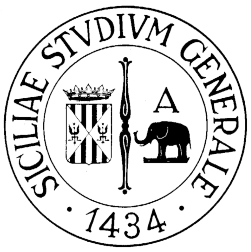
UNIVERSITA' DEGLI STUDI DI CATANIA
FACOLTA' DI INGEGNERIA
CORSO DI LAUREA DI INGEGNERIA INFORMATICA



R.E.A.R.

an exocentric vision framework for mobile robot teleoperations

defended on February 14, 2007

Student:
Daniele Ferro

Supervisors:
Prof. Salvatore Livatino
Prof. Giovanni Muscato

In the future everyone will be
a visiting researcher for 15 minutes
unknown

Contents

1 Introduction **1**
- 1.1 Virtual Presence and Telerobotics . . . 1
- 1.2 Mobile robot . . . 4
  - 1.2.1 Encoder . . . 5
  - 1.2.2 Laser . . . 7
  - 1.2.3 Sonar . . . 8
  - 1.2.4 Bumper . . . 10
  - 1.2.5 Image sensors . . . 10
  - 1.2.6 Odometry . . . 12
  - 1.2.7 Inertial navigation . . . 16
  - 1.2.8 Video based positioning . . . 16
- 1.3 The 3morduc platform . . . 19
  - 1.3.1 Sensors and actuators . . . 19
  - 1.3.2 Internet communication . . . 21
- 1.4 Improvements on telepresence . . . 26
- 1.5 Proposed investigation . . . 28

2 The Exocentric vision issue **29**
- 2.1 Why exocentric vision ? . . . 29
  - 2.1.1 Time Follower's: an overview . . . 30

3 Setting up a Morduc simulator **33**

4 Introducing OpenGL **35**
- 4.1 Some notes on OpenGL . . . 36
  - 4.1.1 The Depth Buffer . . . 36
  - 4.1.2 OpenGL viewing model . . . 37
  - 4.1.3 Matrix transformations . . . 37
- 4.2 Setting up a texture . . . 40
- 4.3 Lighting in OpenGL . . . 43
- 4.4 Perspective in OpenGL . . . 45

5 REAR: a framework for virtual exocentric vision systems **47**
- 5.1 Class diagram . . . 50
- 5.2 Classes . . . 52
  - 5.2.1 The Robot Class . . . 52
  - 5.2.2 The Camera Class . . . 53
  - 5.2.3 The DataManager Class . . . 54
- 5.3 Interfaces . . . 57
  - 5.3.1 The IDataLogic interface . . . 57

5.3.2 The IImageSelector interface 58

6 An exocentric vision system for Morduc **61**
6.1 Classes inherited from Robot class 63
6.1.1 The Morduc Class 63
6.2 Classes implementing IDataLogic interface 66
6.2.1 The DataLogicLogSimulator Class 66
6.2.2 The DataLogicLogMorduc Class 69
6.2.3 The DataLogicMorduc Class 70
6.3 Classes implementing IImageSelector interface 71
6.3.1 The spacial metric algorithm 71
6.3.2 The SpacialMetricCalc class 72
6.3.3 The sweep metric algorithm 73
6.3.4 The SweepMetricCalc class 78
6.3.5 Another sweep metric algorithm 79
6.3.6 The AnotherSweepMetricCalc class 81

7 Performance Evaluation **83**
7.1 Parameters 84
7.2 Test preparation 86
7.3 Tests results 87
7.3.1 Rectangle test evaluation 87
7.3.2 Ellipse test evaluation 88
7.3.3 Broken lines test evaluation 88
7.4 Final considerations 91

8 Future works **93**

9 Getting the source code **95**

A Questionnaire **97**

Bibliography **99**

Chapter 1

Introduction

Contents

1.1 Virtual Presence and Telerobotics 1
1.2 Mobile robot 4
1.2.1 Encoder 5
1.2.2 Laser 7
1.2.3 Sonar 8
1.2.4 Bumper 10
1.2.5 Image sensors 10
1.2.6 Odometry 12
1.2.7 Inertial navigation 16
1.2.8 Video based positioning 16
1.3 The 3morduc platform 19
1.3.1 Sensors and actuators 19
1.3.2 Internet communication 21
1.4 Improvements on telepresence 26
1.5 Proposed investigation 28

1.1 Virtual Presence and Telerobotics

Man has always dreamt of being able to be present in more than one place at the same time.
In Hindu Mythology all the gods had multiple avatars, which allowed them to exist in multiple domains simultaneously and to serve the purpose of representing the original in different places. In this way, gods could accomplished more useful work at the same time.
This dream has always been one of the aims to achieve for researchers and industries. With the advent of Internet and advanced computer and electronic technologies, we can get a step closer to obtain presence in more than one place at a time.
What we need is the equivalent of a body in the remote environment, with which we can move around, perform the proper actions through, and observe with. By combining elements of Internet and telerobotics it is possible to transparently immerse users into navigable real remote worlds and to make such systems accessible from

any networked computer in the world. In essence, we obtain *Virtual Presence*.
Virtual Presence removes the barriers of distance and offers a reasonable facsimile of instantaneous travel to anywhere on earth from any networked computer. In Virtual Presence we have a telerobot which provides a form (a physical avatar) that can be moved round in a remote space. A telerobot is a robot that accepts instructions from a distance, generally from a trained human operator. The human operator thus performs live actions in a distant environment and through sensors can gauge the consequences.
The telerobot is hence equipped with a web-camera, laser scanning and others type of sensors (see chapter 1.3 for *3morduc* concrete example). All these devices are used to provide the necessary feedback for an effective feel of the remote domain.
More formally, telerobotic is based on two main concepts, referred to as *teleoperation* and *telepresence*.
Teleoperation means 'doing work at a distance', although 'work' may mean almost anything. The term 'distance' is also vague: it can refer to a physical distance, where the operator is separated from the robot by a large distance, but it can also refer to a change in scale, where for example in robotic surgery a surgeon may use micro-manipulator technology to conduct surgery on a microscopic level.
Telepresence means 'feeling like you are somewhere else'. Thus a user, sitting at a remote computer far away from the target domain, can comfortably roam about, observe and even command specific action to be performed in the vicinity of the telerobot.
There are many jobs which a human could perform better than a robot, but for some reasons humans either do not want to or can not perform it. The job may be too dangerous, for example exploring inside a volcano; other jobs must instead be executed in location physically inaccessible. For example, exploring another planet, inspecting areas with very high level of radiations, cleaning the inside of a long pipe or performing laparoscopic surgery.
Looking at these new perspectives robots are going to be more complex than factories ones, built and programmed to accomplish few repetitive tasks. Different kind of operations are not easy to automate, and nowadays it is hard to develop autonomous robots, so it is necessary to drive them by teleoperation and virtual reality.
Looking back in history, some of our most influential technologies - such as the telescope, telephone, and television - were developed to provide knowledge at a distance. Telerobotic systems date back to the need for handling radioactive materials in the 1940s, and in the 1950's they were developed to facilitate action at a distance. Remote controlled robot are now being applied to exploration, bomb disposal, and surgery.
Specialists use telerobots to actively explore environments such as Mars and Chernobyl, while military personnel increasingly employ reconnaissance drones and telerobotic missiles. In the summer of 1997, the film Titanic included scenes with undersea telerobots and NASA's Mars Sojourner telerobot successfully completed a mission on Mars.

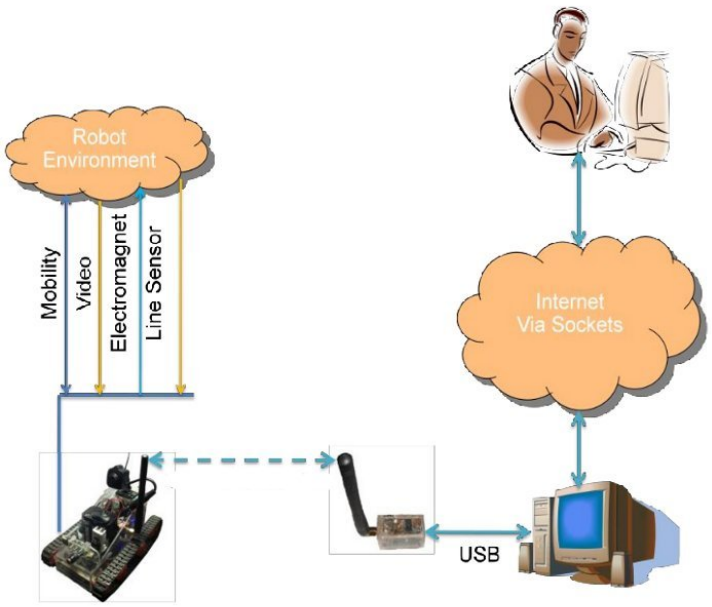


Figure 1.1: Generic Interaction human-robot through Internet

The Internet dramatically extends our scope and reach, since we can simply teleguide the robot from any part of the world. The bidirectional structure of the Internet offers furthermore a new means for actions: through one communication channel telerobotic device can send data collected by its sensors to the operator, in order to fulfil the illusion of being in the remote place and to have a robust feedback; on the other channel the operator can send the proper commands to be performed by the robot. Illustration 1.1 gives a graphical example.

1.2 Mobile robot

Mobile robots (or telerobot) have the capability to move around in their environment and are not fixed to one physical location.
As stated in [1], Leonard and Durrant-Whyte (in 1991) summarized the general problem of mobile robot navigation by three questions:

- `Where am I ?`
- `Where am I going ?`
- `How should I get there ?`

This section surveys the some basic notions in sensors and technologies that aim at answering the first question.
We will often refer to the expression *robot navigation*, that is primarily about guiding a mobile robot to a desired destination or along a pre-specified path. For these reasons teleoperator must be able, with an as great as possible precision, to know the robot's position in its environment, which consists of landmarks and obstacles. In order to achieve this objective the robot needs to be equipped with sensors suitable to localise the robot throughout the path it is to follow. These sensors may give overlapping or complementary information and may also sometimes be redundant. Because of the lack of a single, generally good method, developers of automated guided vehicles (AGVs) and mobile robots usually combine two or more methods to localize the robot.
All these sensor measurements can be fused to estimate the robot's position by using a sensor fusion algorithm. Sensor fusion in this case is the method of integrating data from distinctly different sensors to estimate the robot's position.
The sensors used to localize the mobile robot can be categorized into two groups: internal and external sensors. The ones belonging to the first group measure some internal state of the robot, such as motion, acceleration, direction.
Sensors belonging to the second group (external sensors) provide information about objects in the workspace surrounding the robot.
Sensors that will be exposed in this section are:

- `Encoder`
  Internal sensor, details in chapter 1.2.1
- `Laser`
  External sensor, details in chapter 1.2.2
- `Sonar`
  External sensor, details in chapter 1.2.3
- `Bumper`
  External sensor, details in chapter 1.2.4

- `Image sensor`
  External sensor, details in chapter 1.2.5

Data collected with previous sensors must be processed with proper methods to determinate robot's position. Among these we will briefly cite:

- `Odometry`
  Details in chapter 1.2.6
- `Inertial Navigation`
  Details in chapter 1.2.7
- `Video based positioning`
  Details in chapter 1.2.8

1.2.1 Encoder

The simplest type of incremental encoder is a single-channel *tachometer encoder*, basically an instrumented mechanical light chopper that produces a certain number of sine or square wave pulses for each shaft revolution. Adding pulses increases the resolution (and subsequently the cost) of the unit. These relatively inexpensive

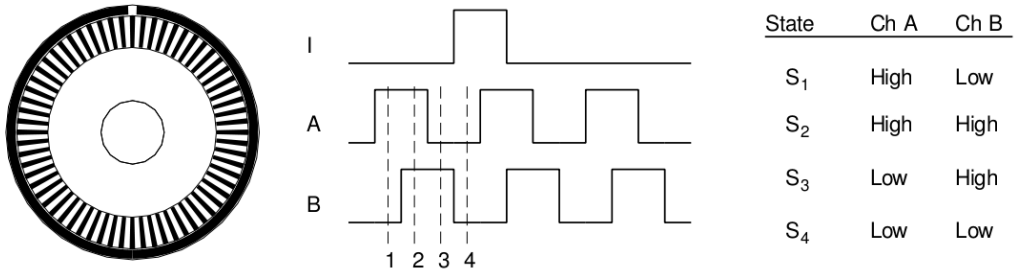


Figure 1.2: The observed phase relationship between Channel A and B pulse trains can be used to determine the direction of rotation with a phase-quadrature encoder, while unique output states S1 - S4 allow for up to a four-fold increase in resolution. The single slot in the outer track generates one index pulse per disk rotation.

devices are well suited as velocity feedback sensors in medium to high speed control systems, but run into noise and stability problems at extremely slow velocities due to quantization errors.
The tradeoff here is resolution versus update rate: improved transient response requires a faster update rate, which for a given line count reduces the number of possible encoder pulses per sampling interval.
In addition to low-speed instabilities, single-channel tachometer encoders are also incapable of detecting the direction of rotation and thus cannot be used as position sensors. Phase-quadrature incremental encoders overcome these problems by

adding a second channel, displaced from the first, so the resulting pulse trains are 90 degrees out of phase, as shown in figure 1.2.
This technique allows the decoding electronics to determine which channel is leading the other and hence ascertain the direction of rotation, with the added benefit of increased resolution.
The incremental nature of the phase-quadrature output signals dictates that any resolution of angular position can only be relative to some specific reference, as opposed to absolute. Establishing such a reference can be accomplished in a number of ways. For applications involving continuous 360-degree rotation, most encoders incorporate as a third channel a special index output that goes high once for each complete revolution of the shaft (see figure 1.2).
Intermediate shaft positions are then specified by the number of encoder up counts or down counts from this known index position.
One disadvantage of this approach is that all relative position information is lost in the event of a power interruption.
Interfacing an incremental encoder to a computer is not a trivial task. A simple state-based interface as implied in Figure 1.1 is inaccurate if the encoder changes direction at certain positions, and false pulses can result from the interpretation of the sequence of state changes.
Absolute encoders are typically used for slower rotational applications that require positional information when potential loss of reference from power interruption cannot be tolerated.
Instead of the serial bit streams of incremental designs, absolute optical encoders provide a parallel word output with a unique code pattern for each quantized shaft position. The most common coding schemes are *Gray code*, natural binary, and binary-coded decimal.
The Gray code (for inventor Frank Gray of Bell Labs) is characterised by the fact that only one bit changes at a time, a decided advantage in eliminating asynchronous ambiguities caused by electronic and mechanical component tolerances (see figure 1.3a). Binary code, on the other hand, routinely involves multiple bit changes when incrementing or decrementing the count by one. For example, when going from position 255 to position 0 in figure 1.3b, eight bits toggle from 1s to 0s. Since there is no guarantee all threshold detectors monitoring the detector elements tracking each bit will toggle at the same precise instant, considerable ambiguity can exist during state transition with a coding scheme of this form.
Some type of handshake line signaling valid data available would be required if more than one bit were allowed to change between consecutive encoder positions.
Absolute encoders are best suited for slow and/or infrequent rotations such as steering angle encoding, as opposed to measuring high-speed continuous (i.e., drive wheel) rotations as would be required for calculating displacement along the path of travel.
A potential disadvantage of absolute encoders is their parallel data output, which requires a more complex interface due to the large number of electrical leads.

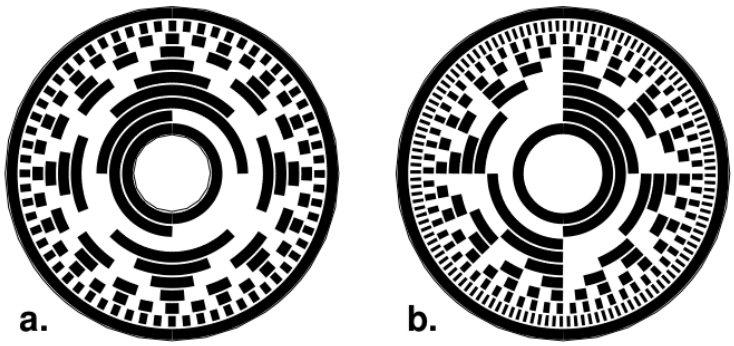


Figure 1.3: Rotating an 8-bit absolute Gray code disk. a. Counterclockwise rotation by one position increment will cause only one bit to change. b. The same rotation of a binary-coded disk will cause all bits to change in the particular case (255 to 0) illustrated by the reference line at 12 o'clock.

1.2.2 Laser

In laser-based systems, also known as *laser radar* or *lidar*, a beam of light is emitted in a rapid sequence of short bursts aimed directly at the object being ranged.
The time required for a given pulse to reflect off the object and return is measured and used to calculate distance to the target based on the speed of light. Accuracies for early sensors of this type could approach a few centimeters over the range of 1 to 5 meters.
The method above is also used in ultrasonic sensors (we refer to chapter 1.2.3 for more details), where sound pulses are transmitted and the reflection time is measured (also known as *time of flight* or shortly *TOF*).
The TOF is directly proportional to the distance between the scanner and the object. By indicating with L the distance between the laser and the obstacle, whereas with c the light speed, the formula to reckon up TOF is:

$$t_{flight} = \frac{2L}{c} \Longrightarrow L = \frac{t_{flight}}{2} \cdot c$$

The quality of time of flight range sensors manly depends on:

- Uncertainties about the exact time of arrival of the reflected signal
- Inaccuracies in the time of fight measure
- Interaction with the target (surface, specular reflections)
- Variation of propagation speed
- Speed of mobile robot and target

Another easier method, working with laser beams, is to transmit 100% amplitude modulated light beam at a defined frequency and compare the phase shift between

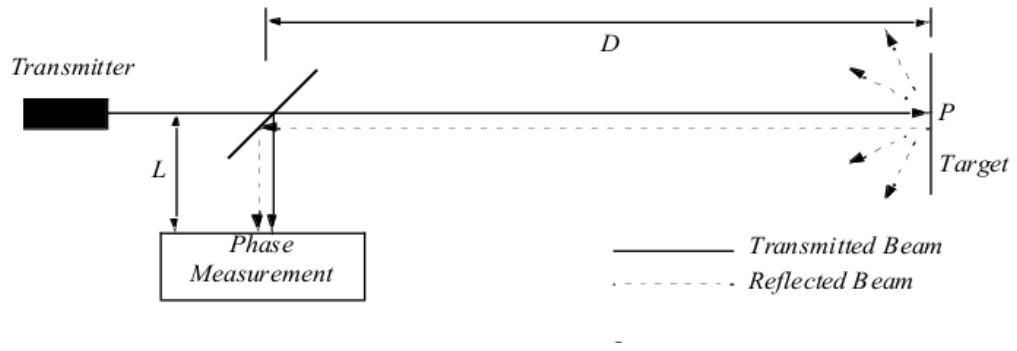


Figure 1.4: Phase-Shift Measurement

the transmitted and the reflected light (a scheme is presented in figure 1.4).
This scanner has much higher resolution. The mathematical formulas to obtain distance follow. We indicate with f the modulating frequency; with c, once again, the light speed. We can calculate the wavelength λ as:

$$\lambda = \frac{f}{c}$$

The distance covered by the emitted light, D', referring to the distances D and L showed in figure 1.4, is equal to:

$$D' = L + 2 \cdot D = L + \frac{\theta}{2 \cdot \pi} \cdot \lambda$$

where θ is the phase difference between transmitted and reflected beacon, return by the module 'phase measurement'. In the end, distance D is reckoned by the following formula:

$$D = \frac{\lambda}{4 \cdot \pi} \cdot \theta$$

Confidence in the range (phase estimate) is inversely proportion to the square of the received signal amplitude. Pointing the measuring beam to rotating mirror a fast scanning can be accomplished in a vertical dimension, but the whole scanner has to be able to move horizontally if 3D scanning is needed. A schematic drawing for 2D scanner laser is showed in figure 1.5. In figure 1.6 a typical range image of a 2D laser range sensor with a rotating mirror is shown. The lengths of the lines through the measurement points indicate the uncertainties.

1.2.3 Sonar

Ultrasonic TOF ranging is today the most common technique employed on indoor mobile robotics systems, primarily due to the ready availability of low-cost systems and their ease of interface.

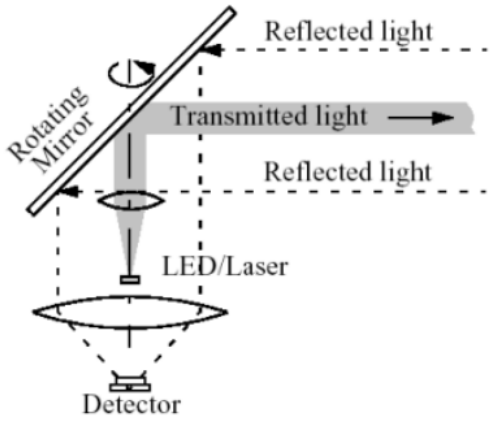


Figure 1.5: Schematic drawing of laser range sensor with rotation mirror.

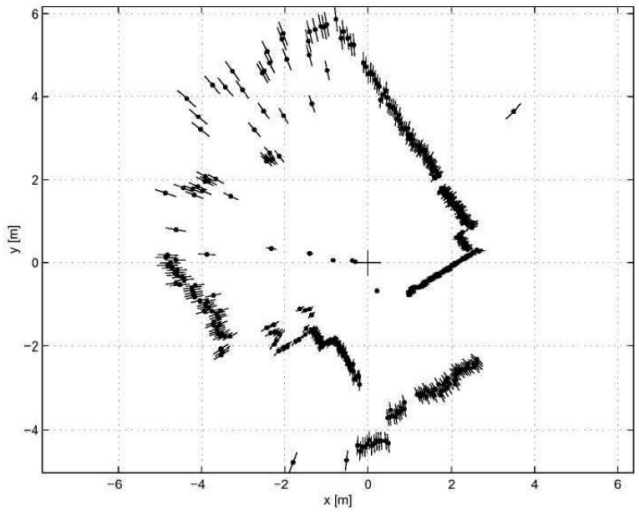


Figure 1.6: Typical range image of a 2D laser range sensor with a rotating mirror.

