ViSP 2.6.3: Visual Servoing Platform

Image manipulation and processing

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

September 28, 2012

François Chaumette
Eric Marchand
Nicolas Melchior
Antony Saunier
Fabien Spindler
Romain Tallonneau
Aurélien Yol

Contents

Chapter 1

Image manipulation

1.1 The image structure

1.1.1 Image representation

An image is defined in ViSP as an instance of the container `vpImage<Type>` which contains a regular grid of pixels (see figure 1.1), each pixel value being of type `Type`. The image size is `height x width`.

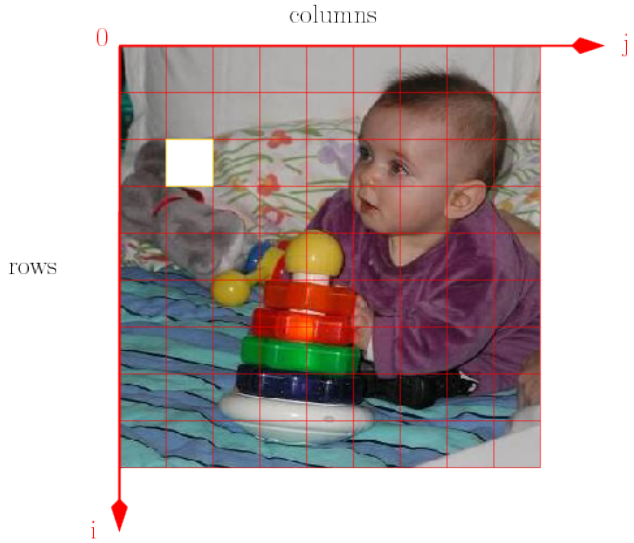


Figure 1.1: ViSP image representation. The pixel in white is at the coordinate (2,1).

Data structure Each image is built using two structures (a `bitmap` array which size is `[width*height]`) and an array of pointers `row` (which size is `[height]`). The ith element in the `row` array `row[i]` is a pointer toward the ith line of the image (ie, `bitmap + i*width`, see figure 1.2). Such a structure allows a fast access to each element of the image (see section 1.1.4). If `i` is the ith row and `j` the jth column the value of this pixel is given by `I[i][j]` (that is equivalent to `row[i][j]`).

In ViSP, images are implemented through a template class. Therefore the type of each element of the array is not *a priori* defined.

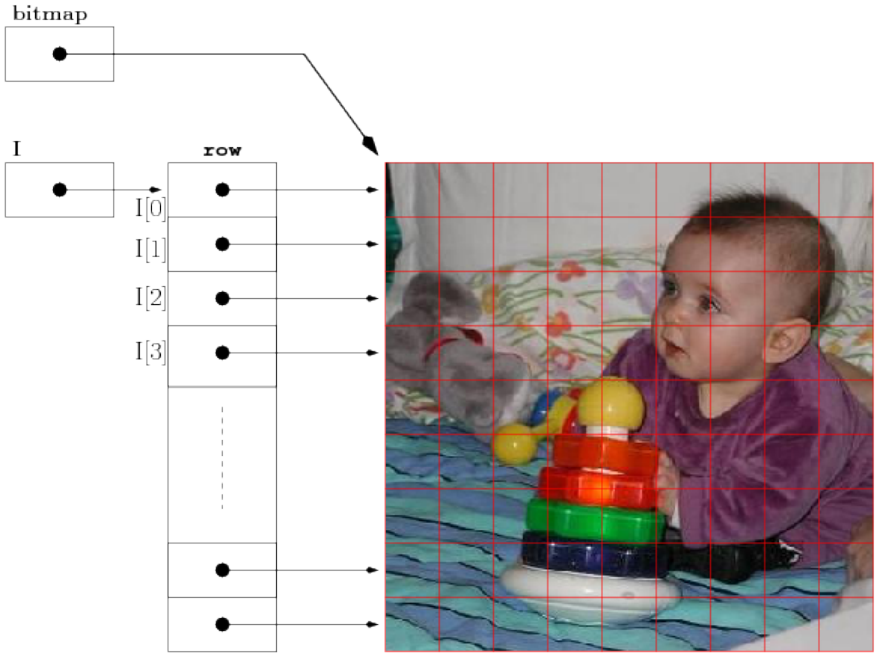


Figure 1.2: ViSP image data structure.

In ViSP the `vpImage` structure is implemented as follow to which is added members function (constructor, destructor, operators ...):

```
template<class Type>
class vpImage
{
public:
  Type *bitmap ;          // points toward the bitmap

private:
  Type **row ;            // points the row pointer array
  unsigned int npixels ;  // number of pixels in the image
  unsigned int width ;    // number of columns
  unsigned int height ;   // number of rows
}
```

In order to use the `vpImage` class, it is necessary to include its header file.

```
#include <visp/vpImage.h>
```

1.1.2 Image types

1.1.2.1 Luminance images

Luminance (or greyscale) images are implemented in ViSP using the pixel data type `unsigned char` which allows to represent 8 bits images (see figure 1.3). In that case each pixel can take an integer value in the $[0, 255]$ interval. Here we create such an empty image.

```
vpImage<unsigned char> I;
```

Figure 1.3: Example of a luminance image: `vpImage<unsigned char>`.

1.1.2.2 RGB images

The term RGB (Red, Green, Blue) stands for a color representation commonly used in digital imaging. A class intended to support the RGB pixel type is available in ViSP: the `vpRGBa` class. You could also define your own pixel class and use it to instantiate a custom image type. The `vpRGBa` class is nothing more than an array of four contiguous `unsigned char` elements respectively for the Red, Green, Blue component. An additional element is provided to align data on 32 bits. In order to use the `vpRGBa` class, it is necessary to include its header file.

```
1 #include <visp/vpRGBa.h>
```

Then a color image can be created (see figure 1.4):

```
1 vpImage<vpRGBa> Irgba;
```

1.1.2.3 YUV images

At this time, YUV images are not implemented in ViSP but it is possible to convert YUV image buffers to implemented image formats (see section 1.1.6).

1.1.3 Allocating and releasing images

First, the header files respectively for the image and the RGB type class must be included.

```
1 #include <visp/vpImage.h>
2 #include <visp/vpRGBa.h>
```

Then we must decide with the data type used to represent the pixels.

```
1 vpImage<unsigned char> Ig; // declares an empty 8 bits greyscale image.
2 vpImage<vpRGBa> Irgba;     // declares an empty 32 bits color image.
```

Figure 1.4: Example of a RGB color image: `vpImage<vpRGBa>`.

Memory allocation. We can then allocate the memory to store the image data. For that, we need to know the image size. After this step, the two arrays `bitmap` and `row` of the `vpImage` class will be created, initialized and linked together as presented figure 1.2.

```
I.init(height, width) ;    // resizes I with the dimensions height x width.
I.resize(height, width) ;  // resizes I with the dimensions height x width.
```

It is also possible to use directly the following constructors :

```
vpImage<unsigned char> Ig(height, width) ;  // declares a height x width greyscale image.
vpImage<vpRGBa> Irgba(height,width) ;       // declares a height x width color image.
```

To create directly an image with all the pixels set at a particular `value`, we use :

```
vpImage<unsigned char> I(height, width, value) ;
//or
vpImage<unsigned char> I;
I.init(height,width,value);
//or
vpImage<unsigned char> I;
I.init(height,width);
I = value ;
```

Memory releasing. Releasing the image is automatically done when the destructor of the `vpImage` instance is called.

1.1.4 Accessing pixel data

As previously stated, the structure of the ViSP image class allows a fast access to each element of the image.

Reading access. To access a given pixel data at the coordinate (`i`,`j`) for reading, you can use one of the following operators:

```
vpImage<unsigned char> I;
unsigned char value;
unsigned int i,j;

value = I[i][j];  // where i,j are the row and column coordinates.
// or
value = I(i,j);   // where i,j are the row and column coordinates.
```

Or directly deal with the `bitmap` image data:

```
vpImage<unsigned char> I;
unsigned char value;
unsigned int i,j;
unsigned int width  = I.getWidth();

value = *(I.bitmap + i*width + j) // where i,j are the row and column coordinates.
```

Writing access. To access this pixel data for writing, you can use equivalent operations:

```
I[i][j] = value;  // where i,j are the row and column coordinates.
// or
I(i,j,value);     // where i,j are the row and column coordinates.
// or dealing with the bitmap pointer
*(I.bitmap + i*width + j) = value;
```

You can also get the value of a pixel at a non integer location with bilinear interpolation:

```
vpImage<unsigned char> I;
unsigned char value = I.getPixelBI(i,j); // where i,j are floats.
                                         // The returned value is rounded to the nearest
                                         // unsigned char (or to the nearest used type)
// or
double dvalue = I.get(i,j); // where i,j are double. The result is express as a double.
                            // This operator is available only for unary types,
                            // like unsigned char (not for vpRGBa for example).
```

Important remark. These operations set or return the pixel value at position (`i`, `j`) corresponding respectively to the row and column position of the considered pixel. To provide high-performance access there is no verification to ensure that $0 \leq$ `i` $<$ `height` and $0 \leq$ `j` $<$ `width`. Since the memory allocated in the bitmap array is continuous, that means that if (`i`, `j`) is outside the image you will manipulate a pixel that is not as expected. To highlight this remark, we provide hereafter an example where the considered pixel is outside the image:

```
unsigned int width = 320;
unsigned int height = 240;
vpImage<unsigned char> I(height, width); // Create an 320x240 image
// Set pixel coordinates that is outside the image
unsigned int i = 100;
unsigned int j = 400;
unsigned char value;
value = I[i][j]; // Here we will get the pixel value at position (101, 80)
```

1.1.5 Image conversion

It can be useful to convert images from a format to another. The `vpImageConvert` class has been implemented to satisfy this need. The first step is to include the header file of the `vpImageConvert` class.

```
#include <visp/vpImageConvert.h>
```

You can then use member functions to convert greyscale image into color images and *vice-versa*:

```
vpImage<unsigned char> Ig; // a greyscale image
vpImage<vpRGBa> Irgba;     // a color image
... // image manipulations
vpImageConvert::convert(Ig,Irgba); // convert a greyscale to a color image.
vpImageConvert::convert(Irgba,Ig); // convert a color to a greyscale image.

vpImage<unsigned char> *pR, *pG, *pB, *pA; // pointers to greyscale images.

vpImageConvert::split(Irgba,pR,pG,pB,pA); // split Irgba channels into 4 greyscale images.
```

1.1.6 Importing an image from a buffer

This section presents how to import data into the `vpImage` class. This is particularly useful for interfacing with other libraries. For the following functionalities, we assume the buffers use contiguous block of memory.

The first step is to include the header file of the `vpImage` class.

```
#include <visp/vpImage.h>
```

Then we create a greyscale image, and resize it with the good dimensions.

```
vpImage<unsigned char> Ig;
Ig.resize(height, width);
```

If the source buffer has the same format than the destination image, the most powerful method is to use the standard C `memcpy` function.

```
unsigned char* src;                   // a buffer image of size height*width
memcpy(Ig.bitmap, src,height*width);  // copy of the memory block in the vpImage instance
```

32 bits RGB color images can be copied in the same manner. In that case the place in memory is `4*height*width`.

```
vpImage<vpRGBa> Irgba;
Irgba.resize(height, width);
unsigned char* src;                        // a buffer image of size 4*height*width
memcpy(Irgba.bitmap, src,4*height*width); // copy of the memory block in the vpImage instance
```

If the source buffer has not the same format than the destination image, we have to convert it in the good format.

In that case, the first step is to include the header file for the `vpImageConvert` class.

```
#include <visp/vpImageConvert.h>
```

Then we replace the previous `memcpy` function by the conversion method corresponding to the need. Here we convert a BGR color coded buffer into the buffer of a `vpImage<vpRGBa>`

```
vpImage<vpRGBa> Irgba;
Irgba.resize(height, width);

unsigned char* bgr;   // a buffer an image of size height x width coded in format BGR

vpImageConvert::BGRToRGBa(bgr,Irgba.bitmap,width,height);
```

Here an other example for converting a YUV422 color coded buffer into the buffer of a `vpImage<unsigned char>`

```
vpImage<unsigned char> Ig;
Ig.resize(height, width);

unsigned char* yuv;   // a buffer an image of size height x width coded in format YUV422

vpImageConvert::YUV422ToGrey(yuv,Ig.bitmap,width*height);
```

Other formats are also available. You can find BGR, RGB, YUV444, YUV411, YUV422, YUV420, YCbCr, YYCrCb conversion. More details can be found on `vpImageConvert` documentation class available on ViSP API documentation (see http://www.irisa.fr/lagadic/visp/publication.html).

