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

#ifndef CVPROCESSOR_H_
#define CVPROCESSOR_H_

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

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

#include "CvProcessorException.h"
#include "MeanValue.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;

        /**
         * Mean/Std, min & max processing time type
         */
        typedef MeanValue<clock_t, double> ProcessTime;

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

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

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

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

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

        /**
         * The verbose level for printed messages
         */
        VerboseLevel verboseLevel;
```



```

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

        /**
         * Mean process time (averaged process times)
         */
        ProcessTime meanProcessTime;

        /**
         * 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 level 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
```

```
181             * @param name the name of the image we're looking for
182             * @return the image registered by this name in the additionnal images
183             * map
184             * @throw CvProcessorException#INVALID_NAME is used name is not already
185             * registerd in the images
186             */
187            const Mat & getImage(const char * name) const
188                throw (CvProcessorException);
189
190            /**
191             * Get image by name
192             * @param name the name of the image we're looking for
193             * @return the image registered by this name in the additionnal images
194             * map
195             * @throw CvProcessorException#INVALID_NAME is used name is not already
196             * registerd in the images
197             */
198            const Mat & getImage(const string & name) const
199                throw (CvProcessorException);
200
201            /**
202             * Get image pointer by name
203             * @param name the name of the image we're looking for
204             * @return the image pointer registered by this name in the additionnal
205             * images map
206             * @throw CvProcessorException#INVALID_NAME is used name is not already
207             * registerd in the images
208             */
209            Mat * getImagePtr(const char * name)
210                throw (CvProcessorException);
211
212            /**
213             * Get image pointer by name
214             * @param name the name of the image we're looking for
215             * @return the image registered by this name in the additionnal images
216             * map
217             * @throw CvProcessorException#INVALID_NAME is used name is not already
218             * registerd in the images
219             */
220            Mat * getImagePtr(const string & name)
221                throw (CvProcessorException);
222            // ------------------------------------------------------------------
223            // Options settings and gettings
224            // ------------------------------------------------------------------
225            /**
226             * Number of channels in source image
227             * @return the number of channels of source image
228             */
229            int getNbChannels() const;
230
231            /**
232             * Type of the source image
233             * @return the openCV type of the source image
234             */
235            int getType() const;
236
237            /**
238             * Get the current verbose level
239             * @return the current verbose level
240             */
241            VerboseLevel getVerboseLevel() const;
242
243            /**
244             * Set new verbose level
245             * @param level the new verobse level
246             */
247            virtual void setVerboseLevel(const VerboseLevel level);
248
249            /**
250             * Return processor processing time of step index [default implementation
251             * returning only processTime, should be reimplemented in subclasses]
252             * @param index index of the step which processing time is required,
253             * 0 indicates all steps, and values above 0 indicates step #. If
254             * required index is bigger than number of steps then all steps value
255             * should be returned.
256             * @return the processing time of step index.
257             * @note should be reimplemented in subclasses in order to define
258             * time/feature behaviour
259             */
260            virtual double getProcessTime(const size_t index = 0) const;
261
262            /**
263             * Return processor mean processing time of step index [default
264             * implementation returning only processTime, should be reimplemented
265             * in subclasses]
266             * @param index index of the step which processing time is required,
267             * 0 indicates all steps, and values above 0 indicates step #. If
268             * required index is bigger than number of steps then all steps value
269             * should be returned.
270             * @return the mean processing time of step index.
```



```
271             * @note should be reimplemented in subclasses in order to define
272             * time/feature behaviour
273             * @param index
274             */
275            virtual double getMeanProcessTime(const size_t index = 0) const;
276
277            /**
278             * Return processor processing time std of step index [default
279             * implementation returning only processTime, should be reimplemented
280             * in subclasses]
281             * @param index index of the step which processing time is required,
282             * 0 indicates all steps, and values above 0 indicates step #. If
283             * required index is bigger than number of steps than all steps value
284             * should be returned.
285             * @return the mean processing time of step index.
286             * @note should be reimplemented in subclasses in order to define
287             * time/feature behaviour
288             * @param index
289             */
290            virtual double getStdProcessTime(const size_t index = 0) const;
291
292            /**
293             * Return processor minimum processing time of step index [default
294             * implementation returning only processTime, should be reimplemented
295             * in subclasses]
296             * @param index index of the step which processing time is required,
297             * 0 indicates all steps, and values above 0 indicates step #. If
298             * required index is bigger than number of steps than all steps value
299             * should be returned.
300             * @return the mean processing time of step index.
301             * @note should be reimplemented in subclasses in order to define
302             * time/feature behaviour
303             * @param index
304             */
305            virtual clock_t getMinProcessTime(const size_t index = 0) const;
306
307            /**
308             * Return processor maximum processing time of step index [default
309             * implementation returning only processTime, should be reimplemented
310             * in subclasses]
311             * @param index index of the step which processing time is required,
312             * 0 indicates all steps, and values above 0 indicates step #. If
313             * required index is bigger than number of steps than all steps value
314             * should be returned.
315             * @return the mean processing time of step index.
316             * @note should be reimplemented in subclasses in order to define
317             * time/feature behaviour
318             * @param index
319             */
320            virtual clock_t getMaxProcessTime(const size_t index = 0) const;
321
322            /**
323             * Reset mean and std process time in order to re-start computing
324             * new mean and std process time values.
325             */
326            virtual void resetMeanProcessTime();
327
328            /**
329             * Indicates if processing time is per feature processed in the current
330             * image or absolute
331             * @return
332             */
333            bool isTimePerFeature() const;
334
335            /**
336             * Sets Time per feature processing time unit
337             * @param value the time per feature value (true or false)
338             */
339            virtual void setTimePerFeature(const bool value);
340
341            /**
342             * Send to stream (for showing processor attributes values)
343             * @param out the stream to send to
344             * @return a reference to the output stream
345             */
346            virtual ostream & toStream(ostream & out) const;
347
348            /**
349             * Send to any stream template
350             * @tparam Stream the stream type
351             * @param out the output stream
352             * @return a reference to the output stream
353             * @note this template method needs to be implemented in the header so
354             * it could be available in any source (.cpp) file that need a specific
355             * instantiation of this template method, for instance:
356             * @code
357             * template ostream & CvProcessor::toStream_Impl<ostream>(ostream &) const;
358             * @endcode
359             */
360            template <typename Stream>
```

```
361         Stream & toStream_Impl(Stream & out) const
362         {
363             out << "Verbose Level = ";
364             switch (verboseLevel)
365             {
366                 case VERBOSE_NONE:
367                     out << "None";
368                     break;
369                 case VERBOSE_ERRORS:
370                     out << "Only error messages";
371                     break;
372                 case VERBOSE_WARNINGS:
373                     out << "Error & warning messages";
374                     break;
375                 case VERBOSE_NOTIFICATIONS:
376                     out << "Error + warning + notifications";
377                     break;
378                 case VERBOSE_ACTIVITY:
379                     out << "Error + warning + notifications + log";
380                     break;
381                 case NBVERBOSELEVEL:
382                 default:
383                     out << "Unkonwn";
384                     break;
385             }
386
387             out << '\n' << "Images = " << '\n';
388
389             map<string, Mat*>::const_iterator cit;
390
391             for (cit = images.begin(); cit ≠ images.end(); ++cit)
392             {
393                 Mat * currentImage = cit→second;
394
395                 out << '\t' << cit→first.c_str() << "(" << currentImage→cols << 'x'
396                     << currentImage→rows << 'x' << currentImage→channels() << ")[";
397                 switch (currentImage→depth())
398                 {
399                     case CV_8U:
400                         out << "8-bit unsigned integers]";
401                         break;
402                     case CV_8S:
403                         out << "8-bit signed integers]";
404                         break;
405                     case CV_16U:
406                         out << "16-bit unsigned integers]";
407                         break;
408                     case CV_16S:
409                         out << "16-bit signed integers]";
410                         break;
411                     case CV_32S:
412                         out << "32-bit signed integers]";
413                         break;
414                     case CV_32F:
415                         out << "32-bit floating-point numbers]";
416                         break;
417                     case CV_64F:
418                         out << "64-bit floating-point numbers]";
419                         break;
420                     default:
421                         out << "Unknwon number type]";
422                         break;
423                 }
424
425                 out << '\n';
426             }
427
428             out << "Time per feature = " << (timePerFeature ? "Yes" : "No")
429                 << '\n';
430
431             return out;
432         }
433
434     protected:
435         // ----------------------------------------------------------------------
436         // Setup and cleanup attributes
437         // ----------------------------------------------------------------------
438         /**
439          * Setup internal attributes according to source image
440          * @param sourceImage a new source image
441          * @param fullSetup full setup is needed when source image is changed
442          * @pre sourceImage is not NULL
443          * @note this method should be reimplemented in sub classes
444          */
445         virtual void setup(Mat * sourceImage, const bool fullSetup = true);
446
447         /**
448          * Clean up internal attributes before changing source image or
449          * cleaning up class before destruction
450          * @note this method should be reimplemented in sub classes
```



```
451          */
452         virtual void cleanup();
453 };
454
455 /**
456  * Send to output stream operator
457  * @param out the output stream to send to
458  * @param proc the processor to send to the output stream
459  * @return a reference to the output stream used
460  */
461 ostream & operator <<(ostream & out, const CvProcessor & proc);
462
463 /**
464  * Converts en enum element into its integral type.
465  * Iff the enum is defined as int as its base type
466  * @param e the enum item to be converted into its underlying type
467  */
468 template<typename E>
469 constexpr auto integral(const E e) → typename underlying_type<E>::type
470 {
471    return static_cast<typename underlying_type<E>::type>(e);
472 }
473
474 #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),
    meanProcessTime(clock_t(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;
    resetMeanProcessTime();
    addImage("source", this->sourceImage);
}

/*
 * Clean up internal attributes before changing source image or
 * cleaning up class before destruction
 * @note this method should be reimplemented in sub classes
 */
void CvProcessor::cleanup()
{
    if (verboseLevel ≥ VERBOSE_ACTIVITY)
    {
        clog << "CvProcessor::cleanup()" << endl;
    }

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

/*
 * Changes source image
 * @param sourceImage the new source image
 * @throw CvProcessorException#NULL_IMAGE when new source image is NULL
 */
void CvProcessor::setSourceImage(Mat *sourceImage)
    throw (CvProcessorException)
{
    if (verboseLevel ≥ VERBOSE_NOTIFICATIONS)
    {
        clog << "CvProcessor::setSourceImage(" << (unsigned long) sourceImage
             << ")" << endl;
    }

    // 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
```

```
181     {
182         retValue = true;
183     }
184 
185     return retValue;
186 }
187 /*
188  * Update named image in additionnal images.
189  * @param name the name of the image
190  * @param image the image reference
191  * @post the image located at key name is updated.
192  */
193 //void CvProcessor::updateImage(const char * name, Mat * image)
194 //{
195 //  // Search for this name in the map
196 //  map<string, Mat*>::iterator it;
197 //  for (it = images.begin(); it != images.end(); ++it)
198 //  {
199 //      if (it->first == name)
200 //      {
201 //          (it->second->release();
202 //          images.erase(it);
203 //      }
204 //  }
205 //
206 //  string sname(name);
207 //
208 //  updateImage(sname, image);
209 //}
210 
211 /*
212  * Update named image in additionnal images.
213  * @param name the name of the image
214  * @param image the image reference
215  * @post the image located at key name is updated.
216  */
217 //void CvProcessor::updateImage(const string & name, const Mat & image)
218 //{
219 //  clog << "update image " << name << " with " << (long) &image << endl;
220 //  images.erase(name);
221 //
222 //  addImage(name, image);
223 //}
224 
225 /*
226  * Get image by name
227  * @param name the name of the image we're looking for
228  * @return the image registered by this name in the additionnal images
229  * map
230  * @throw CvProcessorException#INVALID_NAME is used name is not already
231  * registerd in the images
232  */
233 const Mat & CvProcessor::getImage(const char *name) const
234     throw (CvProcessorException)
235 {
236     string sname(name);
237 
238     return getImage(sname);
239 }
240 
241 /*
242  * Get image pointer by name
243  * @param name the name of the image we're looking for
244  * @return the image pointer registered by this name in the additionnal
245  * images map
246  * @throw CvProcessorException#INVALID_NAME is used name is not already
247  * registerd in the images
248  */
249 const Mat & CvProcessor::getImage(const string & name) const
250     throw (CvProcessorException)
251 {
252     // Search for this name
253     map<string, Mat*>::const_iterator cit;
254     for (cit = images.begin(); cit ≠ images.end(); ++cit)
255     {
256         if (cit→first ≡ name)
257         {
258             if (cit→second→data ≡ NULL)
259             {
260                 // image contains no data
261                 throw CvProcessorException(CvProcessorException::NULL_DATA,
262                                            name.c_str());
263             }
264             return *(cit→second);
265         }
266     }
267 
268     // not found : throw exception
269     throw CvProcessorException(CvProcessorException::INVALID_NAME,
270                                name.c_str());
```



```
271 }
272 
273 /*
274  * Get image pointer by name
275  * @param name the name of the image we're looking for
276  * @return the image pointer registered by this name in the additionnal
277  * images map
278  * @throw CvProcessorException#INVALID_NAME is used name is not already
279  * registerd in the images
280  */
281 Mat * CvProcessor::getImagePtr(const char *name)
282     throw (CvProcessorException)
283 {
284     string sname(name);
285 
286     return getImagePtr(sname);
287 }
288 
289 /*
290  * Get image pointer by name
291  * @param name the name of the image we're looking for
292  * @return the image registered by this name in the additionnal images
293  * map
294  * @throw CvProcessorException#INVALID_NAME is used name is not already
295  * registerd in the images
296  */
297 Mat * CvProcessor::getImagePtr(const string & name)
298     throw (CvProcessorException)
299 {
300     // Search for this name
301     map<string, Mat*>::const_iterator cit;
302     for (cit = images.begin(); cit ≠ images.end(); ++cit)
303     {
304         if (cit→first ≡ name)
305         {
306             if (verboseLevel ≥ VERBOSE_ACTIVITY)
307             {
308                 clog << "getImagePtr(" << name << "): returning : "
309                      << (long) (cit→second) << endl;
310             }
311             return cit→second;
312         }
313     }
314 
315     // not found : throw exception
316     throw CvProcessorException(CvProcessorException::INVALID_NAME, name.c_str());
317 }
318 
319 /*
320  * Number of channels in source image
321  * @return the number of channels of source image
322  */
323 int CvProcessor::getNbChannels() const
324 {
325     return nbChannels;
326 }
327 
328 /*
329  * Type of the source image
330  * @return the openCV type of the source image
331  */
332 int CvProcessor::getType() const
333 {
334     return type;
335 }
336 
337 /*
338  * Get the current verbose level
339  * @return the current verbose level
340  */
341 CvProcessor::VerboseLevel CvProcessor::getVerboseLevel() const
342 {
343     return verboseLevel;
344 }
345 
346 /*
347  * Set new verbose level
348  * @param level the new verobse level
349  */
350 void CvProcessor::setVerboseLevel(const VerboseLevel level)
351 {
352     if ((level ≥ VERBOSE_NONE) ∧ (level < NBVERBOSELEVEL))
353     {
354         verboseLevel = level;
355     }
356 
357     cout << "Verbose level set to: ";
358     switch (verboseLevel)
359     {
360         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;
    }

    /*
     * Return processor mean 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 mean processing time of step index.
     * @note should be reimplemented in subclasses in order to define
     * time/feature behaviour
     * @param index
     */
    double CvProcessor::getMeanProcessTime(const size_t) const
    {
        return meanProcessTime.mean();
    }

    /*
     * Return processor processing time std 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 mean processing time of step index.
     * @note should be reimplemented in subclasses in order to define
     * time/feature behaviour
     * @param index
     */
    double CvProcessor::getStdProcessTime(const size_t) const
    {
        return meanProcessTime.std();
    }

    /*
     * Return processor minimum 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 mean processing time of step index.
     * @note should be reimplemented in subclasses in order to define
     * time/feature behaviour
     * @param index
     */
    clock_t CvProcessor::getMinProcessTime(const size_t) const
    {
        return meanProcessTime.min();
```



```
    }

    /*
     * Return processor maximum 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 mean processing time of step index.
     * @note should be reimplemented in subclasses in order to define
     * time/feature behaviour
     * @param index
     */
    clock_t CvProcessor::getMaxProcessTime(const size_t) const
    {
        return meanProcessTime.max();
    }

    /*
     * Reset mean and std process time in order to re-start computing
     * new mean and std process time values.
     */
    void CvProcessor::resetMeanProcessTime()
    {
        meanProcessTime.reset();
    }


    /*
     * 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;
    }

    /*
     * Send to stream (for showing processor attributes values)
     * @param out the stream to send to
     * @return a reference to the output stream
     */
    ostream & CvProcessor::toStream(ostream & out) const
    {
        return toStream_Impl<ostream>(out);
    }

    /*
     * Send to output stream operator
     * @param out the output stream to send to
     * @param proc the processor to send to the output stream
     * @return a reference to the output stream used
     */
    ostream & operator <<(ostream & out, const CvProcessor & proc)
    {
        return proc.toStream(out);
    }

    /*
     * Proto instantiation of CvProcessor template method
     * Stream & CvProcessor::toStream_Impl<Stream>(Stream &) const with concrete
     * type ostream
     */
    template ostream & CvProcessor::toStream_Impl<ostream>(ostream &) const;
```

```
#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;
         * }
         */

    private:
        /**
         * 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:
            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;
```



```cpp
        case CvProcessorException::ALLOC_FAILURE:
            output << "New element allocation failure" << endl;
            break;
        case CvProcessorException::FILE_READ_FAIL:
            output << "Unable to read file" << endl;
            break;
        case CvProcessorException::FILE_PARSE_FAIL:
            output << "File parse error" << endl;
            break;
        case CvProcessorException::FILE_WRITE_FAIL:
            output << "Unable to write file" << endl;
            break;
        default:
            output << "Unknown exception" << endl;
            break;
    }

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


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

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

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

#ifndef QCVPROCESSOR_H_
#define QCVPROCESSOR_H_

#include <QObject>
#include <QDebug>
#include <QString>
#include <QRegExp>
#include <QMutex>
#include <QThread>
#include "CvProcessor.h"
Q_DECLARE_METATYPE(CvProcessor::ProcessTime)

/**
 * 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 QString numberFormat;

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

        /**
         * format used to format Mean/Std time values : <mean> Â± <std>
         */
        static QString meanStdFormat;

        /**
         * format used to format Min/Max time values : <min> / <max>
         */
        static QString minMaxFormat;

        /**
         * The Source image mutex in order to avoid concurrent access to
         * the source image (typically the source image may be currently
         * modified by the capture for instance)
         */
        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;

        /**
         * String used to store formatted min/max time values
         */
        QString processMinMaxTimeString;

    public:

        /**
         * QcvProcessor constructor
         * @param image the source image
         * @param imageLock the mutex for concurrent access to the source image.
         * In order to avoid concurrent access to the same image
         * @param updateThread the thread in which this processor should run
         * @param parent parent QObject
```



```
         */
        QcvProcessor(Mat * image,
                     QMutex * imageLock = NULL,
                     QThread * updateThread = NULL,
                     QObject * parent = NULL);

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

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

        /**
         * Get the format c-string for numbers
         * @return the format string for numbers (e.g.: "%5.2f")
         */
        static const char * getNumberFormat();

        /**
         * Get the format c-string for std dev of numbers
         * @return the format string for numbers (e.g.: " Â± %4.2f")
         */
        static const char * getStdFormat();

        /**
         * Get the format c-string for min / max of numbers
         * @return the format string for numbers (e.g.: "%5.2f / %5.2f")
         */
        static const char * getMinMaxFormat();

        /**
         * Send to debug stream (for showing processor attributes values)
         * @param dbg the debug stream to send to
         * @return a reference to the output stream
         */
        virtual QDebug & toDBStream(QDebug & dbg) const;

        /**
         * Friend QDebug output operator
         * @param dbg the debug stream
         * @param proc the QcvProcessor to send to debug stream
         * @return the debug stream
         */
        friend QDebug & operator <<(QDebug & dbg, const QcvProcessor & proc);

    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 (reimplemented as a slot).
         * @param value the time per feature value (true or false)
         */
        virtual void setTimePerFeature(const bool value);

        /**
         * Reset mean and std process time in order to re-start computing
         * (reimplemented as a slot)
         * new mean and std process time values.
         */
        virtual void resetMeanProcessTime();

    signals:
        /**
         * Signal emitted when update is complete
         */
```

```
181         void updated();
182
183         /**
184          * Signal emitted when processor has finished.
185          * Used to tell helper threads to quit
186          */
187         void finished();
188
189         /**
190          * Signal emitted when source image is reallocated
191          */
192         void imageChanged();
193
194         /**
195          * Signal emitted when source image is reallocated
196          * @param image the new source image pointer or none if just
197          * image changed notification is required
198          */
199         void imageChanged(Mat * image);
200
201         /**
202          * Signal emitted when source image colors changes from color to gray
203          * or from gray to color
204          */
205         void imageColorsChanged();
206
207         /**
208          * Signal emitted when source image size changes
209          */
210         void imageSizeChanged();
211
212         /**
213          * Signal emitted when processing time has changed
214          * @param formattedValue the new value of the processing time
215          */
216         void processTimeUpdated(const QString & formattedValue);
217
218         /**
219          * Signal emitted when min/max processing time has changed
220          * @param formattedValue the new value of the processing time
221          */
222         void processTimeMinMaxUpdated(const QString & formattedValue);
223
224         /**
225          * Signal emitted when processing time has changed
226          * @param time the new processing time
227          */
228         void processTimeUpdated(const CvProcessor::ProcessTime * time);
229
230         /**
231          * Signal to set text somewhere
232          * @param message the message
233          */
234         void sendText(const QString & message);
235
236         /**
237          * Signal to send update message when something changes
238          * @param message the message
239          * @param timeout number of ms the message should be displayed
240          */
241         void sendMessage(const QString & message, int timeout = defaultTimeOut);
242 };
243
244 #endif /* QCVPROCESSOR_H_ */
```



```
1  /*
2   * QCvProcessor.cpp
3   *
4   *  Created on: 19 fÃ©vr. 2012
5   *      Author: davidroussel
6   */
7
8  #include <QRegExpValidator>
9  #include <QMetaType>
10 #include <QDebug>
11 #include "QcvProcessor.h"
12
13 /*
14  * Proto instantiation of CvProcessor template method
15  * Stream & CvProcessor::toStream_Impl<Stream>(Stream &) const with concrete
16  * type Qdebug
17  */
18 template QDebug & CvProcessor::toStream_Impl<QDebug>(QDebug &) const;
19
20 /*
21  * Default timeout to show messages
22  */
23 int QcvProcessor::defaultTimeOut = 5000;
24
25 /*
26  * Number format used to format numbers into QStrings
27  */
28 QString QcvProcessor::numberFormat = QString::fromUtf8("%7.0f");
29
30 /*
31  * The regular expression used to validate new number formats
32  * @see #setNumberFormat
33  */
34 QRegExp QcvProcessor::numberRegExp("%[+- 0#]*[0-9]*([.][0-9]+)?[efEF]");
35
36 /*
37  * format used to format Mean/Std time values : <mean> Â± <std>
38  */
39 QString QcvProcessor::meanStdFormat = numberFormat + QString::fromUtf8(" Â± %5.0f");
40
41 /*
42  * format used to format Min/Max time values : <min> / <max>
43  */
44 QString QcvProcessor::minMaxFormat = numberFormat +  QString::fromUtf8("/") +
45                                      numberFormat;
46
47 /*
48  * QcvProcessor constructor
49  * @param image the source image
50  * @param imageLock the mutex for concurrent access to the source image
51  * In order to avoid concurrent access to the same image
52  * @param updateThread the thread in which this processor should run
53  * @param parent parent QObject
54  */
55 QcvProcessor::QcvProcessor(Mat * image,
56                            QMutex * imageLock,
57                            QThread * updateThread,
58                            QObject * parent) :
59     CvProcessor(image), // <-- virtual base class constructor first
60     QObject(parent),
61     sourceLock(imageLock),
62     updateThread(updateThread),
63     message(),
64     processTimeString()
65 {
66     if (updateThread ≠ NULL)
67     {
68         this→moveToThread(updateThread);
69
70         connect(this, SIGNAL(finished()), updateThread, SLOT(quit()),
71                 Qt::DirectConnection);
72
73         updateThread→start();
74     }
75 }
76
77 /*
78  * QcvProcessor destructor
79  */
80 QcvProcessor::~QcvProcessor()
81 {
82     // Lock might be already destroyed in source object so don't try to unlock
83
84     message.clear();
85     processTimeString.clear();
86
87     emit finished();
88
89     if (updateThread ≠ NULL)
90     {
```

