threadNumber = 3;
    // parse arguments for --threads tag
    for (QListIterator<QString> it(argList); it.hasNext(); )
    {
        QString currentArg(it.next());

        if (currentArg ≡ "-t" ∨ currentArg ≡"--threads")
        {
            // Next argument should be thread number integer
            if (it.hasNext())
            {
                QString threadString(it.next());
                bool convertOk;
                threadNumber = threadString.toInt(&convertOk,10);
                if (¬convertOk ∨ threadNumber < 1 ∨ threadNumber > 3)
                {
                    qWarning("Warning: Invalid thread number %d",threadNumber);
                    threadNumber = 3;
                }
            }
            else
            {
                qWarning("Warning: thread tag found with no following thread number");
            }

        }
    }
    // -------------------------------------------------------------------------
    // Create Capture factory using program arguments and
    // open Video Capture
    // -------------------------------------------------------------------------
    CaptureFactory factory(argList);
    factory.setSkippable(true);

    // Helper thread for capture
    QThread * capThread = NULL;
    if (threadNumber > 1)
    {
        capThread = new QThread();
    }

    // Capture
    QcvVideoCapture * capture = factory.getCaptureInstance(capThread);
```

```cpp

    // -------------------------------------------------------------------------
    // Create Fourier Processor
    // -------------------------------------------------------------------------
    // Helper thread for processor
    QThread * procThread = NULL;
    if (threadNumber > 2)
    {
        procThread = new QThread();
    }
    else
    {
        if (threadNumber > 1)
        {
            procThread = capThread;
        }
    }

    // Processsor
    QcvSimpleDFT * processor = NULL;
    if (procThread ≡ NULL)
    {
        processor = new QcvSimpleDFT(capture→getImage());
    }
    else
    {
        if (procThread ≠ capThread)
        {
            processor = new QcvSimpleDFT(capture→getImage(),
                                         capture→getMutex(),
                                         procThread);
        }
        else // procThread == capThread
        {
            processor = new QcvSimpleDFT(capture→getImage(),
                                         NULL,
                                         procThread);
        }
    }

    // -------------------------------------------------------------------------
    // Connects capture to processor
    // -------------------------------------------------------------------------
    // Connects capture update to QHistandLUT update
    QObject::connect(capture, SIGNAL(updated()),
                     processor, SLOT(update()));

    // connect capture changed image to QHistandLUT set input
    QObject::connect(capture, SIGNAL(imageChanged(Mat*)),
                     processor, SLOT(setSourceImage(Mat*)));
    // -------------------------------------------------------------------------
    // Now that Capture & Histogram are on then
    // add our MainWindow as toplevel
    // and launches app
    // -------------------------------------------------------------------------
    MainWindow w(capture, processor);
    w.show();

    usage(argv[0]);

    int retVal = app.exec();

    // -------------------------------------------------------------------------
    // Cleanup & return
    // -------------------------------------------------------------------------
    delete processor;
    delete capture;
    bool sameThread = capThread ≡ procThread;

    if (capThread ≠ NULL)
    {
        delete capThread;
    }

    if (procThread ≠ NULL ∧ ¬sameThread)
    {
        delete procThread;
    }

    return retVal;
}

```

```
/*
 * Usage function shown just before launching QApp
 * @param name the name of the program (argv[0])
 */
void usage(char * name)
{
    cout << "usage : " << basename(name) << " "
         << "[−d | −−device] <device number> "
         << "[−v | −−video] <video file> "
         << "[−s | −−size] <width>x<height> "
         << "[−m | −−mirror]"
         << "[−g | −−gray]"
         << endl;
}
```