

# ViSP 2.8.0: Visual Servoing Platform

## Getting Started for Unix platforms

---

Lagadic project  
<http://www.irisa.fr/lagadic>

July 24, 2013

Manikandan Bakthavatchalam  
François Chaumette  
Eric Marchand  
Nicolas Melchior  
Fabien Spindler



## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Where and how downloading ViSP</b>	<b>4</b>
<b>3</b>	<b>How to build ViSP under a Unix system</b>	<b>5</b>
3.1	Build ViSP thanks to a classical makefile . . . . .	6
3.2	Build ViSP thanks to Eclipse . . . . .	6
3.3	Build ViSP thanks to KDevelop . . . . .	7
3.4	Build ViSP thanks to XCode . . . . .	7
<b>4</b>	<b>How to use ViSP as a third party library under a Unix system</b>	<b>8</b>
4.1	How to create a HelloWorld project using ViSP and CMake . . . . .	8
4.2	How to create a HelloWorld project using ViSP with a classical Makefile . . . . .	10
4.3	How to create a HelloWorld project using ViSP with GNU Autotools . . . . .	10
<b>5</b>	<b>Additional information</b>	<b>11</b>
5.1	Third party libraries used by ViSP . . . . .	11
5.2	How to execute ViSP examples . . . . .	13

This getting started for Unix platforms is for those who want to use ViSP under a Unix system like Linux or Mac OSX and who do not intend to participate in the development of the library. Its goal is to help them to start writing a program using ViSP as a third party library without going into details. It does not replace the ViSP source code documentation available on the website:

<http://www.irisa.fr/lagadic/visp>

Thereafter, the different steps between the download and the use of the library will be described.

## 1 Introduction

Before to download ViSP and try to build it, it is advised to install a C/C++ compiler (typically GNU g++) and the last version of CMake which can be found at the address : <http://www.cmake.org>. If you want it is possible to use an IDE like Eclipse, KDevelop, XCode or QtCreator.

As described in section 5.1, some ViSP capabilities require third party libraries to be installed. But they are not required to build ViSP. Do not worry, if you use a function which requires another library you do not have installed yet, you will be warned during the execution of ViSP examples.

## 2 Where and how downloading ViSP

First you have to know that there are two ways to download ViSP source code. The simplest one consists in downloading the tar.gz file which can be found at the address:

<http://www.irisa.fr/lagadic/visp/download.html>

It contains a release version of the source code. Uncompress the tarball in the folder of your choice. To do it you can use a terminal and when you are positioned in the desired folder use the command: `tar -zxvf ViSP-2.8.0.tar.gz`. After finishing this step, go directly to Section 3.

The other way to get ViSP is to download it from Subversion repository hosted on InriaGForge <http://gforge.inria.fr/projects/visp/>. Subversion is a tool for a team of developers which enable to manage the source code during the development process. The advantage is that you can have the current development version of the code. The drawback is that it is not necessary stable and the last functions could be not documented yet. Prior to download something from Subversion repository you have to install a Subversion client (more details on <http://subversion.tigris.org/>). Then you can use the following address to get ViSP by checking out the source code files:

`svn://scm.gforge.inria.fr/svn/visp/trunk/ViSP`

Regardless the method you used to download ViSP, you have now a version of the source code which must be build to be used.

### 3 How to build ViSP under a Unix system

Now, the step consists in preparing the build by creating a classical makefile or a project depending on your compiler and/or your IDE. It will be done thanks to CMake.

1. Open a terminal and go to the directory where you want to build ViSP. This directory, for example `/local/ViSP/ViSP-build`, will be called `VISP_BUILD_DIR` later in the document. Create it if it does not exist yet.

An advice is to have two different folders for the source code and the build version. There are two advantages to do this. Firstly, the folder which contains the source code will not be contaminated by files created by CMake. So, if you want to modify ViSP it will be easier. Secondly, it allows to have more than one build version of ViSP. Indeed, there are numerous possibilities to build it depending on the third party libraries you are using. So you could use the version which matches the best to your own project.

