Università degli Studi di Padova

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE
Corso di Laurea Magistrale in Ingegneria Informatica

Multi-camera calibration in distributed vision system

Candidato:
Andrea Cervellin

Relatore:
Emanuele Menegatti

Correlatore:
Stefano Ghidoni

Anno Accademico 2010-2011

Ai miei genitori
che mi hanno sempre aiutato e sostenuto
in questo importante percorso.

A mio fratello
che ha sempre creduto in me.

A Laura e a tutte le persone
che mi sono state vicine.

Abstract

This work focuses on a calibration of a set of heterogeneous cameras (pinhole and omnidirectional cameras), where images processing and analysis are necessary to retrieve information about the framed scene. The main problem considered in this work regards the determination of a generic point position placed on the floor and shown by a set of cameras. With the objective to correlate a generic floor point to a point in the camera video streaming, it is necessary to compute the main parameters that describe the cameras model and those that correlate the cameras to the floor. For the first parameters are used some tools found in the literature. For the latter a simple and well defined procedure and a toolbox for the pinhole camera have been developed. Moreover, the results of quality tests and verifications coming from the *Click and View* application specifically builtare shown. Finally it was included *Click and Go* application, a robot controller that use this system.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Summary

The purpose of this work is the calibration of a set of heterogeneous cameras (pinhole and omnidirectional cameras) located indoors (figure 1.1). The main objective consists in the determination of the parameters that describe the cameras model (intrinsic parameters) and the parameters that correlate cameras measurements with the corresponding measurements in the real environment (extrinsic parameters). With the extrinsic parameters that bind the image plane to the real world, it is possible to estimate with precision the position of a generic item on the floor and also inside the image plane of each camera. In order to allow the communication of different devices that analyzed the scene, it has been performed an initial setup phase to install and test the middleware for the communication layer. With this layer it is possible to acquire data coming from heterogeneous cameras to gather all video streaming giving a complete picture of the environment. To do that has been developed a first application called *Click and View* that also tests and verifies the calibration quality and a second application called *Click and Go* to manage and drive a robot inside the environment. Finally the results of the work are shown.

Figure 1.1: Pictures of the environment.

1.2 State of the art

Camera calibration is the first step towards computational computer vision. The parameters obtained from the calibration are essential to obtain a precise 3D reconstruction of real world object from projected 2D images. There are two different steps to obtain these main parameters of the cameras:

- the determination of the intrinsic parameters that describe the camera model;
- the estimation of the extrinsic parameters that relate the image plane with respect to the real floor plane.

For both parameters are available some Matlab or C++ tools able to extract them. Most of these tools are based on the Zhengyou Zhang's work described in [2]. The work is based on these key points:

- take several images of a planar chessboard;
- find chessboard in each image;
- find subpixel corners of each chessboard;
- get the chessboard coordinate in 2D worldspace of each chessboard corner;
- estimate a homography for each image;
- estimate intrinsic matrix from the set of homographies;
- estimate extrinsic parameters for each chessboard;
- estimate coefficients of distortion.

Compared with classical techniques [3] [4], this is easy to use and flexible as it only requires the camera to observe a planar pattern from a few (at least two) different orientations.

In fact the cameras calibration based on the Zhang technique are the following: for the pinhole camera are the Matlab[1] "Caltech Calibration Toolbox" by Jean-Yves Bouguet and the C++ tool developed using the OpenCv[2] library (an open source computer vision library). For the omnicamera, it is the Matlab[3] toolbox by Davide Scaramuzza.

Regarding the extrinsic calibration, it is more difficult find an appropriate and specific tools for this project requirements. In the works analyzed during the initial stage, there isn't a clear and well defined method to obtain the extrinsic parameters of a fixed camera looking to the floor. As a results, for both the pinhole and the omnidirectional camera it was necessary develop an alternative procedure (and the related toolbox in the pinhole camera case) to obtain the correct extrinsic parameters matrix.

[1]http://www.vision.caltech.edu/bouguetj/calib_doc/

[2]http://opencv.itseez.com/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#calibratecamera

[3]https://sites.google.com/site/scarabotix/ocamcalib-toolbox

1.3 Content

This document is structured as follows:

- in the second chapter is described the installation of the Network-Integrated Multimedia Middleware (NMM), the problems encountered and the testing of distributed nodes;
- in the third chapter is described the calibration of the pinhole camera and omnidirectional camera: initial theoretical references, the explanation for the automatic extraction of camera parameters and then the description of the C++ tools for the pinhole camera extraction;
- in the fourth chapter is described the *Click and View* application that use all the camera parameters previously extracted to map with precision every point on the floor onto each camera window;
- in the fifth chapter is described the *Click and Go* application that use the application previously developed to controll a robot inside the enviroment;
- in the sixth chapter there are the results about the camera calibration and applications;
- in the seventh there are the final conclusions about the work.

Chapter 2

Network-Integrated Multimedia Middleware (NMM)

2.1 Introduction to Network-Integrated Multimedia Middleware

2.1.1 Introduction

This chapter provides a description of the Network-Integrated Multimedia Middleware taken from NMM documentation.
Besides the PC, an increasing number of multimedia devices – such as set-top boxes, PDAs, tablet, smartphone and mobile phones – already provide networking capabilities. However, today's multimedia infrastructures adopt a centralized approach, where all multimedia processing takes place within a single system. Conceptually, such approaches consist of two isolated applications, a server and a client and the network is used for streaming predefined content from a server to clients. The realization of complex scenarios is therefore complicated and error-prone.

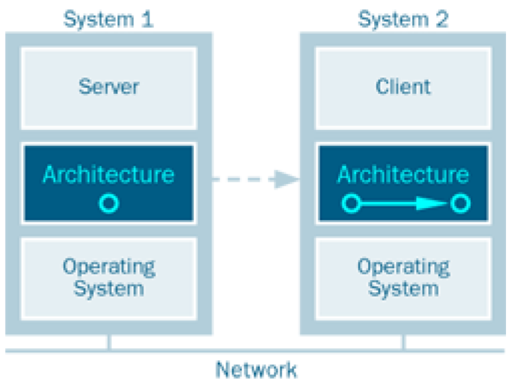


Figure 2.1: Client/server streaming consists of two isolated applications that do not provide fine-grained control or extensibility.

The Network-Integrated Multimedia Middleware[1] (NMM) overcomes these limitations by

[1]http://www.motama.com/nmm.html

enabling access to all resources within the network: distributed multimedia devices and software components can be transparently controlled and integrated into an application. In contrast to all other multimedia architectures available, NMM is a *middleware*, i.e. a distributed software layer running in between distributed systems and application.

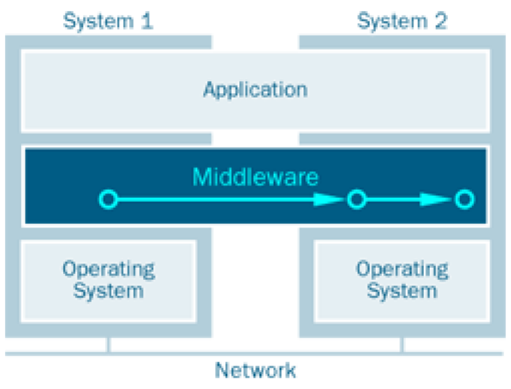


Figure 2.2: A multimedia middleware is a distributed software layer that eases application development by providing transparency.

As an example, this allows for the quick and easy development of an application that receives TV from a remote device – including the transparent control of the distributed TV receiver. Even a PDA with only limited computational power can run such an application: the media conversions needed to adapt the audio and video content to the resources provided by the PDA can be distributed within the network. While the distribution is transparent for developers, no overhead is added to all locally operating parts of the application. To this end, NMM also aims at providing a standard multimedia framework for all kinds of desktop applications.

NMM is both an active research project at Saarland University in Germany and an emerging Open Source project. NMM runs on a variety of operating systems and hardware platforms (please refer to NMM FAQ for details). NMM is implemented in C++, and distributed under a dual-license: NMM is released under 'free' licenses, such as the GPL, and commercial licenses.

2.1.2 Nodes, Jacks, and Flow Graphs

The general design approach of the NMM architecture is similar to other multimedia architectures. Within NMM, all hardware devices (e.g. a TV board) and software components (e.g. decoders) are represented by so called nodes. A node has properties that include its input and output ports, called jacks, together with their supported multimedia formats. A format precisely defines the multimedia stream provided, e.g. by specifying a human readable type, such as 'audio/raw' for uncompressed audio streams, plus additional parameters, such as the sampling rate of an audio stream. Since a node can provide several inputs or outputs, its jacks are labeled with tags. Depending on the specific kind of a node, its innermost loop produces data, performs a certain operation on the data, or consumes data. Our system distinguishes between different types of nodes: a source produces data and has one output jack. A sink consumes data, which it receives from its input jack. A filter has one input and one output jack. It only modifies the data of the stream and does not change its format or format specific parameters. A converter also has one input and one output jack but can change the format of the data (e.g. from raw video to compressed video) or may change format specific parameters (e.g. the video resolution).

A multiplexer has several input jacks and one output jack; a demultiplexer has one input jack and several output jacks. Furthermore, there is also a generic mux-demux node available. A guide [2] provides step-by-step instructions on how to develop of a new node. These nodes can be connected to create a flow graph, where every two connected jacks need to support a 'matching' format, i.e. the formats of the connected input jack respectively output jack need to provide the same type and all parameters and the respective values present in one format need to be available for the other and vice versa. The structure of this graph then specifies the operation to be performed, e.g. the decoding and playback of an MP3 file.

2.1.3 Messaging System

The NMM architecture uses a uniform messaging system for all communication. There are two types of messages. Multimedia data is placed into buffers. Event forward control information such as a change of speaker volume. Events are identified by a name and can include arbitrary typed parameters. There are two different types of interaction paradigms used within NMM. First, messages are streamed along connected jacks. This type of interaction is called instream and is most often performed in downstream direction, i.e. from sources to sinks; but NMM also allows for sending messages in upstream direction. Notice that both buffers and events can be sent instream. For instream communication, so called composite events are used that internally contain a number of events to be handled within a single step of execution. Instream events are very important for multimedia flow graphs. For example, the end of a stream (e.g. the end of a file) can be signaled by inserting a specific event at the end of a stream of buffers. External listener objects can be registered to be notified when certain events occur at a node (e.g. for updating the GUI upon the end of a file or for selecting a new file). Events are also employed for the second type of interaction called out-of-band, i.e. interaction between the application and NMM objects, such as nodes or jacks. Events are used to control objects or for sending notifications from objects to registered listeners.

2.1.4 Interfaces

In addition to manually sending events, object-oriented interfaces allow to control objects by simply invoking methods, which is more type-safe and convenient then sending events. These interfaces are described in NMM Interface Definition Language (NMM IDL)[3] that is similar to CORBA IDL. According to the coding style[4] of NMM, interfaces start with a capital 'I'. For each description, an IDL compiler creates an interface and implementation class. While and implementation class is used for implementing specific functionality within a node, an interface class is exported for interacting with objects. During runtime, supported events and interfaces can be queried by the application. Notice that interfaces described in NMM IDL describe out-of-band and instream interaction.

2.1.5 Distributed Flow Graphs

What is special about NMM is the fact that NMM flow graphs can be distributed across the network: local and remote multimedia devices or software components encapsulated within nodes can be controlled and integrated into a common multimedia processing flow graph, a distributed flow graph. While this distribution is transparent for application developers, no overhead is

[2] http://www.motama.com/nmmdocs_plugins.html
[3] http://www.motama.com/nmmdocs_idl.html
[4] http://www.motama.com/nmmdocs_style.html

added to all locally operating parts of the graph: In such cases, references to already allocated messages are simply forwarded – no networking is performed at all. The following shows an example for a distributed flow graph for playing back encoded (compressed) files, e.g. MP3 files. A source node for reading data from the local file system (AudioReader) is connected to a node for decoding audio streams (AudioDecoder). This decoder is connected to a sink node for rendering uncompressed audio using a sound board (AudioRenderer). Once the graph is started, the source nodes reads a certain amount of data from a given file, encapsulates it into buffers, e.g. of size 1024 bytes, and The application controls all parts of this flow graph using interfaces, e.g. INode for controlling the generic aspects of all instantiated nodes. Notice that three different hosts are present in our example. The application itself runs on host1, the source node on host2, and the decoder and sink node on host3. Therefore, NMM automatically creates networking connections between the application and the three distributed nodes (out-of-band interaction), but also between the source and the decoder node (instream interaction). Therefore, compressed MPEG audio data is transmitted over the network. Notice that such a simple but distributed flow graph already provides many benefits. First, it allows an application to access files stored on distributed systems without the need for a distributed file system, such as NFS. Second, the data streaming between connected distributed nodes is handled automatically by NMM. Third, the application acts as 'remote control' for all distributed nodes. As an example, this allows for transparently changing the output volume of the remote sound board by a simple method invocation on a specific interface, e.g. IAudioDevice.

2.1.6 Distributed Synchronization

Since NMM flow graphs can be distributed, they allow for rendering audio and video on different systems. For example, the video stream of an MPEG2 file can be presented on a large screen connected to a PC while the corresponding audio is played on a mobile device. To realize synchronous playback of nodes distributed across the network, NMM provides a generic distributed synchronization architecture[5]. This allows for achieving lip-synchronous playback as required for the above described setup. In addition, media presentations can also be performed on several systems simultaneously. A common application is the playback of the same audio stream using different systems located in different rooms of a household – a home-wide music system. The basis for performing distributed synchronization is a common source for timing information. We are using a static clock within each address space. This clock represents the system clock that is globally synchronized by the Network Time Protocol (NTP) (http://www.ntp.org/) and can therefore be assumed to represent the same time basis throughout the network. With our current setup, we found the time offset between different systems to be in the range of 1 to 5 ms, which is sufficient for our purposes. A primer[6] describes how to set up NTP for NMM.

2.1.7 Registry Service

The registry service in NMM allows discovery, reservation, and instantiation of nodes available on local and remote hosts. On each host a unique registry server administrates all NMM nodes available on this particular system. For each node, the server registry stores a complete node description that includes the specific type of a node (e.g. 'sink'), its name (e.g. 'PlaybackNode'), the provided interfaces (e.g. 'IAudioDevice' for increasing or decreasing the output volume), and the supported input and output formats (e.g. an input format 'audio/raw' including additional parameters, such as the sampling rate). The application uses a registry client to send requests

[5]http://www.motama.com/nmmdocs_time.html
[6]http://www.motama.com/nmmdocs_ntp.html

to registry servers running on either the local or remote hosts. Registry servers are contacted by connecting to a well-known port. After successfully processing the request the server registry reserves the requested nodes. Nodes are then created by a factory either on the local or remote host. For nodes to be instantiated on the same host, the client registry will allocate objects within the address space of the application to avoid the overhead of an interprocess communication. To setup and create complex distributed flow graphs, an application can either request each node separately or use a graph description as query. Such a description includes a set of node descriptions connected by edges. For an application to be able to create a distributed flow graph, the NMM application called serverregistry needs to be running on each participating host. For purely locally operating applications this is not required. Then, a server registry is running within the application itself but not accessible from remote hosts. Before a server registry can be used, it needs to determine which devices and software components are available on a particular host. Therefore, the registry needs to be initialized once using following command.

2.1.8 Clic - An Application for Setting up NMM Multimedia Flow Graphs

The NMM framework is used to build up multimedia applications. The basic components in this framework are nodes which perform a certain functionality, like reading a video file or decoding it. Such nodes can be connected to a flow graph to perform a certain task like watching a movie or transcoding it into another format. Especially transcoder jobs do not need any additional user interaction if the application is running. Writing such an application without any user interaction is currently straightforward. First, the application requests these nodes from the registry, connects them to a flow graph and finally starts the entire flow graph. The application *clic (Command Line Interaction and Configuration)* is used to build such a standard NMM application automatically from a textual description which is called 'graph description'. Furthermore, clic allows to use the graph builder of NMM that automatically creates a distributed flow graph from a given URL: the source of media, the audio output, and the video output can be distributed.

A graph description consists of two major parts. The first part is an (optional) comment introduced by the character %. The second part specifies the flow graph which describes how to connect the nodes. The following example describes a simple graph to play a .wav audio file.

```
% This graph description realizes a simple WAV player.
% Use the -i option of clic to specify an WAV file

WavReadNode ! ALSAPlaybackNode
```

To run this graph description, copy it into a file, e.g. wavplay.gd, and start clic with the following command:

```
./clic wavplay.gd -i /home/bob/audio/song.wav
```

The option -i is used to specify the input file. To exit the application, enter q at the command line and press enter. Note: If you call clic on linux host with an input file located on a windows host, you need to quote the path to the input file.

```
./clic wavplay.gd -i "c:\home\bob\audio\song.wav"
```

For detailed information about all options simply type:

```
./clic -h
```

Appendix A describes the location of the graph descriptions that are included in the current NMM release.

Will now follow the details of the graph description and describes how to connect the nodes. A node is identified by its name, which is (normally) the C++ class name. The exclamation mark (!) is used to denote a connection between two nodes as seen in the graph description for playing WAV files. In this example the statement

```
WavReadNode ! ALSAPlaybackNode
```

results in clic to request the nodes WavReadNode and ALSAPlaybackNode. If all nodes can be successfully requested, the WavReadNode is connected to the ALSAPlaybackNode, and the MPEGAudioDecodeNode to the PlaybackNode. Then, clic sets the specified parameters from the command line, for example an input file, and starts the entire flow graph.