Active sonar creates an ultrasonic pulse, often called 'ping', then listens for reflections 'echo'. Referring to figure 1.7, the ping wave is drawn in red, the echo wave in green.
The received signal is commonly processed by measuring the time of flight, already explained in previous section (1.2.2). This time is depending on the speed of the sound in air, and thus the temperature, humidity, air pressure, and so on may effect measurements.
The knowledge of time of flight enables the computation of the distance to the target, which reflected the pulse.

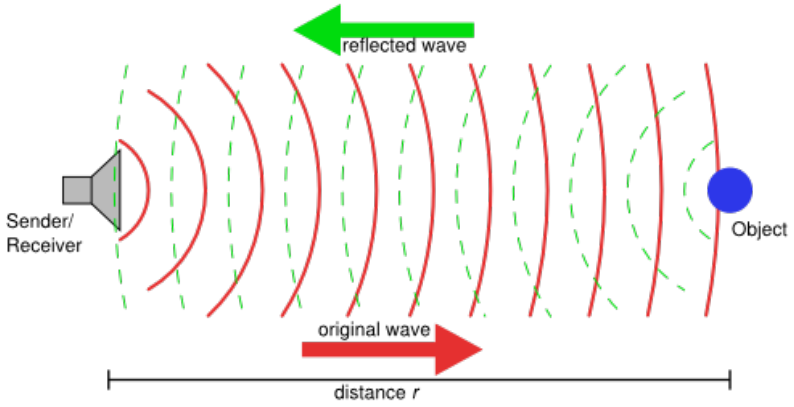


Figure 1.7: Sonars in action

The sonar field of view is a cone and the sensitive area increases proportionately with the distance. It is necessary to introduce an inhibition time to avoid the false obstacles due to the ping signal. On the other hand, the inhibition time does not allow reading distance too short.
The main drawback with the sonar sensor is its wide beam of perception, which causes the fact that it is impossible from reading returned data to identify the object position within the beam. Some of the other sonar drawback are specular reflection and the possibility of a crosstalk.
Figure 1.8 shows an example of crosstalk. Crosstalk is a phenomenon in which one sonar picks up the echo from another. One can distinguish between a. direct effectiveness of ultrasonic sensors in crosstalk and b. indirect crosstalk.

1.2.4 Bumper

The basic explanation given for why mobile robots need instrumented bumpers is for a last line of defense type of sensor, one that takes over if the IR (infra-red), sonar, or other navigational sensors fail to detect an obstacle. Or, more bluntly, if your other sensors misses an obstacle, the bumper will indicate the robot has impacted with an object of unknown size and shape. The robot then can initiate some sort of evasive/escape maneuvers to hopefully get back on track to its destination.
Another advantage of using bumpers is that the impacts derived from collisions are cushioned, avoiding seriously damages to the robot's instruments.

1.2.5 Image sensors

An image sensor is a device that converts an optical image to an electric signal; it is used mostly in digital cameras and other imaging devices.
Early sensors were video camera tubes but a modern one is typically a charge-coupled device (also known as *CCD*) or a complementary metal-oxid-semiconductor

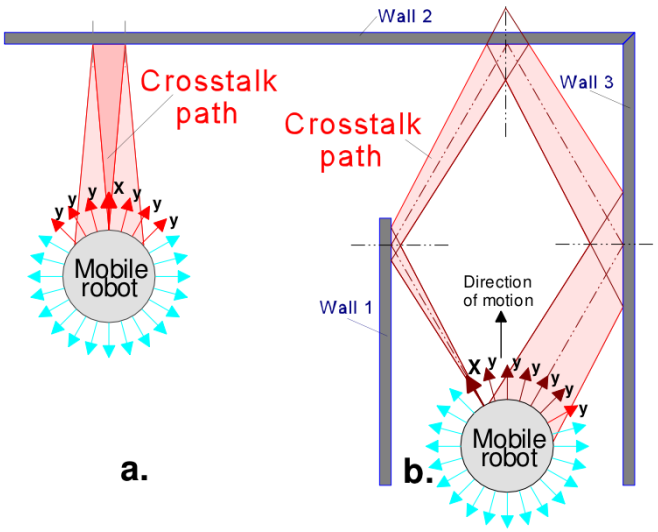


Figure 1.8: Example of crosstalk effects with sonar sensors.

(*CMOS*) active pixel sensor.
Today, most digital still cameras use either a CCD image sensor or a CMOS sensor. Both types of sensor accomplish the same task of capturing light and converting it into electrical signals.
A CCD is an analog device. When light strikes the chip it is held as a small electrical charge in each photo sensor. The charges are converted to voltage one pixel at a time as they are read from the chip. Additional circuitry in the camera converts the voltage into digital information.
A CMOS chip is a type of active pixel sensor made using the CMOS semiconductor process. Extra circuitry next to each photo sensor converts the light energy to a voltage; additional circuitry on the chip may be included to convert the voltage to digital data.
Neither technology has a clear advantage in image quality. On one hand, CCD sensors are more susceptible to vertical smear from bright light sources when the sensor is overloaded. CMOS can potentially be implemented with fewer components, use less power, and/or provide faster readout than CCDs. CMOS sensors are less expensive to manufacture than CCD sensors.
There are many parameters that can be used to evaluate the performance of an image sensor, including its dynamic range, its signal-to-noise ratio, its low-light sensitivity, etc. For sensors of comparable types, the signal-to-noise ratio and dynamic range improve as the size increases.
For more details see [2].

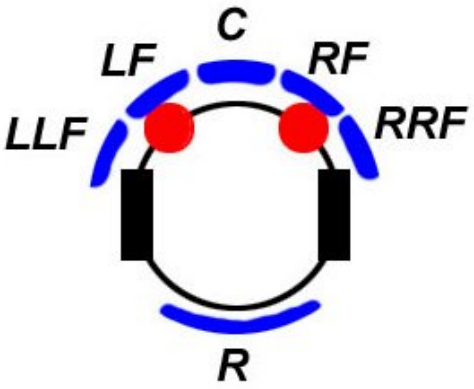


Figure 1.9: A robot provided with five bumpers, four on front side and one on rear side.

Figure 1.10: A CMOS camera.

1.2.6 Odometry

Odometry is the most widely used navigation method for mobile robot positioning. It is well known that odometry provides good short-term accuracy, is inexpensive, and allows very high sampling rates. However, the fundamental idea of odometry is the integration of incremental motion information over time, which leads inevitably to the accumulation of errors.
Particularly, the accumulation of orientation errors will cause large position errors which increase proportionally with the distance traveled by the robot. Despite these limitations, most researchers agree that odometry is an important part of a robot navigation system and that navigation tasks will be simplified if odometric accuracy can be improved. Odometry is used in almost all mobile robots, for various reasons:

- odometry data can be fused with absolute position measurements to provide better and more reliable position estimation;
- odometry can be used in between absolute position updates with landmarks. Given a required positioning accuracy, increased accuracy in odometry allows for less frequent absolute position updates. As a result, fewer landmarks are

needed for a given travel distance;

- many mapping and landmark matching algorithms assume that the robot can maintain its position well enough to allow the robot to look for landmarks in a limited area;
- in some cases, odometry is the only navigation information available. For example when no external reference is available, when circumstances preclude the placing or selection of landmarks in the environment, or when another sensor subsystem fails to provide usable data.

Odometry is based on simple equations, depending on mobility configuration chosen for the robot, that are easily implemented and that utilise data from inexpensive incremental wheel encoders.

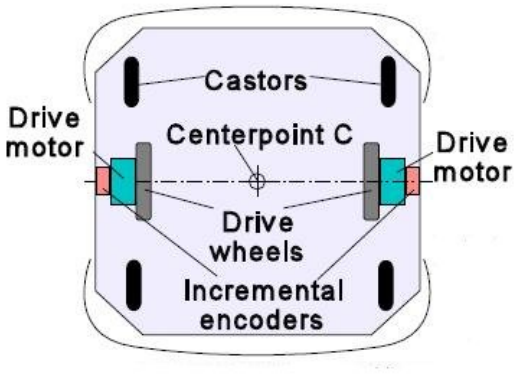


Figure 1.11: A CMOS camera.

In *differential-drive*, one of the available robot configuration shown in figure 1.11, incremental encoders are mounted onto the two drive motors to count the wheel revolutions.
The robot can compute, by simple geometric equations, the momentary position of the vehicle relative to a known starting position.
Suppose that at sampling interval i the left and right wheel encoders show a pulse increment of $N_{L,i}$ and $N_{R,i}$, respectively. Suppose further that

$$c_m = \frac{\pi \cdot D_n}{n \cdot C_e}$$

where
c_m = conversion factor that translates encoder pulses into linear wheel displacement
D_n = nominal wheel diameter (in mm)
C_e = encoder resolution (in pulses per revolution)
n = gear ratio of the reduction gear between the motor (where the encoder is attached) and the drive wheel

we can compute the incremental travel distance for the left and right wheel, $\triangle U_{L,i}$ and $\triangle U_{R,i}$, according to

$$\triangle U_{L,i} = c_m \cdot N_{L,i}$$
$$\triangle U_{R,i} = c_m \cdot N_{R,i}$$

and the incremental linear displacement of the robot's centerpoint C, denoted $\triangle U_i$, according to

$$\triangle U_i = \frac{\triangle U_{L,i} + \triangle U_{R,i}}{2}$$

Next, we compute the robot's incremental change of orientation

$$\triangle \theta_i = \frac{\triangle U_{R,i} - \triangle U_{L,i}}{b}$$

where b is the wheelbase of the vehicle, ideally measured as the distance between the two contact points between the wheels and the floor.
The robot's new relative orientation θ_i can be computed from

$$\theta_i = \theta_{i-1} + \triangle \theta_i$$

and the relative position of the centerpoint is

$$x_i = x_{i-1} + \triangle U_i \cdot \cos \theta_i$$
$$y_i = y_{i-1} + \triangle U_i \cdot \sin \theta_i$$

where
x_i, y_i = relative position of the robot's centerpoint C at instant i-th.
Unfortunately, odometry is based on the assumption that wheel revolutions can be translated into linear displacement relative to the floor.
This assumption is only of limited validity. One extreme example is wheel slippage: if one wheel was to slip on, say, an oil spill, then the associated encoder would register wheel revolutions even though these revolutions would not correspond to a linear displacement of the wheel.
Along with the extreme case of total slippage, there are several other more subtle reasons for inaccuracies in the translation of wheel encoder readings into linear motion. All of these error sources fit into one of two categories: systematic errors and non-systematic errors. Systematic errors include:

- unequal wheel diameters
- average of actual wheel diameters differs from nominal wheel diameter
- actual wheelbase differs from nominal wheelbase
- misalignment of wheels

- finite encoder resolution
- finite encoder sampling rate

Non-systematic errors include:

- travel over uneven floors.
- travel over unexpected objects on the floor.
- Wheel-slippage due to slippery floors, over-acceleration, fast turning (skidding), external or internal forces, non-point wheel contact with the floor.

The clear distinction between systematic and non-systematic errors is of great importance for the effective reduction of odometry errors. For example, systematic errors are particularly grave because they accumulate constantly. On most smooth indoor surfaces systematic errors contribute much more to odometry errors than non-systematic errors. However, on rough surfaces with significant irregularities, non-systematic errors are dominant.

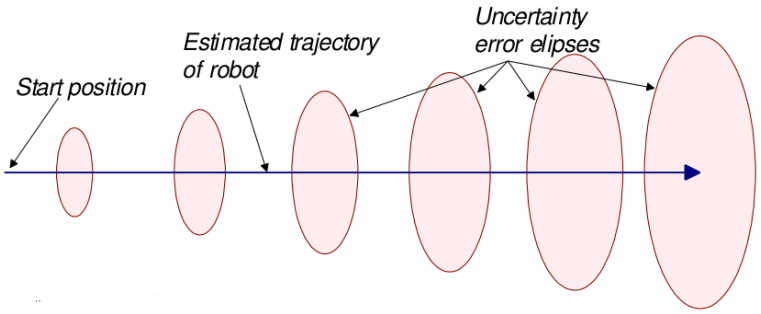


Figure 1.12: Growing 'error ellipses' indicate the growing position uncertainty with odometry.

The problem with non-systematic errors is that they may appear unexpectedly (for example, when the robot traverses an unexpected object on the ground), and they can cause large position errors. Typically, when a mobile robot system is installed with a hybrid odometry/landmark navigation system, the frequency of the landmarks is determined empirically and is based on the worst-case systematic errors. Such systems are likely to fail when one or more large non-systematic errors occur. It is noteworthy that many researchers develop algorithms that estimate the position uncertainty of a robot. With this approach each computed robot position is surrounded by a characteristic 'error ellipse', which indicates a region of uncertainty for the robot's actual position (see figure 1.12).
Typically, these ellipses grow with travel distance, until an absolute position measurement reduces the growing uncertainty and thereby 'resets' the size of the error ellipse.

1.2.7 Inertial navigation

An alternative method for enhancing dead reckoning is inertial navigation, initially developed for deployment on aircraft.
The principle of operation involves continuous sensing of minute accelerations in each of the three directional axes and integrating over time to derive velocity and position. A gyroscopically stabilized sensor platform is used to maintain consistent orientation of the three accelerometers throughout this process.
Although fairly simple in concept, the specifics of implementation are rather demanding. This is mainly caused by error sources that adversely affect the stability of the gyros used.
Experimental results indicate that a purely inertial navigation approach is not realistically advantageous (i.e., too expensive) for mobile robot applications.
Inertial navigation is attractive mainly because it is self-contained and no external motion information is needed for positioning. One important advantage of inertial navigation is its ability to provide fast, low-latency dynamic measurements.
Furthermore, inertial navigation sensors typically have noise and error sources that are independent from the external sensors. For example, the noise and error from an inertial navigation system should be quite different from that of, say, a landmark-based system.
Fundamentally, gyros provide angular rate and accelerometers provide velocity rate (i.e. acceleration) information. Dynamic information is provided through direct measurements. However, the main disadvantage is that the angular rate data and the linear velocity rate data must be integrated once and twice (respectively), to provide orientation and linear position, respectively.
Thus, even very small errors in the rate information can cause an unbounded growth in the error of integrated measurements.

1.2.8 Video based positioning

A core problem in robotics is the determination of the position and orientation of a mobile robot in its environment. The basic principles of landmark-based and map-based positioning also apply to the vision-based positioning or localization which relies on optical sensors in contrast to ultrasound, laser and inertial sensors. Common optical sensors include cameras using CCD arrays.
Visual sensing provides a tremendous amount of information about a robot's environment, and it is potentially the most powerful source of information among all the sensors used on robots.
Due to the wealth of information, however, extraction of visual features for positioning is not an easy task. The problem of localization by vision has received considerable attention and many techniques have been suggested. The basic components of the localization process are:

- representations of the environment
- sensing models

- localization algorithms

Most localization techniques provide absolute or relative position and/or the orientation. Techniques vary substantially, depending on the sensors, their geometric models, and the representation of the environment.
The geometric information about the environment can be given in the form of landmarks, object models and maps in two or three dimensions. A vision sensor or multiple vision sensors should capture image features or regions that match the landmarks or maps.
On the other hand, landmarks, object models, and maps should provide necessary spatial information that is easy to be sensed. When landmarks or maps of an environment are not available, landmark selection and map building should be part of a localization method.
Geometric models of photometric cameras are of critical importance for finding geometric position and orientation of the sensors. The most common model for photometric cameras is the pin-hole camera with perspective projection as shown in figure 1.13. Photometric cameras using optical lens can be modeled as a pin-hole camera.

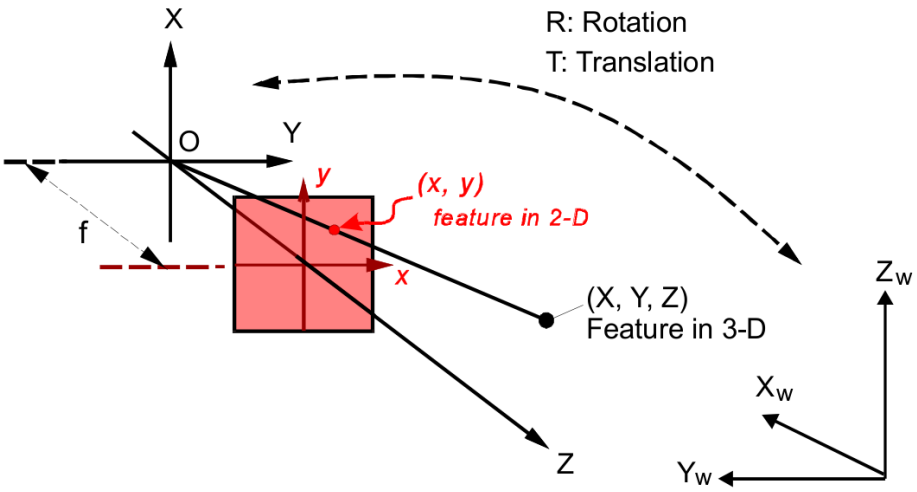


Figure 1.13: Perspective camera model.

The coordinate system (X, Y, Z) is a three-dimensional camera coordinate system, and (x, y) is a sensor (image) coordinate system. A three-dimensional feature in an object is projected onto the image plane (x, y). The relationship for this perspective projection is given by

$$x = f \cdot \frac{X}{Z}$$

$$y = f \cdot \frac{Y}{Z}$$

where f is the distance the two systems' origins.
Although the range information is collapsed in this projection, the angle or orientation of the object point can be obtained if the focal length f is known and there is no distortion of rays due to lens distortion.
The internal parameters of the camera are called 'intrinsic camera parameters' and they include the effective focal length f, the radial lens distortion factor, and the image scanning parameters, which are used for estimating the physical size of the image plane.
The orientation and position of the camera coordinate system (X, Y, Z) can be described by six parameters, three for orientation and three for position, and they are called 'extrinsic camera parameters'. They represent the relationship between the camera coordinates (X, Y, Z) and the world or object coordinates (XW, YW, ZW).
Landmarks and maps are usually represented in the world coordinate system. The problem of localization is to determine the position and orientation of a sensor (or a mobile robot) by matching the sensed visual features in one or more image(s) to the object features provided by landmarks or maps. Obviously a single feature would not provide enough information for position and orientation, so multiple features are required.
Depending on the sensors, the sensing schemes, and the representations of the environment, localization techniques vary significantly.

1.3 The 3morduc platform

Figure 1.14: The 3morduc robotic platform

The telerobot used to develop teleoperation research is called *3MO.R.D.U.C.*, acronym for '3rd version of the Mobile Robot DIEES University of Catania'. 3morduc is a mobile-robot, able to move forward, backward and turn its direction, as directed by the remote operator. It has been successfully used in several test and experimental work regarding teleoperation and telepresence.
The robot, actually located at the University of Catania, is a differential-driven mobile robot, showed in figure 1.14.
As every mobile robot, it is equipped with some internal and external sensors, through which is possible retrieve information about, respectively, the status of the robot (e.g. its position) or the data about the environment (e.g. distance from obstacles).

1.3.1 Sensors and actuators

The movement is performed by means of two 40W DC engines, model *Maxon F2260*, connected with the motor shaft by a gear box (transmission rate 1/19). On the other side the motor shaft is linked with two rubber wheels, while a third castor wheel can freely turn to realize the differential-driven model.
The robot is compound by three shelves, each one connected to the next. On the lower level two lead batteries are situated, able to erogate 12 Volts at 18 Amperes. The electrical autonomy is granted for 30-40 minutes.
Besides, on the same lower level is located an electronic board controlling different modules, each one predisposed to manage a specific task as movements, sensors and communication.
Type and number of sensors available on 3morduc are:

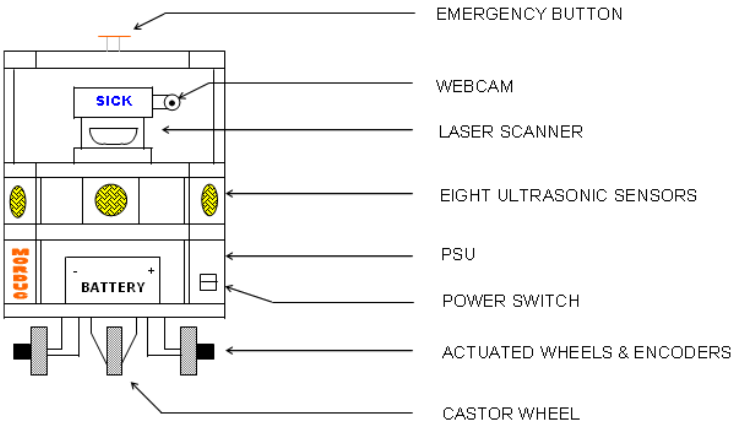


Figure 1.15: 3morduc's schematic figure

- `belt of bumpers`
  A belt of bumpers (in total 16 switches) is dislocated around the entire perimeter on the robot base, just over the wheels level.
  See chapter *Bumper* (1.2.4) for more details about this type of sensor.