```
91          // Wait until update thread has received the "finished" signal through
92          // "quit" slot
93          updateThread→wait();
94      }
95  }
96
97  /*
98   * Sets new number format
99   * @param format the new number format
100  */
101 void QcvProcessor::setNumberFormat(const char * format)
102 {
103     /*
104      * The format string should validate the following regex
105      * %[+- 0#]*[0-9]*([.][0-9]+)?[efEF]
106      */
107     QRegExpValidator validator(numberRegExp, NULL);
108
109     QString qFormat(format);
110     int pos = 0;
111     if (validator.validate(qFormat,pos) ≡ QValidator::Acceptable)
112     {
113         numberFormat = format;
114         meanStdFormat = format + QString::fromUtf8(" Â± ") + format;
115         minMaxFormat = format + QString::fromUtf8("/") + format;
116     }
117     else
118     {
119         qWarning("QcvProcessor::setNumberFormat(%s) : invalid format", format);
120     }
121 }
122
123 /*
124  * Send to stream (for showing processor attributes values)
125  * @param dbg the debug stream to send to
126  * @return a reference to the output stream
127  */
128 QDebug & QcvProcessor::toDBStream(QDebug & dbg) const
129 {
130     return toStream_Impl<QDebug>(dbg);
131 }
132
133 /*
134  * Friend QDebug output operator
135  * @param dbg the debug stream
136  * @param proc the Qcvprocessor to send to debug stream
137  * @return the debug stream
138  */
139 QDebug & operator << (QDebug & dbg, const QcvProcessor & proc)
140 {
141     proc.toDBStream(dbg.nospace());
142     return dbg.space();
143 }
144
145 /*
146  * Update computed images slot and sends updated signal
147  * required
148  */
149 void QcvProcessor::update()
150 {
151     /*
152      * Important note : CvProcessor::update() should NOT be called here
153      * since it should be called in QcvXXXprocessor subclasses such that
154      * QcvXXXProcessor::update method should contain :
155      *  - call to CvXXXProcessor::update() (not QCvXXXProcessor)
156      *  - emit signals from QcvXXXProcessor
157      *  - call to QcvProcessor::update() (this method) to
158      *      - emit updated signal
159      *      - emit standard process time strings signals
160      *  - or
161      *      - emit updated signal in QcvXXXProcessor
162      *      - customize your processtimes and emit time strings signals
163      */
164     emit updated();
165     processTimeString.sprintf(meanStdFormat.toStdString().c_str(),
166                               getMeanProcessTime(0), getStdProcessTime(0));
167 //  processMinMaxTimeString.sprintf(minMaxFormat.toStdString().c_str(),
168 //                                  getMinProcessTime(0), getMaxProcessTime(0));
169     emit processTimeUpdated(processTimeString);
170 //  emit processTimeMinMaxUpdated(processMinMaxTimeString);
171     emit processTimeUpdated(&meanProcessTime);
172 }
173
174 /*
175  * Changes source image slot.
176  * Attributes needs to be cleaned up then set up again
177  * @param image the new source Image
178  * @post Various signals are emitted:
179  *  - imageChanged(sourceImage)
180  *  - imageCchanged()
```



```
181  *  - if image size changed then imageSizeChanged() is emitted
182  *  - if image color space changed then imageColorsChanged() is emitted
183  */
184 void QcvProcessor::setSourceImage(Mat *image)
185     throw (CvProcessorException)
186 {
187     Size previousSize(sourceImage→size());
188     int previousNbChannels(nbChannels);
189
190     if (sourceLock ≠ NULL)
191     {
192         sourceLock→lock();
193         // qDebug() << "QcvProcessor::setSourceImage: lock";
194     }
195
196     CvProcessor::setSourceImage(image);
197
198     if (sourceLock ≠ NULL)
199     {
200         // qDebug() << "QcvProcessor::setSourceImage: unlock";
201         sourceLock→unlock();
202     }
203
204     emit imageChanged(sourceImage);
205
206     emit imageChanged();
207
208     if ((previousSize.width ≠ image→cols) ∨
209         (previousSize.height ≠ image→rows))
210     {
211         emit imageSizeChanged();
212     }
213
214     if (previousNbChannels ≠ nbChannels)
215     {
216         emit imageColorsChanged();
217     }
218
219     // Force update
220     update();
221 }
222
223 /*
224  * Sets Time per feature processing time unit (reimplemented as a slot).
225  * @param value the time per feature value (true or false)
226  */
227 void QcvProcessor::setTimePerFeature(const bool value)
228 {
229     CvProcessor::setTimePerFeature(value);
230 }
231
232 /*
233  * Reset mean and std process time in order to re-start computing
234  * (reimplemented as a slot)
235  * new mean and std process time values.
236  */
237 void QcvProcessor::resetMeanProcessTime()
238 {
239     CvProcessor::resetMeanProcessTime();
240 }
241
242
243 /*
244  * Get the format c-string for numbers
245  * @return the format string for numbers (e.g.: "%5.2f")
246  */
247 const char * QcvProcessor::getNumberFormat()
248 {
249     return numberFormat.toStdString().c_str();
250 }
251
252 /*
253  * Get the format c-string for std dev of numbers
254  * @return the format string for numbers (e.g.: " Â± %4.2f")
255  */
256 const char * QcvProcessor::getStdFormat()
257 {
258     return meanStdFormat.toLocal8Bit().data();
259 }
260
261 /*
262  * Get the format c-string for min / max of numbers
263  * @return the format string for numbers (e.g.: "%5.2f / %5.2f")
264  */
265 const char * QcvProcessor::getMinMaxFormat()
266 {
267     return minMaxFormat.toLocal8Bit().data();
268 }
```

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

#ifndef CVHISTOGRAMS_H_
#define CVHISTOGRAMS_H_

#include <vector>
using namespace std;

#include "CvProcessor.h"

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

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

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



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

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

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

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

        /**
         * Checks whether to compute additionnal gray level histogram
         * from RGB components.
         * @note has no impact if number of channels in the image is not 3
         */
        bool computeGray;

        /**
         * Number of computed histograms.
         * @note could be bigger than the number of channels in the image
         * if an additional gray level histogram is computed.
         */
        size_t nbHistograms;

        /**
         * The histogram values (an array containing "bins" elements).
         * if image has 3 channels (BGR), a fourth histogram is computed
         * according to the computeGray attribute in order to compute the
```

```
181         * equivalent gray level histogram.
182         * @see #BGR2Gray
183         */
184        vector<float *> histograms;
185
186        /**
187         * Maximum value found in all histograms
188         */
189        float maxValue;
190
191        /**
192         * The cumulative histogram computed by cumulatively sum "hist".
193         * (an array containing "bins" elements)
194         */
195        vector<float *> cumulHistograms;
196
197        /**
198         * Maximum value found in all cumulative histogram.
199         * @note cumulative maximum should be the number of pixels
200         * in the image but when histogram is time cumulative it is
201         * a multiple of number of pixels.
202         */
203        float cMaxValue;
204
205        /**
206         * checks whether histograms are time cumulative or not.
207         * if time cumulative histogram value are not cleared before
208         * updating the histogram values.
209         */
210        bool timeCumulative;
211
212        // ------------------------------------------------------------------
213        // LUT attributes
214        // ------------------------------------------------------------------
215        /**
216         * Gray level transfert function
217         */
218        Mat monoTransfertFunc;
219
220        /**
221         * Colors transfert functions
222         */
223        Mat colorTransferFunc;
224
225        /**
226         * Current LUT to apply.
227         * Alternatively receives monoTransfertFunc or colorTransferFunc address
228         * depending on the transfert function to apply
229         */
230        Mat * lut;
231
232        /**
233         * Current LUT type
234         */
235        TransfertType lutType;
236
237        /**
238         * Previous LUT type. Used to avoid recomputing LUTs that does not
239         * depend on image histogram such as NONE, GAMMA and NEGATIVE
240         */
241        TransfertType previousLutType;
242
243        /**
244         * Current percentage for LUTs that requires such a parameter
245         */
246        float lutParam;
247
248        /**
249         * previous percentage for LUTs that requires such a parameter.
250         * Needed to know if LUT not depending on image histogram should be
251         * refreshed when param changes, such as Gamma
252         */
253        float previousLutParam;
254
255        /**
256         * Maximum percentage for LUTs that requires such a parameter
257         */
258        static const float maxParam;
259
260        /**
261         * Minimum percentage for LUTs that requires such a parameter
262         */
263        static const float minParam;
264
265        /**
266         * Indicates if LUT has been updated
267         */
268        bool lutUpdated;
269
270        // ------------------------------------------------------------------
```



```
271        // Drawing attributes
272        // ------------------------------------------------------------------
273        /**
274         * checks whether to show cumulative histograms in the drawing or
275         * regular histograms
276         */
277        bool showCumulative;
278
279        /**
280         * components to show in the drawing
281         */
282        vector<bool> showComponent;
283
284        /**
285         * image width of the histogram drawing frame
286         */
287        size_t histWidth;
288
289        /**
290         * image height of the histogram drawing frame
291         */
292        size_t histHeight;
293
294        /**
295         * image width of the LUT drawing frame
296         */
297        size_t lutWidth;
298
299        /**
300         * image height of the LUT drawing frame
301         */
302        size_t lutHeight;
303
304        /**
305         * drawing color for the histograms
306         */
307        vector<Scalar> displayColors;
308
309        /**
310         * The color Matrices to draw each histogram
311         */
312        vector<Mat> histComponents;
313
314        /**
315         * The Frame to draw all histograms in
316         */
317        Mat histDisplayFrame;
318
319        /**
320         * The color Matrices to draw each LUT
321         */
322        vector<Mat> lutComponents;
323
324        /**
325         * The Frame to draw all LUTs in
326         */
327        Mat lutDisplayFrame;
328
329        /**
330         * The frame to draw transformed image when LUT is applied
331         */
332        Mat outDisplayFrame;
333
334        // ------------------------------------------------------------------
335        // Time measurement attributes
336        // ------------------------------------------------------------------
337
338        /**
339         * Update histogram time when new frames appends
340         */
341        clock_t updateHistogramTime1;
342
343        /**
344         * Mean update histogram time when new frames appends
345         */
346        ProcessTime meanUpdateHistogramTime1;
347
348        /**
349         * LUT computing time
350         */
351        clock_t computeLUTTime;
352
353        /**
354         * Mean LUT computing time
355         */
356        ProcessTime meanComputeLUTTime;
357
358        /**
359         * LUT drawing time
360         */
```

```
361         clock_t drawLUTTime;
362
363         /**
364          * LUT drawing time
365          */
366         ProcessTime meanDrawLUTTime;
367
368         /**
369          * LUT apply time on image
370          */
371         clock_t applyLUTTime;
372
373         /**
374          * mean LUT apply time on image
375          */
376         ProcessTime meanApplyLUTTime;
377
378         /**
379          * Update histogram time after LUT is applied (when needed)
380          */
381         clock_t updateHistogramTime2;
382
383         /**
384          * mean update histogram time after LUT is applied (when needed)
385          */
386         ProcessTime meanUpdateHistogramTime2;
387
388         /**
389          * Histogram drawing time
390          */
391         clock_t drawHistogramTime;
392
393         /**
394          * Mean histogram drawing time
395          */
396         ProcessTime meanDrawHistogramTime;
397
398     public:
399
400         /**
401          * Histogram constructor
402          * @param image the image to use for computing histograms
403          * @param drawColors the drawing colors of the histogram
404          * @param computeGray checks whether to compute 4th gray level
405          * histogram on BGR image or not
406          * @param drawHeight the drawing height of the histogram window
407          * @param drawWidth the drawing width of the histogram window
408          * @param timeCumulation checks whether to compute time cumulative
409          * histograms or not.
410          */
411         CvHistograms(Mat * image,
412                      const bool computeGray = true,
413                      const size_t drawHeight = 256,
414                      const size_t drawWidth = 512,
415                      const bool timeCumulation = false);
416
417         /**
418          * Histogram destructor.
419          * clears histogram values and release display frame
420          */
421         virtual ~CvHistograms();
422
423         /**
424          * Update histogram, LUT and resulting images
425          */
426         virtual void update();
427
428         /**
429          * Update histograms values.
430          */
431         virtual void updateHistogram();
432
433         /**
434          * Value reading access operator
435          * @param i the ith histogram to access. if i is invalid, 0 is returned
436          * @param j the jth bin value of the ith histogram to access. if j is
437          * invalid, 0 is returned.
438          * @param cumulative checks whether to return regular histogram value
439          * or cumulative histogram value
440          * @return the value in the jth bin of the ith histogram
441          * @par usage :
442          * @code
443          *  float ithValue = myHist(i,j);
444          *  float ithCumulativeValue = myHist(i,j,true);
445          * @endcode
446          */
447         float operator()(const size_t i,
448                          const size_t j,
449                          const bool cumulative = false) const;
450
```



```
451         /**
452          * Value reading/writing access operator
453          * @param i the ith histogram to access. if i is invalid, 0 is returned
454          * @param j the jth bin value of the ith histogram to access. if j is
455          * invalid, 0 is returned.
456          * @param cumulative checks whether to return regular histogram value
457          * or cumulative histogram value
458          * @return the value in the jth bin of the ith histogram
459          * @par usage :
460          * @code
461          *  float myHist(i,j) = ithValue;
462          *  float myHist(i,j,true) = jthCumulativeValue;
463          * @endcode
464          */
465         float & operator()(const size_t i,
466                            const size_t j,
467                            const bool cumulative = false);
468
469         /**
470          * Number of bins in all histograms
471          * @return the Number of bins in all histograms
472          */
473         static size_t getBins();
474
475         /**
476          * Get the number of histograms computed
477          * @return the current number of histograms computed by this class
478          */
479         size_t getNbHistograms() const;
480
481         /**
482          * Gets the additionnal gray histogram status
483          * @return true if additional gray level histogram is computed,
484          * false otherwise
485          */
486         bool isComputeGray() const;
487
488         /**
489          * Maximum histograms value:
490          * @return the maximum value in all histograms
491          */
492         float getMaxValue() const;
493
494         /**
495          * Maximum cumulative histograms value:
496          * @return the maximum value in all histograms
497          * @note regular cumulative maximum value is the number of pixels in
498          * the image, but when timecumulative is activated it can be bigger.
499          */
500         float getCMaxValue() const;
501
502         /**
503          * Time cumulative histogram status read access
504          * @return the time cumulative histogram status
505          */
506         bool isTimeCumulative() const;
507
508         /**
509          * Time cumulative histogram status read access
510          * @param value the value to set for time cumulative status
511          */
512         virtual void setTimeCumulative(const bool value);
513
514         /**
515          * Cumulative histogram status read access
516          * @return the cumulative histogram status
517          */
518         bool isCumulative() const;
519
520         /**
521          * Cumulative histogram status read access
522          * @param value the value to set for cumulative status
523          */
524         virtual void setCumulative(const bool value);
525
526         /**
527          * Ith histogram component shown status read access
528          * @param i the ith histogram component
529          * @return true if this component show status is true
530          */
531         bool isShowComponent(const size_t i) const;
532
533         /**
534          * Ith histogram component shown status write access
535          * @param i the ith histogram component
536          * @param value the value to set for this component show status
537          */
538         virtual void setShowComponent(const size_t i,
539                                       const bool value);
540
```

```
541         /**
542          * Indicates if LUT has been updated or if it has not changed
543          * @return true if LUT has been updated
544          */
545         bool isLUTUpdated() const;
546
547         /**
548          * Gets the current LUT type
549          * @return the current LUT type
550          */
551         TransfertType getLutType() const;
552
553         /**
554          * Sets the current LUT type
555          * @param lutType the new LUT type
556          */
557         virtual void setLutType(const TransfertType lutType);
558
559         /**
560          * Gets the current parameter value for LUTs using a percentage parameter
561          * @return the current LUT parameter
562          */
563         float getLUTParam() const;
564
565         /**
566          * Sets the current LUT % parameter
567          * @param lutParam the new LUT parameter
568          */
569         virtual void setLUTParam(const float currentParam);
570
571         /**
572          * Return processor processing time of step index [default implementation
573          * returning only processTime, should be reimplemented in subclasses]
574          * @param index index of the step which processing time is required,
575          * 0 indicates all steps, and values above 0 indicates step #. If
576          * required index is bigger than number of steps than all steps value
577          * should be returned.
578          * @return the processing time of step index.
579          * @note should be reimplemented in subclasses in order to define
580          * time/feature behaviour
581          */
582         virtual double getProcessTime(const size_t index) const;
583
584         /**
585          * Return processor mean processing time of step index [default
586          * implementation returning only processTime, should be reimplemented
587          * in subclasses]
588          * @param index index of the step which processing time is required,
589          * 0 indicates all steps, and values above 0 indicates step #. If
590          * required index is bigger than number of steps than all steps value
591          * should be returned.
592          * @return the mean processing time of step index.
593          * @note should be reimplemented in subclasses in order to define
594          * time/feature behaviour
595          * @param index
596          */
597         virtual double getMeanProcessTime(const size_t index = 0) const;
598
599         /**
600          * Return processor processing time std of step index [default
601          * implementation returning only processTime, should be reimplemented
602          * in subclasses]
603          * @param index index of the step which processing time is required,
604          * 0 indicates all steps, and values above 0 indicates step #. If
605          * required index is bigger than number of steps than all steps value
606          * should be returned.
607          * @return the mean processing time of step index.
608          * @note should be reimplemented in subclasses in order to define
609          * time/feature behaviour
610          * @param index
611          */
612         virtual double getStdProcessTime(const size_t index = 0) const;
613
614         /**
615          * Reset mean and std process time in order to re-start computing
616          * new mean and std process time values.
617          */
618         virtual void resetMeanProcessTime();
619
620         /**
621          * output operator for Histograms
622          * @param out the output stream
623          * @param h the histograms to print on the stream
624          * @return a reference to the output stream so it can be cumulated
625          */
626         friend ostream & operator <<<> (ostream & out,
627                                         const CvHistograms<T,channels> & h);
628     protected:
629         /**
630          * Setup attributes when source image is changed
```



```
631          * @param image source Image
632          * @param completeSetup
633          * @param computeGray checks if additionnal gray level histogram should
634          * be computed
635          * @param drawHeight histogram draw height
636          * @param drawWidth histogram draw width
637          * @param timeCumulation cheks time cumulation status
638          */
639         virtual void setup(Mat * image,
640                            const bool completeSetup = false);
641         /**
642          * Cleanup attributes before changing source image or cleaning class
643          * before destruction
644          */
645         virtual void cleanup();
646
647         /**
648          * Draws selected histogram(s) in drawing frame and returns the drawing
649          * frame
650          * @return the updated drawing frame.
651          * @post depending on several attributes one or several histograms
652          * have been drawn in the drawing frame wich is returned
653          *  - if #showCumulative is true then cumulative histograms are drawn
654          *  otherwise regular histograms are drawn
655          *  - each histogram is drawn only if its showComponent[i] is true.
656          */
657         virtual void drawHistograms();
658
659         /**
660          * Draws selected transfert function in drawing frame and returns the
661          * drawing frame
662          * @param lut the LUT to draw : the LUT may contains 1 or several
663          * channels
664          * @return the updated drawing frame
665          */
666         virtual void drawTransfertFunc(const Mat * lut);
667
668         /**
669          * Compute linear transfert function (LUT) : no change in image levels
670          * @return the LUT containing the corresponding transfert function,
671          * the returned matrix contains only one channel corresponding to
672          * the graylevel LUT which should be applied to all color channels of
673          * the image
674          * @post the result is stored in monoTransfertFunc
675          * @note It's useless to compute a color Linear LUT since all channels
676          * would contain the exact same values.
677          */
678         Mat * computeLinearGrayLUT();
679
680         /**
681          * Compute linear transfert function (LUT) : no change in image levels
682          * @return the LUT containing the corresponding transfert function,
683          * the returned matrix contains 3 channels corresponding to
684          * the color LUT which should be applied to all color channels of
685          * the image
686          * @post the result is stored in colorTransfertFunc
687          * @note It's useless to compute a color Linear LUT since all channels
688          * would contain the exact same values.
689          */
690         Mat * computeLinearColorLUT();
691
692         /**
693          * Compute the optimal dynamic LUT for preserving "percentDynamic"
694          * percent of the whole image lightness range.
695          * @param percentDynamic the gray level percentage to spread on the
696          * whole (100%) gray level range in the image
697          * @return the LUT containing the corresponding transfert function,
698          * the returned matrix contains only one channel corresponding to
699          * the graylevel LUT which should be applied to all color channels of
700          * the image
701          * @post the result is stored in monoTransfertFunc
702          */
703         Mat * computeGrayOptimalLUT(const unsigned int percentDynamic);
704
705         /**
706          * Compute the optimal dynamic LUTs (one for each channel) for preserving
707          * "percentDynamic" percent of the whole image color ranges.
708          * @param percentDynamic the colors level percentage to spread on the
709          * whole (100%) colors level range in the image
710          * @return the LUT containing the corresponding transfert functions,
711          * the returned matrix contains as much channels as the image and
712          * corresponding to the color level LUT which should be applied to
713          * each color channels of the image
714          * @post the result is stored in colorTransfertFunc
715          */
716         Mat * computeColorOptimalLUT(const unsigned int percentDynamic);
717
718         /**
719          * Computes the transfert function corresponding to gray level
720          * equalization
```

```
721             * @return the matrix containing the gray level equalization LUT to
722             * apply on the image
723             * @post the result is stored in monoTransfertFunc
724             */
725            Mat * computeGrayEqualizeLUT();
726
727            /**
728             * Computes the transfert functions corresponding to each channel
729             * level equalization
730             * @return the matrix containg each channel level equalization LUT to
731             * apply on the image
732             * @post the result is stored in colorTransferFunc
733             */
734            Mat * computeColorEqualizeLUT();
735
736            /**
737             * Compute the LUT corresponding to thresholded image with tPercent
738             * of the pixel population on each side of the threshold according
739             * to the cumulative gray level histogram
740             * @param tPercent percent of the population on each side of the
741             * threshold
742             * @return the LUT containing the corresponding transfert function,
743             * the returned matrix contains only one channel corresponding to
744             * the graylevel LUT which should be applied to all color channels of
745             * the image
746             * @post the result is stored in monoTransfertFunc
747             */
748            Mat * computeGrayThresholdLUT(const float tPercent);
749
750            /**
751             * Compute the LUT corresponding to thresholded image with tPercent
752             * of the pixel components population on each side of the
753             * thresholds according to the cumulative color histograms
754             * @param tPercent percent of the population on each side of the
755             * thresholds
756             * @return the matrix containing each channel level equalization LUT to
757             * apply on the image
758             * @post the result is stored in colorTransferFunc
759             */
760            Mat * computeColorThresholdLUT(const float tPercent);
761
762            /**
763             * Compute gamma LUT.
764             * \f$v(k) = x(k)^{\gamma}\f$
765             * @param tPercent
766             * @return the matrix containing the gamma LUT (mono)
767             */
768            Mat * computeGammaLUT(const float tPercent);
769
770            /**
771             * Compute the LUT corresponding to negative image
772             * @return the matrix containing the negative LUT (mono)
773             */
774            Mat * computeNegativeLUT();
775
776            /**
777             * Compute and returns the current transfert function to be applied
778             * on the image, eventually with the current LUT parameter
779             * @return the mono or color LUT matrix to apply on the image depending
780             * on the lutType
781             * @see TransfertType
782             */
783            Mat * computeLUT();
784
785            /**
786             * Apply current LUT (if != NULL) to the source image to produce the
787             * outFrame
788             * @return true if LUT has been applied, false if lut is NULL or
789             * lutType is NONE
790             */
791            virtual bool drawTransformedImage();
792    };
793
794    #endif /* CVHISTOGRAMS_H_ */
```



```
1   /*
2    * CvHistograms.cpp
3    *
4    *  Created on: 22 fÃ©vr. 2012
5    *      Author: David Roussel
6    */
7   #include <cmath>    // for powf function
8   #include <iostream> // for input / output streams
9   #include <limits>   // for numeric limits (max value of type T)
10  using namespace std;
11
12  #include <opencv2/imgproc/imgproc.hpp>
13
14  #include "CvHistograms.h"
15
16  /*
17   * Number of bins in the histogram.
18   * All histogram populations ranges from 0 to bins-1
19   */
20  template <typename T, size_t channels>
21  const size_t CvHistograms<T, channels>::bins = (size_t) powf(2, sizeof(T) * 8);
22
23  /*
24   * 3 coefficients for additionnal grayscale histogram from RGB image :
25   *  - \f$Coef {red} = 0.30\f$
26   *  - \f$Coef {green} = 0.59\f$
27   *  - \f$Coef {blue} = 0.11\f$
28   * @note Be aware that OpenCV Color images are ususally encoded in BGR
29   * format instead of RGB.
30   */
31  template <typename T, size_t channels>
32  const float CvHistograms<T, channels>::BGR2Gray[] = {0.11, 0.59, 0.30};
33
34  /*
35   * Maximum percentage for LUTs that requires such a parameter
36   */
37  template <typename T, size_t channels>
38  const float CvHistograms<T, channels>::maxParam = 100.0;
39
40  /*
41   * Minimum percentage for LUTs that requires such a parameter
42   */
43  template <typename T, size_t channels>
44  const float CvHistograms<T, channels>::minParam = 0.0;
45
46  /*
47   * Histogram constructor
48   * @param image the image to use for computing histograms
49   * @param drawColors the drawing colors of the histogram
50   * @param computeGray checks whether to compute 4th gray level
51   * histogram on BGR image or not
52   * @param drawHeight the drawing height of the histogram window
53   * @param drawWidth the drawing width of the histogram window
54   * @param timeCumulation checks whether to compute time cumulative
55   * histograms or not.
56   */
57  template <typename T, size_t channels>
58  CvHistograms<T, channels>::CvHistograms(Mat * image,
59                                          const bool computeGray,
60                                          const size_t drawHeight,
61                                          const size_t drawWidth,
62                                          const bool timeCumulation)
63      : CvProcessor(image),
64        computeGray(computeGray),
65        timeCumulative(timeCumulation),
66        monoTransfertFunc(1, bins, CV_8UC1),
67        colorTransferFunc(1, bins, CV_8UC(channels)),
68        lut(NULL),
69        lutType(NONE),
70        previousLutType(NBTRANS),
71        lutParam(80.0),
72        previousLutParam(80.0),
73        showCumulative(false),
74        histWidth(drawWidth),
75        histHeight(drawHeight),
76        lutWidth(bins),
77        lutHeight(bins),
78        histDisplayFrame(drawHeight, drawWidth, CV_8UC(channels)),
79        lutDisplayFrame(bins, bins, CV_8UC(channels)),
80        outDisplayFrame(image->size(), image->type())
81
82  {
83      // Partial setup since lots has been done in initialisation list above
84      setup(image, false);
85
86      addImage("histogram", &histDisplayFrame);
87      addImage("lut", &lutDisplayFrame);
88      addImage("out", &outDisplayFrame);
89  }
90
```