1.1.7 Importing an image from OpenCV

It's also possible to convert an image from or into the OpenCV `IplImage` format if you want to use both ViSP and OpenCV libraries. This feature is only available if OpenCV was installed on your computer and detected as a ViSP third party library.

```
#include <visp/vpImageConvert.h>
#include <highgui.h>  // for OpenCV IplImage declaration

vpImage<unsigned char> Ig ; // a greyscale image
vpImage<vpRGBa> Irgba ;     // a color image
IplImage* Icv = NULL ;      // an OpenCV image

vpImageConvert::convert(Ig,Icv) ;    // convert a greyscale image to a one channel IplImage.
vpImageConvert::convert(Irgba,Icv) ; // convert a color image to a 3 channels IplImage.
vpImageConvert::convert(Icv,Ig) ;    // convert a IplImage to a greyscale image.
vpImageConvert::convert(Icv,Irgba) ; // convert a IplImage to a color image.
```

1.1.8 Importing an image from YARP

It's also possible to convert an image from or into the YARP `yarp::sig::ImageOf<>` format if you want to use both ViSP and YARP libraries. This feature is only available if YARP was installed on your computer and detected as a ViSP third party library.

```
#include <visp/vpImageConvert.h>
#include <yarp/sig/Image.h>  // for YARP image declaration

vpImage<unsigned char> Ig ; // a greyscale image
vpImage<vpRGBa> Irgba ;     // a color image
yarp::sig::ImageOf< yarp::sig::PixelMono > Igy;      // a YARP greyscale image
yarp::sig::ImageOf< yarp::sig::PixelRgba > Irgbay;   // a YARP 4 channels color image
yarp::sig::ImageOf< yarp::sig::PixelRgb > Irgby;     // a YARP 3 channels color image

vpImageConvert::convert(Ig,Igy) ;       // ViSP greyscale image to a YARP one channel image
vpImageConvert::convert(Irgba,Irgbay) ; // ViSP color image to a YARP 4 channels image
vpImageConvert::convert(Irgba,Irgby) ;  // ViSP color image to a YARP 3 channels image
vpImageConvert::convert(Igy,Ig) ;       // YARP pixel mono image to a greyscale image
vpImageConvert::convert(Irgbay,Irgba) ; // YARP 4 channels image to a ViSP color image
vpImageConvert::convert(Irgby,Irgba) ;  // YARP 3 channels image to a ViSP color image
```

1.2 Reading and writing images

ViSP allows to read (respectively write) greyscale and color images from (respectively on) the disk.

First, to read (respectively write) images from (respectively on) a file it is required to include the header file of the `vpImageIo` class.

```
#include <visp/vpImageIo.h>
```

Then, the image type should be defined by specifying the type used to represent pixels.

```
vpImage<unsigned char> Ig; // Grey level images
// or
vpImage<vpRGBa> Irgba;     // RGBa color images
```

The image type defines how the data will be represented once it is loaded into memory. This type does not have to correspond exactly to the type stored in the file. The reader will make automatically the conversion into a greyscaled image if you load a color image file in a `vpImage<unsigned char>` instance.

At that time in ViSP supported image file formats are: binary portable anymap formats (PNM) (portable graymap PGM P5 and portable pixmap PPM P6), compressed PNG and JPEG formats.

You can finally call the reader functionality,

```
// We consider first a greylevel image container Ig
vpImageIo::read(Ig, "./myGreyscaleImage.pgm") ; // reads a PGM P5 file from the disk.
vpImageIo::read(Ig, "./myGreyscaleImage.ppm") ; // reads a PGM P6 file from the disk.
vpImageIo::read(Ig, "./myGreyscaleImage.png") ; // reads a PNG file from the disk.
vpImageIo::read(Ig, "./myGreyscaleImage.jpeg") ; // reads a JPEG file from the disk.
// We consider now a color image container Irgba
vpImageIo::read(Irgba, "./myColorImage.pgm") ;  // reads a PPM P5 file from the disk.
vpImageIo::read(Irgba, "./myColorImage.ppm") ;  // reads a PPM P6 file from the disk.
vpImageIo::read(Irgba, "./myColorImage.png") ;  // reads a PNG file from the disk.
vpImageIo::read(Irgba, "./myColorImage.jpeg") ;  // reads a JPEG file from the disk.
```

or the writer functionality.

```
vpImageIo::write(Ig, "./myNewGreyscaleImage.pgm") ; // writes a PGM P5 file on the disk.
vpImageIo::write(Ig, "./myNewGreyscaleImage.ppm") ; // writes a PGM P6 file on the disk.
vpImageIo::write(Ig, "./myNewGreyscaleImage.png") ; // writes a PNG file on the disk.
vpImageIo::write(Ig, "./myNewGreyscaleImage.jpeg") ; // writes a JPEG file on the disk.
vpImageIo::write(Irgba, "./myNewColorImage.pgm") ;  // writes a PPM P5 file on the disk.
vpImageIo::write(Irgba, "./myNewGreyscaleImage.ppm") ; // writes a PGM P6 file on the disk.
vpImageIo::write(Irgba, "./myNewGreyscaleImage.png") ; // writes a PNG file on the disk.
vpImageIo::write(Irgba, "./myNewGreyscaleImage.jpeg") ; // writes a JPEG file on the disk.
```

`read(...)` and `write(...)` members function are general function which call one of the specific reader or writer described in sections 1.2.1, 1.2.2 and 1.2.3 depending on the filename extension.

1.2.1 Reading and writing portable anymap (PNM) images

ViSP allows to read and write non compressed portable anymap image formats (PNM). This feature doesn't require any specific third party library.

Thus `readPGM(...)` reads the contents of a portable gray pixmap (PGM P5), allocate memory for the corresponding grey level or color RGBa image, and set the `vpImage` with the content of the file.

Moreover, `readPPM(...)` reads the contents of a portable pixmap (PPM P6), allocate memory for the corresponding grey level or color RGBa image, and set the `vpImage` with the content of the file.

```
// reads a PGM P5 file from the disk. No conversion is requested here.
vpImageIo::readPGM(Ig, "./myGreyscaleImage.pgm") ;
// reads a PGM P5 file from the disk and convert the grey level image to a RGBa image.
vpImageIo::readPGM(Irgba, "./myGreyscaleImage.pgm") ;
// reads a PPM P6 file from the disk and convert the image to a grey level image.
vpImageIo::readPPM(Ig, "./myColorImage.ppm") ;
```

```
// reads a PPM P6 file from the disk. No conversion is requested here.
vpImageIo::readPPM(Irgba, "./myColorImage.ppm") ;
```

`writePGM(...)` and `writePPM(...)` write respectively the content of a `vpImage` in a PGM P5 file or in a PPM P6 file.

```
// writes a PGM P5 file on the disk.
vpImageIo::writePGM(Ig, "./myNewGreyscaleImage.pgm") ;
// writes a PGM P5 file on the disk. The RGBa color image is converted in a grey level image
// to match the file format
vpImageIo::writePPM(Irgba, "./myNewColorImage.pgm") ;
// writes a PPM P6 file on the disk. Here the grey level image Ig is converted to match
// the file format.
vpImageIo::writePGM(Ig, "./myNewGreyscaleImage.ppm") ;
// writes a PPM P6 file on the disk.
vpImageIo::writePPM(Irgba, "./myNewColorImage.ppm") ;
```

1.2.2 Reading and writing PNG images

ViSP allows to read and write compressed PNG images by using respectively `readPNG(...)` and `writePNG(...)` functions. This feature requires that libpng is installed and detected during ViSP configuration stage (see http://www.irisa.fr/lagadic/visp/libraries.html).

`readPNG(...)` reads the contents of a compressed PNG image, allocate memory for the corresponding grey level or color RGBa image, and set the `vpImage` with the content of the file.

```
// reads a PNG file from the disk and convert the data in a grey level image
vpImageIo::readPNG(Ig, "./myGreyscaleImage.png") ;
// reads a PNG file from the disk and convert the data in a RGBa image.
vpImageIo::readPGM(Irgba, "./myGreyscaleImage.png") ;
```

`writePNG(...)` writes the content of a `vpImage` in a PNG file.

```
// writes a grey level image to a PNG file on the disk
vpImageIo::writePNG(Ig, "./myNewGreyscaleImage.png") ;
// writes a color RGBa image to a PNG file on the disk.
vpImageIo::writePNG(Irgba, "./myNewColorImage.png") ;
```

1.2.3 Reading and writing JPEG images

ViSP allows also to read and write compressed JPEG images by using respectively `readJPEG(...)` and `writeJPEG(...)` functions. This feature requires that libjpeg is installed and detected during ViSP configuration stage (see http://www.irisa.fr/lagadic/visp/libraries.html).

`readJPEG(...)` reads the contents of a compressed JPEG image, allocate memory for the corresponding grey level or color RGBa image, and set the `vpImage` with the content of the file.

```
// reads a JPEG file from the disk and convert the data in a grey level image
vpImageIo::readJPEG(Ig, "./myGreyscaleImage.jpeg") ;
// reads a JPEG file from the disk and convert the data in a RGBa image.
vpImageIo::readPGM(Irgba, "./myGreyscaleImage.jpeg") ;
```

`writeJPEG(...)` writes the content of a `vpImage` in a JPEG file.

```
// writes a grey level image to a JPEG file on the disk
vpImageIo::writeJPEG(Ig, "./myNewGreyscaleImage.jpeg") ;
// writes a color RGBa image to a JPEG file on the disk.
vpImageIo::writeJPEG(Irgba, "./myNewColorImage.jpeg") ;
```

1.3 Graphical user interface

ViSP provides various classes to display images (GUI) depending on your system and installed third party library using either the X11 system or higher level libraries such as GTK, Direct3D, Windows GDI (Graphic Device Interface) or the OpenCV GUI (see http://www.irisa.fr/lagadic/visp/libraries.html). For that, a generic class `vpDisplay` has been proposed from which a particular display class can be derived and some `vpDisplay` pure virtual methods have to be defined within this new class.

At this date in ViSP the implemented display classes are:

- `vpDisplayX` using the X11 system (available on Linux and Mac OSX),
- `vpDisplayGTK` using the GTK multi-platform library,
- `vpDisplayGDI` using the Windows Graphics Device Interface (GDI) (available on Windows),
- `vpDisplayD3D` using the Direct3D (part of DirectX) API under Windows.
- `vpDisplayOpenCV` using the Intel multi-platform OpenCV library.

All these display interfaces are only available if the corresponding third party libraries are installed and detected during the CMake configuration process. To know which capabilities are available on your computer, you can check the header file `include/vpConfig.h` or the more generic `ViSP-third-party.txt` text file available in the built tree. The defined macros allowing the use of the previous classes are respectively `VISP_HAVE_X11`, `VISP_HAVE_GTK`, `VISP_HAVE_GDI`, `VISP_HAVE_D3D9` and `VISP_HAVE_OPENCV`.

Here, we will use the `vpDisplayGTK` interface to illustrate our explanations.

Construction and initialization. The display principle associates one display to one image. In each display, we have an image buffer to make drawings in the memory. We render the image buffer on the screen only when needed (for example at the end of a serie of drawings). The initialization step, links a `vpDisplayGTK` instance to a `vpImage` instance, creates a window and allocates the image buffer.

```
#include <visp/vpImage.h>
#include <visp/vpDisplayGTK.h>

vpImage<unsigned char> I ; // declares a greyscale image
... // image initialization
vpDisplayGTK display ;
display.init(I) ; // initializes the display with default parameters.
// or
display.init(I,winx,winy,"Window title") ; // initializes the display and creates a named
                                           // window at the position (winx,winy) on the screen.
```

Drawing. Drawings are made in the image buffer thanks to static `vpDisplay` member functions. Here are the major ones. See the `vpDisplay` class documentation on ViSP Doxygen documentation to get the complete list of drawing functionalities.

```
vpDisplay::display(I) ; // draws the entire image I in the buffer of the display linked to I.
                        // All stuff drawn before are erased.

vpDisplay::displayPoint(I,...) ;      // draws a point at a given pixel coordinates
vpDisplay::displayCross(I,...) ;      // draws a cross
vpDisplay::displayLine(I,...) ;       // draws a line between two points
```

```
vpDisplay::displayDotLine(I,...) ;    // draws a dashed line between two points
vpDisplay::displayArrow(I,...) ;      // draws an arrow between two points
vpDisplay::displayRectangle(I,...) ;  // draws a rectangle
vpDisplay::displayCircle(I,...) ;     // draws a circle
vpDisplay::displayCharString(I,...) ; // draws a text
...
```

Rendering. After drawings in the buffer image, you have to render the buffer on the screen

```
vpDisplay::flush(I);
```

Without this line, you will see nothing on the screen.

Grabbing. You may have to retrieve what is drawn in the display image buffer (and generally displayed on the screen). The following member function makes a copy of the display image buffer linked to a `vpImage` into a `vpImage<vpRGBa>` instance `Irgba`.

```
vpDisplay::getImage(I,Irgba) ;
```

Mouse events handling. The `vpDisplay` class provides also functionalities to handle mouse events appearing in the opened display window.

```
#include <visp/vpMouseButton.h>
unsigned int i,j; // row and column position in the image.
vpMouseButton::vpMouseButtonType button;

vpDisplay::getClick(I); // waits for a click down in the display window associated to I.
vpDisplay::getClick(I,i,j); // waits for a click down, retrieve the pixel coordinates
vpDisplay::getClick(I,i,j,button); // waits for a click down, retrieve the pixel coordinates
                                   // and the pressed mouse button.

vpDisplay::getClickUp(I); // waits for a click up in the display window associated to I.
vpDisplay::getClickUp(I,i,j); // waits for a click up, retrieve the pixel coordinates
vpDisplay::getClickUp(I,i,j,button); // waits for a click up, retrieve the pixel coordinates
                                     // and the released mouse button.
```

All the previous functions have a blocking behaviour. Your program is stopped in these functions until you click in the displayed window. To change this behavior to non-blocking, you can add a boolean set as `false` after the last argument of these functions. In that case, the used function checks for a mouse event in the event stack of the window and returns `false` if there is not such an event or `true` if there is. The caught event is then removed from the stack.

Destruction. The display window is automatically closed when the destructor of your `vpDisplay` instance is called. To close a display window without destroying your `vpDisplay` instance, you have to call:

```
vpDisplay::close(I) ;
```

After that you will have to reinitialize the display to be able to use it.

1.3.1 Available GUI

All the display interfaces depend on third party libraries. Below you will find the specific ones interfaced to provide a graphical user interface for images. More details are given on http://www.irisa.fr/lagadic/visp/libraries.html.

X11R6 The display interface for X11 window system has been interfaced in `vpDisplayX` class using the X11 library (Xlib): the lowest level of programming interface to X11.

