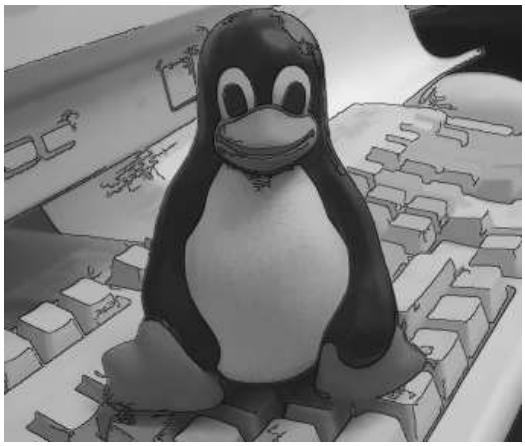
Qt's Image, Video and Computer Vision Library

0.0.5



Warning:
Development of this library is still on Alpha stage. This means documentation, API and functionality may vary for the first release version of this library.

Index

Introduction

About the library.

QVision is an image and computer vision framework, developed by the PARP Computer Perception Research Group, from the University of Murcia, Spain. It's main purpose is to help research for new algorithms in the fields of **video and image processing**, and **Computer Vision**. It's open source, so it is oriented to academic and research audience.

Copyright, license and warranty.

Documentation.

This is the actual API. It's format is Doxygen, so you can find anything you will find in a Doxygen documentation API. Besides, it includes some tematic sections about starting to work and using the QVision :

- The **Installation guide** reviews how to install and configure the framework from zero, over a clean system.
- The **Programming guide** is a good starting point for QVision programming. It reviews the creation of a very simple application with the framework, and includes links to the most important parts of the documentation for futher info and advanced usage.

Feedback.

These mails are for contact purposes, bugs, questions, etc:

Pedro E. Lopez de Teruel Alcolea <pedroe@ditec.um.es>
Antonio L. Rodríguez López <alrl1@alu.um.es>

Feel free to send bug reports, suggestions about new features and functionality, improvements in the documentation, or any information you find useful in the development of the QVision library, but don't abuse of them. Remember that karma bites back.

Installation guide

This guide is supposed to help the installation and configuration of the QVision over a cleaned system.

Package and program dependencies of the QVision.

Right now, QVision needs the following libraries and programs to be installed in the system. In a future, it will be a modular library, so you will be able to compile the core functionality without unnecessary functionality:

- Qt, Framework for high performance, cross-platform application development. Version 4.0 or later.
  This is a core library, and is needed for the QVision core.
- Intel(R) Integrated Performance Primitives (IPP). Version 5.2 or later.
  This library is needed for fast image filtering and processing.
- MPlayer and MEncoder.
  This program are needed for the QVMplayerCamera class, mainly for reading from a video source.
- QWT - Qt Widgets for Technical Applications. Version 5.0.1-2 or later.
  This library is needed for some of the widgets in the graphical user interface.
- GSL - GNU Scientific Library. Version 1.9-3 or later.
  This library is needed for math computations, on matrices, vectors, and tensors.

If you are planning to install QVision on an Ubuntu or Debian system, you can directly install the following packages using apt-get, synaptic, or adept package managers (or similar packaging system):

- Packages for Qt library: *libqt4-core, libqt4-debug, libqt4-dev, libqt4-gui, libqt4-qt3support, qt4-designer, qt4-dev-tools* and *qt4-doc*.
  Version 4.3.2-0ubuntu3.1 of these packages was tested and worked for current version of QVision.
- Packages for QWT library: *libqwt5-qt4* and *libqwt5-qt4-dev*.
  Version 5.0.1-2 of these packages was tested and worked for current version of QVision.
- Packages for MPlayer: *mencoder* and *mplayer*.
  Version 2:1.0 was tested and worked for current version of QVision.
- Packages for GSL: *libgsl0* and *libgsl0-dev*.
  Version 1.9-3 was tested and worked for current version of QVision. Blas and LaPack libraries will be installed too with this library.
- Packages for Intel's IPP: there are no packages available in apt-get for that library. A free version is available for Linux platforms. You can download it from Intel's IPP homepage. Version 5.3 of this library was tested and worked for current version of QVision.

Also, some Ubuntu systems may need to install package *g++*, needed for *Qt* to compile.

Downloading the framework.

Latest release and older versions can be downloaded from the following url:

http://forja.rediris.es/frs/?group_id=321

Configuring.

Once you have the tar file for QVision, named *QVision.<version>.tgz*, copy it to your home directory (or a temporary location for compiling), and untar-it using this line:

```
# tar xzvf QVision.<version>.tgz
```

Then you should copy the file *QVision/config.pri.example* to a file in the same directory, *QVision*, named *config.pri*. It has some performance and system configuration parameters inside used to compile QVision library. Change the content of the variables *INSTALL_PATH*, *IPP_DIR*, *QWT_INCLUDE*, *QWT_LIB_DIRECTORY*, and *QWT_LIB_NAME*, according to the specifications found in the *config.pri.example* file about them. They contain information about the location of the QWT and Intel's IPP libraries, and the install path for the QVision library.

Beside system and directories configuration, you can also tune performance and debugging options in the 'config.pri' file. You may uncomment the line

```
CONFIG += release
```

to compile a faster version of the library, or else the library will compile in debug mode, including debug and error checking at execution time code. You can also uncomment the line:

```
DEFINES += QT_NO_DEBUG_OUTPUT
```

if you don't want to be printed lots of debug information while the program is running when compiling in debug mode, that will heavily decrease the execution time.

Compilation and Install.

Once customized the program, type:

```
# cd QVision
# qmake
# make
```

to compile the library. When compilation is done, you should install the library. There will be copied some files in the directory specified in the variable 'INSTALL_PATH'. If that route is in your home directory, or other place you have permissions to write, you should do the following:

```
# make install
```

or else you should use *sudo* console command, to copy the files as a super-user:

```
# sudo make install
```

to make the installation proceed. If you need to delete the installation, simply compile again, and use the following line:

```
# sudo make uninstall
```

it will erase QVision's library files from the directory where you installed it previously.

Programming guide

QVision projects

QVision is developed over Qt library, so every program should be compiled inside a Qt project.

To create a Qt project you need to write a *.pro* file. This is like an advanced *Makefile* file, were you should specify the source and header files, compilation options, libraries, and other stuff, that you will include or use in your project. With this *.pro* file you can use the *qmake* program, which is a tool from the Qt library, to create the specific *Makefile* for the project, that you can use to finaly compile the program, with the Qt binaries linked correctly, and everything properly configured for your machine.

To make a program based on QVision you should create a Qt project file, that at least references to all the source and header files, and includes the project file for QVision *qvproject.pri*. That file should be located in the install directory of the QVision, and should be included in the *.pro* file with this line:

```
include(<path to QVision>/qvproject.pri)
```

where *<path to QVision>* should be the absolute path to QVision install directory. For example:

```
include(/usr/local/QVision.0.0.5/qvproject.pri)
```

This line includes in the project all the references to the library binaries, and configurations that require the use of the QVision.

In the next section it is shown an example *.pro* file for a simple QVision project. For futher info about the *qmake* tool, and the sintax of *.pro* files you can check the online manual for QMake.

The first program

Here is detailed how to make a simple QVision project, with a *.pro* file and all the code in just one *.cpp* source file. Supossing you have installed QVision in the url */usr/local/QVision.0.0.5* you can create a file called *example.pro* with the following content:

```
include(/usr/local/QVision.0.0.5/qvproject.pri)
TARGET = example

# Input
SOURCES += example.cpp
```

This file must include the project file for QVision projects:

```
include(/usr/local/QVision.0.0.5/qvproject.pri)
```

If you don't have QVision installed in that directory just change the path in the first line according to what is explained in section **QVision projects**. Next, you can create the file *example.cpp* which will contain the code for the application. Note that this file is referenced in the file *example.pro* at the line:

```
SOURCES += example.cpp
```

That makes the project to include this source file. File *example.cpp* will contain the following file includes:

```
#include <QVMplayerCamera>
#include <QVApplication>
#include <QVGUI>
```

and the following <it>main</it> function:

```
int main(int argc, char *argv[])
        {
        // QVApplication object
        QVApplication app(argc, argv, "Example program for QVision library. Play a video on a canvas window.");

        QVMPlayerCamera camera("Video");        // QVCamera object
        PlayerWorker worker("Player worker");   // QVWorker object
        camera.link(&worker,"Input image");

        // GUI object
        QVGUI interface;

        // Video/image output object
        QVImageCanvas imageCanvas("Test image");
        imageCanvas.linkProperty(worker,"Output image");

        return app.exec();
        }
```

The class **QVMPlayerCamera** is a subclass for **QVCamera**. Both represent video or image inputs in the QVision framework, but the class **QVCamera** is virtual, and every class modelling video or image inputs in the QVision should inherit from it. Actually, it only has one subclass, the **QVMPlayerCamera** class, which is a flexible video input that can read frames from diverse video file formats, webcams and digital video cameras. It is based on the *MPlayer* program. To allow a worker object to read frames from a video source, it should be linked to the camera object, specifying the name of the image for the worker, with a line like this:

```
camera.link(&worker,"Input image");
```

The object **QVApplication** registers worker and camera objects in the system, and process command line input parameters. It inherits from the QApplication class, so it has a similar functionality. It is the most important object in a QVision application, so it always should be created first in the *main()* function, before any worker, camera or graphic interface object. Only one object derived from this class should be created in a QVision application.

The class **QVGUI** is the main GUI widget. This object reads the workers and cameras registered in the **QVApplication**

object, so it contains buttons and sliders to control the algorithm parameters (computer vision or video processing algorithms parameters) for the system, and the video input flow, allowing the user to stop an resume frame read from the video sources opened by the program, in execution time. Also, it can display information about cpu usage stadistics, frame rate, seconds read from the video inputs, etc... and it allows the user to stop and close the program. It is recommended to create only one object of its type for any QVision application.

The class **QVImageCanvas** is a video or image output widget. Any object created with it will display an image window showing an output image from the worker object. Like the camera object, it should be connected to the worker to read the resulting images, specifying the name of the output image with a line like this:

```
imageCanvas.linkProperty(worker,"Output image");
```

Warning:
Because the *exec()* function is not *virtual* in the QApplication class, make sure you call the *exec()* function from the **QVApplication** class, not from the QApplication class, in a QVision application. Basically, avoid creating and using **QVApplication** objects like this:

```
int main(int argc, char *argv[])
        {
        // QVApplication object
        QApplication *app = new QVApplication(argc, argv, "Example program. Play a video on a canvas window.");

        [...}

        return app->exec(); // this will execute QApplication::exec() function, not QVApplication::exec() function.
        }
```

The worker object

As it was previously commented, worker objects should contain the code for the computer vision and/or image processing algorithms, that the application uses. Every worker object must define two functions: a constructor, that adds the proper dynamic properties for the worker, and a **QVWorker::iterate()** function. The latter is a virtual function that should be redefined containing the code for the *Computer vision / Image processing algorithms* task/block in the application.

This function should read input images, parameters, and other data inputs from dynamic properties contained in the worker, process them, and store the resulting images, data structures, or values, in other dynamic properties contained in the worker object.

The file *example.cpp* must contain the following code for the worker:

```
class PlayerWorker: public QVWorker
        {
        public:
                PlayerWorker(QString name): QVWorker(name)
                        {
                        addProperty< QVImage<uChar,1> >("Input image", inputFlag|outputFlag);
                        addProperty< QVImage<uChar,1> >("Output image", outputFlag);
                        }

                void iterate()
                        {
                        QVImage<uChar,1> image = getPropertyValue< QVImage<uChar,1> >("Input image");

                        // image processing / computer vision code should go here

                        setPropertyValue< QVImage<uChar,1> >("Output image", image);
                        }
        };
```

This worker has no input parameters, and it just reads an input image, and stores it in an output property.