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

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

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

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

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

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

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

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



```

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

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

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

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

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

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

/*
 * Gets the additionnal gray histogram status
 * @return true if additional gray level histogram is computed,
 * false otherwise
 */
template <typename T, size_t channels>
bool CvHistograms<T, channels>::isComputeGray() const
{
    return computeGray;
}

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

/*
 * Maximum cumulative histograms value;
 * @return the maximum value in all histograms
 * @note regular cumulative maximum value is the number of pixels in
 * the image, but when timecumulative is activated it can be bigger.
 */
template <typename T, size_t channels>
```

```
271 float CvHistograms<T, channels>::getCMaxValue() const
272 {
273     return cMaxValue;
274 }
275
276 /*
277  * Value reading access operator
278  * @param i the ith histogram to access. if i is invalid, 0 is returned
279  * @param j the jth bin value of the ith histogram to access. if j is
280  * invalid, 0 is returned.
281  * @param cumulative checks whether to return regular histogram value
282  * or cumulative histogram value
283  * @return the value in the jth bin of the ith histogram
284  * @par usage :
285  * @code
286  *  float ithValue = myHist(i,j);
287  *  float ithCumulativeValue = myHist(i,j,true);
288  * @endcode
289  */
290 template <typename T, size_t channels>
291 float CvHistograms<T, channels>::operator()(const size_t i,
292                                             const size_t j,
293                                             const bool cumulative) const
294 {
295     if (i < nbHistograms)
296     {
297         if (j < bins)
298         {
299             if (¬cumulative)
300             {
301                 return (const float) histograms[i][j];
302             }
303             else
304             {
305                 return (const float) cumulHistograms[i][j];
306             }
307         }
308         else
309         {
310             cerr << "CvHistograms::operator() const invalid second index "
311                  << "j = " << j << endl;
312             return operator()(i, bins - 1);
313         }
314     }
315     else
316     {
317         cerr << "CvHistograms::operator() const invalid first index i = " << i
318              << endl;
319         return operator()(nbHistograms - 1, j);
320     }
321 }
322
323 /*
324  * Value reading/writing access operator
325  * @param i the ith histogram to access. if i is invalid, 0 is returned
326  * @param j the jth bin value of the ith histogram to access. if j is
327  * invalid, 0 is returned.
328  * @param cumulative checks whether to return regular histogram value
329  * or cumulative histogram value
330  * @return the value in the jth bin of the ith histogram
331  * @par usage :
332  * @code
333  *  float myHist(i,j) = ithValue;
334  *  float myHist(i,j,true) = jthCumulativeValue;
335  * @endcode
336  */
337 template <typename T, size_t channels>
338 float & CvHistograms<T, channels>::operator()(const size_t i,
339                                               const size_t j,
340                                               const bool cumulative)
341 {
342     if (i < nbHistograms)
343     {
344         if (j < bins)
345         {
346             if (¬cumulative)
347             {
348                 return histograms[i][j];
349             }
350             else
351             {
352                 return cumulHistograms[i][j];
353             }
354         }
355         else
356         {
357             cerr << "CvHistograms::operator() invalid second index j = " << j
358                  << endl;
359             return operator()(i, bins - 1);
360         }
```



```
361     }
362     else
363     {
364         cerr << "CvHistograms::operator() invalid first index i = " << i
365              << endl;
366         return operator()(nbHistograms - 1, j);
367     }
368 }
369
370 /*
371  * Time cumulative histogram status read access
372  * @return the time cumulative histogram status
373  */
374 template <typename T, size_t channels>
375 bool CvHistograms<T, channels>::isTimeCumulative() const
376 {
377     return timeCumulative;
378 }
379
380 /*
381  * Time cumulative histogram status read access
382  * @param value the value to set for time cumulative status
383  */
384 template <typename T, size_t channels>
385 void CvHistograms<T, channels>::setTimeCumulative(const bool value)
386 {
387     timeCumulative = value;
388 }
389
390 /*
391  * Cumulative histogram status read access
392  * @return the cumulative histogram status
393  */
394 template <typename T, size_t channels>
395 bool CvHistograms<T, channels>::isCumulative() const
396 {
397     return showCumulative;
398 }
399
400 /*
401  * Cumulative histogram status read access
402  * @param value the value to set for cumulative status
403  */
404 template <typename T, size_t channels>
405 void CvHistograms<T, channels>::setCumulative(const bool value)
406 {
407     showCumulative = value;
408 }
409
410 /*
411  * Ith histogram component shown status read access
412  * @param i the ith histogram component
413  * @return true if this component show status is true
414  */
415 template <typename T, size_t channels>
416 bool CvHistograms<T, channels>::isShowComponent(const size_t i) const
417 {
418     if (i < nbHistograms)
419     {
420         return showComponent[i];
421     }
422     else
423     {
424         return false;
425     }
426 }
427
428 /*
429  * Ith histogram component shown status write access
430  * @param i the ith histogram component
431  * @param value the value to set for this component show status
432  */
433 template <typename T, size_t channels>
434 void CvHistograms<T, channels>::setShowComponent(const size_t i,
435                                                  const bool value)
436 {
437     //  clog << "Set Showcomponent nÂ° " << i << (value ? "true" : "false") <<
438     //endl;
439     if (i < nbHistograms)
440     {
441         showComponent[i] = value;
442     }
443 }
444
445 /**
446  * Update histogram, LUT and resulting images
447  */
448 template <typename T, size_t channels>
449 void CvHistograms<T, channels>::update()
450 {
```

```
451     clock_t start;
452     processTime = 0;
453 
454     // Compute histogram
455     start = clock();
456 
457     updateHistogram();
458 
459     updateHistogramTime1 = clock() - start;
460     processTime += updateHistogramTime1;
461     meanUpdateHistogramTime1 += updateHistogramTime1;
462 
463     // Compute requested LUT
464     start = clock();
465 
466     lut = computeLUT();
467 
468     computeLUTTime = clock() - start;
469     processTime += computeLUTTime;
470     meanComputeLUTTime += computeLUTTime;
471 
472     if (isLUTUpdated())
473     {
474         // draw TransfertFunction to lutDisplayFrame
475         start = clock();
476 
477         drawTransfertFunc(lut);
478 
479         drawLUTTime = clock() - start;
480         processTime += drawLUTTime;
481         meanDrawLUTTime += drawLUTTime;
482     }
483 
484     // Try to apply LUT
485     start = clock();
486 
487     bool lutApplied = drawTransformedImage();
488 
489     applyLUTTime = clock() - start;
490     processTime += applyLUTTime;
491     meanApplyLUTTime += applyLUTTime;
492 
493     if (lutApplied)
494     {
495         // if LUT has been applied histogram should be updated
496         start = clock();
497 
498         updateHistogram();
499 
500         updateHistogramTime2 = clock() - start;
501         processTime += updateHistogramTime2;
502         meanUpdateHistogramTime2 += updateHistogramTime2;
503     }
504     else
505     {
506         updateHistogramTime2 = 0;
507     }
508 
509     // Finally draw Histogram
510     start = clock();
511 
512     drawHistograms();
513 
514     drawHistogramTime = clock() - start;
515     processTime += drawHistogramTime;
516     meanDrawHistogramTime += drawHistogramTime;
517     meanProcessTime += processTime;
518 }
519 
520 /*
521  * Update histograms values.
522  */
523 template <typename T, size_t channels>
524 void CvHistograms<T, channels>::updateHistogram()
525 {
526     maxValue = 0.0;
527     cMaxValue = 0.0;
528 
529     // reset histograms values if necessary
530     if (¬timeCumulative)
531     {
532         // reset histograms values (including evt gray level histogram)
533         for (size_t i = 0; i < nbHistograms; i++)
534         {
535             float * h = histograms[i];
536             for (size_t j = 0; j < bins; j++)
537             {
538                 h[j] = 0.0;
539             }
540         }
```



```
541     }
542 
543     // creating iterators over image
544     MatConstIterator_<Vec<T, channels> > iterator =
545         sourceImage→begin<Vec<T, channels> >();
546     MatConstIterator_<Vec<T, channels> > end =
547         sourceImage→end<Vec<T, channels> >();
548 
549     // updateHistogram histograms values
550     for (; iterator ≠ end; ++iterator)
551     {
552         Vec<T, channels> pixel = *iterator;
553 
554         for (size_t i = 0; i < channels; i++)
555         {
556             // updateHistogram corresponding histogram bin
557             float histValue = ++histograms[i][(size_t) pixel[i]];
558 
559             // updateHistogram max value if needed
560             if (histValue > maxValue)
561             {
562                 maxValue = histValue;
563             }
564         }
565     }
566 
567     // eventually updates gray level histogram
568     if (computeGray ∧ (channels ≡ 3))
569     {
570         for (size_t l = 0; l < channels; l++)
571         {
572             for (size_t i = 0; i < bins; i++)
573             {
574                 histograms[HIST_GRAY][i] += BGR2Gray[l] * histograms[l][i];
575             }
576         }
577     }
578 
579     // update cumulative histograms
580     for (size_t h = 0; h < nbHistograms; h++)
581     {
582         float * regularHistogram = histograms[h];
583         float * cumulativeHistogram = cumulHistograms[h];
584 
585         size_t b;
586         cumulativeHistogram[0] = regularHistogram[0];
587         for (b = 1; b < bins; b++)
588         {
589             cumulativeHistogram[b] =
590                 cumulativeHistogram[b - 1] + regularHistogram[b];
591         }
592 
593         // b == bins now, so checks if last is greater than max value
594         if (cumulativeHistogram[b - 1] > cMaxValue)
595         {
596             cMaxValue = cumulativeHistogram[b - 1];
597         }
598     }
599 }
600 
601 /*
602  * Draws selected histogram(s) in drawing frame and returns the drawing
603  * frame
604  * @return the updated drawing frame.
605  * @post depending on several attributes one or several histograms
606  * have bee drawn in the drawing frame wich is returned
607  *  - if #showCumulative is true then cumulative histograms are drawn
608  *  otherwise regular histograms are drawn
609  *  - each histogram is drawn only if its showComponent[i] is true.
610  */
611 template <typename T, size_t channels>
612 void CvHistograms<T, channels>::drawHistograms()
613 {
614     float curveStep = (float) histWidth / (float) bins;
615     vector<float *> * valuesPtr;
616     float max;
617     if (showCumulative)
618     {
619         valuesPtr = &cumulHistograms;
620         max = cMaxValue;
621     }
622     else
623     {
624         valuesPtr = &histograms;
625         max = maxValue;
626     }
627 
628     // Fill the drawing frame with black
629     rectangle(histDisplayFrame,
630               Point(0, 0),
```

```
631                 Point(histWidth - 1, histHeight - 1),
632                 Scalar(0x00, 0x00, 0x00, 0x00),
633                 CV_FILLED);
634
635     // Draw the bins (reversed)
636     for (size_t h = 0; h < nbHistograms; h++)
637     {
638         // fills this color histogram frame with black
639         rectangle(histComponents[h],
640                   Point(0, 0),
641                   Point(histWidth - 1, histHeight - 1),
642                   Scalar(0x00, 0x00, 0x00, 0x00),
643                   CV_FILLED);
644
645         // if this color histogram should be drawn
646         if (showComponent[h])
647         {
648             for (size_t i = 0; i < bins; i++)
649             {
650                 // draws each bin (reversed) in this color hist. frame
651                 rectangle(
652                     histComponents[h], // the image to draw in
653                     Point(i * curveStep,
654                           histHeight - 1),   // first corner of this bin
655                     Point((i + 1) * curveStep, // second corner of this bin
656                           histHeight - 1 -
657                               cvRound(((*valuesPtr)[h][i] / max) * histHeight)),
658                     displayColors[h], // current color
659                     CV_FILLED,        // filled rectangle
660                     CV_AA);           // antialiased line
661             }
662             // adds this color histogram frame to the drawing frame
663             add(histDisplayFrame, histComponents[h], histDisplayFrame);
664         }
665     }
666 }
667
668 /*
669  * Draws selected transfert function in drawing frame and returns the
670  * drawing frame
671  * @param lut the LUT to draw : the LUT may contains 1 or several
672  * channels
673  * @return the updated drawing frame
674  */
675 template <typename T, size_t channels>
676 void CvHistograms<T, channels>::drawTransfertFunc(const Mat * lut)
677 {
678     float curveStep = (float) lutWidth / (float) bins;
679
680     const Mat * currentLUT;
681
682     if (lut ≠ NULL)
683     {
684         currentLUT = lut;
685     }
686     else // identity LUT should be computed
687     {
688         currentLUT = computeLinearGrayLUT();
689     }
690
691     size_t lutChannels = (size_t) currentLUT→channels();
692
693     // Fill the drawing frame with black
694     rectangle(lutDisplayFrame,
695               Point(0, 0),
696               Point(lutWidth - 1, lutHeight - 1),
697               Scalar(0x00, 0x00, 0x00, 0x00),
698               CV_FILLED);
699
700     // Draw the bins (reversed)
701     if (lutChannels ≡ 1)
702     {
703         // draws directly in histDisplayFrame with white color
704         for (size_t i = 0; i < bins; i++)
705         {
706             rectangle(
707                 lutDisplayFrame,                    // the image to draw in
708                 Point(i * curveStep, lutHeight - 1), // first corner of this bin
709                 Point((i + 1) * curveStep, // second corner of this bin
710                       lutHeight - 1 -
711                           cvRound(((float) currentLUT→at<T>(0, i) / bins) *
712                                   lutHeight)),
713                 displayColors[3], // current color
714                 CV_FILLED,        // filled rectangle
715                 CV_AA);           // antialiased line
716         }
717     }
718     else // lutChannels == 3 or others
719     {
720         // draws in each colorLUTFrams and adds it to histDisplayFrame
```



```
721         for (size_t c = 0; c < lutChannels; c++)
722         {
723             if (showComponent[c])
724             {
725                 // Fill the color drawing frame with black
726                 rectangle(lutComponents[c],
727                           Point(0, 0),
728                           Point(lutWidth - 1, lutHeight - 1),
729                           Scalar(0x00, 0x00, 0x00, 0x00),
730                           CV_FILLED);
731
732                 for (size_t i = 0; i < bins; i++)
733                 {
734                     rectangle(lutComponents[c], // the image to draw in
735                         Point(i * curveStep,
736                               lutHeight - 1), // first corner of this bin
737                         Point((i + 1) * curveStep, // second corner of this bin
738                               lutHeight - 1 -
739                               cvRound(((float)
740                                       currentLUT→at<Vec<T, channels> >(0, i)[c] / bins) *
741                                   lutHeight)),
742                         displayColors[c], // current color
743                         CV_FILLED,        // filled rectangle
744                         CV_AA);           // antialiased line
745                 }
746                 add(lutDisplayFrame, lutComponents[c], lutDisplayFrame);
747             }
748         }
749     }
750 }
751
752 /*
753  * Indicates if LUT has been updated or if it has not changed
754  * @return true if LUT has been updated
755  */
756 template <typename T, size_t channels>
757 bool CvHistograms<T, channels>::isLUTUpdated() const
758 {
759     return lutUpdated;
760 }
761
762 /*
763  * Gets the current LUT type
764  * @return the current LUT type
765  */
766 template <typename T, size_t channels>
767 typename CvHistograms<T, channels>::TransfertType
768 CvHistograms<T, channels>::getLutType() const
769 {
770     return lutType;
771 }
772
773 /*
774  * Sets the current LUT type
775  * @param lutType the new LUT type
776  */
777 template <typename T, size_t channels>
778 void CvHistograms<T, channels>::setLutType(const TransfertType lutType)
779 {
780     previousLutType = this→lutType;
781
782     computeLUTTime = 0;
783     drawLUTTime = 0;
784     applyLUTTime = 0;
785     updateHistogramTime2 = 0;
786
787     if (lutType < NBTRANS)
788     {
789         this→lutType = lutType;
790     }
791     else
792     {
793         this→lutType = NONE;
794     }
795 }
796
797 /*
798  * Gets the current parameter value for LUTs using a percentage parameter
799  * @return the current LUT parameter
800  */
801 template <typename T, size_t channels>
802 float CvHistograms<T, channels>::getLUTParam() const
803 {
804     return lutParam;
805 }
806
807 /*
808  * Sets the current LUT % parameter
809  * @param lutParam the new LUT parameter
810  */
```

```
811 template <typename T, size_t channels>
812 void CvHistograms<T, channels>::setLUTParam(const float currentParam)
813 {
814     previousLutParam = lutParam;
815
816     if (currentParam > maxParam)
817     {
818         this→lutParam = maxParam;
819     }
820     else if (currentParam < minParam)
821     {
822         this→lutParam = minParam;
823     }
824     else
825     {
826         this→lutParam = currentParam;
827     }
828 }
829
830 /*
831  * Return processor processing time of step index [default implementation
832  * returning only processTime, should be reimplemented in subclasses]
833  * @param index index of the step which processing time is required,
834  * 0 indicates all steps, and values above 0 indicates step #. If
835  * required index is bigger than number of steps than all steps value
836  * should be returned.
837  * @return the processing time of step index.
838  * @note should be reimplemented in subclasses in order to define
839  * time/feature behaviour
840  */
841 template <typename T, size_t channels>
842 double CvHistograms<T, channels>::getProcessTime(const size_t index) const
843 {
844     switch (index)
845     {
846         case (CvHistograms<T, channels>::UPDATE_HISTOGRAM):
847             return (double) updateHistogramTime1;
848         case (CvHistograms<T, channels>::COMPUTE_LUT):
849             return (double) computeLUTTime;
850         case (CvHistograms<T, channels>::DRAW_LUT):
851             return (double) drawLUTTime;
852         case (CvHistograms<T, channels>::APPLY_LUT):
853             return (double) applyLUTTime;
854         case (CvHistograms<T, channels>::UPDATE_HISTOGRAM_AFTER_LUT):
855             return (double) updateHistogramTime2;
856         case (CvHistograms<T, channels>::DRAW_HISTOGRAM):
857             return (double) drawHistogramTime;
858         default:
859             return (double) processTime;
860     }
861 }
862
863 /*
864  * Return processor mean processing time of step index [default
865  * implementation returning only processTime, should be reimplemented
866  * in subclasses]
867  * @param index index of the step which processing time is required,
868  * 0 indicates all steps, and values above 0 indicates step #. If
869  * required index is bigger than number of steps than all steps value
870  * should be returned.
871  * @return the mean processing time of step index.
872  * @note should be reimplemented in subclasses in order to define
873  * time/feature behaviour
874  * @param index
875  */
876 template <typename T, size_t channels>
877 double CvHistograms<T, channels>::getMeanProcessTime(const size_t index) const
878 {
879     switch (index)
880     {
881         case (CvHistograms<T, channels>::UPDATE_HISTOGRAM):
882             return (double) meanUpdateHistogramTime1.mean();
883         case (CvHistograms<T, channels>::COMPUTE_LUT):
884             return (double) meanComputeLUTTime.mean();
885         case (CvHistograms<T, channels>::DRAW_LUT):
886             return (double) meanDrawLUTTime.mean();
887         case (CvHistograms<T, channels>::APPLY_LUT):
888             return (double) meanApplyLUTTime.mean();
889         case (CvHistograms<T, channels>::UPDATE_HISTOGRAM_AFTER_LUT):
890             return (double) meanUpdateHistogramTime2.mean();
891         case (CvHistograms<T, channels>::DRAW_HISTOGRAM):
892             return (double) meanDrawHistogramTime.mean();
893         default:
894             return (double) meanProcessTime.mean();
895     }
896 }
897
898 /*
899  * Return processor processing time std of step index [default
900  * implementation returning only processTime, should be reimplemented
```



```
901  * in subclasses]
902  * @param index index of the step which processing time is required,
903  * 0 indicates all steps, and values above 0 indicates step #. If
904  * required index is bigger than number of steps than all steps value
905  * should be returned.
906  * @return the mean processing time of step index.
907  * @note should be reimplemented in subclasses in order to define
908  * time/feature behaviour
909  * @param index
910  */
911 template <typename T, size_t channels>
912 double CvHistograms<T, channels>::getStdProcessTime(const size_t index) const
913 {
914     switch (index)
915     {
916         case (CvHistograms<T, channels>::UPDATE_HISTOGRAM):
917             return (double) meanUpdateHistogramTime1.std();
918         case (CvHistograms<T, channels>::COMPUTE_LUT):
919             return (double) meanComputeLUTTime.std();
920         case (CvHistograms<T, channels>::DRAW_LUT):
921             return (double) meanDrawLUTTime.std();
922         case (CvHistograms<T, channels>::APPLY_LUT):
923             return (double) meanApplyLUTTime.std();
924         case (CvHistograms<T, channels>::UPDATE_HISTOGRAM_AFTER_LUT):
925             return (double) meanUpdateHistogramTime2.std();
926         case (CvHistograms<T, channels>::DRAW_HISTOGRAM):
927             return (double) meanDrawHistogramTime.std();
928         default:
929             return (double) meanProcessTime.std();
930     }
931 }
932
933 /*
934  * Reset mean and std process time in order to re-start computing
935  * new mean and std process time values.
936  */
937 template <typename T, size_t channels>
938 void CvHistograms<T, channels>::resetMeanProcessTime()
939 {
940     CvProcessor::resetMeanProcessTime();
941     meanUpdateHistogramTime1.reset();
942     meanComputeLUTTime.reset();
943     meanDrawLUTTime.reset();
944     meanApplyLUTTime.reset();
945     meanUpdateHistogramTime2.reset();
946     meanDrawHistogramTime.reset();
947 }
948
949 /*
950  * Compute linear transfert function (LUT) : no change in image levels
951  * @return the LUT containing the corresponding transfert function,
952  * the returned matrix contains only one channel corresponding to
953  * the graylevel LUT which should be applied to all color channels of
954  * the image
955  * @post the result is stored in monoTransfertFunc
956  * @note It's useless to compute a color Linear LUT since all channels
957  * would contain the exact same values.
958  */
959 template <typename T, size_t channels>
960 Mat * CvHistograms<T, channels>::computeLinearGrayLUT()
961 {
962     for (size_t i = 0; i < bins; i++)
963     {
964         monoTransfertFunc.at<T>(0, i) = i;
965     }
966
967     return &monoTransfertFunc;
968 }
969
970 /*
971  * Compute linear transfert function (LUT) : no change in image levels
972  * @return the LUT containing the corresponding transfert function,
973  * the returned matrix contains 3 channels corresponding to
974  * the color LUT which should be applied to all color channels of
975  * the image
976  * @post the result is stored in colorTransfertFunc
977  * @note It's useless to compute a color Linear LUT since all channels
978  * would contain the exact same values.
979  */
980 template <typename T, size_t channels>
981 Mat * CvHistograms<T, channels>::computeLinearColorLUT()
982 {
983     for (size_t c = 0; c < channels; c++)
984     {
985         for (size_t i = 0; i < bins; i++)
986         {
987             colorTransferFunc.at<Vec<T, channels> >(0, i)[c] = i;
988         }
989     }
990
```

```
991     return &colorTransferFunc;
992 }
993
994 /*
995  * Compute the optimal dynamic LUT for preserving "percentDynamic"
996  * percent of the whole image lightness range.
997  * @param percentDynamic the gray level percentage to spread on the
998  * whole (100%) gray level range in the image
999  * @return the LUT containing the corresponding transfert function,
1000 * the returned matrix contains only one channel corresponding to
1001 * the graylevel LUT which should be applied to all color channels of
1002 * the image
1003 * @post the result is stored in monoTransfertFunc
1004 *          ^
1005 * maxVal   |                ________
1006 *          |              /
1007 *          |            /
1008 *          |          /          dy/dx = (maxVal-minVal) / (maxThresIndex - minthresIndex - 1)
1009 *          |        /
1010 *          |      /
1011 * minVal   |     /     maxThresIndex
1012 *          +----+------+------------->
1013 *   minThresIndex
1014 */
1015 template <typename T, size_t channels>
1016 Mat * CvHistograms<T, channels>::computeGrayOptimalLUT(const unsigned int percentDynamic)
1017 {
1018     if (computeGray ∧ nbHistograms ≡ 4)
1019     {
1020         float threshold = (100 - percentDynamic) / 200.0;
1021         float imageSize = sourceImage→rows * sourceImage→cols;
1022         float minThres = imageSize * threshold;
1023         float maxThres = imageSize - minThres;
1024
1025         size_t minThresIndex = 0;
1026         size_t maxThresIndex = bins;
1027
1028         T minVal = 0;
1029         T maxVal = numeric_limits<T>::max(); // 255 for uchar
1030
1031         // finds minThresIndex in cumulHistograms[HIST_GRAY][i=0..bins]
1032         // TODO Ã  complÃ©ter ...
1033
1034         // finds maxThresIndex in cumulHistograms[HIST_GRAY][i=0..bins]
1035         // TODO Ã  complÃ©ter ...
1036
1037         // fill monoTransfertFunc before minThresIndex with minVal
1038         // TODO Ã  complÃ©ter ...
1039
1040         // fill monoTransfertFunc between minThresIndex and maxThesIndex with Dy/Dx Values
1041         float slope = (float)(maxVal - minVal) /
1042                       (float)(maxThresIndex-1-minThresIndex);
1043
1044         // TODO Ã  complÃ©ter ...
1045
1046         // fill monoTransfertFunc after maxThresIndex with maxVal
1047         // TODO Ã  complÃ©ter ...
1048     }
1049     else
1050     {
1051         cerr << "CvHistograms<T,channels>::computeGrayOptimalLUT : "
1052              << "There is no gray histogram !" << endl;
1053     }
1054
1055     return &monoTransfertFunc;
1056 }
1057
1058 /*
1059  * Compute the optimal dynamic LUTs (one for each channel) for preserving
1060  * "percentDynamic" percent of the whole image color ranges.
1061  * @param percentDynamic the colors level percentage to spread on the
1062  * whole (100%) colors level range in the image
1063  * @return the LUT containing the corresponding transfert functions,
1064  * the returned matrix contains as much channels as the image and
1065  * corresponding to the color level LUT which should be applied to
1066  * each color channels of the image
1067  * @post the result is stored in colorTransfertFunc
1068  */
1069 template <typename T, size_t channels>
1070 Mat * CvHistograms<T, channels>::computeColorOptimalLUT(const unsigned int percentDynamic)
1071 {
1072     float threshold = (1 - (percentDynamic / 100.0)) / 2.0;
1073     float imageSize = sourceImage→rows * sourceImage→cols;
1074     float minThres = imageSize * threshold;
1075     float maxThres = imageSize - minThres;
1076
1077     size_t minThresIndex[channels];
1078     size_t maxThresIndex[channels];
1079     T minVal = 0;
1080     T maxVal = numeric_limits<T>::max(); // 255 for uchar;
```



```
1081     float slope[channels];
1082
1083     for (size_t c = 0; c < channels; c++)
1084     {
1085         minThresIndex[c] = 0;
1086         maxThresIndex[c] = bins;
1087
1088         // finds minThresIndex in cumulHistograms[c][...] for this channel
1089         // TODO Ã  complÃ©ter ...
1090
1091         // finds maxThresIndex in cumulHistograms[c][...] for this channel
1092         // TODO Ã  complÃ©ter ...
1093
1094         // fill colorTransferFunc before minThresIndex with minVal
1095         // TODO Ã  complÃ©ter ...
1096
1097         // ramp slope for this channel = Dy/Dx
1098         slope[c] = (float)(maxVal - minVal) /
1099                    (float)(maxThresIndex[c]-1-minThresIndex[c]);
1100
1101         // fill colorTransferFunc between minThresIndex and maxThesIndex with regular ramp
1102         // TODO Ã  complÃ©ter ...
1103
1104         // fill colorTransferFunc after maxThresIndex with maxVal
1105         // TODO Ã  complÃ©ter ...
1106     }
1107
1108     return &colorTransferFunc;
1109 }
1110
1111 /*
1112  * Computes the transfert function corresponding to gray level
1113  * equalization
1114  * @return the matrix containing the gray level equalization LUT to
1115  * apply on the image
1116  * @post the result is stored in monoTransfertFunc
1117  */
1118 template <typename T, size_t channels>
1119 Mat * CvHistograms<T, channels>::computeGrayEqualizeLUT()
1120 {
1121     T maxVal = numeric_limits<T>::max();
1122     if (computeGray ∧ nbHistograms ≡ 4)
1123     {
1124         /*
1125          * Equalisation consists in applying the corresponding cumulative
1126          * histogram (cumulHistograms[HIST_GRAY][i=0..bins] normalized to
1127          * maxVal)
1128          * as a mono transfert function
1129          */
1130         // TODO Ã  complÃ©ter ...
1131     }
1132     else
1133     {
1134         cerr << "CvHistograms<T,channels>::computeGrayEqualizeLUT : "
1135              << "There is no gray level histogram" << endl;
1136     }
1137
1138     return &monoTransfertFunc;
1139 }
1140
1141 /*
1142  * Computes the transfert functions corresponding to each channel
1143  * level equalization
1144  * @return the matrix containg each channel level equalization LUT to
1145  * apply on the image
1146  * @post the result is stored in colorTransferFunc
1147  */
1148 template <typename T, size_t channels>
1149 Mat * CvHistograms<T, channels>::computeColorEqualizeLUT()
1150 {
1151     // 255 for uchar;
1152     T maxVal = numeric_limits<T>::max();
1153     /*
1154      * Color equalisation consists in applying the corresponding cumulative
1155      * histogram (cumulHistograms[c=0..channels][i=0..bins] normalized to
1156      * maxVal)
1157      * as a color transfert function
1158      */
1159     for (size_t c = 0; c < channels; c++)
1160     {
1161         // TODO Ã  complÃ©ter ...
1162     }
1163
1164     return &colorTransferFunc;
1165 }
1166
1167 /*
1168  * Compute the LUT corresponding to thresholded image with tPercent
1169  * of the pixel population on each side of the threshold according
1170  * to the cumulative gray level histogram
```