Include the corresponding header file to use it:

```
#include <visp/vpDisplayX.h>
```

GTK GTK can be used for the display under Linux, Windows, ... You need to install GTK if you want to use `vpDisplayGTK`, a class to display ViSP images.

Include the corresponding header file to use it:

```
#include <visp/vpDisplayGTK.h>
```

Windows Graphics Device Interface (GDI) The Windows GDI allows to use `vpDisplayGDI` class under Microsoft Windows Platforms. It is native on these platforms.

Include the corresponding header file to use it:

```
#include <visp/vpDisplayGDI.h>
```

Direct3D Direct3D can be used for the display under Windows. If installed, you are allowed to use `vpDisplayD3D` class.

Include the corresponding header file to use it:

```
#include <visp/vpDisplayD3D.h>
```

OpenCV: Open Source Computer Vision Library ViSP is interfaced with the multi platform OpenCV library. If installed, you are allowed to use `vpDisplayOpenCV` class.

Include the corresponding header file to use it:

```
#include <visp/vpDisplayOpenCV.h>
```

1.3.2 Example

All the displays are used in the same manner. Here is an example using the `vpDisplayGTK` class. This example is also available in ViSP source tree in `example/manual/image-manipulation/manDisplay.cpp`. It shows how to display an image with some drawings in overlay. The resulting display content is given figure 1.5.

```
#include <visp/vpConfig.h>
#include <visp/vpImage.h>
#include <visp/vpImageIo.h>
#include <visp/vpColor.h>
#include <visp/vpDisplayGTK.h>
#include <visp/vpImagePoint.h>

int main()
{
  // Create a grey level image
  vpImage<vpRGBa> I ;

  // Create image points for pixel coordinates
  vpImagePoint ip, ip1, ip2;
```

```
15
16    // Load a grey image from the disk. Klimt.ppm image is part of the ViSP
17    // image data set available from http://www.irisa.fr/lagadic/visp/download.html
18    std::string filename = "./Klimt.ppm";
19    vpImageIo::read(I, filename) ;
20
21  #ifdef VISP_HAVE_GTK
22    // Create a display using GTK
23    vpDisplayGTK display;
24
25    // For this grey level image, open a GTK display at position 100,100
26    // in the screen, and with title "GTK display"
27    display.init(I, 100, 100, "GTK display") ;
28
29    // Display the image
30    vpDisplay::display(I) ;
31
32    // Display in overlay a red cross at position 100,10 in the
33    // image. The lines are 20 pixels long
34    ip.set_i( 200 );
35    ip.set_j( 200 );
36    vpDisplay::displayCross(I, ip, 20, vpColor::red, 3) ;
37
38    // Display in overlay a horizontal red line
39    ip1.set_i( 10 );
40    ip1.set_j( 0 );
41    ip2.set_i( 10 );
42    ip2.set_j( I.getWidth() );
43    vpDisplay::displayLine(I, ip1, ip2, vpColor::red, 3) ;
44
45    // Display in overlay a vertical green dot line
46    ip1.set_i( 0 );
47    ip1.set_j( 20 );
48    ip2.set_i( I.getWidth() );
49    ip2.set_j( 20 );
50    vpDisplay::displayDotLine(I, ip1, ip2, vpColor::green, 3) ;
51
52    // Display in overlay a blue arrow
53    ip1.set_i( 0 );
54    ip1.set_j( 0 );
55    ip2.set_i( 100 );
56    ip2.set_j( 100 );
57    vpDisplay::displayArrow(I, ip1, ip2, vpColor::blue, 8, 4, 3) ;
58
59    // Display in overlay some circles. The position of the center is 200, 200
60    // the radius is increased by 20 pixels for each circle
61    for (unsigned i=0 ; i < 5 ; i++) {
62      ip.set_i( 200 );
63      ip.set_j( 200 );
64      vpDisplay::displayCircle(I, ip, 20*i, vpColor::white, false, 3) ;
65    }
66
67    // Display in overlay a rectangle.
68    // The position of the top left corner is 300, 200.
69    // The width is 200. The height is 100.
70    ip.set_i( 280 );
71    ip.set_j( 150 );
72    vpDisplay::displayRectangle(I, ip, 270, 30,vpColor::purple, false, 3) ;
73
74    // Display in overlay a yellow string
75    ip.set_i( 300 );
76    ip.set_j( 160 );
77    vpDisplay::displayCharString(I, ip,
78                "ViSP is a marvelous software",
79                vpColor::black) ;
```

```
80    //Flush the display : without this line nothing will appear on the screen
81    vpDisplay::flush(I);
82
83    // Create a color image
84    vpImage<vpRGBa> Ioverlay ;
85    // Updates the color image with the original loaded image and the overlay
86    vpDisplay::getImage(I, Ioverlay) ;
87
88    // Write the color image on the disk
89    filename = "./Klimt.overlay.ppm";
90    vpImageIo::write(Ioverlay, filename) ;
91
92    // If click is allowed, wait for a mouse click to close the display
93    std::cout << "\nA click to close the windows..." << std::endl;
94    // Wait for a blocking mouse click
95    vpDisplay::getClick(I) ;
96
97    // Close the display
98    vpDisplay::close(I);
99  #endif
100
101   return 0;
102 }
```

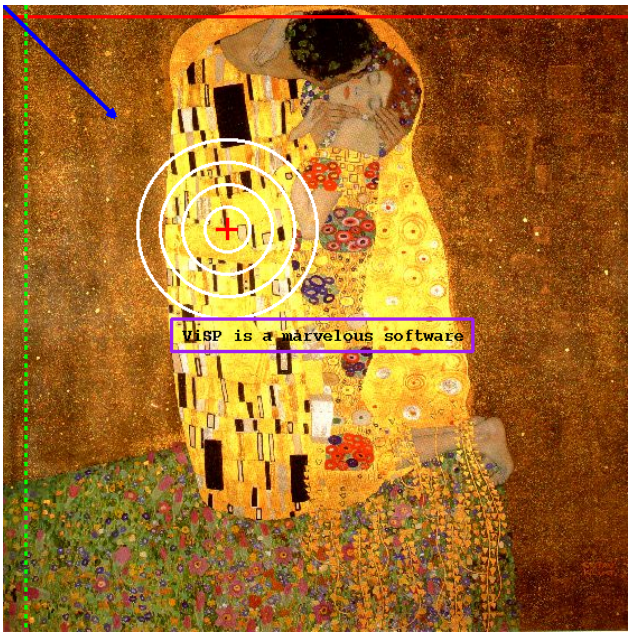


Figure 1.5: Displayed image by the example presented in section 1.3.2

1.4 Image Acquisition

Frame grabber interfaces allow to deal with video devices such as cameras.

1.4.1 Generic frame grabber interface

A generic `vpFrameGrabber` class has been proposed from which a particular framegrabber class can be derived and some `vpFrameGrabber` pure virtual methods have to be defined within this new class (mainly initialization, acquisition, closing methods). Such interface with ViSP is very simple to add since such methods should already exist on the user particular system. In the current version of ViSP, some classical framegrabbers are already considered (IEEE 1394, Video4Linux2, DirectShow,...).

The prototype of the `vpFrameGrabber` can be defined as follow (to which must be added constructors and destructors, copy operators, etc.):

```
class vpFrameGrabber {
  public :
    bool   init ;  // bit 1 if the frame grabber has been initialized

  protected:
    unsigned int height ;  // number of rows in the image
    unsigned int width ;  // number of columns in the image

  public:
    // return the number of rows in the image
    inline  unsigned int getHeight() { return height ; }
    //! return the number of columns in the image
    inline  unsigned int getWidth() { return width ; }

    // initialize the framegrabber
    virtual void open(vpImage<unsigned char> &I) =0 ;
    virtual void open(vpImage<vpRGBa> &I) =0 ;

    // acquire a frame into a vpImage
    virtual void acquire(vpImage<unsigned char> &I) =0 ;
    virtual void acquire(vpImage<vpRGBa> &I) =0 ;

    // close the framegrabber
    virtual void close() =0 ;
} ;
```

1.4.2 Specific interface to video device

At this date in ViSP the implemented interface classes are:

- `vp1394TwoGrabber` using the libdc1394-2.x third party library (available on Linux, Mac OSX and Windows but never tested under Windows),
- `vpV4l2Grabber` using the third party Video4Linux2 library (available on Linux),
- `vpDirectShowGrabber` using the third party Microsoft DirectShow library (available on Windows),
- `vpOpenCVGrabber` using the third party OpenCV library (available on Windows, Linux and Mac OSX),
- `vpDiskGrabber` that does't require a specific third party library.

Depending on third party libraries, to know which capabilities are available on your computer, you can check the built `ViSP-third-party.txt` file that resume all the supported third party libraries that are interfaced with your built. The defined macros allowing the use of the previous classes are respectively `VISP_HAVE_1394_2`, `VISP_HAVE_V4L2`, `VISP_HAVE_DIRECTSHOW` and `VISP_HAVE_OPENCV`. `vpDiskGrabber` is not third party dependant.

1.4.2.1 Interface to firewire camera: libdc1394-2

ViSP uses the libdc1394 library to implement an IEEE 1394 interface. libdc1394-2 is a library that is intended to provide a high level programming interface for application developers who wish to control IEEE 1394 based cameras that conform to the 1394-based Digital Camera Specification. If libdc1394-2.x is installed you can grab images from firewire cameras with `vp1394TwoGrabber` class. This grabber allows single or multi camera acquisition.

This class was tested with Marlin F033C, Marlin F131B and Point Grey Dragonfly 2 cameras.

Here a minimal example of capture from the first camera found on the bus with the current camera settings:

```
#include <visp/vpImage.h>
#include <visp/vp1394TwoGrabber.h>

int main(){
  vpImage<unsigned char> I;
  vp1394TwoGrabber g;
  while(1)
    g.acquire(I);
}
```

However this grabber allows to modify the camera settings.

Declaration First we should include the `vp1394TwoGrabber` header file, declare a framegrabber and the image in which we will put the acquired data:

```
#include <visp/vpImage.h>
#include <visp/vp1394TwoGrabber.h>

vp1394TwoGrabber g;
vpImage<unsigned char> I;
```

Here we declare a monochrome image but it is also possible to use color image container. Then we can set the camera settings.

Camera selection To communicate with a camera, we have to set this one as the active camera. The camera selection is done using the `setCamera(...)` function:

```
g.setCamera(camera);
```

where `camera` is the index of the camera you want to deal with. Its value must be comprised between 0 (the first camera) and the number of cameras found on the bus. If two cameras are connected on the bus, setting `camera` to `1` allows to communicate with the second one.

To know the number of cameras on the bus, use the member function `getNumCameras(...)`

```
unsigned int num_cameras;
g.getNumCameras(num_cameras);
```

Camera settings manipulation Two steps are necessary to set specific camera settings. First you have to set the camera video mode, and then the framerate.

Video mode setting The camera video mode gives the size of the acquired images and their color coding. It can be set by `setVideoMode(...)`. The current camera video mode is given by `getVideoMode(...)`. The allowed values are given in `vp1394TwoGrabber` header file:

```
/*!
  Enumeration of video modes. See libdc1394 2.x header file dc1394/control.h
*/
typedef enum {
  vpVIDEO_MODE_160x120_YUV444   = DC1394_VIDEO_MODE_160x120_YUV444,
  vpVIDEO_MODE_320x240_YUV422   = DC1394_VIDEO_MODE_320x240_YUV422,
  vpVIDEO_MODE_640x480_YUV411   = DC1394_VIDEO_MODE_640x480_YUV411,
  vpVIDEO_MODE_640x480_YUV422   = DC1394_VIDEO_MODE_640x480_YUV422,
  vpVIDEO_MODE_640x480_RGB8     = DC1394_VIDEO_MODE_640x480_RGB8,
  vpVIDEO_MODE_640x480_MONO8    = DC1394_VIDEO_MODE_640x480_MONO8,
  vpVIDEO_MODE_640x480_MONO16   = DC1394_VIDEO_MODE_640x480_MONO16,
  vpVIDEO_MODE_800x600_YUV422   = DC1394_VIDEO_MODE_800x600_YUV422,
  vpVIDEO_MODE_800x600_RGB8     = DC1394_VIDEO_MODE_800x600_RGB8,
  vpVIDEO_MODE_800x600_MONO8    = DC1394_VIDEO_MODE_800x600_MONO8,
  vpVIDEO_MODE_1024x768_YUV422  = DC1394_VIDEO_MODE_1024x768_YUV422,
  vpVIDEO_MODE_1024x768_RGB8    = DC1394_VIDEO_MODE_1024x768_RGB8,
  vpVIDEO_MODE_1024x768_MONO8   = DC1394_VIDEO_MODE_1024x768_MONO8,
  vpVIDEO_MODE_800x600_MONO16   = DC1394_VIDEO_MODE_800x600_MONO16,
  vpVIDEO_MODE_1024x768_MONO16  = DC1394_VIDEO_MODE_1024x768_MONO16,
  vpVIDEO_MODE_1280x960_YUV422  = DC1394_VIDEO_MODE_1280x960_YUV422,
  vpVIDEO_MODE_1280x960_RGB8    = DC1394_VIDEO_MODE_1280x960_RGB8,
  vpVIDEO_MODE_1280x960_MONO8   = DC1394_VIDEO_MODE_1280x960_MONO8,
  vpVIDEO_MODE_1600x1200_YUV422 = DC1394_VIDEO_MODE_1600x1200_YUV422,
  vpVIDEO_MODE_1600x1200_RGB8   = DC1394_VIDEO_MODE_1600x1200_RGB8,
  vpVIDEO_MODE_1600x1200_MONO8  = DC1394_VIDEO_MODE_1600x1200_MONO8,
  vpVIDEO_MODE_1280x960_MONO16  = DC1394_VIDEO_MODE_1280x960_MONO16,
  vpVIDEO_MODE_1600x1200_MONO16 = DC1394_VIDEO_MODE_1600x1200_MONO16,
  vpVIDEO_MODE_EXIF      = DC1394_VIDEO_MODE_EXIF,
  vpVIDEO_MODE_FORMAT7_0 = DC1394_VIDEO_MODE_FORMAT7_0,
  vpVIDEO_MODE_FORMAT7_1 = DC1394_VIDEO_MODE_FORMAT7_1,
  vpVIDEO_MODE_FORMAT7_2 = DC1394_VIDEO_MODE_FORMAT7_2,
  vpVIDEO_MODE_FORMAT7_3 = DC1394_VIDEO_MODE_FORMAT7_3,
  vpVIDEO_MODE_FORMAT7_4 = DC1394_VIDEO_MODE_FORMAT7_4,
  vpVIDEO_MODE_FORMAT7_5 = DC1394_VIDEO_MODE_FORMAT7_5,
  vpVIDEO_MODE_FORMAT7_6 = DC1394_VIDEO_MODE_FORMAT7_6,
  vpVIDEO_MODE_FORMAT7_7 = DC1394_VIDEO_MODE_FORMAT7_7
} vp1394TwoVideoModeType;
```

All these video modes are not supported by your camera. The list of your camera supported video modes is given by `getVideoModeSupported(...)`.