- `incremental encoders`
  The two robot motor axes are equipped with incremental encoders, with resolution of 500 pulses per turn. These sensors are useful to calculate heading and position of the robot by using the kinematic model.
  See chapter *Encoder* (1.2.1) for more details about this type of sensor.

- `belt of sonars`
  On the second level are located eight sonar sensors, which measure the distance from an obstacle using the flight time of an ultrasonic signal produced by means of a vibrating piezoelectric sensor.
  See chapter *Sonar* (1.2.3) for more details about this type of sensor.

- `scanner laser`
  In order to detect obstacles on the workspace, the *Laser Measurement Sensor* (LMS) operates by measuring the flight time of a pulsed laser light beam that is reflected by obstacle, to provide a 2D scanning data.
  It is possible to configure different angular resolution (0.25°, 0.5°or 1°pace) with different angular scan (100°or 180°).
  See chapter *Laser* (1.2.2) for more details about this type of sensor. The reference manual about *LMS Sick 200* can be found at [3].

Figure 1.16: LMS Sick mounted on 3morduc

- `stereo cameras`
  On the robot there are also two stereoscopic cameras, each one with a resolution of 1.3 megapixel and fixed focus lens of 4.0 mm.
  The CCD sensors of these cameras have a good noise immunity and sensibility; moreover, it is possible to adjust all the image parameter, e.g. exposure gain, frame rate, resolution.
  The cameras are mounted on a rigid support; it permits to simply adjust the camera distance in a range 5-20 cm.
  Both the cameras can be connected to PC by IEEE 1394 interface, with a frequency of synchronization of 8 KHz.

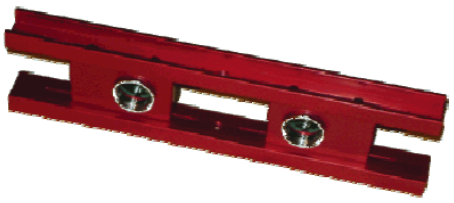


Figure 1.17: STH-MDCS2-VAR-C mounted on 3morduc

  See chapter *Image sensors* (1.2.5) for more details about this type of sensor.
  The reference manual about the *STH-MDCS2-VAR-C* can be found at [4].

More images, video and information about 3morduc can be found at [5].

1.3.2 Internet communication

The network system implemented on the 3morduc is a typical client-server architecture: in order to control the robot through Internet, 3morduc implements an HTTP server.
Through different type of prearranged HTTP requests, a generic HTTP client sends command to the server and retrieve information about 3morduc's status. In this

way, client needs only to open a TCP/IP socket on the HTTP standard port (number 80) to control the robot from remote.

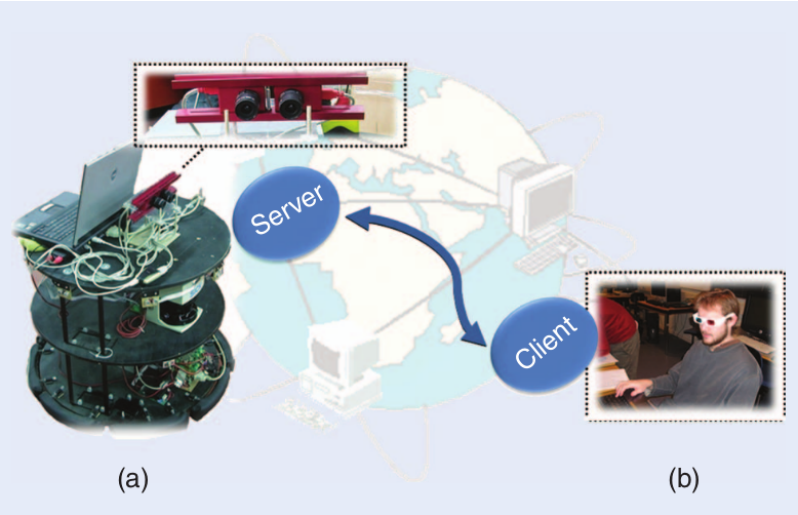


Figure 1.18: A representation of the local-remote system interaction. (a) Mobile robot in Robotics Lab, University of Catania, Italy, equipped with a stereo camera and sitting on the platform, responsible for capturing stereo images or mono images. (b) User located in remote place, using a generic HTTP client to send command to the robot and receive its images and status data.

The Server was developed in Borland *Delphi 7*, an object oriented language derived by Pascal. Several classes are wrapper of the windows A.P.I., making really simple to develop efficient code in fast way. The choice of this programming language come from the necessity to include it as part of the control system designed in Catania for the 3morduc robot.
A first client was developed using *MFC* (*Microsoft Foundation Classes*) in Visual C++ (with supporting framework *Visual Studio 2005*). It uses *OpenGL* libraries to create 3D synthetic images and to handle the different kinds of used VR instruments provided by the client platform. For a specific study view [6].
Different clients has been developed after the one described above, aiming to improve user's skill to move the robot in the remote environment (some will be briefly described in 1.4). There are clients tightly bound to Windows platform, and others built on free libraries and multi-platform code (like C++), which makes clients platform independent. The communication protocol chosen allows to access the server through a standard interface: every programming language allows to send and retrieve HTTP requests without any effort.
We will survey now, more in details, messages exchange between client and server. After creating a socket and opening the communication channel, client is able to send request to 3morduc and receive correlated responses.
Client request is built with only an HTTP header, without the body. The related

fields are listed in table 1.3.1.

Header line	Description
GET http://<URL> HTTP/1.1 <CLRF>	Retrieve (execute) the document (action) identified by <URL>, using HTTP protocol version 1.1.
Host: <HOST> <CLRF>	<HOST> is the IP serve address to which route the request.
User-agent: <CLIENT> <CLRF>	<CLIENT> is a string identifying the client. It is used for log purpose only.
<CLRF>	A black line to indicate HTTP header end.

Table 1.3.1: HTTP header request structure.

Camera images are provided together as a unique 1280x480 pixels image, (by joining two images of 640x480, left and right).
A laser map of the environment is a jpeg or bmp 200x200 image. The map-building algorithm is based on the laser scanner sensor and produces a black/white image map, where each pixel represents a square area 10cm x 10cm. A black pixel corresponds to an obstacle, otherwise the space is free. The map-building algorithm used is a classical 'occupancy grid', based on the laser row data.
Changing the requested URL (which is always directed to the same host, i.e. the 3morduc server), client can submit three different type of request. Every requested resorce matches a specific command, so a complete URL used by the client will appear as:

http://MoruducIPAddress/URLResourceName

The first requests' group allows client to fetch laser map or camera image and immediately after command the robot. 3morduc can be moved forward or backward with a fixed pace, while its direction can be turned with a fixed angle (towards left or right, from camera point of view). All these commands have the following structure:

`<image_type>.<command>.<body_format>`

where

<*image_ type*> can assume one of the value described in table 1.3.2;
<*command*> can assume one of the value described in table 1.3.3;
<*body_format*> can assume one of the value described in table 1.3.4.

Be careful that with the commands above user do not receiver last robot's image or status data, because the provided information is relative to the moment before robot executes the issued command.
The second requests' group allows client to fetch laser map or camera image, without submitting any command to the robot. In these case commands have the following structure:

Image type	Description
stereo	Retrieve image from camera.
laser	Retrieve laser map image.
laserimg	Retrieve image from camera and laser map in one frame.
datilaserandImg	Retrieve image from camera and data from laser in HTTP header response.

Table 1.3.2: Image type field in URL.

Command type	Description
fow	Move robot forward.
bak	Move robot backward.
lft	Turn robot direction to the left.
rgh	Turn robot direction to the right.

Table 1.3.3: Command type field in URL.

Body format	Description
jpg	To obtain jpeg image.
bmp	To obtain bmp image.

Table 1.3.4: Body format field in URL.

<image_type>.<body_format>

where

<*image_ type*> can assume one of the value described in table 1.3.2;
<*body_format*> can assume one of the value described in table 1.3.4.

Last command's group define an URL structure to issue commands to the robot without requesting any image or data.

<command>.<how>

In this case all the possible valid URLs are:

- *step.fow*
  Move robot forward;

- *step.bak*
  Move robot backward;

- *turn.rgh*
  Turn robot to the right;

- *turn.lft*
  Turn robot to the left;

On server side, 3morduc sends an acknowledge message as request replay. Since this is an HTTP message, it contains an HTTP header and an HTTP body. The former has always the structure summarized in table 1.3.5, with some exception.
The 'Data' field is included in HTTP header only if a 'datilaserandImg' value has been submitted as image type in the URL request. Moreover, if client issue only a command without requesting any robot's image or data (the type of commands belonging to the third group), the response HTTP body will carry no information, so fields 'Content-Length' and 'Content-Type' will be missing.

Header line	Description
HTTP/1.1 200 OK <CLRF>	HTTP implemented version is 1.1, the result code related to the request is 200 (that means everything went fine).
Server:Morduc/t/x/y/theta/collision/ mindist <CLRF>	Returned values are: name of the server; t time in millisecond; x and y abscissa and ordinate for robot position, in meters; *theta* rotation angle, in degrees; *collision* number of collisions; *mindist* distance from nearest obstacle, in meter.
Data: Laser/<laser_data1>/ <laser_data2>/.../<laser_data181>/ <CLRF>	Distance measured with laser scanner, in meters. Every scan sweeps 180°.
Content-Type: image/jpeg <CLRF>	HTTP body contains a jpeg image. This can be a 200x200 image if a laser map of the environment was requested, or a 1240x480 image (composed by two 640x480 images, from right and left camera) if camera image was requested.
Content-Length: <bytesnumber> <CLRF>	Image dimension, in byte.
<CLRF>	A black line to indicate HTTP header end.

Table 1.3.5: HTTP header response structure.

1.4 Improvements on telepresence

Various type of instrument and technologies have been adopted in order to improve the teleoperator's ability in controlling the robot. In these chapter we will cite some features developed on *3Morduc* robot by different researcher groups. Each work has been developed autonomously, and each one is orthogonal to the other, so we can think to implement all the different approaches together, rather only one at a time. These improvements can be distinguished in two main categories. The first attempted to introduce 3D visualisation, obtained by different kind of facilities: anaglyph stereo, polarized filters or separated displays (used in *HMD*, Head Mounted Displays).
Human being can perceive the environment they are in through their five senses, and among these the most important is doubtless the sense of sight. For these reasons a telerobot is equipped with various sensor devices, but what often is considered essential is at least a single camera to provide user with remote images.
The last research works has exploited stereoscopic visualisation, since the use of 2D images involves many limitations in performing tasks in 3D environment. With mono images we lose the depth perception, while with stereoscopic vision the user feels more immersed and present in the virtual environment.
The use of stereo vision in teleoperation is expected to improve navigation performance and driver capabilities, as explained in [6]. A user can more easily adapt to new environment, and also perceive the spatial localisation of the objects and the obstacles. This is a key feature in interaction jobs to avoid errors or collisions, caused by wrong estimation of a distance.

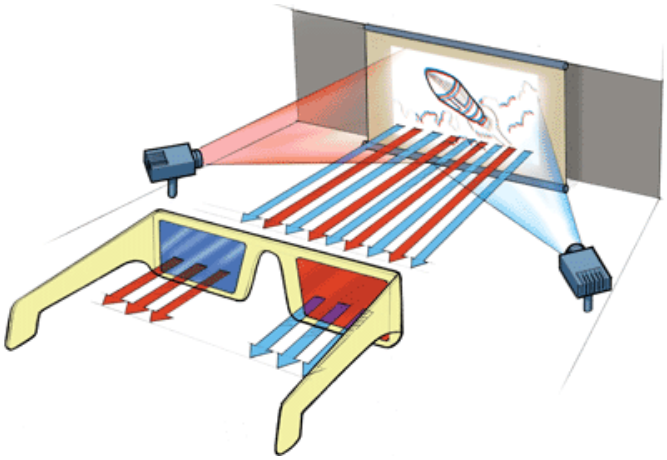


Figure 1.19: 3d polarized glasses

Thanks to the robot double camera, two images from different point of view are retrieved by the client (i.e. the teleoperator system) and processed in order to recreate a 3D perspective, by one specific technology (anaglyph stereo, polarized filter, HMD, and so on). Figure 1.19 shows an example with 3d polarized glasses.

The second approach was instead based on *Augmented Reality*. An augmented reality system generates a composite view for the user, by combining the real scenes with virtual scenes generated by the computer. In this way the scene is 'augmented' with additional information, enhancing operator's perception of the remote world. Information about depth perception, not available on 2D images, is recovered by laser sensor data. A laser scan sweeps the area around the robot and returns distance values from near objects. Augmented reality is used to mix information from camera and laser: starting from stereo or mono pictures, information about distance is provided using colour palette for making simple the perception of objects' proximity during teleguide operation. A bundle of red lines is drawn starting from the robot and ending in nearest objects' bases; for the further ones a bundle of green lines is shown instead. In both case, it is possible to visualise the exact distance from the object.
More details can be found in [7], while an example is provided in figure 1.20.

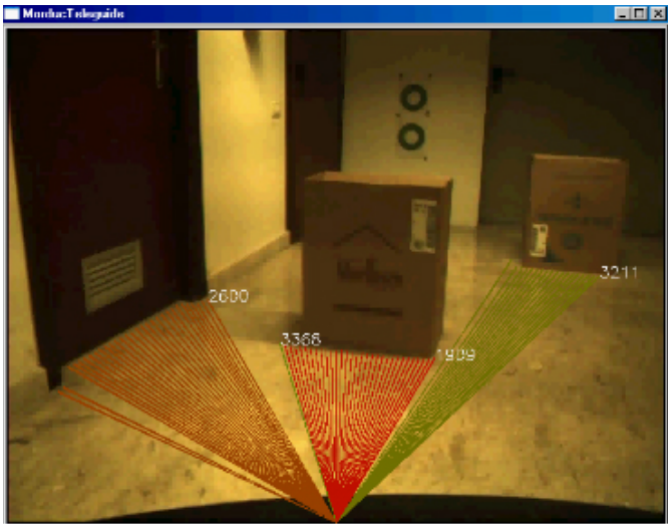


Figure 1.20: 3d polarized glasses

A better perception of remote world can be achieved by other means, for example by an *exocentric* point of view. Next chapter (1.5) will further explain this concept.

1.5 Proposed investigation

The project developed at the University of Hertfordshire, UK, during period of three months, was focused to implement a new way of teleguiding 3morduc from a remote station.
The final result was a framework, called *R.E.A.R.*, able to provide an external point of view (i.e. an 'exocentric vision') from which the operator can watch and control the robot.
Thanks to an external point of view, the operator can easier perceive the environment the robot moves in. Distances between robot and obstacles are immediately recognized, since the latter are compared with robot's dimension.
The external point of view would need another camera situated on the rear part of the robot. We will try to 'simulate' this camera with the images provided by the frontal (and unique) camera.
Besides, the code represents a framework because it defines the basic steps to implement an exocentric vision, regardless of how the robot used, the communication model implemented, and other specific detail.
Before presenting deeper details, it must be underlined that all the basic work has been accomplished together with the student Loris Fichera, without whom *R.E.A.R.* and even this document would not be written.
Next section will better introduce the reader to the problem with appropriate discussions and examples. After describing the author's intention, several chapter about *how* the target has been reached will follow.
First effort was directed to find a proper 3morduc simulator, to move first steps in our project: with a simulator is easier change the environment and edit some features that would require much more time and efforts with the real server. Test cases are indeed faster, and produce code more reliable. Further details can be found in chapter 3.
A new version of *R.E.A.R.* has been then written in order to use the real 3morduc robot, by the communication system exposed in chapter 1.3.2. Since *R.E.A.R.* was designed to be a framework, only a limited number of classes had to be added. The authors' beginning target was fully achieved: adding a concrete implementation of a particular robot, with a proper communication channel, does not require built the all the program from scratch.
Finally, it is important to underline that even the main algorithm can be changed regardless of all the other component. So, the algorithm used with a simulator can be used when the system communicate with real robot.
All these properties increasely grow some good software properties, like reusability, reliability and adaptability. Nevertheless, since the code written in C++ is supported only by free libraries, it is fully platform independent.

Chapter 2

The Exocentric vision issue

Contents

2.1 Why exocentric vision ? 29
2.1.1 Time Follower's: an overview 30

2.1 Why exocentric vision ?

Teleoperated mobile robots prove to be extremely useful when there is the need for performing operations in places that are inaccessible or dangerous for human beings - e.g. search-and-rescue missions within unknown regions or into collapsed buildings, caves, etc.
The supervisors' research group has been widely involved during the last years in such a field: [8]. As testing platform, they have been using *3morduc*, a differential-driven mobile robot - see chapter 1.3 - equipped with a pair of *Videre Design* [9] stereo cameras and a laser scanner.
They have been focused mainly on the making of a reliable hardware and software infrastructure which could make a remote operator able to drive the Morduc in comfort.
Indeed, our work focuses on *robot-operator interaction* and, hence, on how to improve such interaction.
Analyzing previous work and data produced by the supervisors' research group, it emerges that the stereo cameras mounted on top of Morduc were used to provide the remote operators a *first-person* point of view. In literature, such a system is also called an *egocentric* vision system.
According to [10], *by observing the camera image without an efficient human interface system, the operator tends to misinterpret the robot's position and direction.* This is due to the fact that *it's difficult for an operator not accustomed to the vehicle to estimate the vehicle's position and direction and the distances to a target strictly based on camera images from the first person viewpoint.*
In order to improve the interaction between the robot and the operator an exocentric camera would be effective since it would provide a view of the robot in the operating environment and, thus, a better understanding of where the robot is located into the environment and its actual direction.
Unfortunately the use of an exocentric camera is not straightforward: for example,

it could be mounted on a rear-mounted protuberance of the robot, but such a protuberance would terribly limit the robot activity and its moving abilities.
To avoid such complications, [10] proposes *Time Follower's*, an approach to provide a *virtual exocentric view* of a mobile robot.

2.1.1 Time Follower's: an overview

Time Follower's aims at providing an exocentric view of a mobile robot using an egocentric-mounted camera. The approach is simple: it is based on the use of previously recorded first-person images to provide comprehensible third-person imagery. A 3D representation of the robot is overlapped to such images in order to get the work done. A graphical representation of the system is depicted in figure 2.1, while figure 2.2 shows an example of what appears to the end user.

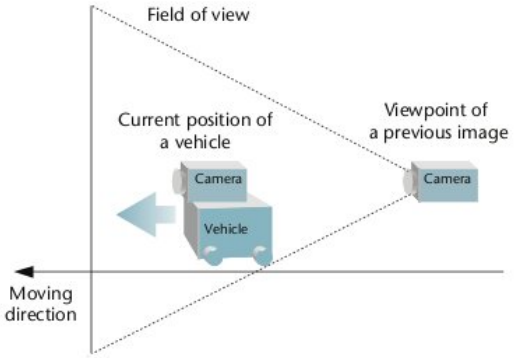


Figure 2.1: The Morduc robotic platform

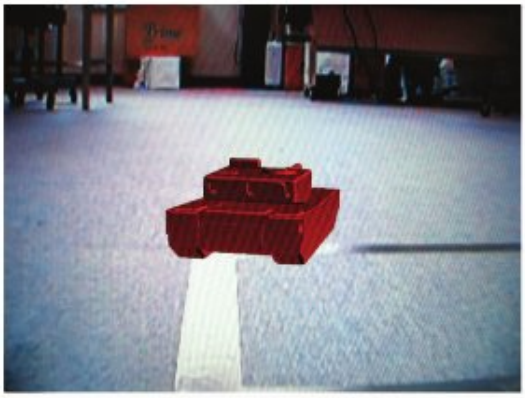


Figure 2.2: A virtual exocentric vision example

Key issues of such a system are:

- how to choose the *best* image

- how to determine the *right* place to draw the robot

As for the first issue, the *best* image is the one that allows the generation of the *most comprehensible* external robot view. [10] provides a set of three effective metrics to determine which, of a set of previously recorded images, is the one to be chosen.
Once the image is chosen, where to overlap the 3D robot representation is quite complex. It is not only a matter of *where* draw the robot on the image, but also of *how* to set up lighting, robot dimensions, perspective to make the end user not aware of the fact that what he is seeing is not the actual environment.
The issues introduced above will be deeply analyzed in the following of this document.