In several cases you want to configure some node specific parameters like the location of a node. Thus, the clic syntax allows to set such parameters which are prefaced by the symbol '#' followed by a parameter which is identical to the corresponding C++ method provided by the node description of the component.
These parameters must be written after the node name and its jack tag. Currently the following parameter types are supported:

- `setLocation(<string>)`: The setLocation parameter expects a string as argument and thereby allows to specify the host, from where the node is requested. The string can either include the host name of the system or the IP address in dotted-decimal notation (e.g. "127.0.0.1") Note: The application serverregistry must be running on this host.
- `setLocation(<string>)`: The setLocation parameter expects a string as argument and thereby allows to specify the host, from where the node is requested. The string can either include the host name of the system or the IP address in dotted-decimal notation (e.g. "127.0.0.1") Note: The application serverregistry must be running on this host.
- `setPort(<int>)`: The setPort parameter expects an integer as argument and allows to specify the port a server registry is listening on. By default the port 22801 is used.
- `setSharingType(<sharing-type>)`: This parameter expects a sharing type as argument and allows to specify whether the node can be reused from other applications or not.

```
% This graph description describes a simple WAV player where the
%  WavReadNode is running on host with IP address 192.168.1.1 .
% The serverregistry on host with IP address
% is listening on port 22801.
% Use the -i option of clic to specify the WAV file

WavReadNode #setLocation("192.168.1.1")
! ALSAPlaybackNode
```

In this graph description the WavReadNode is not requested from the local system. Instead the node is requested from a serverregistry running on host with IP address 192.168.1.1 which accepts incoming requests on port 22801. If the node can not be requested from this serverregistry the application terminates with a corresponding error message.

2.2 Installation of NMM

The installation section describes the hardware requirements, network configuration and how to install and test the Open Source version of NMM for Linux.

On Linux, most NMM examples and applications require a graphic board with configured Xv extension (refer to the output of xvinfo) and a sound board or chip that can natively playback different sampling rates such as 44.1 kHz. All other hardware, such as cameras, is optional.

To allow one running NMM system to access another running NMM system, the port 22801 and the port range 5000-6000 must not be blocked by a firewall.

In the official NMM installation documentation[7] there are some errors and in some cases the procedure is not clear and not complete. Following is reported a step by step guide for the correct installation. Furthermore we assume that the NMM package is called `nmm-2.2.0.tar.gz` and it will be extracted to the directory `/home/user/nmm-2.2.0`.

The setting of the libraries is the more difficult part of the installation. Some of these must be compiled and built on-site, with all the problems due to 32 and 64 bits architecture.

2.2.1 Download NMM and source code of the library

Download NMM Open Source version from official site [8].
Download also the source code of the `libliveMedia`[9] and `ulxmlrpcpp`[10] (recommended version 1.7.5) libraries.

2.2.2 Installing necessary libraries from Linux repository

Run the following commands:

```
# Update available packages in source repositories
sudo apt-get update
```

[7]http://www.motama.com/download/installation-linux.pdf
[8]http://www.motama.com/nmmdownload.html
[9]http://www.networkmultimedia.org//Download/external/nmm-2.2.0/external-libraries-optional/live.2009.06.02-14.tar.gz
[10]http://sourceforge.net/projects/ulxmlrpcpp/files/ulxmlrpcpp/

```
# Install necessary libraries
sudo apt-get install liba52-0.7.4-dev libfaad-dev ffmpeg
libmp3lame-dev libraw1394-dev libmad0-dev libdvdnav-dev
libdvdread-dev libogg-dev libvorbis-dev libshout3-dev fftw-dev
liblivemedia-dev cdparanoia libpng12-dev libasound2-dev
libx264-dev libjpeg62-dev imagemagick mplayer vlc transcode
ogmtools libxml+ +2.6-dev expat openssl nasm libltdl-dev
libavcodec-dev libavformat-dev libx11-dev libcdparanoia-dev
libmpeg2-4-dev libssl-dev libxext-dev libxv-dev libexpat1-dev
libmagick++-dev

sudo apt-get install libfftw3-dev libssh-dev
```

2.2.3 Creating links for faad and neaacdec libraries

It is necessary to create in `/usr/local/include` the directory `faad` and place the links of `/usr/include/faad.h` and `/usr/include/neaacdec.h` in it. This is necessary because NMM search those files in `/usr/local/include/faad`

```
sudo mkdir -p /usr/local/include/faad
cd /usr/local/include/faad
sudo ln -s /usr/include/faad.h
sudo ln -s /usr/include/neaacdec.h
```

2.2.4 Installing libliveMedia

It is necessary to compile and install liveMedia library, previously downloaded, directly from the source code.

```
# Unpack
tar xzf live.2009.06.02.tar.gz
cd live

# Enter in livemedia/include, open RTPSink.hh, add the following line
# after line 60:
# u_int32_t currentTimestamp() const {return fCurrentTimestamp;}
#
# Add -DUSE_SYSTEM_RANDOM to COMPILE_OPTS in config.linux
# (to avoid segfault)
# In 64 bit system add also -fPIC to COMPILE_OPTS in config.linux

# Build static libraries
./genMakefiles linux
make

# Build shared library
# In 64 bit system add also -fPIC
gcc -shared -o libliveMedia.so.2009.06.02 */*.a
```

```
# Enter in /usr/local/include and create live directory
sudo mkdir live

# Install to /usr/local/
sudo find -name \*.hh -exec cp "{}" /usr/local/include/live ";"
sudo find -name \*.h -exec cp "{}" /usr/local/include/live ";"

# Change permission settings of lybrary's header
# in /usr/local/include/live/
sudo chmod -R 644 /usr/local/include/live/*.*

# Create library and its link
sudo cp libliveMedia.so.2009.06.02 /usr/local/lib
cd /usr/local/lib
sudo ln -s libliveMedia.so.2009.06.02 libliveMedia.so
cd /usr/lib
sudo ln -s /usr/local/lib/libliveMedia.so.2009.06.02 libliveMedia.so
```

2.2.5 Installing ulxmlrpcpp

Also for this library it is necessary to compile and install from the source code. Before compiling and installing the library, make sure the following packages are already installed on your system: `docbook-xsl fop doxygen libblitz0ldbl libblitz-doc libblitz-dev`.

```
# Necessary libraries for ulxmlrpcpp installing
sudo apt-get install docbook-xsl
sudo apt-get install fop
sudo apt-get install doxygen
sudo apt-get install libblitz0ldbl
sudo apt-get install libblitz-doc
sudo apt-get install libblitz-dev

# Unpack
tar xfj ulxmlrpcpp-1.7.5-src.tar.bz2
cd ulxmlrpcpp-1.7.5

# Configure, build and install
./configure
make
sudo make install
```

2.2.6 Setting CPPFLAGS

Set the environment variable CPPFLAGS to avoid the failure of the ffmpeg test library.

```
export CPPFLAGS="-D__STDC_CONSTANT_MACROS"
```

2.2.7 Setting LD_LIBRARY_PATH

Set the environment variable LD_LIBRARY_PATH appropriately. The paths you specify in this variable must include all paths where external libraries needed by NMM are installed.

```
export LD_LIBRARY_PATH=/usr/local/lib:$LD_LIBRARY_PATH
```

2.2.8 Installing NMM

Now finally install NMM.

```
# Enter in nmm-2.2.0 directory
cd /home/user/nmm-2.2.0/

# Run the configure script
./configure --with-extra-libs=/usr/local/lib --with-extraincludes=/
usr/local/include --prefix=/home/user/nmm-2.2.0-installed
--enable-all

#Build NMM
make

# Install NMM
make install
```

If the procedure was successful, NMM is installed in /home/user/nmm-2.2.0-installed. Typing the following commands, the result should be similar to the one below.

```
# Enter in nmm-2.2.0 installed directory
cd /home/bob/nmm-2.2.0-installed/bin

# Run the serverregistry
./serverregistry -s
```

```
serverregistry and Network-Integrated Multimedia Middleware (NMM)
Version 2.2.0
Copyright (C) 2005-2010
Motama GmbH, Saarbruecken, Germany
http://www.motama.com
See licence for terms and conditions of usage
No plugin information available! If you start NMM for the first time
this information is created automatically.
Note: If you ever change your hardware configuration or update the NMM
version you must delete the file
/home/user/.nmm/plugins.2.2.0.userpc._home_user_nmm-2.2.0-installed
or run 'serverregistry -s' again.
Create config file with plugin information ...
```

```
Loading plugins...

AACAudioDecodeNode            available
AC3DecodeNode                 available
AC3ParseNode                  available
ALSAPlaybackNode              available
ALSARecordNode                available
AVDemuxNode                   available
```

2.2.9 Test Audio/Video rendering

Test NMM using the *clic* application.

```
# Enter in nmm-2.2.0 installed directory
cd /home/bob/nmm-2.2.0-installed/bin

# Run the following command for the audio test
./clic ../share/nmm/gd/linux/playback/audio/wavplay.gd -i

# Run the following command for the video test
./clic ../share/nmm/gd/linux/playback/video/noise_yv12.gd
```

After video test you should hear the WAV file being played back and after video test you should see a window showing some white noise.

2.3 Testing of distributed nodes

The final test for the correct functioning of NMM concerns the distributed nodes (figure 2.3). In fact, the previous tests were conducted within the same PC but it is necessary to test NMM on a network where many PCs communicate with each other.

During the tests some problems of communication between nodes on different PCs have been encountered. After a long search and many tests a strange problem with the operating systems has been found. In fact, the problem occurs when you try to communicate with a PC with Ubuntu 10.04:

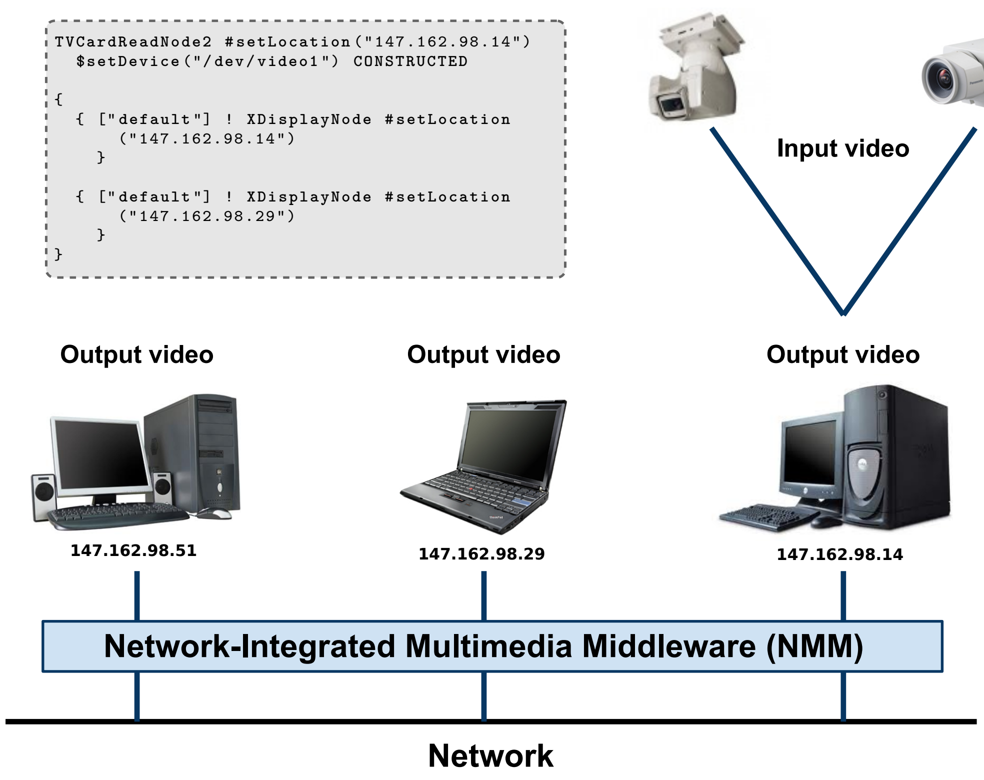
Ubuntu 10.04 (Lucid Lynx)

- *It can process data from other remote hosts with Ubuntu 10.04*
- *All other remote hosts can not process data from it*

Ubuntu 10.10 (Maverick Meerkat)

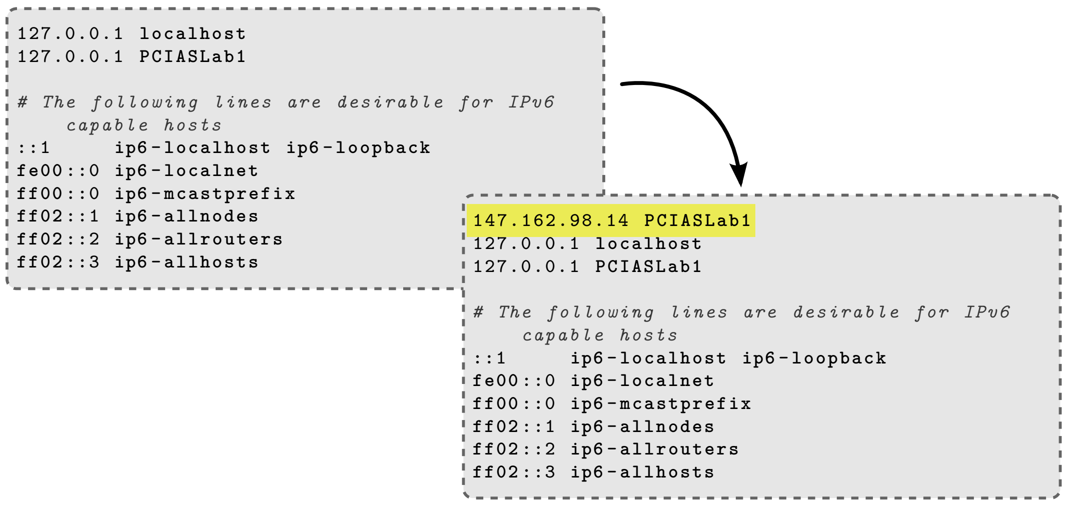
- *It can process data from other remote hosts with Ubuntu 10.04*
- *All other remote hosts can process data from it*v

The problem is in `/etc/hosts` file, only in Ubuntu 10.04. To solve the problem it is necessary to add to this file, as first line, the IP address and the host name of that computer (figure 2.4).



```
TVCardReadNode2 #setLocation("147.162.98.14")
  $setDevice("/dev/video1") CONSTRUCTED

{
  { ["default"] ! XDisplayNode #setLocation
      ("147.162.98.14")
    }

  { ["default"] ! XDisplayNode #setLocation
      ("147.162.98.29")
    }
}
```

Figure 2.3: Graphical representation of the distributed system. At the top left an example of a graph for the distributed test.



```
127.0.0.1 localhost
127.0.0.1 PCIASLab1

# The following lines are desirable for IPv6
    capable hosts
::1      ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
ff02::3 ip6-allhosts
```

```
147.162.98.14 PCIASLab1
127.0.0.1 localhost
127.0.0.1 PCIASLab1

# The following lines are desirable for IPv6
    capable hosts
::1      ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
ff02::3 ip6-allhosts
```

Figure 2.4: Solution of the problem in the distributed nodes.

Chapter 3

Camera calibration

In this chapter, it is describe the used camera and their calibrations. The camera is a pinhole camera type, which consists in an imaginary wall with a tiny hole in the center that blocks all rays except those passing through the tiny aperture in the center. Unfortunately, a real pinhole is not a very good way to make images because it does not gather enough light for rapid exposure. This is why pinhole cameras use lenses to gather more light than what would be available at a single point. The downside, however, is that gathering more light with a lens not only forces us to move beyond the simple geometry of the pinhole model but also the lens' use introduces distortion. For this reason a camera calibration is necessary in order to correct the main deviations caused by the adoption of the lens.

Camera calibration is important also for relating camera measurements with measurements in the real three-dimensional world. This point is the core of this work: finding a simple and reliable method to extract the extrinsic parameters for a pinhole camera placed at a precise point in the world. In fact, the relation between the camera's natural units (pixels) and the units of the physical world (e.g., meters) is a critical component in any attempt to minimize the difference between the predicted image point position, as computed from the projection of the 3D scene points, and the actual image point position, as observed on the image. These extrinsic parameters allow to estimate with precision the position of a generic single point positioned on the floor and also in the image plane of the camera.

3.1 Pinhole Camera

For this work a pinhole camera was used. According to this simple model, light is envisioned as entering from the scene or a distant object, but only a single ray enters from any particular point. In a pinhole camera, this point is then "projected" onto an imaging surface. As a result, the image on this image plane (also called the projective plane) is always in focus, and the size of the image relative to the distant object is given by a single camera's parameter: the *focal length* (f). This is shown in figure 3.1. In addition to the f focal length of the camera, we can see Z, the distance from the camera to the object; X is the length of the object and x is the object's image on imaging plane. Furthermore, we can see two similar triangles, both having parts of the projection line (from x to X) as their hypotenuses. The catheti of the left triangle are x and f and the catheti of the right triangle are X and Z. Since the two triangles are similar it follows that $-\frac{x}{f} = \frac{X}{Z}$ or $x = f\frac{X}{Z}$.

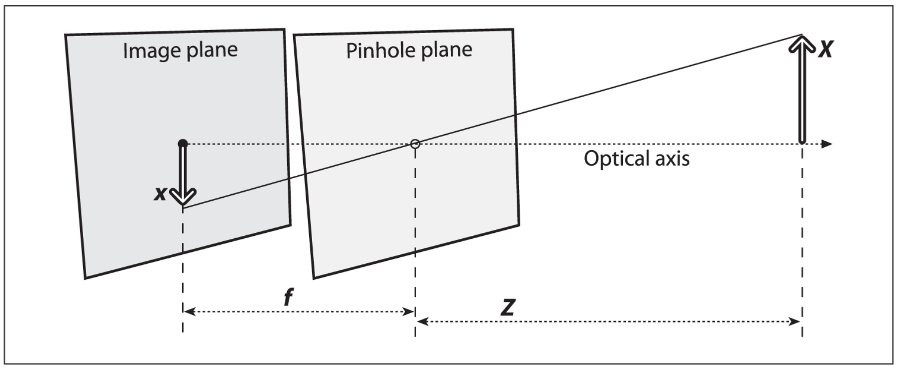


Figure 3.1: Pinhole camera model. Figure from [1].

We shall now rearrange the pinhole camera model to a form that is equivalent but in which mathematically comes out easier. In the next figure (figure 3.2), we swap the pinhole and the image plane. The main difference is that the object now appears right-side up.