The video mode is expressed as an `int`. To be more explicit, `videoMode2string(...)` converts the video mode identifier into a string containing the description of the video mode.

```
int videoMode;
g.getVideoMode(videoMode); // gets the current video mode identifier
std::cout << "The current videoMode is : " << g.videoMode2string(videoMode) << std::endl;
```

In the case of `FORMAT7` video modes, the color coding type must be placed separately. It can be set by `setColorCoding(...)`. The current camera color coding type is given by `getColorCoding(...)`. The allowed values are given in `vp1394TwoGrabber` header file:

```
/*!
  Enumeration of color codings. See libdc1394 2.x header file dc1394/control.h
*/
typedef enum {
  vpCOLOR_CODING_MONO8   = DC1394_COLOR_CODING_MONO8,
```

```
  vpCOLOR_CODING_YUV411  = DC1394_COLOR_CODING_YUV411,
  vpCOLOR_CODING_YUV422  = DC1394_COLOR_CODING_YUV422,
  vpCOLOR_CODING_YUV444  = DC1394_COLOR_CODING_YUV444,
  vpCOLOR_CODING_RGB8    = DC1394_COLOR_CODING_RGB8,
  vpCOLOR_CODING_MONO16  = DC1394_COLOR_CODING_MONO16,
  vpCOLOR_CODING_RGB16   = DC1394_COLOR_CODING_RGB16,
  vpCOLOR_CODING_MONO16S = DC1394_COLOR_CODING_MONO16S,
  vpCOLOR_CODING_RGB16S  = DC1394_COLOR_CODING_RGB16S,
  vpCOLOR_CODING_RAW8    = DC1394_COLOR_CODING_RAW8,
  vpCOLOR_CODING_RAW16   = DC1394_COLOR_CODING_RAW16
} vp1394TwoColorCodingType;
```

All these color coding type are not supported by your camera. The list of your camera supported color coding types is given by `getColorCodingSupported(...)`.

The color coding type is expressed as an `int`. To be more explicit, `colorCoding2string(...)` converts the color coding type identifier into a string containing the description of the color coding type.

```
int colorCoding;
g.getColorCoding(colorCoding); // gets the current color coding identifier
std::cout << "Current color coding : " << g.colorCoding2string(colorCoding) << std::endl;
```

Setting color coding for non `FORMAT7` video modes will be without effect.

Framerate setting The camera framerate can be set by `setFramerate(...)`. The current camera framerate is given by `getFramerate(...)`. The allowed values are given in `vp1394TwoGrabber` header file:

```
/*!
  Enumeration of framerates. See libdc1394 2.x header file dc1394/control.h
*/
typedef enum {
  vpFRAMERATE_1_875 = DC1394_FRAMERATE_1_875,
  vpFRAMERATE_3_75  = DC1394_FRAMERATE_3_75,
  vpFRAMERATE_7_5   = DC1394_FRAMERATE_7_5,
  vpFRAMERATE_15    = DC1394_FRAMERATE_15,
  vpFRAMERATE_30    = DC1394_FRAMERATE_30,
  vpFRAMERATE_60    = DC1394_FRAMERATE_60,
  vpFRAMERATE_120   = DC1394_FRAMERATE_120,
  vpFRAMERATE_240   = DC1394_FRAMERATE_240
} vp1394TwoFramerateType;
```

All these framerates are not supported by your camera. The list of your camera supported framerates is given by `getFramerateSupported(...)`.

The framerate is expressed as an `int`. To be more explicit, `framerate2string(...)` converts the framerate identifier into a string containing the description of the framerate.

```
int framerate;
g.getFramerate(framerate); // gets the current framerate identifier
std::cout << "The current framerate is : " << g.framerate2string(framerate) << std::endl;
```

Ring buffer size setting The ring buffer is organized as a contiguous block of memory-mapped frame buffers waiting to be filled and internally set to a queued state. Filling of the first buffer can start as soon as you create a `vp1394TwoGrabber` instance. Each buffer is set to the ready state as soon as it is filled. Frame transmission continues until you close the grabber by calling `close()` method. If all of the buffers are filled during the capture then the capture stops (that means that you loose recent frames) until you close the grabber or make space by calling `acquire()` capture functions. In ViSP the default ring buffer size is set to 4. Depending on your computer, it can be useful to change this value:

```
g.setRingBufferSize(2);
```

It is also possible to know the current ring buffer size:

```
unsigned int ringBufferSize;
ringBufferSize = g.getRingBufferSize();
```

Acquisition The acquisition is done using the `acquire(...)` function .

```
while(1)
  g.acquire(I);
```

If an image is available in the framebuffer, it returns this image in `I`. In the other case, it waits for a new one.

Closing The framegrabber closing useful to stop the image capture is done either by the `vp1394TwoGrabber` destructor, or by an explicit call to the `close()` function.

Here an example of multi camera capture. This example is also available in the ViSP source tree in `example/manual/image-manipulation/manGrab1394-2.cpp`. A more complete example can also be found in `example/framegrabber/grab1394Two.cpp` in the ViSP source tree.

```
#include <visp/vpImage.h>
#include <visp/vp1394TwoGrabber.h>

int main(){
  unsigned int ncameras; // Number of cameras on the bus
  vp1394TwoGrabber g;
  g.getNumCameras(ncameras);
  vpImage<unsigned char> *I = new vpImage<unsigned char> [ncameras];

  // If the first camera supports vpVIDEO_MODE_640x480_YUV422 video mode
  g.setCamera(0);
  g.setVideoMode(vp1394TwoGrabber::vpVIDEO_MODE_640x480_YUV422);

  // If all cameras support 30 fps acquisition
  for (unsigned int camera=0; camera < ncameras; camera ++) {
    g.setCamera(camera);
    g.setFramerate(vp1394Two::vpFRAMERATE_30);
  }

  while(1) {
    for (unsigned int camera=0; camera < ncameras; camera ++) {
      // Acquire successively images from the different cameras
      g.setCamera(camera);
      g.acquire(I[camera]);
    }
  }
  delete [] I;
}
```

1.4.2.2 Interface to firewire camera: apple quicktime

At this date, the apple quicktime interface is not implemented.

1.4.2.3 V4l2 interface : Video For Linux 2

Under Linux if V4l2 is installed, with `vpV4l2Grabber` class you can grab images from USB cameras or also from analogic cameras connected to a PCI TV board. This grabber allows single camera acquisition.

This class was tested with a Pinnacle PCTV Studio/Rave board but also with the following webcams (Logitech QuickCam Vision Pro 9000, Logitech QuickCam Orbit AF, Dell latitude E6400 internal webcam).

Here a minimal example of capture from the first video input port with the default camera settings:

```
#include <visp/vpImage.h>
#include <visp/vpV4l2Grabber.h>

int main(){
  vpImage<unsigned char> I;
  vpV4l2Grabber g;
  g.open(I);
  while(1)
    g.acquire(I);
}
```

However this grabber allows to modify the camera settings.

Declaration First we should include the `vpV4l2Grabber` header file, declare a framegrabber and the image in which we will put the acquired data:

```
#include <visp/vpImage.h>
#include <visp/vpV4l2Grabber.h>

vpV4l2Grabber g;
vpImage<unsigned char> I;
```

Here we declare a monochrome image but it is also possible to use color image container. Then we can set the camera settings.

Camera settings manipulation

Input board and camera selection If several acquisition boards are installed or if the board is not mounted at `/dev/video0`, we have to set the device name thanks to the `setDevice(...)` function.

```
g.setDevice(device);
```

where `device` is the name of the mounted point of the board we want to deal with.

To communicate with a camera on that board, we have to set this one as the active camera. The camera selection is done using the `setInput(...)` function:

```
g.setInput(camera);
```

where `camera` is the index of the video input port on the acquisition board we want to deal with. Its value must be comprised between `0` (the first port, the default value) and the number of ports on the board. If two ports are available on the board, setting `camera` to `1` allows to communicate with the second one.

Image resolution setting The camera image resolution can be set by `setScale(...)`.

This function sets the decimation factor applied to full resolution images. The scale should be between 1 and 16. Setting the scale factor to `2` will produce half size images with reference to the full resolution images.

```
int scale;
g.setScale(scale); // set the decimation factor
std::cout << "The current framerate is : " << g.framerate2string(framerate) << std::endl;
```

`setWidth(...)` and `setHeight(...)` can also be used to set this factor.

Framerate setting The camera framerate can be set by `setFramerate(...)`. The current camera framerate is given by `getFramerate(...)`. The allowed values are given in `vpV4l2Grabber` header file:

```
/*!
  Frame rate type for capture.
*/
typedef enum
  {
    framerate_50fps, //!< 50 frames per second
    framerate_25fps  //!< 25 frames per second
  } vpV4l2FramerateType;
```

Ring buffer setting `setNBuffers(...)` Set the number of buffers required for streaming data.

For non real-time applications the number of buffers should be set to `1`. For real-time applications to reach 25 fps or 50 fps a good compromise is to set the number of buffers to `3`.

Initialization When all the settings have been placed, the camera can be initialized. The image in which the captured data will be placed must also be resized. This operation is achieved thanks to the `open(...)` function:

```
g.open(I);
```

Now, the framegrabber is ready to acquire frames.

Acquisition The acquisition is done using the `acquire(...)` function .

```
while(1)
  g.acquire(I);
```

If an image is available in the framebuffer, it returns this image in `I`. In the other case, it waits for a new one.

Closing The framegrabber closing useful to stop the image capture is done either by the `vpV4l2Grabber` destructor, or by an explicit call to the `close()` function.

Here an example of single camera capture. This example is also available in `example/manual/image-manipulation/manGrabV4l2.cpp` in the ViSP source tree. A more complete example can also be found in the ViSP source tree in `example/framegrabber/manGrabV4l2.cpp`.

```
#include <visp/vpImage.h>
#include <visp/vpV4l2Grabber.h>
int main(){
  vpImage<unsigned char> I; // Grey level image

  vpV4l2Grabber g;
  g.setInput(2);     // Input 2 on the board
  g.setWidth(768);   // Acquired images are 768 width
  g.setHeight(576); // Acquired images are 576 height
  g.setNBuffers(3); // 3 ring buffers to ensure real-time acquisition
  g.open(I);         // Open the grabber
  while(1)
    g.acquire(I);     // Acquire a 768x576 grey image
}
```

1.4.2.4 Windows interface : DirectShow

ViSP uses the DirectShow library to implement a Windows interface. If DirectShow is installed and detected you can grab images from cameras which support this interface with `vpDirectShowGrabber` class. This grabber allows only single camera acquisition. However it is possible to deal with several cameras using one grabber per camera.

Here a minimal example of capture from the first camera found on the bus with the current camera settings:

```
#include <visp/vpImage.h>
#include <visp/vpDirectShowGrabber.h>

int main(){
  vpImage<unsigned char> I;
  vpDirectShowGrabber g;
  while(1)
    g.acquire(I);
}
```

However this grabber allows to modify the camera settings.

Declaration and Initialisation First we should include the `vpDirectShowGrabber` header file, declare a framegrabber and the image in which we will put the acquired data:

```
#include <visp/vpImage.h>
#include <visp/vpDirectShowGrabber.h>

vpDirectShowGrabber g;
vpImage<unsigned char> I;
```

Here we declare a monochrome image but it is also possible to use color image container.

Initialisation We have then to initialize the grabber.

```
g.open();
```

Then we can set the camera settings.

Camera selection To communicate with a camera, we have to set this one as the active camera. The camera selection is done using the `setDevice(...)` function:

```
g.setDevice(camera);
```

where `camera` is the index of the camera you want to deal with. Its value must be comprised between 0 (the first camera) and the number of cameras found on the bus. If two cameras are connected on the bus, setting `camera` to `1` allows to communicate with the second one.

To know the number of cameras on the bus, use the member function `getDeviceNumber()`

```
unsigned int num_cameras;
g.getDeviceNumber(num_cameras);
```

To have more information about cameras on the bus, use the member function `displayDevices()`. It displays the list of devices on the standard output (camera index, name, ...).

Camera settings manipulation Two steps are necessary to set specific camera settings. First you have to set the camera video mode, and then the framerate.

Video mode setting The camera video mode gives the size of the acquired images and their color coding. It can be set by `setMediaType(...)`. The current camera video mode is given by `getMediaType()`.

The list of your camera supported video modes is displayed on the standard output by `getStreamCapabilities()`.

The member function `setImageSize(...)` can also be used to change the size of the acquired images if a corresponding video mode is available with the current color coding.