Chapter 3

Setting up a Morduc simulator

A basic exocentric vision system for Morduc would need, at least, some images captured from the camera mounted on top of it and, for each image, the actual position of the robot at the time it was captured.
While images could be used as a texture on which draw a 3d representation of the robot, information about position are essential in order to draw the robot consistently, i.e. in the position the observer would see it if he was seeing the robot by means of an actual external camera.
An exocentric vision systems performs at its best when there's a large static space where the robot can move in. Since the real robot can not be easily teleguided in wide environment because of its size and the lack of a large room in Catania, where the robot is situated, a simulator has been used to reproduce the best set of data. The simulator can be thought of as a server, which receives requests and returns responses: the first are the commands sent by the user to move the robot, the latter the egocentric images and the robot position data.
Furthermore, with a simulator is extremely easy to change the environment where the robot is teleguided, so we can test the exocentric vision with an infinite number of environments without physically moving the robot in different places. In this way software development of exocentric vision can be faster, because it is simple and immediate to establish several test cases.
The first simulator adopted was *Rosen* ([11]). Written in *Erlang* [12], *Rosen* has been developed at the *University of Catania* in order to simulate the behaviour of autonomous mobile robots (AMRs).
Rosen has been used as test bench for robots taking part in the *Eurobot* competition [13], and hence, for robots with completely different features from the ones of Morduc.
As soon as we realized *Rosen* does not meet our needs, we started looking for a more suitable simulator.
In 2006, at the *Aalborg University*, Filippo Privitera wrote a simulator specifically intended for the Morduc platform [14].
Such asimulator reproduces the Morduc itself (actually a 3d model of it) situated in a customizable room: the position of walls can be specified by the user, by giving the simulator a black and white bitmap image with the room planimetry, to build the whole environment from.
Besides, Privitera's simulator allows to enable the stereoscopic vision, with anaglyph or polarized method (both types are applied on the egocentric camera). Other infor-

mations like the number of collisions or the robot distance from the nearest obstacle are provided by simulator.
The simulator was written using the Microsoft Foundation Class (MFC) framework and has been used for testing user ability in tele operating driving a robot, in comparison with the actual robot to quantify the differences between the two facilities. With a simulator specifically built for the Morduc robot, it was not difficult to edit the source code to obtain what we needed. First of all, we needed to record data about egocentric vision and robot status, because they are the input value of the exocentric vision simulator. In order to achieve this, we edited the source code to allows the user to store data: by pressing the 'P' key keyboard the actual information (e.g. the actual camera image and robot status) are recorded in log files.
Every session in Privitera's simulator has its own identifier (a integer number). When the 'P' key is pressed the simulator write a new line in the text file named 'data_⟨number of session⟩.txt', creating the file if it does not exist. Each line contains four float number values, with the following meaning:

1. x coordinate
2. y coordinate
3. theta value (in radiant s)
4. timestamp

where the timestamp refers to the beginning of the simulation.
Beside the text file there are several PNG images, each for every line written in 'data_⟨number of session⟩.txt'. These files are named 'screenshot_⟨number of session⟩_⟨timestamp⟩.png', where the number of session indicates for every screenshot - i.e. the egocentric vision - the proper text file, and the timestamp the line with the associated status of the robot.
The software which implement the exocentric vision control will look for text and image files related to a specific session, in order to read the necessary input and draw the robot correctly. It must be able to choose the right image to use as background, among those previously read; to draw the robot over the background in the right position and orientation, depending on its route; to prevent or signal collisions to the user, and so on.

Chapter 4

Introducing OpenGL

Contents

4.1 Some notes on OpenGL 36
4.1.1 The Depth Buffer 36
4.1.2 OpenGL viewing model 37
4.1.3 Matrix transformations 37
4.2 Setting up a texture 40
4.3 Lighting in OpenGL 43
4.4 Perspective in OpenGL 45

OpenGL is designed as a software interface to the graphics hardware on your computer. It defines a platform-independent API; the API is then implemented in software and/or hardware for various machine architectures. The advantage is that OpenGL programs are easily portable to a variety of computers. OpenGL provides basic commands to support rendering. In particular, OpenGL doesn't provide functionality to support windows or user interaction (like keyboard presses or mouse actions); these features are provided by a separate library called GLUT (the OpenGL Utility Toolkit). GLUT also provides some higher-level features, such as more complex geometrical objects.
OpenGL is a state machine, which means that you specify various states or modes which remain in effect until changed. Each command that is executed is carried out within the current state. States include things like the current window (where drawing will appear), colour, viewing and projection matrices, drawing modes, positions and characteristics of lights, materials, and features. These elements will not be introduced throughout this document, as they are already explained in various tutorials [15] o manual books [16].
Nevertheless, it is important to keep the idea of a "state machine" in mind as you work with OpenGL in order to understand the effects of a given command. The current state is not reset when a function starts or ends - the current state is in effect until something changes it, regardless of where in the program the next thing is.
In the following, we are going to analyse some peculiar aspects of OpenGL. Anyway, this chapter is not intended to be an exhaustive guide to OpenGL programming - like [17], [16], but, instead, as a collection of short how-to regarding the aspects of OpenGL which we have dealt with the most during our work.

4.1 Some notes on OpenGL

One of the design target of OpenGL was to made its API portable among multiple architectures. In order to do so, OpenGL designers decided not to make use of polymorphism and inheritance when they had to provide multiple versions of the same command taking different parameters. They used, instead, the following scheme:

```
glCommandName{NTd}()
```

that is, every OpenGL function is preceded by `gl` and can be followed by

- `N`
  the number of parameters the function takes
- `T`
  the parameters type
- `d`
  if present, it states that the function takes pointers as arguments

For example, the `glVertex` can be invoked as:

- `glVertex2i(1, 3)`
- `glVertex2f(1.0, 3.5)`

As you might have observed from the simple example above, OpenGL commands use the prefix 'gl' and initial capital letters for each word making up the command name ('glClearColor()', for example). Similarly, OpenGL defined constants begin with 'GL_', use all capital letters, and use underscores to separate words (for example, 'GL_COLOR_BUFFER_BIT').
You might also have noticed some seemingly extraneous letters appended to some command names (for example, the '3f' in 'glColor3f()' and 'glVertex3f()').

4.1.1 The Depth Buffer

When drawing up a scene with OpenGL, the order in which polygons are drawn greatly affects the blended result, especially when it comes to 3D.
The depth buffer keeps track of the distance between the viewpoint and the portion of the object occupying a given pixel in a window on the screen; when another candidate colour arrives for that pixel, it's drawn only if its object is closer to the viewpoint, in which case its depth value is stored in the depth buffer. With this method, obscured (or hidden) portions of surfaces aren't drawn and, therefore, aren't used for blending.
In order to enable the use of the depth buffer, the `glutInitDisplayMode()` can be used, passing the GLUT_DEPTH macro as an argument.

4.1.2 OpenGL viewing model

OpenGL not only provides function to specify models for three-dimensional objects but also allows to specify the position for each of the models we want to display in a scene and also the point from which to view the scene.

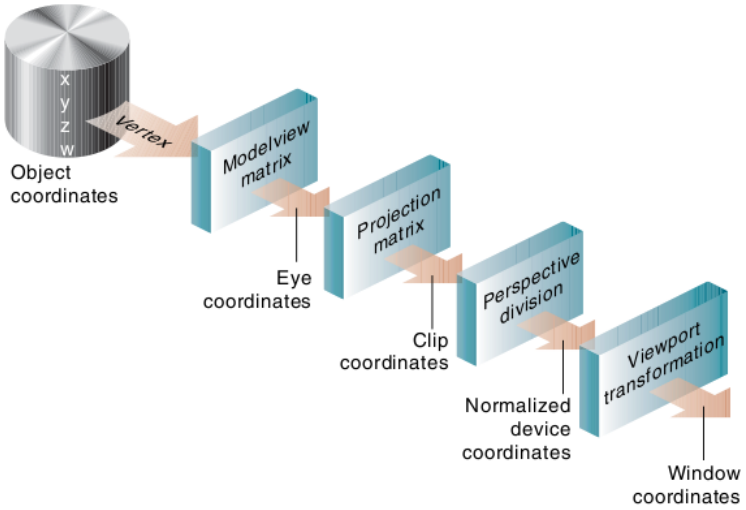


Figure 4.1: The OpenGL pipeline

The transformation process used to produce a scene for viewing is analogous to taking a photograph with a camera (in fact, it is often addressed as *the camera analogy* [18]) and consists of four steps, known as *the OpenGL pipeline* (see figure 4.1).
At the beginning of the pipeline, we have the *raw* coordinates of the models we want to put on the scene. Such coordinates are then multiplied by the a 4x4 **modelview** matrix, that is, a matrix that stores both *model* and *viewing* transformations and which elements have been set accordingly to *actually* where to place items in the scene.
The second step involves the **projection** matrix, which is applied to the incoming object coordinates to define a viewing volume. Of course, objects outside this volume are clipped so that they are not drawn in the final scene.
In the last two steps of the pipe, incoming coordinates are transformed into actual two-dimensional window coordinates and other stuff, such as the light intensity for each pixel, are calculated according to depth.

4.1.3 Matrix transformations

To edit the **modelview** and **projection** matrices, OpenGL provides quite a lot of functions: some of them are *general purpose*, that is, they can be used to edit both of them, others are specifically intended to edit either the modelview or the

projection matrix.
The most used *general purpose* functions are

- `glMatrixMode(GLenum mode)`
  mode specifies which matrix is the *current* matrix, that is, the matrix that is to be edited by following functions

- `glLoadIdentity()`
  sets the *current* matrix to be 4x4 identity matrix

For what concerns editing the **modelview** matrix, OpenGL provides a set of functions that allows the specifying of the transformations one would like to apply without manually editing the modelview matrix. The most used are

- `glTranslate(TYPE x, TYPE y, TYPE z)`
  Multiplies the current matrix by a matrix that moves (translates) an object by the given x-, y-, and z-values (or moves the local coordinate system by the same amounts)

- `glRotate(TYPE angle, TYPE x, TYPE y, TYPE z)`
  Multiplies the current matrix by a matrix that rotates an object (or the local coordinate system) in a counterclockwise direction about the ray from the origin through the point (x, y, z). The angle parameter specifies the angle of rotation in degrees.

Two examples are shown in figures 4.2, 4.3: the first one shows the effects of invoking `glTranslatef(0.0, 0.0, -5.0)` on the view, while the second one shows how to rotate a model of 45 degrees invoking `glRotatef(45.0, 0.0, 0.0, 1.0)`.

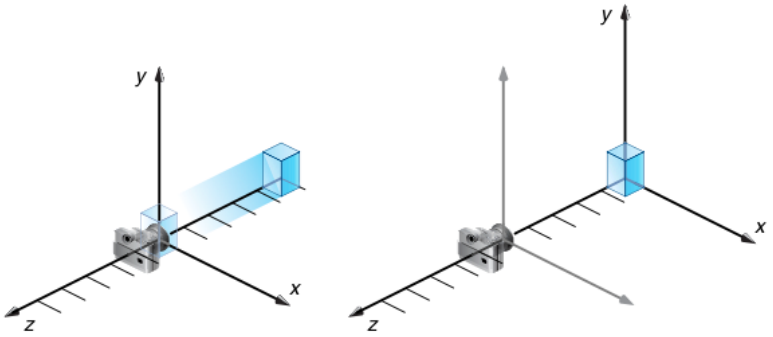


Figure 4.2: A view transformations by means of a glTranslate invocation

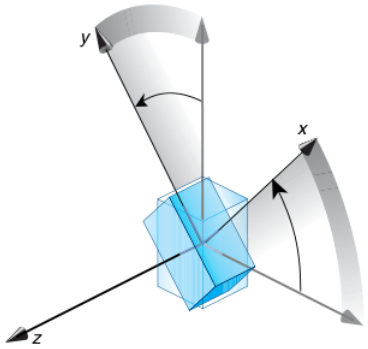


Figure 4.3: A model transformation by means of a glRotate invocation

4.2 Setting up a texture

According to [17] *texture mapping is a concept that takes a moment to grasp but a lifetime to master.*
As for our work, we were simply interested in drawing an image as the background of a window. It is not a complex task but, it could be very tricky, especially when one does not have a deep understanding of what is happening *under the hood.*
OpenGL provides 1D, 2D and 3D textures. A 2D texture is enough for our purposes. There exists a well-defined sequence of actions to draw a texture into an OpenGL application. Such a sequence made of the following steps:

1. obtain an unused texture object identifier with `glGenTextures()`, and create a texture object using `glBindTexture()`
2. set texture-object state parameters
3. specify the texture image using `glTexImage2D()` or `gluBuild2DMipmaps()`
4. before rendering geometry that uses the texture object, bind the texture object with glBindTexture()
5. before rendering geometry, enable texture mapping
6. send geometry to OpenGL with appropriate texture coordinates

Let us show an example of how to take a preloaded image, binding it to a texture and draw it as the background of a window. First of all, we would like to define a helper structure which we will call `Image` to store the actual image and its size.

Listing 4.1: The Image structure

```
struct Image {
  unsigned long sizeX;
  unsigned long sizeY;
  char * data;
};

typedef struct Image Image;
```

Now, let us suppose to have defined a `ImageLoad()` function that loads an image from disk and returns an object of class `Image`. Let us see how to put in code the procedure described above:

Listing 4.2: Texture example

```
  Gluint * texture;
  Image * image;

  // allocate space for the image
  image = (Image *) malloc(sizeof(Image));
```

```
if (image == NULL) {
  // ERROR!
  exit(0);
}

// load image from disk
if (!ImageLoad("image.bmp", image)) {
  exit(1);
}

// obtain an unused texture object
glGenTextures(1, texture);

// Bind 2d texture (x and y size)
glBindTexture(GL_TEXTURE_2D, * texture);

// now let us set up some state parameters:
// scale linearly when image bigger than texture
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                               GL_LINEAR);

// scale linearly when image smaller than texture
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                               GL_LINEAR);

gluBuild2DMipmaps(GL_TEXTURE_2D, 3, image1->sizeX,
                  image1->sizeY, GL_RGB, GL_UNSIGNED_BYTE,
                  image1->data);

// enable texture mapping
glEnable(GL_TEXTURE_2D);
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glLoadIdentity();

// deactivate depth (Z Axis)
glDepthMask(false);

glBegin( GL_QUADS );

// actually map texture
{
  glTexCoord2f( 0.f, 0.f );
  glVertex2f( -1, -1 );
```

```
    glTexCoord2f( 0.f, 1.f );
    glVertex2f( -1, 1.f );

    glTexCoord2f( 1.f, 1.f );
    glVertex2f( 1.f, 1.f );

    glTexCoord2f( 1.f, 0.f );
    glVertex2f( 1.f, -1 );
  }

  glEnd();

  // reactivate depth (Z axis)
  glDepthMask(true);

  glPopMatrix();
  glMatrixMode(GL_PROJECTION);
  glPopMatrix();
  glMatrixMode(GL_MODELVIEW);
  glDisable(GL_TEXTURE_2D);
```

4.3 Lighting in OpenGL

As shown in figure 4.4 OpenGL features three types of light:

- Ambient light
- Diffuse light
- Specular light

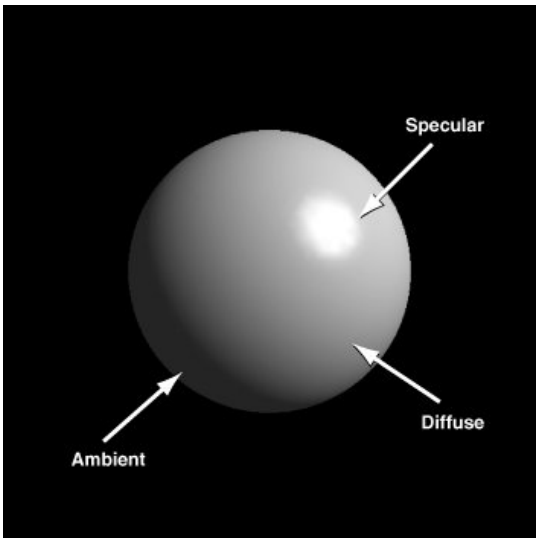


Figure 4.4: Lighting in OpenGL

Ambient light simulates indirect lighting. It illuminates all geometry in a scene at the same intensity.
Diffuse light illuminates a surface based on its orientation to a light source. OpenGL diffuse lighting adheres to Lambert's law, in which the amount of illumination is proportional to the cosine of the angle between the surface normal and the light vector, and the diffused light reflects equally in all directions.
Specular light approximates the reflection of the light source on a shiny surface.
At first glance, controlling OpenGL lighting might appear complex. The amount of code required to obtain simple lighting effects, however, is actually quite small. That is:

Listing 4.3: Lighting example

```
// Enable OpenGL lighting
glEnable( GL_LIGHTING );

// Enable a single light source
glEnable( GL_LIGHT0 );
```

`GL_LIGHT0` is one of the light points available to OpenGL programmers. For every light point, one can set its position and parameters for each light component. For example:

Listing 4.4: Complex lighting example

```
GLfloat lightpos[] = { 0.0f, 0.0f, 1.0f, 0.0f };
GLfloat lightcolor[] = { 1.0f, 1.0f, 1.0f };
GLfloat ambcolor[] = { 1.0f, 1.0f, 1.0f };

glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glLightfv(GL_LIGHT0, GL_POSITION, lightpos);
glLightfv(GL_LIGHT0, GL_AMBIENT, lightcolor);
glLightfv(GL_LIGHT0, GL_DIFFUSE, lightcolor);
glLightfv(GL_LIGHT0, GL_SPECULAR, lightcolor);
```

4.4 Perspective in OpenGL

Perspective in OpenGL can be something very trick to work with. It all starts from the definition of *frustum*, as the portion of space seen from the point of view (see [19] for further information). To define the frustum we use the function named `gluPerspective()` [20], which takes four parameters.
The first parameter indicates the *FOV* - i.e. Field Of View - along the Y axis, in degrees: the chosen value is 60. The second one expresses ratio between actual width and height, in our case 624/442. The last two parameters specify the distance from the viewer to the near clipping plane and to the far one, according to the frustum definition.
By using two close values for the far and near clipping planes distance the application will not be able to display 3d objects, unless the point of view is relatively close to them. If we want to be able to see 3d objects as they are moving away from the point of view, we need to increase the difference between the last two parameters of the `gluPerspective()` function. In our case the chosen values are 0.001 and 100000 (its ratio is equal to 10exp8).
The `gluPerspective()` function affects the **projection** matrix, previously selected by changing matrix mode. Finally, in order to set our point of view, we must change matrix again, this time to work with the **modelview** matrix.
The function named `gluLookAt()` allows to set the point of view, with its coordinates, the coordinates of the point to look at and its orientation.
See [21] for further details.

Listing 4.5: OpenGL perspective example

```
/* define the projection transformation */
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(60, 624/442, 0.001, 100000);

/* define the viewing transformation */
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(0.0, 0.0, 10.0,
          0.0, 0.0, 0.0,
          0.0, 1.0, 0.0);
```

Chapter 5

REAR: a framework for virtual exocentric vision systems

Contents

5.1 Class diagram 50
5.2 Classes 52
5.2.1 The Robot Class 52
5.2.2 The Camera Class 53
5.2.3 The DataManager Class 54
5.3 Interfaces 57
5.3.1 The IDataLogic interface 57
5.3.2 The IImageSelector interface 58

In this section we will describe REAR - *Rear Exocentric Augmented Reality*, a framework for the development of virtual exocentric vision systems.
It has been written in C++, following an object-oriented approach, and implements the minimal set of tools and functionalities needed to create representations of a mobile robot in its environment through the use of augmented reality techniques, as described in section 2.
For what concerns graphics, REAR relies on OpenGL. It makes use of three software components: a *camera*, a 3D model of the *robot* itself and a *texture*.
The camera provides the user with a view of the OpenGL space from a certain point-of-view and with a certain field-of-view. It identifies a *viewing frustum* - as shown in figure 5.1 - whose volume corresponds to the portion of OpenGL space displayed on the user's screen.
To give the user the illusion of seeing the robot from an external point-of-view, the 3d model of the robot is drawn within the viewing frustum, while the more distant frustum base is applied a texture displaying a picture previously captured from the robot's egocentric camera.
An example of what a user sees is showed in figure 5.2.
Main issues in the approach we have just described are 1. *where* to draw the robot within the viewing frustum and 2. *which* of the captured images is to be used as background.
For what concerns the first issue, one can intuitively guess that the simplest way to determine the robot position within the frustum is to know the current robot

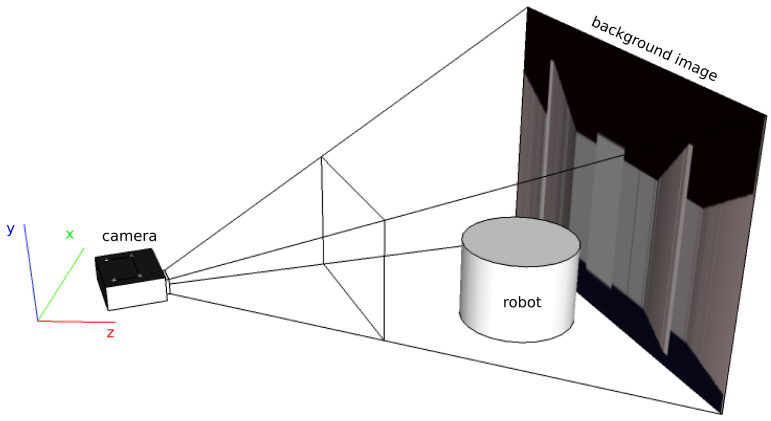


Figure 5.1: The OpenGL space

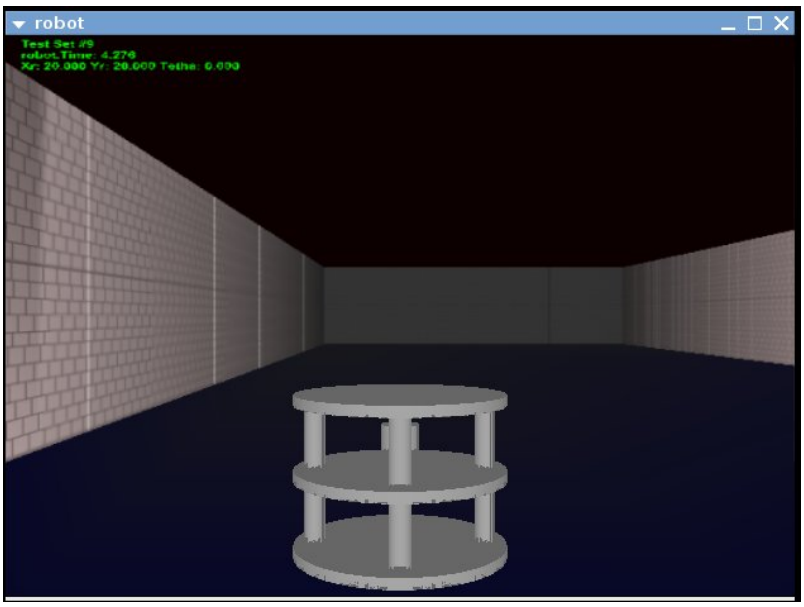


Figure 5.2: A snapshot from a REAR-based application

position and orientation and the position and orientation of the egocentric camera when the snapshot was taken and, then, to set the position of the 3d model of the robot and the point-of-view of the camera accordingly. Well, that's what REAR does.
Therefore, to run, every REAR concrete implementation needs, at least:

- a set of snapshots captured from the robot egocentric camera, together with the robot position at the time it was taken

- the robot's current position

Such data is retrieved every time the operator, using the interface provided by REAR itself, sends a motion command to the robot.
After collecting new data, which include new robot status and the associated egocentric camera image, REAR chooses an image to set as background and draw the robot model within the frustum. So, for what concerns issue no. 2, as already underlined in [10], let us say that there is not a *unique* way to determine which image is to be set as background, since different image selection algorithms would differently affect user perceptions and. We will have a deeper look at some image selection algorithms in section 5.3.2.
Finally, the overall execution loop is resumed by the flowchart showed in figure 5.3.