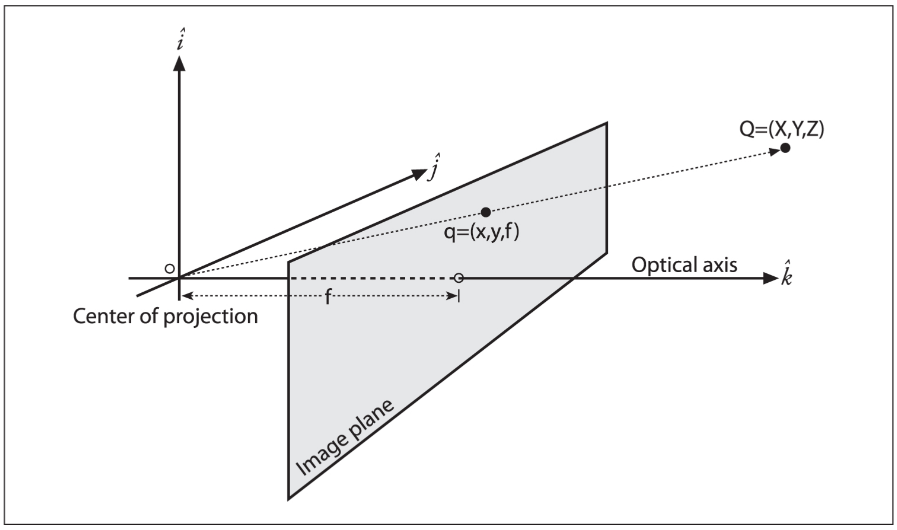


Figure 3.2: Easier pinhole camera model. Figure from [1].

The point in the pinhole is reinterpreted as the *center of projection.* In this way, every ray leaves a point on the distant object and heads for the center of projection. The point at the intersection of the image plane and the optical axis is referred as the principal point.

This new frontal image plane is the equivalent of the old projective or image plane. The image of the distant object is exactly the same size as it was on the image plane. The similar triangles relationship $\frac{x}{f} = \frac{X}{Z}$ is more directly evident than before. The negative sign is gone because the object image is no longer upside down. The principal point is equivalent to the center of the image but, due to manufacturing defects coming through the camera assembling, the center of the video sensor is usually not perfectly aligned on the optical axis. Two new parameters, c_x and c_y, must be introduced to model a possible displacement of the center of coordinates on the projection screen.

Now it is possible to define the following equations that project a world point Q (whose coordinates are (X,Y,Z)) onto the image plane at some pixel location given by (x_{img}, y_{img}):

$$x_{img} = f_x \left(\frac{X}{Z}\right) + c_x \quad , \quad y_{img} = f_y \left(\frac{Y}{Z}\right) + c_y$$

Note that we have introduced two different focal lengths (f_x, f_y); the reason is that the individual pixels on a typical low-cost imager are rectangular rather than square.

One aspect that characterizes each camera is the type of lens installed. In theory, it is possible to define a lens that does not introduce any distortion. Actually, no lens is perfect. For this reason it is necessary to consider the distortion introduced by the camera lens to limit the error introduced by the projection of the points.

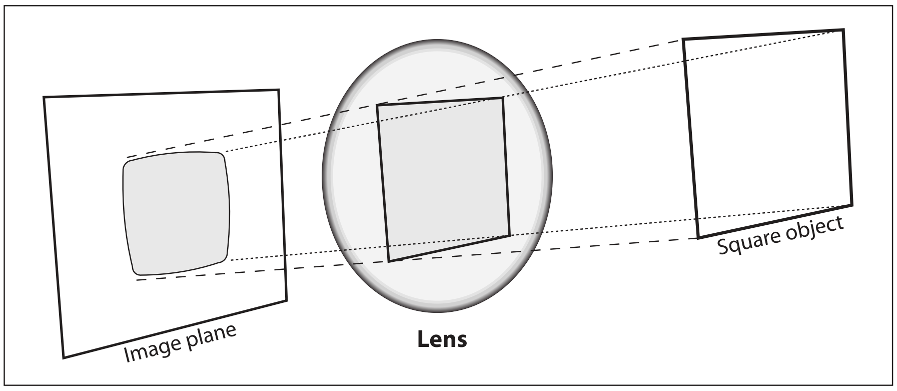


Figure 3.3: Radial distortion. Figure from [1].

Here below we describe the two main lens distortions and how to model them.

Radial distortion: the lenses of real cameras often noticeably distort the location of pixels near the edges of the imager. This bulging phenomenon is the source of the "barrel" or "fish-eye" effect. In this distortion, image magnification decreases with distance from the optical axis. The apparent effect is that of an image which has been mapped around a sphere (or barrel). Fisheye lenses, which take hemispherical views, utilize this type of distortion as a way to map an infinitely wide object plane into a finite image area (figure 3.3).

Tangential distortion: this distortion is due to imperfect centering ("decentering") of the lens components and other manufacturing defects in a compound lens (figure 3.4).

The first distortion is small and can be characterized by the first few terms of a Taylor series expansion around r = 0. For highly distorted cameras such as fish-eye lenses we can use three terms (k_1, k_2, k_3). In general, the radial location of a point on the imager will be rescaled according to the following equations:

$$x_{corr} = x\left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right) \qquad y_{corr} = y\left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right)$$

Here, (x, y) is the original location (on the imager) of the distorted point and (x_{corr}, y_{corr}) is the new location as a result of the correction. Tangential distortion is minimally characterized by two additional parameters, p_1 and p_2, such that:

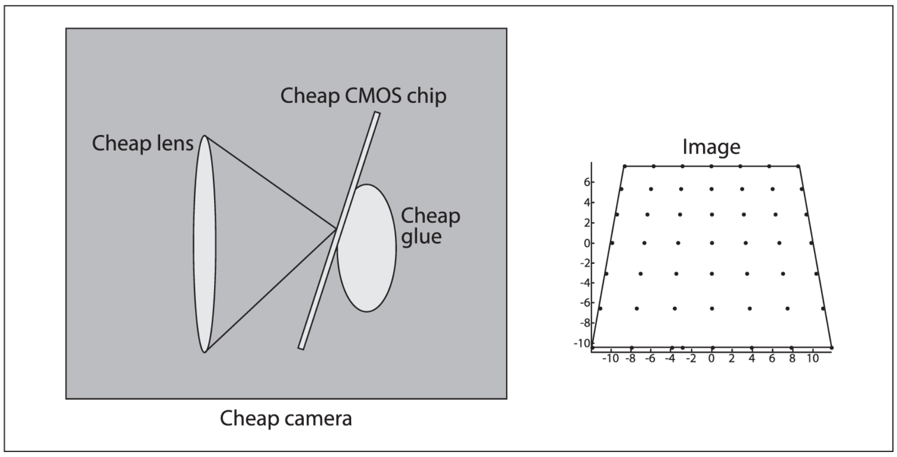


Figure 3.4: Tangential distortion. Figure from [1].

$$x_{corr} = x + \left[2p_1 y + p_2\left(r^2 + 2x^2\right)\right] \qquad y_{corr} = y + \left[p_1\left(r^2 + 2y^2\right) + 2p_2 x\right]$$

Thus in total there are five distortion coefficients that we require $(k_1, k_2, k_3, p_1, p_2)$ that will be determined during the calibration.

3.1.1 Calibration

In this section, will be presented how to find the main parameters that describe the pinhole camera model:

Intrinsic parameters

- Focal lenght $[f_x, f_y]$
- Principal point (usually the image center) $[c_x, c_y]$
- Distortion parameters $[k_1, k_2, k_3, p_1, p_2]$

OpenCV[1], the library of programming functions mainly aimed at real time computer vision, provides several algorithms to help us compute these intrinsic parameters.

We will see also how to find the main parameters that relate image measurements with measurements in the real world:

Extrinsic parameters

- Rotation matrix $R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$
- Translation vector $t = [t_1, t_2, t_3]$

For this parameters we used the OpenCv function to develop a simple tool for the automatic extraction.

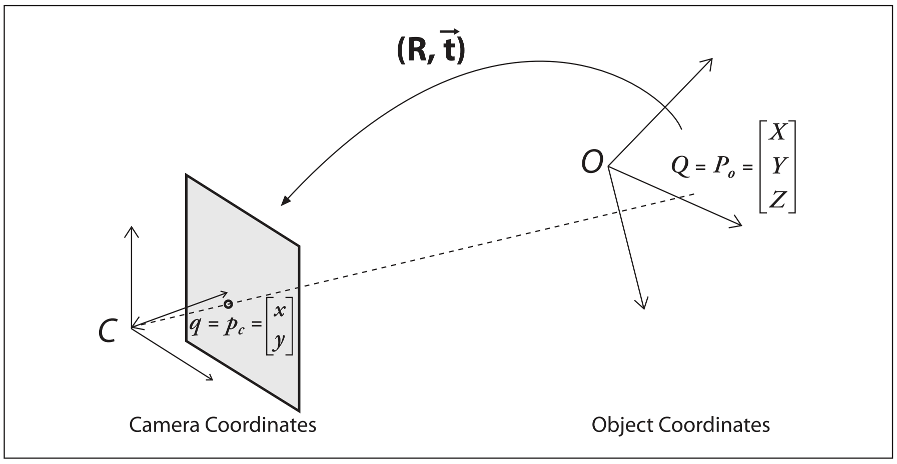

(R, $\vec{t}$)
O
$Q = P_o = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$
C
$q = p_c = \begin{bmatrix} x \\ y \end{bmatrix}$
Camera Coordinates
Object Coordinates

Figure 3.5: Converting from object to camera systems. Figure from [1].

The scene view from the camera is formed by projecting 3D points of an object into the image plane (figure 3.5) using this perspective transformation:

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

where:

- (X, Y, Z) are the coordinates of a 3D point in the world coordinate space
- (u, v) are the coordinates of the projection point in pixels
- $\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} = \text{K}$ is called *intrinsic parameters matrix*
- $\begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix}$ is called *extrinsic parameters matrix*
- s is the scale factor

Intrinsic parameters

Regarding the estimation of intrinsic parameters there are different software applications. The choice is made between the use of a toolbox developed in C++ and one developed in a Matlab[2].

A simple toolbox developed in c++ language was used for homogeneity with the subsequent development of a software for extracting extrinsic parameters, designed in C++ too. It is based

[1]http://opencv.willowgarage.com/
[2]http://www.vision.caltech.edu/bouguetj/calib_doc/

on the OpenCv library and it is a modified version of the proposed algorithm in the *Example 11-1* in [1].

The idea consists in showing to the camera a set of scene points for which their 3D position is known, and then determine where this points are in the image plane. This 3D scene points are generated by a flat chessboard pattern. This pattern creates points that represent the corners of each square. Then, the tool performs the following tasks:

- it loads the images of a planar chessboard
- for each image
  - it looks for the chessboard and it extracts image corners using `cvFindChessboardCorners`
  - it finds the sub-pixel accurate location of a corner using `cvFindCornerSubPix`
- it computes the *intrinsic parameters* using `cvCalibrateCamera2`

This last function is able to compute the *intrinsic parameters* from the real position of 3D points and their position in the image plane. Furthermore, the function returns the *distortion parameters* that characterizes the camera.

3.1.2 Find extrinsic parameters

The extrinsic parameters relate camera measurements with measurements in the real world. Thanks to these parameters, it is possible to link every point on the floor with the correspondent point in the image plane of the camera.

For example, it is necessary to have these parameters if we want obtain the real coordinates of a point on the floor in meters from the correspondent point on the image plane (figure 3.6).

This kind of transformation is described by rotations and a translations.

The rotation is a rigid body movement which keeps a point fixed. It is possible to consider the three dimensional rotation like a decomposition into a two dimensional rotation around each axes. In this way, if we want to rotate around the x, y, and z axes in this sequence, with respective rotation angles ψ, φ and θ, it is sufficient to multiply the respective rotation matrices as following $R = R(\theta)R(\varphi)R(\psi)$ where

$$R(\psi) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\psi) & \sin(\psi) \\ 0 & -\sin(\psi) & \cos(\psi) \end{bmatrix}$$

$$R(\varphi) = \begin{bmatrix} \cos(\varphi) & 0 & \sin(\varphi) \\ 0 & 1 & 0 \\ \sin(\varphi) & 0 & \cos(\varphi) \end{bmatrix}$$

$$R(\theta) = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and R is exactly the rotation matrix of the extrinsic parameters. The translation vector instead, is the offset from the origin of the image camera system to the origin of the real world system.

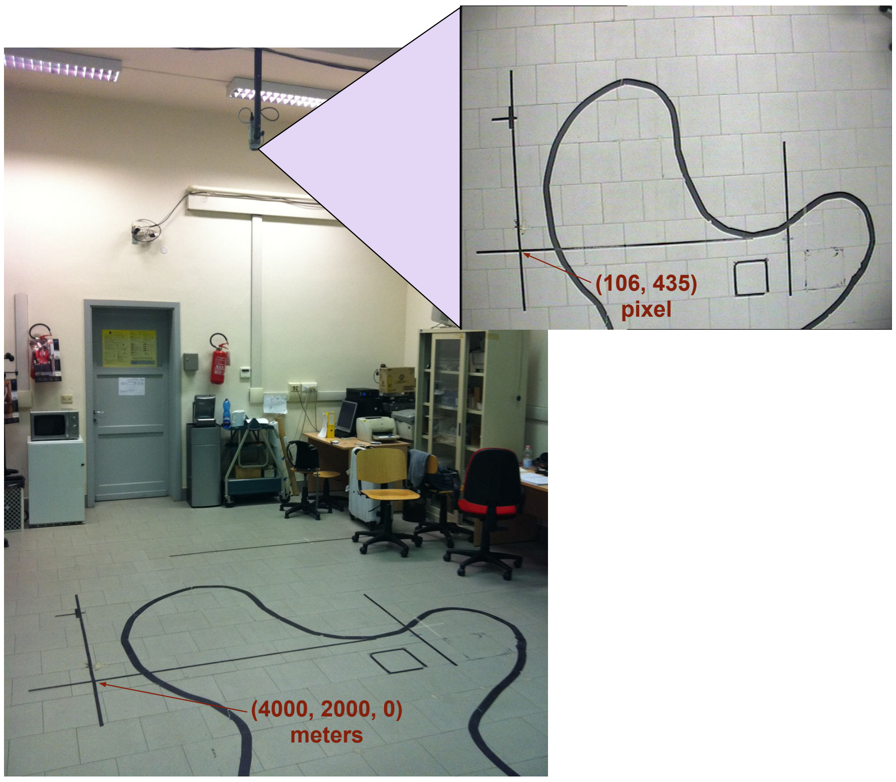


Figure 3.6: View of the same point in the image plane of the camera and on the floor

In order to find these parameters a manually extraction method was used for a single camera. Then an alternative automatic method has been developed that use a c++ tool specifically realized for this project. The automatic extraction tool has been also used to verify the parameters obtained with the manually method allowing a fast, easy and accurate estimation.

Manual extraction

The manual extraction is necessary to know where the camera is placed with respect to the world coordinate system. So it is necessary to take a rule and manually measure parameters such as the global coordinates of the camera or its tilt and pan angles. But, sometimes, it is not so easy to obtain these measure values. Then, it is necessary to calculate the rotation matrix and the translation vector from the measurements.

The translation vector t is the distance vector (with respect to x-axes, y-axes and z-axes) of the camera optical center from the world coordinate system and it is expressed in meters (figure 3.7).

The rotation matrix R is the product of three rotation matrices $R(\psi)$, $R(\varphi)$ and $R(\theta)$ about each axis (figure 3.8), where ψ, φ, θ are the respective rotation angles measured with a protractor.

To manually extract the extrinsic parameters, we need the global coordinates and the tilt and pan angles of each camera that we want to calibrate. But inevitably, there are inaccuracies in the manual measurements. This procedure should be done for each camera, requiring a hard and long effort.

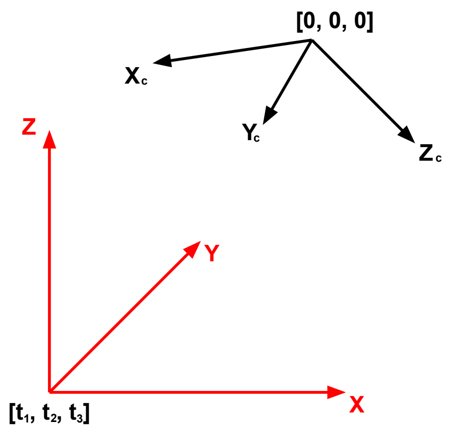


Figure 3.7: The translation vector.

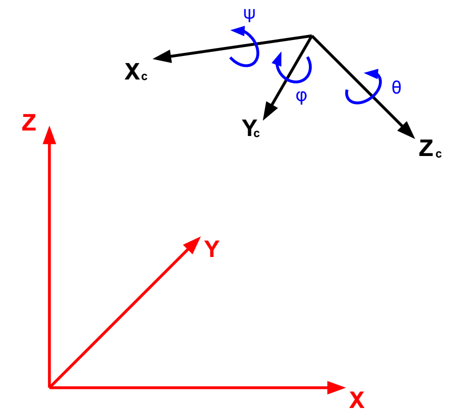


Figure 3.8: The rotation matrix about each axis

Automatic extraction

The next step of this work was to devise an automatic and much less laborious procedure to extract the extrinsic parameters. In order to map the real coordinates of a floor's point expressed in meters from the corresponding point on the image plane, it is necessary to have the roto-translation matrix of the camera.

OpenCV library provides a function that returns the rotation matrix and the translation vector from the array of object points in the object coordinate space and the array of corresponding image points. This function is called `cvFindExtrinsicCameraParams2`.

The idea is to place a chessboard in a specific point of the floor and through the function *cvFindChessboardCorners* extract the corners in the image coordinates. The obtained image is then used in conjunction with the coordinates of all corners in the world system as input of the function `cvFindExtrinsicCameraParams2`. In this way it is possible to obtain the rotation matrix and translation vector which correlates the point of image to the corresponding points of the floor.

The toolbox for extrinsic parameters extraction

The toolbox has been developed in C++ language and it is based on the use of OpenCv library. The input is an image of a chessboard placed in a "well-know point of the world" and specifically on the floor (figure 3.9). The output are the rotation matrix and translation vector of the camera that take this image.