2. Use the command : `ccmake [options] <path-to-visp-source>`. You need to use options if you want to use an IDE. It will enable you to create a project corresponding to this IDE. In this case you need to replace `[options]` by: `-G "Eclipse CDT4 - Unix Makefiles"` if you are using Eclipse, `-G KDevelop3` if you are using KDevelop 3 or `-G Xcode` if you are using XCode. If you use no option a classical makefile will be create. Thus for example if you have the ViSP source code in the folder `/local/ViSP/ViSP-code`, you will use the command: `ccmake [options] /local/ViSP/ViSP-code`. You can access more information about CMake with the command: `ccmake --help`. As presented Figure 1, CMake GUI is executed in a terminal.

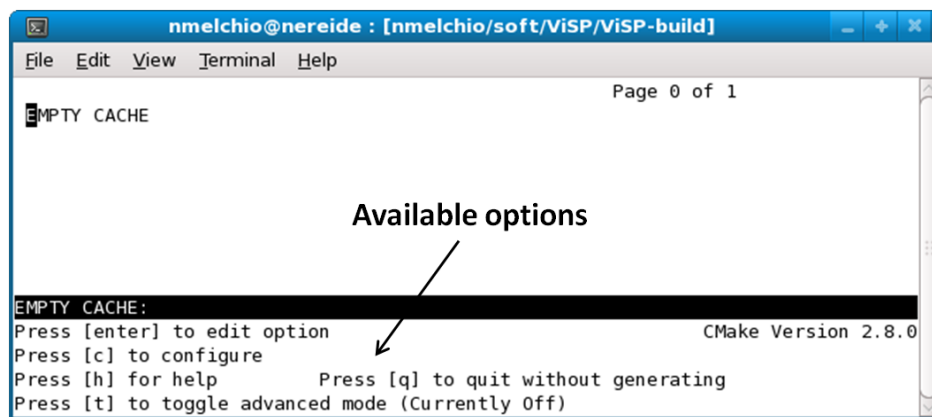


Figure 1: CMake GUI obtained with CMake 2.8.0 that allows to configure ViSP on your computer.

3. Press the "c" key on your keyboard (see also Fig. 1) to configure the build. Continue to press the "c" key until "Press [g] for generate" appears. During this step you are allowed to access to the whole options while pressing the "t" key and to modify those you want. Figure 2 shows for example how to print advanced options that can be modified like the `CMAKE_INSTALL_PREFIX` variable used to specify where ViSP will be installed.
4. After pressing the "c" key, you will be allowed to press the "g" key. CMake will generate the useful configuration files in the current folder. After finishing, the window is closed automatically. Now you are allowed to build ViSP binaries corresponding to the library and the examples.

```

nmelchio@nereide : [nmelchio/soft/ViSP/ViSP-build]
File Edit View Terminal Help
Page 3 of 14
List of Options
CMAKE_EXE_LINKER_FLAGS
CMAKE_EXE_LINKER_FLAGS_DEBUG
CMAKE_EXE_LINKER_FLAGS_MINSIZE
CMAKE_EXE_LINKER_FLAGS_RELEASE
CMAKE_EXE_LINKER_FLAGS_RELWITH
CMAKE_INSTALL_PREFIX
CMAKE_LINKER
CMAKE_MAKE_PROGRAM
CMAKE_MODULE_LINKER_FLAGS
CMAKE_MODULE_LINKER_FLAGS_DEBU
CMAKE_MODULE_LINKER_FLAGS_MINS
CMAKE_MODULE_LINKER_FLAGS_RELE
CMAKE_MODULE_LINKER_FLAGS_RELW
CMAKE_NM
CMAKE_OBJCOPY
CMAKE_OBJDUMP
CMAKE_RANLIB
Change the install prefix
/usr/local
/usr/bin/ld
/usr/bin/gmake
/usr/bin/nm
/usr/bin/objcopy
/usr/bin/objdump
/usr/bin/ranlib
CMAKE_EXE_LINKER_FLAGS_DEBUG: Flags used by the linker during debug builds.
Press [enter] to edit option
Press [c] to configure
Press [h] for help
Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently On)
CMake Version 2.8.0

```

Figure 2: This CMake snapshot is obtained after pressing "t" key to print advanced options. It shows where to set the `CMAKE_INSTALL_PREFIX` variable specifying the directory where ViSP will be installed after the build. By default, this variable is set to `/usr/local`.