Compiling and executing the program

Once files *example.cpp* and *example.pro* are created and stored in the same directory, you can compile from that location writting these instructions in the command line:

```
# qmake
# make
```

This will generate the binary *example*. Note that the name of the executable was specified in the *example.pro* file, with the line:

```
TARGET = example
```

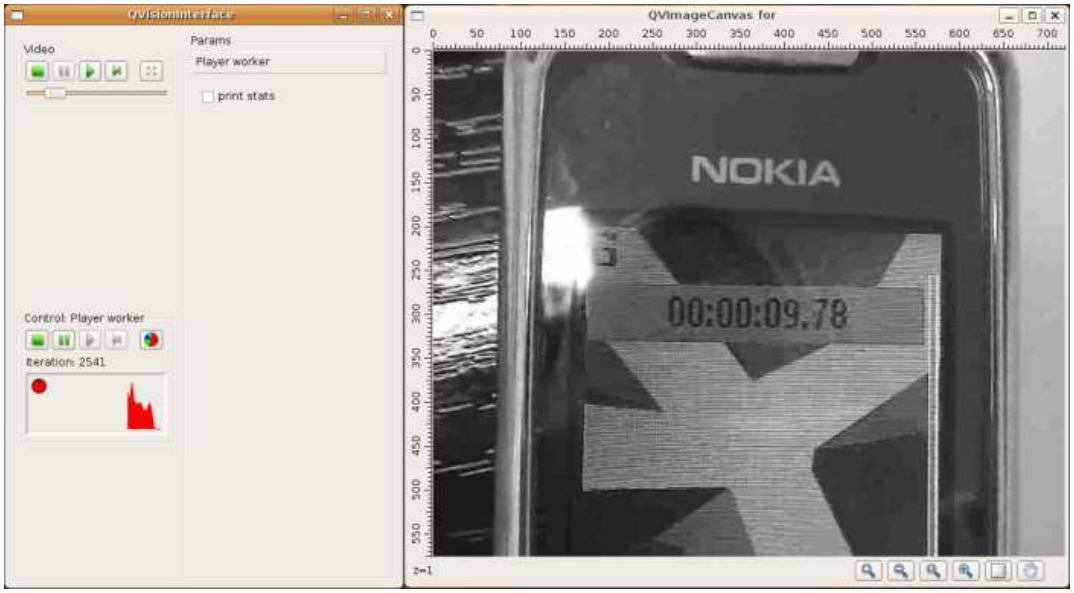
You can execute the example program with this command line:

```
# ./example --URL=http://perception.inf.um.es/public_data/videos/misc/minuto.avi
```

The program will start and display the video pointed by the URL in a window:

User interface

The program opens some windows and widgets for user input-output. Below is an image of the look of the interface:



Beside that interface, the user can set initial values for some properties, through comman line parameters. You can check the properties of the *example* program with the command line parameter *help*. Every QVision application recognices that parameter, and shows a list of the usage and main properties that you can set through the command line. The output of the program *example* when invoqued with that parameter:

```
# ./example --help
```

is:

```
Usage: ./example [OPTIONS]
Example program for QVision library. Play a video on a canvas window.

Input parameters for Video:
--Rows=[int] (def. 0) ............................. Rows to open the camera.
--Cols=[int] (def. 0) ............................. Columns to open the camera.
--RealTime=[true,false](def. false)  If the camera should be opened in real time mode.
--Deinterlaced=[true,false](def. false)  If the camera should be opened in deinterlaced mode.
--NoLoop=[true,false](def. false)  If the camera should be opened in no loop mode.
--RGBMEncoder=[true,false](def. false)  If the camera should be opened in RGB using mencoder.
--URL=[text] ............................ URL of the video source (see doc.).

Input parameters for Player worker:
--print stats=[true,false](def. false)  Enables realtime stats console output for worker.
```

You can change the input video source, changing the value for the *URL* command line parameter, so the example program reads from a video file different than the file *minuto.avi*, as well as other video input configuration parameters such as the size of the image (parameters *Cols* and *Rows*), amongst others. For example:

```
# ./example --URL=http://perception.inf.um.es/public_data/videos/misc/penguin.dv
```

Or:

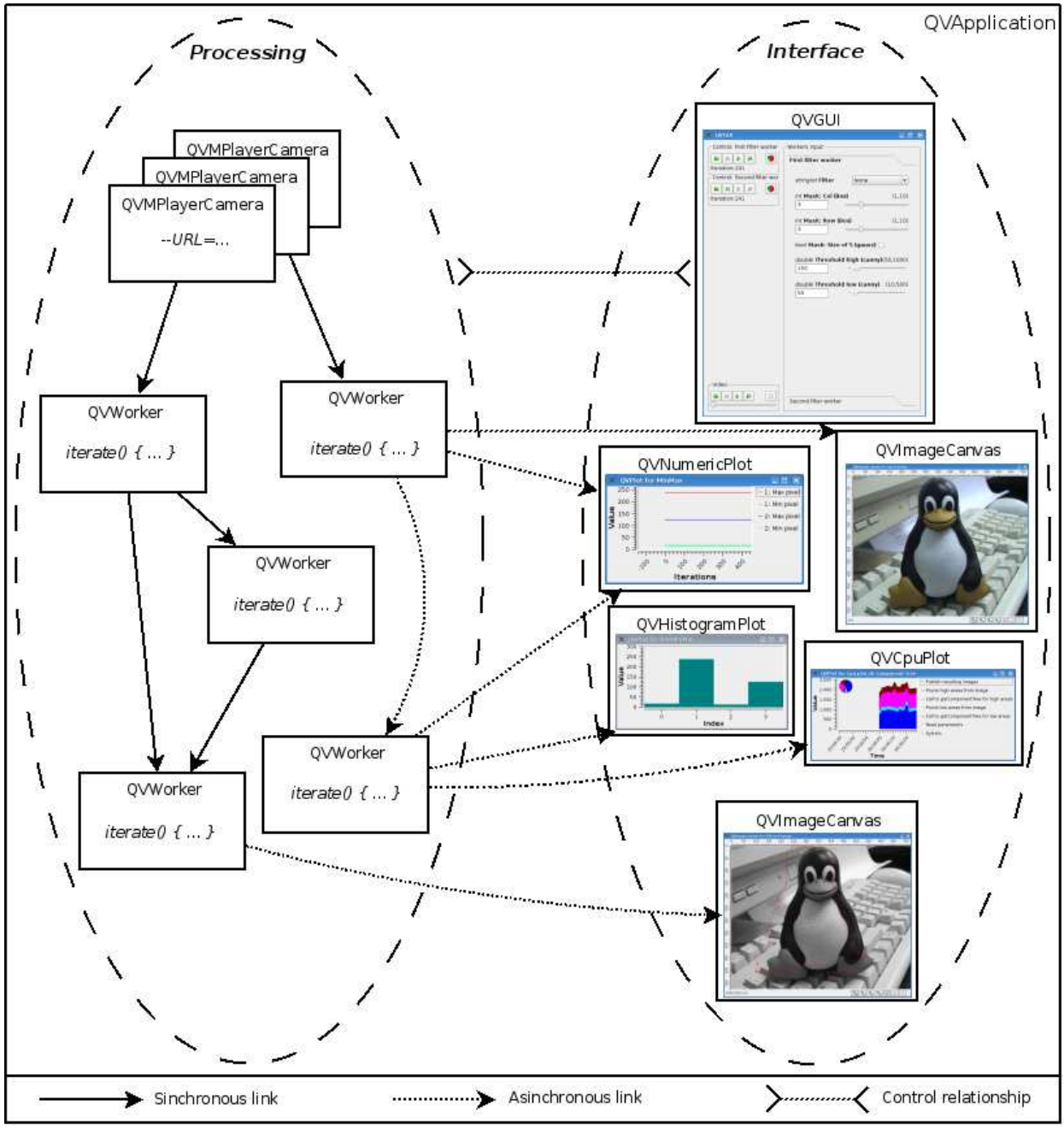
```
# ./example --URL=http://perception.inf.um.es/public_data/videos/misc/penguin.dv --Rows=240 --Cols=
```

QVision's command line parameter system will be explained in the section TheGUI, along the graphical interface, but it is best recommended to read the following section of this documentation, Programming, to understand how to add and use these parameters.

An advanced programming example

Program Structure

Conceptually, a QVision program haves the following structure:



In witch the items have the next functionality:

- **QVApplication** wraps all following items.
- Each **QVMPlayerCamera** generates a input parameters program's section, in witch the user can indicate the input video URL, size, ... And holds a **QVImage** output property containing video's current shot.
- Each one-way arrow indicates a propery link between two objects, it can be synchronous or asynchronous linked (continuous or discontinuous arrow), depending if a object must wait the end of the next object's iteration.
- Each **QVWorker** is a processing unit, we can redefine its iterate() to process continously any input property and store it in any ouput property.
- **QVGUI** lets us interact with the processing objects: we can stop, play and pause cameras and workers, and we can interact with unlinked workers input properties of some property types (bool, int, double, QString and **QVIndexedStringList**).
- Each plot or canvas shows properties of some property types linked to it. For example, **QVNumericPlot** shows int and double properties, **QVHistogramPlot** shows QList<double> properties, **QVCpuPlot** shows QVCpuStat properties and **QVImageCanvas** shows QVImages properties.

First Step: creating basic files

Now it is shown an example *.pro* file for a simple QVision project. For futher info about the *qmake* tool, and the sintax of *.pro* files you can check the online manual for QMake.

Here is detailed how to make a simple QVision project, with a *.pro* file and all the code in just one *.cpp* source file. Supossing you have installed QVision in the url */usr/local/QVision.0.0.5* you can create a file called *features.pro* with the following content:

```
include(/usr/local/QVision.0.0.5/qvproject.pri)
TARGET = features
```

```
# Input
SOURCES += features.cpp
```

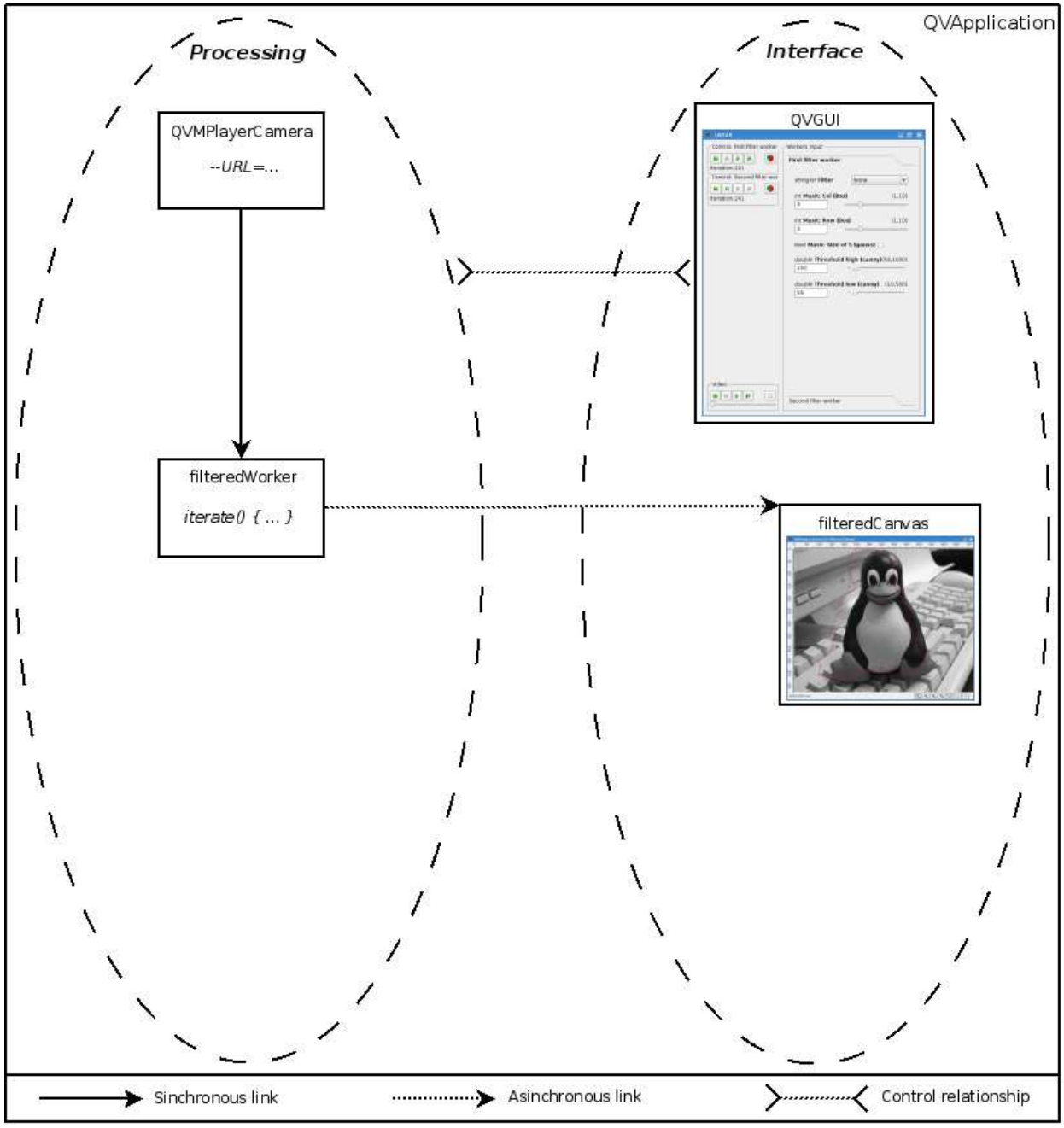
If you don't have QVision installed in that directory just change the path in the first line according to what is explained in section **QVision projects**.