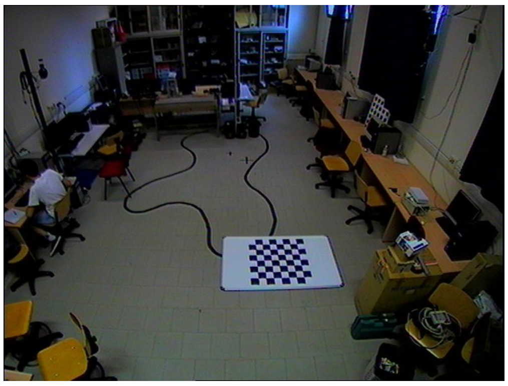


Figure 3.9: Original image input

At the beginning, the software shows some indications to correctly place the chessboard on the floor in order to perform the best calibration. (Figure 3.10)

```
/*Info for the toolbox usage*/
The location with respect to the global reference system of the
corner on the chessboard nearest the origin of world coordinate
system (labelled as the "Known world point" in the diagram)
should be known.

Place the chessboard as shown
```

It is necessary to enter the chessboard parameters.

```
/*Chessboard info*/
Enter the number of corners along the Y direction:
Enter the number of corners along the X direction:
Enter the size of each square: (mm)
```

It loads the image specified in the input argument, the first intrinsic parameters for undistorting the image and the intrinsic parameters for the undistorted image (all the intrinsic parameters should be inside `data/parameters/` folder). After that it shows the undistorted image. This image is the original input image after the undistortion with the first intrinsic parameters (figure 3.11(a)).

Now it is necessary to enter the coordinates of the "well-know point of the world" where the chessboard is placed on the floor. In this way, by knowing the side's size and the numbers of chessboard square, we know the world coordinates in meters for each corner.

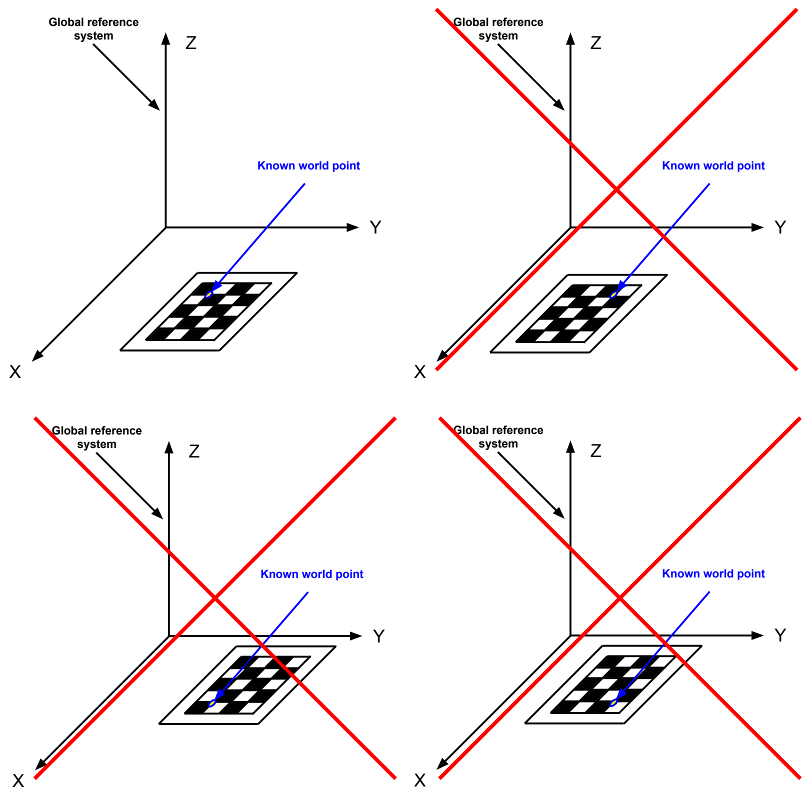


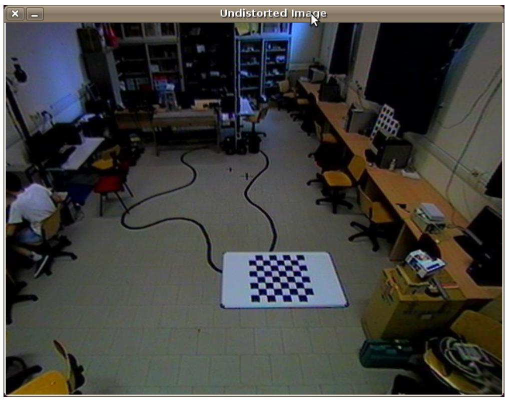
Figure 3.10: How to place the chessboard on the floor. Example of correct positioning (top-left figure) and of three possible errors (top-right, bottom-left and bottom- right)

```
/*Known world point info*/
Enter the x-axis global coordinate of the known point.
Enter the y-axis global coordinate of the known point.
Enter the z-axis global coordinate of the known point.
```

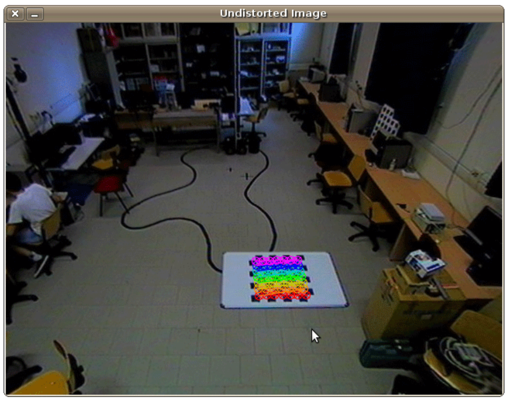
With the OpenCv library it is possible to compute the undistorted image for extracting the image plane coordinates about the corners (figure 3.11(b)).

```
//Finds the positions of the internal corners of the chessboard.
cvFindChessboardCorners(...);

//Refine their location
cvFindCornerSubPix(...);

//Draws the individual chessboard corners detected
cvDrawChessboardCorners(...);
```

Finally, with OpenCv again, from the corners coordinates in the image plane and their correspondent floor coordinates, it is possible to obtain the rotation vector and the translation vector (figure 3.12).



(a) Original undistorted image



(b) Original undistorted image after the extraction of the corners

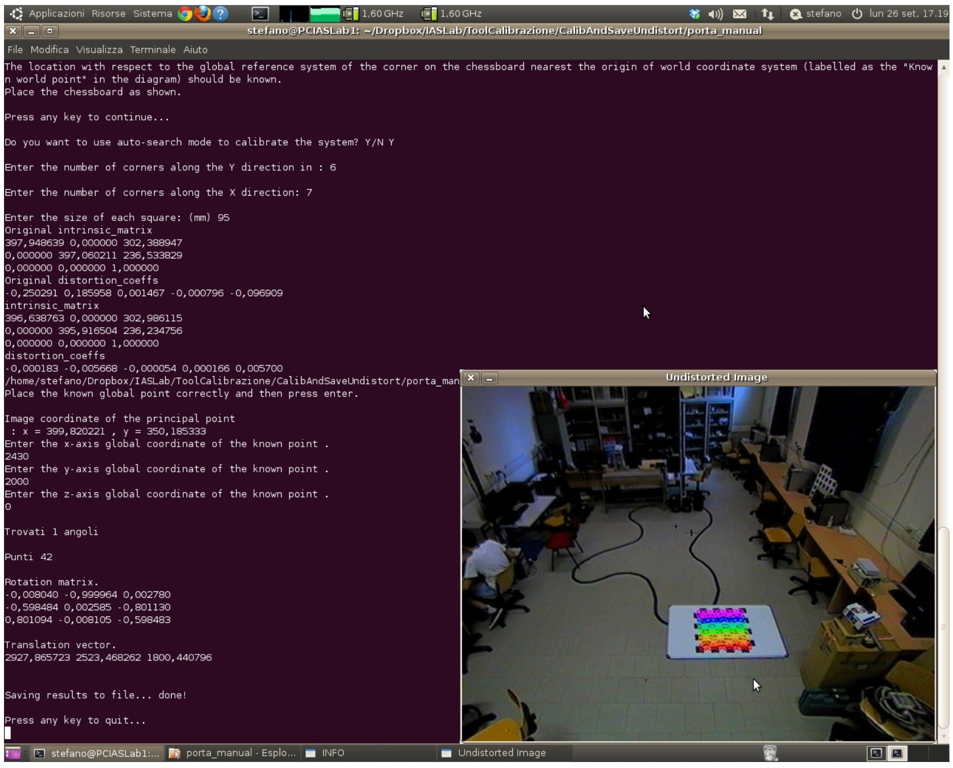
Figure 3.11: Original input image after the undistorting phase.

```
//The function estimates the rotation vector and
//the translation vector.
cvFindExtrinsicCameraParams2(...);

//Converts a rotation vector to a rotation matrix.
cvRodrigues2(...);
```

Manual corners selection If the automatic corners extraction does not work, the possibility to manually select the corners has been implemented. In fact there is a manual selection mode where it is possible to specify the four border corners of the chessboard, so that the tool can extract from there the rotation matrix and the translation vector. The four corners are inserted in the image with a click on the correspondent position in the image windows (green X in figure 3.13). After that, the tool refines their location with the *cvFindCornerSubPix* function (blue X in figure 3.13).

The figure 3.14 shows a plot of a camera position where the extrinsic parameters are found in both ways. The result obtained by the manual selection mode is very similar to the automatic extraction mode. This means that it is a good way to perform the calibration when the automatic extraction mode does not work.

The chessboard problems The function *cvFindChessboardCorners* attempts to determine whether the input image is a view of the chessboard pattern and locate the internal chessboard corners. It was necessary to increase the size of the squares of the chessboard, because the great distance from the floor of the camera did not allow automatic detection of corners. This board with 7 squares in height and 6 square about width has been need, where each square had side of 95 millimeters. Furthermore, a thick white border was left around the chessboard, in order to impose its detectability.


Applicazioni Risorse Sistema
1,60 GHz
1,60 GHz
stefano
lun 26 set, 17.19
stefano@PCIASLab1: ~/Dropbox/IASLab/ToolCalibrazione/CalibAndSaveUndistort/porta_manual
File Modifica Visualizza Terminale Aiuto
Undistorted Image
stefano@PCIASLab1:...
porta_manual - Esplo...
INFO
Undistorted Image

```
The location with respect to the global reference system of the corner on the chessboard nearest the origin of world coordinate system (labelled as the "Know
n world point" in the diagram) should be known.
Place the chessboard as shown.

Press any key to continue...

Do you want to use auto-search mode to calibrate the system? Y/N Y

Enter the number of corners along the Y direction in : 6

Enter the number of corners along the X direction: 7

Enter the size of each square: (mm) 95
Original intrinsic_matrix
397,948639 0,000000 302,388947
0,000000 397,060211 236,533829
0,000000 0,000000 1,000000
Original distortion_coeffs
-0,250291 0,185958 0,001467 -0,000796 -0,096909
intrinsic_matrix
396,638763 0,000000 302,986115
0,000000 395,916504 236,234756
0,000000 0,000000 1,000000
distortion_coeffs
-0,000183 -0,005668 -0,000054 0,000166 0,005700
/home/stefano/Dropbox/IASLab/ToolCalibrazione/CalibAndSaveUndistort/porta_man
Place the known global point correctly and then press enter.

Image coordinate of the principal point
 : x = 399,820221 , y = 350,185333
Enter the x-axis global coordinate of the known point .
2430
Enter the y-axis global coordinate of the known point .
2000
Enter the z-axis global coordinate of the known point .
0

Trovati 1 angoli

Punti 42

Rotation matrix.
-0,008040 -0,999964 0,002780
-0,598484 0,002585 -0,801130
0,801094 -0,008105 -0,598483

Translation vector.
2927,865723 2523,468262 1800,440796


Saving results to file... done!

Press any key to quit...
```

Figure 3.12: Screen of the results window

The double calibration of the intrinsic parameters After the image loading phase, the toolbox loads two different kind of intrinsic parameters. These intrinsic parameters comes from two different calibrations:

- first calibration with original images;
- second calibration with the images that are undistorted using the previous parameters.

The first set of parameters is used only to undistort the input image. In fact it is characterized by the distortion of the camera lens. The second set is used to estimate the new center of the image after the undistorting phase. The distortion component estimated in the second set of parameters should be near zero as the source images for this calibration have been already undistorted. Furthermore, the focal lengths estimated with these two calibrations should be very similar.

3.1.3 Conclusions

The problems encountered during the development of the tool for extrinsic parameters extraction required a big effort to be resolved. At first the size of the chessboard was the critical point. It must be not too big and cumbersome but at the same time it must contain a high number of squares with the side large enough to allow the automatic extraction of corners. The second important problem was image quality. In fact the camera has not a high resolution and the environment light conditions were not optimal, making even harder locating the chessboard

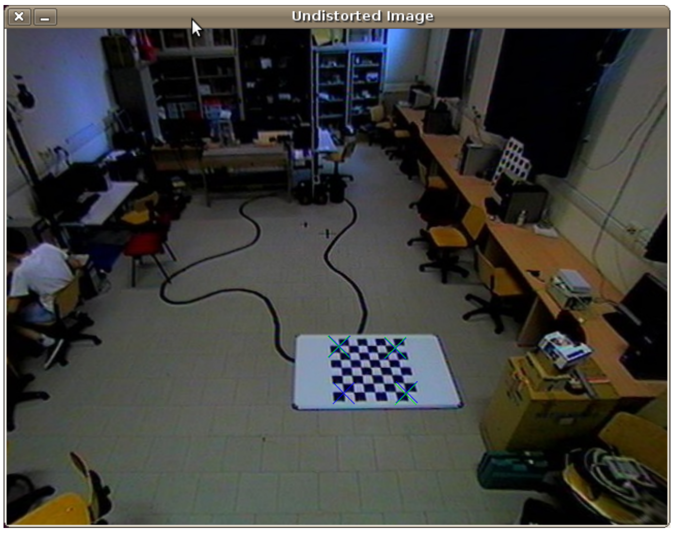


Figure 3.13: Manual selection mode

corners. At last, but not least, there is the problem encountered about the interpretation of the obtained rotation matrix and translations vector. In the three-dimensional space is more complex to extract from the rotation matrix the final result of the rotation. It is possible to verify the results only with tools such as Matlab. As shown in figure 3.15 the Matlab plot of the three cameras inside the test room fully reflects their real location in the environment and their orientation. The estimated coordinate systems of each camera are valid also from a metric point of view.

3.2 Omnidirectional Camera

In this section, is provided a description of the omnidirectional camera model. The omnidirectional cameras combines the use of a mirrors and of a prospective cameras. This particular type of camera presents a 360-degree field of view in the horizontal plane. Generally it is possible divided two main categories of omnidirectional cameras : *non central cameras* and *central cameras*. The particularity of the first type is that the optical rays coming from the camera and they are reflected by the mirror surface do not intersect into a unique point (figure 3.16(a)). The central cameras instead (figure 3.16(b)), are systems where every optical ray, which is reflected by the mirror surface, intersects into a unique point, called *single efective viewpoint*. A catadioptric systems is composed by lenses and mirrors and it can be realized using a hyperbolic mirror. In this way all the direct rays that hit the mirror (who meet the fire about hyperbole (F')) converge in the focal point of the prospective camera (F).

Our model is based on the following assumptions [3]:

- The mirror camera system is a central system, thus, there exist a point in the mirror where every reflected ray intersects in. This point is considered the axis origin of the camera coordinate system XYZ.

[3]the same of https://sites.google.com/site/scarabotix/ocamcalib-toolbox

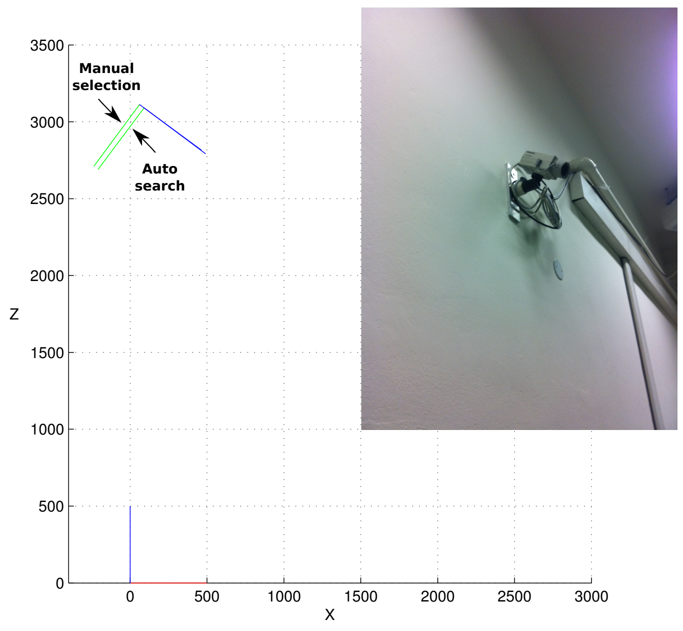


Figure 3.14: It is possible to see how the result obtained by the manual selection mode is very similar to the automatic extraction mode.

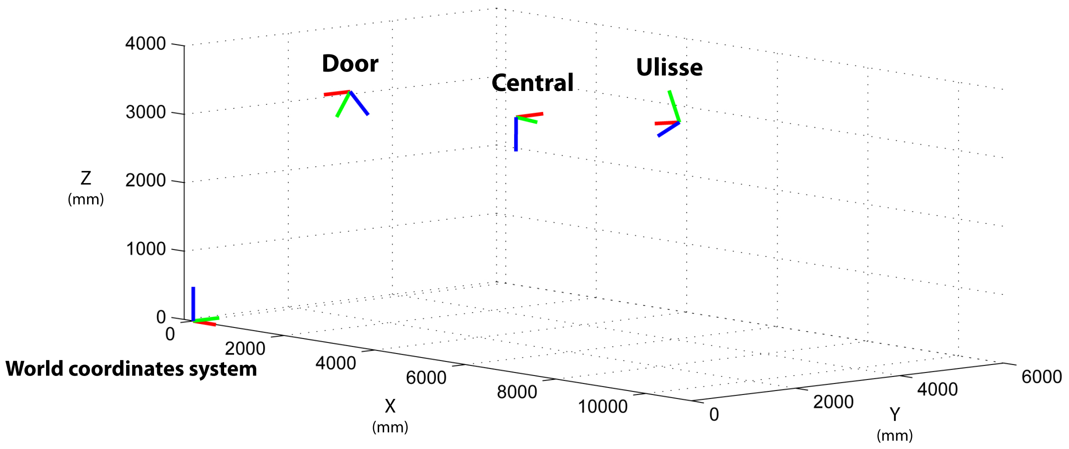
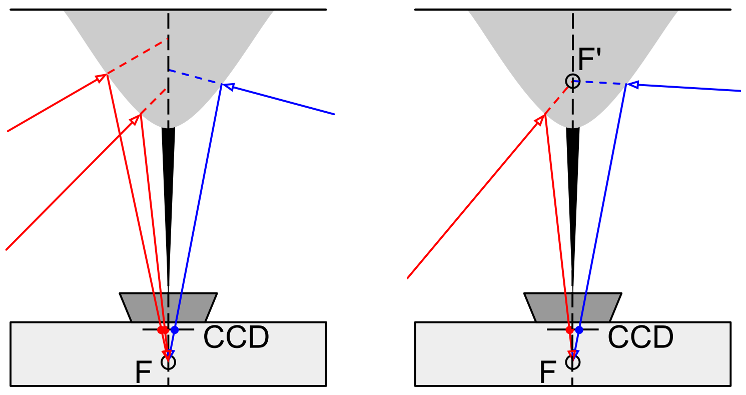


Figure 3.15: Plot of the three cameras inside the test room

- The camera and mirror axes are well aligned: only small deviations of the rotation are considered into the model.
- The mirror is rotationally symmetrical with respect to its axis.
- The lens distortion of the camera is not considered in the model. Camera lens distortion has not been included because omnidirectional cameras using mirrors usually need large focal length to focalize the image on the mirror. Thus, lens distortion can be really neglected.

