Qt Camera Manager

Technical guide

by Thomas Dubrulle and Antonin Durey
with Tomas Holt and Grethe Sandstrak collaboration

2014 edition


HiST
TRONDHEIM


Vizlab

Table of contents

Pages 4 to 6, 11 and 15 by Thomas Dubrulle
Pages 1 to 3, 7 to 9 and 12 to 14 by Antonin Durey
Pages 9 and 10 taken from last year's documentation

I) Setup the environment

1.1) On Windows

Please remind that MinGW is NOT compatible with FlyCapture (June 2014), and as such you shouldn't use MinGW version for any libraries you need.

1.1.1) Qt

You can easily install a 32 bit version of Qt on windows. Just go on the official website, download a Visual studio version of the library, run it and follow the instructions.

As for now, Digia -the firm behind Qt- doesn't offer an installer of Qt 64 bits for Windows. Therefore, you will need to compile yourself all the Qt library and Qt Creator by using a 64-bits compiler (with the compiler included in Visual Studio express) if you want to go to the 64 bits way. It is not recommended though as it can be very hard to make it. If you can find another method like pre-compiled builds, use it instead.

In the event that FlyCapture becomes compatible with MinGW, another solution is to use a pre-compiled non-official version of Qt. You can find one of them on: http://sourceforge.net/projects/mingwbuilds/files/ in "external-binary-packages/Qt-Builds".

The latest 64-bits version should be taken and extracted in the directory of your choice. An executable file named qtbinpatcher should be located in the root of the archive. You should execute it in order to set automatically the PATH variable.

1.1.2) FlyCapture

To install the FlyCapture 2 SDK, go on this page, you will need to create an account and login: http://www.ptgrey.com/support/downloads/downloads_admin/Download.aspx. Search for the latest version of FlyCapture, download it and install it. If you have any issue or question regarding FlyCapture, you can contact the support of Point Grey research who will help you

1.1.3) Integrated Development Environment (IDE)

The project was created with Microsoft Visual Studio professionnal v2010 or MSVC 2010. You can find a trial version of it on the web. You need then to download the Service pack 1 for MSVC 2010 : http://www.microsoft.com/en-us/download/details.aspx?id=23691 . It will be used to install the Visual Studio Qt add-in, which is needed to integrate Qt in your project : http://qt-project.org/downloads .

You finally need to link all the libraries with the compiler and the linker within the project settings.

For Qt:

Open Visual Studio 2010.
Go to menu Qt5 > Qt Options and check that the Default Qt Version is the version you use
Go to menu Qt5 > Qt Project Settings and verify that Core, GUI and Widget, and OpenGL are checked.
Go to menu Project > Properties, into Configuration Properties :
- In General,
- Check if Platform Toolset is set to v100 (the version 2010 of MSVC is v100. Bigger numbers for newer versions)
- Change Character Set to Not Set
- In C++ > General, add these to Additional Include Directories value if they are not already present :
$(QTDIR)\include;
$(QTDIR)\include\QtWidgets;
$(QTDIR)\include\QtGui;
$(QTDIR)\include\QtCore;
$(QTDIR)\mkspecs\win32-msvc2010;
- In Linker > General, add this to Additional Include Directories value if it's not present :
$(QTDIR)\lib;
- In Linker > Input, add these to Additional Include Directories value if they're not present :
qtmaind.lib;
Qt5Widgets.lib;
Qt5Gui.lib;
Qt5Core.lib;

[For now, there is a issue regarding QGLWidget, the widget that allows to use OpenGL with Qt. It seems that its header cannot be found on Windows, leading to compiler errors.]

For FlyCapture:

Open Visual Studio 2010.
Go to menu Project > Properties, into Configuration Properties :
- In General,
set -In Platform Toolset is set to v100 (the version 2010 of MSVC is v100. Bigger numbers for newer versions)
Change Character Set to Not Set.
- In C++ > General, add these to Additional Include Directories :
C:\Program Files\Point Grey Research\FlyCapture2\include;
C:\Program Files x86\Point Grey Research\FlyCapture2\include;
- In Linker > General,
- set Enable Incremental Linking to No.
- set Ignore Import Library to Yes.
- set Use Library Dependency Inputs to Yes.
- add these to Additional Include Directories value :
C:\Program Files\Point Grey Research\FlyCapture2 (or wherever your FlyCapture folder is)
lib; C:\Program Files x86\Point Grey Research\FlyCapture2\lib; (same thing)
- In Linker > Input, add this to Additional Include Directories value :
FlyCapture2d_v100.lib (v100 is the platform toolset)

If you want to use Qt Creator, here are some instructions. Please mind that Qt Creator was not used for the Windows version, and thus it may need some more work to make the project work correctly:

After having installed Qt with any way, run QtCreator (its executable is located in the "bin" file). You may have to configure your own kit of development to make it work. To do this, go in the "tools/options..." select "build and run" and create a new kit in the kit panel.