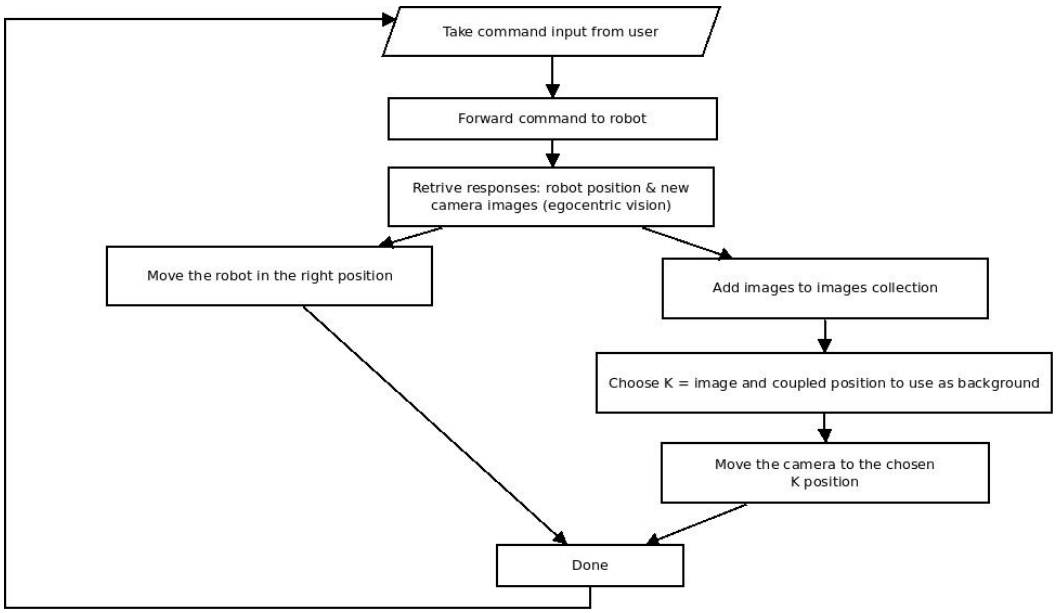


Figure 5.3: Application flowchart

5.1 Class diagram

Let us have a look at REAR class diagram, showed in figure 5.4.

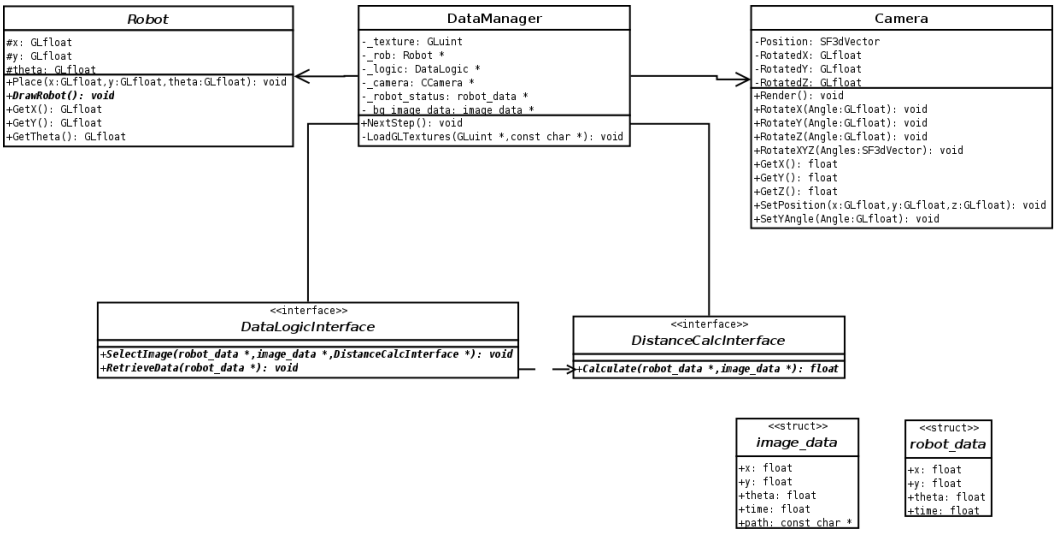


Figure 5.4: REAR class diagram

Two of the components needed by REAR, the robot model and the camera, are mapped on two dedicated classes, `Robot` and `Camera`, respectively.
The former is an *abstract class*, serves as a storage for the current position and orientation of the actual robot and declare a pure virtual method in which subclasses must include all of the procedure to draw the OpenGL model of the robot itself.
The latter is instead a *helper class*, since there exists no actual camera within the OpenGL space. The camera model is just an abstraction used to give a more intuitive way to set the portion of space the user is looking at. In fact, the `Camera` class provides methods to set the user's point-of-view within the OpenGL space (see chapter 5.2.2).
In the middle, between `Robot` and `Camera`, there is the `DataManager` class. It is intended to be the core of every REAR based application, since it internally implements the loop showed in figure 5.3. It acts as a *mediator* among all of the other classes, in that it manages their interaction and, hence, reduces coupling between them.
The diagram also features two interfaces that have to be implemented by programmers who want to create their own concrete exocentric vision system: `IDataLogic` and `IImageSelector`.
The former decouples the application core, that is the `DataManager`, from the data source. This way, `DataManager` code can be totally unaware of the technology used to retrieve data from the robot - e.g. sockets, web services, etc.
`IImageSelector`, instead, defines the interface of the component which implements

the image selection algorithm. In this case, the use of an interface leaves programmers able to change, and even to implement new image selection algorithms without having to worry about changing other classes.
Finally, the class diagram presents two structures, `image_data` and `robot_data`, whose declarations are reported in listing 5.1.
The former is used to store all the metadata relative to a specific snapshot - i.e. the position and the orientation of the camera when it was taken, a timestamp and an array of chars that can be used by programmers who would like to add other data - e.g. a code or a sequence number.
The latter is similar to the former, it is meant to be used by `DataManager` to exchange information about the current robot position and orientation with lower-level objects, in a compact way.

Listing 5.1: REAR data structures

```
struct image_data {
  float x;
  float y;
  float theta;
  float time;
  char path[100];
};

struct robot_data {
  float x;
  float y;
  float theta;
  float time;
};
```

In the following of this section, we will have a look *under the hood* of the classes and interfaces we have just introduced.
We will see how REAR functionalities are mapped onto them and, then, how to correctly subclass/use them in order to build a concrete exocentric vision system.

5.2 Classes

This section will describe the concrete classes building up REAR. Refer to figure 5.4 for a more complete vision.

5.2.1 The Robot Class

Let us have a look at the declaration of the `Robot` class, reported in listing 5.2.

Listing 5.2: `Robot` class declaration

```
class Robot
{
 private:
  GLfloat x;
  GLfloat y;
  GLfloat theta;

 public:
  Robot();
  GLfloat GetX();
  GLfloat GetY();
  GLfloat GetTheta();
  void Place(GLfloat x, GLfloat y, GLfloat theta);
  virtual void DrawRobot() = 0;
};
```

As already stated, private attributes of `Robot` are used to store the current position and orientation of the actual robot. Such attributes are initialized with all-zeros by the constructor, can be read by means of invoking their *getter* methods and can be set by means of the `Place()` method.
`Robot` also declares an abstract *hook* method, `DrawRobot()`. Such a method is called by the `DataManager` every time it wants to render the robot model and, hence, programmers who wants to use REAR for their own robot should subclass `Robot` and implement `DrawRobot()` as a procedure that draws their custom robot model using the OpenGL API.
When doing that, one must always keep in mind that REAR makes use of a three-dimensional OpenGL space, while position information is often represented as a `(x,y)` pair. `R.E.A.R.` maps the `x` value of such a pair onto the x axis of its OpenGL space, while the `y` value is mapped onto the z axis.
That is, the actual XY plan corresponds to the OpenGL XZ plan. Such a correspondence is represented in figure 5.5.
Moreover, to draw the robot in the right position within the OpenGL space, the first thing a `DrawRobot()` should do is to invoke OpenGL function `glTranslate()`.
A skeleton of a concrete implementation of `DrawRobot()` would look like this:

Listing 5.3: `DrawRobot()` skeleton

```
  glMatrixMode(GL_MODELVIEW);
```

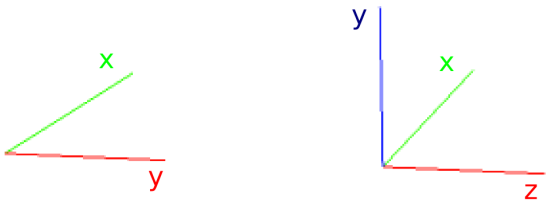


Figure 5.5: Reference systems

```
glPushMatrix();

// set robot position
glTranslatef(this -> x, 0.0f, this -> y);

// rotate the robot
glRotatef(this -> theta, 0.0f, 1.0f, 0.0f);

// code to actually draw the robot

glPopMatrix();
```

5.2.2 The Camera Class

OpenGL does not provide any *camera*, anyway it proved to be extremely useful to have a helper class that permits to easily set position and orientation of the user's point of view and sight, without worrying about lower-level OpenGL details.
A basic implementation of such a helper class is suggested in [22]. Such an implementation is slightly similar the one featured by REAR, whose declaration is reported in listing 5.4.

Listing 5.4: `Camera` class declaration

```
class Camera
{
 private:
  SF3dVector Position;
  GLfloat RotatedX, RotatedY, RotatedZ;
  GLfloat _theta;
```

```
 public:
  Camera();
  void Render ( void );
  void Move ( SF3dVector Direction );
  GLfloat GetX();
  GLfloat GetY();
  GLfloat GetZ();
  GLfloat GetTheta();
  void SetPosition(GLfloat x, GLfloat y, GLfloat z);
  void SetYAngle ( GLfloat Angle );
  void RotateX ( GLfloat Angle );
  void RotateY ( GLfloat Angle );
  void RotateZ ( GLfloat Angle );
  void RotateXYZ ( SF3dVector Angles );
};
```

The `Camera` class allows to move and rotate the user point of view, independently from the robot.
This is essential for our purposes, since REAR must be able to draw the robot everywhere in the OpenGL space, regardless of where the camera is, and vice versa.
`Camera` methods are pretty simple: they just make use of OpenGL basic commands (such as `glTranslate` and `glRotate`) to actually move the OpenGL reference system and, hence, move the point-of-view and change the sight direction.
The only thing to care about when using such a class is to invoke the OpenGL commands `glMatrixMode(GL_MODELVIEW)` and `glLoadIdentity()` before invoking its `Render()` method. This avoids previous modifications of the `GL_MODELVIEW` matrix from affecting the positioning of the camera.

5.2.3 The DataManager Class

As already stated, the `DataManager` class is the *core* of the whole framework. Its declaration is reported in listing 5.5.

Listing 5.5: `DataManager` class declaration

```
class DataManager
{
 private:
  GLuint _texture[1];
  Robot * _rob;
  Camera * _camera;
  IDataLogic * _logic;
  IImageSelector * _calculator;

  robot_data * _robot_status;
  image_data * _bg_image_data;

  /* bind the specified image to a texture */
```

```
  void LoadGLTextures(GLuint *, const char *);

 public:
  DataManager(Robot *, DataLogic *, Camera *,
              IImageSelector *);
  ~DataManager();
  void NextStep(int command = 0);
};
```

Of course, `DataManger` is meant to be used as a *singleton*, that is, there must be a unique instance of `DataManager` within the context of a REAR based application. As its name suggests, `DataManager` manages and coordinates all of the components of the exocentric vision system and, hence, keeps a private reference to all of them: the `Robot` instance (see section 5.2.1), the `Camera` instance (see section 5.2.2), a `IDataLogic` object, a `IImageSelector` object and an OpenGL texture id number. Once the application is started, it's `DataManger`'s duty to move robot and camera within the OpenGL space. Moreover, it's also responsible for 1. providing the user an interface to send commands to the robot, 2. retrieving position data from the actual robot every time the user asks for it, 3. for picking one of the available snapshots, 4. displaying it in the background.
All of these operations are performed within the scope of `NextStep()` method, reported in listing 5.6.

Listing 5.6: The `DataManager::NextStep()` method

```
void DataManager::NextStep(int command) {

  image_data old_image;

  // save metadata of the currently displayed
  // image
  old_image.x = _bg_image_data -> x;
  old_image.y = _bg_image_data -> y;
  old_image.theta = _bg_image_data -> theta;


  // send command to the robot
  _logic->Command(command);

  // retrieve new data from the robot
  _logic->RetrieveData(_robot_status);

  // move robot with _robot_status data
  _rob->Place(_robot_status->x,
              _robot_status->y,
              _robot_status->theta );

  // choose an image to set as background
  _logic->SelectImage(_robot_status, _bg_image_data,
```

```
                        _calculator);

  // if the chosen image is not the currently
  // displayed one, then  move the camera
  // into the new position and change its
  // orientation accorgindly
  if ( old_image.x != _bg_image_data -> x ||
       old_image.y != _bg_image_data -> y ||
       old_image.theta != _bg_image_data -> theta )
    {
      _camera -> SetPosition( _bg_image_data -> x,
                              0.f,
                              _bg_image_data -> y);

      _camera -> SetYAngle( _bg_image_data -> theta - 90);
    }

  // actually set the chosen image as background
  LoadGLTextures(_texture, _bg_image_data->path);
}
```

`NextStep()` has to be called every time the user wants to send a command to the robot. Such a command, encoded as an integer, has to be passed to `NextStep()` as an argument.
Then, `DataManager` asks its `IDataLogic` instance to send the command to the robot and to retrieve the new robot position and orientation, which will be stored in the `_robot_status` private attribute.
To select an image to set as background, `DataManager` invokes the `IDataLogic::SelectImage()` method, passing it the current robot position and orientation, an object of type `IImageSelector` which encapsulates the image selection algorithm and a structure of type `image_data`.
The method will fill the fields of such a structure with the metadata of the selected image, so that the `DataManager()` can pick it up, load it as a texture and display it on the background of the viewing frustum - by means of calling the `LoadGLTextures()` function.
In order to give the illusion of watching the scene from the point the selected image was taken, `NextStep()` also moves the camera and change its orientation accordingly to the image metadata.

5.3 Interfaces

This section will describe the interface that makes REAR a real framewrok. Refer to figure 5.4 for a complete vision.

5.3.1 The IDataLogic interface

`IDataLogic` provides a communication interface with the robot, decoupling the `DataManager` from the actual technology used to interact with it and to collect data from it. Its declaration is:

Listing 5.7: `IDataLogic` declaration

```
class IDataLogic {
 public:
  virtual void Command(int) = 0;
  virtual void RetrieveData(robot_data *) = 0;
  virtual void SelectImage(robot_data *, image_data *,
                           IImageSelector *) = 0;
};
```

When implementing a class of type `IDataLogic` programmers should keep in mind the followings:

- `RetrieveData()` must fill the passed robot_data structure with the retrieved position and orientation of the robot. It must also keep track of all retrieved snapshots.
- `SelectImage()` is passed the robot's current position and orientation and an object of type `IImageSelector`. In order to obtain the image to set as background (whose metadata will be saved in the `image_data` structure), an invocation of the `IImageSelector::ChooseImage()` is required (for further details, see chapter 5.3.2).

A skeleton of either `RetrieveData()` and `SelectImage()` is presented in listing 5.8.

Listing 5.8: `IDataLogic` methods skeleton

```
void DataLogic::RetrieveData( robot_data * data )
{
  // actually retrieve new data

  // store the collected snapshot into an
  // internal data structure

  data -> x = retrieved_robot_data.x;
  data -> y = retrieved_robot_data.y;
  data -> theta = retrieved_robot_data.theta;
  data -> time = retrieved_robot_data.time;
```

```
  return;
}

void DataLogic::SelectImage( robot_data * robot_status,
                             image_data * bg_image_data,
                             IImageSelector * calculator )
{
  // say we have used a vector to store
  // metadata of the collected snapshots,
  // then we could invoke the ChooseImage method

  selector -> ChooseImage( robot_status, bg_image_data,
                           &_images_collection );
}
```

5.3.2 The IImageSelector interface

Last, but not least, the `IImageSelector` interface defines the type of classes that encapsulate an image selection algorithm. It declares just a method, that we have already encountered in the previous section:

Listing 5.9: `IImageSelector` declaration

```
class IImageSelector
{
 public:
  virtual void ChooseImage(robot_data *, image_data *,
                           std::vector<image_data> *) = 0;
};
```

`ChooseImage()` is passed the current robot position and orientation and a vector of images metadata. It will process them and will fill the passed `image_data` structured fields with the metadata of the selected image.
Say we want to define a class that implements the selection method 2 presented in [10]. That selection method chooses the image taken as near as possible to the robot current position.
The `Calculate()` function, called within the following code, estimates the euclidean distance between robot and a generic image. If we decided to choose the closest image to the robot, we would show the egocentric vision, because the algorithm would always select the image coupled with distance equal to zero. We remember that the egocentric vision image is always present in our images' set.
In order to not show only the egocentric vision, the `ChooseImage()` does not return the image coupled with the minimum distance, but the one (if available) with the minimum distance greater than zero.
We could, then, write:

Listing 5.10: A possible `ChooseImage` implementation

```
class SpacialMetricSelector : public IImageSelector
{
  public:
  void ChooseImage(robot_data *, image_data *,
                   std::vector<image_data> *);
};


void SpacialMetricSelector::
      ChooseImage( robot_data * robot_status,
                   image_data * bg_image_data,
                   std::vector<image_data> *
                       _images_collection)
{

  float distances[_images_collection->size()];
  float min;

  // calculate the distance for each stored image
  int i = 0;
  for (std::vector<image_data>::iterator it =
         _images_collection->begin();
       it != _images_collection->end();
       it++)
    {
      distances[i] = Calculate(robot_status, &*it);
      i++;
    }

  // find the minimum distance
  i = 0;
  min = distances[0];
  for (int j = 1; j < _images_collection->size(); j++)
    {
      if ( ( distances[j] < min && min != 0) ||
           ( distances[j] > min && min == 0)     )
        {
          i = j;
          min = distances[j];
        }
    }

  // return the selected image data
  bg_image_data->x = (*_images_collection)[i].x;
  bg_image_data->y = (*_images_collection)[i].y;
  bg_image_data->theta = (*_images_collection)[i].theta;
  bg_image_data->time = (*_images_collection)[i].time;
}
```

Chapter 6

An exocentric vision system for Morduc

Contents

6.1 Classes inherited from Robot class 63
6.1.1 The Morduc Class 63
6.2 Classes implementing IDataLogic interface 66
6.2.1 The DataLogicLogSimulator Class 66
6.2.2 The DataLogicLogMorduc Class 69
6.2.3 The DataLogicMorduc Class 70
6.3 Classes implementing IImageSelector interface 71
6.3.1 The spacial metric algorithm 71
6.3.2 The SpacialMetricCalc class 72
6.3.3 The sweep metric algorithm 73
6.3.4 The SweepMetricCalc class 78
6.3.5 Another sweep metric algorithm 79
6.3.6 The AnotherSweepMetricCalc class 81

This section will show how to use REAR to implement a concrete exocentric vision system, specifically designed for Morduc.
As section 5 explains, in order to get things done, all we need to write is:

- a concrete subclass of `Robot`
- an implementation of the `IDataLogic` interface
- an implementation of the `IImageSelector` interface

The class diagram of the complete system is reported in figure 6.1.
As it's easy to notice, it features two implementations of the `IImageSelector` interface: `SpacialMetricCalc`, `SweepMetricCalc` and `AnotherSweepMetricCalc` (an improvement algorithm from the previous one).
`SpacialMetricCalc` is a variant of the image selection method number 2 reported in [10] and whose possible implementation is showed in sections 6.3.1 and 6.3.2.
The latters, instead, encapsulates a brand new selection algorithm. It will be widely discussed later from section 6.3.3 to section 6.3.6.