### 3.1 Build ViSP thanks to a classical makefile

Build ViSP thanks to a classical makefile is very easy. Open a terminal and go to the `VISP_BUILD_DIR` folder where CMake created the makefiles. Then use the command `make`. Thus the library will be created and the examples will be compiled. The binaries will be contained in the current folder (the one which has been created during the step 1, Section 3). You can also use the command `make install` which will install the ViSP headers and library in the path corresponding to the `CMAKE_INSTALL_PREFIX` (by default set to `/usr/local`). You are allowed to modify this path during the step 3, Section 3.

It is also possible to build the html documentation. The only one condition is that Doxygen and the "dot" tool provided by Graphviz must have been detected by CMake. If it is the case, open a terminal and go to the `VISP_BUILD_DIR` folder where CMake created the makefiles. Then use the command `make html-doc`. The documentation entry point is then `VISP_BUILD_DIR/doc/html/index.html`.

### 3.2 Build ViSP thanks to Eclipse

Build ViSP thanks to Eclipse is easy. It requires that you used the option `"-G "Eclipse CDT4 - Unix Makefiles"` when executing CMake. If you did it correctly, open Eclipse. Then clicks on `file→Import...` A window appears. Select `General→Existing Projects into Workspace` and click on the button `Next`. Here you have to select the folder where the project files were created by CMake (ie your `VISP_BUILD_DIR`). Then click on `Finish` to import the project. You still have to build the project. For that, select it in the `Project Explorer` and then click on `Project→Build Project`.

It is possible to install the ViSP headers and library in the path corresponding to the `CMAKE_INSTALL_PREFIX` (by default set to `/usr/local` but it can be changed during the step 3, Section 3). For that, select the project in the `Project Explorer` and then click on `Project→Make Target→Build...` A window appears with a list of targets. Select `install` and click on `build`].

It is also possible to build the html documentation. The only one condition is that Doxygen and the "dot" tool provided by Graphviz must have been detected by CMake. If it is the case, select the project in the Project Explorer and then click on Project→Make Target→Build... A window appears with a list of targets. Select html-doc and click on build]. The documentation entry point is then `VISP_BUILD_DIR/doc/html/index.html`.

### 3.3 Build ViSP thanks to KDevelop

Build ViSP thanks to KDevelop is also very easy. It requires that you used the option "-G KDevelop3" when launching CMake. If you did it correctly you have to open the KDevelop project which has been created by CMake and build it (KDevelop Build→Build Target→all menu). Thus the library will be created and the examples will be compiled. The binaries will be contained in the current folder. You can also build the target Build→Build Target→install which will install the ViSP headers and library in the path corresponding to the `CMAKE_INSTALL_PREFIX` (by default set to `/usr/local`). You are allowed to modify this path during the step 3, Section 3.

It is also possible to build the html documentation. The only one condition is that Doxygen and the "dot" tool provided by Graphviz must have been detected by CMake. If it is the case, build the target Build→Build Target→html-doc. The documentation entry point is then `VISP_BUILD_DIR/doc/html/index.html`.

### 3.4 Build ViSP thanks to XCode

Build ViSP thanks to XCode is also very easy. It requires that you used the option "-G XCode" when launching CMake. If you did it correctly you have to open the XCode project which has been created by CMake. Then build the target called `ALL_BUILD` (see Fig. 3). The binaries will be contained in the folder you created in step 1, Section 3. If you want to install ViSP you can also build the target named `install`. It will install the ViSP headers and library in the path corresponding to the `CMAKE_INSTALL_PREFIX` (by default set to `/usr/local`). You are allowed to modify this path during the step 3.

It is also possible to build the html documentation. The only one condition is that Doxygen and the "dot" tool provided by Graphviz must have been detected by CMake. If it is the case, open the XCode project which has been created by CMake. Then build the target called `html-doc`. By default, the documentation entry point is then `VISP_BUILD_DIR/doc/html/index.html`.

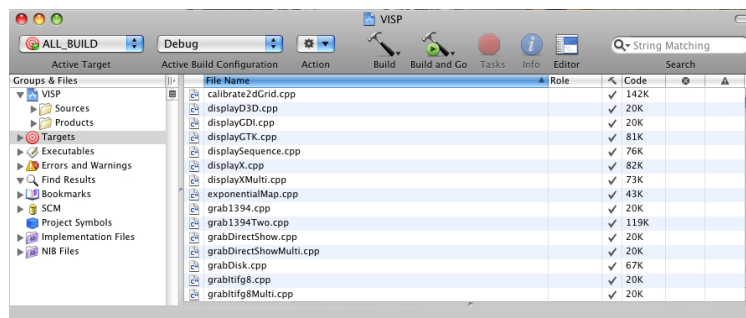


Figure 3: This XCode snapshot shows which button you have to use and which target textbox you have to set in order to build ViSP.