A kit is composed of 3 elements: the Qt version, the compiler and the debugger.

- You can add a new Qt Version by finding its Qmake file (normally in the "bin" directory of Qt).
- The compiler is not integrated in the pre-compiled build. You may add your own by downloading it and installing it. Qt creator usually detect automatically the compiler once installed.
- As with the compiler, the debugger is not included in the pre-compiled build and need to be downloaded and installed. It is not compulsory to have a debugger to start programming, but it can be really helpful when it comes to debugging, since you can stop when you want.

When the installation is complete, it can be a good idea to create a simple program with some Qt code in it to be sure it works properly.

1.2) On Linux

Warning : the commands are the Ubuntu commands. If you have a different Linux system, check it by yourself and please update this file.

1.2.1) Ubuntu

First, you need to install the GCC and GPP compiler : *sudo apt-get install build-essential gcc* (It must be already on your computer).

Next, you need the Qt API. For this project, you will necessary need QtCreator because the graphic interface has been made with the graphic editor of QtCreator, and only QtCreator can read this kind of file. To install these, use the command *sudo apt-get install qt-sdk.*
This will normally install *QtCreator* too.

Then, you need the FlyCapture SDK. Download it at the following link (account needed to download): *http://www.ptgrey.com/support/downloads/downloads_admin/Download.aspx*
Select your camera, your model and your OS. In the results, go to software, and download the right tar.gz (32/64 bits). Decompress it, and launch **install_flycapture.sh**

Finally, you will need the OpenGL library. Some functions used in the project come from this library, and some come from the QGL library, which is an adaptation of the OpenGL library with Qt. Because Qt is already installed on your computer, the last thing to do is install OpenGL. In a terminal, simply use the command *sudo apt-get install freeglut3-dev*.

We suggest you to test to create and launch a simple C++ main at the end of the first step, and a simple Qt application at the end of the second step to be sure that these two first steps are working.

To generate the **.tex** and **.html** files, install doxygen with the command *sudo apt-get install doxygen doxygen-gui doxygen-doc*.
Run it with the command *doxywizard*. Choose the directory where the source files are. Choose the directory where to put the **.tex** and **.html** files. Go to Run tab, and Run it.

1.3) Checking

You can check if you compile in 32 or 64 bits by creating a simple program with one main, and containing the line `cout << sizeof(void *) << endl;`
When you run the program, if the printout is `8` (8 bytes * 8=64bits), you compiled in 64 bits. But if it is `4`, you have compiled in 32 bits.

II) Implementation

One class diagram can be found in the project directory. It was made with StarUML on Windows. So you should consider using it to have a better view of all classes. You can still use the .jpg version if you cannot use it.

2.1) QtCreator and main

If you reached all the configuration steps without troubles, then launch QtCreator. In the Edit section, make a right click into the project tree, and reach the file *.pro* (which must be named *ProjetNorvege.pro*), and load it. It will load all the files (header, cpp and the ui files).

The *main.cpp* file is the entrance of the software. It ill simply create a `MainWindow`.

2.2) MainWindow

2.2.1) Main presentation

The `MainWindow` class, is, at its name indicates, the main window of the software. It is divided into 4 parts :

- The menu bar : created with the `MenuBar` class
- The tool bar : created with the *mainwindow.ui*
- The left menu : created with the *mainwindow.ui*, but considerably used in `MainWindow`. Contains 2 tabs : camera and project, see next page
- The central widget : contains `QMdiSubWindow` classes : `CalibrationViewerWidget`, `ConfigFileViewerWidget`, `ImageViewerWidget`, `SocketViewerWidget`[1], and a `QMdiSubWindow` in the `activeCameraEntry` class (which is an internal class of `AbstractCameraManager`, see next page).

The *mainwindow.h* also contains 2 internal classes, which are 2 *QThreads*. They are classes to detect automatically new cameras, and update automatically the camera properties.

In the `MainWindow` class, the code is cut in 6 parts :

- the main functions : constructor, destructor
- the toolbar functions
- the menubar function : there is only a slot
- the camera tree functions : right click, double click...
- the thread functions : there are 2 functions, because there are 2 internal threads
- the project tree functions : right click, double click...

1 respectively page 12, 11, 12 and 13 in this document

2.2.2) Camera Tree

The cameras you plug will be automatically detected thanks to the `ThreadDetectCamera` class, which is a internal class of `MainWindow` (see *mainwindow.h*). In the `MainWindow`, you will find functions for the right click, the 4 actions coming from the right click (about group and name) and functions about changing the current item.