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

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

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

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

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

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

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

    return &monoTransfertFunc;
}

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

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

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

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

            mThresIndex[c] = thresIndex;

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

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



```
    {
        cerr << "CvHistograms<T,channels>::computeColorThresholdLUT : "
             << "percentage should be between 0 and 100 : " << tPercent << endl;
    }

    return &colorTransferFunc;
}

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

    // Apply (x^gamma)*bins where x=i/bins in monoTransfertFunc
    // TODO Ã  complÃ©ter ...

    return &monoTransfertFunc;
}

/*
 * Compute the LUT corresponding to negative image
 * @return the matrix containing the negative LUT (mono)
 */
template <typename T, size_t channels>
Mat * CvHistograms<T, channels>::computeNegativeLUT()
{
    // Apply (bins - 1 -i) in  monoTransfertFunc
    // TODO Ã  complÃ©ter ...

    return &monoTransfertFunc;
}

/*
 * Compute and returns the current transfert function to be applied
 * on the image, eventually with the current LUT parameter
 * @return the mono or color LUT matrix to apply on the image depending
 * on the lutType
 * @see TransfertType
 */
template <typename T, size_t channels>
Mat * CvHistograms<T, channels>::computeLUT()
{
    Mat * lut = NULL;

    lutUpdated = true;

    switch (lutType)
    {
        case NONE:
            /*
             * Identity LUT
             * Linear LUT does not depend on histogram so if previous
             * LUT was already Linear then don't compute it again, just
             * return the last LUT
             */
            if (previousLutType ≠ lutType)
            {
                lut = computeLinearGrayLUT();
            }
            else
            {
                lut = &monoTransfertFunc;
                lutUpdated = false;
            }
            break;
        case THRESHOLD_GRAY:
            /*
             * LUT to split pixels below param % to black and pixels over
             * param % to white based on graylevel cumulative histogram
             */
            lut = computeGrayThresholdLUT(lutParam);
            break;
        case THRESHOLD_COLOR:
            /*
             * LUT to split param% of the pixel components to black and
             * 100-param% to full B, G or R based on cumulative histograms
             * components
             */
            lut = computeColorThresholdLUT(lutParam);
            break;
```

```
1351            case DYNAMIC_GRAY:
1352                /*
1353                 * LUT to spread param% of the pixel levels over 100% of the dynamic
1354                 * based on cumulative gray level histogram
1355                 */
1356                lut = computeGrayOptimalLUT(lutParam);
1357                break;
1358            case DYNAMIC_COLOR:
1359                /*
1360                 * LUT to spread param% of the pixel components levels over 100% of
1361                 * the dynamic based on cumulative color histograms
1362                 */
1363                lut = computeColorOptimalLUT(lutParam);
1364                break;
1365            case EQUALIZE_GRAY:
1366                /*
1367                 * Gray level histogram equalization LUT
1368                 */
1369                lut = computeGrayEqualizeLUT();
1370                break;
1371            case EQUALIZE_COLOR:
1372                /*
1373                 * Color components histograms equalization LUTs
1374                 */
1375                lut = computeColorEqualizeLUT();
1376                break;
1377            case GAMMA:
1378                /*
1379                 * Gamma LUT does not depend on histogram so if previous
1380                 * LUT was already Gamma then don't compute it again, just
1381                 * return the last LUT
1382                 */
1383                if ((previousLutType ≠ lutType) ∨ (previousLutParam ≠ lutParam))
1384                {
1385                    lut = computeGammaLUT(lutParam);
1386                }
1387                else
1388                {
1389                    lut = &monoTransfertFunc;
1390                    lutUpdated = false;
1391                }
1392                break;
1393            case NEGATIVE:
1394                /*
1395                 * Negative LUT does not depend on histogram so if previous
1396                 * LUT was already Negative then don't compute it again, just
1397                 * return the last LUT
1398                 */
1399                if (previousLutType ≠ lutType)
1400                {
1401                    lut = computeNegativeLUT();
1402                }
1403                else
1404                {
1405                    lut = &monoTransfertFunc;
1406                    lutUpdated = false;
1407                }
1408                break;
1409            default:
1410                cerr << "CvHistograms<T,channels>::applyLUT : unknown LUT" << endl;
1411                break;
1412        }
1413
1414        if ((previousLutType ≠ lutType) ∨ (previousLutParam ≠ lutParam))
1415        {
1416            resetMeanProcessTime();
1417        }
1418
1419        previousLutType = lutType;
1420        previousLutParam = lutParam;
1421
1422        return lut;
1423    }
1424
1425    /*
1426     * Apply current LUT (if != NULL) to the source image to produce the
1427     * outFrame
1428     * @return true if LUT has been applied, false if lut is NULL or
1429     * lutType is NONE
1430     */
1431    template <typename T, size_t channels>
1432    bool CvHistograms<T, channels>::drawTransformedImage()
1433    {
1434        if ((lut ≠ NULL) ∧ (lutType ≠ NONE))
1435        {
1436            LUT(*sourceImage, *lut, outDisplayFrame);
1437            return true;
1438        }
1439        else
1440        {
```



```
1441            outDisplayFrame = *sourceImage;
1442            return false;
1443        }
1444    }
1445
1446    /*
1447     * output operator for Histograms
1448     * @param out the output stream
1449     * @param h the histograms to print on the stream
1450     * @return a reference to the output stream so it can be cumulated
1451     */
1452    template <typename T, size_t channels>
1453    ostream & operator<<(ostream & out, const CvHistograms<T, channels> & h)
1454    {
1455        for (size_t i = 0; i < h.bins; i++)
1456        {
1457            out << i << ":";
1458
1459            for (size_t j = 0; j < h.nbHistograms; j++)
1460            {
1461                out << h.histograms[i][j] << " ";
1462            }
1463
1464            out << endl;
1465        }
1466
1467        return out;
1468    }
1469
1470    // ============================================================================
1471    // Templates proto instanciations
1472    // ============================================================================
1473
1474    // template class instanciation
1475    // for gray level images
1476    template class CvHistograms<uchar, 1>;
1477    template ostream & operator<<(ostream &, const CvHistograms<uchar, 1> &);
1478
1479    // for BGR or YUV images
1480    template class CvHistograms<uchar, 3>;
1481    template ostream & operator<<(ostream &, const CvHistograms<uchar, 3> &);
```

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

#ifndef QCVEHISTOGRAMS_H_
#define QCVEHISTOGRAMS_H_

#include <QString>
#include <QMutex>

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

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

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

    protected:

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

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

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

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

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

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

        /**
         * Self lock for operations from multiple threads
         * @note may be NULL if there is no update thread
         */
        QMutex * selfLock;

    public:

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



```
                      const size_t drawHeight = 256,
                      const size_t drawWidth = 512,
                      const bool timeCumulation = false,
                      QObject * parent = NULL);

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

    protected:

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

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

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

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

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

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

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

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

        /**
         * Reset mean and std process time in order to re-start computing
         * new mean and std process time values.
         */
        virtual void resetMeanProcessTime();

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

        /**
         * Signal sent when transformed image has been reallocated
         * @param image the new transformed image
         */
```

```
181         void outImageChanged(Mat * image);
182
183         /**
184          * Signal sent when update is completed AND histogram image changes
185          */
186         void histogramImageUpdated();
187
188         /**
189          * Signal sent when histogram image has been reallocated
190          * @param image the new histogram image
191          */
192         void histogramImageChanged(Mat * image);
193
194         /**
195          * Signal sent when update is completed AND LUT image changes
196          */
197         void lutImageUpdated();
198
199         /**
200          * Signal sent when lut image has been reallocated;
201          * @param image the new LUT image
202          */
203         void lutImageChanged(Mat * image);
204
205         /**
206          * Signal emitted when histogram is updated with a new image
207          * @param formattedValue string containing the formatted time value
208          */
209         void histogramTime1Updated(const QString & formattedValue);
210
211         /**
212          * Signal emitted when LUT is computed
213          * @param formattedValue string containing the formatted time value
214          */
215         void computeLUTTimeUpdated(const QString & formattedValue);
216
217         /**
218          * Signal emitted when LUT is drawn
219          * @param formattedValue string containing the formatted time value
220          */
221         void drawLUTTimeUpdated(const QString & formattedValue);
222
223         /**
224          * Signal emitted when LUT is applied on image
225          * @param formattedValue string containing the formatted time value
226          */
227         void applyLUTTimeUpdated(const QString & formattedValue);
228
229         /**
230          * Signal emitted when histogram is updated after LUT has been applied
231          * @param formattedValue string containing the formatted time value
232          */
233         void histogramTime2Updated(const QString & formattedValue);
234
235         /**
236          * Signal emitted when histogram is drawn
237          * @param formattedValue string containing the formatted time value
238          */
239         void drawHistogramTimeUpdated(const QString & formattedValue);
240 };
241
242 #endif /* QCVEHISTOGRAMS_H_ */
```



```
1  /*
2   * QcvHistograms.cpp
3   *
4   *  Created on: 14 fÃ©vr. 2012
5   *      Author: davidroussel
6   */
7
8  #include <QDebug>
9  #include "QcvHistograms.h"
10
11 /*
12  * QcvHistograms constructor
13  * @param image the source image
14  * @param computeGray indicates if an aditionnal gray level histogram
15  * should be computed
16  * @param drawHeight histogram drawing height
17  * @param drawWidth histogram drawing width
18  * @param timeCumulation indicates if timecumulation is on for histogram
19  * @param imageLock the mutex for concurrent access to the source image.
20  * In order to avoid concurrent access to the same image
21  * @param updateThread the thread in which this processor should run
22  * @param parent parent QObject
23  */
24 QcvHistograms::QcvHistograms(Mat * image,
25                              QMutex * imageLock,
26                              QThread * updateThread,
27                              const bool computeGray,
28                              const size_t drawHeight,
29                              const size_t drawWidth,
30                              const bool timeCumulation,
31                              QObject * parent) :
32     CvProcessor(image),
33     QcvProcessor(image, imageLock, updateThread, parent),
34     CvHistograms8UC3(image, computeGray, drawHeight, drawWidth, timeCumulation),
35     selfLock(updateThread ≠ NULL ? new QMutex() :
36                                    (imageLock ≠ NULL ? imageLock : NULL))
37 {
38     QcvProcessor::setNumberFormat("%7.0f");
39 }
40
41 /*
42  * QImageHistogram destructor
43  */
44 QcvHistograms::~QcvHistograms()
45 {
46     updateHistogramTime1String.clear();
47     computeLUTTimeString.clear();
48     drawLUTTimeString.clear();
49     applyLUTTimeString.clear();
50     updateHistogramTime2String.clear();
51     drawHistogramTimeString.clear();
52
53     if (selfLock ≠ NULL)
54     {
55         selfLock→lock();
56         selfLock→unlock();
57         delete selfLock;
58     }
59 }
60
61 /*
62  * Update computed images and sends displayImageChanged signal if
63  * required
64  */
65 void QcvHistograms::update()
66 {
67     bool hasSourceLock = (sourceLock ≠ NULL) ∧ (sourceLock ≠ selfLock);
68     if (hasSourceLock)
69     {
70         sourceLock→lock();
71     }
72
73     bool hasLock = selfLock ≠ NULL;
74     if (hasLock)
75     {
76         selfLock→lock();
77     }
78
79     /*
80      * Update Histogram images
81      */
82     CvHistograms8UC3::update();
83
84     if (hasLock)
85     {
86         selfLock→unlock();
87     }
88
89     if (hasSourceLock)
90     {
```

```
91          sourceLock→unlock();
92      }
93
94      /*
95       * emit time measurement signals
96       */
97      const char * format = meanStdFormat.toStdString().c_str();
98      updateHistogramTime1String.sprintf(format,
99                                         getMeanProcessTime(UPDATE_HISTOGRAM) / 1000.0,
100                                        getStdProcessTime(UPDATE_HISTOGRAM) / 1000.0);
101     emit(histogramTime1Updated(updateHistogramTime1String));
102
103     computeLUTTimeString.sprintf(format,
104                                  getMeanProcessTime(COMPUTE_LUT) / 1000.0,
105                                  getStdProcessTime(COMPUTE_LUT) / 1000.0);
106     emit(computeLUTTimeUpdated(computeLUTTimeString));
107     if (isLUTUpdated())
108     {
109         drawLUTTimeString.sprintf(format,
110                                   getMeanProcessTime(DRAW_LUT) / 1000.0,
111                                   getStdProcessTime(DRAW_LUT) / 1000.0);
112         emit(drawLUTTimeUpdated(drawLUTTimeString));
113     }
114
115     applyLUTTimeString.sprintf(format,
116                                getMeanProcessTime(APPLY_LUT) / 1000.0,
117                                getStdProcessTime(APPLY_LUT) / 1000.0);
118     emit(applyLUTTimeUpdated(applyLUTTimeString));
119
120     if ((lut ≠ NULL) ∧ (lutType ≠ NONE))
121     {
122         updateHistogramTime2String.sprintf(format,
123                                            getMeanProcessTime(UPDATE_HISTOGRAM_AFTER_LUT) / 1000.0,
124                                            getStdProcessTime(UPDATE_HISTOGRAM_AFTER_LUT) / 1000.0);
125         emit(histogramTime2Updated(updateHistogramTime2String));
126     }
127
128     drawHistogramTimeString.sprintf(format,
129                                     getMeanProcessTime(DRAW_HISTOGRAM) / 1000.0,
130                                     getStdProcessTime(DRAW_HISTOGRAM) / 1000.0);
131     emit(drawHistogramTimeUpdated(drawHistogramTimeString));
132
133     /*
134      * emit updated signal
135      */
136     QcvProcessor::update(); // emits updated signal
137 }
138
139 /*
140  * Changes source image slot.
141  * Attributes needs to be cleaned up then set up again
142  * @param image the new source Image
143  */
144 void QcvHistograms::setSourceImage(Mat * image) throw (CvProcessorException)
145 {
146     Size previousSize(sourceImage→size());
147     int previousNbChannels(nbChannels);
148     bool hasLock = selfLock ≠ NULL;
149     if (hasLock)
150     {
151         selfLock→lock();
152     }
153
154     CvProcessor::setSourceImage(image);
155
156     if (hasLock)
157     {
158         selfLock→unlock();
159     }
160
161     emit imageChanged(sourceImage);
162
163     emit imageChanged();
164
165     if ((previousSize.width ≠ image→cols) ∨
166         (previousSize.height ≠ image→rows))
167     {
168         emit imageSizeChanged();
169     }
170
171     if (previousNbChannels ≠ nbChannels)
172     {
173         emit imageColorsChanged();
174     }
175
176     // notifies any connected component to change source images
177     emit outImageChanged(&outDisplayFrame);
178     emit histogramImageChanged(&histDisplayFrame);
179     emit lutImageChanged(&lutDisplayFrame);
180 }
```



```
181
182
183 /*
184  * Time cumulative histogram status read access
185  * @param value the value to set for time cumulative status
186  */
187 void QcvHistograms::setTimeCumulative(const bool value)
188 {
189     bool hasLock = selfLock ≠ NULL;
190     if (hasLock)
191     {
192         selfLock→lock();
193     }
194
195     CvHistograms8UC3::setTimeCumulative(value);
196
197     if (hasLock)
198     {
199         selfLock→unlock();
200     }
201
202     message.clear();
203     message.append(tr("Time Cumulative Histogram is "));
204     if (value)
205     {
206         message.append(tr("on"));
207     }
208     else
209     {
210         message.append(tr("off"));
211     }
212     emit sendMessage(message, defaultTimeOut);
213 }
214
215 /*
216  * Cumulative histogram status read access
217  * @param value the value to set for cumulative status
218  */
219 void QcvHistograms::setCumulative(const bool value)
220 {
221     bool hasLock = selfLock ≠ NULL;
222     if (hasLock)
223     {
224         selfLock→lock();
225     }
226
227     CvHistograms8UC3::setCumulative(value);
228
229     if (hasLock)
230     {
231         selfLock→unlock();
232     }
233
234     message.clear();
235     message.append(tr("Cumulative Histogram is "));
236     if (value)
237     {
238         message.append(tr("on"));
239     }
240     else
241     {
242         message.append(tr("off"));
243     }
244
245     emit sendMessage(message, defaultTimeOut);
246 }
247
248 /*
249  * Ith histogram component shown status write access
250  * @param i the ith histogram component
251  * @param value the value to set for this component show status
252  */
253 void QcvHistograms::setShowComponent(const size_t i, const bool value)
254 {
255     bool hasLock = selfLock ≠ NULL;
256     if (hasLock)
257     {
258         selfLock→lock();
259     }
260
261     CvHistograms8UC3::setShowComponent(i, value);
262
263     if (hasLock)
264     {
265         selfLock→unlock();
266     }
267
268     message.clear();
269     switch (i)
270     {
```

```
271             case 0:
272                 message.append(tr("Red"));
273                 break;
274             case 1:
275                 message.append(tr("Green"));
276                 break;
277             case 2:
278                 message.append(tr("Blue"));
279                 break;
280             case 3:
281                 message.append(tr("Gray"));
282                 break;
283             default:
284                 message.append(tr("Unkown"));
285                 break;
286         }
287         message.append(tr(" histogram Component is "));
288 
289         if (value)
290         {
291             message.append(tr("on"));
292         }
293         else
294         {
295             message.append(tr("off"));
296         }
297 
298         emit sendMessage(message, defaultTimeOut);
299     }
300 
301     /*
302      * Sets the current LUT type
303      * @param lutType the new LUT type
304      */
305     void QcvHistograms::setLutType(const TransfertType lutType)
306     {
307         bool hasLock = selfLock ≠ NULL;
308         if (hasLock)
309         {
310             selfLock→lock();
311         }
312 
313         CvHistograms8UC3::setLutType(lutType);
314 
315         if (hasLock)
316         {
317             selfLock→unlock();
318         }
319 
320         message.clear();
321         message.append(tr("Current transfert function is "));
322         switch (lutType)
323         {
324             case NONE:
325                 message.append(tr("Identity"));
326                 break;
327             case THRESHOLD_GRAY:
328                 message.append(tr("Threshold based on gray histogram"));
329                 break;
330             case DYNAMIC_GRAY:
331                 message.append(tr("Optimal dynamic based on gray histogram"));
332                 break;
333             case EQUALIZE_GRAY:
334                 message.append(tr("Equalize based on gray histogram"));
335                 break;
336             case THRESHOLD_COLOR:
337                 message.append(tr("Threshold based on color histograms"));
338                 break;
339             case DYNAMIC_COLOR:
340                 message.append(tr("Optimal dynamic based on color histograms"));
341                 break;
342             case EQUALIZE_COLOR:
343                 message.append(tr("Equalize based on color histograms"));
344                 break;
345             case GAMMA:
346                 message.append(tr("Gamma"));
347                 break;
348             case NEGATIVE:
349                 message.append(tr("Inverse"));
350                 break;
351             default:
352                 message.append(tr("unknown"));
353                 break;
354         }
355 
356         emit sendMessage(message, defaultTimeOut);
357     }
358 
359     /*
360      * Reset mean and std process time in order to re-start computing
```



```
361      * new mean and std process time values.
362      */
363     void QcvHistograms::resetMeanProcessTime()
364     {
365         CvHistograms8UC3::resetMeanProcessTime();
366     }
367 
368     /*
369      * Draws selected histogram(s) in drawing frame and returns the drawing
370      * frame
371      * @return the updated drawing frame.
372      */
373     void QcvHistograms::drawHistograms()
374     {
375         CvHistograms8UC3::drawHistograms();
376         emit histogramImageUpdated();
377     }
378 
379     /*
380      * Draws selected transfert function in drawing frame and returns the
381      * drawing frame
382      * @param lut the LUT to draw : the LUT may contains 1 or several
383      * channels
384      * @return the updated drawing frame
385      */
386     void QcvHistograms::drawTransfertFunc(const Mat *lut)
387     {
388         CvHistograms8UC3::drawTransfertFunc(lut);
389         emit lutImageUpdated();
390     }
391 
392     /*
393      * Apply current LUT (if != NULL) to the source image to produce the
394      * outFrame
395      * @return true if LUT has been applied, false if lut is NULL or
396      * lutType is NONE
397      */
398     bool QcvHistograms::drawTransformedImage()
399     {
400         bool result = CvHistograms8UC3::drawTransformedImage();
401         emit outImageUpdated();
402         return result;
403     }
```

```cpp
#include <cmath>
#include <opencv2/core/core.hpp>    // for MeanValue<cv::Mat, cv::Mat> specialization

#include "MeanValue.h"

/*
 * Constructor.
 * Initialize sum & sum2 to T(0) and count to 0
 * @param initialValue [optional] a T specimen can be provided in order
 * to initialise sum and sum2 by copying the specimen
 * @param initialMinimum [optional] initial value of minimum and minimum
 * reset value
 */
template <typename T, typename R>
MeanValue<T, R>::MeanValue(const T & initialValue,
                           const T & initialMinimum) :
    sum(initialValue),
    sum2(initialValue),
    count(0),
    minValue(initialMinimum),
    maxValue(initialValue),
    resetMinValue(initialMinimum),
    resetMaxValue(initialValue)
{
}

/*
 * Copy constructor
 * @param mv the other mean value to copy
 */
template <typename T, typename R>
MeanValue<T, R>::MeanValue(const MeanValue<T, R> & mv) :
    sum(mv.sum),
    sum2(mv.sum2),
    count(mv.count),
    minValue(mv.minValue),
    maxValue(mv.maxValue),
    resetMinValue(mv.resetMinValue),
    resetMaxValue(mv.resetMaxValue)
{
}

/*
 * Move constructor
 * @param mv the other mean value to copy
 */
template <typename T, typename R>
MeanValue<T, R>::MeanValue(MeanValue<T, R> && mv) :
    sum(mv.sum),
    sum2(mv.sum2),
    count(mv.count),
    minValue(mv.minValue),
    maxValue(mv.maxValue),
    resetMinValue(mv.resetMinValue),
    resetMaxValue(mv.resetMaxValue)
{
}

/*
 * Destructor
 */
template <typename T, typename R>
MeanValue<T, R>::~MeanValue()
{
}

/*
 * Function call operator
 * @param value value to add to the values sum and values square sum
 * @post elements count has been increased
 */
template <typename T, typename R>
void MeanValue<T, R>::operator ()(const T & value)
{
    sum += value;
    sum2 += value * value;
    count++;
    if (value > maxValue)
    {
        maxValue = value;
    }
    if (value < minValue)
    {
        minValue = value;
    }
}

/*
 * Self increment operator
 * @param value value to add to the values sum and values square sum
```



```cpp
 * @post elements count has been increased
 * @note does the same thing as Function call operator
 */
template <typename T, typename R>
void MeanValue<T, R>::operator +=(const T & value)
{
    operator()(value);
}

/*
 * Copy operator from another mean value
 * @param mv the mean value to copy
 * @return a reference to the current mean value
 */
template <typename T, typename R>
MeanValue<T, R> & MeanValue<T, R>::operator =(const MeanValue<T, R> & mv)
{
    sum = mv.sum;
    sum2 = mv.sum2;
    count = mv.count;
    minValue = mv.minValue;
    maxValue = mv.maxValue;
    // can't copy resetMinValue & resetMaxValue 'cause they're constants

    return *this;
}

/*
 * Move operator from another mean value
 * @param mv the mean value to move
 * @return a reference to the current mean value
 */
template <typename T, typename R>
MeanValue<T, R> & MeanValue<T, R>::operator =(MeanValue<T, R> && mv)
{
    sum = mv.sum;
    sum2 = mv.sum2;
    count = mv.count;
    minValue = mv.minValue;
    maxValue = mv.maxValue;
    // can't copy resetMinValue & resetMaxValue 'cause they're constants

    return *this;
}

/*
 * Cast operator to result type
 * @return the mean value
 */
template <typename T, typename R>
MeanValue<T, R>::operator R() const
{
    return mean();
}

/*
 * Compute mean value : E(X) = sum/nbElements
 * @return the mean value of all added elements.
 */
template <typename T, typename R>
R MeanValue<T, R>::mean() const
{
    if (count != 0)
    {
        return R(sum / (R) count);
    }
    else
    {
        return R(0);
    }
}

/*
 * Compute standard deviation of values : sqrt(E(X^2) - E(X)^2)
 * @return the standard deviation of all added elements.
 */
template <typename T, typename R>
R MeanValue<T, R>::std() const
{
    if (count != 0)
    {
        R ex = mean();
        double ex2 = sum2 / (double) count;
        return R(sqrt(ex2 - double(ex * ex)));
    }
    else
    {
        return R(0);
    }
}
```

```
181
182 /*
183  * Minimum recorded value accessor
184  * @return the minimum recorded value (until reset)
185  */
186 template <typename T, typename R>
187 T MeanValue<T, R>::min() const
188 {
189     if (count ≠ 0)
190     {
191         return minValue;
192     }
193     else
194     {
195         return T(0);
196     }
197 }
198
199 /*
200  * Maximum recorded value accessor
201  * @return the maximum recorded value (until reset)
202  */
203 template <typename T, typename R>
204 T MeanValue<T, R>::max() const
205 {
206     if (count ≠ 0)
207     {
208         return maxValue;
209     }
210     else
211     {
212         return T(0);
213     }
214 }
215
216 /*
217  * Reset added values, square values and count to 0
218  */
219 template <typename T, typename R>
220 void MeanValue<T, R>::reset()
221 {
222     sum = T(0);
223     sum2 = T(0);
224     count = 0;
225     minValue = resetMinValue;
226     maxValue = resetMaxValue;
227 }
228
229 /*
230  * Output operator for MeanValue
231  * @param out the output stream
232  * @param mv the MeanValue to print on the output stream
233  * @return a reference to the current output stream
234  * @post put mean value Â± std value on the stream
235  */
236 template <typename T, typename R>
237 ostream & operator <<(ostream & out, const MeanValue<T, R> & mv)
238 {
239     out << mv.mean() << " Â± " << mv.std() << " [" << mv.min() << "…"
240         << mv.max() << "]";
241
242     return out;
243 }
244
245 // -----------------------------------------------------------------------------
246 // Specializations for MeanValue<cv::Mat, cv::Mat>
247 // -----------------------------------------------------------------------------
248
249 /**
250  * Function call operator (specialization for MeanValue<cv::Mat, cv::Mat>)
251  * @param value value to add to the values sum and values square sum
252  * @post elements count has been increased
253  */
254 template <>
255 void MeanValue<cv::Mat>::operator ()(const cv::Mat & value)
256 {
257     sum += value;
258     sum2 += value * value.t();
259     count++;
260     int rows = value.rows;
261     int cols = value.cols;
262     for (int i = 0; i < rows; i++)
263     {
264         for (int j = 0; j < cols; j++)
265         {
266             /*
267              * FIXME Caution accessing pixels values in double only works
268              * with matrices of double
269              */
270             double & currentMin = minValue.at<double>(i, j);
```



```
271             double & currentMax = maxValue.at<double>(i, j);
272             double currentValue = value.at<double>(i, j);
273             if (currentValue < currentMin)
274             {
275                 currentMin = currentValue;
276             }
277             if (currentValue > currentMax)
278             {
279                 currentMax = currentValue;
280             }
281         }
282     }
283 }
284
285 /**
286  * Compute mean value (specialization for MenValue<cv::Mat, cv::Mat>):
287  * E(X) = sum/nbElements
288  * @return the mean value of all added elements.
289  */
290 template <>
291 cv::Mat MeanValue<cv::Mat>::mean() const
292 {
293     if (count ≠ 0)
294     {
295         return cv::Mat(sum * double(1.0/(double)count));
296     }
297     else
298     {
299         return cv::Mat(sum * double(0));
300     }
301 }
302
303 /**
304  * Compute standard deviation of values (specialization for
305  * MeanValue<cv::Mat; cv::Mat>): sqrt(E(X^2) - E(X)^2)
306  * @return the standard deviation of all added elements.
307  */
308 template <>
309 cv::Mat MeanValue<cv::Mat>::std() const
310 {
311     if (count ≠ 0)
312     {
313         cv::Mat ex = mean();
314         cv::Mat ex2 = sum2 * double(1.0 / (double) count);
315         int rows = sum.rows;
316         int cols = sum.cols;
317         cv::Mat result(rows, cols, CV_64FC1);
318
319         for (int i = 0; i < rows; i++)
320         {
321             for (int j = 0; j < cols; j++)
322             {
323                 double exij = ex.at<double>(i,j);
324                 result.at<double>(i,j) = sqrt( ex2.at<double>(i,j) - (exij * exij) );
325             }
326         }
327
328         return result;
329     }
330     else
331     {
332         return cv::Mat(sum2 * double(0.0));
333     }
334 }
335
336 /**
337  * Minimum recorded value accessor (specialization for
338  * MeanValue<cv::Mat; cv::Mat>)
339  * @return the minimum recorded value (until reset)
340  */
341 template <>
342 cv::Mat  MeanValue<cv::Mat>::min() const
343 {
344     if (count ≠ 0)
345     {
346         return minValue;
347     }
348     else
349     {
350         return cv::Mat();
351     }
352 }
353
354 /**
355  * Maximum recorded value accessor (specialization for
356  * MeanValue<cv::Mat; cv::Mat>)
357  * @return the maximum recorded value (until reset)
358  */
359 template <>
360 cv::Mat MeanValue<cv::Mat>::max() const
```

```
361 {
362     if (count ≠ 0)
363     {
364         return maxValue;
365     }
366     else
367     {
368         return cv::Mat();
369     }
370 }
371
372 /**
373  * Reset added values (specialization for MeanValue<cv::Mat, cv::Mat>),
374  * square values and count to 0
375  */
376 template <>
377 void MeanValue<cv::Mat>::reset()
378 {
379     sum *= double(0);
380     sum2 *= double(0);
381     count = 0;
382     minValue = resetMinValue;
383     maxValue = resetMaxValue;
384 }
385
386 // ------------------------------------------------------------------------
387 // Template protoinstanciations for
388 //  - int
389 //  - clock_t (unsigned long)
390 //  - float
391 //  - double
392 //  - cv::Mat
393 //  - Pose
394 // ------------------------------------------------------------------------
395
396 // Proto instanciations
397 template class MeanValue<int, double>;
398 template class MeanValue<clock_t, double>;
399 template class MeanValue<float, double>;
400 template class MeanValue<double>;
401 template class MeanValue<int, float>;
402 template class MeanValue<clock_t, float>;
403 template class MeanValue<float>;
404 template class MeanValue<double, float>;
405 template class MeanValue<cv::Mat>;
406
407 // Output operators proto-instanciations
408 template ostream & operator << (ostream &, const MeanValue<int, double> &);
409 template ostream & operator << (ostream &, const MeanValue<clock_t, double> &);
410 template ostream & operator << (ostream &, const MeanValue<float, double> &);
411 template ostream & operator << (ostream &, const MeanValue<double> &);
412 template ostream & operator << (ostream &, const MeanValue<int, float> &);
413 template ostream & operator << (ostream &, const MeanValue<clock_t, float> &);
414 template ostream & operator << (ostream &, const MeanValue<float> &);
415 template ostream & operator << (ostream &, const MeanValue<double, float> &);
416 template ostream & operator << (ostream &, const MeanValue<cv::Mat> &);
```



```
1  /*
2   * QcvMatWidget.cpp
3   *
4   *  Created on: 28 fÃ©vr. 2011
5   *      Author: davidroussel
6   */
7  #include <QtDebug>
8
9  #include <opencv2/imgproc.hpp>
10
11 #include "QcvMatWidget.h"
12
13 /*
14  * Default size when no image has been set
15  */
16 QSize QcvMatWidget::defaultSize(640, 480);
17
18 /*
19  * Default aspect ratio when image is not set yet
20  */
21 double QcvMatWidget::defaultAspectRatio = 4.0/3.0;
22
23 /*
24  * Drawing color
25  */
26 const Scalar QcvMatWidget::drawingColor(0xFF,0xCC,0x00,0x88);
27
28 /*
29  * Drawing width
30  */
31 const int QcvMatWidget::drawingWidth(3);
32
33 /*
34  * OpenCV QT Widget default constructor
35  * @param parent parent widget
36  * @param mouseSense mouse sensivity
37  */
38 QcvMatWidget::QcvMatWidget(QWidget *parent,
39                            MouseSense mouseSense) :
40     QWidget(parent),
41     sourceImage(NULL),
42     aspectRatio(defaultAspectRatio),
43     mousePressed(false),
44     mouseSense(mouseSense),
45 //  count(0)
46     pixelScale(devicePixelRatioF())
47 {
48     setup();
49 }
50
51 /*
52  * OpenCV QT Widget constructor
53  * @param the source image
54  * @param parent parent widget
55  * @param mouseSense mouse sensivity
56  */
57 QcvMatWidget::QcvMatWidget(Mat * sourceImage,
58                            QWidget *parent,
59                            MouseSense mouseSense) :
60     QWidget(parent),
61     sourceImage(sourceImage),
62     aspectRatio((double)sourceImage->cols / (double)sourceImage->rows),
63     mousePressed(false),
64     mouseSense(mouseSense),
65 //  count(0)
66     pixelScale(devicePixelRatioF())
67 {
68     setup();
69 }
70
71 /*
72  * OpenCV Widget destructor.
73  * Releases displayImage.
74  */
75 QcvMatWidget::~QcvMatWidget()
76 {
77     displayImage.release();
78 }
79
80 /*
81  * paint event reimplemented to draw content (in this case only
82  * draw in display image since final rendering method is not yet available)
83  * @param event the paint event
84  */
85 void QcvMatWidget::paintEvent(QPaintEvent * event)
86 {
87     Q_UNUSED(event);
88
89     if (displayImage.data ≠ NULL)
90     {
```

```
91          // evt draw in image
92          if (mousePressed)
93          {
94              // if MOUSE_CLICK only draws a cross
95              if (mouseSense > MOUSE_NONE)
96              {
97                  if (¬(mouseSense & MOUSE_DRAG))
98                  {
99                      if (mouseMoved)
100                     {
101                         drawCross(draggedPoint);
102                     }
103                     else
104                     {
105                         drawCross(pressedPoint);
106                     }
107                 }
108                 else  // else if MOUSE_DRAG starts drawing a rectangle
109                 {
110                     drawRectangle(selectionRect);
111                 }
112             }
113         }
114     }
115     else
116     {
117         qWarning("QcvMatWidget::paintEvent : image.data is NULL");
118     }
119 }
120
121 /*
122  * Widget setup
123  */
124 void QcvMatWidget::setup()
125 {
126     layout = new QHBoxLayout();
127     layout→setContentsMargins(0,0,0,0);
128     setLayout(layout);
129 }
130
131 /*
132  * Sets new source image
133  * @param sourceImage the new source image
134  */
135 void QcvMatWidget::setSourceImage(Mat * sourceImage)
136 {
137     // qDebug("QcvMatWidget::setSourceImage");
138
139     this→sourceImage = sourceImage;
140
141     // re-setup geometry since height x width may have changed
142     aspectRatio = (double)sourceImage→cols / (double)sourceImage→rows;
143     // qDebug ("aspect ratio changed to %4.2f", aspectRatio);
144
145 }
146
147 /*
148  * Converts BGR or Gray source image to RGB display image
149  * @see #sourceImage
150  * @see #displayImage
151  */
152 void QcvMatWidget::convertImage()
153 {
154 //  qDebug("Convert image");
155
156     int depth = sourceImage→depth();
157     int channels = sourceImage→channels();
158
159     // Converts any image type to RGB format
160     switch (depth)
161     {
162         case CV_8U:
163             switch (channels)
164             {
165                 case 1: // gray level image
166                     cvtColor(*sourceImage, displayImage,CV_GRAY2RGB);
167                     break;
168                 case 3: // Color image (OpenCV produces BGR images)
169                     cvtColor(*sourceImage, displayImage, CV_BGR2RGB);
170                     break;
171                 default:
172                     qFatal("This number of channels (%d) is not supported",
173                            channels);
174                     break;
175             }
176             break;
177         default:
178             qFatal("This image depth (%d) is not implemented in QcvMatWidget",
179                    depth);
180             break;
```



```
181     }
182 }
183
184 /*
185  * Callback called when mouse button pressed event occurs.
186  * reimplemented to send pressPoint signal when left mouse button is
187  * pressed
188  * @param event mouse event
189  */
190 void QcvMatWidget::mousePressEvent(QMouseEvent *event)
191 {
192     if (mouseSense > MOUSE_NONE)
193     {
194 //      qDebug("mousePressEvent(%d, %d) with button %d",
195 //             event->pos().x(), event->pos().y(), event->button());
196         mousePressed = true;
197         pressedPoint = event→pos();
198         pressedButton = event→button();
199
200         if((event→button() ≡ Qt::LeftButton) ∧ (mouseSense & MOUSE_DRAG))
201         {
202             // initialise selection rect
203             selectionRect.setTopLeft(pressedPoint);
204             selectionRect.setBottomRight(pressedPoint);
205         }
206
207         emit pressPoint(pressedPoint, pressedButton);
208     }
209 }
210
211 /*
212  * Callback called when mouse move event occurs.
213  * reimplemented to send dragPoint signal when mouse is dragged
214  * (after left mouse button has been pressed)
215  * @param event mouse event
216  */
217 void QcvMatWidget::mouseMoveEvent(QMouseEvent *event)
218 {
219     mouseMoved = true;
220     draggedPoint = event→pos();
221
222     if ((mouseSense & MOUSE_DRAG) ∧ mousePressed)
223     {
224 //      qDebug("mouseMoveEvent(%d, %d) with button %d",
225 //             event->pos().x(), event->pos().y(), event->button());
226
227         selectionRectFromPoints(pressedPoint, draggedPoint);
228
229         emit dragPoint(draggedPoint);
230     }
231 }
232
233 /*
234  * Callback called when mouse button released event occurs.
235  * reimplemented to send releasePoint signal when left mouse button is
236  * released
237  * @param event mouse event
238  */
239 void QcvMatWidget::mouseReleaseEvent(QMouseEvent *event)
240 {
241     if ((mouseSense > MOUSE_NONE) ∧ mousePressed)
242     {
243 //      qDebug("mouseReleaseEvent(%d, %d) with button %d",
244 //             event->pos().x(), event->pos().y(), event->button());
245         mousePressed = false;
246         mouseMoved = false;
247         releasedPoint = event→pos();
248         emit releasePoint(releasedPoint, pressedButton);
249
250         if ((event→button() ≡ Qt::LeftButton) ∧ (mouseSense & MOUSE_DRAG))
251         {
252             selectionRectFromPoints(pressedPoint, releasedPoint);
253             emit releaseSelection(selectionRect, event→button());
254         }
255     }
256 }
257
258 /*
259  * Draw Cross
260  * @param p the cross center
261  */
262 void QcvMatWidget::drawCross(const QPoint & p)
263 {
264     int x0 = p.x();
265     int y0 = p.y();
266     int x1, x2, x3, x4;
267     int y1, y2, y3, y4;
268     int offset = 10;
269
270     x1 = x0 - 2*offset;
```

```
271     x2 = x0 - offset;
272     x3 = x0 + offset;
273     x4 = x0 + 2*offset;
274     y1 = y0 - 2*offset;
275     y2 = y0 - offset;
276     y3 = y0 + offset;
277     y4 = y0 + 2*offset;
278
279     Point p1a(x1, y0);
280     Point p1b(x2, y0);
281     Point p2a(x3, y0);
282     Point p2b(x4, y0);
283     Point p3a(x0, y1);
284     Point p3b(x0, y2);
285     Point p4a(x0, y3);
286     Point p4b(x0, y4);
287
288     line(displayImage, p1a, p1b, drawingColor, drawingWidth, CV_AA);
289     line(displayImage, p2a, p2b, drawingColor, drawingWidth, CV_AA);
290     line(displayImage, p3a, p3b, drawingColor, drawingWidth, CV_AA);
291     line(displayImage, p4a, p4b, drawingColor, drawingWidth, CV_AA);
292 }
293
294 /*
295  * Draw rectangle
296  * @param r the rectangle to draw
297  */
298 void QcvMatWidget::drawRectangle(const QRect & r)
299 {
300     int x1 = r.left();
301     int x2 = r.right();
302     int y1 = r.top();
303     int y2 = r.bottom();
304
305     Point p1(x1, y1);
306     Point p2(x2, y2);
307
308     rectangle(displayImage, p1, p2, drawingColor, drawingWidth, CV_AA);
309 }
310
311 /*
312  * Modifiy selectionRect using two points
313  * @param p1 first point
314  * @param p2 second point
315  */
316 void QcvMatWidget::selectionRectFromPoints(const QPoint & p1, const QPoint & p2)
317 {
318     int left, right, top, bottom;
319     if (p1.x() < p2.x())
320     {
321         left = p1.x();
322         right = p2.x();
323     }
324     else
325     {
326         left = p2.x();
327         right = p1.x();
328     }
329
330     if (p1.y() < p2.y())
331     {
332         top = p1.y();
333         bottom = p2.y();
334     }
335     else
336     {
337         top = p2.y();
338         bottom = p1.y();
339     }
340
341     selectionRect.setLeft(left);
342     selectionRect.setRight(right);
343     selectionRect.setTop(top);
344     selectionRect.setBottom(bottom);
345 }
346
347
348
349 /*
350  * Widget minimum size is set to the contained image size
351  * @return le size of the image within
352  */
353 //QSize QcvMatWidget::minimumSize() const
354 //{
355 //  return sizeHint();
356 //}
357
358
359 /*
360  * Size hint (because size depends on sourceImage properties)
```



```
361  * @return size obtained from sourceImage
362  */
363 QSize QcvMatWidget::sizeHint() const
364 {
365     if (sourceImage != NULL)
366     {
367         return QSize(sourceImage->cols, sourceImage->rows);
368     }
369     else
370     {
371         return defaultSize;
372     }
373 }
374
375 /*
376  * Gets Mat widget mouse clickable status
377  * @return true if widget is sensitive to mouse click
378  */
379 bool QcvMatWidget::isMouseClickable() const
380 {
381     return (mouseSense & MOUSE_CLICK);
382 }
383
384 /*
385  * Gets Mat widget mouse dragable status
386  * @return true if widget is sensitive to mouse drag
387  */
388 bool QcvMatWidget::isMouseDragable() const
389 {
390     return (mouseSense & MOUSE_DRAG);
391 }
392
393 /*
394  * Update slot customized to include convertImage before actually
395  * updating
396  */
397 void QcvMatWidget::update()
398 {
399 //  count++;
400 //  qDebug() << "QcvMatWidget::update " << count;
401 //  std::cerr << "{o";
402     convertImage();
403     QWidget::update();
404 //  std::cerr << "}";
405 }
406
407 /*
408  * Recompute pixel scale according to screen pixel scale.
409  * Used with Hi DPI devices (such as retina screens)
410  * @post pixel scale have been updated according to
411  * devicePixelRatioF provided by the QPaintDevice super class
412  */
413 void QcvMatWidget::screenChanged()
414 {
415     pixelScale = devicePixelRatioF();
416 //  qDebug() << "Pixel scale updated to" << pixelScale;
417 }
418
419 // ----------------------------------------------------------------------------
420 // convertImage old algorithm
421 // ----------------------------------------------------------------------------
422 //  int cvIndex, cvLineStart;
423 //  // switch between bit depths
424 //  switch (displayImage.depth())
425 //  {
426 //      case CV_8U:
427 //          switch (displayImage.channels())
428 //          {
429 //              case 1: // Grav level images
430 //                  if ( (displayImage.cols != image.width()) ||
431 //                       (displayImage.rows != image.height()) )
432 //                  {
433 //                      QImage temp(displayImage.cols, displayImage.rows,
434 //                              QImage::Format_RGB32);
435 //                      image = temp;
436 //                  }
437 //                  cvIndex = 0;
438 //                  cvLineStart = 0;
439 //                  for (int y = 0; y < displayImage.rows; y++)
440 //                  {
441 //                      unsigned char red, green, blue;
442 //                      cvIndex = cvLineStart;
443 //                      for (int x = 0; x < displayImage.cols; x++)
444 //                      {
445 //                          // DO it
446 //                          red   = displayImage.data[cvIndex];
447 //                          green = displayImage.data[cvIndex];
448 //                          blue  = displayImage.data[cvIndex];
449 //
450 //                          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;
//  }
```



```
//#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.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);
```



```cpp

    // 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");
    }
}
```

```cpp
/*
 * 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)
{
}

/*
 * OpenCV QT Widget constructor
 * @param parent parent widget
 */
QcvMatWidgetGL::QcvMatWidgetGL(Mat * sourceImage,
                               QWidget *parent,
                               MouseSense mouseSense) :
    QcvMatWidget(sourceImage, parent, mouseSense),
    gl(NULL)
{
    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, GL_RGB, &pixelScale, 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);
    gl→update();
}
```



```cpp
/*
 * 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 Constructor
 * @param image the RGB image to draw in the pixel buffer
 * @param format pixel format
 * @param pixelScale pixel scale pointer from container
 * @param parent the parent widget
 */
QGLImageRender::QGLImageRender(const Mat & image,
                               const GLenum format,
                               float * pixelScale,
                               QWidget *parent) :
    QGLWidget(parent),
    image(image),
    pixelFormat(format),
    pixelScale(pixelScale)
{
    if (¬doubleBuffer())
    {
        qWarning("QGLImageRender::QGLImageRender caution : no double buffer");
    }
    if (this→image.data ≡ NULL)
    {
        qWarning("QGLImageRender::QGLImageRender caution : image data is null");
    }
    if (this→pixelScale ≡ NULL)
    {
        qCritical("QGLImageRender::QGLImageRender caution : pixel scale 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::resizeGL(int width, int height)
{
//  qDebug("GL resizeGL ...");

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

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

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

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

    glClear(GL_COLOR_BUFFER_BIT);

    if (image.data ≠ NULL)
    {
        /* apply the right translate so the image drawing starts top left */
        glRasterPos4f(0.0f,(GLfloat)(image.rows), 0.0f, 1.0f);
        /*
         * for hi dpi displays
         *  typically pixelScale =
         *  - 1.0 for normal displays
         *  - 2.0 for hidpi displays
```

```
91          */
92          glPixelZoom(*pixelScale, -(*pixelScale));
93
94          glDrawPixels(image.cols, image.rows, pixelFormat,
95                       GL_UNSIGNED_BYTE, image.data);
96          // In any circumstance you should NOT use glFlush or swapBuffers() here
97      }
98      else
99      {
100         qWarning("Nothing to draw");
101     }
102 }
103
104 QSize QGLImageRender::sizeHint () const
105 {
106     return minimumSizeHint();
107 }
108
109 QSize QGLImageRender::minimumSizeHint() const
110 {
111     if (image.data ≠ NULL)
112     {
113         return QSize(image.cols, image.rows);
114     }
115     else
116     {
117         qWarning("QGLImageRender::minimumSizeHint : probably invalid sizeHint");
118         return QSize(320,240);
119     }
120 }
121
122 QSizePolicy QGLImageRender::sizePolicy () const
123 {
124     return QSizePolicy(QSizePolicy::Fixed, QSizePolicy::Fixed);
125 }
```