Framerate setting The camera framerate can be set by `setFramerate(...)`.

According the DirectShow documentation, the effective framerate applied is the nearest framerate supported by your camera.

The current camera framerate is given by `getFramerate(...)`.

Acquisition The acquisition is done using the `acquire(...)` function .

```
while(1)
  g.acquire(I);
```

If an image is available in the framebuffer, it returns this image in `I`. In the other case, it waits for a new one.

Here an example of single camera capture. This example is also available in the ViSP source tree in `example/manual/image-manipulation/manGrabDirectShow.cpp`. More complete examples can also be found in `example/framegrabber/grabDirectShow.cpp` and `example/framegrabber/grabDirectShowMulti.cpp` in the ViSP source tree.

```
#include <visp/vpImage.h>
#include <visp/vpDirectShowGrabber.h>

int main(){
  vpImage<unsigned char> I; // Grey level image

  vpDirectShowGrabber g; // Create the grabber
  if(g.getDeviceNumber() == 0) //test if a camera is connected
```

```
  {
    g.close();
    exit(0);
  }

  g.open(); // Initialize the grabber

  g.setImageSize(640,480); // If the camera supports 640x480 image size
  g.setFramerate(30); // If the camera supports 30fps framerate

  while(1)
    g.acquire(I); // Acquire an image
}
```

1.4.2.5 Disk interface

The disk interface has been implemented as a virtual video device to "grab" images from the disk thanks to the `vpDiskGrabber` class. This class is an interface to the `vpImageIo` class. See section 1.2 to have more information about the supported image formats.

Here an example of capture from the directory `/tmp`. We want to acquire 100 images from the first named `image0001.pgm` by steps of 3.

```
#include <visp/vpImage.h>
#include <visp/vpDiskGrabber.h>

int main(){
  vpImage<unsigned char> I; // Grey level image

  // Declare a framegrabber able to read a sequence of successive
  // images from the disk
  vpDiskGrabber g;

  // Set the path to the directory containing the sequence
  g.setDirectory("/tmp");
  // Set the image base name. The directory and the base name constitute
  // the constant part of the full filename
  g.setBaseName("image");
  // Set the step between two images of the sequence
  g.setStep(3);
  // Set the number of digits to build the image number
  g.setNumberOfZero(4);
  // Set the first frame number of the sequence
  g.setImageNumber(1);
  // Set the file extension of the images of the sequence
  g.setExtension("pgm");

  // Open the framegrabber by loading the first image of the sequence
  g.open(I) ;

  // this is the loop over the image sequence
  for(int cpt = 0; cpt < 100; cpt++)
  {
    // read the image and then increment the image counter so that the next
    // call to acquire(I) will get the next image
    g.acquire(I) ;
  }
}
```

1.4.2.6 Multi-platform interface : OpenCV

ViSP can also use the OpenCV third party library to implement a multiplatform interface. If OpenCV is installed and detected you can grab images from cameras which support this interface with `vpOpenCVGrabber` class. This grabber allows only single camera acquisition. However it is possible to deal with several cameras using one grabber by camera.

Here a minimal example of capture from the first camera found on the bus with the current camera settings:

```
#include <visp/vpImage.h>
#include <visp/vpOpenCVGrabber.h>

int main(){
  vpImage<unsigned char> I;
  vpOpenCVGrabber g;
  while(1)
    g.acquire(I);
}
```

However this grabber allows to modify few camera settings.

Declaration and Initialisation First we should include the `vpOpenCVGrabber` header file, declare a framegrabber and the image in which we will put the acquired data:

```
#include <visp/vpImage.h>
#include <visp/vpOpenCVGrabber.h>

vpOpenCVGrabber g;
vpImage<unsigned char> I;
```

Here we declare a monochrome image but it is also possible to use a color image container.

Device type selection We have then to choose the type of device we want to use.

```
g.setDeviceType(deviceType);
```

The variable `deviceType` is an `int` that can take different values which are :

- CV_CAP_ANY : Look for any kind of device type. Stop the research as soon as a camera is found.
- CV_CAP_MIL : Usable if the MIL third party library is installed on your computer and detected by OpenCV.
- CV_CAP_VFW : Usable if the framework Video for Windows is detected by OpenCV
- CV_CAP_V4L : Usable if Video for Linux is available.
- CV_CAP_V4L2 : Usable if Video for Linux Two is available.
- CV_CAP_FIREWIRE : Usable if a third party library dedicated to firewire devices is detected by OpenCV.
- CV_CAP_IEEE1394 : Usable if a third party library dedicated to firewire devices is detected by OpenCV.

- CV_CAP_DC1394 : Usable if a third party library dedicated to firewire devices is detected by OpenCV.
- CV_CAP_CMU1394 : Usable if a third party library dedicated to firewire devices is detected by OpenCV.

If we have more than one device with the expected type it is possible to choose the one to use. Notice that each camera is linked to a number which represents its index. The first one is indexed by 0, the second by 1 and so on. For example, if we have two firewire cameras, to select the first one or the second one we can use :

```
g.setDeviceType(CV_CAP_IEEE1394+0); //The first camera

g.setDeviceType(CV_CAP_IEEE1394+1); //The second camera
```

Initialisation We have then to initialize the grabber.

```
g.open();
```

Camera settings manipulation Two steps are necessary to set specific camera settings. First you have to set the camera video size, and then the framerate.

Video size setting The camera video size can be set by `setWidth(...)` and `setHeight(...)`. The current camera video size is given by `getWidth()` and `getHeight()`.

Framerate setting The camera framerate can be set by `setFramerate(...)`. According to the OpenCV documentation, the frame rate can be modified only if the camera allows it. The current camera framerate is given by `getFramerate(...)`.

Acquisition The acquisition is done using the `acquire(...)` function .

```
while(1)
  g.acquire(I);
```

If an image is available in the framebuffer, it returns this image in `I`. In the other case, it waits for a new one.

Closing The framegrabber closing useful to stop the image capture is done either by the `vpOpenCVGrabber` destructor, or by an explicit call to the `close()` function.

Problem under Windows A problem can appear under Windows depending on the camera you use. Indeed the image can be flipped vertically. If you see such a problem, you can fix it by using the function `setFlip(...)`.

```
g.setFlip(true);
```

Here an example of single camera capture. This example is also available in the ViSP source tree in `example/manual/image-manipulation/manGrabOpenCV.cpp`. More complete examples can also be found in `example/framegrabber/grabOpenCV.cpp` in the ViSP source tree.

```
#include <visp/vpImage.h>
#include <visp/vpOpenCVGrabber.h>

int main(){
  vpImage<unsigned char> I; // Grey level image

  g.open();             // Initialize the grabber

  g.setWidth(640);
  g.setHeight(480);     // If the camera supports 640x480 image size
  g.setFramerate(30);   // If the camera supports 30fps framerate

  while(1)
    g.acquire(I);       // Acquire an image
}
```

Chapter 2

Tracking in image sequences

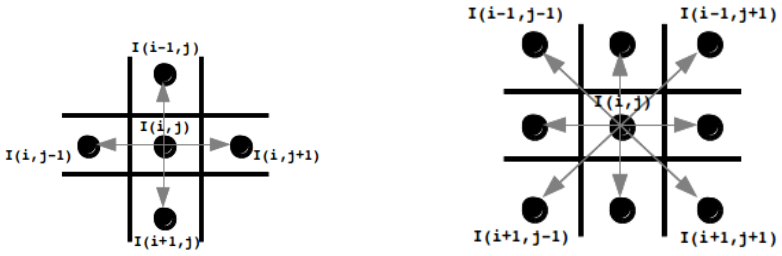
2.1 Tracking a blob

In ViSP blob trackers are implemented in `vpDot` and `vpDot2` classes. Both classes track set of pixels with specific value. To belong to a blob or dot, a pixel value has to be between a minimum and a maximum value: $\lambda_{min} < I(i,j) < \lambda_{max}$ with I the current image where you track the dot and i, j the pixel coordinates. In both classes, these values can be defined with the functions: `setGrayLevelMin(int)`, `setGrayLevelMax(int)` and `setGrayLevelPrecision(double)`.
`vpDot` and `vpDot2` classes track dots with two different approaches, the two next subsection describe their principle.

2.1.1 vpDot

Principle

`vpDot` starts from one point and looks to its connexities in order to know if they are part of the dot or not. A connexity is part of the dot if its value is also between the ranges.
The connexities scheme can be either 4 or 8 and can be changed by the function `setConnexity(vpConnexityType)`.



(a) (b)

Figure 2.1: (a) Four connexities, (b) Eight connexities.

This blob tracking is recursive. It goes through the neighborhood until it reaches the border. Reaching the border means that one of the connexity of the current pixel is out of range.

Example

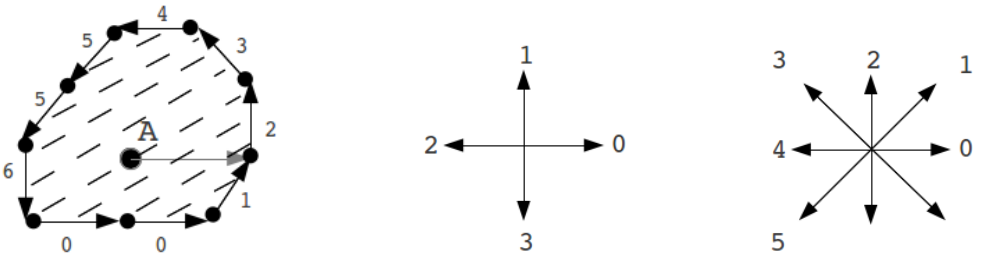
```
#include <visp/vpImage.h>
#include <visp/vpImageIo.h>
#include <visp/vpImagePoint.h>
#include <visp/vpDot.h>
#include <visp/vpDisplayX.h>
#include <visp/vpVideoReader.h>

int main(int argc, const char ** argv)
{
  vpImage<unsigned char> I;

  vpVideoReader reader;
  reader.setFileName("videoTest.mpg");
  reader.open(I);
  reader.acquire(I);

  vpDisplayX display;

  display.init(I, 640, 480, "Test tracking") ;
  vpDisplay::display(I) ;
  vpDisplay::flush(I);

  vpDot d ;

  // by using setGraphics, we request to see the all the pixel of the dot
  // in green on the screen.
  d.setGraphics(true) ;

  // we can also request to compute the dot moment m00, m10, m01, m11, m20, m02
  d.setComputeMoments(true);

  //We choose the connexity scheme
  d.setConnexity(vpDot::CONNEXITY_4);

  d.initTracking(I) ; //Tracking initialisation (by click)

  long numberOfFrame = 200;
  long currentFrame  = 0;
  while (currentFrame < numberOfFrame)
  {
    reader.getFrame(I,currentFrame)
    d.track(I) ;
    vpImagePoint cog = d.getCog(); //We get the center of Gravity

    vpDisplay::display(I) ;
    // display a red cross (size 10) in the image at the dot center
    // of gravity location
    vpDisplay::displayCross(I, cog, 10, vpColor::red) ;
    vpDisplay::flush(I) ;

    currentFrame++;
  }

  return 0;
}
```

Listing 2.1: Exemple of vpDot tracking.

2.1.2 vpDot2

Principle

Contrary to `vpDot`, `vpDot2` doesn't go through all the neighborhood of the starting point. Starting from a point A, `vpDot2` tries to reach the "right border". Once the border has been reached, it just consists in following it. This creates a Freeman chain.



(a) (b) (c)

Figure 2.2: (a) Example of vpDot2 tracking using 8-connexities neighborhood, (b) Corresponding Freeman values for four connexities, (c) Corresponding Freeman value for eight connexities.

Let's imagine the user clicked on the point A from the figure 2.2a to initialise the dot tracking. It will go all way right until neighbor are part of the dot and until the border is reached (grey arrow figure 2.2a). Afterwards, it will follow the border using connexities values and will create a Freeman chain. For the example presented in the figure 2.2a, and using the code of figure 2.2c (because it is a 8-connexities neighborhood), the Freeman chaine associated to the border will be : "2 3 4 5 5 6 0 0 1".

Remark: `vpDot2` is much more faster than `vpDot` because it doesn't need to go through all the neighbors of the initial point.

Moreover, in `vpDot2`, it's also possible to specify the shape that the tracking must have by using the function `vpDot2::setEllipsoidShapePrecision(double)`. The value passed as parameter is between 0 and 1. A zero value means you want to track a non ellipsoid shape dot.

Example

`vpDot2` works the same way as `vpDot`. You can have a look at the algorithm 2.1 and change `vpDot` by `vpDot2`.

2.2 Moving-edge trackers

2.2.1 General principle

In ViSP, the edge trackers are based on the spatiotemporal Moving Edge (ME) algorithm ([2]). This algorithm does not require an edge extraction since it manipulates only sample points and image intensities.

Subsequently, the edges can be approximated to any type of curve. In ViSP, line, ellipse and Nurbs trackers are implemented.

Each point, sampling the curve, is tracked from one image to another along the normal to the curve at this point. This search is performed using convolution in order to have a real-time tracking. The convolution mask depends on the angle in the image of the normal to the curve for the given sample. These masks are precomputed during the initialisation of the algorithm. See figure 2.3 for more details.