The omnidirectional camera model allows the transition from a given 2D pixel point $p = (u, v)$



(a) Non central system (b) Central system

Figure 3.16: Omnidirectional camera type.

to the 3D vector $p' = (x, y, z)$ emanating from the mirror effective viewpoint (figure 3.17). The two coordinates x and y are proportional, respectively, to u and v. The z-coordinate, however, depends on the distance of the point p from the center of the camera:

$$z = f(\rho) = f(\sqrt{u^2 + v^2})$$

Now it is necessary to set the $f(\rho)$ functions. The model describes the function $f(\rho)$ in terms of a polynomial, whose coefficients are the calibration parameters to be estimated. That is:

$$f(\rho) = a_0 + a_1\rho + a_2\rho^2 + a_3\rho^3 + ...$$

By the way, the vector $p' = (x, y, f(\rho))$ is not sufficient for fully determine the real position in world $P = (X, Y, Z)$ of the point p (figure 3.18). Due the next section describes how to correlate the point $p = (u, v)$ with the correspondent point $P = (X, Y, Z)$ in the world.

3.2.1 Calibration

In this section, we will see how to find the main parameters that describe the omnidirectional camera model:

Intrinsic parameters

- Principal point (usually the image center) $[c_x, c_y]$
- The coefficients a_i of the $f(\rho) = a_0 + a_1\rho + a_2\rho^2 + a_3\rho^3 + ...$

The OcamCalib Toolbox for Matlab[4], developed by Davide Scaramuzza [5], allows us to extract these intrinsic parameters.

We will see also how to correlate the point $p = (u, v)$ with the correspondent point $P = (X, Y, Z)$ in the world (figure 3.18):

[4]https://sites.google.com/site/scarabotix/ocamcalib-toolbox
[5]https://sites.google.com/site/scarabotix

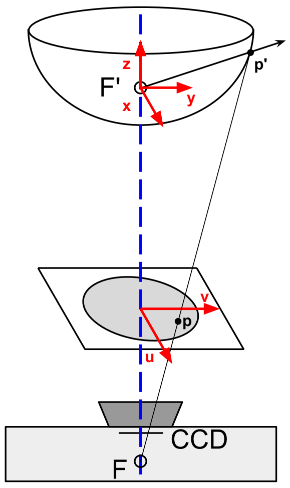


Figure 3.17: Omnidirectional camera system

Extrinsic parameters

– Homography: in literature does not found a satisfactory and easy to use method to correlate the floor plan with the image plane, so it was chosen to use the homography concept to do this. After, it was developed a procedure to relate the image plane point to the correspondents world floor point using the homography

We will use a Matlab function, developed by Peter Kovesi[6], to estimate this parameters.

Intrinsic parameters

To estimate the intrinsic parameters, it was used the OcamCalib Toolbox for Matlab[7], developed by Davide Scaramuzza. The Toolbox implements the procedure initially described in the paper [5] and later extended in [6] and [7]. It allows a user to easily and quickly calibrate the omnidirectional camera through two steps. First, it requires that the user collect a few pictures of a chessboard shown at different positions and orientations. Then, it automatically extracts the corner points of the chessboard from the parameters that characterize it (the number of squares present along the X and Y direction, and the size of the square along the X and Y directions).

After that, for the calibration, it is necessary to chose which polynomial order use for the $f(\rho)$ function. Several experiments on different camera models showed that a polynomial order equal to four gives the best results. In fact the average error with the third degree of the $f(\rho)$ after the refinement phase is `0.521318`, while the fourth is `0.486544`. The output of the calibration phase are the main parameters that describe the omnidirectional camera model:

- *ocam_model.ss* contains the polynomial coefficients a_i of the $f(\rho) = a_0 + a_1\rho + a_2\rho^2 + a_3\rho^3 + ...$;
- *ocam_model.xc* is the row coordinate of the center of the omnidirectional image;
- *ocam_model.yc* is the column coordinate of the center of the omnidirectional image;

[6]http://www.csse.uwa.edu.au/ pk/
[7]https://sites.google.com/site/scarabotix/ocamcalib-toolbox

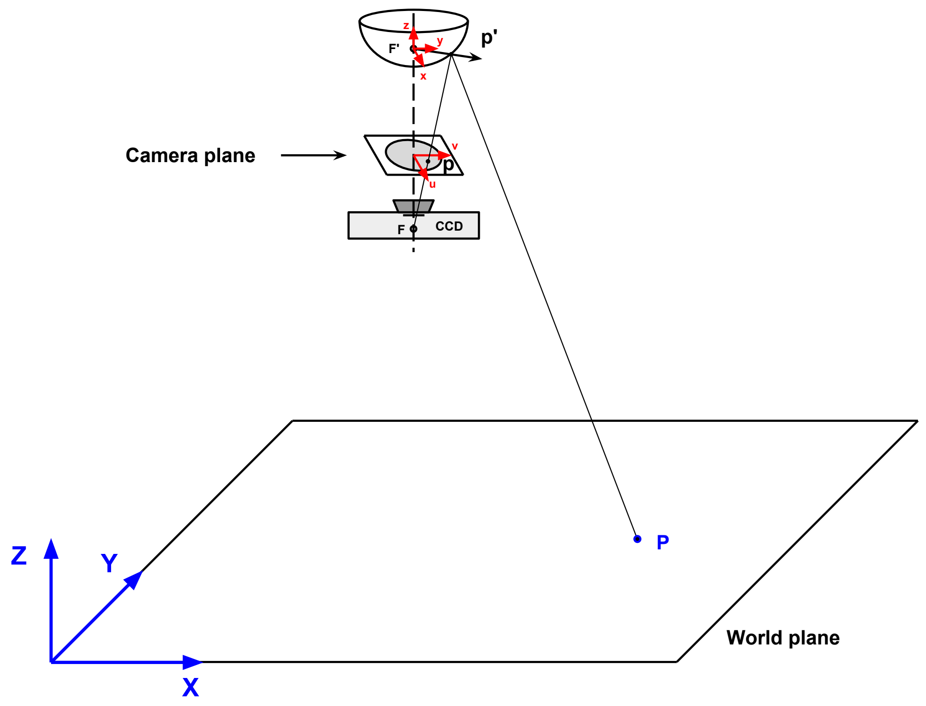


Figure 3.18: Omnidirectional camera placed in the real world

- *ocam_model.width* width of the images under process;
- *ocam_model.height* height of the images under process;
- *ocam_model.c*, *ocam_model.d*, *ocam_model.e* are the affine transformation parameters.

These three last parameters are necessary to model errors and misalignments digitizing artifacts. Finally all this parameters are saved in *ocam_model* Matlab workspace and are also export in a .txt file (*calib_results.txt*).

Extrinsic parameters

In the computer vision field, the *homography* is an invertible transformation as a projective mapping from one plane to another. In this case the first plane correspond to the world floor, while the second is refer to the omnidirectional image plane.

The floor plane in the world coordinate system can be considered as a two dimensional plane as the third dimension Z happens to be 0. If the Z is set as zero then the perspective projection matrix can be simplified as follows to represent a mapping between the two dimensional floor plane and the image camera plane.

$$H = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}$$

H is the 3-by-3 matrix called *homography matrix*. This matrix is independent from the camera model. In fact it is calculated from the projection $p' = (x, y, z)$ onto the unit sphere of the image point in pixel $p = (u, v)$ and the homogeneous coordinates of the same point in world coordinates $P = (X, Y, 1)$. This means that if we change the omnidirectional camera with another and we place it in the very position, H remained the same. Obviously the parameters about the camera model change.

To switch from the coordinates in pixel of a point on the image plane (u, v) to the vector that represent the same point in a unit sphere (x, y, z) there is the `cam2world` function. Viceversa, to project a point that rappresent the vector in the unit sphere (x, y, z) onto the image plane in pixel (u, v) there is the `world2cam` function. All this functions depend by the omnicamera intrinsic parametrers and they are developed by Davide Scaramuzza.

The *homography matrix* is determined with the `homography2d.m` Matlab function developed by Peter Kovesi[8]. From four or more points in a plane about the omnicamera image coordinates projected on the unit sphere and the correspondent points in the world system, this function computes the 3-by-3 homography matrix. The `homography2d.m` Matlab function code follows the normalized direct linear transformation algorithm (DLT algorithm) given by Hartley and Zisserman in [8].

```
% Back-projects the pixel point p = (u,v) onto the unit sphere
% p' = (x, y, z).

p' = cam2world(p, ocam_model)

% Calculating the homography matrix using the function developed
% by Peter Kovesi.

H = homography2d(p', P)
```

Also in this case, the idea is to place a chessboard in a specific point of the floor and through the function *Extract grid corners*, present inside Scaramuzza Matlab toolbox, extract the corners in the image plane coordinates. Due to the low quality resolution of the camera and the great distance from the floor, it was necessary to use a "big" chessboard. This chessboard was built specifically for this purpose. It was designed with 3 squares in height and 4 square about width. Each square has a side that measures 250 millimeters and there is a thick white border all around. The square side dimension has been fit to optimized the corners' extraction.

The homography matrix H relates the positions of the unit sphere points to the points on the floor by the following equations:

$$P = Hp' \quad \rightarrow \quad \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$p' = H^{-1}P \quad \rightarrow \quad \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix}^{-1} \begin{bmatrix} X \\ Y \\ 1 \end{bmatrix}$$

The next step is how to relate the image plane point to the correspondents world floor point.

[8]http://www.csse.uwa.edu.au/ pk/Research/MatlabFns/Projective/homography2d.m

3.2.2 From world to camera and viceversa

From the image plane to the floor plane

This first transformation permits to map a single point in pixel from the image plane to the correspondent point in meters on the floor. The correct procedure for this transformation is the following:

- take in input the image point $p = (u, v)$ where u,v are the pixel coordinates;
- transform p in the correspondent point in the unit sphere $p' = (x, y, z)$ with `cam2world` function;
- extract the point $P' = (X', Y', Z')$ as following: $P' = Hp'$;
- normalizes P' in order to obtain homogeneous coordinates $\frac{P'}{P'(3)} = (\frac{X'}{Z'}, \frac{Y'}{Z'}, \frac{Z'}{Z'}) = (X, Y, 1)$ where X and Y are the real floor coordinates in meters and $P = (X, Y, Z) = (X, Y, 0)$.

From the floor plane to the image plane

This transformation permits to map a single point in meters on the floor to the correspondent point in pixel from the image plane. The correct procedure for this transformation is the following:

- take in input $P = (X, Y, Z)$ that is the floor point in meters. It is possible to write $P = (X, Y, 0)$;
- transform P in the correspondent homogeneous coordinates $P' = (X, Y, 1)$;
- extract the vector $p'' = (x'', y'', z'')$ as following: $p'' = H^{-1}P'$;
- extract the point vector p' refers to a unit sphere from the vector defined by p''. Note that unit sphere has this property: $X^2 + Y^2 + Z^2 = 1$. For a generic sphere is true this property: $X^2 + Y^2 + Z^2 = r^2$ where r is the radius of the sphere. So to derive p' from p'' it is necessary to divide p'' by its r in this way:

$$r^2 = x''^2 + y''^2 + z''^2$$
$$\pm r = \sqrt{r^2}$$
$$p' = (x', y', z') = \left(\frac{x''}{-r}, \frac{y''}{-r}, \frac{z''}{-r}\right)$$

- project p' onto the image plane and returns the pixel coordinates: $p = (u, v)$ with `world2cam` function. Note that in the previous formula it was used the negative radius $(-r)$ to calculate p' as using the positive sign results in "Not a number" error of the `world2cam` function.

3.2.3 Conclusions

The problems encountered during the extraction of the extrinsic parameters essentially reflect those found for the pinhole camera. Also in this case the chessboard size and the image quality did not allow the detection of corners. To overcome this problem was built a "big" chessboard with the square that has a side of 250 millimeters. The tests carry out about the projection of a single world point to image plane and viceversa, demonstrate that all parameters (both intrinsic and extrinsic) lead to a correct projection of points as shown in figure 3.19.

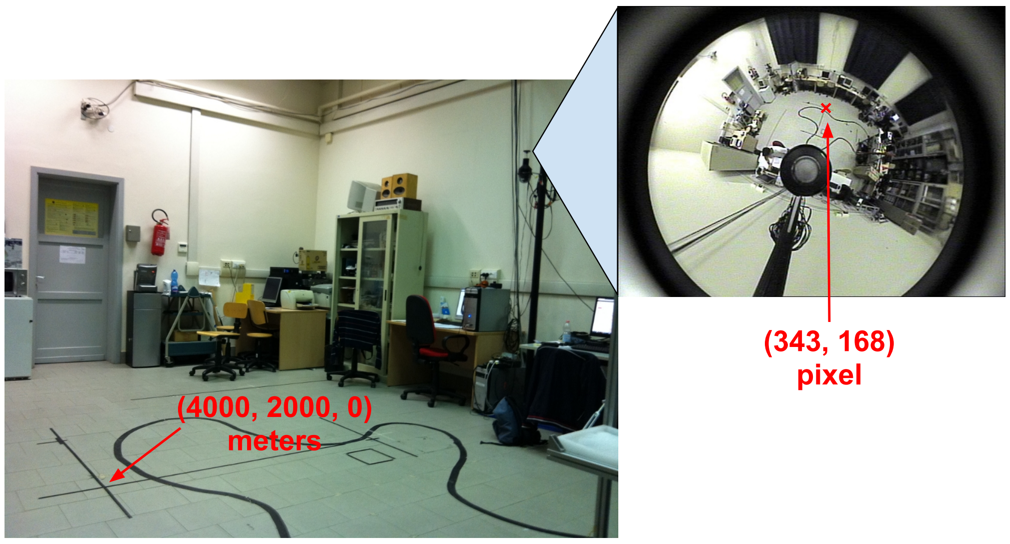


Figure 3.19: View of the same point in the omnicamera image plane and on the floor

Chapter 4

Application: Click and View

This chapter provides a description of the multi-camera tool developed in a distributed vision system. A set of heterogeneous cameras are located indoors and are installed into the Intelligent Autonomous Systems (IAS) laboratory in Padua (figure 4.1). This system allows to gather all video streaming coming from each camera, giving a complete picture of the environment. With this tool it is possible to map with precision every point on the floor onto each camera window and also derive the coordinates in meters of the same point projected on the floor. To do this it is necessary to use the intrinsic and extrinsic parameters of each camera. In this way this application is also the right way to test and verify the previous phase of calibration. The system developed can be used in any environment where there is a need to know precise items position or someone inside and on the floor. With the possibility of using a distributed system of cameras (and thus have a more point of view) it is possible to obtain more detailed information.

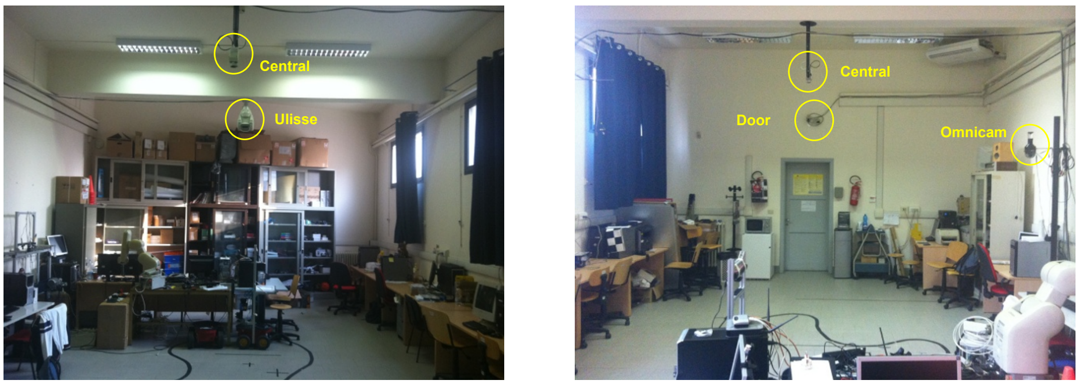


Figure 4.1: Pictures of IAS laboratory.

In the environment are positioned four cameras (figure 4.2):

- the first is located over the door. This is a fixed pinhole camera and it is called *Door*
- the second is located in the center and it frames the floor. Also this is a fixed pinhole camera and it is called *Central*

- the third is on the side of the *Central*. It is a fixed omnidirectional camera. Its name is *Omnicam*
- the last is positioned towards the end of the room. This is a PTZ camera which is a particular type of pinhole camera with remote directional and zoom control. Its name is *Ulisse*.