2.2.3) Project Tree

In `MainWindow`, you will find functions for the right click, loading projects (which is done recursively with `createTreeFolder()` and `createTreeItem()`), and double click which is the way to open files.

2.4) Camera implementation

2.3.1) Flycapture Implementation

The management of these cameras is divided into two class : `FlyCamera` and `FlyCameraManager`. This classes extends `AbtractCamera` and `AbstractCameraManager` to implement API of FlyCapture into the app.

For further details on the implementation of each class and its methods, please refer to the doxygen documentation.

2.3.2) FlyCameraManager

The `FlyCameraManager` class implements the `AbstractCameraManager` and contains a vector of `FlyCameras`. It defines the list of properties which are available with the FlyCapture cameras and that will be used in the `FlyCamera` object. Each property is set manually and contains a name that refer to the camera property as well as min and max values which are mostly used for the GUI since the FlyCapture Cameras automatically adapt the value if it goes out of range.

The `detectNewCameras()` method detect all connected cameras thanks to the `BusManager` which contains a list of all physical cameras connected. The `BusManager` is included in the FlyCapture API.

2.3.3) FlyCamera Parameters

Management of parameters is simple and is based on two method : one to set value to the camera, and another to get a value from camera in order to update the GUI. In case of FlyCapture cameras, we have a pointer to the current camera instance. Each property has an automatic mode and a value which is set and update at the same time as the value. The value of a `FlyCamera` property can be a integer or a float number, so a single FlyCamera property has three attributes to update with the same value in order to be sure to always set all types. When there is an update, the value is fetched from camera, and to decide which value is the good the method tests the number of decimals (set in `CameraManager`) and choose `valueA` for 0 decimals or `absValue` for 1 or more.

2.3.4) Image capture

First of all, we need to set the camera in capture mode with the `StarCapture()` method, then we need to get the image with `retrieveBuffer()` method. Once the image is stored, we must convert it in order to be able to display it in the Qt interface. As the image from FlyCapture cameras is a 8bits and we need a 32bits image, we reproduce the bits as to obtain a 32bits black and white image. When live view mode is on, `startAutoCapture()` method is called in `FlyCamera` class, which will loop and grab an image until live view is turned off, that is to say when `stopAutoCapture()` method is called.

2.3.5) Trigger mode

In the GUI, the trigger mode of the `Flycapture` cameras can be set on and off via the cameras properties block. This setting allows the user to use an external trigger to control the image capturing.

The trigger mode has 4 parameters : onOff, mode, parameter and source. To configure an external trigger, we have to set mode, parameter, source to 0, and onOff to true. When the automatic trigger is off, hitting the capture-one-image button would freeze the program because it’s waiting for the external trigger. The software trigger is therefore fired in these conditions to avoid that problem.

Warning : We (students of 2014) did not work on that part. We only read and tried to understand this part of the code. This documentation has been taken from the 2013 students' documentation. We are not responsible for any mistakes in it.

2.4) Opening files

2.4.1) ConfigViewerWidget

The structure of the option file wizard is composed of two classes, in an effort to keep the flexibility at its maximum.

The first class, called ConfigFileViewerWidget, show the window in the main area of the application. It then reads the informations contained in the option file with the second class, the “ConfigFileReader”, so that it displays them on the screen. The advantage with this structure is that unless the format of the file is significantly changed, the ConfigFileReader class shouldn't need to be changed at all, as it performs all tasks that the ConfigFileViewerWidget needs: reading one parameter with type checking or not (in the case we don't know its type) at the desired position.

On its side, the ConfigFileViewerWidget has been cut into several methods that add each one part of the window. It has been designed so that it is easy to modify the way the wizard work.

There are several methods that add a specific widget to edit a certain type of parameters. They all follow the same pattern: they create the widget and add it to the wizard layout with the specified parameter at the given position. They all returns the newly created widgets so that they can be used elsewhere.

if you want to add another type of parameter, only the createWizard() and saveWizard() methods need to be changed by adding a new condition in it. You may also add a new method to add the new widget to the wizard.

To save the wizard into the QtextField, the saveWizard() method use lists of parameter indexes along with their corresponding widget. It then loop on all of them, and replace the old value with the one contained in the widget.

You can notice there are two classes in the ConfigFileViewerWidget: the ConfigFileViewerWidget itself and the pathEditBox. The PathEditBox is simply a field with a button that allows the edition of paths with an explorer, which is more convenient than writing it yourself. If you want, you can move it to another file and use it for other plug-ins, and add an include in the configFileViewerWidget, since the two aren't normally intertwined together.