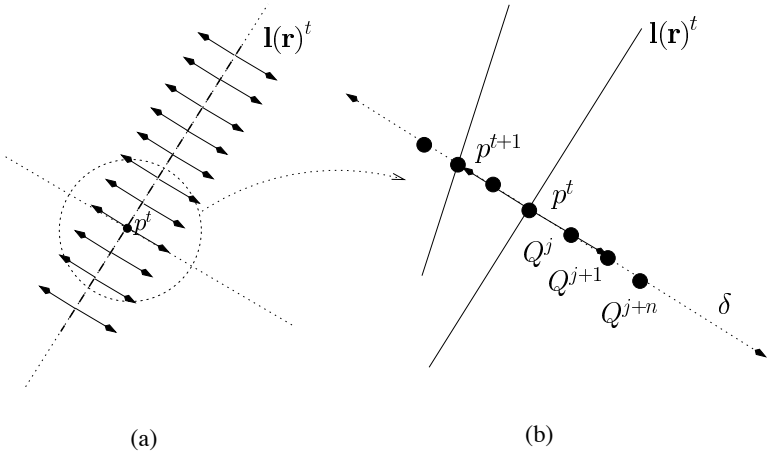


Figure 2.3: $l(r)^t$: edge at the previous iteration. In (a), the arrows represent the search range along the normal for each sample point. In (b), every candidate position is tested by a convolution with predetermined mask depending on the angle of the normal. (c) and (d) represent the masks for an angle of respectively $180\,\mathrm{deg}$ and $45\,\mathrm{deg}$.

The equation of the curve is computed from the sample points coordinates. In order to be robust to the image processing errors, a robust technique (M-Estimator, [3]) is used during the minimisation.

2.2.2 Implementation

2.2.2.1 General framework

As explained in the previous section, the trackers are based on a set of sample points. Some common declarations have been implemented in the generic `vpMeTracker` class. This class cannot be used directly since some of its functions are pure virtual.

A minimal prototype for this class is:

```
1 class vpMeTracker
2 {
3   // List of tracked moving edge points
4   std::list<vpMeSite> list ;
5   // Moving edge initialisation parameters
6   vpMe *me ;
```

```
7
8    // Constructor/destructor
9    vpMeTracker() ;
10   vpMeTracker(const vpMeTracker& meTracker) ;
11   virtual ~vpMeTracker() ;
12   void init() ;
13
14   // Display contour
15   virtual void display(const vpImage<unsigned char> &I, vpColor col)=0;
16   // Sample pixels at a given interval
17   virtual void sample(const vpImage<unsigned char> &image)=0;
18
19   void initTracking(const vpImage<unsigned char>& I);
20   // Track sampled pixels
21   void track(const vpImage<unsigned char>& I);
22   // Displays the status of moving edge sites
23   void display(const vpImage<unsigned char>& I);
24   // Set the moving edges parameters.
25   void setMe(vpMe *me);
26 };
```

Some of the functions, the pure virtual ones, depend on the type of curve to track, while others, corresponding to the low level tracking, are implemented in this generic class.

A `vpMeSite` is a sample point. It contains the position of the points and some information concerning the low level tracking (for example the image intensity).

The `me` attribute contains the configuration of the tracker. For more details, please refer to section 2.2.3.

2.2.3 Moving edge configuration

Moving edge trackers implemented in ViSP are based on the tracking of independant `vpMeSite`. This tracking is quite robust to changes in illumination and noise, but it still needs to be configured with a couple of parameters. This configuration is implemented in the `vpMe` class.

A minimal prototype for the `vpMe` class is:

```
1  class vpMe
2  {
3    double threshold; // Likelihood ratio threshold
4    double mu1; // Contrast continuity parameter (left boundary)
5    double mu2; // Contrast continuity parameter (right boundary)
6    unsigned int range; // Seek range on both sides of the reference pixel
7    unsigned int mask_size; // Convolution masks' size in pixels (masks are square)
8    double sample_step; // Distance between sampled points (in pixels)
9    int points_to_track; // Number of points to track (used only for NURBS tracking)
10   unsigned int n_mask; // Number of convolution masks available for tracking; defines resolution
11   unsigned int anglestep; // number of oriented masks used to perform the convolution
12
13   // Default constructor that initiale the previous parameters
14   vpMe::vpMe()
15   {
16     threshold = 1500 ;
17     mu1 = 0.5 ;
18     mu2 = 0.5 ;
19     range = 4 ;
20     mask_size = 5 ;
21     sample_step = 10 ;
22     n_mask = 180 ;
23     anglestep = (180 / n_mask) ;
24     ...
25   }
26
27   // Parameter setters
```

```
28   void setThreshold(const double t) { threshold = t; } // Set the likelihood threshold
29   void setMu1(const double mu1) { this->mu1 = mu1; }
30   void setMu2(const double mu2) { this->mu2 = mu2; }
31   void setRange(const unsigned int r) { range = r; }
32   void setMaskSize(const unsigned int s) { mask_size = s; ... }
33   void setSampleStep(const double s) { sample_step = s; }
34   void setPointsToTrack(const int n) { points_to_track = n; }
35   void setMaskNumber(const unsigned int n) { n_mask = n; anglestep = 180 / n_mask; ...}
36 };
```

When a candidate position is searched along the normal to the curve, the results of the convolution between the image and the mask is compared to the `threshold` value. The higher the threshold is, the sharper the transition must be to accept a point. Using a low value facilitate the detection of the edge in a low contrast environment, however it may introduce outliers in the minimisation.

The `mu1` and `mu2` values are used to reject a new candidate position for the sample point if the new convolution is too small (for `mu1`) or too high (for `mu2`) compared to the previous one. For example, if $\texttt{mu1} = \texttt{mu2} = 0.5$, the sample point is accepted only if the value of the new convolution is between 50% and 150% of the previous convolution. This procedure allows to reject points with a different transition (black-to-white instead of the previous white-to-black for example).

The `range` attribute corresponds to the distance in pixel along the normal to the curve where the new edge is searched. This value is dependant on the expected displacement of the edge between two consecutive images. The higher the value is, the slower will be the algorithm and the more there will be outliers.

The `anglestep` and `sample_step` attributes correspond to the distance between two consecutives sample points on a curve. If the object is very simple, and if the environment is not too noisy, a high value can be sufficient.

The `mask_size` value corresponds to the size of the convolution mask. This value influences the `threshold` attribute. Usually, a size of 5 or 7 allows a good detection and a good framerate.

The `n_mask` corresponds to the number of mask used to compute the angle of the edge at the sampling point. If $\texttt{n\_mask} = 180$, the step, in degree, between two consecutive mask is $\frac{360°}{180} = 2°$.

2.2.3.1 Line tracker

The `vpMeLine` class inherits from the generic `vpMeTracker` and computes the equation of a straight line from the sample points.

The equation of a line is:

$$ai + bj + c = 0 \tag{2.1}$$

Or, in polar coordinates:

$$i\cos(\theta) + b\sin(\theta) - \rho = 0 \tag{2.2}$$

With:

$$\theta = arctan(\frac{b}{a}) \tag{2.3}$$

$$\rho = \frac{-c}{\sqrt{a^2 + b^2}} \tag{2.4}$$

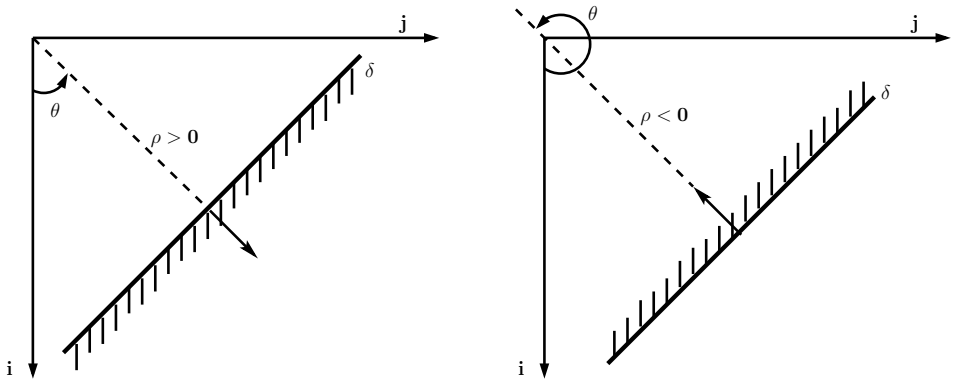


Figure 2.4: Convention to compute the values of (ρ, θ). The dashed side of the line represents the darkest region. Left figure shows the case of a white to dark pixel intensity transition, while the right figure shows a dark to white transition.

θ can be between 0 and 2π. ρ can be positive or negative. For the `vpMeLine`, the values are dependent on the pixel intensity transition (white-to-dark or dark-to-white relatively to the top left corner of the image) in the middle of the line (see figure 2.4).

The extremities of the line are updated at every iteration. A new point is added if its response to the edge detector is similar to the closest points belonging to the line.

The minimisation computes the new a, b and c parameters of the line, however the ρ, θ values are also available.

An example of use is available in ViSP (`example/tracking/trackMeLine.cpp`). A minimal example is:

```
#include <visp/vpImage.h>
#include <visp/vpMeLine.h>

int main()
{
  vpImage<unsigned char> I(240, 320);

  // acquire an image (from a file, from a webcam, ...).

  // Set the moving-edges tracker parameters
  vpMe me;
  me.setRange(25);
  me.setPointsToTrack(20);
  me.setThreshold(15000);
  me.setSampleStep(10);

  // Initialize the moving-edges line tracker parameters
  vpMeLine line;
  line.setMe(&me);

  // Initialize the tracker by clicking on two image points
  line.initTracking(I);

  while ( 1 )
  {
    // ... Here the code to read or grab the next image.

    // Track the line.
```

```
29      line.track(I);
30
31      std::cout << "(rho,theta) = (" << line.getRho() << "," << line.getTheta() << ")" << std::endl;
32    }
33    return 0;
34  }
```

2.2.3.2 Ellipse-circle tracker

The `vpMeEllipse` class inherits from the generic `vpMeTracker` and computes the equation of an ellipse or a circle.

The generic equation of an ellipse, defined by the points (i, j), is:

$$i^2 + K_0 j^2 + 2K_1 ij + 2K_2 i + 2K_3 j + K_4 = 0 \tag{2.5}$$

In the case of a circle, $K_0 = 1$ and $K_1 = 0$.

It is also possible to describe the equation of the ellipse using three parameters a, b and e. They represent respectively the semiminor axis, the semimajor axis and the angle between the major axis and the i axis (see figure 2.5).

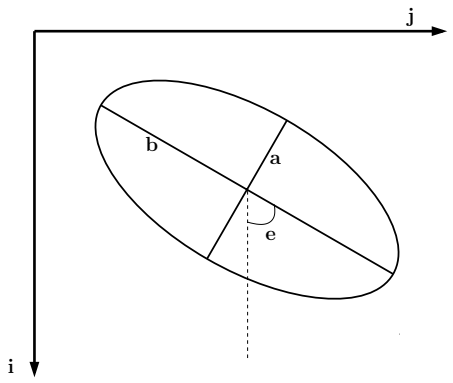


Figure 2.5: Definition of the (a, b, c) parameters.

In that case, the coordinates of of a point (i, j) which belongs to the ellipse are given by:

$$\begin{cases} i = i_c + bcos(e)cos(\alpha) - asin(e)sin(\alpha) \\ j = j_c + bsin(e)cos(\alpha) - acos(e)sin(\alpha) \end{cases} \tag{2.6}$$

with (i_c, j_c) the center of the ellipse and α the angle between $[0, 2\pi]$ covering the ellipse:

Along with the equation of the ellipse, several moments (up to the second order) are computed. For example, the 0 order moment can be obtained with the `get_m00()` method.

If the pattern to track is a circle, it is possible to enforce this constraint with the method `setCircle(true)`. In that case, the minimisation does not take into account the parameters K_0 and K_1. The result is therefore more robust.

An example is available in ViSP (`example/tracking/trackMeEllipse.cpp`). The minimal example is strictly similar to the example for the line tracker (except for the `getRho()` and `getTheta()` methods).

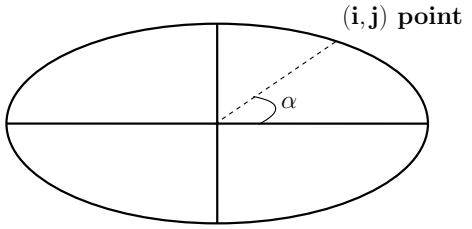


Figure 2.6: Definition of the α angle.

2.2.3.3 Nurbs tracker

Nurbs stands for Non-Uniform Rational Basis Spline. The `vpMeNurbs` class inherits from the generic `vpMeTracker` and intends to fit a Nurbs to the detected moving edges.

A Nurbs can be defined by:

- a knot vector $U = u_0, ..., u_m$, with $u_i < u_{i+1}, i = 0, ..., m$. The first and the last values are copied p times, with p the degree of the B-Spline basis functions.
- the B-Spline basis functions $N_{i,p}$ defined as:

$$N_{i,0}(u) = \begin{cases} 1 & \text{if } u_i \leq u_{i+1} \\ 0 & \text{else} \end{cases} \tag{2.7}$$

$$N_{i,p}(u) = \frac{u - u_i}{u_{i+1} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1} \tag{2.8}$$

- the control points P_i (defined by the images points).
- the weights w_i associated to each points ($w_i \geq 0$).

It is possible to use the Canny edge detector to seek the new extremities of the curve (with the method `seekExtremitiesCanny()`). This possibility is activated only if OpenCV (`http://opencv.willowgarage.com`) is linked with ViSP.

An example is available in ViSP (`example/tracking/trackMeNurbs.cpp`). The minimal example is strictly similar to the example for the line tracker (except for the `getRho()` and `getTheta()` methods).

2.3 KeyPoint detection and matching

It is possible to extract in an image a set of points of interest. These points are robust to some modifications in the image (change in illumination, change in position,...) so that they can be detected on several images. A signature for each point is extracted in the neighbouring environment and is used to match the same point between two different images. The state of the art contains a lot of different algorithms, more or less robust to change in illumination, in orientation, in scale,...

Nowadays, two algorithms are available in ViSP:

- SURF: stands for *Speeded Up Robust Feature* [1].
- Ferns: [5].