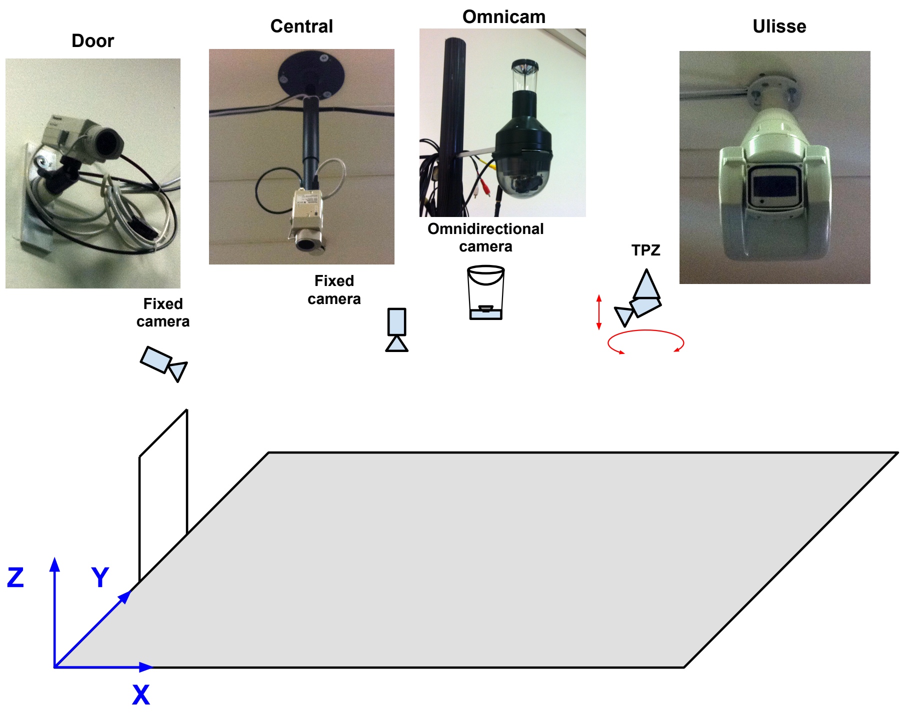


Figure 4.2: Scheme of the IAS laboratory and cameras inside

The tool is based on the use of NMM middleware introduced in the Chapter 2. In this way it is possible to use all four cameras that are connected on different computers (figure 4.3).

The *Click and View* tool displays four windows, one for each camera. In each window it is possible to click in a single point on the floor. This point is projected and shown in each of the other fixed camera windows. The PTZ camera instead gives the opportunity to rotate around the vertical and horizontal axes. For this reason it rotates and frames the same point on the floor shown in the center of the correspondent windows (figure 4.4).

For the development of the tool it was necessary to create several NMM nodes to manage the video stream, cameras, click on the windows and communication between different devices.

4.1 The nodes

In this section, we will describe the nodes developed for the *Click and View* tool used in the distributed vision system. These nodes must be integrated in NMM middleware and are developed in C++ language. Here is a brief overview of the nodes involved in the tool and then a more accurate description of the main nodes created:

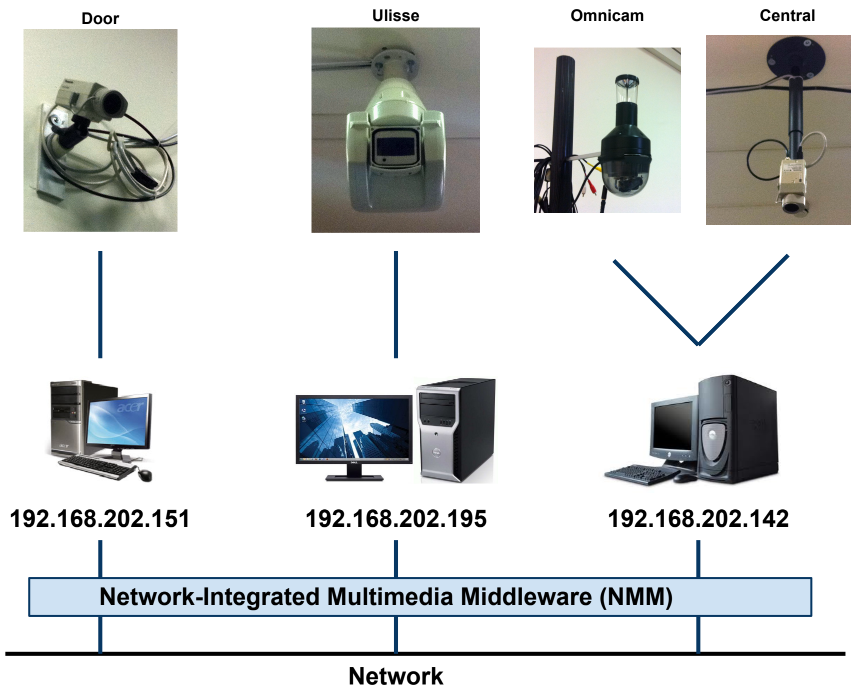


Figure 4.3: Distributed system of cameras in IAS Lab.

- `TVCardReadNode2`:

  it is a source node that allows to manage the video device such as a camera. By the method *setDevice* it is possible to select which frame grabber use in input (e.g. */dev/video0*)

- `YUVtoRGBConverterNode`

  this node converts an incoming YUV image to a BGR image (blue, green, red and ignored channel, in this order)

- `ImageRotationNode`

  this node rotates the input image by an angle α. This angle is the argument of the method *setAngle* (e.g. *setAngle(α)*).

- `VideoScalingNode`

  this node scales the input image about height and width by an inserted resolution. The methods to set this are *setXResolution* and *setYResolution*.

- `VideoUndistortingNode`

  this node undistorts the input stream defected by distortion. By the method *setConfig* it receives the name of the camera and it loads the correct parameters for the undistortion

- `MouseDetectionNode`

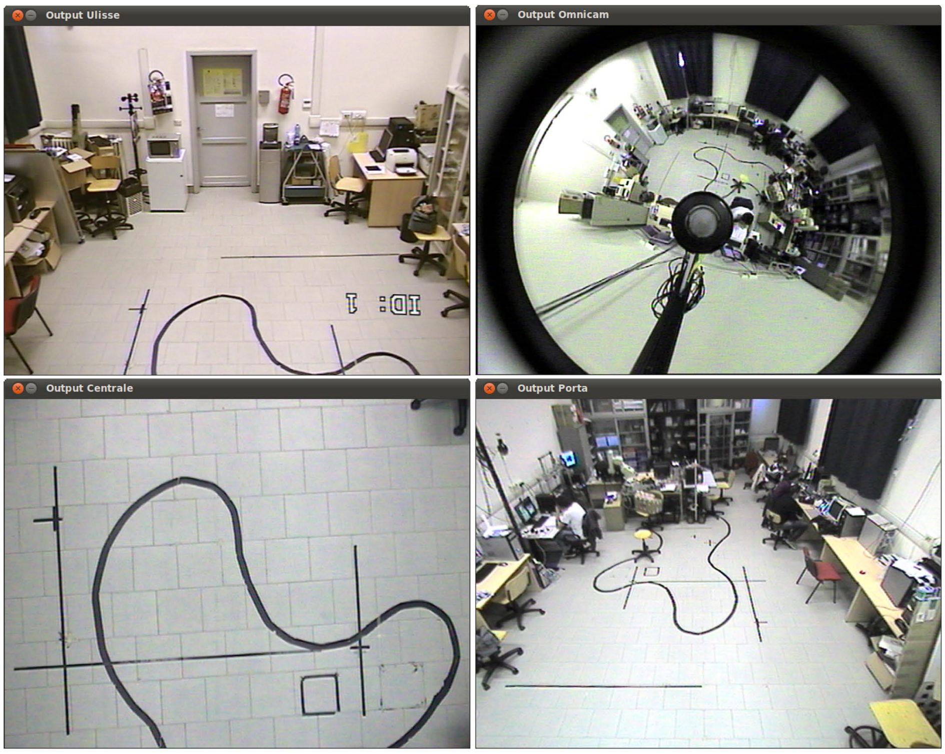


Figure 4.4: Screen of the Click And View application.

this node manages the stream in input, the click on the camera window and the event from and to other camera inside the distributed vision system. Also for this, there is the method *setConfig* that receives the name of the camera and loads the correct parameters

- `VideoEventReceiverNode`

  this node manages the communication between different device. By the method *setJacksNum* it is possible to set the number of devices to be connected to each other.

4.1.1 VideoUndistortingNode

This node was specially created for undistorting the video stream from all cameras that are based on the pinhole camera model. It is a *GenericFilterNode*. A filter node has exactly one input stream and one output stream, both named "default", and the output format is always identical to the input format. It changes only the video data but not the format data. For this node is used the OpenCV library. In fact after the initial calibration phase, the intrinsic parameters are available, and particularly the distortion parameters, of each camera. With this parameters and the `cv::undistort` function it is possible to undistort the original video stream of the camera, defected by radial and tangential distortion (figure 4.5). In input the node receives the name of the camera to undistort the video stream by the method *setConfig* and it loads the intrinsic parameters to be used. Finally, the node computes in real time the video stream transformation frame by frame (e.g. in figure 4.6).

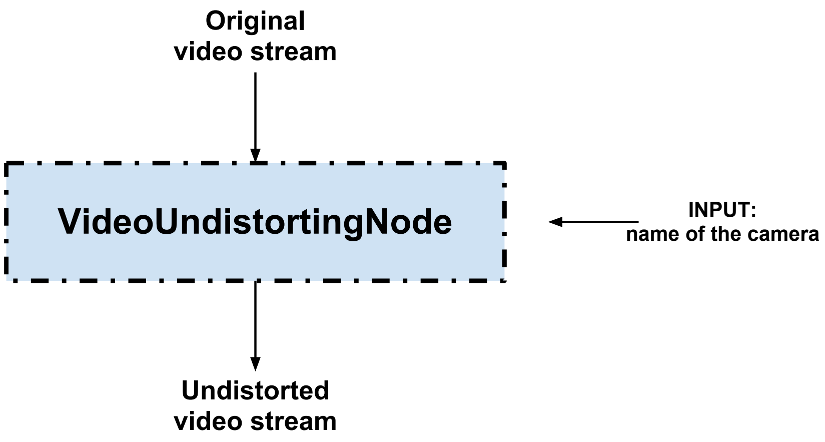
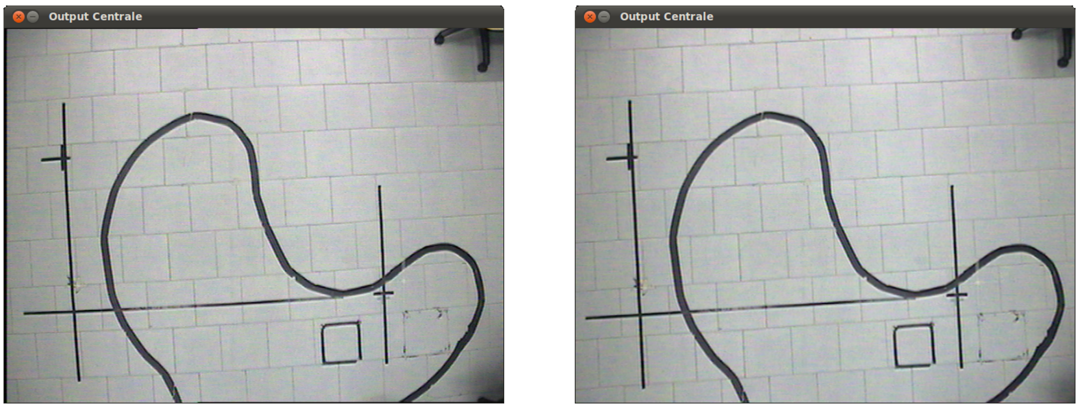


Figure 4.5: Video undistorting node.

```
//Extracts from the video stream buffer a single frame
in_frame = in buffer

//Undistorts the frame with the intrinsic parameters
cv::undistort(in_frame, out_frame, intrinsic, distortion)

//Returns the undistorted frame
return out_buffer
```



(a) The original image (b) The same image after this node

Figure 4.6: VideoUndistortingNode.

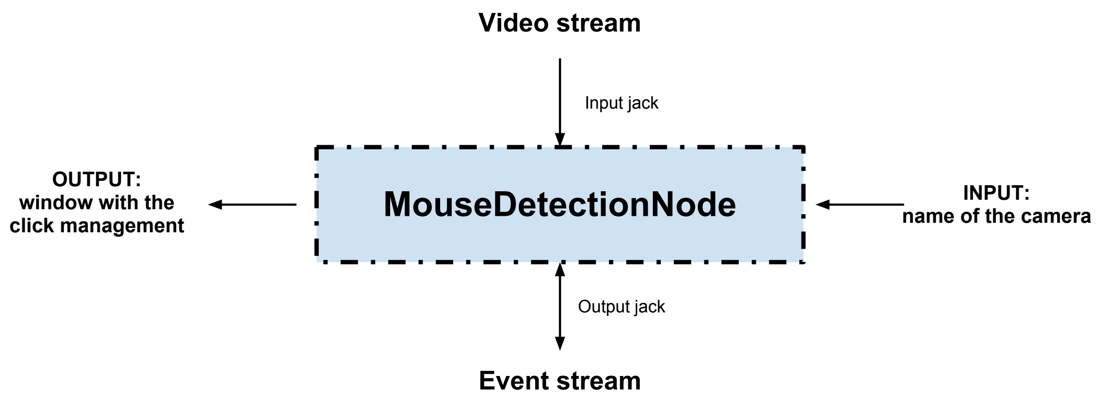


Figure 4.7: Mouse detection node.

4.1.2 MouseDetectionNode

The MouseDetectionNode manages the input video stream, the click on the camera window and also the event. It is a *GenericProcessorNode*. A processor node has exactly one input stream and one output stream. The input stream manages the video and the output stream manages the event. At first, by the method *setConfig* that allows enter the name, the node creates a camera object of the same type of the relative camera model. It loads its intrinsic parameters and the extrinsic ones (rotation matrix and translation vector) about the camera with respect to the floor of the IAS laboratory. If the camera type is TPZ, the node sends also to the camera the control command (tilt and pan angle) for moving it.

```
// Creates the camera object from the inserted name
if(cameraName == "Porta" || cameraName == "Centrale")
{
  camera = new Pinhole(cameraName);
}
else if(cameraName == "Ulisse")
{
  camera = new Ulisse(cameraName);
}else if(cameraName == "Omnicam")
{
  camera = new Omnicam(cameraName);
}
```

```
// Loads intrinsic matrix
manager.loadMatrix(pack + "/Intrinsics" + name + ".xml");
intrinsic = manager.getMatrix("Intrinsics");

// Loads rotation matrix
manager.loadMatrix(pack + "/RotationMatrix" + name + ".xml");
rotationMatrix = manager.getMatrix("RotationMatrix");

// Loads translation vector
manager.loadMatrix(pack + "/TranslationVector" + name + ".xml");
translationVector = manager.getMatrix("TranslationVector");
```

The MouseDetectionNode manages also the incoming video stream. It uses the OpenCV library to compute the stream in input and provides to create the window to show frame by frame the video of the camera.

```
// Creates the output window
cvNamedWindow(...);

// Shows the single frame
cvShowImage(...);
```

In this window it is possible to click and select a single point. This requires the creation of a mouse callback. With `cvSetMouseCallback` it is possible to do this.

```
// Manages the click on the windows
cvSetMouseCallback(..., my_mouse_callback, ...);
```

The callback is the function `my_mouse_callback(int event, int x, int y, int flags, void* param)` where x and y are the mouse click position and event is a code representing what mouse action occurs.

```
// Mouse callback function
void my_mouse_callback(int event, int x, int y, int flags, void
    *param){

  ...

  switch(event)
  // Detects the left click of the mouse
  case CV_EVENT_LBUTTONDOWN:{

    ...

  }
}
```

After the detection of mouse click, the node must send to the other cameras the coordinates of the floor point clicked. To perform this, first the MouseDetectionNode converts the image coordinates in pixel to the correspondent world floor coordinates and then it sends them to the VideoEventReceiverNode as an event. This last node provides to send the world coordinates to other cameras.

```
// Tranforms the image plane coordinates (x, y) in
// the (X, Y, Z) world floor cordinates
cv::Mat world = (camera->cam2world(x_pixel, y_pixel, ...));
```

The event is created with a particular function of NMM and it is sent on the output jack.

```
// Creates the event with world coordinates
CEvent* ce = new CEvent();

...

float wX = world.at<float>(0, 0);
float wY = world.at<float>(1, 0);
float wZ = world.at<float>(2, 0);

TInValue<float>* worldX = new TInValue<float>(wX);
TInValue<float>* worldY = new TInValue<float>(wY);
TInValue<float>* worldZ = new TInValue<float>(wZ);

// Creates the event with MOUSE_CLICK_CAMERA name
Event* e = new Event("MOUSE_CLICK_CAMERA", 3);
e->setValue(0,worldX);
e->setValue(1,worldY);
e->setValue(2,worldZ);
ce->insertEvent(e);

// Sends the event to the output jack
sendEventDownstr(ce);
```

The output jack of the node remains also in listen state. The reason is that the node must be ready to project in its own window a point sent by another camera. In this case the MouseDetectionNode receives the world coordinates of the point as an event, computes this point with the `world2cam` function and plots it in the image plane on the windows. If the camera type is TPZ, the node sends also to the camera the control command for moving it.

```
// MouseDetectionNode event listener
// Captures WORLD_COORDINATES event
registerEventListener("WORLD_COORDINATES",
  new TEDObject3<EventListener,
  float, float, float, Result(EventListener:: * )
  (const float, const float, const float),Result>
  (&m_event_listener, &EventListener::receivedWorldCord));

// Computes the received world point
Result EventListener::receivedWorldCord(const float worldX, const
    float worldY, const float worldZ)
{
  ...

  cv::Mat cam = (cameraListener->world2cam(worldX, worldY, worldZ));

  x_event = (int)cam.at<float>(0, 0);
  y_event = (int)cam.at<float>(1, 0);

  ...
}
```

4.1.3 VideoEventReceiverNode

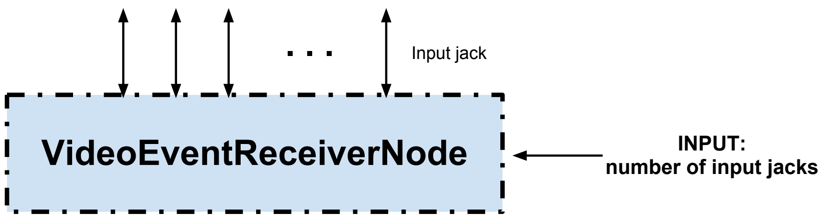


Figure 4.8: Video event receiver node.