## 4 How to use ViSP as a third party library under a Unix system

### 4.1 How to create a HelloWorld project using ViSP and CMake

In this section you will learn how to create a HelloWorld project using ViSP as a third party library. This step is very simple if you still use CMake to configure your project. As for building ViSP, you will be allowed to use a classical makefile or a project if you are using an IDE.

1. First you have to create a folder where you want to put the HelloWorld project.
2. Then create inside this folder the `HelloWorld.cpp` file you want to build and a text file called `CMakeLists.txt` that corresponds to the HelloWorld configuration file that will be used by CMake. The following simple example shows you how to fill in these files <sup>1</sup>.

HelloWorld.cpp :

```

1 #include <iostream>
2
3 #include <visp/vpDebug.h>
4 #include <visp/vpImage.h>
5 #include <visp/vpImageIo.h>
6
7 int main()
8 {
9     std::cout << "ViSP Hello World example" <<std::endl;
10
11     vpImage<unsigned char> I(288, 384);
12
13     I = 128;
14
15     std::cout << "ViSP creates \"/>

```

CMakeLists.txt :

```

1 PROJECT(HelloWorld)
2
3 CMAKE_MINIMUM_REQUIRED(VERSION 2.6)
4
5 FIND_PACKAGE(VISP REQUIRED)
6 IF(VISP_FOUND)
7     INCLUDE(${VISP_USE_FILE})
8 ENDIF(VISP_FOUND)
9
10 ADD_EXECUTABLE(HelloWorld HelloWorld.cpp)

```

3. Then, open a terminal, position it to the folder you want to contain the project (the one you created in step 1, Section 4.1) and start CMake using the command `ccmake [options] <path-to-helloworld-source>`. The options are the same as the one presented Section 3, step 2. The path to the source code relates to your HelloWorld source code.

<sup>1</sup>HelloWorld.cpp and CMakeLists.txt files are available in ViSP source tree in `example/manual/hello-world/CMake` directory.



4. You can now press the "c" key on your keyboard (see Fig. 4).

If CMake says that it can't find the ViSP library (`VISP_DIR` variable is than set to `VISP_DIR-NOTFOUND`, see Fig. 4), you may indicate the path to the folder containing a build version of ViSP. More precisely, you have to give the path to the `ViSPConfig.cmake` file. Typically, you can find it in `VISP_BUILD_DIR` (replace `VISP_BUILD_DIR` with the path to the folder where you build ViSP, see Section 3, step 1). If you install ViSP (see Section 3, step 4), you can also set `VISP_DIR` to the following path `/usr/local/VISP`.

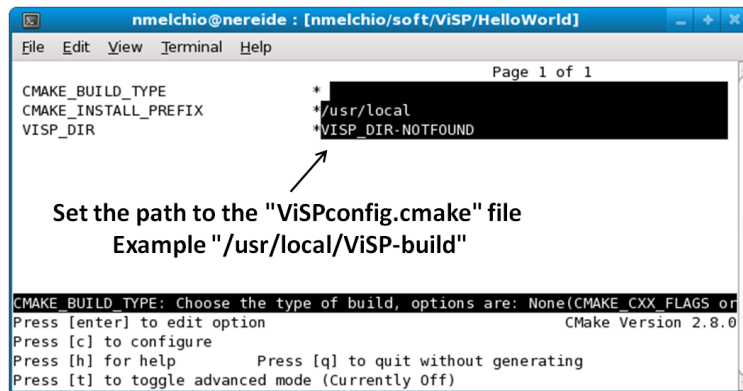


Figure 4: CMake based configuration of the HelloWorld project that uses ViSP as a third party library. This snapshot shows where to set the location of the ViSP library that will be used as a third party project.

5. press the "c" key until having the right to press the "g" key (see Fig. 5). It is the same as for ViSP. CMake will create a classical makefile or a project depending on the options you used when you started CMake. You only need now to build your project.

The advantage to use CMake is that all the links are automatically done and especially with the third party libraries on which ViSP is depending.