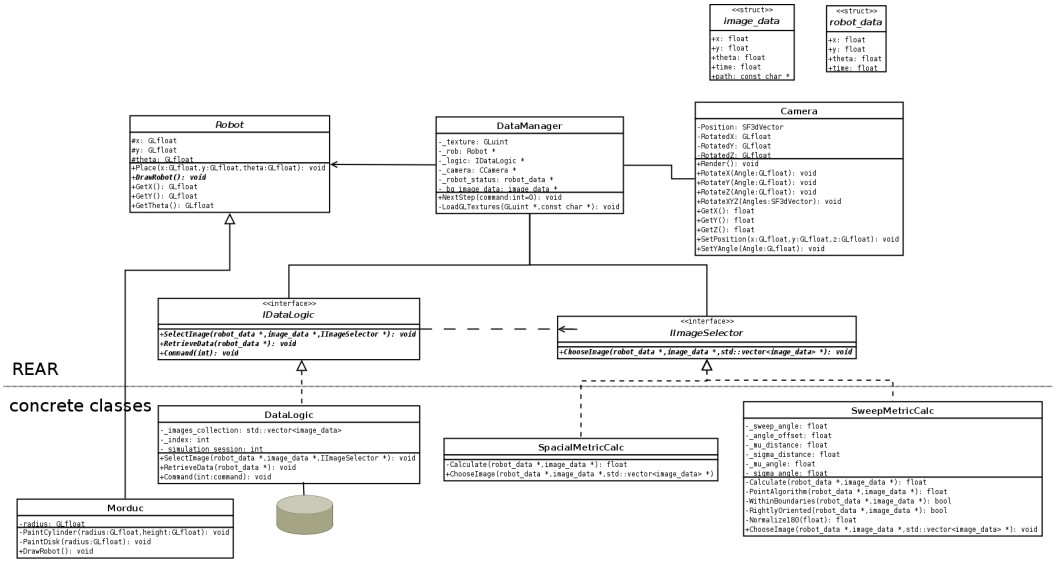


Figure 6.1: Class diagram of the whole system

6.1 Classes inherited from Robot class

In this chapter will be exposed classes which inherits from *Robot* class. They have to implement, at least, the *DrawRobot()* pure virtual father's method, because REAR can not know in other way how to represent the robot to overlap on the background image chosen.

6.1.1 The Morduc Class

This class allows to draw the robot when data and images are retrieved from the log files created with the simulator program (see chapter 3).

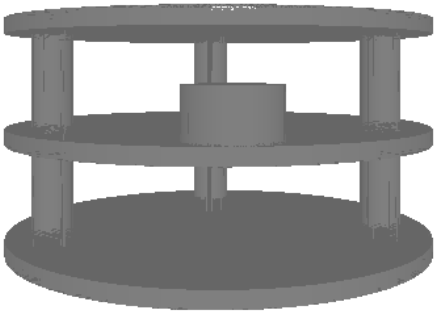


Figure 6.2: Three dimensional model of Morduc

The `Morduc` class publicly inherits from `Robot` and, hence, implements the `DrawRobot()` method:

Listing 6.1: `Morduc` class declaration

```
class Morduc : public Robot
{
 private:
  GLfloat radius;
  void PaintCylinder(GLfloat radius,GLfloat height);
  void PaintDisk(GLfloat radius);

 public:
  Morduc(float radius = 4.0f);
  void DrawRobot();
};
```

Such a method, reported in listing 6.2, contains the procedure to actually draw the OpenGL model representing Murdoc (in reality, Murdoc looks like a stack of three disks linked by thin cylinders - see figure 6.2).

Listing 6.2: `Morduc::DrawRobot()` function

```
void Morduc::DrawRobot()
{
  GLfloat reflectance_black[] = { 0.2f, 0.2f, 0.2f};
  GLfloat reflectance_white[] = { 0.8f, 0.8f, 0.8f};

  glMatrixMode(GL_MODELVIEW);
  glPushMatrix();

  // set robot reflectance (it is black)
  glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT,
               reflectance_black);

  // set robot position
  glTranslatef(this -> x, 0.0f, this -> y);

  glRotatef(this -> theta, 0.0f, 1.0f, 0.0f);

  // translate on z axis
  // not to make the robot wipe the floor
  glTranslatef(0.0f,0.08f,0.0f);

  glScalef(radius, radius, radius);

  // draw robot
  PaintCylinder(1.0f, 0.1);
  PaintDisk(-1.0f);
  glTranslatef(0.0f, 0.1f, 0.0f);
  PaintDisk(1.0f);

  glTranslatef(0.0f, 0.6f, 0.0f);

  PaintCylinder(1.0f, 0.1f);
  PaintDisk(-1.0f);
  glTranslatef(0.0f, 0.1f, 0.0f);
  PaintDisk(1.0f);

  glTranslatef(0.8f, 0.0f, 0.0f);
  glColor3f(0.5f, 0.5f, 0.5f);
  PaintCylinder(0.2f, 0.3f);
  glTranslatef(0.0f, 0.3f, 0.0f);
  PaintDisk(0.2f);

  glTranslatef(0,0.401,0);
  glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT,
               reflectance_white);
  PaintDisk(0.1f);
  glTranslatef(0,-0.701,0);
  glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT,
```

```
                reflectance_black);

  glTranslatef(-0.8f, 0.0f, 0.0f);
  glColor3f(0.1f, 0.1f, 0.1f);
  glTranslatef(0.0f ,0.6f, 0.0f);
  PaintCylinder(1.0f, 0.1f);
  PaintDisk(-1.0f);
  glTranslatef(0.0f, 0.1f, 0.0f);
  PaintDisk(1.0f);

  glTranslatef(0.0f, -1.5f, 0.0f);
  glTranslatef(0.0f, 0.0f, 0.8f);
  PaintCylinder(0.1f, 1.5f);
  glTranslatef(0.0f, 0.0f, -1.60f);
  PaintCylinder(0.1f, 1.5f);
  glTranslatef(-0.8f, 0.0f, 0.8f);

  PaintCylinder(0.1f, 1.5f);
  glTranslatef(0.8f, 0.0f, 0.0f);

  glScalef(1/radius, 1/radius, 1/radius);

  glPopMatrix();
}
```

Notice how such a function implements the skeleton presented in listing 5.3.
As you have surely noticed, `Morduc` contains also some private attributes, precisely two methods - `PaintCylinder()` and `PaintDisk()` - and a float field - `radius`.
The two methods are just *helper functions*, that is, functions that contains a procedure to draw, respectively, a *disk* and a *cylinder*. `radius`, instead, stores the value of the radius for the of the cylinders which make up the robot. After some calibration, the default `radius` value has been set to 4.0.

6.2 Classes implementing IDataLogic interface

In this chapter will be exposed classes which implements from *IDataLogic* interface (exposed in details in chapter 5.3.1).
They differ on the way images and robot's status data are collected: a concrete class will use log file saved in a prefixed location on disk (by using session log data), another one will ask the server to send new images and data thought Internet network, to implement a real-time teleguide.

6.2.1 The DataLogicLogSimulator Class

For the reasons explained in section 3, we used a Morduc simulator in order to generate and record test data to use during *off-line* tests.
For every simulation session, such data consist of:

- a set of captured images, encoded in png format, with 632x453 pixel for each
- a log file reporting sampled odometry data for all the session

The format of the log file is the one described in section 3.
The concrete `DataLogicLogSimulator` class will, then, provide robot's odometry data and snapshots simply by reading those files.
Let us have a look at its declaration:

Listing 6.3: `DataLogic` declaration

```
class DataLogicLogSimulator : public IDataLogic
{
 private:
  std::vector<image_data> _images_collection;
  int _index;
  int _simulation_session;

 public:
  DataLogicLogSimulator(int);
  ~DataLogicLogSimulator();
  void Command(int);
  void RetrieveData(robot_data *);
  void SelectImage(robot_data *, image_data *,
                   IImageSelector *);
};
```

Upon instantiation, `DataLogicLogSimulator` objects needs to be configured with a session identifier, to be passed to the constructor.
In order for the `DataLogicLogSimulator` objects to find them, log data must be stored within the relative path `../log`, with respect to the actual execution path. Such a directory must contain a single subdirectory for every logged session, named `log_`⟨`number of session`⟩.

Let us now have a look at how `DataLogicLogSimulator` implements `IDataLogic` methods:

Listing 6.4: `DataLogic::Command()` method

```
void DataLogicLogSimulator::Command(int command)
{

  // sends the command to the robot

  // in our case, just
  // increase index to point the next line of the file
  _index++;
}
```

The `Command()` method is pretty simple in this implementation. Since `DataLogicLogSimulator` has been designed for offline testing, there's no actual command to send to the robot. For this reason `Command()` will not take into account the received command, the only thing that it does is incrementing the private attribute `_index`, which will be used by `RetrieveData()` to select which line of the log file is to read next.

Listing 6.5: `DataLogic::RetrieveData()` method

```
void DataLogicLogSimulator::RetrieveData(robot_data * data)
{
  // pointer to text file containing data
  FILE * position_data;
  char line[50];

  std::string * line_read;
  std::string line_values[4];
  std::string position_data_name;
  std::ostringstream o;

  int time;
  int line_number = _index;

  // grabbed image metadata
  image_data grabbed_frame_data;

  o << "../log/log_"
    << _simulation_session
    << "/data_"
    << _simulation_session << ".txt";

  position_data_name = o.str();

  position_data = fopen(position_data_name.c_str(), "rt");

  while(fgets(line, 50, position_data) &&
```

```
        line_number > 1)
    {
      line_number--;
    }

  line_read = new std::string(line);

  std::string buf;
  std::stringstream ss(*line_read);

  // Create vector to hold our words
  std::vector<std::string> tokens;

  // put token in vector element
  while (ss >> buf)
    tokens.push_back(buf);

  data->x = atof ( tokens[0].c_str() );
  data->y = atof( tokens[1].c_str() );
  data->theta = TO_DEGREES(- atof ( tokens[2].c_str() ));
  data->time = atof ( tokens[3].c_str() );

  time = (int) data->time;

  // clear the stream and
  // add a new value to it
  o.str("");
  o.clear();
  o << "../log/log_"
    << _simulation_session
    << "/screenshot_"
    << _simulation_session
    << "_" << time << ".png";

  // fill grabbed frame metadata
  grabbed_frame_data.x = data->x;
  grabbed_frame_data.y = data->y;
  grabbed_frame_data.theta = data->theta;
  grabbed_frame_data.time = data->time;

  strcpy(grabbed_frame_data.path, o.str().c_str());

  // store the collected metadata
  // if it's not already stored
  for (std::vector<image_data>::iterator it =
          _images_collection.begin();
       it != _images_collection.end();
       it++)
    {
```

```
      if ( (*it).time == grabbed_frame_data.time )
        {

          return;
        }
    }

  _images_collection.push_back(grabbed_frame_data);
  return;
}
```

What the `RetrieveData()` do is just read the line from the log file identified by the `_index` private attribute of the class. Once read, the line is *parsed* and the odometry data contained in it is used to fill the `robot_data` passed as argument, in order to return the *current* robot position.
Behind the curtain, `RetrieveData()` also creates a new `image_data` structure, fills it with the data just gathered and then adds it to the `_image_collection` private attribute. This way, every time a new line of the file is read, and, hence, every time a new captured image is available, the `DataLogicLogSimulator` is made able to keep track of it using its metadata, without actually loading the image - since it would be useless and time-consuming.

Listing 6.6: `DataLogic::SelectImage()` method

```
void DataLogicLogSimulator::
SelectImage(robot_data * robot_status,
            image_data * bg_image_data,
            IImageSelector * selector)
{

  // since our data are already stored with vector,
  // we simply pass its reference
  selector -> ChooseImage(robot_status,
                          bg_image_data,
                          & _images_collection);
}
```

Finally, `SelectImage()` calls the `ChooseImage` method on an object of type `IImageSelector`, and passes it, as last parameter, a reference to its internal `_images_collection`, so that the selector can actually select the image to set as background.

6.2.2 The DataLogicLogMorduc Class

As the one exposed before, *DataLogicLogMorduc* class fetches new images and data from static textual file and images stored in a fixed directory, but in this case data are formatted as returned by the 3morduc's server in a real-time session. Indeed, this class was created to develop in off-line mode the class *DataLogicMorduc*, which by

Internet connection makes possible teleguiding the real robot (see following chapter 6.2.3).
For every simulation session, we dispose of:

- a set of captured images, encoded in jpeg format, with 640x480 pixels each
- a log file for each image, reporting sampled odometry data

So, in the proper folder, there will be jpeg format file named *screenshot_<uniquenumber>.jpg* and the coupled robot's data status in *data_<uniquenumber>.txt*. As you can notice, information are stored differently from one collected by the simulator (exposed in chapter 3), so we need another concrete class implementing *IDataLogic* interface (5.3.1).

6.2.3 The DataLogicMorduc Class

This class allows REAR to teleguide the real 3morduc robot, situated at DIEES laboratory, in Catania (Italy).

6.3 Classes implementing IImageSelector interface

As already said, there exists many image selection algorithms. Three of them are presented in [10] and, according to the results presented, two proved to be unsatisfactory for they do not produced results that could effectively improve the quality of the operator-robot interaction in situations in which the robot moves along complex trajectories.
A third one is told to be better but, according to us, it has been described poorly and, furthermore, no actual implementation of it is given.
We, then started an investigation on image selection algorithms. The rest of this chapter will present the theoretical results of our investigations: specifically, it will introduce three algorithms, which have been implemented by means of the definition of three subclasses of the `IImageSelector` interface (exposed in details in chapter 5.3.2).
All classes use a `Calculate` method to assign a *score*. Such a score is computed according to a specific *metric*, that is, a function that, given a pair <`robot_data, image_data`>, returns a number.
The `robot_data` identifies the robot current position, while the `image_data` contains the position in which the image we want to calculate the score of, was taken.
In all cases, after assigning a score to every collected image, the one with the lower score will be returned (or better, assigned to the `image_data` pointer passed to `IImageSelector::ChooseImage` method).
This way, it is possible to find which of the snapshots is the *closest* (according to the chosen metric) to the current robot position.

6.3.1 The spacial metric algorithm

The spacial metric algorithm is a slightly different variant of the selection method 2 presented in [10].
The `SpacialMetricCalc` class, which encapsulates the algorithm, makes use of the Euclidean metric, that is, its `Calculate` method returns a value that depends from the spacial distance between the current robot position and the position in which the snapshot was taken.
Once all spacial distances have been calculated, such values are passed as input to the *triangle* function showed in picture 6.3.
Since the `ChooseImage` method will select the image with the minimum score, it's easy to figure out that images that has been taken when camera was around 5 spacial units distant from the current robot position are most likely to bet chosen.
The motivation lying the algorithm is the following: if the euclidean distance of the snapshots from the current robot position is around zero, using such images as background will result in displaying only a part of the 3D model of the robot. If, instead, such a distance is very high, the model will be drawn very far from the camera's (and user's) point-of-view. There, hence, exists, an *optimal distance* that allows to draw the robot entirely, but not too far from the camera's point-of-view.

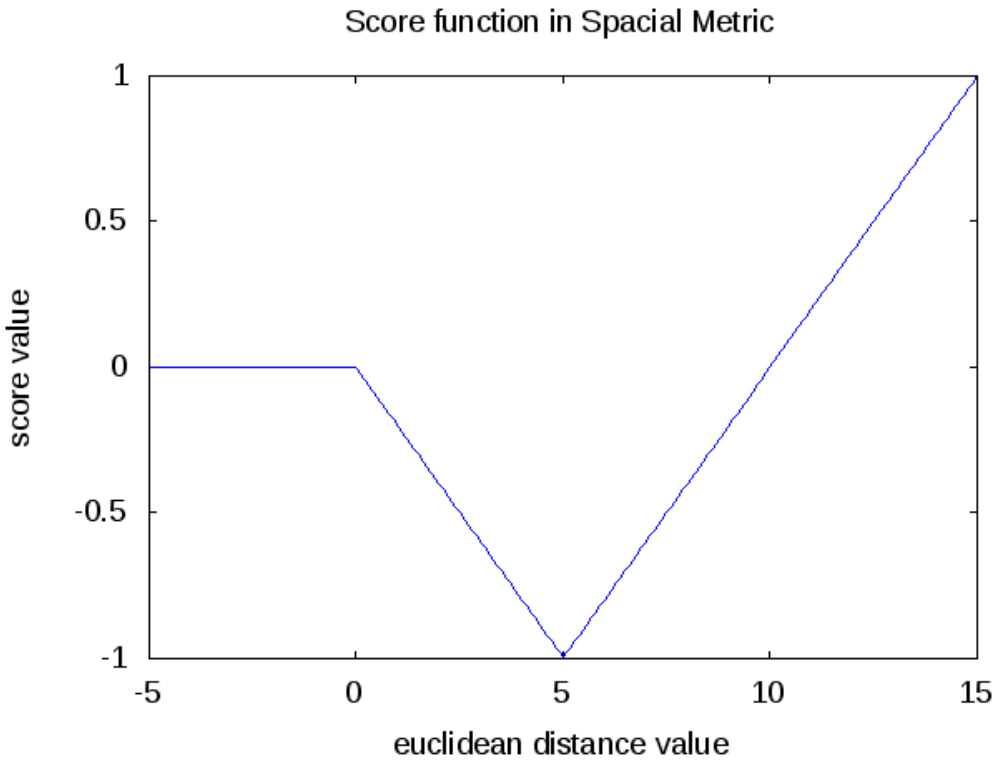


Figure 6.3: Spacial Metric Function

On the one hand, this algorithm is very simple to implement. On the other hand, it does not care about image orientation. It could, for instance, choose an image that does not include the robot from its point of view. This limit makes the algorithm good only for situations in which the robot moves along straight trajectories. The *Sweep Metric Algorithm*, presented section 6.3.3, and its advanced version, *Another Sweep Metric Algorithm* (section 6.3.5) will try to go beyond this limit.

6.3.2 The SpacialMetricCalc class

The `SpacialMetricCalc` class encapsulates the algorithm presented in the previous section. The value of the *optimal* distance is hardcoded in the `Calculate` function:

Listing 6.7: `SweepMetricCalc` class declaration

```
float SpacialMetricCalc::
          Calculate(robot_data * robot_status,
                    image_data * bg_image_data) {

  float distance;
  float score;
```

```
  float optimal_distance = 20;

  distance =
    sqrt( pow((robot_status -> x ) -
              (bg_image_data -> x), 2) +
          pow((robot_status -> y ) -
              (bg_image_data -> y), 2) );

  if ( distance <= optimal_distance )
    score = distance / optimal_distance;
 else
    score = - ( distance - 2 * optimal_distance) /
            optimal_distance;

  return (- score);
}
```

It is not a good solution since, every time one wants to change such a values should recompile the code. On the other hand, this is just a prototype of an algorithm we abandoned almost immediately. That is why we did not make any code improvement.

6.3.3 The sweep metric algorithm

The *Sweep Metric Algorithm*, encapsulated into the `SweepMetricCalc` class, aims at being a more complete and flexible image selection algorithm than the one we analyzed in the previous section. Let us assume to have mobile robot moving on a plan, taking snapshots from its egocentric-mounted camera. Such a situation is represented in figure 6.4:
the black square represents the robot, with its orientation indicated by the arrow "a", starting from the square.
The several blue circles represent instead the previous taken images. The orientation of each image - i.e. the orientation of the robot when they were taken - is shown by an arrow starting from each circle.
We will refer to the line normal to the half line 'a' as 'b'. By rotating clockwise and counterclockwise the 'a' line in the robot centre with a predefined angle (named 'sweep angle') we will obtain the 'c' and 'd' lines. These define a new portion of the plane (colored with fading red), namely the *sweep area*.
Since we would like to see the robot from its rear, all images taken within this area will be taken into account when selecting the image to set as background. The other ones will be discarded.
Then, we have to discard all the images with an orientation angle that differs too much from the robot current orientation angle. If the difference between the two angles is greater than a specified threshold, the image would make the robot being not included within the viewing frustum.
For instance, if we choose an image whose difference angle with the robot current

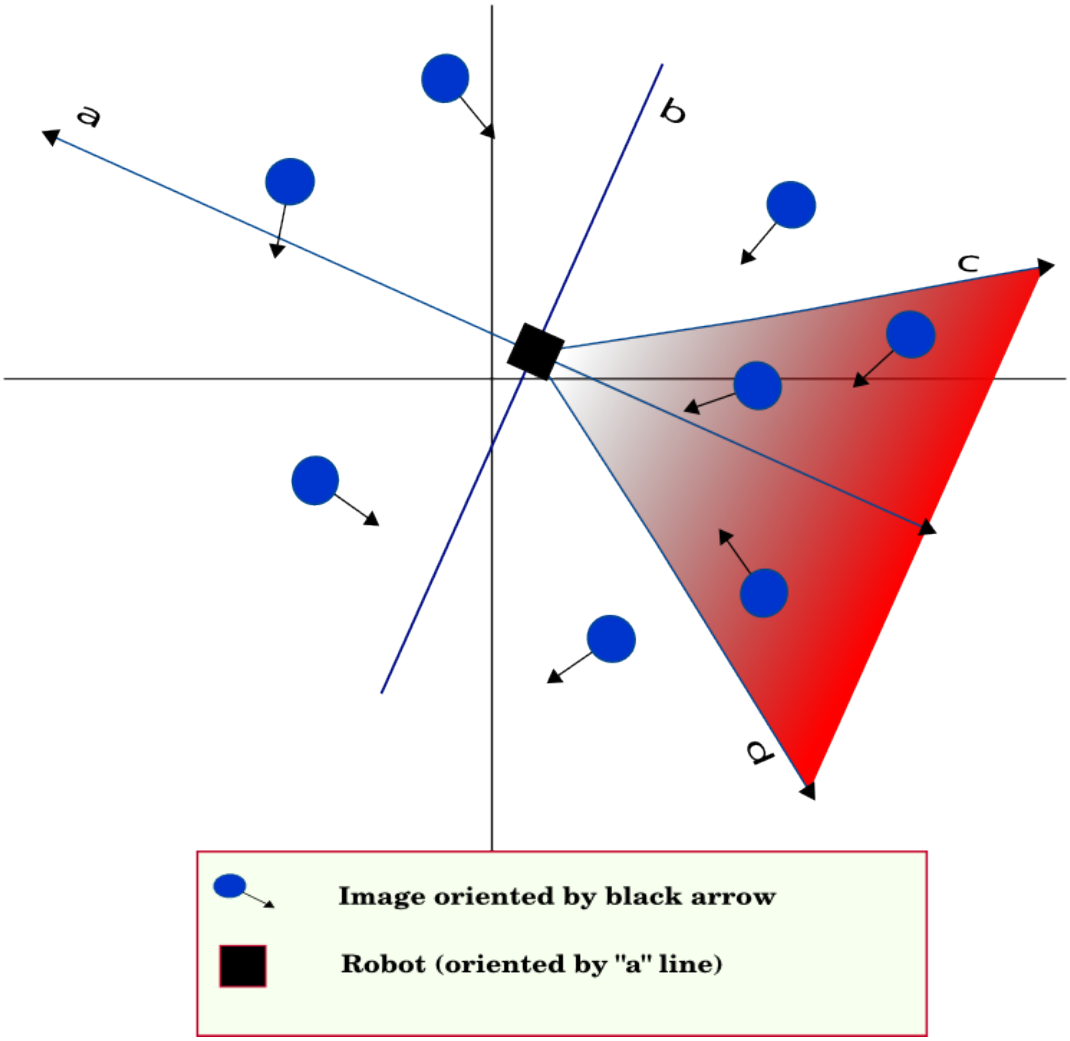


Figure 6.4: Sweep Angle Algorithm

orientation is 180 degrees, it means that robot and the camera will be oriented in opposite ways, therefore the camera will not see the robot.
To recognize all discarded images, the `Calculate` method assigns them -2 as score. The score of all the remaining image is computed as sum of two factors: the first takes into account the image angle orientation: the more the image orientation angle is close to the robot orientation angle, the more the score is high.