This node manages the sending of the events between different cameras. It is a *GenericMultiplexerNode*. A multiplexer node has multiple input stream and exactly one output stream, which is named "default". It use only the input jacks that are called in this way: *jack*`N` where `N` is the number of the input jack. At first by the *setJacksNum* function it sets the number of the jacks that correspond to the number of connected device. The default setting is *setJacksNum*=1. Each jack is listening for an event.

```
// VideoEventRegisterNode event listener
VideoEventReceiverNode::VideoEventReceiverNode(const char* name )
: GenericMultiplexerNode(name),
num_Jacks(1)
{
  ...

  // Captures the MOUSE_CLICK_CAMERA event
  registerEventListener("MOUSE_CLICK_CAMERA",
    new TEDObject3<EventListener, float, float, float,
    Result(EventListener:: * )
    (const float, const float, const float),Result>
    (&m_event_listener, &EventListener::sendToOtherCamera));

}
```

When an event arrives, the nodes provides to forward it to all the input jacks by `sendToOtherCamera` function. The event contains the world coordinates of the point clicked in one of the camera window.

```
// Creates the event to send
Result EventListener::sendToOtherCamera(const float worldX, const
    float worldY, const float worldZ)
{
  ...

  CEvent* ce = new CEvent();

  TInValue<float>* wX = new TInValue<float>(worldX);
  TInValue<float>* wY = new TInValue<float>(worldY);
  TInValue<float>* wZ = new TInValue<float>(worldZ);

  // Creates the event with WORLD_COORDINATES name
  Event* e = new Event("WORLD_COORDINATES", 3);
  e->setValue(0,wX);
  e->setValue(1,wY);
    e->setValue(2,wZ);
    ce->insertEvent(e);

  // Calls the sender function
    videoevent->sendEvent(ce);

    return SUCCESS;
}
```

At last the node sends the event created in each of the input jacks.

```
// Sends the event
void VideoEventReceiverNode::sendEvent(CEvent* event)
{
  ...

  // Sends the event in each of the input jacks
  sendEventUpstr(event);

  ...
}
```

4.2 The graph

In this section, we will describe how interconnect the nodes to create the distributed vision system. The nodes can be connected to create a flow graph, where every two connected jacks need to support a "matching" format. In this case it is not possible to use the *clic* application because the graph is too complex (figure 4.9). Therefore it is necessary to create manually the graph through the C++ language.

The application itself runs on the host with *192.168.202.195* ip address. For each camera it is necessary to specify a sequence of nodes to be connected. The node are installed in each computer inserted in the distributed vision system. The *TVCardReadNode2* node runs in the correspondent computer where every camera is connected. Every other node runs in the *192.168.202.195* computer. For this reason in the graph it is necessary to specify the location about the node if this is not running in the application computer.

```
// Settings of the TVCardReadNode2 node about each camera

//Porta
NodeDescription tvcard1("TVCardReadNode2");
tvcard1.setLocation("192.168.202.151");
//Ulisse
NodeDescription tvcard2("TVCardReadNode2");
//Centrale
NodeDescription tvcard3("TVCardReadNode2");
tvcard3.setLocation("192.168.202.142");
//Omni
NodeDescription tvcard4("TVCardReadNode2");
tvcard4.setLocation("192.168.202.142");
```

Moreover, it is necessary to set the port of the computer where the camera is connected.

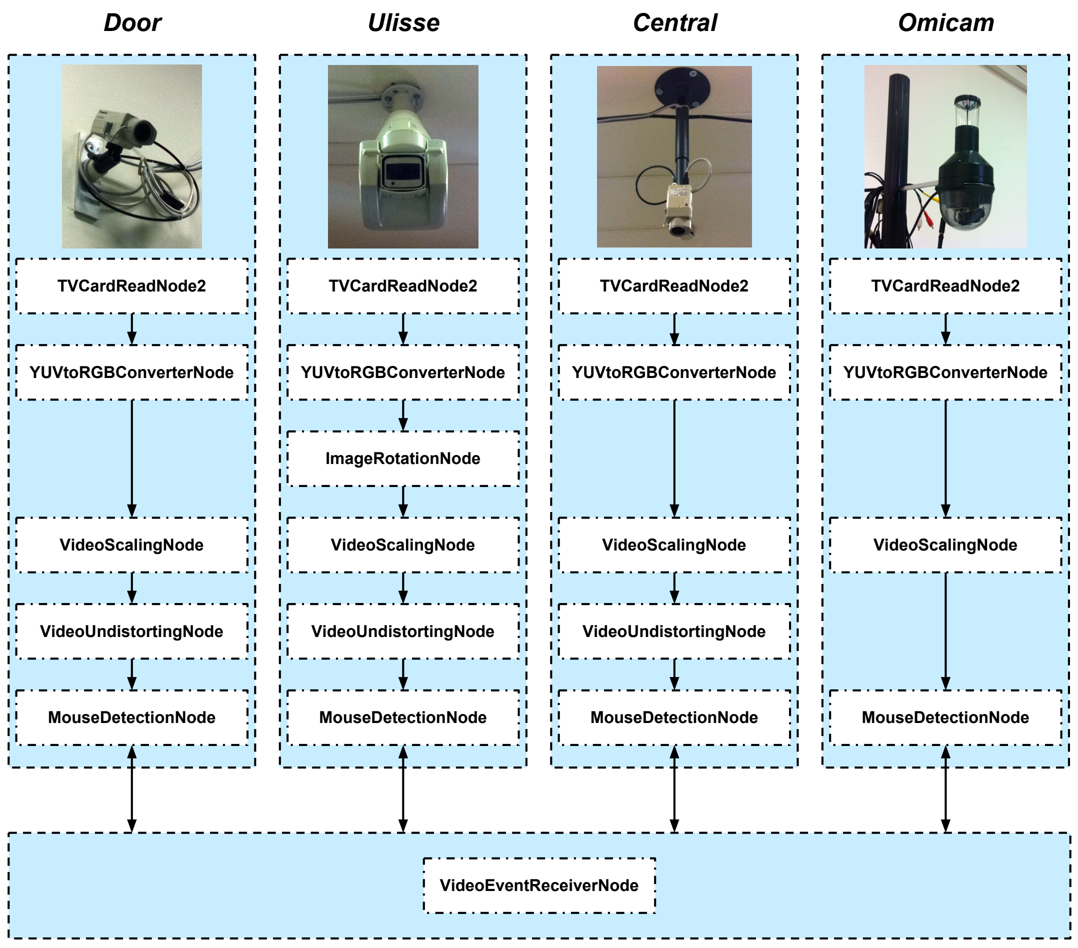


Figure 4.9: The Click and View graph.

```
//Setting the device from tvcard1
//Porta
INode* tv = graph.getINode(tvcard1);
ITVCard_var tv1 (tv->getParentObject()->getCheckedInterface<ITVCard>());
tv1->setDevice("/dev/video0");

// Setting the device from tvcard2
//Ulisse
tv = graph.getINode(tvcard2);
ITVCard_var tv2 (tv->getParentObject()->getCheckedInterface<ITVCard>());
tv2->setDevice("/dev/video0");

//Setting the device from tvcard3
//Centrale
tv = graph.getINode(tvcard3);
ITVCard_var tv3 (tv->getParentObject()->getCheckedInterface<ITVCard>());
tv3->setDevice("/dev/video0");

//Setting the device from tvcard4
//Omni
tv = graph.getINode(tvcard4);
ITVCard_var tv4 (tv->getParentObject()->getCheckedInterface<ITVCard>());
tv4->setDevice("/dev/video1");
```

For each camera is also necessary convert the video format from YUV to RGB. To do this every `TVCardReadNode2` is connected directly with the own `YUVtoRGBConverterNode`. After this, only for the *Ulisse* camera is necessary to rotate the video stream by 180 degrees.

Afterwards, every video stream is scale with the `VideoScalingNode` in the 640 x 480 resolution format and after only for the *Door*, *Central* and *Ulisse* camera it is necessary to undistort the video with `VideoUndistortingNode`.

Finally, every output video stream from this last node enters in the correspondent `MouseDetectionNode`. So there are four output jacks from the four different cameras. All these jacks are connected to the four input jacks about the one `VideoEventReceiverNode` node.

After the nodes' configuration, it is necessary build the graph. This is the right way to connect the node:

```
// Constructions of the graph
GraphDescription graph;

...

//Setting Porta
graph.addEdge(tvcard1, yuv1);
graph.addEdge(yuv1, scale_video_porta);
graph.addEdge(scale_video_porta, und_porta);
graph.addEdge(und_porta, mouse_detection_porta);
graph.addEdge(mouse_detection_porta, video_event, "jack0");

//Setting Ulisse
graph.addEdge(tvcard2, yuv2);
graph.addEdge(yuv2, image_rotation);
graph.addEdge(image_rotation, scale_video_ulisse);
graph.addEdge(scale_video_ulisse, und_ulisse);
graph.addEdge(und_ulisse, mouse_detection_ulisse);
graph.addEdge(mouse_detection_ulisse, video_event, "jack1");

//Setting Centrale
graph.addEdge(tvcard3, yuv3);
graph.addEdge(yuv3, scale_video_centrale);
graph.addEdge(scale_video_centrale, und_centrale);
graph.addEdge(und_centrale, mouse_detection_centrale);
graph.addEdge(mouse_detection_centrale, video_event, "jack2");

//Setting Omni
graph.addEdge(tvcard4, yuv4);
graph.addEdge(yuv4, scale_video_omni);
graph.addEdge(scale_video_omni, mouse_detection_omni);
graph.addEdge(mouse_detection_omni, video_event, "jack3");
```

At last it is necessary to run the graph:

```
// Realize and start graph

graph.realizeGraph();

graph.startGraph();
```

Chapter 5

Application: Click and Go

This chapter provides a description of the *Click and Go* application (figure 5.1). This tool allows to manage the movement of a robot inside the environment controlled by the previous *Click and View* application. The robot is initially positioned in a specific well-know floor point. Using the output screens of the previous application and clicking on a generic floor point, the robot goes from its position to the desired clicked point. In this way, by controlling a resource such as a robot inside the environment, it is possible to extract more information. A camera equipped into the robot allows to have a mobile camera moving inside the environment. So, it is possible to analyze some details that the quality of fixed cameras do not allow. To develop this application, it is necessary to make possible the cooperation of two different systems:

- the Network-Integrated Multimedia Middleware (NMM), the middleware that manages the communication between different camera in the *Click and View* application;
- the Robot Operating System[1] (ROS), a framework for robot software development.

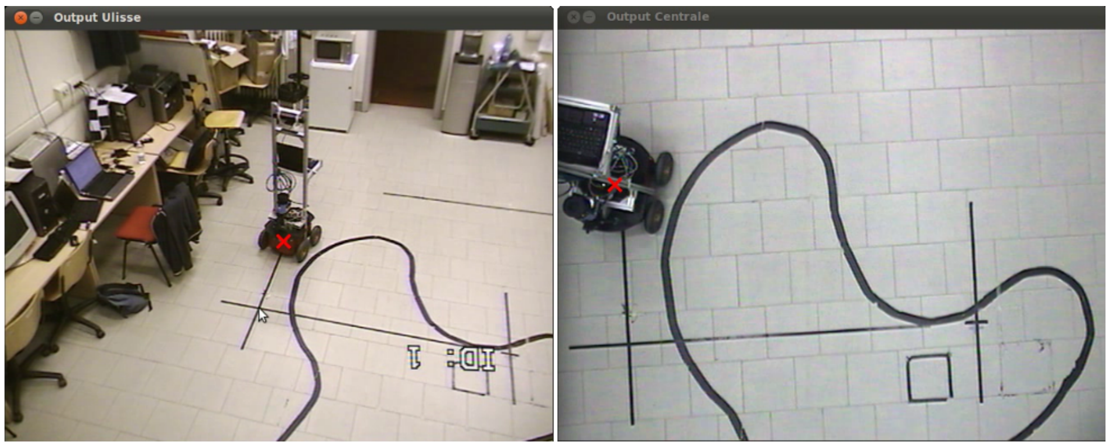


Figure 5.1: Screen of the Click And Go application

[1]http://www.ros.org/wiki/

5.1 Components

5.1.1 Robot Operating System (ROS)

ROS (Robot Operating System) is a framework for robot software development, providing operating system-like functionality on top of a heterogeneous computer cluster. ROS provides standard operating system services such as hardware abstraction, low-level device control, implementation of commonly-used functionalities, message-passing between processes and package management. It is based on a graph architecture where processing takes place in nodes that may receive, post and multiplex sensor, control, state, planning, actuator and other messages.

The graph architecture As already mentioned ROS is based on a graph architecture. The graph is the peer-to-peer network of ROS processes that share data. Below is provided a quick overview of the graph concepts:

- a **node** is an executable program that uses ROS to communicate with other nodes;
- nodes can publish messages to a **topic** as well subscribe to a **topic** to receive messages;
- a **messages** is a ROS data type used when subscribing or publishing to a topic;
- **master** is service the name for ROS (i.e. helps nodes to find each other);
- the publish/subscribe model is a very flexible communication paradigm. Request/reply is done via a **service** which is defined by a pair of messages: one for the request and one for the reply.

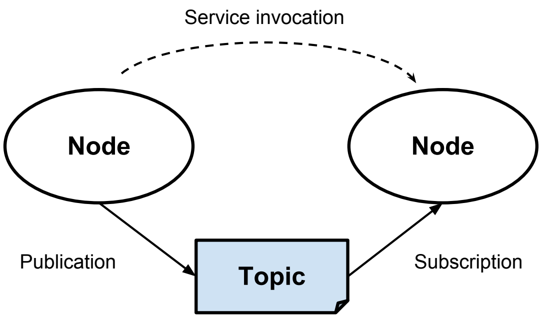


Figure 5.2: Communication between nodes in ROS

5.1.2 The robot

The Pioneer 3-AT (figure 5.3) is a highly versatile four wheel drive robotic platform. Pioneer 3-AT is a small four-motor skid-steer robot ideal for all-terrain operation or laboratory experimentation. It comes complete with one battery, emergency stop switch, wheel encoders and a microcontroller with ARCOS firmware, as well as Pioneer SDK advanced mobile robotics software development package. P3-AT's powerful motors and four knobby wheels can reach speeds of 0.8 meters per second and carry a payload of up to 12 kg. The P3-AT uses 100 tick encoders with inertial correction recommended for dead reckoning to compensate for skid-steering.

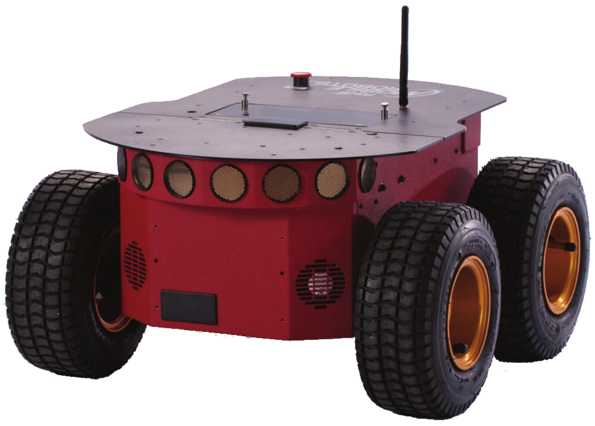


Figure 5.3: Pioneer 3-AT

The core software provides a framework for controlling and receiving data from all MobileRobots[2] platforms, as well as most accessories. Includes open source infrastructures and utilities useful for writing robot control software, support for network sockets and an extensible framework for client-server network programming.

5.2 The application

This application permits to move the robot (placed in a specific floor point) in a free obstacles environment. It was developed a new ros package in order to manage the communication between the *Click and View* application and the robot (figure 5.4).

At first, the `VideoEventReceiverNode` of the previous application has been modified. It is necessary to publish a message with the floor coordinates (in meters) in the correct topic with the ROS command `rostopic`. This command must be execute from the command shell.

```
// Composition of the command to execute
stringstream msgs;
msgs << "rostopic pub coordinates geometry_msgs/Pose ’{x: "<< worldX/1000.0
    <<", y: "<< worldY/1000.0 <<", z: 0.0}’ ’{x: 0.0, y: 0.0, z: 0.0, w:
    0.0}’ -1 &";
// Invokes the command processor to execute the command.
system(msgs.str().c_str());
```

The application that manages the sending of information to the robot, is developed in C++ language. It is part of a node that must be integrated in the ROS system. This node subscribes to the topic *coordinates*. The topic is the same where the *Click and View* application publishes the message with the floor world coordinates.