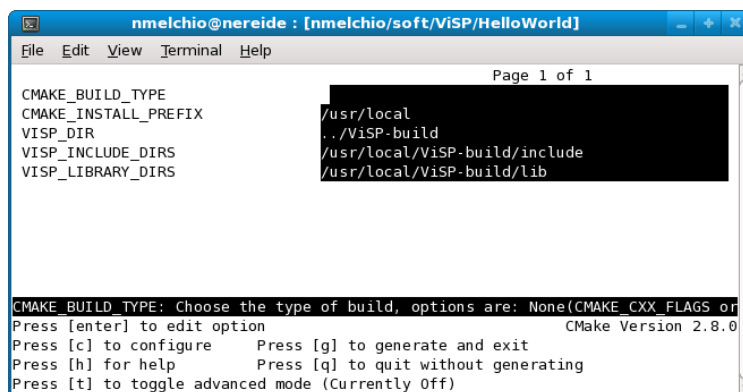


Figure 5: CMake based configuration of the HelloWorld project that uses ViSP as a third party library. You are now allowed to press the "g" key in order to generate the build material (makefiles or IDE project files).

## 4.2 How to create a HelloWorld project using ViSP with a classical Makefile

It is also possible to develop your project using ViSP as a third party library without the help of CMake. It could be indeed interesting if you are already developing your own program using a classical makefile. In order to help users to integrate the ViSP library, ViSP provides a `visp-config` shell script which can be called in the makefile. Its goal is to set automatically the links to the ViSP library and the libraries used as third party libraries during the building of ViSP. Of course it sets the include directories corresponding to these libraries too. This file is produced during the ViSP configuration (see Section 3, step 3) and is located in `VISP_BUILD_DIR/bin` directory. If you install ViSP (see Section 3, step 4), you will also find this file in `/usr/local/bin`. If you need more help about the `visp-config` shell script, you can use the command `visp-config --help` in a terminal. The following steps explain how to create a HelloWorld project using a classical makefile without the help of CMake.

1. First you have to create a folder where you want to put the HelloWorld project.
2. Then, create inside this folder the `HelloWorld.cpp` file you want to build. You can use the same `cpp` file as the one presented in the Section 4.1. You have also to create the `Makefile`. The following simple example shows how to fill in the `Makefile` file for the GNU `g++` compiler <sup>2</sup>.

Makefile :

```

1 CXX          = g++
2 VISP_BUILD_DIR = /local/ViSP/ViSP-build
3 VISP_CFLAGS   = '$(VISP_BUILD_DIR)/bin/visp-config --cflags'
4 VISP_LDFLAGS  = '$(VISP_BUILD_DIR)/bin/visp-config --libs'
5
6 HelloWorld: HelloWorld.cpp
7   $(CXX) $(VISP_CFLAGS) -o HelloWorld HelloWorld.cpp $(VISP_LDFLAGS)

```

The variable `VISP_BUILD_DIR` has to be set to the folder containing the build version of ViSP .

3. Finally, just use the command `make` to build the project.

## 4.3 How to create a HelloWorld project using ViSP with GNU Autotools

It is also possible to develop your project using ViSP as a third party library using GNU Autotools (`aclocal`, `autoconf`, `automake`). They will create a classical makefile. The following steps explain how to create a HelloWorld project using Autotools.

1. First you have to create a folder where you want to put the HelloWorld project.
2. Then, you have to create four different files. The first one is `configure.ac` which is requested by `autoconf` to generate the `configure` shell script. The second one is `Makefile.am` which is used by `automake` and is useful to produce the makefile material. The third one is the M4 macro file `have_visp.m4` where `HAVE_VISP_IFELSE` macro is defined. And finally the fourth file is the `HelloWorld.cpp` source code. You can use the same `cpp` file as the one presented in the Section 4.1. The following simple example shows how to fill in the `.ac` and `.am` files. The `.m4` file can be found in the `macros` directory coming with ViSP source code <sup>3</sup>.

<sup>2</sup>`HelloWorld.cpp` and `Makefile` files are available in ViSP source tree in `example/manual/hello-world/Makefile` directory.