2.4.2) ImageViewerWidget

The `ImageViewerWidget` class is the class to open the **grupper images**. There is 5 functions inside :

- the constructor,
- `initializingImage()` : initialize the image with the time number provided in parameter,
- `initializingPoints()` : initialize the points according to the number time (kept in mind as attribute in the class),
- `mousePressEvent()` : click on the image,
- `wheelEvent()` : using the mouse wheel.

2.4.3) CalibrationViewerWidget

The `CalibrationViewerWidget` class is the class to open and read the **calibration_summary** file. We can cut the functions in 4 parts :

- the constructor and initializing functions,
- the changing view functions : `showTextView()` and `showTableView()`,
- the right click function and assimilated,
- the left click function and assimilated.

`showTextView()` put a `TextEdit` as main widget. This widget is in fact a `CalibrationEdit`, which is an internal class of `CalibrationViewerWidget`. It has been reimplemented to be able to use `mousePressEvent()`.

The left click functions are only used in the text view. In its assimilated functions, you will find functions as `select()`, `calculateShowFailed()`, `calculateShowUseless()`, `calculUselessCombinations()`, and the two functions used to make the combo sort : `sortCombo()` and `executeSortChange()`.

These functions are sometimes really difficult to understand. In the calculation functions, keep in mind the `enum type Calibration` is crucial to keep in mind which is the state of each combination. In the functions which display lines, or not, the best thing is the function `moveCursor(QTextCursor::MoveOperation, QTextCursor::MoveMode)` to select some text, `removeSelectedText()` to remove text, and `insertText()` to, of course, insert text.

We do not advise you to remove some lines and test the project without these. At best, the project will compile, launch, but make some things strange you will not understand, particularly because there are many loops and the operation are made several times. Maybe removing the loops could be a better idea to test, it depends of your feelings...

Many comment lines have been put to try to help you at best to understand how we thought it and how we made it.

2.4.4) SocketViewerWidget

The `SocketViewerWidget` class is the class to open and read the **socket** file, which is the file which contains the 3D datas. We can cut the functions in 4 parts :

- the constructor and initializing functions,
- the changing view functions : `showTextView()`, `showTableView()` and `show3DView()`,
- the right click function and assimilated,
- the assimilated functions for the table view : `areaBarsMoved()`, `displayToolTip()`, `valueChanged()` and `getTimeSlider()`.

There are only assimilated functions for the table view, because the text view does nothing, and the functions for the 3D view are in the `WidgetGL` class (see next page).

The widgets containing the values are `CoordinatesLabel`, which is an internal class extending `QLabel`. It has been reimplemented to be able to use the `mouseMoveEvent()` function, which allow the `QToolTip` displaying.

2.5) WidgetGL

The `WidgetGL` class is the class which inherits from `QGLWidget`, which is the **QtGL** component to use **QtGl** and **OpenGL** functions inside. There are 5 functions in this class:

- the constructor
- `initializeGL()`, initializing function coming from `QGLWidget`
- `initializingCameraCoordinates()`, function to initialize the cameras coordinates.
- `paintGL()`, which is the function where you put all your painting desired.
- `resizeGL()`, function to resize cleanly 3D painting area according to the widget size
- `showView()`, show the view according to the time provided as parameter
- `eventFiltrer()`, check all the events : right and left click, key pressed and released, and wheel event.
- `setXRotation()`, `setYRotation()`, `setZRotation()` : setting the new rotation angle.

This class may also contains code which is not working. This is because we have not enough time to finish it :

- the right click : it should provided default view angle, so that the user would not have to deal with X, Y, Z key, and wheel mouse
- the camera points are well initialized, but not drawn every time. Seem to have a problem, maybe about depth...
- the left click : clicking on a point to have its number and coordinate. Currently, the conversion from 3D coordinates to 2D is not working.

III) Miscallenous things

Tips:

- **You may find in the files several comments named "//TTODO"**. They tell improvements or bug fixes we didn't have the time to do. Usually they have some explanations in them, as well as some ideas on how to do them. I hope they will be useful if you want to improve even more the application :).

- **You can find most if not all of the methods you need in Qt**, which have similarities with Java standard library. Using them instead of system dependant methods ensure that you keep your project multi-platform.

- **The installation of the project (not the software themselves) on Windows is more complex than on Linux**, but it may be useful to have someone working on Windows while the other is on an Unix system to constantly check the compatibility of the project on these two platforms.

- **On Visual Studio, if you get an LNK error (generally at the beginning when you install the project), it is often because the linker doesn't find a method, a class or a library in all the files you have included**. Go to your project settings and check within General > linker the include directories you have added.