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

```
91      skip(skip),
92      size(0, 0),
93      originalSize(0, 0),
94      frameRate(0.0),
95      statusMessage()
96  {
97      if (updateThread ≠ NULL)
98      {
99          moveToThread(this→updateThread);
100         connect(this, SIGNAL(finished()), updateThread, SLOT(quit()),
101                 Qt::DirectConnection);
102     }
103
104     timer→setSingleShot(false);
105     connect(timer, SIGNAL(timeout()), SLOT(update()));
106
107     if (grabTest())
108     {
109         setSize(width, height);
110         QString message("Camera ");
111         message.append(QString::number(deviceId));
112         message.append(" ");
113         int delay = grabInterval(message);
114         if (updateThread ≠ NULL)
115         {
116             updateThread→start();
117         }
118         timer→start(delay);
119         qDebug("timer started with %d ms delay", delay);
120         emit timerChanged(delay);
121     }
122     else
123     {
124         qDebug() << "QcvVideoCapture::QcvVideoCapture(" << deviceId
125                  << "): grab test failed";
126     }
127 }
128
129 /*
130  * QcvVideoCapture constructor from file name
131  * @param fileName video file to open
132  * @param flipVideo mirror image
133  * @param gray convert image to gray
134  * @param skip indicates capture can skip an image. When the capture
135  * result has not been processed yet, or when false that capture should
136  * wait for the result to be processed before grabbing a new image.
137  * This only applies when #updateThread is not NULL.
138  * @param width desired width or 0 to keep capture width
139  * @param height desired height or 0 to keep capture height
140  * @param updateThread the thread used to run this capture
141  * @param parent the parent QObject
142  */
143 QcvVideoCapture::QcvVideoCapture(const QString & fileName,
144                                  const bool flipVideo,
145                                  const bool gray,
146                                  const bool skip,
147                                  const unsigned int width,
148                                  const unsigned int height,
149                                  QThread * updateThread,
150                                  QObject * parent) :
151     QObject(parent),
152     filename(fileName),
153     capture(fileName.toStdString()),
154     timer(new QTimer(updateThread ≡ NULL ? this : NULL)),
155     updateThread(updateThread),
156     mutex(QMutex::NonRecursive),
157     lockLevel(0),
158     liveVideo(false),
159     flipVideo(flipVideo),
160     resize(false),
161     directResize(false),
162     gray(gray),
163     skip(skip),
164     size(0, 0),
165     originalSize(0, 0),
166     frameRate(0.0),
167     statusMessage()
168 {
169     if (updateThread ≠ NULL)
170     {
171         moveToThread(this→updateThread);
172         connect(this, SIGNAL(finished()), updateThread, SLOT(quit()),
173                 Qt::DirectConnection);
174     }
175
176     timer→setSingleShot(false);
177     connect(timer, SIGNAL(timeout()), SLOT(update()));
178
179     if (grabTest())
180     {
```



```
181         setSize(width, height);
182         QString message("File ");
183         message.append(fileName);
184         message.append(" ");
185
186         int delay = grabInterval(message);
187         if (updateThread ≠ NULL)
188         {
189             updateThread→start();
190         }
191         timer→start(delay);
192         qDebug("timer started with %d ms delay", delay);
193         emit timerChanged(delay);
194     }
195 }
196
197 /*
198  * QcvVideoCapture destructor.
199  * releases video capture and image
200  */
201 QcvVideoCapture::~QcvVideoCapture()
202 {
203     // wait for the end of an update
204     if (updateThread ≠ NULL)
205     {
206         if (lockLevel ≡ 0)
207         {
208             // qDebug() << "QcvVideoCapture::~QcvVideoCapture: lock in thread"
209             //          << QThread::currentThread();
210             mutex.lock();
211         }
212         lockLevel++;
213
214         emit finished();
215     }
216
217     if (timer ≠ NULL)
218     {
219         if (timer→isActive())
220         {
221             timer→stop();
222             qDebug("timer stopped");
223         }
224
225         timer→disconnect(SIGNAL(timeout()), this, SLOT(update()));
226     }
227
228     if (updateThread ≠ NULL)
229     {
230         lockLevel--;
231         if (lockLevel ≡ 0)
232         {
233             mutex.unlock();
234         }
235
236         // Wait until the updateThread receives the "finished" signal through
237         // "quit" slot
238         updateThread→wait();
239
240         delete timer; // delete unparented timer
241     }
242
243     // releasse OpenCV ressources
244     filename.clear();
245     capture.release();
246     imageDisplay.release();
247     imageFlipped.release();
248     imageResized.release();
249     image.release();
250
251 //  qDebug() << "QcvVideoCapture destroyed";
252 }
253
254 /*
255  * Open new device Id
256  * @param deviceId device number to open
257  * @param width desired width or 0 to keep capture width
258  * @param height desired height or 0 to keep capture height
259  * @return true if device has been opened and checked and timer launched
260  */
261 bool QcvVideoCapture::open(const int deviceId,
262                            const unsigned int width,
263                            const unsigned int height)
264 {
265     if (updateThread ≠ NULL)
266     {
267         if (lockLevel ≡ 0)
268         {
269             mutex.lock();
270         }
```

```
271         lockLevel++;
272     }
273
274     filename.clear();
275     if (timer→isActive())
276     {
277         timer→stop();
278         qDebug("timer stopped");
279     }
280
281     if (capture.isOpened())
282     {
283         capture.release();
284     }
285
286     if (¬image.empty())
287     {
288         image.release();
289     }
290
291     capture.open(deviceId);
292
293     bool grabbed = grabTest();
294
295     if (grabbed)
296     {
297         setSize(width, height);
298
299         statusMessage.clear();
300         statusMessage.append("Camera ");
301         statusMessage.append(QString::number(deviceId));
302         statusMessage.append(" ");
303         int delay = grabInterval(statusMessage);
304         timer→start(delay);
305         liveVideo = true;
306         qDebug("timer started with %d ms delay", delay);
307         emit timerChanged(delay);
308         emit imageChanged(&imageDisplay);
309     }
310     if (updateThread ≠ NULL)
311     {
312         lockLevel--;
313         if (lockLevel ≡ 0)
314         {
315             mutex.unlock();
316         }
317     }
318
319     return grabbed;
320 }
321
322 /*
323  * Open new video file
324  * @param fileName video file to open
325  * @param width desired width or 0 to keep capture width
326  * @param height desired height or 0 to keep capture height
327  * @return true if video has been opened and timer launched
328  */
329 bool QcvVideoCapture::open(const QString & fileName,
330                            const unsigned int width,
331                            const unsigned int height)
332 {
333     filename = fileName;
334
335     if (timer→isActive())
336     {
337         timer→stop();
338         qDebug("timer stopped");
339     }
340
341     if (updateThread ≠ NULL)
342     {
343         if (lockLevel ≡ 0)
344         {
345             mutex.lock();
346         }
347         lockLevel++;
348     }
349
350     if (capture.isOpened())
351     {
352         capture.release();
353     }
354
355     if (¬image.empty())
356     {
357         image.release();
358     }
359
360     capture.open(fileName.toStdString());
```



```
361
362     bool grabbed = grabTest();
363
364     if (grabbed)
365     {
366         setSize(width, height);
367         // qDebug() << "open setSize done";
368         statusMessage.clear();
369         statusMessage.append("file ");
370         statusMessage.append(fileName);
371         statusMessage.append(" opened");
372
373         int delay = grabInterval(statusMessage);
374         timer→start(delay);
375         liveVideo = false;
376         qDebug("timer started with %d ms delay", delay);
377         emit timerChanged(delay);
378         emit imageChanged(&imageDisplay);
379     }
380
381     if (updateThread ≠ NULL)
382     {
383         lockLevel--;
384         if(lockLevel ≡ 0)
385         {
386             mutex.unlock();
387         }
388     }
389
390     return grabbed;
391 }
392
393 /*
394  * Size accessor
395  * @return the image size
396  */
397 const QSize & QcvVideoCapture::getSize() const
398 {
399     return size;
400 }
401
402 /*
403  * Sets #imageDisplay size according to preferred width and height
404  * @param width desired width
405  * @param height desired height
406  * @pre a first image have been grabbed
407  */
408 void QcvVideoCapture::setSize(const unsigned int width,
409                               const unsigned int height)
410 {
411     if ((updateThread ≠ NULL))
412     {
413         if (lockLevel ≡ 0)
414         {
415             mutex.lock();
416         }
417         lockLevel++;
418     }
419
420     unsigned int preferredWidth;
421     unsigned int preferredHeight;
422
423     // if not empty then release it
424     if (¬imageResized.empty())
425     {
426         imageResized.release();
427     }
428
429     if ((width ≡ 0) ∧ (height ≡ 0)) // reset to original size
430     {
431         if (directResize) // direct set size to original size
432         {
433             setDirectSize((unsigned int)originalSize.width(),
434                           (unsigned int)originalSize.height());
435             // image is updated into setDirectSize
436         }
437         preferredWidth = image.cols;
438         preferredHeight = image.rows;
439
440         resize = false;
441         imageResized = image;
442     }
443     else // width != 0 or height != 0
444     {
445         if ((width ≡ (unsigned int)image.cols) ∧
446             (height ≡ (unsigned int)image.rows)) // unchanged
447         {
448             preferredWidth = image.cols;
449             preferredHeight = image.rows;
450             imageResized = image;
```

```
451
452             if (((int)preferredWidth ≡ originalSize.width()) ∧
453                 ((int)preferredHeight ≡ originalSize.height()))
454             {
455                 resize = false;
456             }
457             else
458             {
459                 resize = true;
460             }
461         }
462         else // width or height have changed
463         {
464             /*
465              * Resize needed
466              */
467             preferredWidth = width;
468             preferredHeight = height;
469
470             resize = true;
471
472             if (directResize)
473             {
474                 setDirectSize(preferredWidth, preferredHeight);
475                 imageResized = image;
476             }
477             else
478             {
479                 imageResized = Mat(preferredHeight, preferredWidth, image.type());
480             }
481         }
482     }
483
484     if (updateThread ≠ NULL)
485     {
486         lockLevel--;
487         if (lockLevel ≡ 0)
488         {
489             mutex.unlock();
490         }
491     }
492
493     qDebug("QcvVideoCapture resize is %s [%s]",
494            (resize ? "ON" : "OFF"),
495            (directResize ? "direct" : "soft"));
496
497     size.setWidth(preferredWidth);
498     size.setHeight(preferredHeight);
499     statusMessage.clear();
500     statusMessage.sprintf("Size set to %dx%d", preferredWidth, preferredHeight);
501     emit messageChanged(statusMessage, messageDelay);
502
503
504     /*
505      * imageChanged signal is delayed until setGray is called into
506      * setFlipVideo
507      */
508     // Refresh image chain
509     setFlipVideo(flipVideo);
510 }
511
512 /*
513  * Sets #imageDisplay size according to preferred width and height
514  * @param size new desired size to set
515  * @pre a first image have been grabbed
516  */
517 void QcvVideoCapture::setSize(const QSize & size)
518 {
519     setSize(size.width(), size.height());
520 }
521
522 /*
523  * Sets video flipping
524  * @param flipVideo flipped video or not
525  */
526 void QcvVideoCapture::setFlipVideo(const bool flipVideo)
527 {
528     bool previousFlip = this→flipVideo;
529     this→flipVideo = flipVideo;
530
531     if (updateThread ≠ NULL)
532     {
533         if (lockLevel ≡ 0)
534         {
535             mutex.lock();
536         }
537         lockLevel++;
538     }
539
540     if (¬imageFlipped.empty())
```



```
541     {
542         imageFlipped.release();
543     }
544
545     if (flipVideo)
546     {
547         imageFlipped = Mat(imageResized.size(), imageResized.type());
548     }
549     else
550     {
551         imageFlipped = imageResized;
552     }
553
554     if (updateThread ≠ NULL)
555     {
556         lockLevel--;
557         if (lockLevel ≡ 0)
558         {
559             mutex.unlock();
560         }
561     }
562
563     if (previousFlip ≠ flipVideo)
564     {
565         statusMessage.clear();
566         statusMessage.sprintf("flip video is %s", (flipVideo ? "on" : "off"));
567         emit messageChanged(statusMessage, messageDelay);
568         emit imageChanged(&imageDisplay);
569     }
570
571     /*
572      * imageChanged signal is delayed until setGray is called
573      */
574     // refresh image chain
575     setGray(gray);
576 }
577
578 /*
579  * Sets video conversion to gray
580  * @param grayConversion the gray conversion status
581  */
582 void QcvVideoCapture::setGray(const bool grayConversion)
583 {
584     bool previousGray = gray;
585
586     gray = grayConversion;
587
588     if (updateThread ≠ NULL)
589     {
590         if (lockLevel ≡ 0)
591         {
592             mutex.lock();
593         }
594         lockLevel++;
595     }
596
597     if (¬imageDisplay.empty())
598     {
599         imageDisplay.release();
600     }
601
602     if (gray)
603     {
604         imageDisplay = Mat(imageFlipped.size(), CV_8UC1);
605     }
606     else
607     {
608         imageDisplay = imageFlipped;
609     }
610
611     if (updateThread ≠ NULL)
612     {
613         lockLevel--;
614         if (lockLevel ≡ 0)
615         {
616             mutex.unlock();
617         }
618     }
619
620     if (previousGray ≠ grayConversion)
621     {
622         statusMessage.clear();
623         statusMessage.sprintf("gray video is %s", (gray ? "on" : "off"));
624         emit messageChanged(statusMessage, messageDelay);
625     }
626
627     /*
628      * In any cases emit image changed since
629      *  - setSize may have been called
630      *  - setFlipVideo may have been called
```

```
631         */
632     emit imageChanged(&imageDisplay);
633 }
634
635 /*
636  * Gets resize state.
637  * @return true if imageDisplay have been resized to preferred width and
638  * height, false otherwise
639  */
640 bool QcvVideoCapture::isResized() const
641 {
642     return resize;
643 }
644
645 /*
646  * Gets direct resize state.
647  * @return true if image can be resized directly into capture.
648  * @note direct resize capabilities are tested into #grabTest which is
649  * called in all constructors. So #isDirectResizeable should not be
650  * called before #grabTest
651  */
652 bool QcvVideoCapture::isDirectResizeable() const
653 {
654     return directResize;
655 }
656
657 /*
658  * Gets video flipping status
659  * @return flipped video status
660  */
661 bool QcvVideoCapture::isFlipVideo() const
662 {
663     return flipVideo;
664 }
665
666 /*
667  * Gets video gray converted status
668  * @return the converted to gray status
669  */
670 bool QcvVideoCapture::isGray() const
671 {
672     return gray;
673 }
674
675 /*
676  * Gets the image skipping policy
677  * @return true if new image can be skipped when previous one has not
678  * been processed yet, false otherwise.
679  */
680 bool QcvVideoCapture::isSkippable() const
681 {
682     return skip;
683 }
684
685 /*
686  * Gets the current frame rate
687  * @return the current frame rate
688  */
689 double QcvVideoCapture::getFrameRate() const
690 {
691     return frameRate;
692 }
693
694 /*
695  * Image accessor
696  * @return the image
697  */
698 Mat * QcvVideoCapture::getImage()
699 {
700     return &imageDisplay;
701 }
702
703 /*
704  * The source image mutex
705  * @return  the mutex used on image access
706  */
707 QMutex * QcvVideoCapture::getMutex()
708 {
709     return &mutex;
710 }
711
712 /*
713  * Performs a grab test to fill #image
714  * @return true if capture is opened and successfully grabs a first
715  * frame into #image, false otherwise
716  */
717 bool QcvVideoCapture::grabTest()
718 {
719 //  qDebug("Grab test");
720     bool result = false;
```



```
721
722     if (capture.isOpened())
723     {
724 #ifndef Q_OS_LINUX // V4L does not support these queries
725         int capWidth = capture.get(CV_CAP_PROP_FRAME_WIDTH);
726         int capHeight = capture.get(CV_CAP_PROP_FRAME_HEIGHT);
727
728         qDebug("Capture grab test with %d x %d image", capWidth, capHeight);
729 #endif
730         // grabs first frame
731         if (capture.grab())
732         {
733             bool retrieved = capture.retrieve(image);
734             if (retrieved)
735             {
736                 size.setWidth(image.cols);
737                 size.setHeight(image.rows);
738                 originalSize.setWidth(image.cols);
739                 originalSize.setHeight(image.rows);
740
741                 /*
742                  * Tries to determine if direct resizing in capture is possible
743                  * by setting original size through properties
744                  * Typically :
745                  *  - camera capture might be resizable
746                  *  - video file capture may not be resizable
747                  */
748                 directResize = setDirectSize(image.cols, image.rows);
749
750                 qDebug("Capture direct resizing is %s",
751                        (directResize ? "on" : "off"));
752
753                 result = true;
754             }
755             else
756             {
757                 qFatal("Video Capture unable to retreive image");
758             }
759         }
760         else
761         {
762             qFatal("Video Capture can not grab");
763         }
764     }
765     else
766     {
767         qFatal("Video Capture is not opened");
768     }
769
770     return result;
771 }
772
773 /*
774  * Get or compute interval between two frames
775  * @return interval between two frames
776  * @pre capture is already instanciated
777  */
778 int QcvVideoCapture::grabInterval(const QString & message)
779 {
780     int frameDelay = defaultFrameDelay;
781
782     // Tries to get framerate from capture
783     // ------------------------------------------------------------------
784     // Caution : on some systems getting video parameters is forbidden !
785     // For instance it does not work with linuxes equipped with V4L
786     // ------------------------------------------------------------------
787 #ifndef Q_OS_LINUX
788     frameRate = capture.get(CV_CAP_PROP_FPS);
789 #else
790     frameRate = -1.0;
791 #endif
792
793     /*
794      * if capture obtained frameRate is inconsistent, then we'll try to find out
795      * by ourselves
796      */
797     if (frameRate <= 0.0)
798     {
799         /*
800          * If live Video : grab a few images and measure elapsed time
801          */
802         if (liveVideo)
803         {
804             QElapsedTimer localTimer;
805             localTimer.start();
806
807             for (size_t i=0; i < defaultFrameNumberTest; i++)
808             {
809                 capture >> image;
810             }
```

```
811
812             frameDelay = (int)(localTimer.elapsed() / defaultFrameNumberTest);
813             frameRate = 1.0/((double)frameDelay/1000.0);
814             qDebug("Measured capture frame rate is  %4.2f images/s", frameRate);
815         }
816         /*
817          * FIXME else ???
818          * video files read through capture should provide framerate with
819          * capture.get(CV_CAP_PROP_FPS) but what happens if they don't ???
820          */
821     }
822     else
823     {
824         qDebug("%s Capture frame rate = %4.2f", message.toStdString().c_str(),
825                                                 frameRate);
826         frameDelay = 1000/frameRate;
827     }
828
829     statusMessage.sprintf("%s frame rate = %4.2f images/s",
830                           message.toStdString().c_str(), frameRate);
831     emit messageChanged(statusMessage, messageDelay);
832
833     return frameDelay;
834 }
835
836 /*
837  * Tries to set capture size directly on capture by using properties.
838  *  - CV_CAP_PROP_FRAME_WIDTH to set frame width
839  *  - CV_CAP_PROP_FRAME_HEIGHT to set frame height
840  * @param width the width property to set on capture
841  * @param height the height property to set on capture
842  * @return true if capture is opened and if width and height have been
843  * set successfully through @code capture.set(...) @endcode. Returns
844  * false otherwise.
845  * @post if at least width or height have been set successfully, capture
846  * image is released then updated again so it will have the right
847  * dimensions.
848  */
849 bool QcvVideoCapture::setDirectSize(const unsigned int width,
850                                     const unsigned int height)
851 {
852 #ifdef Q_OS_LINUX
853     Q_UNUSED(width);
854     Q_UNUSED(height);
855 #endif
856     bool done = false;
857
858     /*
859      * We absolutely need this lock in order to safely set width and
860      * height directly into the capture, so if mutex is already locked
861      * we should wait for it to be unlocked before continuing. Moreover,
862      * if mutex is NON-recursive and already locked, the call to lock() could
863      * lead to a DEADlock, so mutex HAS to be recursive !
864      */
865
866 #ifndef Q_OS_LINUX
867     if (capture.isOpened())
868     {
869         bool setWidth = capture.set(CV_CAP_PROP_FRAME_WIDTH, (double)width);
870         bool setHeight = capture.set(CV_CAP_PROP_FRAME_HEIGHT, (double)height);
871         if (setWidth ∨ setHeight)
872         {
873             // release old capture image
874             image.release();
875
876             // force image update to get the right size
877             capture >> image;
878
879             done = true;
880         }
881     }
882 #endif
883
884     return done;
885 }
886
887 /*
888  * update slot triggered by timer : Grabs a new image and sends updated()
889  * signal iff new image has been grabbed, otherwise there is no more
890  * images to grab so kills timer
891  */
892 void QcvVideoCapture::update()
893 {
894     bool locked = true;
895     bool image_updated = false;
896
897     if (updateThread ≠ NULL)
898     {
899         if (skip)
900         {
```



```
901             locked = mutex.tryLock();
902             if (locked)
903             {
904                 lockLevel++;
905             }
906         }
907         else
908         {
909             if (lockLevel ≡ 0)
910             {
911                 mutex.lock();
912             }
913             lockLevel++;
914         }
915     }
916
917     if (capture.isOpened() ∧ locked)
918     {
919         capture >> image;
920
921         if (¬image.data) // captured image has no data
922         {
923             statusMessage.clear();
924
925             if (liveVideo)
926             {
927                 if (timer→isActive())
928                 {
929                     timer→stop();
930                     qDebug("timer stopped");
931                 }
932
933                 capture.release();
934
935                 statusMessage.sprintf("No more frames to capture ...");
936                 emit messageChanged(statusMessage, 0);
937                 qDebug("%s", statusMessage.toStdString().c_str());
938             }
939             else // not live video ==> video file
940             {
941                 // We'll try to rewind the file back to frame 0
942                 bool restart = capture.set(CV_CAP_PROP_POS_FRAMES, 0.0);
943
944                 if (restart)
945                 {
946                     statusMessage.sprintf("Capture restarted");
947                     emit messageChanged(statusMessage,
948                                         QcvVideoCapture::messageDelay);
949                     emit restarted();
950                     qDebug("%s", statusMessage.toStdString().c_str());
951
952                     // Refresh image chain resized -> flipped -> gray
953                     setSize(size);
954                 }
955                 else
956                 {
957                     capture.release();
958
959                     statusMessage.sprintf("Failed to restart capture ...");
960                     emit messageChanged(statusMessage, 0);
961                     emit finished();
962                     qDebug("%s", statusMessage.toStdString().c_str());
963                 }
964             }
965         }
966         else // capture image has data
967         {
968             /*
969              * CAUTION
970              * image->imageResized->imageFlipped->imageDisplay
971              * constitute an image chain, so when size is changed with
972              * setSize it should call setFlipVideo which should call
973              * setGray
974              */
975
976             // resize image
977             if (resize ∧ ¬directResize)
978             {
979                 cv::resize(image, imageResized, imageResized.size(), 0, 0,
980                     INTER_AREA);
981             }
982             /*
983              * else imageResized.data is already == image.data
984              */
985
986             // flip image horizontally if required
987             if (flipVideo)
988             {
989                 flip(imageResized, imageFlipped, 1);
990             }
```

```
991                 /*
992                  * else imageFlipped.data is already == imageResized.data
993                  */
994
995                 // convert image to gray if required
996                 if (gray)
997                 {
998                     cvtColor(imageFlipped, imageDisplay, CV_BGR2GRAY);
999                 }
1000                /*
1001                 * else imageDisplay.data is already == imageFlipped.data
1002                 */
1003                image_updated = true;
1004            }
1005
1006            if (updateThread ≠ NULL)
1007            {
1008                lockLevel--;
1009                if (lockLevel ≡ 0)
1010                {
1011                    mutex.unlock();
1012                }
1013            }
1014
1015            if (image_updated)
1016            {
1017                emit updated();
1018            }
1019        }
1020        else
1021        {
1022            // mutex hasn't been locked, so we skipped one capture
1023            // qDebug() << "Capture skipped an image (level " << lockLevel << ")";
1024        }
1025    }
```



```
1   /*
2    * CaptureFactory.cpp
3    *
4    *  Created on: 11 fÃ©vr. 2012
5    *      Author: davidroussel
6    */
7
8   #include <cstdlib>  // for NULL
9   #include <QDebug>
10  #include <QFile>
11  #include <QtGlobal>
12  #include <QStringListIterator>
13  #include "CaptureFactory.h"
14
15  /*
16   * Capture Factory constructor.
17   * Arguments can be
18   *  - [-d | --device] <device number> : camera number
19   *  - [-f | --file] <filename> : video file name
20   *  - [-m | --mirror] : flip image horizontally
21   *  - [-g | --gray] : convert to gray level
22   *  - [-s | --size] <width>x<height>: preferred width and height
23   * @param argList program the argument list provided as a list of
24   * strings
25   */
26  CaptureFactory::CaptureFactory(const QStringList & argList) :
27      capture(NULL),
28      deviceNumber(0),
29      liveVideo(true),
30      flippedVideo(false),
31      grayVideo(false),
32      skipImages(false),
33      preferredWidth(0),
34      preferredHeight(0),
35      videoPath()
36  {
37      // C++ Like iterator
38      // for (QStringList::const_iterator it = argList.begin(); it != argList.end(); ++it)
39      // Java like iterator (because we use hasNext multiple times)
40      for (QListIterator<QString> it(argList); it.hasNext(); )
41      {
42          QString currentArg(it.next());
43
44          if (currentArg ≡ "-d" ∨ currentArg ≡"--device")
45          {
46              // Next argument should be device number integer
47              if (it.hasNext())
48              {
49                  QString deviceString(it.next());
50                  bool convertOk;
51                  deviceNumber = deviceString.toInt(&convertOk,10);
52                  if (¬convertOk ∨ deviceNumber < 0)
53                  {
54                      qWarning("Warning: Invalid device number %d",deviceNumber);
55                      deviceNumber = 0;
56                  }
57                  liveVideo = true;
58              }
59              else
60              {
61                  qWarning("Warning: device tag found with no following device number");
62              }
63          }
64          else if (currentArg ≡ "-v" ∨ currentArg ≡ "--video")
65          {
66              // Next argument should be a path name to video file or URL
67              if (it.hasNext())
68              {
69                  videoPath = it.next();
70                  liveVideo = false;
71              }
72              else
73              {
74                  qWarning("file tag found with no following filename");
75              }
76          }
77          else if (currentArg ≡ "-m" ∨ currentArg ≡ "--mirror")
78          {
79              flippedVideo = true;
80          }
81          else if (currentArg ≡ "-g" ∨ currentArg ≡ "--gray")
82          {
83              grayVideo = true;
84          }
85          else if (currentArg ≡ "-k" ∨ currentArg ≡ "--skip")
86          {
87              skipImages = true;
88          }
89          else if (currentArg ≡ "-s" ∨ currentArg ≡ "--size")
90          {
```

```
91              if (it.hasNext())
92              {
93                  // search for <width>x<height>
94                  QString sizeString = it.next();
95                  int xIndex = sizeString.indexOf(QChar('x'), 0,
96                          Qt::CaseInsensitive);
97                  if (xIndex ≠ -1)
98                  {
99                      QString widthString = sizeString.left(xIndex);
100                     preferredWidth = widthString.toUInt();
101                     qDebug("preferred width is %d", preferredWidth);
102
103                     QString heightString = sizeString.remove(0, xIndex+1);
104                     preferredHeight = heightString.toUInt();
105                     qDebug("preferred height is %d", preferredHeight);
106                 }
107                 else
108                 {
109                     qWarning("invalid <width>x<height>");
110                 }
111             }
112             else
113             {
114                 qWarning("size not found after --size");
115             }
116         }
117     }
118 }
119
120 /*
121  * Capture factory destructor
122  */
123 CaptureFactory::~CaptureFactory()
124 {
125 }
126
127 /*
128  * Set the capture to live (webcam) or file source
129  * @param live the video source
130  */
131 void CaptureFactory::setLiveVideo(const bool live)
132 {
133     liveVideo = live;
134 }
135
136 /*
137  * Set device number to use when instanciating the capture with
138  * live video.
139  * @param deviceNumber the device number to use
140  */
141 void CaptureFactory::setDeviceNumber(const int deviceNumber)
142 {
143     if (deviceNumber ≥ 0)
144     {
145         this→deviceNumber = deviceNumber;
146     }
147     else
148     {
149         qWarning("CaptureFactory::setDeviceNumber: invalid number %d", deviceNumber);
150     }
151 }
152
153 /*
154  * Set path to video file when #liveVideo is false
155  * @param path the path to the video file source
156  */
157 void CaptureFactory::setFile(const QString & path)
158 {
159     if (QFile::exists(path))
160     {
161         videoPath = path;
162     }
163     else
164     {
165         qWarning() << QObject::tr("CaptureFactory::setFile: path") << path
166                    << QObject::tr(" does not exist");
167     }
168 }
169
170 /*
171  * Set video horizontal flip state (useful for selfies)
172  * @param flipped the horizontal flip state
173  */
174 void CaptureFactory::setFlipped(const bool flipped)
175 {
176     flippedVideo = flipped;
177 }
178
179 /*
180  * Set gray conversion
```



```
181  * @param gray the gray conversion state
182  */
183 void CaptureFactory::setGray(const bool gray)
184 {
185     grayVideo = gray;
186 }
187
188 /*
189  * Set video grabbing skippable. When true, grabbing is skipped when
190  * previously grabbed image has not been processed yet. Otherwise,
191  * grabbing new image wait for the previous image to be processed.
192  * This only applies if capture is run in a separate thread.
193  * @param skip the video grabbing skippable state
194  */
195 void CaptureFactory::setSkippable(const bool skip)
196 {
197     skipImages = skip;
198 }
199
200 /*
201  * Set video size (independently of video source actual size)
202  * @param width the desired image width
203  * @param height the desired image height
204  */
205 void CaptureFactory::setSize(const size_t width, const size_t height)
206 {
207     preferredWidth = (int)width;
208     preferredHeight = (int)height;
209 }
210
211 /*
212  * Set video size (independently of video source actual size)
213  * @param size the desired video size
214  */
215 void CaptureFactory::setSize(const QSize & size)
216 {
217     preferredWidth = size.width();
218     preferredHeight = size.height();
219 }
220
221 /*
222  * Provide capture instanciated according to values
223  * extracted from argument lists
224  * @param updateThread the thread to run this capture or NULL if this
225  * capture run in the current thread
226  * @return the new capture instance
227  */
228 QcvVideoCapture * CaptureFactory::getCaptureInstance(QThread * updateThread)
229 {
230     // ------------------------------------------------------------------------
231     // Opening Video Capture
232     // ------------------------------------------------------------------------
233     if (liveVideo)
234     {
235         qDebug() << "opening device # " << deviceNumber;
236     }
237     else
238     {
239         qDebug() << "opening video file " << videoPath;
240     }
241
242     qDebug() << "Opening ";
243     if (liveVideo)
244     {
245         // Live video feed
246         qDebug() << "Live Video ... from camera # " << deviceNumber;
247         capture = new QcvVideoCapture(deviceNumber,
248                                       flippedVideo,
249                                       grayVideo,
250                                       skipImages,
251                                       preferredWidth,
252                                       preferredHeight,
253                                       updateThread);
254     }
255     else
256     {
257         // Video file or stream
258         qDebug() << videoPath << " ... ";
259         capture = new QcvVideoCapture(videoPath,
260                                       flippedVideo,
261                                       grayVideo,
262                                       skipImages,
263                                       preferredWidth,
264                                       preferredHeight,
265                                       updateThread);
266     }
267
268     return capture;
269 }
```