<sup>3</sup>`HelloWorld.cpp`, `configure.ac`, `Makefile.am` and `have_visp.m4` files are available in ViSP source tree in `example/manual/hello-world/Autotools` directory.

configure.ac.txt :

```

1 AC_PREREQ(2.50)
2 AC_INIT>HelloWorld, 1.0)
3 AM_INIT_AUTOMAKE
4 AC_PROG_CXX
5 AC_CONFIG_FILES([Makefile])
6
7 AC_HAVE_VISP_IFELSE(have_visp=yes,have_visp=no)
8 if test "$have_visp" = "xyes"; then
9     CXXFLAGS="$CXXFLAGS $ac_visp_cflags "
10    LIBS="$LIBS $ac_visp_libs "
11 fi
12
13 AC_OUTPUT

```

Makefile.am :

```

1 bindir=./
2 bin_PROGRAMS>HelloWorld
3 HelloWorld_SOURCES>HelloWorld.cpp

```

3. Then you have to execute the following commands in a terminal.

```

aclocal -I .
touch NEWS README AUTHORS ChangeLog
automake -a
autoconf
./configure [options]

```

You may have to use options while executing the command `configure`. Indeed, if you use no options, `./configure` will try to find the needed `visp-config` shell script from the default path `/usr/bin`. To specify the location of the script (which is located in `VISP_BUILD_DIR/bin`), you have to use the option `--with-visp-install-bin`. For example if you build ViSP in the folder `/usr/local/ViSP-build` you have to use the command: `./configure --with-visp-install-bin=/usr/local/ViSP-build/bin`. If you installed ViSP, it is also possible to use the command: `./configure --with-visp-install-bin=VISP_INSTALL_DIR/bin`. Do not forget to replace `VISP_INSTALL_DIR` by the right path (see Section 3, step 3).

4. Finally, just use the command `make` to build the project.

## 5 Additional information

### 5.1 Third party libraries used by ViSP

Many ViSP functionalities require third party libraries. This is in particular the case for simulation, framegrabbing and image viewer capabilities that require respectively Ogre 3D or Coin, SoQt and Qt, libdc1394 or Video For Linux 2 and X11, GTK2 or GDI. If you want to know the entire list of third party libraries that can be used in ViSP you can get the information on ViSP website. Table 1 summarize these third party libraries and gives the environment variable names that could be set to help CMake to detect them.

ViSP capabilities	Third party library	Corresponding environment variable
Image viewer <sup>a</sup>	X11	none
	GTK2	GTK2_DIR
	OpenCV	OPENCV_DIR
SVD computation <sup>b</sup>	Lapack	LAPACK_DIR
	OpenCV	OPENCV_DIR
	GSL	GSL_DIR
Image bridges <sup>c</sup> and computer vision <sup>d</sup>	OpenCV	OpenCV_DIR
	Yarp	YARP_DIR
	Coin	COIN_DIR or COINDIR
	XML2	XML2_DIR
Frame grabbing <sup>e</sup>	libdc1394-2	DC1394_DIR
	Video For Linux 2	V4L2_DIR
	OpenCV	OpenCV_DIR
Specific devices	Keenect	LIBFRENECT_HOME
Robots <sup>f</sup>	Biclops	BICLOPS_HOME
	Pioneer	ARIA_HOME
Simulator	Ogre	OGRE_HOME and OGRE_MEDIA_DIR
	OIS	none
	Coin	COIN_DIR or COINDIR
	SoQt	COIN_DIR or COINDIR or SOQT_DIR
	Qt	QTDIR
Camera parameters parser	Coin	COIN_DIR or COINDIR
	SoXt	COIN_DIR or COINDIR
Image reading and writing	XML2	XML2_DIR
	iconv	XML2_DIR or ICONV_DIR
	libjpeg	LIBJPEG_DIR
HTML documentation	libpng	LIBPNG_DIR
	FFmpeg	FFMPEG_DIR
	Doxygen	DOXYGEN_DIR
	Graphviz	GRAPHVIZ_DIR

Table 1: List of environment variables that can be set thanks to the `export` or `setenv` command depending on your Unix shell system to help CMake to detect third party libraries that may be used to build ViSP. For example you may use a command like "`export COIN_DIR=/local/coin`" or "`setenv COIN_DIR /local/coin`" to indicate where Coin is installed.

<sup>a</sup>Only one device is requested to show ViSP images. X11, GTK and OpenCV are alternatives. We suggest to use X11 which is native on Unix.