Next, you can create the file *features.cpp*:

```
#include <QVApplication>
#include <QVMPlayerCamera>
#include <QVGUI>
#include <QVImageCanvas>
#include <QVFilterSelectorWorker>

int main(int argc, char *argv[])
        {
        QVApplication app(argc, argv,
                "Example program for QVision library. Obtains several features from input video frames."
                );

        QVFilterSelectorWorker<uChar, 3> filterWorker("Filter worker");

        QVMPlayerCamera camera("Video");
        camera.link(&filterWorker, "Input image");

        QVGUI interface;

        QVImageCanvas filteredCanvas("Input");
        filteredCanvas.linkProperty(filterWorker, "Output image");

        return app.exec();
        }
```

This example create a simple QVisionApplication, with the following items: contains an input **QVMPlayerCamera** to get the input source video; a **QVWorker**, that is of an internal QVision worker type (QVFilterSelectorWorker) that lets us select a filter to apply over the "Input image" property; a **QVGUI** interface for interacts with the items; and a **QVImageCanvas** to show the output processed image. As well as creating these items, we need to connect them, as doing in 'camera.link(&filterWorker, "Input image")' and 'filteredCanvas.linkProperty(filterWorker, "Output image")', in witch we connect the camera to the worker and the worker to the canvas.

This example's first part has the following structure:



Note that the file *features.cpp* is referenced in the file *features.pro*, at the line:

```
SOURCES += features.cpp
```

Also note that as it is explained in the previous section **The first program**, the project file also includes the project file for QVision projects:

```
include(/usr/local/QVision.0.0.5/qvproject.pri)
```

If both files are in the same directory, you can compile from that location writting these instructions in the command line:

```
# qmake
# make
```

This will generate the binary *features*. Note that the name of the executable was specified in the *features.pro* file, in the line:

```
TARGET = features
```

You can execute the example program with this command line:

```
# ./features --URL=http://perception.inf.um.es/public_data/videos/misc/penguin.dv
```

It is just a simple video player, that will show the video in the file pointed by the URL, in a window, with some control widgets.

Second Step: adding ower oun worker

Now we will add a new worker to process the filtered video:

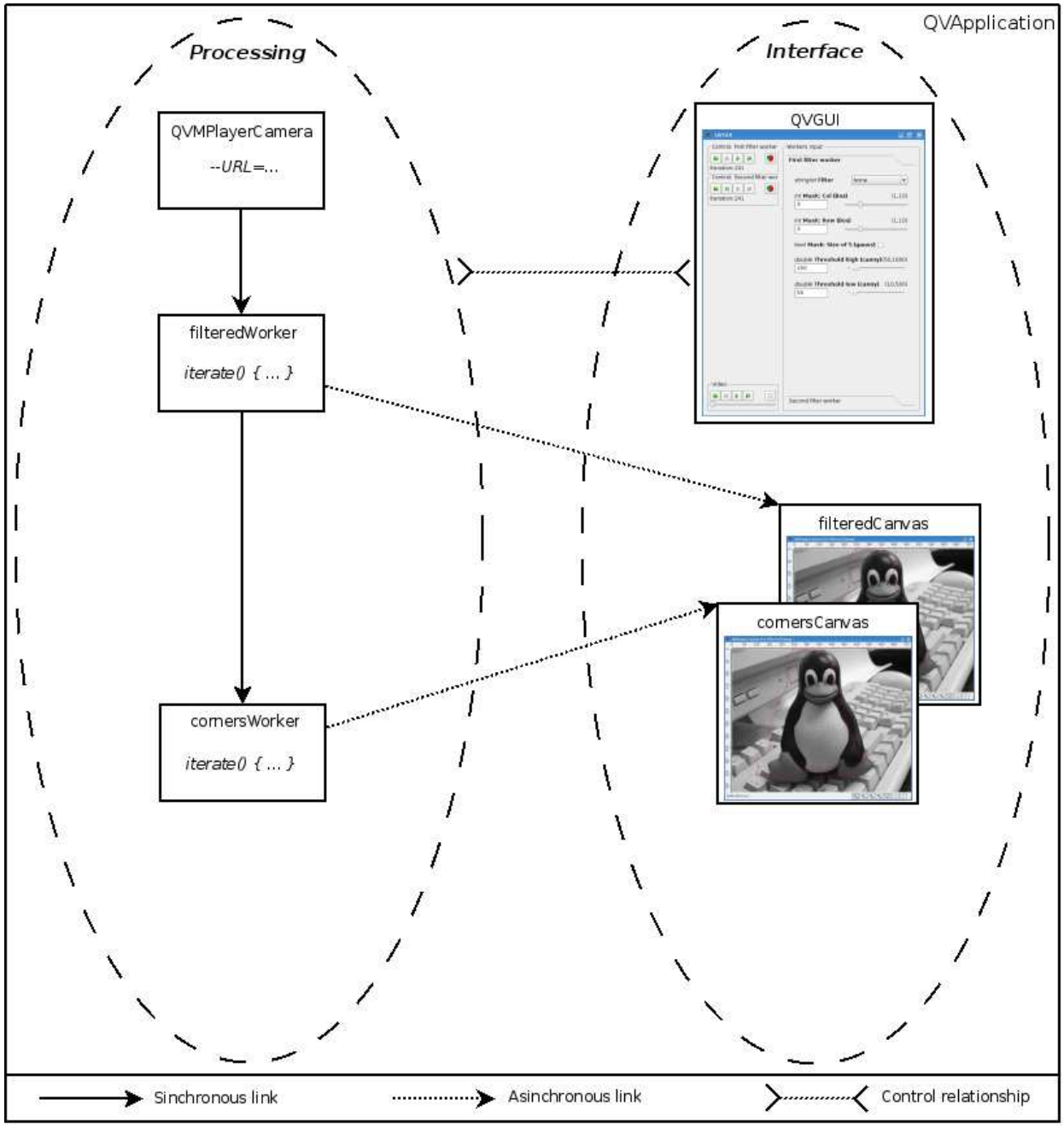
```
#include <QVApplication>
```

```
#include <QVMPlayerCamera>
#include <QVGUI>
#include <QVImageCanvas>
#include <QVFilterSelectorWorker>

class HarrisExtractorWorker: public QVWorker
        {
        public:
                HarrisExtractorWorker(QString name): QVWorker(name)
                        {
                        addProperty< int >("Points", inputFlag, 15, "window size ", 1, 100);
                        addProperty< double >("Threshold", inputFlag, 1.0, "window size ", 0.0, 256.0);
                        addProperty< QVImage<uChar,3> >("Input image", inputFlag|outputFlag);
                        addProperty< QList<QPointF> >("Corners", outputFlag);
                        }

                void iterate()
                        {
                        // 0. Read input parameters
                        QVImage<uChar> image = getPropertyValue< QVImage<uChar,3> >("Input image");
                        const double threshold = getPropertyValue<double>("Threshold");
                        const int pointNumber = getPropertyValue<int>("Points");
                        timeFlag("grab Frame");

                        // 1. Obtain corner response image.
                        QVImage<sFloat> cornerResponseImage(image.getRows(), image.getCols());
                        FilterHessianCornerResponseImage(image, cornerResponseImage);
                        timeFlag("Corner response image");

                        // 2. Local maximal filter.
                        QList<QPointF> hotPoints = GetMaximalResponsePoints3(cornerResponseImage, threshold);
                        timeFlag("Local maximal filter");

                        // 3. Output resulting data.
                        setPropertyValue< QList<QPointF> >("Corners",
                                                           hotPoints.mid(MAX(0,hotPoints.size() - pointNumber)));
                        }
        };

int main(int argc, char *argv[])
        {
        QVApplication app(argc, argv,
                "Example program for QVision library. Obtains several features from input video frames."
                );

        QVFilterSelectorWorker<uChar, 3> filterWorker("Filter worker");
        HarrisExtractorWorker cornersWorker("Corners Worker");

        QVMPlayerCamera camera("Video");
        camera.link(&filterWorker, "Input image");

        filterWorker.linkProperty("Output image", &cornersWorker, "Input image", QVWorker::SynchronousLink);

        QVGUI interface;

        QVImageCanvas filteredCanvas("Input");
        filteredCanvas.linkProperty(filterWorker, "Output image");

        QVImageCanvas cornersCanvas("Corners");
        cornersCanvas.linkProperty(cornersWorker, "Input image");
        cornersCanvas.linkProperty(cornersWorker,"Corners", Qt::blue, false);

        return app.exec();
        }
```

In this extensión, we have added a new worker, we have linked this worker "Input image" to the "Output image" of the filterworker in order to generate a image processed twice, the **QVWorker::SynchronousLink** parameter indicates that each filterworker iteration waits for the previous new worker's iteration. Besides we have created a new canvas and we have linked it to the new worker "Input image" (it doesn't change the image) and a new workers QList<QPointF> property, "Corners", the canvas will draw these points.

The new worker, from now 'cornersWorker', get the corners of a image with a harris estractor. The cornersWorker must reimplement the constructor and the iterator metods: in the constructor it adds three input properties, the image and two unlinked intut properties that we can change from the **QVGUI** interface, and a a output property, tne corners of the image; in the iterate it get the input properties, process the input image (using two QVision internal algorithym FilterHessianCornerResponseImage and GetMaximalResponsePoints3) and set the output property value, besides we can se some timeFlags, they mark time point to be represented in the cpu stats graphic plots (the GUI holds one of this).

This example's second part has the following structure:



Third Step: adding more functionality

Now we will add two new workers to ower example:

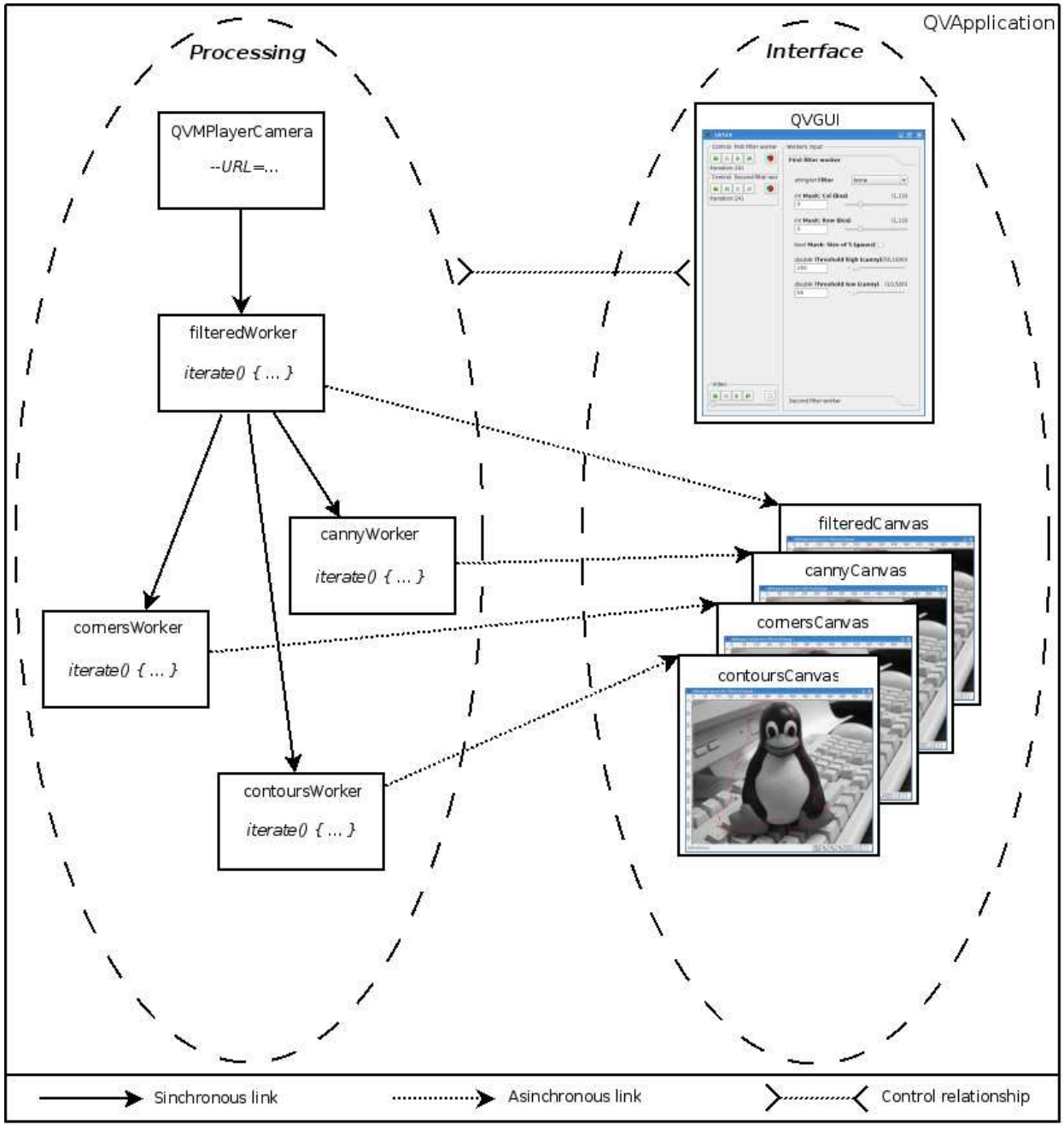
```
#include <QVApplication>
#include <QVMPlayerCamera>
#include <QVGUI>
#include <QVImageCanvas>
#include <QVPolyline>
#include <QVFilterSelectorWorker>

class CannyOperatorWorker: public QVWorker
        {
        public:
           CannyOperatorWorker(QString name): QVWorker(name)
              {
              addProperty<double>("cannyHigh", inputFlag, 150, "High threshold for Canny operator", 50, 1000);
              addProperty<double>("cannyLow", inputFlag, 50, "Low threshold for Canny operator", 10, 500);
              addProperty<bool>("applyIPE", inputFlag, TRUE, "If we want to apply the IPE algorithm");
              addProperty<double>("paramIPE", inputFlag, 5.0,
                                  "IPE parameter (max. allowed distance to line)", 1.0, 25.0);
              addProperty<bool>("intersectLines", inputFlag, TRUE,
                                "If we want IPE to postprocess polyline (intersecting lines)");
              addProperty<int>("minLengthContour", inputFlag, 25,
                               "Minimal length of a contour to be considered", 1, 150);
              addProperty<int>("showNothingCannyImage", inputFlag, 0,
                               "If we want nothing|Canny|original image to be shown",0,2);
              addProperty<bool>("showContours", inputFlag, TRUE, "If we want contours to be shown");

              addProperty< QVImage<uChar,1> >("Output image", outputFlag);
              addProperty< QVImage<uChar,3> >("Input image", inputFlag|outputFlag);
              addProperty< QList<QVPolyline> >("Output contours", outputFlag);
              }

                void iterate()
                        {
                        // 0. Read input parameters
```

```
                        const double cannyHigh = getPropertyValue<double>("cannyHigh");
                        const double cannyLow = getPropertyValue<double>("cannyLow");
                        const bool applyIPE = getPropertyValue<bool>("applyIPE");
                        const double paramIPE = getPropertyValue<double>("paramIPE");
                        const bool intersectLines = getPropertyValue<bool>("intersectLines");
                        const int minLengthContour = getPropertyValue<int>("minLengthContour");
                        const int showNothingCannyImage = getPropertyValue<int>("showNothingCannyImage");
                        const bool showContours = getPropertyValue<bool>("showContours");
                        QVImage<uChar,1> image = getPropertyValue< QVImage<uChar,3> >("Input image");
                        const uInt cols = image.getCols(), rows = image.getRows();

                        QVImage<sFloat> imageFloat(cols, rows), dX(cols, rows), dY(cols, rows), dXNeg(cols, rows);
                        QVImage<uChar> canny(cols, rows), buffer;

                        // 1. Convert image from uChar to sShort
                        Convert(image, imageFloat);
                        timeFlag("Convert image from uChar to sShort");

                        // 2. Obtain horizontal and vertical gradients from image
                        FilterSobelHorizMask(imageFloat,dY,3);
                        FilterSobelVertMask(imageFloat,dX,3);
                        MulC(dX, dXNeg, -1);
                        timeFlag("Obtain horizontal and vertical gradients from image");

                        // 3. Apply Canny operator
                        CannyGetSize(canny, buffer);
                        Canny(dXNeg, dY, canny, buffer, cannyLow,cannyHigh);
                        timeFlag("Apply Canny operator");

                        // 4. Get contours
                        const QList<QVPolyline> contourList = getLineContoursThreshold8Connectivity(canny, 128);
                        timeFlag("Get contours");

                        QList<QVPolyline> outputList;
                        foreach(QVPolyline contour,contourList)
                                {
                                if(contour.size() > minLengthContour)
                                        {
                                        if(applyIPE)
                                           {
                                           QVPolyline IPEcontour;
                                           IterativePointElimination(contour,IPEcontour,paramIPE,FALSE,intersectLines);
                                           outputList.append(IPEcontour);
                                           }
                                        else
                                                outputList.append(contour);
                                        }
                                }
                        timeFlag("IPE on contours");

                        // 5. Publish resulting data
                        if(showNothingCannyImage == 1)
                                setPropertyValue< QVImage<uChar,1> >("Output image",canny);
                        else if(showNothingCannyImage == 2)
                                setPropertyValue< QVImage<uChar,1> >("Output image",image);
                        else    {
                                QVImage<uChar> whiteImage(cols, rows);
                                Set(whiteImage,255);
                                setPropertyValue< QVImage<uChar,1> >("Output image",whiteImage);
                                }
                        if(showContours)
                                setPropertyValue< QList< QVPolyline> >("Output contours",outputList);
                        else
                                setPropertyValue< QList< QVPolyline> >("Output contours",QList<QVPolyline>());

                        timeFlag("Publish results");
                        }
        };

class ContourExtractorWorker: public QVWorker
        {
        public:
                ContourExtractorWorker(QString name): QVWorker(name)
                   {
                   addProperty<int>("Threshold", inputFlag, 128,
                                    "Threshold for a point to count as pertaining to a region", 0, 255);
                   addProperty<int>("MinAreaIPE", inputFlag, 0,
                                    "Minimal area to keep points in the IPE algorithm", 0, 50);
                   addProperty< QVImage<uChar,3> >("Input image", inputFlag|outputFlag);
                   addProperty< QList<QVPolyline> >("Internal contours", outputFlag);
                   addProperty< QList<QVPolyline> >("External contours", outputFlag);
                   }

                void iterate()
                        {
                        // 0. Read input parameters
                        QVImage<uChar,1> image = getPropertyValue< QVImage<uChar,3> >("Input image");
                        const uInt      threshold = getPropertyValue< int >("Threshold"),
                                        minAreaIPE = getPropertyValue< int >("MinAreaIPE");

                        timeFlag("Read input parameters");

                        // 1. Get contours from image
                        const QList<QVPolyline> contours = getConnectedSetBorderContoursThreshold(image, threshold);
                        timeFlag("Get contours from image");

                        // 2. Apply IPE
                        QList<QVPolyline> ipeContours;

                        foreach(QVPolyline polyline, contours)
                                {
                                QVPolyline ipePolyline;
                                IterativePointElimination(polyline, ipePolyline, minAreaIPE);
                                if (ipePolyline.size() > 0)
                                        ipeContours.append(ipePolyline);
                                }

                        timeFlag("IPE filtering");
```

```
                        // 3. Export contours to output property
                        QList<QVPolyline> internalContours, externalContours;

                        foreach(QVPolyline polyline, ipeContours)
                                if (polyline.direction)
                                        internalContours.append(polyline);
                                else
                                        externalContours.append(polyline);

                        setPropertyValue< QList< QVPolyline> >("Internal contours",internalContours);
                        setPropertyValue< QList< QVPolyline> >("External contours",externalContours);
                        timeFlag("Computed output contours");
                        }
        };

class HarrisExtractorWorker: public QVWorker
        {
        ...

int main(int argc, char *argv[])
        {
        QVApplication app(argc, argv,
                "Example program for QVision library. Obtains several features from input video frames."
                );

        QVFilterSelectorWorker<uChar, 3> filterWorker("Filter worker");
        CannyOperatorWorker cannyWorker("Canny Operator Worker");
        ContourExtractorWorker contoursWorker("Contours Extractor Worker");
        HarrisExtractorWorker cornersWorker("Corners Worker");

        QVMPlayerCamera camera("Video");
        camera.link(&filterWorker, "Input image");

        filterWorker.linkProperty("Output image", &cannyWorker, "Input image", QVWorker::SynchronousLink);
        filterWorker.linkProperty("Output image", &contoursWorker, "Input image", QVWorker::SynchronousLink);
        filterWorker.linkProperty("Output image", &cornersWorker, "Input image", QVWorker::SynchronousLink);

        QVGUI interface;

        QVImageCanvas filteredCanvas("Input");
        filteredCanvas.linkProperty(filterWorker, "Output image");

        QVImageCanvas cannyCanvas("Canny");
        cannyCanvas.linkProperty(cannyWorker,"Output image");
        cannyCanvas.linkProperty(cannyWorker,"Output contours");

        QVImageCanvas contourCanvas("Contours");
        contourCanvas.linkProperty(contoursWorker, "Input image");
        contourCanvas.linkProperty(contoursWorker,"Internal contours", Qt::red);
        contourCanvas.linkProperty(contoursWorker,"External contours", Qt::blue);

        QVImageCanvas cornersCanvas("Corners");
        cornersCanvas.linkProperty(cornersWorker, "Input image");
        cornersCanvas.linkProperty(cornersWorker,"Corners", Qt::blue, false);

        return app.exec();
        }
```

In this extensión, we have added two new workers, like the previous step. In this case we have added a contour extractor and a canny operator. Those workers have QList<QVPolyline> properties, that have been linked to two new canvas, the canvas will draw those polylines over the image.

This example's third part has the following structure:



Fourth Step: adding a plot graph

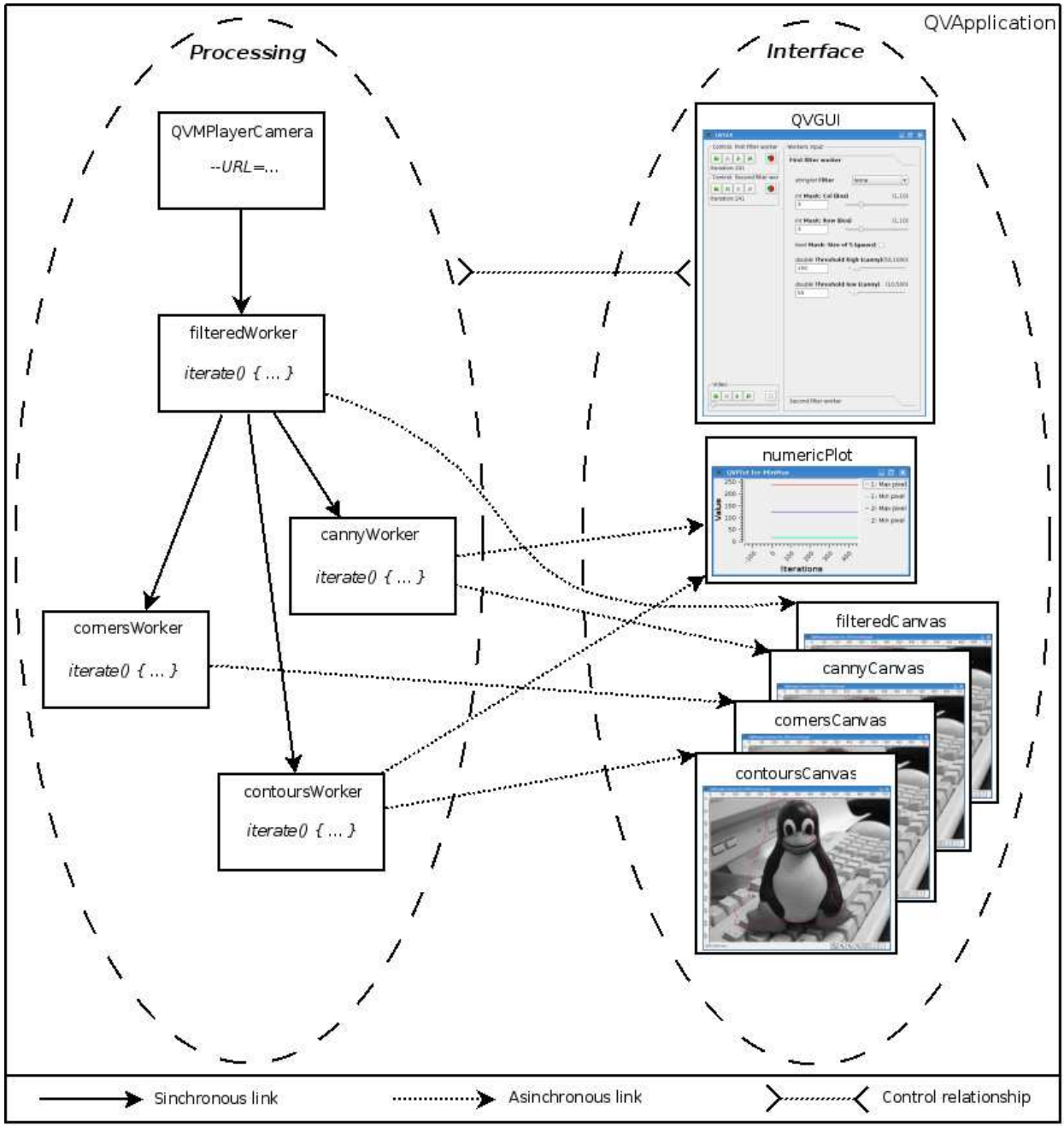
Now we will add a graphic plot over the number of contours that the two new workers generate:

```
#include <QVApplication>
#include <QVMPlayerCamera>
#include <QVGUI>
#include <QVImageCanvas>
#include <QVPolyline>
#include <QVFilterSelectorWorker>

class CannyOperatorWorker: public QVWorker
        {
        public:
           CannyOperatorWorker(QString name): QVWorker(name)
              {
              addProperty<double>("cannyHigh", inputFlag, 150, "High threshold for Canny operator", 50, 1000);
              addProperty<double>("cannyLow", inputFlag, 50, "Low threshold for Canny operator", 10, 500);
              addProperty<bool>("applyIPE", inputFlag, TRUE, "If we want to apply the IPE algorithm");
              addProperty<double>("paramIPE", inputFlag, 5.0,
                                  "IPE parameter (max. allowed distance to line)", 1.0, 25.0);
              addProperty<bool>("intersectLines", inputFlag, TRUE,
                                "If we want IPE to postprocess polyline (intersecting lines)");
              addProperty<int>("minLengthContour", inputFlag, 25,
                               "Minimal length of a contour to be considered", 1, 150);
              addProperty<int>("showNothingCannyImage", inputFlag, 0,
                               "If we want nothing|Canny|original image to be shown",0,2);
              addProperty<bool>("showContours", inputFlag, TRUE, "If we want contours to be shown");

              addProperty< QVImage<uChar,1> >("Output image", outputFlag);
              addProperty< QVImage<uChar,3> >("Input image", inputFlag|outputFlag);
              addProperty< QList<QVPolyline> >("Output contours", outputFlag);
              addProperty<int>("Num output contours", outputFlag);
              }

                void iterate()
                        {
```

```
                        ...

                        setPropertyValue<int>("Num output contours",outputList.size());
                        timeFlag("Publish results");
                        }
        };

class ContourExtractorWorker: public QVWorker
        {
        public:
                ContourExtractorWorker(QString name): QVWorker(name)
                   {
                   addProperty<int>("Threshold", inputFlag, 128,
                                    "Threshold for a point to count as pertaining to a region", 0, 255);
                   addProperty<int>("MinAreaIPE", inputFlag, 0,
                                    "Minimal area to keep points in the IPE algorithm", 0, 50);
                   addProperty< QVImage<uChar,3> >("Input image", inputFlag|outputFlag);
                   addProperty< QList<QVPolyline> >("Internal contours", outputFlag);
                   addProperty< QList<QVPolyline> >("External contours", outputFlag);
                   addProperty<int>("Num internal contours", outputFlag);
                   addProperty<int>("Num External contours", outputFlag);
                   }

                void iterate()
                        {
                        ...

                        setPropertyValue< QList< QVPolyline> >("Internal contours",internalContours);
                        setPropertyValue< QList< QVPolyline> >("External contours",externalContours);
                        setPropertyValue<int>("Num internal contours",internalContours.size());
                        setPropertyValue<int>("Num External contours",externalContours.size());
                        timeFlag("Computed output contours");
                        }
        };

class HarrisExtractorWorker: public QVWorker
        {
        ...

int main(int argc, char *argv[])
        {
        QVApplication app(argc, argv,
                "Example program for QVision library. Obtains several features from input video frames."
                );

        ...

        QVNumericPlot numericPlot("Num contours");
        numericPlot.linkProperty(cannyWorker, "Num output contours");
        numericPlot.linkProperty(contoursWorker);

        return app.exec();
        }
```