2.3.1 SURF

2.3.1.1 General description

The *Speeded Up Robust Feature* (SURF) algorithm [1] is an image point detector and descriptor. SURF detects *blob-like* features; bright region in a dark background and *vice versa*. For each point, a signature or descriptor of the neighbouring region is computed. The descriptor is either a vector of 64 or 128 values.

This detector is fast (it is possible to achieve a real-time application) and is invariant in translation, in rotation and in scale. However, it is not invariant to affine transformations.

2.3.1.2 Implementation

The SURF algorithm is implemented in the `vpKeyPointSurf` class. It requires OpenCV to work. This class inherits from the pure virtual `vpBasicKeyPoint` that contains the list of keypoints in the reference image and in the current image.

The following minimal prototype shows the main functions:

```
class vpKeyPointSurf: public vpBasicKeyPoint
{
  vpKeyPointSurf ()
  virtual   ~vpKeyPointSurf ()
  unsigned int  buildReference (const vpImage< unsigned char > &I);

  unsigned int  matchPoint (const vpImage< unsigned char > &I);

  void  display (const vpImage< unsigned char > &Iref, const vpImage< unsigned char > &Icurrent);
  void  display (const vpImage< unsigned char > &Icurrent);

  void  setHessianThreshold (double hessianThreshold);
  void  setDescriptorType (vpDescriptorType descriptorType);

  const vpImagePoint *  getAllPointsInReferenceImage ();
  void  getReferencePoint (const unsigned int index, vpImagePoint &referencePoint);
  void  getMatchedPoints (const unsigned int index,
                          vpImagePoint &referencePoint, vpImagePoint &currentPoint);

  unsigned int  getIndexInAllReferencePointList (const unsigned int indexInMatchedPointList);
  unsigned int  getReferencePointNumber () const;
  unsigned int  getMatchedPointNumber () const;

  const std::vector< vpImagePoint > &   getReferenceImagePointsList () const;
  const std::vector< vpImagePoint > &   getCurrentImagePointsList () const;
  const std::vector< unsigned int > &   getMatchedReferencePoints () const;
};
```

Only two parameters are required to use this class:

- the Hessian threshold: only the points with a determinant of the Hessian matrix greater than this threshold are considered. The default value is equal to 500. To have more point, it is possible to lower this value, however, the points will be less robusts;
- the type of descriptor: it can be either *basic* (64 values) or *extended* (128 values). With 64 values, the computation of the descriptor and the matching are faster, however it may introduce more mismatching.

At every iteration, using `getReferenceImagePointsList()` it is possible to get the list of the reference points while using `getCurrentImagePointsList()` it is possible to access to the list of the

current image points. The pairs of points are obtained using the `getMatchedReferencePoints()` function.

2.3.1.3 Example

A complete example is available in ViSP (`example/key-point/keyPointSurf.cpp`). The following example shows the main functions:

```
#include <visp/vpConfig.h>
#include <visp/vpImage.h>
#include <visp/vpKeyPointSurf.h>

int main()
{
  vpImage<unsigned char> Irefrence;
  vpImage<unsigned char> Icurrent;
  vpKeyPointSurf surf;

  //First grab the reference image Irefrence

  //Build the reference SURF points.
  surf.buildReference(Irefrence);

  //Then grab another image which represents the current image Icurrent

  //Match points between the reference points and the SURF points computed in the current image.
  surf.matchPoint(Icurrent);

  //Display the matched points
  surf.display(Irefrence, Icurrent);

  return (0);
}
```

2.3.2 Ferns

2.3.2.1 General description

Unlike the SURF algorithm, based on direct computation of the signature in the image, this method is based on a Bayesian Framework. Ferns [5] is a structure used to classify images patches. This method is really fast and is more robust to affine transformation than the SURF algorithm, on the other hand this method usually gives more outliers.

This algorithm is decomposed into two stages.

Firstly, the learning stage takes as input the reference pattern and extract the points of interests in it. A patch is extracted for every point and several simple binary tests are performed on this patch and on some synthesized views of it (to be robust to affine transformation). This learning stage is very time consumming since it requires the generation of hundreds to thousands of affine transformations.

Secondly, during the detection phase, the points of interests are extracted in the image, and the same simple binary tests are performed. This gives a response which is compared to the responses in the Ferns structure produced during the learning stage. If two responses are similar, then the two points are likely to be the same.

2.3.2.2 Implementation

The class `vpFernClassifier`, based on OpenCV, implements the Ferns classifier. The points are detected using the *YAPE* algorithm.

The minimal prototype below shows the main functions:

```
class vpFernClassifier
{
  vpFernClassifier();

  // build reference
  virtual unsigned int buildReference(const vpImage<unsigned char> &I);
  virtual unsigned int buildReference(const vpImage<unsigned char> &I,
                                      vpImagePoint &iP,
                                      const unsigned int height, const unsigned int width);
  virtual unsigned int buildReference(const vpImage<unsigned char> &I,
                                      const vpRect& rectangle);

  // matching
  virtual unsigned int matchPoint(const vpImage<unsigned char> &I);
  virtual unsigned int matchPoint(const vpImage<unsigned char> &I,
                                  vpImagePoint &iP,
                                  const unsigned int height, const unsigned int width);
  virtual unsigned int matchPoint(const vpImage<unsigned char> &I,
                                  const vpRect& rectangle);

  // display
  virtual void display(const vpImage<unsigned char> &Iref,
                       const vpImage<unsigned char> &Icurrent);
  virtual void display(const vpImage<unsigned char> &Icurrent);

  // io methods
  void load(const std::string& dataFile, const std::string& objectName);
  void record(const std::string& objectName, const std::string& dataFile);

  inline void setBlurSettings(const bool blur, int sigma, int size);

  const std::vector<cv::Point2f>& getRefPt() const {return refPt;}
  const std::vector<cv::Point2f>& getCurPt() const {return curPt;}

};
```

As described in the section 2.3.2.1, the algorithm needs to train a structure on the reference image (function `buildReference()`). This training requires a lot of computation and may take up to several minutes. This training has to be done once for every reference pattern to detect. The training can be saved in a file (up to several hundred megabytes large) using the `record(...)` method.

To detect a reference pattern, the class must be initialised either directly (using the method `buildReference(...)` or using a previous training (using the method `load(...)`). The detection and the matching itself is performed using one of the `matchPoint(...)` method.

It is possible to get the points in the reference image and in the current image using either the methods `getRefPt()` and `getCurPt()` (in this case, the points are in the OpenCV format) or using the methods `getReferenceImagePointsList()` and `getCurrentImagePointsList()` (in the ViSP format).

2.3.2.3 Example

A complete example is available in ViSP (`example/key-point/fernClassifier.cpp`). The following example shows the main functions:

```
#include <visp/vpConfig.h>
#include <visp/vpImage.h>
#include <visp/vpFernClassifier.h>

int main()
{
  vpImage<unsigned char> Ireference;
  vpImage<unsigned char> Icurrent;
  vpFernClassifier fern;

  //First grab the reference image Ireference

  //Build the reference SURF points.
  fern.buildReference(Ireference);

  //Then grab another image which represents the current image Icurrent

  //Match points between the reference points and the SURF points computed in the current image.
  fern.matchPoint(Icurrent);

  //Display the matched points
  fern.display(Irefrence, Icurrent);

  return (0);
}
```

The use of this class is very similar to the `vpKeyPointSurf` class.

2.4 KLT

2.4.1 Description

The Kanade-Lucas-Tomasi (KLT) feature tracker ([4] and [6]) is designed to detect and track a set of image points in a sequence of images. It assumes a small displacement between two consecutives frames.

The tracking is based on the gradients in the image. If d is the displacement of a feature x between two images I_t and I_{t+1} then

$$I_{t+1}(x) = I_t(x + d) \tag{2.9}$$

This expression can be approximated to:

$$I_{t+1}(x) \approx I_t(x) + dI'_t(x) \tag{2.10}$$

Only a patch surrounding a feature is used to compute the displacement of this feature. d is computed by minimisation.

The points tracked are usually detected using the eigen-values of the matrix of the second order derivatives of the image. Only points with high eigenvalues are considered as points of interests.

2.4.2 Implementation

The KLT tracker is implemented in ViSP in the `vpKltOpencv` class. It requires the OpenCV library to work. A minimal prototype for this class is:

```
class vpKltOpencv
{
  vpKltOpencv ()

```

```
5   void  initTracking (const IplImage *I, const IplImage *mask=NULL)
6
7   void  track (const IplImage *I)
8
9   void  display (const vpImage< unsigned char > &I, vpColor color=vpColor::red)
10
11  void  setMaxFeatures (const int input)
12  void  setWindowSize (const int input)
13  void  setQuality (double input)
14  void  setMinDistance (double input)
15  void  setHarrisFreeParameter (double input)
16  void  setBlockSize (const int input)
17  void  setPyramidLevels (const int input)
18
19  int   getNbFeatures () const
20  int   getNbPrevFeatures () const
21  int   getMaxFeatures () const
22
23  void  getFeature (int index, int &id, float &x, float &y) const
24  void  getPrevFeature (int index, int &id, float &x, float &y) const
25  void  addFeature (const int &id, const float &x, const float &y)
26  void  suppressFeature (int index)
27 };
```

This class takes as input a `IplImage*`, a structure defined by OpenCV. It is possible to create an `IplImage*` from a `vpImage` using the function `vpImageConvert::convert(vpImage<>&, IplImage*)`. If it is possible, this conversion is direct and there is no memory recopy.

The detection and the tracking of the features depends on the following parameters:

- The maximum number of features (`setMaxFeatures()`): specifies the maximum number of features to track in the image;
- The size of the window (`setWindowSize()`): specifies the size of the window for the subpixel computation of the coordinates. It is also used for the computation of the optical flow (the new position of the point);
- The quality (`setQuality()`): ratio between the maximal eigen value of the set of points and the minimal one. Any points with an eigen-value below $ratio \times max(eigenvalues)$ is rejected;
- The minimal distance (`setMinDistance()`): the minimal distance, in pixel, between two points during the detection;
- The Harris parameter (`setHarrisFreeParameter()`): the k value in the Harris detector. Usually, a value of $k = 0.04$ is correct;
- The block size (`setBlockSize()`): the size of the averaging block used to track the features;
- The pyramid levels (`setPyramidLevels()`): the maximal level of pyramid. If the level is zero, then no pyramid is computed for the optical flow. The higher the value, the more robust the tracking will be to large displacement between two consecutive frames.

2.4.3 Example

A complete example of the class is available in ViSP (`example/tracking/trackKltOpencv.cpp`). A minimal example is:

```
#include <visp/vpConfig.h>
#include <visp/vpImage.h>
#include <visp/vpDisplay.h>
#include <visp/vpKltOpencv.h>
#include <visp/vpImageConvert.h>

int main()
{
#if VISP_HAVE_OPENCV_VERSION >= 0x010100
  vpImage<unsigned char> I;
  IplImage* Icv = NULL;
  vpKltOpencv klt;

  //First grab the initial image I

  //Convert the image I to the IplImage format.
  vpImageConvert::convert(I, Icv);

  //Initialise the tracking on the whole image.
  klt.initTracking(Icv, NULL);

  while(true)
  {
    // Grab a new image and convert it to the OpenCV format.
    vpImageConvert::convert(I, Icv);

    // Track the features on the current image.
    klt.track(Icv);

    // Display the features tracked at the current iteration.
    klt.display(I);
  }

  cvReleaseImage(&Icv);
#else
  std::cout << "vpKltOpencv requires ViSP with OpenCV." << std::endl;
#endif
  return(0);
}
```

The conversion between the ViSP format and the OpenCV format is required for the initialisation and for the tracking.

Chapter 3

Networking

In this chapter, we will present networking tools through ViSP. Image processing and computer vision algorithms are often heavy in term of computation. Whereas the capacity of computation of most robots is limited. Embedding these algorithms in robots are sometimes hard and painfull, this is the reason why we often need to externalise the computation. And one solution is through networking.

3.1 What is the Transmission Control Protocol

A network is defined as a connection between several computer used to exchange data according to a specific protocol. This protocol is a system of digital message formats and rules for exchanging those messages. In computer science several protocols are availables like the User Datagram Protocol (UDP) or Transmission Control Protocol (TCP). Here, we will focus on the TCP protocol, which is the one used in ViSP.

TCP is the one of the two original components of the suite, complementing the Internet Protocol (IP), and therefore the entire suite is commonly referred to as TCP/IP. TCP provides reliable, ordered delivery of a stream of octets from a program on one computer to another program on another computer. Typically, when two (or more) computer are communicating, we have one server commucating with client(s). Note that a client can also communicate with several servers.

3.2 Server vs Client

3.2.1 Overview

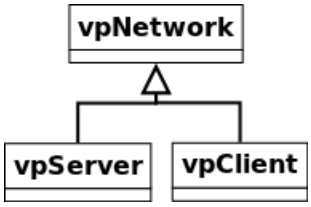


Figure 3.1: Overview of networking classes implemented in ViSP.

In ViSP, like in other libraries in general, servers and clients have a lot in common when there is a two ways communication in place. This is the reason why both `vpServer` and `vpClient` derive from `vpNetwork` (see Figure 3.1), which contains basic networking functionnalities. Indeed, here each member of the network is able to:

1. Send messages over the network;
2. Receive messages from another computer on the network;
3. Create a connection with another computer of the network.

The messaging proccess corresponding to functionality (1) and (2) will be detailed in the next section which shows two different approaches to send and receive messages depending on the situation. Concerning the third point functionality, note that the connection to another member of the network is quite different for the server side than the client one.