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

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

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

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

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

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

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

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

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

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

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

    private:
        /**
         * The UI built in QtDesigner or QtCreator
```



```
         */
        Ui::MainWindow *ui;

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

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

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

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

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

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

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

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

    private slots:

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

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

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

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

        /**
```

```
181         * Menu action when original image size is selected.
182         * Sets capture not to resize image
183         */
184        void on_actionOriginalSize_triggered();
185
186        /**
187         * Menu action when constrained image size is selected.
188         * Sets capture resize to preferred width and height
189         */
190        void on_actionConstrainedSize_triggered();
191
192        /**
193         * Menu action to replace current image rendering widget by a
194         * QcvMatWidgetImage instance.
195         */
196        void on_actionRenderImage_triggered();
197
198        /**
199         * Menu action to replace current image rendering widget by a
200         * QcvMatWidgetLabel with pixmap instance.
201         */
202        void on_actionRenderPixmap_triggered();
203
204        /**
205         * Menu action to replace current image rendering widget by a
206         * QcvMatWidgetGL instance.
207         */
208        void on_actionRenderOpenGL_triggered();
209
210
211        /**
212         * Original size radioButton action.
213         * Sets capture resize to off
214         */
215        void on_radioButtonOrigSize_clicked();
216
217        /**
218         * Custom size radioButton action.
219         * Sets capture resize to preferred width and height
220         */
221        void on_radioButtonCustomSize_clicked();
222
223        /**
224         * Width spinbox value change.
225         * Changes the preferred width and if custom size is selected apply
226         * this custom width
227         * @param value the desired width
228         */
229        void on_spinBoxWidth_valueChanged(int value);
230
231        /**
232         * Height spinbox value change.
233         * Changes the preferred height and if custom size is selected apply
234         * this custom height
235         * @param value the desired height
236         */
237        void on_spinBoxHeight_valueChanged(int value);
238
239        /**
240         * Flip capture image horizontally.
241         * changes capture flip status
242         */
243        void on_checkBoxFlip_clicked();
244
245        /**
246         * Set transfert function to identity
247         */
248        void on_radioButtonIdentity_clicked();
249
250        /**
251         * Set transfert function to inverse
252         */
253        void on_radioButtonInverse_clicked();
254
255        /**
256         * Set transfert function to gamma
257         */
258        void on_radioButtonGamma_clicked();
259
260        /**
261         * Set transfert function to threshold
262         */
263        void on_radioButtonThreshold_clicked();
264
265        /**
266         * Set transfert function to optimal dynamic
267         */
268        void on_radioButtonDynamic_clicked();
269
270        /**
```