To compute such a factor, a Gaussian function, centered in zero, is used. The return value will be therefore always a positive number. The use of a Gaussian function allows to obtain different values even for two very close angle difference, regardless of its variance. Moreover, since Gaussian is a surjective function, it is defined for all real numbers. This way, images with a little orientation difference with the robot

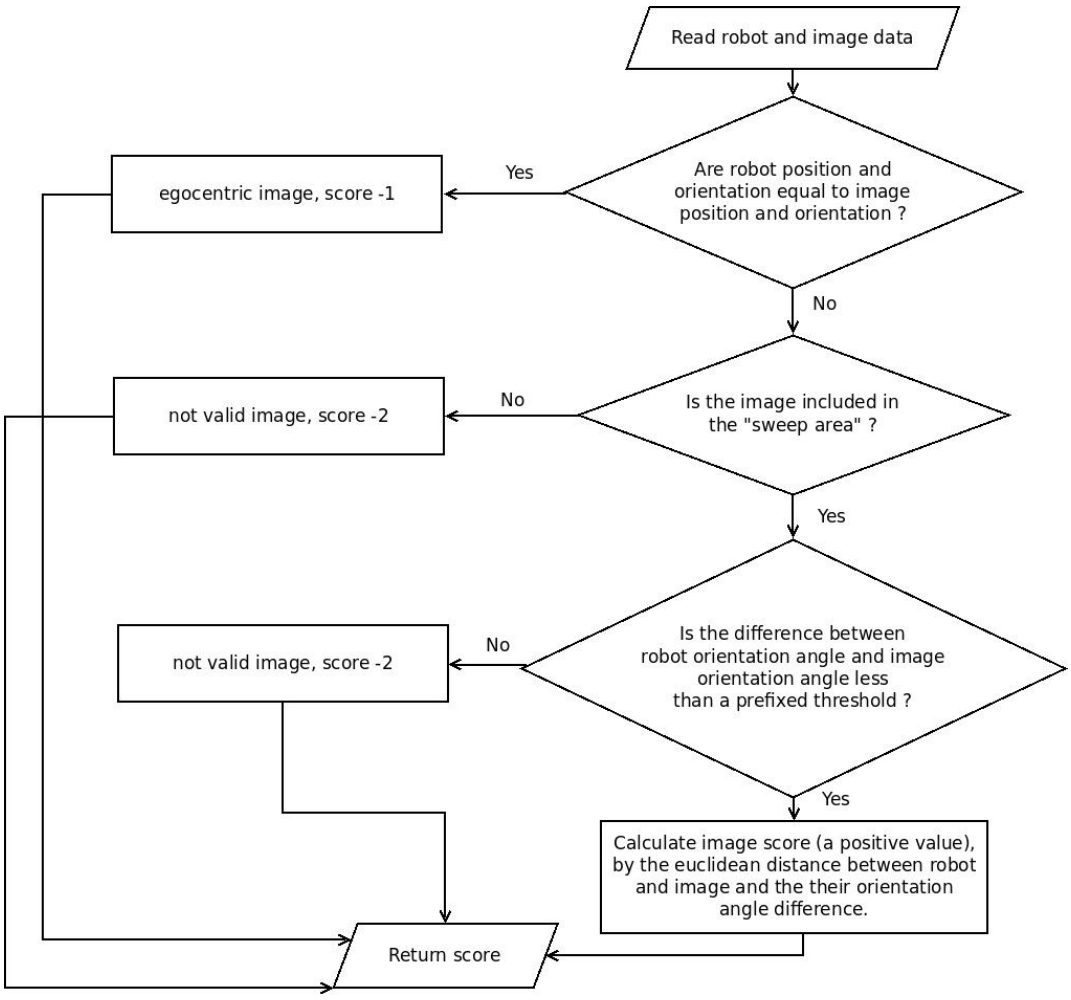
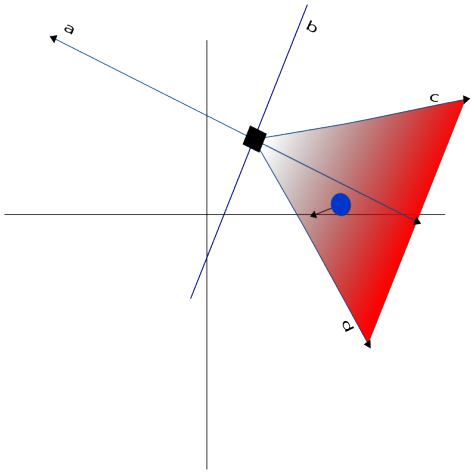


Figure 6.5: Sweep angle diagram

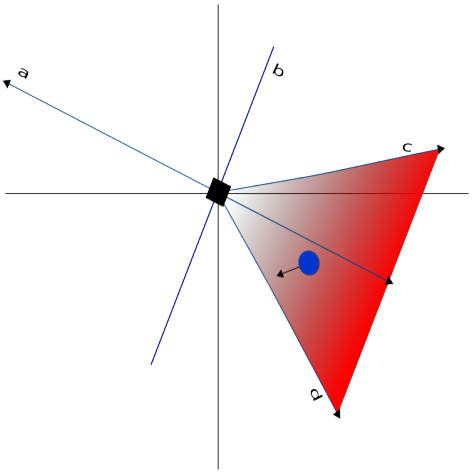
current heading will be assigned a higher value.
The second factor is computed taking into account the Euclidean distance between the position in which the image was taken and the robot current position. The approach followed here is the same of the *spacial metric algorithm* presented in the previous section: given an *optimal* distance, the more the image position is close to such a distance, the more the score assigned to it will be high. This time, instead of using a *triangle* function, we will use a Gaussian function, with a given variance and whose centre will be the optimal distance.
If the image position and orientation coincide with robot position and orientation - i.e. the Euclidean distance between the image and the robot is zero, the image represents the egocentric vision. The score coupled with the egocentric vision will always be -1.

Finally, all computer scores are multiplied by -1 and the image with the littlest score is returned.
If the sweep area does not contain any valid image, all images will have 2 as score. In such a situation, since the `ChooseImage` search the image with the lowest score, the egocentric image will be chosen.

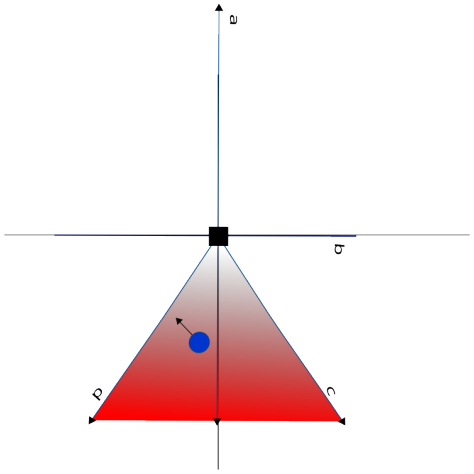
The WithinBoundaries algorithm Checking if an image is included within the *sweep area* could be something tricky to implement. Given image coordinates, robot coordinates and robot orientation, the `WithinBoundaries` method of the `SweepMetricCalc` class will answer (with a true or false response) the question '*is the image included within the sweep area ?*'.
Since the area defined by the 'sweep angle' depends on robot coordinates and orientation, the `Within Boundaries` performs some geometrical tricks in order to give its answer. Such tricks deserve some space to be fully explained.
Before checking whether a point (i.e. an image) is included within the *sweep area*, the robot and image coordinates are translated and then rotated, in order to move the robot in the origin of the axis and to overlap its orientation arrow with the y-axis. The two transformations are shown in figure 6.6.b and 6.6.c, while the robot and image starting coordinates are shown in figure 6.6.a.
After executing these transformations, the sweep area is always situated in the second and third quadrant, as shown in figure 6.6.c. Now the 'c' and 'd' lines pass both from the origin.
If we draw a circle centered in the axis origin, the intersection between the circle and the 'c' and 'd' lines will return four points in the plan. Among these, the points situated in the second and third quadrant define a triangle with the centre of the XY axis: this will be our "sweep area", where to check if an image is included or not. Note that we choose the points with negative Y value (named "A" and "B") due to the translation and rotation operated before. Again, see figure 6.6.d for a graphical example.
The circle radius must be a value large enough to include a wide number of images. In our case we defined it with a value of five hundred, in order to reduce the difference between the abstract 'sweep area' and the actual triangle 'AOB' (figure 6.6.d) used to simplify the algorithm.
We have now reduced the problem to check whether a point lies on the *sweep area* to a well-known problem: checking if a point is included within triangle [23]. The better (and more rapid) way to resolve such problem exploits the cross product between vectors in three-dimensional Euclidean space.
Referring to the image 6.7, the cross product of [A-B] and [A-p] will result a vector pointing out of the screen. On the other hand, the cross product of [A-B] and [A-p'] will result a vector pointing into the screen.
The cross product of [A-B] with the vector from A to any point above the segment AB turns out with a resulting vector points out of the screen, while using any point below AB yields with a vector pointing into the screen. We have to distinguish



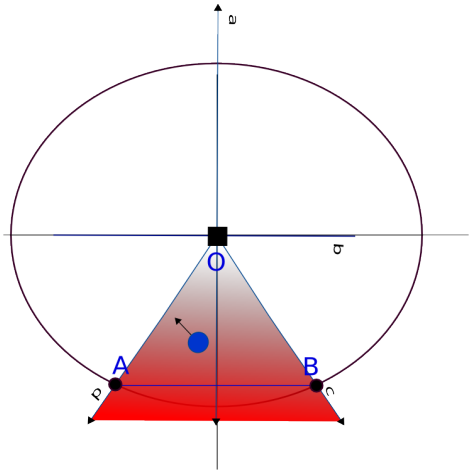
6.6.a Initial robot and image coordinates



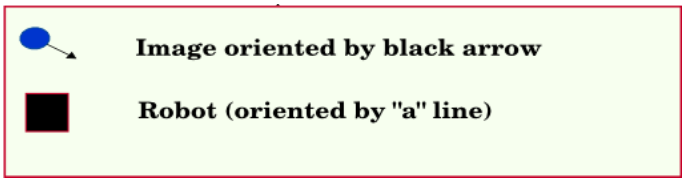
6.6.b Translated system



6.6.c Rotated system



6.6.d Triangle AOB



6.6.e Symbol definition

Figure 6.6: WithinBoundaries algorithm

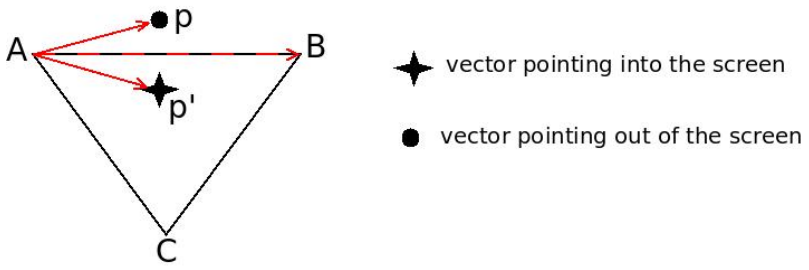


Figure 6.7: Cross product between vectors

which direction a resulting vector must have in order to consider the point 'p' inside the triangle.
Because the triangle can be oriented in any way, what we need is a reference point, that is a point that we know is on a certain side of the line. For our triangle (figure 6.7), this is just the third point C.
Any point 'p', where [A-B] cross [A-p] does not point in the same direction as [A-B] cross [A-C], is not inside the triangle. If the cross products do point in the same direction, then we need to test 'p' with the other lines as well. If the point was on the same side of AB segment as C, and is also on the same side of BC segment as A, and on the same side of CA segment as B, then it is in the triangle.
The main disadvantage regarding the approach above described is that the sweep angle value must be strictly greater than zero and strictly less than ninety degrees. If the sweep angle exceeds previous limits the algorithm will executes with a wrong triangle AOB (we remember that AOB angle, shown in figure 6.6.d, is equal to twice the sweep angle).

6.3.4 The SweepMetricCalc class

Once we are done with the description of the algorithm, let us see how to actually use the `SweepMetricCalc` class.
All is needed in order to use the class, is to instantiate it, using its conversion constructor:

Listing 6.8: `SweepMetricCalc` class declaration

```
SweepMetricCalc::SweepMetricCalc( float sweep_angle,
                                  float angle_offset,
                                  float mu_distance,
                                  float sigma_distance,
                                  float mu_angle,
                                  float sigma_angle );
```

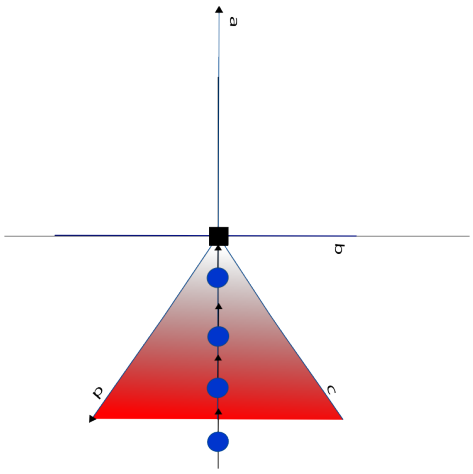
Here is a brief description of the constructor parameters:

- `sweep_angle`
  half the angle which defines the *sweep area*

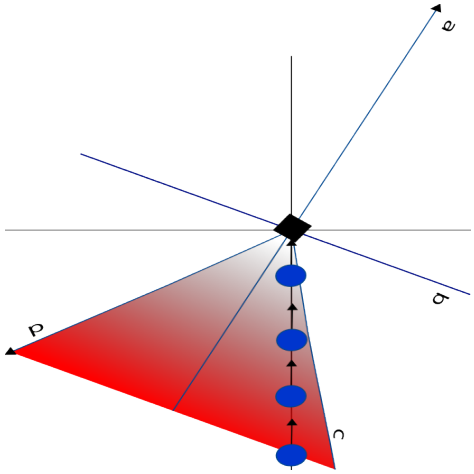
- `angle_offset`
  maximum difference allowed between robot and image orientation
- `mu_distance`
  expected value (i.e. mean value) for the Gaussian which assigns the score on the basis of distance between image and robot
- `sigma_distance`
  standard deviation for the Gaussian which assigns the score on the basis of distance between image and robot
- `mu_angle`
  expected value (i.e. mean value) for the Gaussian which assigns the score on the basis of orientation difference between image and robot
- `sigma_angle`
  standard deviation for the Gaussian which assigns the score on the basis of orientation difference between image and robot

6.3.5 Another sweep metric algorithm

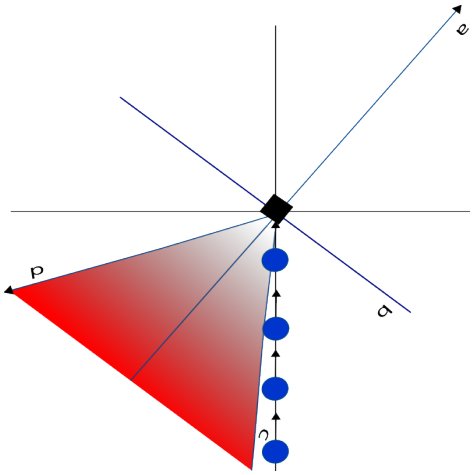
After performing some tests with the *Sweep Metric Algorithm* presented above, one of its major shortcomings detected was the immediate and swift change from exocentric to egocentric point view, caused by turning the robot more then 45 degrees respected to the previous direction.
Further details about test ran and related results can be found in section 7.
Another sweep metric algorithm, for simplicity's sake also called *ASM Algorithm*, was born to provide a better and more comfortable way of guiding the robot.
First of all, we have to define where the previous algorithm failed. Image to drive the robot along a straight direction, until REAR collects a sufficient number of images to present you the robot with an artificial exocentric point of view. This case can be summarized by figure 6.8, which resumes the graphical notation used to explain the original algorithm (see figure 6.4 and its legend). We recall that the blue circles are the egocentric images collected by the robot (with their orientation shown by the black arrow), whereas the black square indicate the robot orientated by the 'a' axis. The read area indicates the sweep angle.
In 6.8.a the robot is correctly drawn by REAR and user can control it from an exocentric point of view, thanks to the previous collected images. When robot begins to turn, for narrow angles of rotation the images are still included in the sweep area, so REAR draws the robot while it is turning (figure 6.8.b).
When the rotation angle begin too large, all the previous collected images become invalid, as shown by figures 6.8.c and especially 6.8.d. For this reason REAR provide immediately the egocentric point of view to the user, but this sudden change of point of view causes disorientation to the teleoperator, who often is no more able to proper collocate the robot in the remote environment. The positive effective of the exocentric vision is at once lost, bringing instead only negative consequences.



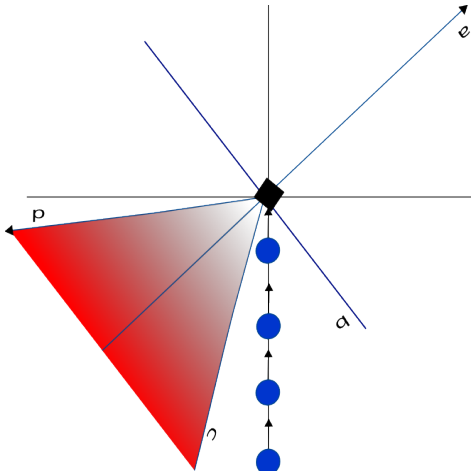
6.8.a Robot and data collected after moving along a straight direction



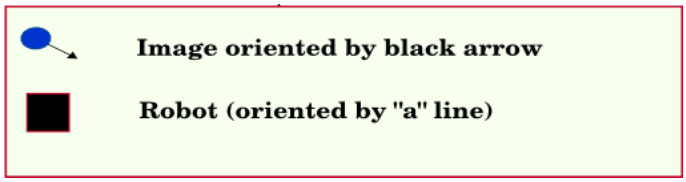
6.8.b Robot begin to rotate, images collected are still valid (i.e. within the sweep area)



6.8.c Keeping on turning, images collected are no more valid.



6.8.d Images collected are no more valid.



6.8.e Symbol definition

Figure 6.8: The main shortcoming in Sweep Metric Algorithm

The solution proposed by ASM is simple: if the robot turns, the sweep angle area is maintained exactly the same as it was before the robot changed its direction. In other words, ASM finds the proper image evaluating robot position before one or more consecutive turns are performed.
Referring to the example shown in figure 6.8, all the turns performed by the robot are seen by the teleoperator from the same point of view, since the algorithm takes into account always the same sweep area and therefore the same images.
When the robot complete its turnings sequence, ASM resumes to work exactly as it did with the original *Sweep Metric Algorithm*, but this time it has at least one image (the last before going forward) from which draw the robot. The latter allows REAR to not show immediately the egocentric point of view, but to provide another exocentric point of view, even though with a coupled distance from the robot which will be surely far from the optimal one.
Howsoever, user's sense of disoriention and confusion are heavily decreased, in particular on strict robot turning.

6.3.6 The AnotherSweepMetricCalc class

AnotherSweepMetricCalc is almost equal to its ancestor. The constructor's parameters are the same (with the same meanings) and there are no special supporting methods.
The only difference relies within the *ChoseImage()* method. As explained in section 5.3.2, the latter takes as input a collection of images, shot during robot's movement. Before proceeding, the reader is warned that a good comprehension of the general score method exposed earlier in this chapter (section 6.3) is required.
The only assumption ASM class does is that images are disposed in the transfered vector following a temporal order, from the older one to the most recent. Therefore, before assigning a score to every image according to the algorithm seen in *Sweep Metric Algorithm*, a simple pre-elaboration code assigns the higher possible score to the last images shot while the robot is turning, in order to exclude them from the feasible ones.
The robot status is at the moment set as it was before the sequence of turns was acted, and then the standard algorithm is resumed to assign a score to the remaining images.
These simple steps allows to greatly improve the original algorithm.