In this extensión, we have added a grafic plot, in this case a **QVNumericPlot** that shows int and double linked properties. To get those properties we have created int properties in the contoursWorker and the cannyWorker, that symbolize the number of contours they generate. And we link those properties to the **QVNumericPlot** (if don't indicate the property name they link all int and double worker's properties).

This example's fourth part has the following structure:


QVApplication
Processing
Interface
QVMPlayerCamera
--URL=...
filteredWorker
iterate() { ... }
cannyWorker
iterate() { ... }
cornersWorker
iterate() { ... }
contoursWorker
iterate() { ... }
QVGUI
numericPlot
filteredCanvas
cannyCanvas
cornersCanvas
contoursCanvas
Sinchronous link
Asinchronous link
Control relationship

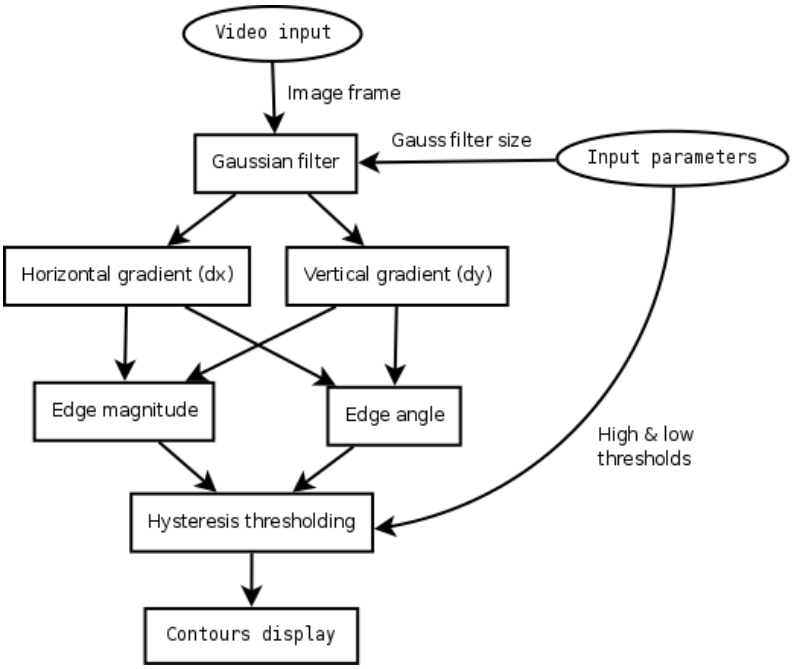
Core classes

Main classes for the programming architecture of the QVision.

QVision proposes a block-oriented programming architecture to develop Computer Vision applications and prototypes. It has some advantages regarding the application design and code parallelization. This group includes classes that are used to create the basic blocks, that will compose in new QVision applications.

QVision programming architecture

QVision is a software framework aimed to develop Computer Vision based applications and prototypes. This kind of applications usually perform intense data processing, which can be logically divided into separated stages, due to the nature of Computer Vision algorithms. A real example for the structure of a C.V. algorithm, the Canny operator, follows:



It shows the application structure implementing the algorithm, divided in several processing stages: Gaussian filtering, horizontal gradient, vertical gradient, hysteresis thresholding, etc...

From a programming point of view, each one of those stages can be easily designed and developed separatedly as independent data processing blocks. When all of them are coded and tested, they can be linked together to create the final application. This paradigm includes some programming advantages common to structured programming:

- *Coding different logical parts of the system separately allows to perform modular tests, increasing **reliability** for the final application.* It is easy to create buggy code for the previous Canny example, due to the complexity of the different processing stages it contains. Being able to develop and test the code for those stages independently is prone to decrease the quantity and relevance of them in the final application.
- *Totally independent blocks can be quite **reusable**, allowing to create new programs using working and well tested blocks from previously developed QVision applications.* As an example, gaussian filtering and image gradients are quite common in C.V. algorithms. Once coded and tested, they could be re-used to create new C.V. applications, saving time and programming effort.
- ***Knowledge sharing** in the programming community is quite direct with this approach.* New algorithms can be coded inside a block, tested, and directly shared with the community, code opened, or built in an object file along with some header files. This advantage is particulary interesting in the Computer Vision field, where image processing and segmentation algorithms can be complicated to reproduce, and a shared copy of the original algorithm developed by the researcher who invented or discovered it can be very helpful for other researchers to reproduce, evaluate and use his or her work.

QVision programming blocks

There are three main kind of programming blocks in the QVision:

- Input blocks: these are usually instances of the following classes:
  - **QVApplication**: every QVision program should create a sole instance of this class. It will keep references to the created workers, image canvases, and camera objects in the application. This object also process the user command line params, which can stablish starting input values for the parameters of the algorithms.
  - **QVGUI**: As the **QVApplication**, each QVision application should create only an instance of this class, but this is optional. When it is instantiated, it automatically creates a window where several widgets offer the user the possibility of controling different aspects of the execution of the application. For example the user can change the values of the parameters of the algorithms at execution time, stop, resume, advance the flow of the input video streams opened by the application, and control the execution of the different processing blocks. This class belongs to the group **Graphical User Interface**.
  - **QVCamera**: the instances for this class represent video sources. This is a virtual class, only contains common functionality for any camera objects that can be think of. Class **QVMPlayerCamera** should be used when creating cameras. It inherits from **QVCamera**, and offers full functionality to read image frames from several video formats and sources, like video files, digital cameras, webcams, etc... The **QVMPlayerCamera** class belongs to the group **Video input/output classes.**
- Output blocks: these blocks are leaf nodes in the graph architecture of a QVision application. They are used to plot a flow of output values, such as numeric values and images. The classes listed below can be used to plot and show graphical result output for the processing blocks. They all are included in the group **Graphical User Interface** :
  - **QVImageCanvas**: objects derived from this class can be linked to a processing block and show an output image to a window in the screen. It offers a bunch of functionality to inspect the contents of the image.
  - **QVNumericPlot**: plots the values of a numeric output from a processing block.
  - **QVHistogramPlot**: plots a list of numeric values in an histogram-like graph.
- Data processing blocks: these are objects instantiated from classes derived from the **QVWorker** class. They contain

the algorithms corresponding to the different processing blocks, which process the data in each stage and produce the output values. Besides the algorithm, they must define some input and output data variables, which can be linked to input blocks, other workers, or output blocks.

Dynamic properties

Dynamic properties are the tool used in the QVision architecture to interconnect the different programming blocks between them. A dynamic property is similar to a property contained in a class (namely stored data values), with the exception that they can be created at execution time, and they offer what is called data introspection.

Objects derived from classes **QVWorker**, **QVCamera**, **QVImageCanvas**, and in general any kind of object that may be connected in the QVision programming architecture can contain dynamic properties. They inherit from a common parent class named **QVPropertyContainer**, which contains the functionality to create new dynamic properties in an object, store and retrieve values in these dynamic properties.

Data introspection means that the list of dynamic properties contained in a property container can be obtained in execution time using a function. Reference to dynamic properties to create, delete, or update their content is always performed using a string identificator unique for the property in the object. Thus introspection of a **QVPropertyContainer** object can be performed using function **QVPropertyContainer::getPropertyList()**, which returns a list of string identifying the dynamic properties contained in the object.

This data introspection allows two important things in the QVision: first, the class **QVGUI** is a graphical interface capable of inspecting the parameters of the workers objects created in the application and offer the user a set of widgets to control them in execution time. Creating a close interface, with a design independant from the final type of the workers that could be created in the application was obtained thanks to the dynamic property introspection.

Second, it allows connecting and performing data sharing between two objects without requiring any design dependency too, so worker objects can be developed independently, and later connected using dynamic properties.

So, when linking a worker to a camera or a video output widget, actually a dynamic property from the worker and a dynamic property for the camera or the widget will be linked, impelling the QVision application to perform data sharing between them whenever a new frame is written by the camera object.

Dynamic properties are an homogeneous way of sharing data between objects, without creating design dependencies between them. No matter the real type of two objects derived from **QVPropertyContainer** class, if they have type compatible dynamic properties, they can be linked to make these property container objects to share data.

You can read more about property containers functionality in the **QVPropertyContainer** class reference page.

Parallel programming with QVision

QVision allows with the processing block division architecture to easily develop multi-threaded applications, with synchronized and thread-safe data sharing, without requiring from the programming explicit use of thread locks, semaphores, or any other typical complication of clasic parallel programming. So QVision applications can improve their performance over parallel architectures if the processing block structure is well designed, without requiring too much parallel programming expertise from the developer.

Each different procesing block is prone to be conveniently mapped onto a different thread, because different processing algorithms work over separable data, and the volume of data processed by a computer vision task is commonly high enough to dedicate a thread to process it. That is why **QVWorker** class inherits from Qt's QThread class. So, worker objects can be considered as threads, that just call their *iterate()* function time and again.

By just breaking a task into several subtasks, and programming each task in a different worker, we are doing parallel computing, because every worker will be executed in a different thread, that the operative system can map onto different cores or CPU's. Also it is convenient for workers to be executed in threads separated from the main application thread, to keep the GUI responsive when a call to a worker's *iterate()* function takes too long.