```
271         * Set transfert function to equalization
272         */
273        void on_radioButtonEqualize_clicked();
274
275        /**
276         * Set transfert function depending on processor to use colors
277         * components of the histogram generating 1 transfert function per image
278         * channels
279         */
280        void on_radioButtonChColor_clicked();
281
282        /**
283         * Set transfert function depending on processor to use gray level
284         * histogram component generating 1 transfert function per image
285         * channels
286         */
287        void on_radioButtonChGray_clicked();
288
289        /**
290         * Modify lut parameter applied to transfert function depending on
291         * histogram
292         * @param value the new value of lutParam
293         */
294        void on_spinBoxlutParam_valueChanged(int value);
295
296        /**
297         * Set histogram mode to normal
298         */
299        void on_radioButtonHMNormal_clicked();
300
301        /**
302         * Set Histogram mode to cumulative
303         */
304        void on_radioButtonHMCumulative_clicked();
305
306        /**
307         * set Histogram mode to time cumulative
308         */
309        void on_radioButtonHMTime_clicked();
310
311        /**
312         * Show/Hides histogram red component
313         */
314        void on_checkBoxHistRed_clicked();
315
316        /**
317         * Show/Hides histogram green component
318         */
319        void on_checkBoxHistGreen_clicked();
320
321        /**
322         * Show/Hides histogram Blue component
323         */
324        void on_checkBoxHistBlue_clicked();
325
326        /**
327         * Show/Hides histogram gray component
328         */
329        void on_checkBoxHistGray_clicked();
330 };
331
332 #endif // MAINWINDOW_H
```

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

#include <QObject>
#include <QFileDialog>
#include <QWindow>
#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 parent parent widget
 */
MainWindow::MainWindow(QcvVideoCapture * capture,
                       QcvHistograms * processor,
                       QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow),
    capture(capture),
    processor(processor),
    preferredWidth(320),
    preferredHeight(240)
{
    ui->setupUi(this);
    ui->scrollArea->setBackgroundRole(QPalette::Mid);

    // ------------------------------------------------------------------------
    // Assertions
    // ------------------------------------------------------------------------
    assert(capture != NULL);

    assert(processor != NULL);

    // ------------------------------------------------------------------------
    // Special widgets initialisation
    // ------------------------------------------------------------------------
    ui->widgetImage->setSourceImage(processor->getImagePtr("out"));
    ui->widgetHistogram->setSourceImage(processor->getImagePtr("histogram"));
    ui->widgetLUT->setSourceImage(processor->getImagePtr("lut"));

    // Replace widgetImage QcvMatWidget instance with QcvMatWidgetImage
    // Sets Source image for widgetImage
    // Connects processor->updated to widgetImage->update
    // Connects processor->outImageChanged to widgetImage->setSourceImage
    setupImageWidget(RENDER_IMAGE);

    // ------------------------------------------------------------------------
    // Signal/Slot connections
    // ------------------------------------------------------------------------

    // Histogram updates to various image widget updates
    connect(processor, SIGNAL(histogramImageUpdated()),
            ui->widgetHistogram, SLOT(update()));

    connect(processor, SIGNAL(lutImageUpdated()),
            ui->widgetLUT, SLOT(update()));

    // Histogram source image changed to various image widget set sources

    connect(processor, SIGNAL(histogramImageChanged(Mat*)),
            ui->widgetHistogram, SLOT(setSourceImage(Mat*)));

    connect(processor, SIGNAL(lutImageChanged(Mat*)),
            ui->widgetLUT, SLOT(setSourceImage(Mat*)));

    // Capture, histogram 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)));

    // Connect UI signals to Capture slots
    connect(this, SIGNAL(sizeChanged(const QSize &)),
            capture, SLOT(setSize(const QSize &))); //, Qt::DirectConnection);
    connect(this, SIGNAL(openDevice(int,uint,uint)),
            capture, SLOT(open(int,uint,uint))); //, Qt::DirectConnection);
    connect(this, SIGNAL(openFile(QString,uint,uint)),
            capture, SLOT(open(QString,uint,uint))); //, Qt::DirectConnection);
    connect(this, SIGNAL(flipVideo(bool)),
            capture, SLOT(setFlipVideo(bool))); //, Qt::DirectConnection);
```



```cpp

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

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

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

    setupUIfromProcessor();
}

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

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

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

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

    emit openDevice(0, width, height);
}

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

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

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

    emit openDevice(1, width, height);
```

```
181 }
182
183 /*
184  * Menu action when Sources->file is selected.
185  * Opens file dialog and tries to open selected file (is not empty),
186  * then sets capture to open the selected file
187  */
188 void MainWindow::on_actionFile_triggered()
189 {
190     int width = 0;
191     int height = 0;
192
193     if (ui->radioButtonCustomSize->isChecked())
194     {
195         width = preferredWidth;
196         height = preferredHeight;
197     }
198
199     QString fileName =
200     QFileDialog::getOpenFileName(this,
201                                  tr("Open Video"),
202                                  "./",
203                                  tr("Video Files (*.avi *.mkv *.mp4 *.m4v)"),
204                                  NULL,
205                                  QFileDialog::ReadOnly);
206
207     qDebug("Opening file %s ...", fileName.toStdString().c_str());
208
209     if (fileName.length() > 0)
210     {
211 //      if (!capture->open(fileName, width, height))
212 //      {
213 //          qWarning("Unable to open device file : %s",
214 //                   fileName.toStdString().c_str());
215 //      }
216         emit openFile(fileName, width, height);
217     }
218     else
219     {
220         qWarning("empty file name");
221     }
222 }
223
224 /*
225  * Menu action to qui application
226  */
227 void MainWindow::on_actionQuit_triggered()
228 {
229     this->close();
230 }
231
232 /*
233  * Menu action when flip image is selected.
234  * Sets capture to change flip status which leads to reverse
235  * image horizontally
236  */
237 void MainWindow::on_actionFlip_triggered()
238 {
239     // capture->setFlipVideo(!capture->isFlipVideo());
240     emit flipVideo(¬capture->isFlipVideo());
241     /*
242      * There is no need to update ui->checkBoxFlip since it is connected
243      * to ui->actionFlip through signals/slots
244      */
245 }
246
247 /*
248  * Menu action when original image size is selected.
249  * Sets capture not to resize image
250  */
251 void MainWindow::on_actionOriginalSize_triggered()
252 {
253     ui->actionConstrainedSize->setChecked(false);
254
255     emit sizeChanged(QSize(0, 0));
256 }
257
258 /*
259  * Menu action when constrained image size is selected.
260  * Sets capture resize to preferred width and height
261  */
262 void MainWindow::on_actionConstrainedSize_triggered()
263 {
264     ui->actionOriginalSize->setChecked(false);
265
266     emit sizeChanged(QSize(preferredWidth, preferredHeight));
267 }
268
269 /*
270  * Changes widgetImage nature according to desired rendering mode.
```



```
271  * Possible values for mode are:
272  *  - IMAGE: widgetImage is assigned to a QcvMatWidgetImage instance
273  *  - PIXMAP: widgetImage is assigned to a QcvMatWidgetLabel instance
274  *  - GL: widgetImage is assigned to a QcvMatWidgetGL instance
275  * @param mode
276  */
277 void MainWindow::setupImageWidget(const RenderMode mode)
278 {
279     // Disconnect first
280     disconnect(processor, SIGNAL(outImageUpdated()),
281                ui->widgetImage, SLOT(update()));
282
283     disconnect(processor, SIGNAL(outImageChanged(Mat*)),
284                ui->widgetImage, SLOT(setSourceImage(Mat*)));
285
286     QWindow * currentWindow = windowHandle();
287     if (mode == RENDER_GL)
288     {
289         disconnect(currentWindow,
290                    SIGNAL(screenChanged(QScreen *)),
291                    ui->widgetImage,
292                    SLOT(screenChanged()));
293         disconnect(currentWindow,
294                    SIGNAL(screenChanged(QScreen*)),
295                    ui->widgetHistogram,
296                    SLOT(screenChanged()));
297         disconnect(currentWindow,
298                    SIGNAL(screenChanged(QScreen*)),
299                    ui->widgetLUT,
300                    SLOT(screenChanged()));
301     }
302
303     // remove widget in scroll area
304     QWidget * w = ui->scrollArea->takeWidget();
305
306     if (w == ui->widgetImage)
307     {
308         // delete removed widget
309         delete ui->widgetImage;
310
311         // create new widget
312         Mat * image = processor->getImagePtr("out");
313         if (image == NULL)
314         {
315             qFatal("Null image out");
316         }
317         if (image->data == NULL)
318         {
319             qFatal("image out NULL data");
320         }
321         switch (mode)
322         {
323             case RENDER_PIXMAP:
324                 ui->widgetImage = new QcvMatWidgetLabel(image);
325                 break;
326             case RENDER_GL:
327                 ui->widgetImage = new QcvMatWidgetGL(image);
328                 break;
329             case RENDER_IMAGE:
330             default:
331                 ui->widgetImage = new QcvMatWidgetImage(image);
332                 break;
333         }
334
335         if (ui->widgetImage != NULL)
336         {
337             ui->widgetImage->setObjectName(QString::fromUtf8("widgetImage"));
338
339             // add it to the scroll area
340             ui->scrollArea->setWidget(ui->widgetImage);
341
342             connect(processor, SIGNAL(outImageUpdated()),
343                     ui->widgetImage, SLOT(update()));
344
345             connect(processor, SIGNAL(outImageChanged(Mat*)),
346                     ui->widgetImage, SLOT(setSourceImage(Mat*)));
347
348             if (mode == RENDER_GL)
349             {
350                 connect(currentWindow,
351                         SIGNAL(screenChanged(QScreen *)),
352                         ui->widgetImage,
353                         SLOT(screenChanged()));
354                 connect(currentWindow,
355                         SIGNAL(screenChanged(QScreen *)),
356                         ui->widgetHistogram,
357                         SLOT(screenChanged()));
358                 connect(currentWindow,
359                         SIGNAL(screenChanged(QScreen *)),
360                         ui->widgetLUT,
```

```
                        SLOT(screenChanged()));
            }

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

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

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

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

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

    // Sets flipCheckbox and menu item states
    bool flipped = capture->isFlipVideo();
    ui->actionFlip->setChecked(flipped);
    ui->checkBoxFlip->setChecked(flipped);
}
```



```


/**
 * Setup UI from processor settings when launching application
 */
void MainWindow::setupUIfromProcessor()
{
    qDebug("Setting up UI from processor");


    // ------------------------------------------------------------------------
    // UI setup according to Histograms options
    // ------------------------------------------------------------------------
    // Histogram channel visibility
    QCheckBox * checkBoxesChannels[4] =
    {
        ui->checkBoxHistRed,
        ui->checkBoxHistGreen,
        ui->checkBoxHistBlue,
        ui->checkBoxHistGray
    };

    size_t nbHistograms = processor->getNbHistograms();

    for (size_t i = 0; i < nbHistograms; i++)
    {
        checkBoxesChannels[i]->setChecked(processor->isShowComponent(i));
    }

    if (nbHistograms < 4)
    {
        for (size_t i = nbHistograms; i < 4; i++)
        {
            checkBoxesChannels[i]->setEnabled(false);
        }
    }

    // Histogram mode
    if (processor->isCumulative())
    {
        ui->radioButtonHMCumulative->setChecked(true);
    }
    else
    {
        ui->radioButtonHMNormal->setChecked(true);
    }

    if (processor->isTimeCumulative())
    {
        ui->radioButtonHMTime->setChecked(true);
    }
    else
    {
        ui->radioButtonHMNormal->setChecked(true);
    }

    // Current LUT
    CvHistograms8UC3::TransfertType lutMode = processor->getLutType();

    switch (lutMode)
    {
        case CvHistograms8UC3::THRESHOLD_GRAY:
        case CvHistograms8UC3::THRESHOLD_COLOR:
            ui->radioButtonThreshold->setChecked(true);
            break;
        case CvHistograms8UC3::DYNAMIC_GRAY:
        case CvHistograms8UC3::DYNAMIC_COLOR:
            ui->radioButtonDynamic->setChecked(true);
            break;
        case CvHistograms8UC3::EQUALIZE_GRAY:
        case CvHistograms8UC3::EQUALIZE_COLOR:
            ui->radioButtonEqualize->setChecked(true);
            break;
        case CvHistograms8UC3::GAMMA:
            ui->radioButtonGamma->setChecked(true);
            break;
        case CvHistograms8UC3::NEGATIVE:
            ui->radioButtonInverse->setChecked(true);
            break;
        case CvHistograms8UC3::NONE:
        default:
            ui->radioButtonIdentity->setChecked(true);
            break;
    }

    // LUT mode : color/gray
    switch (lutMode)
    {
        case CvHistograms8UC3::THRESHOLD_COLOR:
        case CvHistograms8UC3::DYNAMIC_COLOR:
```

```
541         case CvHistograms8UC3::EQUALIZE_COLOR:
542             ui→radioButtonChColor→setChecked(true);
543             break;
544         case CvHistograms8UC3::THRESHOLD_GRAY:
545         case CvHistograms8UC3::DYNAMIC_GRAY:
546         case CvHistograms8UC3::EQUALIZE_GRAY:
547         case CvHistograms8UC3::GAMMA:
548         case CvHistograms8UC3::NEGATIVE:
549         case CvHistograms8UC3::NONE:
550         default:
551             ui→radioButtonChGray→setChecked(true);
552             break;
553     }
554
555     // If there is no additionnal grav level histogram we might change
556     // the channels radio buttons accordingly
557     if (¬processor→isComputeGray())
558     {
559 //      ui->radioButtonChGray->setChecked(false);
560         ui→radioButtonChColor→setChecked(true);
561         ui→radioButtonChGray→setEnabled(false);
562     }
563
564     // LUT param
565     ui→spinBoxlutParam→setValue((int)processor→getLUTParam());
566 }
567
568
569 /**
570  * Re setup processor from UI settings when source image changes
571  */
572 void MainWindow::setupProcessorFromUI()
573 {
574 //  qDebug("Setting up processor from UI");
575
576     // Sets histogram channel visibility
577     processor→setShowComponent(CvHistograms8UC3::HIST_RED,
578                                ui→checkBoxHistRed→isChecked());
579     processor→setShowComponent(CvHistograms8UC3::HIST_GREEN,
580                                ui→checkBoxHistGreen→isChecked());
581     processor→setShowComponent(CvHistograms8UC3::HIST_BLUE,
582                                ui→checkBoxHistBlue→isChecked());
583     if (processor→getNbHistograms() ≥ CvHistograms8UC3::HIST_GRAY)
584     {
585         processor→setShowComponent(CvHistograms8UC3::HIST_GRAY,
586                                    ui→checkBoxHistGray→isChecked());
587     }
588
589     // Sets Histogram mode
590     if (ui→radioButtonHMNormal→isChecked())
591     {
592         processor→setCumulative(false);
593         processor→setTimeCumulative(false);
594     }
595     else if (ui→radioButtonHMCumulative→isChecked())
596     {
597         processor→setCumulative(true);
598         processor→setTimeCumulative(false);
599     }
600     else
601     {
602         processor→setCumulative(false);
603         processor→setTimeCumulative(true);
604     }
605
606     processor→setLUTParam((float)ui→spinBoxlutParam→value());
607
608     if (ui→radioButtonIdentity→isChecked())
609     {
610         processor→setLutType(CvHistograms8UC3::NONE);
611     }
612     if (ui→radioButtonInverse→isChecked())
613     {
614         processor→setLutType(CvHistograms8UC3::NEGATIVE);
615     }
616     if (ui→radioButtonGamma→isChecked())
617     {
618         processor→setLutType(CvHistograms8UC3::GAMMA);
619     }
620     if (ui→radioButtonThreshold→isChecked())
621     {
622         if (ui→radioButtonChGray→isChecked())
623         {
624             processor→setLutType(CvHistograms8UC3::THRESHOLD_GRAY);
625         }
626         else
627         {
628             processor→setLutType(CvHistograms8UC3::THRESHOLD_COLOR);
629         }
630     }
```



```
631     if (ui→radioButtonDynamic→isChecked())
632     {
633         if (ui→radioButtonChGray→isChecked())
634         {
635             processor→setLutType(CvHistograms8UC3::DYNAMIC_GRAY);
636         }
637         else
638         {
639             processor→setLutType(CvHistograms8UC3::DYNAMIC_COLOR);
640         }
641     }
642     if (ui→radioButtonEqualize→isChecked())
643     {
644         if (ui→radioButtonChGray→isChecked())
645         {
646             processor→setLutType(CvHistograms8UC3::EQUALIZE_GRAY);
647         }
648         else
649         {
650             processor→setLutType(CvHistograms8UC3::EQUALIZE_COLOR);
651         }
652     }
653 }
654
655
656 /*
657  * Menu action to replace current image rendering widget by a
658  * QcvMatWidgetImage instance.
659  */
660 void MainWindow::on_actionRenderImage_triggered()
661 {
662     setupImageWidget(RENDER_IMAGE);
663 }
664
665 /*
666  * Menu action to replace current image rendering widget by a
667  * QcvMatWidgetLabel with pixmap instance.
668  */
669 void MainWindow::on_actionRenderPixmap_triggered()
670 {
671     setupImageWidget(RENDER_PIXMAP);
672 }
673
674 /*
675  * Menu action to replace current image rendering widget by a
676  * QcvMatWidgetGL instance.
677  */
678 void MainWindow::on_actionRenderOpenGL_triggered()
679 {
680     setupImageWidget(RENDER_GL);
681 }
682
683 /*
684  * Original size radioButton action.
685  * Sets capture resize to off
686  */
687 void MainWindow::on_radioButtonOrigSize_clicked()
688 {
689     ui→actionConstrainedSize→setChecked(false);
690     emit sizeChanged(QSize(0, 0));
691 }
692
693 /*
694  * Custom size radioButton action.
695  * Sets capture resize to preferred width and height
696  */
697 void MainWindow::on_radioButtonCustomSize_clicked()
698 {
699     ui→actionOriginalSize→setChecked(false);
700     emit sizeChanged(QSize(preferredWidth, preferredHeight));
701 }
702
703 /*
704  * Width spinbox value change.
705  * Changes the preferred width and if custom size is selected apply
706  * this custom width
707  * @param value the desired width
708  */
709 void MainWindow::on_spinBoxWidth_valueChanged(int value)
710 {
711     preferredWidth = value;
712     if (ui→radioButtonCustomSize→isChecked())
713     {
714         emit sizeChanged(QSize(preferredWidth, preferredHeight));
715     }
716 }
717
718 /*
719  * Height spinbox value change.
720  * Changes the preferred height and if custom size is selected apply
```

```
721  * this custom height
722  * @param value the desired height
723  */
724 void MainWindow::on_spinBoxHeight_valueChanged(int value)
725 {
726     preferredHeight = value;
727     if (ui->radioButtonCustomSize->isChecked())
728     {
729         emit sizeChanged(QSize(preferredWidth, preferredHeight));
730     }
731 }
732
733 /*
734  * Flip capture image horizontally.
735  * changes capture flip status
736  */
737 void MainWindow::on_checkBoxFlip_clicked()
738 {
739     /*
740      * There is no need to update ui->actionFlip since it is connected
741      * to ui->checkBoxFlip through signals/slots
742      */
743     // capture->setFlipVideo(ui->checkBoxFlip->isChecked());
744     emit flipVideo(ui->checkBoxFlip->isChecked());
745 }
746
747 /*
748  * Set transfert function to identity
749  */
750 void MainWindow::on_radioButtonIdentity_clicked()
751 {
752     processor->setLutType(CvHistograms8UC3::NONE);
753 }
754
755 /*
756  * Set transfert function to inverse
757  */
758 void MainWindow::on_radioButtonInverse_clicked()
759 {
760     processor->setLutType(CvHistograms8UC3::NEGATIVE);
761 }
762
763 /*
764  * Set transfert function to gamma
765  */
766 void MainWindow::on_radioButtonGamma_clicked()
767 {
768     processor->setLutType(CvHistograms8UC3::GAMMA);
769 }
770
771 /*
772  * Set transfert function to threshold
773  */
774 void MainWindow::on_radioButtonThreshold_clicked()
775 {
776     if (ui->radioButtonChGray->isChecked())
777     {
778         processor->setLutType(CvHistograms8UC3::THRESHOLD_GRAY);
779     }
780     else
781     {
782         processor->setLutType(CvHistograms8UC3::THRESHOLD_COLOR);
783     }
784 }
785
786 /*
787  * Set transfert function to optimal dynamic
788  */
789 void MainWindow::on_radioButtonDynamic_clicked()
790 {
791     if (ui->radioButtonChGray->isChecked())
792     {
793         processor->setLutType(CvHistograms8UC3::DYNAMIC_GRAY);
794     }
795     else
796     {
797         processor->setLutType(CvHistograms8UC3::DYNAMIC_COLOR);
798     }
799 }
800
801 /*
802  * Set transfert function to equalization
803  */
804 void MainWindow::on_radioButtonEqualize_clicked()
805 {
806     if (ui->radioButtonChGray->isChecked())
807     {
808         processor->setLutType(CvHistograms8UC3::EQUALIZE_GRAY);
809     }
810     else
```



```
811     {
812         processor->setLutType(CvHistograms8UC3::EQUALIZE_COLOR);
813     }
814 }
815
816 /*
817  * Set transfert function depending on processor to use colors
818  * components of the histogram generating 1 transfert function per image
819  * channels
820  */
821 void MainWindow::on_radioButtonChColor_clicked()
822 {
823     CvHistograms8UC3::TransfertType type = processor->getLutType();
824     switch (type)
825     {
826         case CvHistograms8UC3::THRESHOLD_GRAY:
827             processor->setLutType(CvHistograms8UC3::THRESHOLD_COLOR);
828             break;
829         case CvHistograms8UC3::DYNAMIC_GRAY:
830             processor->setLutType(CvHistograms8UC3::DYNAMIC_COLOR);
831             break;
832         case CvHistograms8UC3::EQUALIZE_GRAY:
833             processor->setLutType(CvHistograms8UC3::EQUALIZE_COLOR);
834             break;
835         // in all other cases do nothing
836         case CvHistograms8UC3::NONE:
837         case CvHistograms8UC3::GAMMA:
838         case CvHistograms8UC3::NEGATIVE:
839         default:
840             // Nothing
841             break;
842     }
843 }
844
845 /*
846  * Set transfert function depending on processor to use gray level
847  * histogram component generating 1 transfert function per image
848  * channels
849  */
850 void MainWindow::on_radioButtonChGray_clicked()
851 {
852     CvHistograms8UC3::TransfertType type = processor->getLutType();
853     switch (type)
854     {
855         case CvHistograms8UC3::THRESHOLD_COLOR:
856             processor->setLutType(CvHistograms8UC3::THRESHOLD_GRAY);
857             break;
858         case CvHistograms8UC3::DYNAMIC_COLOR:
859             processor->setLutType(CvHistograms8UC3::DYNAMIC_GRAY);
860             break;
861         case CvHistograms8UC3::EQUALIZE_COLOR:
862             processor->setLutType(CvHistograms8UC3::EQUALIZE_GRAY);
863             break;
864         // in all other cases do nothing
865         case CvHistograms8UC3::NONE:
866         case CvHistograms8UC3::GAMMA:
867         case CvHistograms8UC3::NEGATIVE:
868         default:
869             // Nothing
870             break;
871     }
872 }
873
874 /*
875  * Modify lut parameter applied to transfert function depending on
876  * histogram
877  * @param value the new value of lutParam
878  */
879 void MainWindow::on_spinBoxlutParam_valueChanged(int value)
880 {
881     processor->setLUTParam((float)value);
882 }
883
884 /*
885  * Set histogram mode to normal
886  */
887 void MainWindow::on_radioButtonHMNormal_clicked()
888 {
889     processor->setTimeCumulative(false);
890     processor->setCumulative(false);
891 }
892
893 /*
894  * Set Histogram mode to cumulative
895  */
896 void MainWindow::on_radioButtonHMCumulative_clicked()
897 {
898     processor->setTimeCumulative(false);
899     processor->setCumulative(true);
900 }
```

```
901
902 /*
903  * set Histogram mode to time cumulative
904  */
905 void MainWindow::on_radioButtonHMTime_clicked()
906 {
907     processor→setCumulative(false);
908     processor→setTimeCumulative(true);
909 }
910
911 /*
912  * Show/Hides histogram red component
913  */
914 void MainWindow::on_checkBoxHistRed_clicked()
915 {
916     processor→setShowComponent((size_t)CvHistograms8UC3::HIST_RED,
917                                ui→checkBoxHistRed→isChecked());
918 }
919
920 /*
921  * Show/Hides histogram green component
922  */
923 void MainWindow::on_checkBoxHistGreen_clicked()
924 {
925     processor→setShowComponent((size_t)CvHistograms8UC3::HIST_GREEN,
926                                ui→checkBoxHistGreen→isChecked());
927 }
928
929 /*
930  * Show/Hides histogram blue component
931  */
932 void MainWindow::on_checkBoxHistBlue_clicked()
933 {
934     processor→setShowComponent((size_t)CvHistograms8UC3::HIST_BLUE,
935                                ui→checkBoxHistBlue→isChecked());
936 }
937
938 /*
939  * Show/Hides histogram gray component
940  */
941 void MainWindow::on_checkBoxHistGray_clicked()
942 {
943     processor→setShowComponent((size_t)CvHistograms8UC3::HIST_GRAY,
944                                ui→checkBoxHistGray→isChecked());
945 }
```



```
1  #include <QApplication>
2  #include <QThread>
3  #include <QDebug>
4  #include <libgen.h>      // for basename
5  #include <iostream>      // for cout
6
7  #include "QcvVideoCapture.h"
8  #include "CaptureFactory.h"
9  #include "QcvHistograms.h"
10 #include "mainwindow.h"
11
12 /**
13  * Usage function shown just before launching QApp
14  * @param name the name of the program (argv[0])
15  */
16 void usage(char * name);
17
18 /**
19  * Test program OpenCV2 + QT5
20  * @param argc argument count
21  * @param argv argument values
22  * @return QTApp return value
23  * @par usage : <Progname> [--device | -d] <#> | [--file | -f ] <filename>
24  * [--mirror | -m] [--gray | -g] [--size | -s] <width>x<height>
25  *  - device : [--device | -d] <device #> (0, 1, ...) Opens capture device #
26  *  - filename : [--file | -f ] <filename> Opens a video file or URL (including rtsp)
27  *  - mirror : mirrors image horizontally before display
28  *  - gray : turns on source image gray conversion
29  *  - size : [--size | -s] <width>x<height> resize capture to fit desired <width>
30  *  and <height>
31  */
32 int main(int argc, char *argv[])
33 {
34     // ----------------------------------------------------------------------
35     // Meta types registration for using these types in queued signal / slots
36     // ----------------------------------------------------------------------
37     qRegisterMetaType<CvProcessor::ProcessTime>("CvProcessor::ProcessTime");
38
39     // ----------------------------------------------------------------------
40     // Instanciate QApplication to receive special QT args
41     // ----------------------------------------------------------------------
42     QApplication app(argc, argv);
43
44     // Gets arguments after QT specials removed
45     QStringList argList = QCoreApplication::arguments();
46
47     int threadNumber = 3;
48     // parse arguments for --threads tag
49     for (QListIterator<QString> it(argList); it.hasNext(); )
50     {
51         QString currentArg(it.next());
52
53         if (currentArg == "-t" ∨ currentArg =="--threads")
54         {
55             // Next argument should be thread number integer
56             if (it.hasNext())
57             {
58                 QString threadString(it.next());
59                 bool convertOk;
60                 threadNumber = threadString.toInt(&convertOk,10);
61                 if (¬convertOk ∨ threadNumber < 1 ∨ threadNumber > 3)
62                 {
63                     qWarning("Warning: Invalid thread number %d",threadNumber);
64                     threadNumber = 3;
65                 }
66             }
67             else
68             {
69                 qWarning("Warning: thread tag found with no following thread number");
70             }
71         }
72     }
73     // ----------------------------------------------------------------------
74     // Create Capture factory using program arguments and
75     // open Video Capture
76     // ----------------------------------------------------------------------
77     CaptureFactory factory(argList);
78     factory.setSkippable(true);
79
80     // Helper thread for capture
81     QThread * capThread = NULL;
82     if (threadNumber > 1)
83     {
84         capThread = new QThread();
85     }
86
87     // Capture
88     QcvVideoCapture * capture = factory.getCaptureInstance(capThread);
89
90     // ----------------------------------------------------------------------
```

```
91      // Create QHistandLUT
92      // ------------------------------------------------------------------------
93      // Helper thread for processor
94      QThread * procThread = NULL;
95      if (threadNumber > 2)
96      {
97          procThread = new QThread();
98      }
99      else
100     {
101         if (threadNumber > 1)
102         {
103             procThread = capThread;
104         }
105     }
106
107     // Processsor
108     QcvHistograms * histograms = NULL;
109     if (procThread ≡ NULL)
110     {
111         histograms = new QcvHistograms(capture→getImage());
112     }
113     else
114     {
115         if (procThread ≠ capThread)
116         {
117             histograms = new QcvHistograms(capture→getImage(),
118                                            capture→getMutex(),
119                                            procThread);
120         }
121         else // procThread == capThread
122         {
123             histograms = new QcvHistograms(capture→getImage(),
124                                            NULL,
125                                            procThread);
126         }
127     }
128
129     // ------------------------------------------------------------------------
130     // Connects capture to Histograms
131     // ------------------------------------------------------------------------
132     // Connects capture update to QHistandLUT update
133     QObject::connect(capture, SIGNAL(updated()),
134                      histograms, SLOT(update()),
135                      ((threadNumber < 3) ? Qt::DirectConnection :
136                                            Qt::QueuedConnection));
137
138     // connect capture changed image to QHistandLUT set input
139     QObject::connect(capture, SIGNAL(imageChanged(Mat*)),
140                      histograms, SLOT(setSourceImage(Mat*)),
141                      ((threadNumber < 3) ? Qt::DirectConnection :
142                                            Qt::QueuedConnection));
143     // ------------------------------------------------------------------------
144     // Now that Capture & QHistandLUT are on then
145     // add our MainWindow as toplevel
146     // and launches app
147     // ------------------------------------------------------------------------
148     MainWindow w(capture, histograms);
149     w.show();
150
151     usage(argv[0]);
152
153     int retVal = app.exec();
154
155     // ------------------------------------------------------------------------
156     // Cleanup & return
157     // ------------------------------------------------------------------------
158     delete histograms;
159     delete capture;
160     qDebug() << "Processor and Capture deleted";
161     bool sameThread = capThread ≡ procThread;
162
163     if (capThread ≠ NULL)
164     {
165         delete capThread;
166         qDebug() << "Capture Thread deleted";
167     }
168
169     if (procThread ≠ NULL ∧ ¬sameThread)
170     {
171         delete procThread;
172         qDebug() << "Processor Thread deleted";
173     }
174
175     return retVal;
176 }
177
178 /*
179  * Usage function shown just before launching QApp
180  * @param name the name of the program (argv[0])
```



```
181  */
182 void usage(char * name)
183 {
184     cout << "usage : " << basename(name) << " "
185          << "[-d | --device] <device number> "
186          << "[-v | --video] <video file> "
187          << "[-s | --size] <width>x<height> "
188          << "[-m | --mirror]"
189          << "[-t | --threads] <number of threads [1..3]>"
190          << endl;
191 }
```