Chapter 7

Performance Evaluation

Contents

7.1 Parameters **84**
7.2 Test preparation **86**
7.3 Tests results **87**
7.3.1 Rectangle test evaluation 87
7.3.2 Ellipse test evaluation 88
7.3.3 Broken lines test evaluation 88
7.4 Final considerations **91**

This chapter will describe some tests performed at the University of Hertfordshire (Hatfield, UK), along with their results.
The algorithm presented in 6.3.3 (named *SweepMetricAlgorithm*) was used as the image selection algorithm. Several tests have been carried out, each time changing the passed parameters (see 6.3.4).
The test target is to underline the main differences occurring when one or more parameters change their values, within defined ranges. After asking users their options and perceptions about teleguiding the robot with an exocentric vision system, we could at the end identify the advantages and the disadvantages correlated with different sets of initial parameters.
Since tests have been carried out using saved logs (see section 6.2.1), users were not able to actually command the robot, but simply to request the application to go one step further and, hence, just see the robot moving along the trajectory it performed during a recorded session.
Be advised that, due to the low number of testers, the obtained results have not statical validity. On the other hand, they were useful to identify a possible future improvement of the algorithm, built later with *Another Sweep Metric Algorithm* (see section 6.3.5).

7.1 Parameters

Testing the 'sweep metric algorithm' (see chapter 6.3.3) means to define the parameters hold by the `SweepMetricCalc` objects. All these parameters are to be passed to the class constructor, whose signature is the following:

Listing 7.1: `SweepMetricCalc` class declaration

```
SweepMetricCalc::SweepMetricCalc( float sweep_angle,
                                  float angle_offset,
                                  float mu_distance,
                                  float sigma_distance,
                                  float mu_angle,
                                  float sigma_angle );
```

With six parameters it is tricky to evaluate how changing a single one affects on all the others. For this reason we decided to fix same values when testing.
To begin with, `sweep_angle` has been considered a fixed parameter during the tests. We set it to 45 degrees, because greater values do not seem (in previous tests) to get any advantages. On the other hand, value less than 45 degrees would not include enough images to teleguide the robot properly.
`angle_offset` is another fixed parameter, set to 40 degrees. Exceeding this value, we risk not to include the robot within the camera field of view, when camera and robot present an orientation offset greater than 40 degrees. Hence, all those images whose orientation exceeds 40 degrees cannot compete to be the background image, and have to be excluded.
The standard deviations `sigma_distance` and `sigma_angle` belong to the set of fixed values too. Changing the standard deviation in a Gaussian function means only to increase or decrease its higher point, without affecting other Gaussian properties. Because later on it will be selected the greatest value among all the returned ones, without any absolute reference, we do not care about the standard deviations value. `sigma_distance` and `sigma_angle` are set with a default positive amount.
In terms of Gaussian function, the `mu_angle` represent the mean value and, at the same time, the point where the Gaussian function is centred. By computing the function with `mu_angle` in input, it will return the possible maximum value. We remember that this function is used to calculate a score, which has to be as greater as the difference between the robot and image orientations are equal. Because the function input is the difference between the two angles, it follows that the returned value must be maximum when the input is zero, decreasing when the input moves away from zero. `mu_angle` is therefore a fixed parameter, set to zero.
At last, the `mu_distance` is the unique variable parameter. All the general consideration made before for the `mu_angle` remain still valid, but this time we want to obtain the maximum score when the difference between robot and image position is equal to a determinant positive value, decreasing when the difference moves away from it.
If we choose a `mu_distance` close to zero, the selected background image will be

near to the robot actual position. The robot will be drawn only partially and the exocentric vision will be similar to the egocentric. The more `mu_distance` moves away from zero (with a positive value), the more the application will tend to draw (when possible) all the robot to provide a full exocentric control vision, because far image will gain an higher score.

7.2 Test preparation

The target of such tests was to verify *(i)* if, using an exocentric vision system provided with such an algorithm, users are able to perceive the trajectory actually performed by the robot, *(ii)* how much disturbing are sudden point-of-view changes for users, *(iii)* how comfortable is for them to view a robot from a virtual exocentric point-of-view.
People involved in the tests have had no experience in robotics, neither they knew what a virtual exocentric vision system is.
The testing session schedule was the following:

1. testers were given a brief introduction to exocentric vision systems
2. each tester is given three different scenarios and has to complete them, unassisted
3. for each scenario, testers have to answer the questions of a questionnaire[1]

Three different recorded logs have been used, each of which featuring a different trajectory. Each of such session has been tested using three different `mu_distance` values: 5, 15 and 25. Results are shown in next section, 7.3.

[1] a copy of the questionnaire is reported in this document, in section A

7.3 Tests results

7.3.1 Rectangle test evaluation

Figure 7.1 shows the first test trajectory: it is a rectangle. This path has been chosen for testing since it features long straight segments and hard turnings (more than 80 degrees).

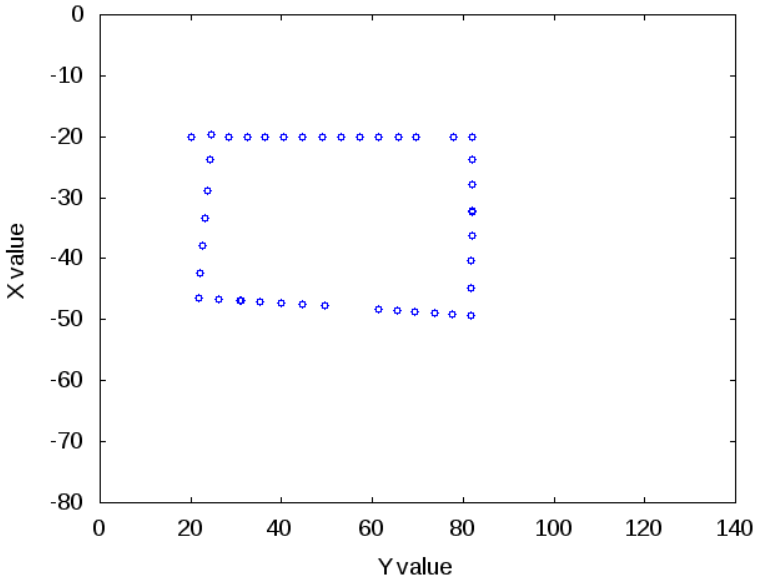


Figure 7.1: Rectangle trajectory test

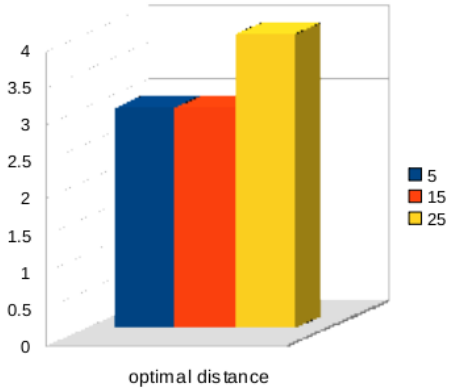


Figure 7.2: Users' disturbance during the rectangle test by sudden change of the point-of-view over a 0/4 scale

The result evidence is that most of the users figured out the trajectory performed by the robot, even tough they perceived sudden point of view changes during hard turnings.
To sum up, they evaluated the experience positively when the optimal distance were set to 5 or 25; not too comfortable when set to 15.

7.3.2 Ellipse test evaluation

Figure 7.3 shows the second test trajectory: it is an ellipse. This path has been chosen for testing since it features long smooth turns.

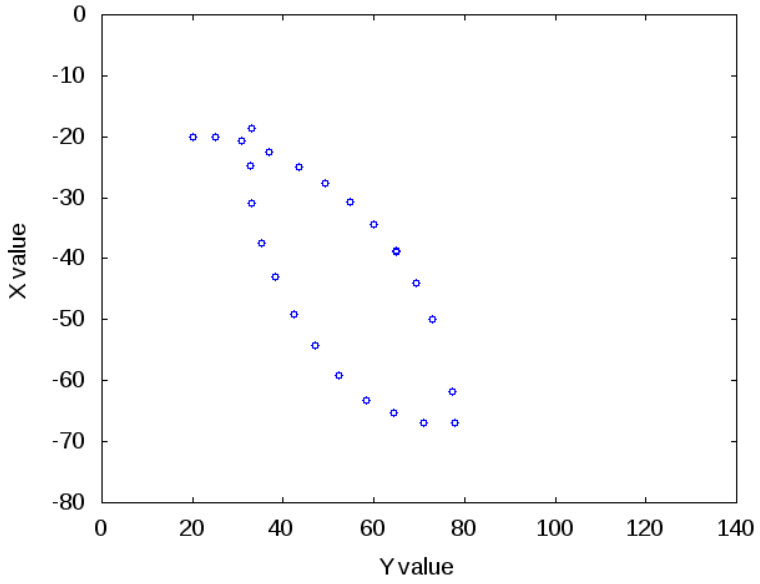


Figure 7.3: Ellipse trajectory test

The result evidence is that all of the users figured out the trajectory performed by the robot. This case allows to point out a specify trend: the more the optimal distance is, the less users perceive sudden point of view changes.
Testers also reported they had very comfortable experience.

7.3.3 Broken lines test evaluation

Figure 7.5 shows the third test trajectory: it is made up of broken lines. This path has been chosen for testing since it features straight lines and sudden hard turnings. The result evidence is the users did not recognise the performed path. Moreover, it also emerged that the more the optimal distance is short, the less are the occurrences of sudden point of view changes.
For the reasons above, testers reported a not very comfortable experience.

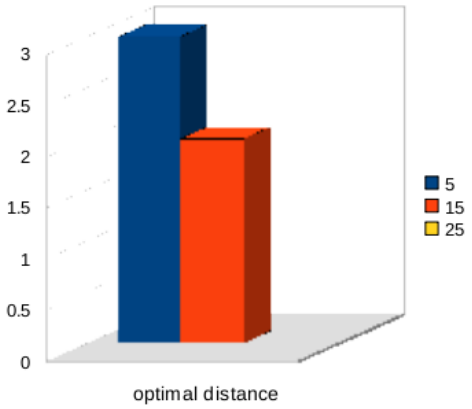


Figure 7.4: Users' disturbance during the ellipse test by sudden change of the point-of-view over a 0/4 scale

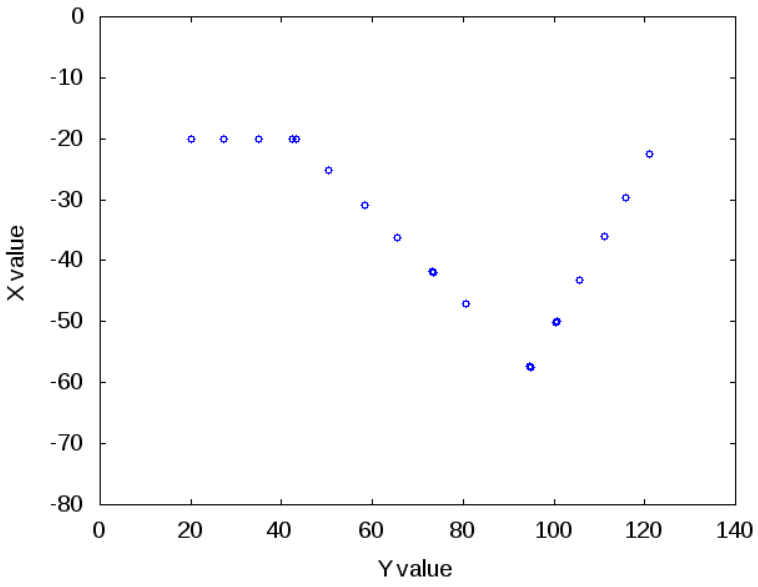


Figure 7.5: Broken lines trajectory test

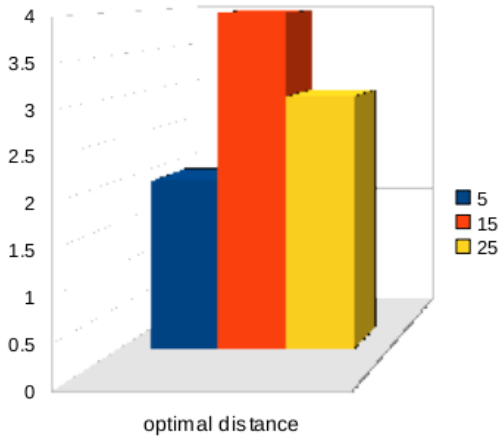


Figure 7.6: Users' disturbance during the broken lines test by sudden change of the point-of-view over a 0/4 scale

7.4 Final considerations

According to us, two chief conclusion can be drawn.
The former is that when the robot moves along straight lines the optimal distance should be set to higher values, in order to show the entire robot model and, hence, to give a *fully* exocentric vision.
The latter one regards turnings. When robot performs hard turnings the *sweep metric algorithm* often selects the egocentric point of view: changing from an exocentric to an egocentric vision causes disturbance to users.
To overcome this deficiency an evolution of the *sweep metric algorithm* was afterwards developed. It is named *another sweep metric algorithm* (proving the authors' lack of imagination), and exposed in sections 6.3.5 and 6.3.6.
Even though the new algorithm has never been tested with users, its benefits are immediately evident, in particular if the robot moves along a 'square' path, where long straight distances are interspersed with strict turns (angles greater than 45 degrees).

Chapter 8

Future works

This section contains some tips for the future development of REAR and the *sweep metric algorithm*.
It would be good to make a comparison between the *sweep angle algorithm* and selection method no. 3 described in [10]. The formula used in [10] to compare the view-point of an image with the robot's current position and direction is not clear at all. Some parameters are ambiguous and there is not an exhaustive explanation about the formula that ties them together. After resolving and implementing the formula, it could be interesting to evaluate the same case tests with the two approaches, in order to underline the advantages and the disadvantages shown by each method and compare them.
New implementations of the `IDataLogic` interface could be developed. One could, for instance, interact by a socket or another data stream) with the simulator making the whole system working *online*, without the need for saved log files and images. Another one could exploit the same approach to connect to the actual Morduc. It would also make possible to actually operate the robot remotely.
In both cases, the class implementing `IDataLogic` must open a communication channel, in order to send commands to the robot and retrieve its data. If the server contacted is the Morduc itself, the concrete `IDataLogic` implementation must know the Morduc communication protocol (see section 2.4 of [24]). Otherwise, if the client communicates with a custom simulator, the communication protocol could be chosen by the developer.
In document [6], Neri and others prove (by several tests) that *3D vision guarantees a major precision in the teleguide and good performances on the obstacles avoidance.* An interesting future development could add the stereoscopic vision to a concrete REAR application, in order to merge the advantages brought by the two different approaches in robot teleguiding.
To implement 3D vision, the robot server must provide both the right and left camera images, whereas the OpenGL functions must draw robot in the proper way on the images retrieved, to render the 3D effect. More details depends on the 3D technologies chosen by the developer: shutter-glasses, anaglyph, polarized, or other.
The work presented in this document dis not take into account collisions. If the environment the robot moves in presents walls or obstacles to avoid, it could be useful to advise user in case of an imminent or already happened collision. A signalling system could be implemented once again with OpenGL, therefore with augmented reality.The laser value, provided by the Morduc, could help to understand when and

where draw warnings, as shown in [7].
A more simple, but doubtless useful, future upgrade would consists in creating a graphical interface, which allows user to define the *sweep metric algorithm* initial parameters in a more friendly way. Currently, all these parameters (e.g. the number of log session or the optimal distance) are given by command line.
Last, but surely not least, two suggestions for the *sweep metric algorithm* reported in section 7.4.
An upgraded version of the algorithm could feature a *variable* optimal distance value. That value could, for instance, be increased when the robot is moving along straight lines and could be decreased when it starts turning.
To avoid sudden point-of-view changings during hard turnings, a possible solution would be to modify the image selection algorithm to have it behaving as follows: when it detects hard turnings, the image to be set as background should provide a greater field-of-view. In other words, instead of selecting the image provided by the egocentric camera, the algorithm should select an image taken in a position more distant than the optimal distance.

Chapter 9

Getting the source code

The REAR project is hosted on *Google Code*:

`http://code.google.com/p/3morduc/`

The latest release of the source code, together with some saved log files, can be obtained by downloading the tar archive named *rear.tar.gz*, from the following URL:

`http://code.google.com/p/3morduc/downloads/list`

Alternatively, to anonymously download a copy of the entire repository:

```
$svn checkout http://3morduc.googlecode.com/svn/trunk/ 3morduc-read-only
```

APPENDIX A

Questionnaire

Questionnaire on

Mobile robots visualization via virtual exocentric vision

The purpose of this questionnaire is to collect your impressions about the experience you just had in visualizing a mobile robot via a virtual exocentric vision system.
Please answer the questions in a candid fashion. All the information you provide is confidential and will remain anonymous. The information you provide will not be used for any other purpose.

1)Did you manage to get the trajectory peformed by the robot? If so, describe it below:

2)Did you feel the robot was moving jerkily?

no 0 1 2 3 4 yes

3)If so, when it was such a behaviour particularly disturbing?

[ ] during hard turnings
[ ] during straight lines
[ ] during broken lines

4)Give an overall evaluation of how comfortable did you feel at understanding what was the robot doing:

not comfortable 0 1 2 3 4 very comfortable

Bibliography

[1] J. Borenstein, H. R. Everett, and L. Feng, *Where am I? Sensors and Methods for Mobile Robot Positioning*, University of Michigan Std., April 2006. [Online]. Available: ftp://ftp.eecs.umich.edu/people/johannb/pos96rep.pdf (Cited on page 4.)

[2] Wikipedia, "Image sensor — wikipedia, the free encyclopedia," 2011, [Online; accessed 16-February-2011]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Image_sensor&oldid=414173394 (Cited on page 11.)

[3] SICK, "LMS200/211/221/291 laser measurement systems." [Online]. Available: http://www.uvs-info.com/pdf/UGV-Datasheets/Sick_Germany_2D-Laser-scanner_LMS_071116.pdf (Cited on page 20.)

[4] VIDERE, "STH-MDCS2-VAR/-C stereo head user's manual." [Online]. Available: http://www.videredesign.com/assets/docs/manuals/sthmdcs2-var.pdf (Cited on page 21.)

[5] DIEES University of Catania, "3rd version of the mobile robot diees university of catania." [Online]. Available: http://www.robotic.diees.unict.it/robots/morduc/morduc.htm (Cited on page 21.)

[6] V. Neri, "Mobile robot tele-guide based on laser sensors," 2008. (Cited on pages 22, 26 and 93.)

[7] D. D. Tommaso and M. Macaluso, "Augmented reality user-interface for robot teleguide based on video and laser data," 2009. (Cited on pages 27 and 94.)

[8] S. Livatino, G. Muscato, S. Sessa, and V. Neri, "Depth-enhanced mobile robot teleguide based on laser images," *Mechatronics*, vol. In Press, Corrected Proof, pp. –, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/B6V43-4YKFMP7-2/2/4cff2a4327b098b7f3d2bda3dc80031b (Cited on page 29.)

[9] "Videre design home page." [Online]. Available: http://www.videredesign.com/ (Cited on page 29.)

[10] M. Sugimoto, G. Kagotani, H. Nii, N. Shiroma, F. Matsuno, and M. Inami, "Time follower's vision: a teleoperation interface with past images," *Computer Graphics and Applications, IEEE*, vol. 25, no. 1, pp. 54 –63, jan.-feb. 2005. (Cited on pages 29, 30, 31, 49, 58, 61, 71 and 93.)

[11] "Rosen - robotic simulation erlang engine." [Online]. Available: http://sourceforge.net/projects/rosen/ (Cited on page 33.)

[12] "The erlang programming language." [Online]. Available: http://www.erlang.org/ (Cited on page 33.)

[13] "Eurobot open competition." [Online]. Available: http://www.eurobot.org/ (Cited on page 33.)

[14] S. Livatino and F. Privitera, "3d visualization technologies for teleguided robots," pp. 240–243, 2006. (Cited on page 33.)

[15] S. Bridgeman, "Opengl tutorial." [Online]. Available: http://math.hws.edu/bridgeman/courses/324/s06/doc/opengl.html (Cited on page 35.)

[16] D. Shreiner, *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1 (7th Edition)*. J. Wiley and Sons, 2009. (Cited on page 35.)

[17] P. Martz, *OpenGL(R) Distilled*. Addison-Wesley Professional, 2006. (Cited on pages 35 and 40.)

[18] D. Shreiner, "The opengl camera analogy." [Online]. Available: http://glprogramming.com/red/chapter03.html (Cited on page 37.)

[19] Wikipedia, "Viewing frustum — wikipedia, the free encyclopedia," 2010. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Viewing_frustum\&oldid=355122422 (Cited on page 45.)

[20] opengl.org, "gluperspective - set up a perspective projection matrix." [Online]. Available: http://www.opengl.org/sdk/docs/man/xhtml/gluPerspective.xml (Cited on page 45.)

[21] "glulookat - define a viewing transformation." [Online]. Available: http://www.opengl.org/sdk/docs/man/xhtml/gluLookAt.xml (Cited on page 45.)

[22] www.codecolony.de, "Using a camera and simple user input." [Online]. Available: http://www.codecolony.de/opengl.htm#camera (Cited on page 53.)

[23] www.blackpawn.com, "Point in triangle test." [Online]. Available: http://www.blackpawn.com/texts/pointinpoly/default.html (Cited on page 76.)

[24] S. D'Asero, "Improving telecommunication performance for robot teleguide: a study of tcp, rtp and rtsp protocols," 2009. (Cited on page 93.)