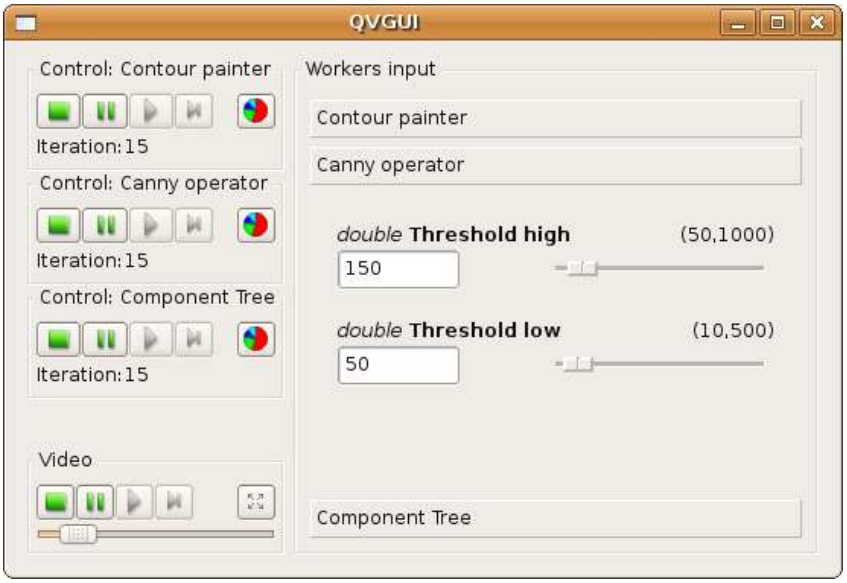
Graphical User Interface

Widget classes for graphical output of QVImage's and other data type structures, and real time parameter inspection.

QVGUI

This class can be used to create an interactive widget that will offer to the user the possibility to control the execution of the different workers registered in the application, the flow of the input video camera objects, and modify the scalar input parameters defined and not linked at the workers (see **Dynamic properties**).

This is a sample snapshot for a **QVGUI** window:



To be correctly initialized a sole instance object of the **QVGUI** class should be created after the **QVApplication** object in the *main* function, and before the call to *exec()*. For a general usage of this class in a real QVision application, see ProgrammingModel section.

QVGUI widgets.

There are two widget areas in the **QVGUI** window:

- **Control** area: contains a *camera widget* and a *worker widget* for each camera and worker object created in the application, respectively. These widgets can be used to stop, resume, and pause the execution and video input flow for the workers and the cameras of the application, amongst other things.
- **Workers input** area: contains a set of widgets allowing the user to modify the values of the parameters for the workers created by the application.

Camera widgets.

For every camera object registered in the system there is a widget in the interface, known as the camera widget, that contains buttons to control its input flow. An example of it is depicted below:

Follows a description of the buttons and their functionality:

Pause button		This button stops the camera from publishing new frames from the video source. If the camera is on real-time mode it will keep reading frames, but won't send them to the workers connected to it.
Resume button		This button resumes grabbing of the images that the registered workers read from the cameras.
Step button		When the camera is paused, this button makes it read the next frame in the video input, but keeps the camera paused.
Stop button		This button stops the camera from reading frames from the input source.
Camera extended controls button		This button opens a new window, with extended control functionality for the camera object, including buttons to control the playing speed. Also, it displays the following information: fps, reproduction speed, size of the video images, seconds elapsed (actual/total) and frames grabbed (read from the video source) / read (sent to the workers). 

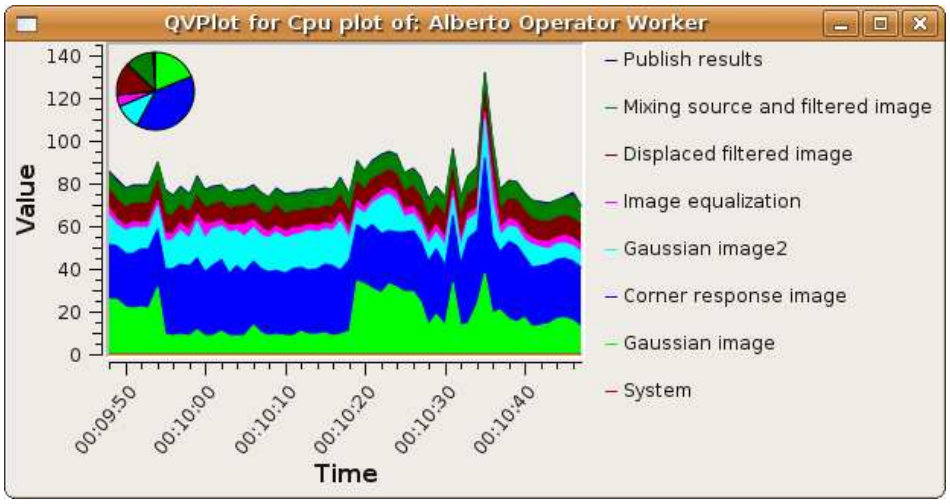
Worker widgets.

As for the cameras, every worker object registered in the system has its own widget in the interface. It contains several buttons that control its execution. An example of it is depicted below:

You can see a preview plot of CPU usage, and some buttons. Execution for the workers can be paused, resumed, stepped, and stopped with them, as if workers were cameras.

Note:
CPU stat plot depends on execution time between time flags, so when two or more workers compete for one CPU, times can differ than the used when each worker is run in its own CPU.

Follows a list with the detailed description of the buttons and their functionality:

Pause button		This button suspends the processing of the worker. Every worker synchronized with it will be stopped as well, until the resume button is pushed.
Resume button		If the worker is paused, pushing this button resumes its execution.
Step button		This button can be pushed when a worker is paused to make it process it its inputs one time. The worker will still be paused after that.
Stop button		This button finish the processing of the worker. Its properties will be unlinked and with a freezed value, so any worker connected to it will read the same values from then on.
Cpu statistics button		This button opens a detailed cpu usage plot for the worker. It shows a window with the cpu time statistical plot of the time flags defined in the *iterate()* function of the worker. You can see an example of this plot in the following figure:  Function **QVWorker::timeFlag()** can be used in the body of the **QVWorker::iterate()** function to configure a time flag in the execution of a worker.

Workers input area.

This side of the graphical user interface contains a tab for each worker created by the application. Each tab shows a set of widgets offering control for the worker's input parameters. The user can modify the values of these input parameters at will in execution time.

Each of these widgets is connected to each input property from the set of created workers

QVImageCanvas

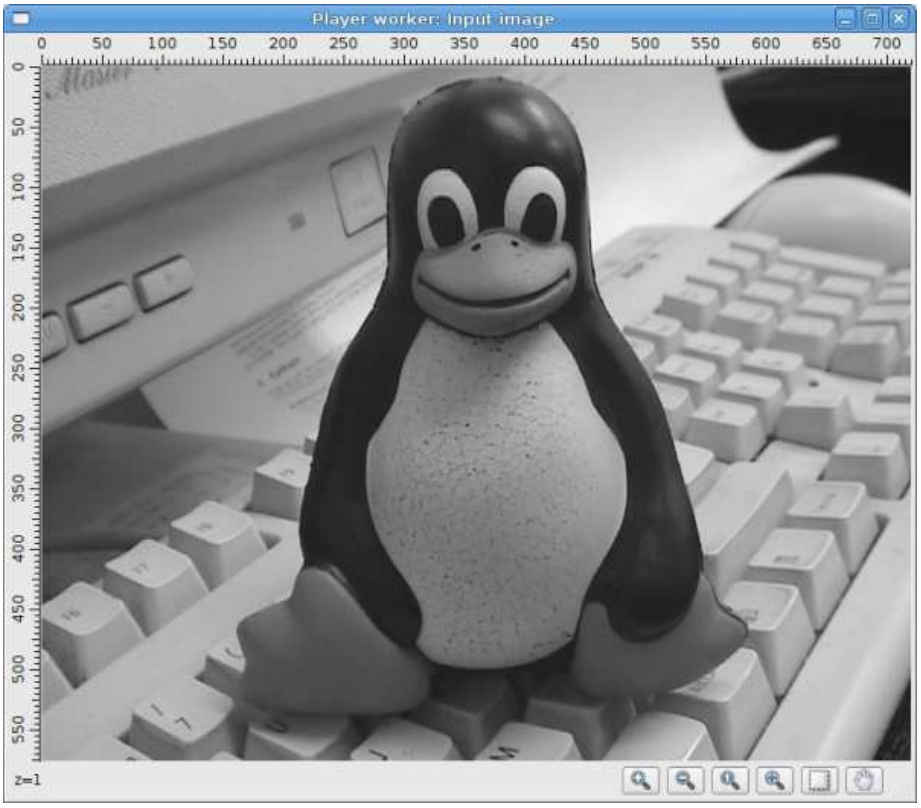
Viewer widget for **QVImage** objects and other data types.

When created an object derived from this class, the interface opens a window that depicts the content of a property of type **QVImage** contained in a property holder (normally a **QVWorker** object).

Usage is as follows: first create a **QVImageCanvas** object in the *main()* function of a program, then link a **QVImage** type property from a property holder:

```
[...]

CannyWorker cannyWorker("Canny");

[...]

QVImageCanvas imageCanvas;
imageCanvas.linkProperty(cannyWorker,"Canny image");

[...]
```

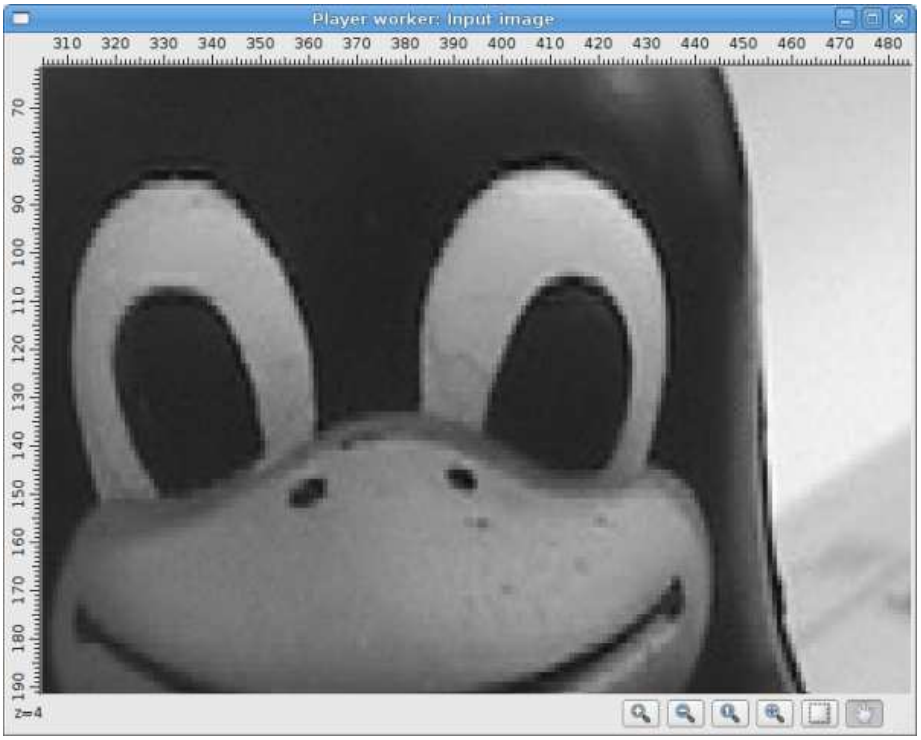
and latter call for the *exec()* function of the **QVApplication** object of the program. This will automatically create a window like the following:



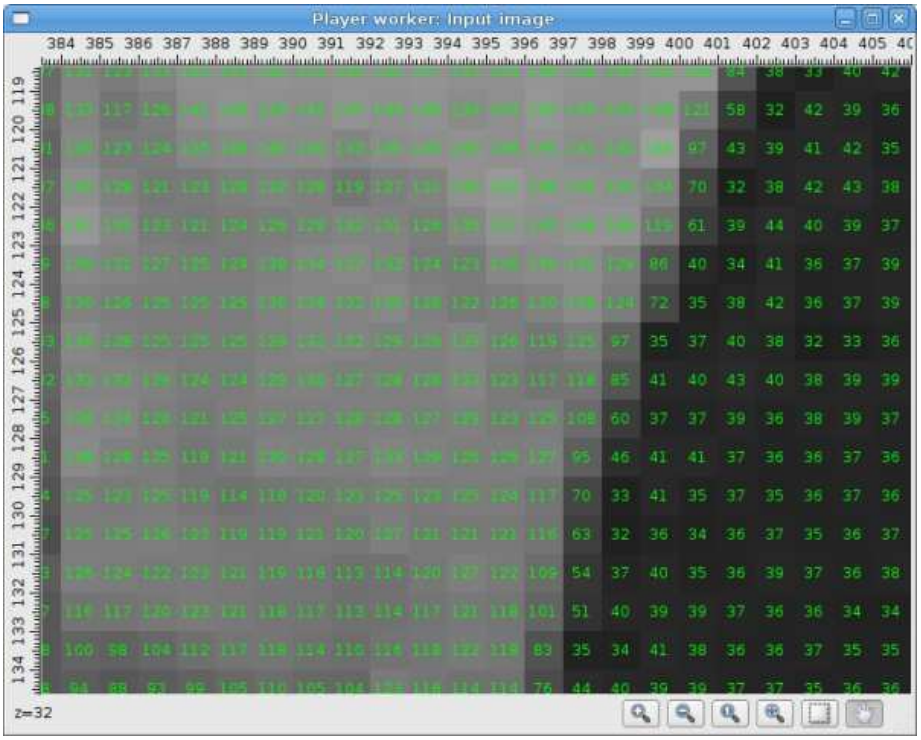
You can see it has a zoom number indicator (*z = 1*), horizontal and vertical rules (in pixels), and some buttons. These latter control zooming and moving around a zoomed area of the image mostly, and a detailed explanation of each one follows:

Zoom in button 	This button zooms the image in. Canvas window doesn't change size, but the area of the image displayed becomes smaller by a factor of 4 (width and height gets divided by 2). The label of the left bottom corner displays the text `z=<zoom>` where <*zoom*> is a number indicating the zoom factor that divides width and height in the actual zoom level.
Zoom out button 	This button divides zoom factor by 2, if it is equal or greater than 2. If the canvas window is bigger than the image at the final zoom, it is adjusted to the size of the latter.
Zoom restore button 	This button sets zoom factor by 1, and resizes canvas window to its original size.
Select zoom region button 	This button lets you select a rectangle in the image to zoom, adjusting zoom factor and canvas window size to it.
Select polyline button 	This button lets you select points in the image. That points list can be get as the "poly select" imageCanvas's property. You can explain this button by pushing it for a few seconds, and you can select a polyline representation, a point list representation, or generate a circular polyline, by a click and drag mode.
Select ROI button 	This button lets you select a rectangle in the image, that rectangle can be get as the "rect select" imageCanvas's property. And can be use to apply operations only to this rectangle.
Drag mode 	This button activates drag mode. With it you can move around the image displayed in the canvas window, if the zoom factor forces the canvas window to show only a sub-region of the image, by holding click and dragging over the depicted image.

In the following figure you can see a zoomed canvas window depicting a sub-region of the original image:



Another interesting feature of image canvas is that at a zoom factor bigger or equal to 32, the canvas renders the gray-scale pixel value over every pixel if the image is gray-scale, or the three different values for each of the RGB channels over every pixel, if the image is RGB, as depicted below:



QVCpuPlot

Class for plot graphs of worker's cpustat properties.

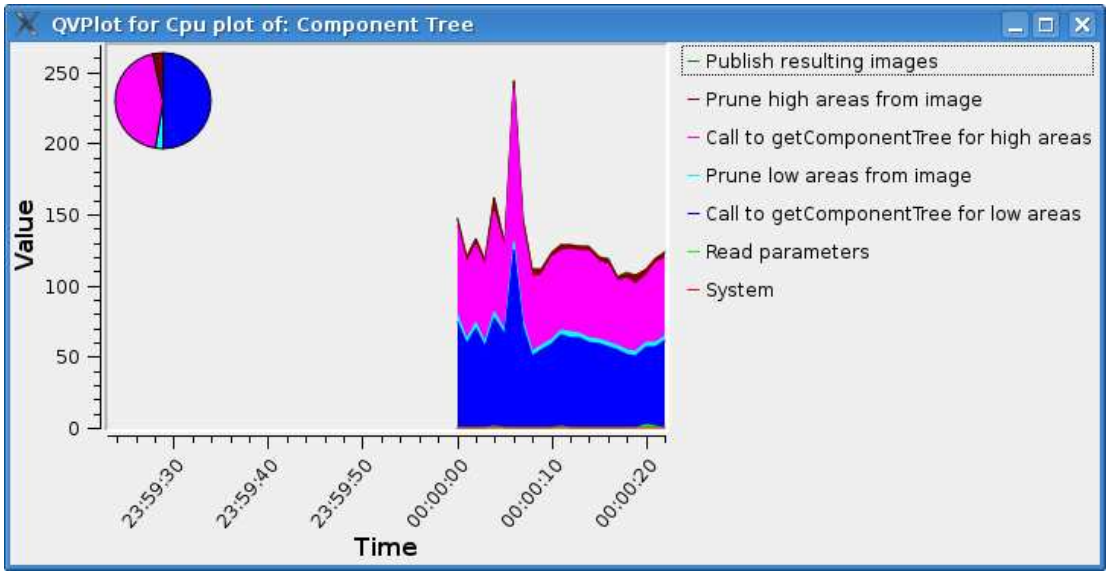
This class lets us plot graphs for QVStat worker's output properties. First, we need workers with QVStat output properties, like this:

```
class MyWorker: public QVWorker
      {
      public:
              MyWorker(QString name): QVWorker(name)
                      {
                      addProperty< QVImage<uChar,1> >("Input image", inputFlag|outputFlag);
                      addProperty< QVImage<uChar,1> >("Output image", outputFlag);
                      addProperty<QVStat>("statistics", outputFlag);

                      ...
```

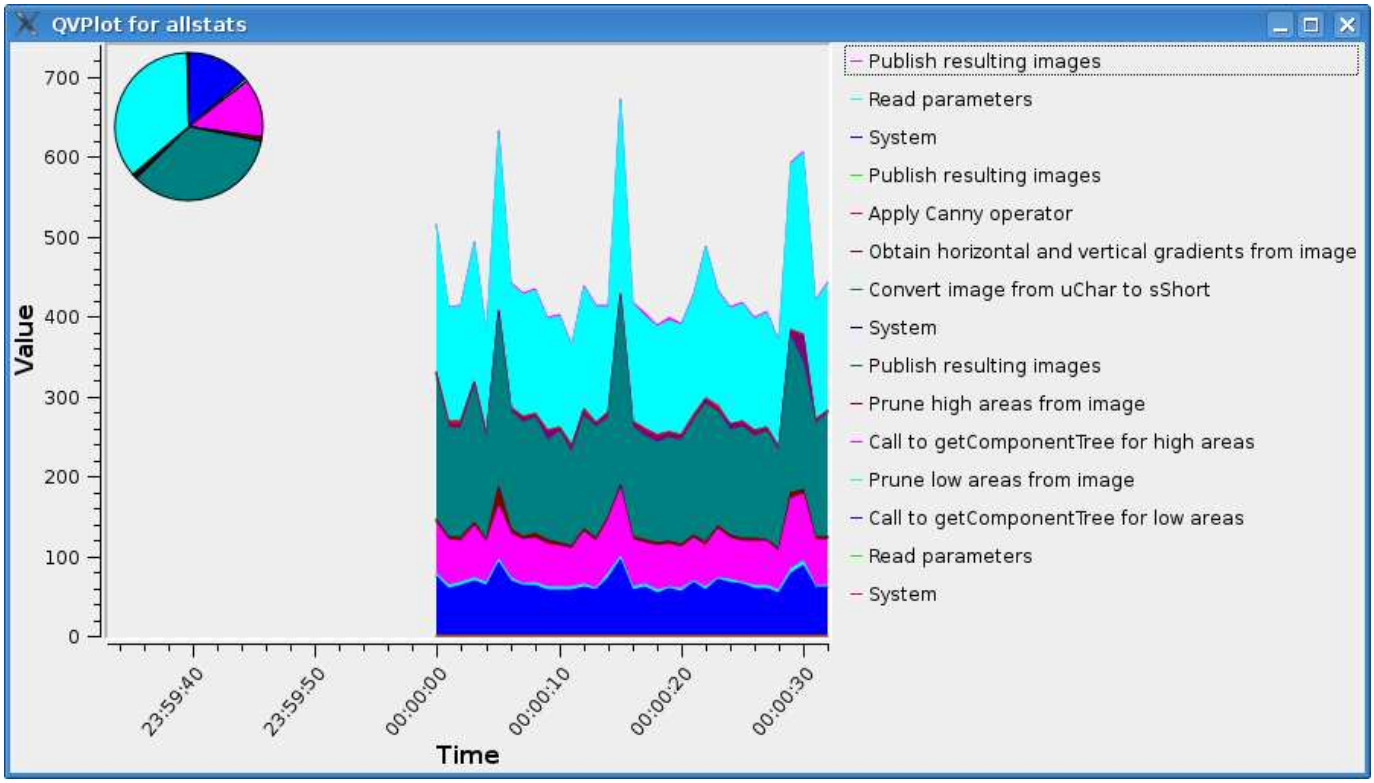
And then, in the main function, we can create a **QVCpuPlot** and link it with that property, like this:

```
...

MyWorker myWorker("worker");
QVCpuPlot cpuPlot("statistics plot");
cpuPlot.linkProperty(myWorker, "statistics");

...
```

Showing us a graph window like this:



If we link some workers QVStat properties to one **QVCpuPlot**, it plots all QVStat's time flags in link order, like this:



All workers have a QVStat output property, "cpu stats", that can be linked like this:

```
...

MyWorker myWorker("worker");
QVCpuPlot cpuPlot("statistics plot");
cpuPlot.linkProperty(myWorker);

...
```

And it's **QVCpuPlot** graph is shown when we puss the graph worker GUI button:

And plots the cpu time between each timeFlag() in the worker's iterate().

QVNumericPlot Class Reference

Class for plot graphs of worker's int and double properties.

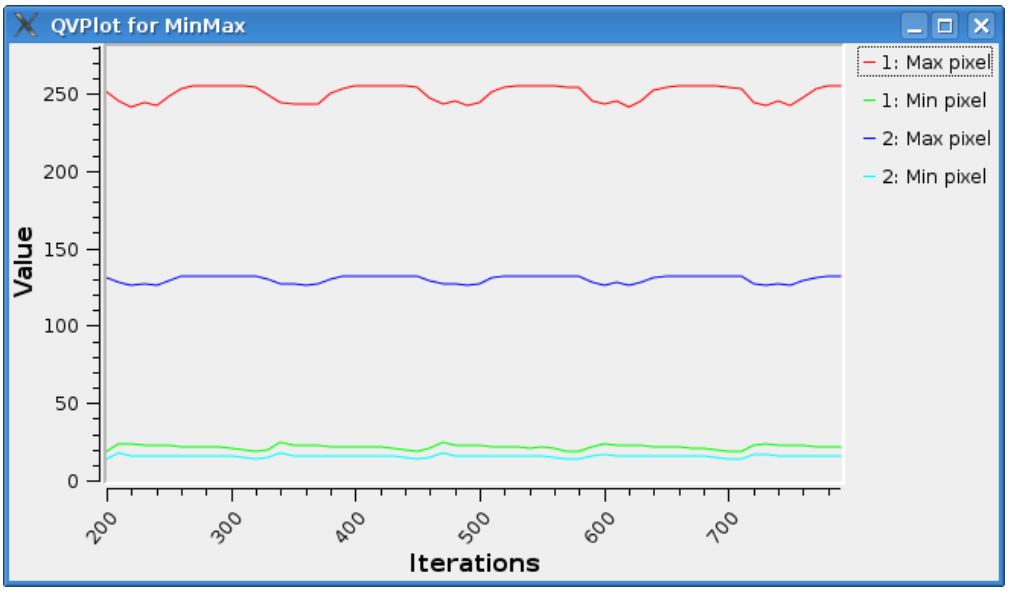
This class lets us plot graphs for int and double worker's output properties. First, we need a worker with int or double output properties, like this:

```
class MyWorker: public QVWorker
      {
      public:
              MyWorker(QString name): QVWorker(name)
                      {
                      addProperty< QVImage<uChar,1> >("Input image", inputFlag|outputFlag);
                      addProperty< QVImage<uChar,1> >("Output image", outputFlag);
                      addProperty<int>("Max pixel", outputFlag);
                      addProperty<int>("Min pixel", outputFlag);

                      ...
```

And then, in the main function, we can create a **QVNumericPlot** and link it with some of those properties, like this:

```
...

MyWorker myWorker("worker");
QVNumericPlot numericPlot("MinMax");
numericPlot.linkProperty(myWorker, "Max pixel");
numericPlot.linkProperty(myWorker, "Min pixel");

...
```

Showing us a graph window like this:



QVHistogramPlot Class Reference

Class for plot worker's histograms.