<sup>b</sup>To compute the pseudo inverse based on the Singular Value Decomposition (SVD) ViSP may use one of the following Lapack, OpenCV or GSL third party library. If Lapack is not found, OpenCV will be used. Then, if OpenCV is not found the Gnu Scientific Library (GSL) will be used. Finally if none of those libraries are found, an internal implementation will be used. We suggest to install Lapack.

<sup>c</sup>ViSP provides OpenCV and Yarp images bridges.

<sup>d</sup>ViSP exploit OpenCV features based for example on key points. Coin and XML2 are used by ViSP model-based tracker (MBT) to load vrml cad models of the object to track and parse tracker parameters respectively .

<sup>e</sup>ViSP implements wrapper over `libdc1394-2` able to grab images from firewire cameras, `V4L2` for USB cameras and OpenCV able to handle firewire and USB cameras.

<sup>f</sup>These robots are interfaced with their native drivers.

If you are interested to know which are the third party libraries used to build ViSP on your computer, you can first check the `ViSP-third-party.txt` text file produced during the CMake configuration stage described in Section 3, step 3. This file is generated in `VISP_BUILD_DIR` directory. Figure 6 shows an example of such a `ViSP-third-party.txt` file content.

Another way to check which are the third party libraries that will be used while building ViSP, is during CMake configuration to press the "`␣`" key on your keyboard to have access to the CMake variables (see Section 3, step 3). If you are sure you install a third party library which is noted as `NOT_FOUND`, it seems that you installed it in a not common folder. So you have the choice to set an environment variable to indicate the path to the library.

CMake detects automatically the available third party libraries on your computer. But for some reasons, you may not want to build ViSP with all the detected libraries. You can disable these libraries during the CMake configuration (see Section 3, step 3). Indeed there are options named `USE_THIRD_PARTY` (see Figure 7) which appear. In that case, `THIRD_PARTY` is the name of the third party library which is automatically detected. To disable one of the third party library, turns the corresponding option to `OFF` (by default they are set to `ON`).