[2]http://www.mobilerobots.com/

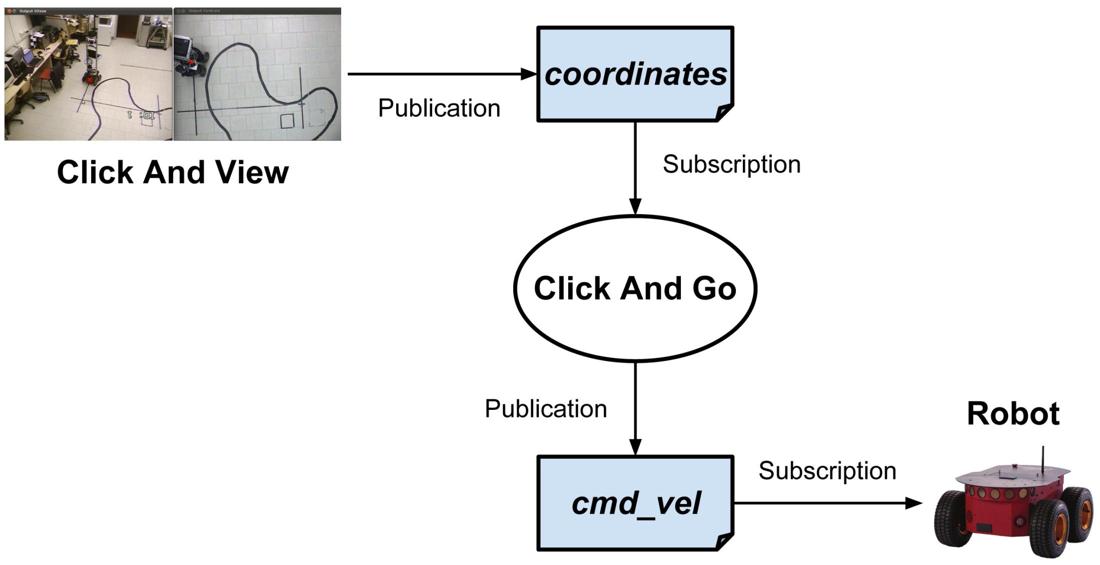


Figure 5.4: Schema of the Click and Go application

```
ros::Subscriber coordinates_sub;
...
ros::NodeHandle n_;
// Waits for a topic "coordinates"
coordinates_sub =
  n_.subscribe<geometry_msgs::Pose>("/coordinates",1,callbackFunction);
```

The type of the message is contained inside the `geometry_msgs` package. This package provides messages for common geometric primitives such as points, vectors and poses. In particular it is part of the `geometry_msgs/Pose` type, that is a representation of pose in free space, composed of postion and orientation. The message will only use the elements position.x and position.y and the `geometry_msgs/Pose` type is also used for manage the position of the robot into the environment.

```
// Includes package
#include <geometry_msgs/Pose.h>
...
// Initialize the current position of the robot
geometry_msgs::Pose currPose;
currPose.position.x= 0;
currPose.position.y= 2.80;
```

When a message in the topic is subscribed, the *callbackFunction* function is called. This calculates the times of rotation and translation of the robot from the received coordinates. In fact the robot is controlled by communicating the duration of the rotation and the translation to be performed (with fixed velocity).

```
void callbackFunction( const geometry_msgs::Pose :: ConstPtr & msg ) {
...

  // Calculates the rotation angle assuming that the robot is aligned with
      the x axis
  double angle = std::atan2((msg->position.y - currPose.position.y),
      (msg->position.x - currPose.position.x));
  // Updating the angle by introducing the current orientation
  // yaw = the current orientation with respect to the x axis
  double angle_r = angle - yaw;

  angle_r = ((int)(angle_r/M_PI*180 + 360 + 180)%360 - 180)*M_PI/180;

  // Calculates the rotation duration at turn_vel velocity
  // The turn_vel velocity is decreased by a TURN_OFF factor
  double durationTurn = std::abs(angle_r) / (turn_vel - TURN_OFF);

  //Calculate time needed to translate (walk)
  double space = std::sqrt( std::pow((std::abs(msg->position.x -
      currPose.position.x)),2) + std::pow((std::abs(msg->position.y -
      currPose.position.y)),2));
  //Calculates the linear movement duration at walk_vel velocity
  // The walk_vel velocity is decreased by a WALK_OFF factor
  double durationWalk = (space) / (walk_vel - WALK_OFF);

...
}
```

Afterwards, it is necessary to publish into the *cmd_vel* topic (this is the topic for the control of the robot) the messages to manage first the rotation an than the translation. These messages are published for all the time of the action to be performed. The duration of this actions is controlled by the `ros::Time` and the `ros::Duration` classes included in the `roslib` package.

```
ros::Time endTimeTurn(0.0);
ros::Time endTimeWalk(0.0);
...

// Starts the rotation action
endTimeTurn = ros::Time::now() + ros::Duration(durationTurn);
while(turn){
  if(ros::Time::now().toSec() < endTimeTurn.toSec())
  {
    // Sets the rotation velocity
    cmd.angular.z = (angle/std::abs(angle))*turn_vel;
    // Publishes the command for robot rotation
    vel_pub_.publish(cmd);
  }else
  {
    // Stops the rotation by setting the rotation velocity to zero
    cmd.angular.z = 0;
    // Publishes the command to stop the rotation
    vel_pub_.publish(cmd);

    sleep(2.0);
    turn = false;
    // Now the translation can be started
    walk = true;
  }
}
```

```
// Starts the translation action
endTimeWalk = ros::Time::now() + ros::Duration(durationWalk);
while(walk){

  if(ros::Time::now().toSec() < endTimeWalk.toSec())
  {
    //Sets the translation velocity
    cmd.linear.x = walk_vel;
    // Publishes the command for robot translation
    vel_pub_.publish(cmd);
  }else
  {
    // Stops the rotation by setting the translation velocity to zero
    cmd.linear.x = 0;
    // Publishes the command to stop the translation
    vel_pub_.publish(cmd);

    sleep(2.0);
    walk = false;
  }
...
}
```

Chapter 6

Experimental results

6.1 Results of the Click and View application

The overall results for the cameras calibration are tested with the *Click and View* application. By analyzing the projection from the point of the image plane to the world plane and viceversa, it is possible to determine the goodness of the cameras calibration. The test room is situated into the Intelligent Autonomous Systems (IAS) laboratory in Padua.

The *Click and View* application uses large amounts of compute resources, for this reason it is necessary to run the application on a high performance computer. The computer used has a Intel Xeon E31225 at 3.10GHz processor and 4 GB of RAM.

6.1.1 Results of pinhole camera calibration

The results about the pinhole camera calibration are anyway really satisfactory. As shown in figure 6.1, by the projection of the point on the floor from world coordinates to the image plane coordinates is possible to obtained very good results.

Furthermore, the experimental results in *Click and View* application that use the calibration parameters obtained with each pinhole camera are satisfying. The next two tables show the average errors in centimeters about the world to camera point projection and after, the camera to world point projection for each pinhole camera. The errors are calculated with respect to the global X and Y axes.

Camera	Average error in X-axes (in cm)	Average error in Y-axes (in cm)
Ulisse	4.00	4.03
Door	9.30	9.00
Central	3.00	3.66

Table 6.1: Average error about point projection from real world coordinates to image plane coordinates.

In the first table it can be seen that the camera called *Door* is affected by an average error greater to the other. The reason is to search in the combination of the low quality camera and the large area of floor that it frames. In fact it shows almost the entire floor of the environment with the same resolution as the other show an area, which compares the first, is less than half. It is also known as the camera called *Central* is the one that has a smaller error. This is due

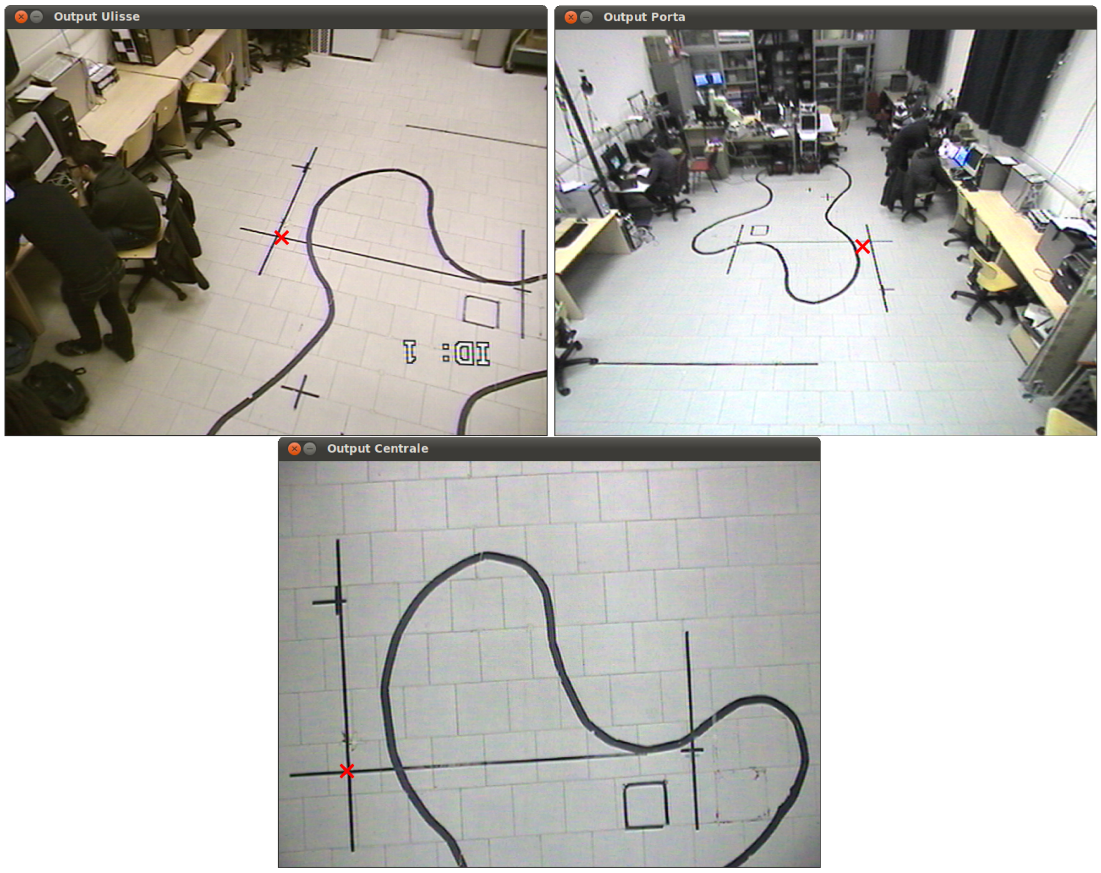


Figure 6.1: Screen of the Clic and View application result in the pinhole camera case

Camera	Average error in X-axes (in cm)	Average error in Y-axes (in cm)
Ulisse	4.45	3.46
Door	5.48	9.93
Central	1.71	3.26

Table 6.2: Average error about point projection from image plane coordinates to real world coordinates.

to the fact that it is frames a narrow area of the floor and it is situated in an optimal position respect the floor (the camera optical axis is perpendicular to the floor). In the second table it can be seen that the cameras called *Door* and *Central* are affected by an average error in Y-axes greater to the average errorin X-axes. This is due to the image size and therefore from the camera resolution. In fact, in this way are mapped a large number of meters in a limited amount of pixel. To increase the performance would be recommended to use a higher quality camera.

6.1.2 Results of omnidirectional camera calibration

Also the results about the omnicamera calibration are really satisfactory. In figure 6.2 is shown the point on the floor projected to the image camera plane.

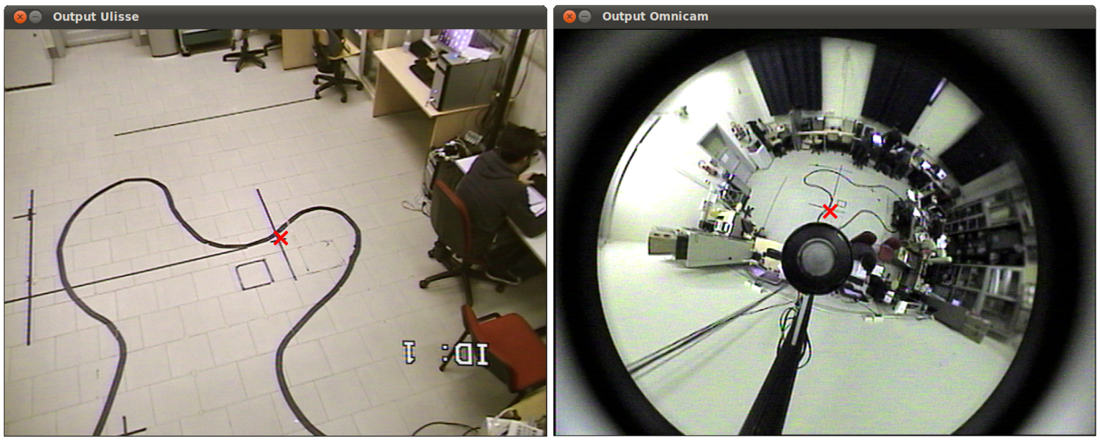


Figure 6.2: Screen of the Click and View application result in the omnidirectional camera case

The next two tables show the average errors in centimeters about the world to camera point projection and after, the camera to world point projection for each pinhole camera. The errors are calculated with respect to the global X and Y axes.

Camera	Average error in X-axes (in cm)	Average error in Y-axes (in cm)
Omnicam	5.33	3.33

Table 6.3: Average error about point projection from real world coordinates to image plane coordinates.

Camera	Average error in X-axes (in cm)	Average error in Y-axes (in cm)
Omnicam	6.07	2.67

Table 6.4: Average error about point projection from image plane coordinates to real world coordinates.

In this case the average error in X-axes is greater to the average error in Y-axes. Even here the problem is due to the combination of the low quality camera and the large area of floor that it frames.

6.2 Results of the Click and Go application

The overall results for the *Click and Go* application are anyway really satisfactory. As shown in figure 6.3 and in figure 6.4 the robot moves itself in the environment in the correctly way. These results are characterized by the previous results. In fact this application takes advantage from the calibration system used in the *Click and View* application. But in this case the robot is aligned with the destination point on the floor and the riprojection error does not affect the final position of the robot.

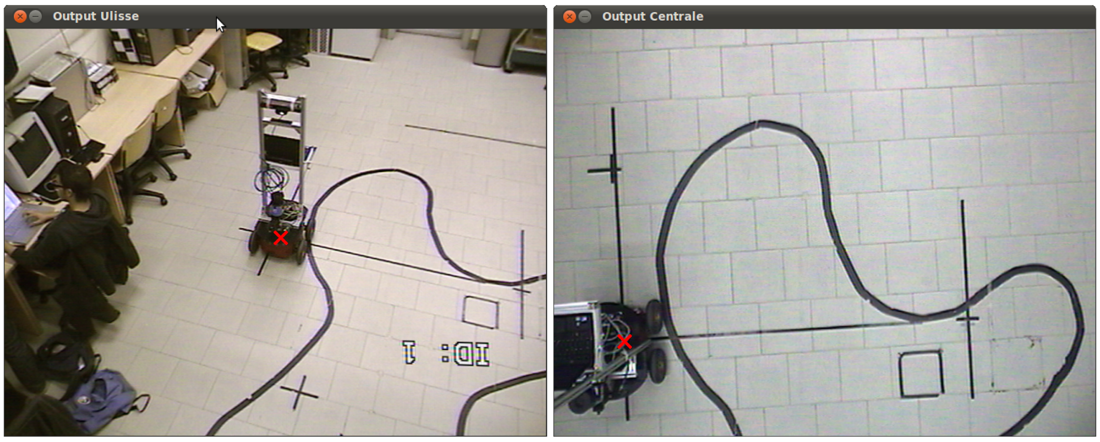


Figure 6.3: Screen of the Click and Go application result

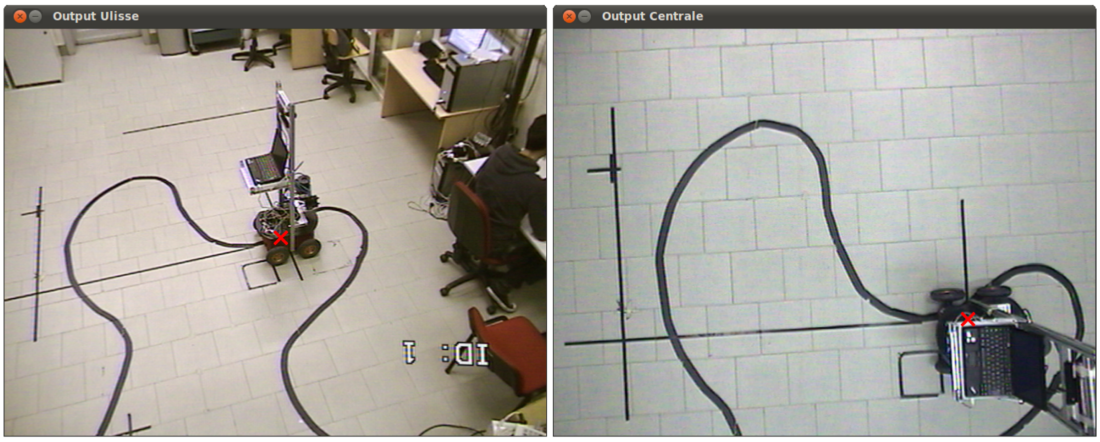


Figure 6.4: Screen of the Click and Go application result

Chapter 7

Conclusions

This work is based on the calibration of a heterogeneous set of cameras. The considered models are the pinhole camera and the omnidirectional camera. For the first it is presented a simple and reliable method and its tool for the extrinsic parameters (rotation matrix and translation vector) extraction. With this tool the user can easily find where each camera is place and its orientation. For the second model is proposed a procedure that permit to extract the extrinsic parameters (homography matrix) using the tools proposed in the literature. The results obtained by applying the parameters about each camera to the *Click and View* and *Click and Go* applications, are really satisfying. The average reprojection error of a generic point from the real world coordinates to image plane coordinates and viceversa does not exceed the ten centimeters in a total area of seventy square meters.

The problems encountered in both case are basically the same: due to the size of the considered environment and the low camera resolution make difficult the detection of the chessboard corners. For this reason it is necessary to built a specific "big" chessboard.

This work can be considered the starting point for another multi-camera procedure: using the pinhole camera tool for extract the extrinsic parameters that related the world plane with one camera plane and then carry out the calibration between each pair of cameras. In this way each camera has one rototranslation matrix with respect to the other cameras and only one camera has the rototranslation matrix with respect to the world plane. There is a global optimization of the reprojection error and a consequent accuracy increase. This possible improvements should be tested and verified with another project that take as a input these camera calibration methods.

Bibliography

[1] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly, Cambridge, MA, 2008.

[2] Z. Zhang. A flexible new technique for camera calibration. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, nov 2000.

[3] R. Tsai. A versatile camera calibration technique for high-accuracy 3d machine vision metrology using off-the-shelf tv cameras and lenses. *Robotics and Automation, IEEE Journal of*, 3(4):323 –344, august 1987.

[4] Olivier Faugeras. *Three-dimensional computer vision: a geometric viewpoint*. MIT Press, Cambridge, MA, USA, 1993.

[5] D. Scaramuzza, A. Martinelli, and R. Siegwart. A flexible technique for accurate omnidirectional camera calibration and structure from motion. In *Proceedings of IEEE International Conference of Vision Systems (ICVS'06)*, January 5-7, 2006.

[6] D. Scaramuzza, A. Martinelli, and R. Siegwart. A toolbox for easy calibrating omnidirectional cameras. In *Proceedings to IEEE International Conference on Intelligent Robots and Systems (IROS 2006)*, October 7-15, 2006.

[7] D. Scaramuzza. *Omnidirectional Vision: from Calibration to Robot Motion Estimation*. PhD thesis, ETH Zurich, 2008.

[8] R. I. Hartley and A. Zisserman. *Multiple View Geometry in Computer Vision*. Cambridge University Press, ISBN: 0521540518, second edition, 2004.