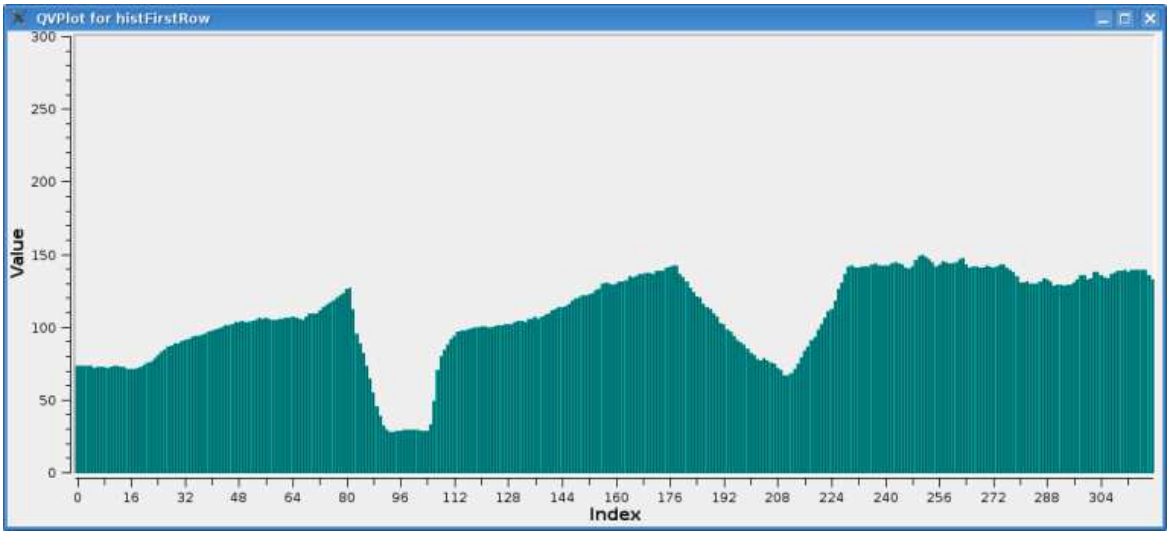
This class lets us plot graphs for a QList<double> worker's output property. First, we need a worker with a QList<double> output property, like this:

```
class MyWorker: public QVWorker
      {
      public:
              MyWorker(QString name): QVWorker(name)
                      {
                      addProperty< QVImage<uChar,1> >("Input image", inputFlag|outputFlag);
                      addProperty< QVImage<uChar,1> >("Output image", outputFlag);
                      addProperty<QList<double> >("FirstRow", outputFlag);

                      ...
```

And then, in the main function, we can create a **QVHistogramPlot** and link it with that property, like this:

```
...

MyWorker myWorker("worker");
QVHistogramPlot histPlot("histFirstRow", false, 10, 300);
histPlot.linkProperty(myWorker, "FirstRow");

...
```

Showing us a graph window like this:


QVPlot for histFirstRow
Value
Index
0
50
100
150
200
250
300
0
16
32
48
64
80
96
112
128
144
160
176
192
208
224
240
256
272
288
304

Using QVIPP library

Comprehensive set of wrapper functions for Intel(R) Integrated Performance Primitives (IPP).

The package QVIPP contains direct wrapping functions that map direct functionality from the library Intel IPP. They are intended to have very easy use, simplified naming specifications and use of **QVImage** objects instead of pointers to image data buffer, to make qvipp a simple interface to the library from Intel.

Signature should be very homogeneous, and should also be related with the name of the function it wraps from the Intel IPP. Thus, for instance the **Copy** functions are actually wrappers for the functions *ippCopy_** from the Intel library. IPP bit-depth function is related to input and output image bit-depth for every function of this package. You can check section **Image bit-depth.** to check correspondencies between **QVImage** bit-depth and the respective equivalent bit-depth type name in the IPP library.

Usage of the package

To use these functions you should include in your source the file ***qvipp.h***:

```
#include <QVIPP>
```

futhermore, these functions are inside the namespace *qvipp*, so you should either specify the use of that namespace, like in the below code:

```
using qvipp;

[...]

Copy(imagen1, imagen2);
```

or you can call all of these functions using their namespace specificator:

```
Copy(imagen1, imagen2);
```

An example of usage: Median filtering

Continuing with the example program from the section **The first program**, we will adapt it to make it apply a median filter of size 15x15 over every input image from the video, and show it on a **QVImageCanvas** window.

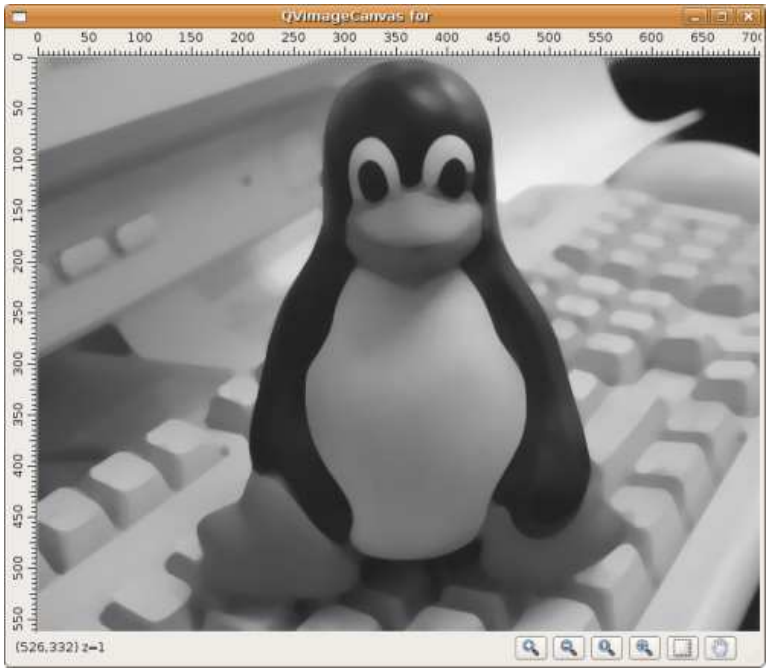
To the code of the source *example.cpp* we will add the following header include line:

```
#include <QVIPP>
```

This will include all the headers for the functions of package **IPP wrapper functions**. Then we will change the content of the function *iterate()* from the class *PlayerWorker* for the following code:

```
void iterate()
        {
        QVImage<uChar,1> inputImage = getPropertyValue< QVImage<uChar,1> >("Input image");

        QVImage<uChar,1> outputImage(inputImage.getCols(), inputImage.getRows());

        FilterMedian(inputImage, outputImage, 15, 15);

        setPropertyValue< QVImage<uChar,1> >("Output image", outputImage);
        }
```

When compiling and executing the program, it will show each frame of the video input in the image canvas, filtered with a median filter of size 15x15:



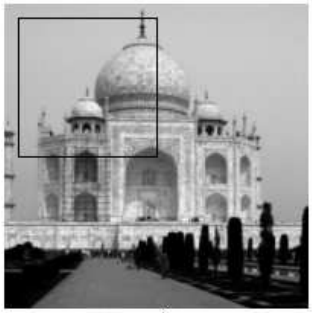
Regions of interest

The ROI of an image is stored in the class **QVGenericImage** as a QRect object. All of the functions in the **IPP wrapper**

functions package, and many functions from the qvdta package take use of the ROI value in their operations. That means they only process the area of the source images contained in the ROI. An example code for this follows:

```
[...]

QVImage<uChar> sourceImage = tajMajal;
sourceimage.setROI(10,10,90,90);        // This specifies an anchor vector for destination image.

QVImage<uChar> destinationImage(200, 200);
Copy(sourceimage, destinationImage);

[...]
```

Figure below shows the result of aplying that code to a given picture. On the left is the source image, with the specified ROI depicted as a hollow rectangle. On the right is the destination image as it is left by the code. We see that the content of the ROI area has been copied to the top left corner of the destination image:



source image

destination image

When using functions that operate over an input image and store the result of their operations on an output image of bigger size, it could be convenient to indicate that not all of the pixels in the output image contain values derived from the input image. In other cases for some image functions, mainly filtering functions such as **FilterGauss**, **FilterMedian**, even if both input and output images have the same size not all of the pixels of the resulting image have values derived from the input image.

It is denoted by *valid data* to the set of pixels in the image returned or modified by a function, that depend on the input image (or images) of the function, or were modified by that function, and are considered to contain relevant data.

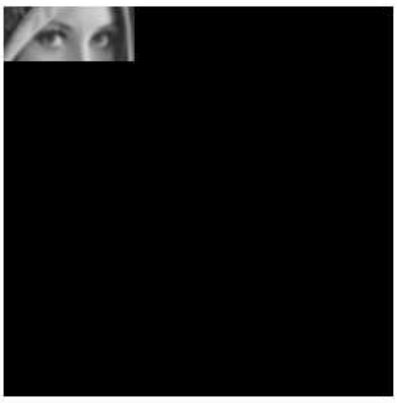
The main function of the ROI is not restricting manually the working area for image functions, but storing the area that contains valid data in the return image. Any function that returns an image should store in its ROI the rectangular area of valid data that it wrote on the image.

An example of why is useful to keep the ROI in the resulting images after operations follows:

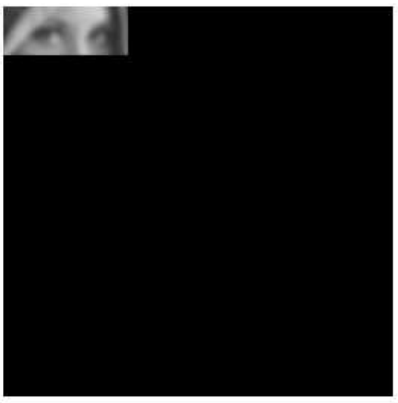
```
[...]

QVImage<uChar> sourceImage = lena;
sourceImage.setROI(100, 110, 90, 40);   // This specifies an initial ROI for source image.

QVImage<uChar> temporaryImage(256, 256);
FilterGauss(sourceImage, temporaryImage, 5);

QVImage<uChar> destinationImage(256, 256);
FilterGauss(temporaryImage, destinationImage, 5);

[...]
```

You can see the results of aplying that code to an input image in the following figure:

source image



temporary image



destination image

On the left is the source image, with the starting ROI depicted as a hollow rectangle. On the right is the destination image as it is left by the code. In the middle is depicted the temporary image. We see that the function **FilterGauss** modifes an area of the resulting image that is 4 pixels smaller in width and height than the input image, if used a mask of size 5.

Storing the ROI between calls to that function, in the intermediate images, allows to aboid specifying the valid area size and location in the image for each call to the filtering function.

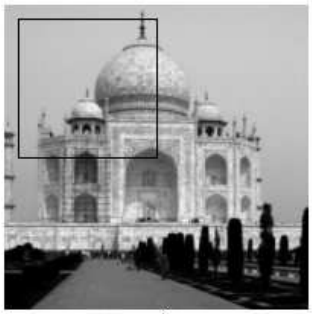
Destination ROI offset.

Many image filtering functions use an extra parameter named *destination ROI offset*. It is a pointer to the location at the

output image of these functions where they should locate the resulting ROI. The following code ilustrates this:

```
[...]
tajMajal.setROI(10,10,90,90);           // This specifies an initial ROI for source image.

QVImage<uChar> destinationImage(200, 200);
destinationImage.setAnchor(60,30);

// Third parameter specifies the location of the ROI in the destination image.
Copy(tajMajal, destinationImage, QVPoint(60,30));
[...]
```

The following picture depicts the result of using that code with the *Taj Majal* image from the previous examples:


source image


destination image

You can see that the output ROI of the destination image has the top left corner in the pixel pointed by the anchor. Not like the example in section **Regions of interest**, where the destination ROI started at the top left corner of the image. You can see that you should specify the anchor **in the output image** for the image function to make use of it.

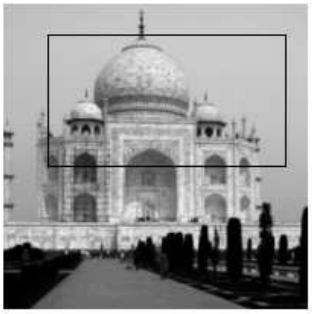
Combining use of ROI and destination offset

When using multiple image input functions, generally their ROI's must have the same size, but it can be located anywhere in those images:

```
[...]
// lena is an image of size 256x256. tajMajal is an image of size 200x200.

lena.setROI(45,65,155,85);                      // This specifies an initial ROI for image sourceImage1.
tajMajal.setROI(30,20,155,85);                  // This specifies an initial ROI for image sourceImage2.
                                                // Both ROI's have a dimention of 155x85 pixels
QVImage<uChar> destinationImage(230,230);       // destinationImage is an image of size 230x230
destinationImage.setAnchor(10,20);              // This specifies an anchor vector for image destinationImage.

Add(lena, tajMajal, destinationImage);
[...]
```

The result of that code is ilustrated in the following picture:


source image #1


source image #2


destination image

In this case function **Add** gets the contents of the ROI's of both input images and writes in the output image, starting at the pixel pointed by the anchor of the output image, the mean of every two pixels in the ROI's of the input images (add function uses by default a factor of 0.5 for the sum of the pixels).