## 5.2 How to execute ViSP examples

Some ViSP examples require data like images or videos as input. They can be downloaded on ViSP website. After download and unzip, you have to set the environment variable `VISP_INPUT_IMAGE_PATH`. It must be set to the parent directory containing the unzipped data. For example, if you download the `ViSP-images-2.8.0.zip` zip file and unzip it in the folder `/usr/local/images`, you will get a new folder named `/usr/local/images/ViSP-images` containing the data. You need then to set the environment variable `VISP_INPUT_IMAGE_PATH` to `/usr/local/images`. Now you should be able to execute the examples that request input data. To set the environment variable it exists many ways depending on your Unix system. You can use for example one of the following commands: `"export VISP_INPUT_IMAGE_PATH=/usr/local/images"` or `"setenv VISP_INPUT_IMAGE_PATH /usr/local/images"`.

```

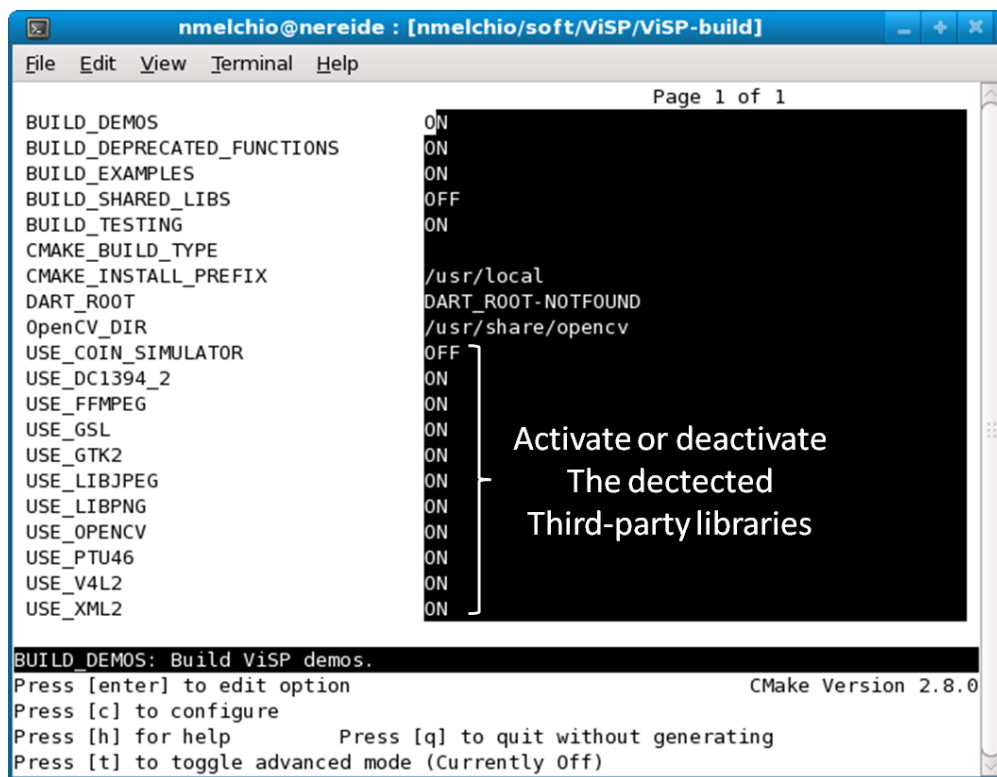
ViSP-third-party.txt ✕
ViSP third-party libraries

Below you will find the list of third party libraries used to
build ViSP on your computer.

Mathematics:
  Gnu Scientific Library      : no
  Lapack/blas                 : yes
Simulator:
  Ogre simulator              : yes
  \- Ogre3D                   : yes
  \- OIS                       : yes
  Coin simulator              : no
  \- Coin3D                    : yes
  \- SoWin                     : no
  \- SoXt                      : no
  \- SoQt                      : no
  \- Qt4                       : no
  \- Qt3                       : no
Robots
  Afma6                       : yes
  Afma4                       : yes
  Biclops                     : no
  Ptu46                       : no
  Pioneer                     :
  Viper S850                  : yes
  Cycab                       : no
Video devices (display)
  X11                         : yes
  GTK                         : yes
  OpenCV                      : yes
  GDI                         : no
  Direct3D                    : no
Framegrabbers
  Firewire libdc1394-1.x      : no
  Firewire libdc1394-2.x      : yes
  Video For Linux Two         : yes
  DirectShow                  : no
  CMU 1394 Digital Camera SDK : no
  OpenCV                      : yes
Specific devices
  Yarp                        : no
  Kinect                     : yes
  \-libfreenect               : yes
  \-libusb-1.0                : yes
  \-pthread                   : yes
Video and image Read/Write:
  FFmpeg                      : yes
  libjpeg                    : yes
  libpng                      : yes
Misc:
  XML2                       : yes
  pthread                    : yes
  OpenMP                     : yes
Documentation:
  Doxygen                    : yes
  Graphviz dot               : yes
ViSP built with C++11 features: no

```

Figure 6: Example of the `ViSP-third-party.txt` file content that indicates which are the third party libraries detected by CMake and used to build ViSP library on your computer.



The screenshot shows a terminal window titled "nmelchio@nereide : [nmelchio/soft/ViSP/ViSP-build]". The window displays the output of the CMake configuration process, listing various options and their current status. A black box with white text is overlaid on the right side of the terminal, containing the instruction: "Activate or deactivate The detected Third-party libraries".

```
File Edit View Terminal Help
Page 1 of 1
BUILD_DEMOS ON
BUILD_DEPRECATED_FUNCTIONS ON
BUILD_EXAMPLES ON
BUILD_SHARED_LIBS OFF
BUILD_TESTING ON
CMAKE_BUILD_TYPE
CMAKE_INSTALL_PREFIX /usr/local
DART_ROOT DART_ROOT-NOTFOUND
OpenCV_DIR /usr/share/opencv
USE_COIN_SIMULATOR OFF
USE_DC1394_2 ON
USE_FFmpeg ON
USE_GSL ON
USE_GTK2 ON
USE_LIBJPEG ON
USE_LIBPNG ON
USE_OPENCV ON
USE_PTU46 ON
USE_V4L2 ON
USE_XML2 ON
BUILD_DEMOS: Build ViSP demos.
Press [enter] to edit option CMake Version 2.8.0
Press [c] to configure
Press [h] for help Press [q] to quit without generating
Press [t] to toggle advanced mode (Currently Off)
```

Figure 7: This CMake snapshot shows where you can find the options used to enable or disable the detected third party libraries.