3.2.2 Server side

To create a network, it automatically requires a server. Then clients will be able to connect to this server. To launch a server, all you need is a valid port. A port is an application-specific or process-specific software construct serving as a communications endpoint in a computer's host operating system. It is associated with an IP address, which will be the adress where the server is launched. The code below shows how to create a server on the port 35000. Note that the port must be below 65535.

```
vpServer server(35000);
```

Then, the server has to be started:

```
server.start();
```

As we said before, the ViSP server can have several clients. The maximum number of clients can be fixed through the `setMaxNumberOfClients(uint)` function.

```
server.setMaxNumberOfClients(10);
```

Once the server is started and ready to host clients, the server's task must be defined in an infinite loop. The first thing it needs to do is obviously to check if there are new connected clients. This is done through the `checkForConnections()` function.

```
while(1){
  server.checkForConnections(); // Check if new clients are connected
  // ...
  // Messaging process
  // ...
}
```

Listing 3.1: Before messages are processed, the server checks if new clients are connected.

3.2.3 Client side

As it is explained before, a network isn't a network without any server. The server is used to host clients. When a server is created and started, it opens a specified port associated to the IP of the server. In conclusion, all the clients need to know the server's IP or hostname and the openned port to connect to it.

```
vpClient client;
client.connectToHostname("servername", 35000); // connection by hostname
// OR
client.connectToIP("134.10.2.3",35000); // connection by IP
```

Note that a client can connect to several servers. Once the client is connected to a server, all you need is to write the infinite loop corresponding to the messaging process (as we saw in Listing 3.1).

3.3 Messaging process

The advantage with TCP/IP is that there is no loss during data transmission. If the packages are not received yet, they are stored in a queue. Besides, packages that are sent always arrive in the same order. For remind, ViSP network tool has been designed in order to externalise computer vision algorithms. Basically, data that will transit will be images, images points, velocities, matrices... As the algorithms hardly depend on these data, no loss is admissible during transmission. This motivates TCP/IP choice.

However, before using ViSP network tool, it is required to define the dialogue in place between the computers of the network. To this end, the followings questions may help.

- Does a member of the network know which other member he is sending/receiving messages?
- Does a member know what he is supposed to receive?
- Is he sending/receiving simple data?
- If a member receives messages from multiple other members, does he have to know in which order he is supposed to receive those messages?

All these questions are primordials and tell you if you should rather use the object mode or the request mode described in the following sections.

3.3.1 Object mode

Overview

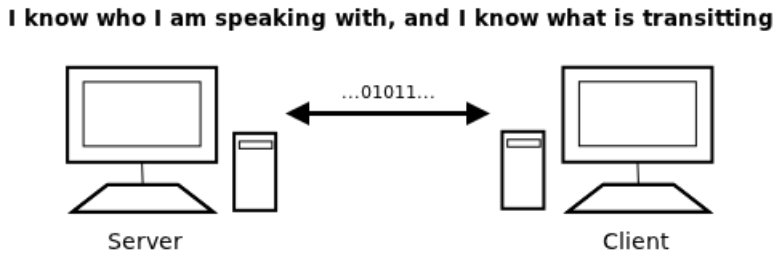


Figure 3.2: Object mode situation.

The object mode corresponds to the simplest situation. Here we assume that there are only two computers in the network: one server and one client (see Figure 3.2). So each member of the network obviously knows who he is talking with. Besides, another important assumption is that each member knows exactly what and when he is receiving (or what and when the other is sending). Once all these criteria are combined, writting

the transmission process for the infinite loop (Listing 3.1) is easy.

Either servers or clients can send and receive messages over the `vpNetwork::send()` and `vpNetwork::receive()` functions.

```
// Server or client sending a message
server.send(&object, sizeof(object));
client.send(&object, sizeof(object));

// Server or client receiving a message
server.receive(&object, sizeof(object));
client.receive(&object, sizeof(object));
```

Listing 3.2: Sending and receiving messages in the object mode.

On Listing 3.2 we can see that object are passed as pointers. This is because the object passed as parameters will be directly used to store the received data (in receiving case).

Warning : As we said before, the object mode can only be used on simple objects. By simple object, we mean an object that doesn't contain any pointer data, vectors, arrays, virtual methods, etc. Simplier, an object that only contains primitive objects.

Example

Let's make an example showing how to send and receive an interger over the network using the object mode. Here, the server side will be quite similar to the client one, as they will have the same task.

```
#include <iostream>
#include <visp/vpServer.h>

int main(int argc,const char** argv)
{
  int port = 35000;
  vpServer serv(port); // Launch the server on localhost
  serv.start();

  bool run = true;
  int val;

  while(run){
    serv.checkForConnections();

    if(serv.getNumberOfClients() > 0)
    {
      if(serv.receive(&val) != sizeof(int)) // Receiving a value from the first client
         std::cout << "Error while receiving" << std::endl;
      else
        std::cout << "Received : " << val << std::endl;

      val ++;
      if(serv.send(&val) != sizeof(int)) // Sending the new value to the first client
        std::cout << "Error while sending" << std::endl;
      else
        std::cout << "Sending : " << val << std::endl;
    }
  }

  return 0;
}
```

Listing 3.3: Server side example that shows how a server can receive an integer from a client, increment it and send it back.

```
#include <iostream>
#include <visp/vpClient.h>

int main(int argc, char **argv)
{
  std::string servername = "localhost";
  unsigned int port = 35000;

  vpClient client;
  client.connectToHostname(servername, port);
  // or client.connectToIP("127.0.0.1",port);

  int val = 0;

  while(1)
  {
    if(client.send(&val) != sizeof(int))    // Sending the new value to the first client
      std::cout << "Error while sending" << std::endl;
    else
      std::cout << "Sending : " << val << std::endl;

    if(client.receive(&val) != sizeof(int)) // Receiving a value from the first client
        std::cout << "Error while receiving" << std::endl;
    else
      std::cout << "Received : " << val << std::endl;
  }

  return 0;
}
```

Listing 3.4: Client side example that shows how a client can connect to a server in order to send and receive an integer.

3.3.2 Request mode

Overview

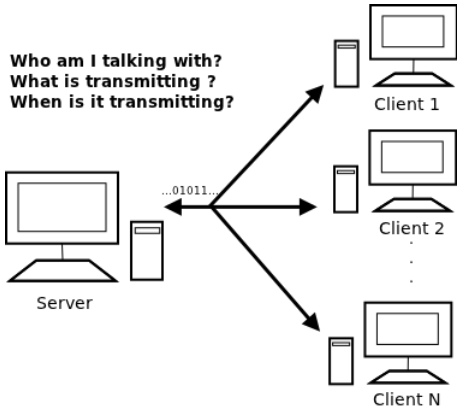


Figure 3.3: Request mode situation.

Contrary to the object mode, in request mode you don't need to know who you are talking with. You also don't need to know what you are exactly supposed to receive. The data transiting on the network will be requests.

What is a request? A request is an alternative way to send any kind of object, or even more several ojects in the same time. A request is identified by a unique ID. Servers and clients are made to handle `vpRequest` objects, which is an abstract class that has to be derived as users wishes. For better understanding, let's define a request aimed to be used for `vpImage` transmission.

```
#ifndef vpRequestImage_H
#define vpRequestImage_H

#include <visp/vpImage.h>
#include <visp/vpRequest.h>

class vpRequestImage : public vpRequest
{
private:
  vpImage<unsigned char> *I;

public:
  vpRequestImage();
  vpRequestImage(vpImage<unsigned char> *);
  ~vpRequestImage();

  virtual void encode();
  virtual void decode();
};
```

Listing 3.5: Declaration of `vpRequestImage` class that defines the request used to transmit images over the network.

When deriving the `vpRequest` class, a user has to redefine `encode()` and `decode()` functions. Those functions will be used to encode data when sending a message over the network and to decode data when receiving this request.

```
#include <vpRequestImage.h>

vpRequestImage::vpRequestImage(){
  request_id = "image";
}

vpRequestImage::vpRequestImage(vpImage<unsigned char> *Im){
  request_id = "image";
  I = Im;
}

vpRequestImage::~vpRequestImage(){}

void vpRequestImage::encode(){
  clear();

  unsigned int h = I->getHeight();
  unsigned int w = I->getWidth();

  addParameterObject(&h);
  addParameterObject(&w);
  addParameterObject(I->bitmap, h*w*sizeof(unsigned char));
}

```

```
25 void vpRequestImage::decode(){
26   if(listOfParams.size() == 3){
27     unsigned int h, w;
28     memcpy((void*)&h, (void*)listOfParams[0].c_str(), sizeof(unsigned int));
29     memcpy((void*)&w, (void*)listOfParams[1].c_str(), sizeof(unsigned int));
30
31     I->resize(h, w);
32     memcpy((void*)I->bitmap, (void*)listOfParams[2].c_str(), h*w*sizeof(unsigned char));
33   }
34 }
```

Listing 3.6: Definition of `vpRequestImage` class that defines the request used to transmit images over the network

As we said, each request must have a different ID. In the example presented in Listing 3.6 the ID is set to `image`. When a member of the network is trying to receive a request, he is first checking the ID. This ID will specify the type of request received and will determine the decoding function to use. This is the reason why, first, each member of the network has to know what kind of request he is suppose to receive. For that we use the `addDecodingRequest()` method that can be used by the client or the server.

```
1 vpRequestImage reqImage(&image);
2 client.addDecodingRequest(&reqImage); // Can also be done on the server
```

Listing 3.7: Add a request to the list of request that the client or server is supposed to receive.

Let's get back to the `encode()` and `decode()` functions. In the example presented Listing 3.7 we want to encode and decode a `vpImage`. A `vpRequest` can have several parameters. When writing the encoding and decoding functions of our request, it is necessary to identify what variables we need. Each variable will represent one parameter. Parameters are added through the `addParameterObject()` function to add an object or through the `addParameter()` to add a message corresponding to string or array of characters. In `vpRequestImage` example presented Listing 3.6, to encode the image important variables are the size of the image and the bitmap. That is why three call to `addParameterObject()` are necessary. When decoding it, all you need to do is to retransform the parameters (that are now on string form) on the right form. After decoding, each of them are stored in `listOfParams` internal variable in the same order than during the encoding.

In request mode, sending and receiving requests works quite similarly than for the object mode.

```
1  vpRequestImage reqImage(&image);
2
3  //    Sending
4  server.sendRequest(reqImage); // OR
5  server.sendAndEncodeRequest(reqImage);
6
7  //    Receiving
8  server.receiveRequest(); // OR
9  server.receiveRequestOnce; // OR
10 server.receiveAndDecodeRequest(); // OR
11 server.receiveAndDecodeRequestOnce();
```

Listing 3.8: Transmission process in request mode. Note that the same methods can also be applied to a client.

For the sending, `sendRequest()` function send a request and suppose that this request has already been encoded. Otherwise, `sendAndEncodeRequest()` can be used. It encodes plus sends the request. Concerning the reception, `receiveRequest()` and `receiveRequestOnce()` methods receive requests but don't decode them. On the other hand `receiveAndDecodeRequest()`

and `receiveAndDecodeRequestOnce()` receive and also decode the request. Note that the difference between `receiveRequest()` and `receiveRequestOnce()` but also between `receiveAndDecodeRequest()` and `receiveAndDecodeRequestOnce()` is that in the first case it receives requests until there is data in the transmission buffer, while in the second case it just receives once.

Example

Here is the complete example of a `vpServer` and `vpClient` communication using the request mode. The client acquire images and sends them to the server that does the display. To this end we use the `vpRequestImage` class defined before in Listings 3.5 and 3.6.

```
#include <visp/vpServer.h>
#include <visp/vpDisplayX.h>
#include <visp/vpDisplayGDI.h>

#include "vpRequestImage.h" // The content of this file is provided in Listing 3.5

int main(int argc,const char** argv)
{
  vpServer serv(35000);
  serv.start();

#ifdef UNIX
  vpDisplayX display;
#else //Win32
  vpDisplayGDI display;
#endif

  vpImage<unsigned char> I;

  vpRequestImage reqImage(&I); // The definition of this class is given in Listing 3.6
  serv.addDecodingRequest(&reqImage);

  bool run = true;
  while(run){
    serv.checkForConnections();

    if(serv.getNumberOfClients() > 0){
      int index = serv.receiveAndDecodeRequestOnce();
      std::string id = serv.getRequestIdFromIndex(index);

      if(id == reqImage.getId()){
        if (! display.isInitialised() )
          display.init(I, -1, -1, "Remote display");

        vpDisplay::display(I) ;
        vpDisplay::flush(I);

        if ( vpDisplay::getClick(I, false) ) // A click in the viewer to exit
          run = false;
      }
    }
  }
  return 0;
}
```

Listing 3.9: Server side example that receive images as request from a client.

```
#include <iostream>

#include <visp/vpClient.h>
```

```
4  #include <visp/vpV4l2Grabber.h>
5  #include <visp/vpImage.h>
6
7  #include "vpRequestImage.h" // The content of this file is provided in Listing 3.5
8
9  int main(int argc, char **argv)
10 {
11 #if defined(VISP_HAVE_V4L2)
12   vpImage<unsigned char> I; // Create a gray level image container
13
14   // Create a grabber based on v4l2 third party lib (for usb cameras under Linux)
15   vpV4l2Grabber g;
16   g.setScale(1);
17   g.setInput(0);
18   g.open(I);
19
20   vpClient client;
21   client.connectToHostname("localhost", 35000);
22   //client.connectToIP("127.0.0.1",port);
23
24   vpRequestImage reqImage(&I); // The definition of this class is given in Listing 3.6
25   while(1)
26   {
27     // Acquire a new image
28     g.acquire(I);
29     client.sendAndEncodeRequest(reqImage);
30   }
31   return 0;
32 #endif
33 }
```

Listing 3.10: Client side example that acquires and sends images as request.

Bibliography

[1] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. In *9th European Conf. on Computer Vision*, pages 404–417, Graz, Austria, May 2006.

[2] P. Bouthemy. A maximum likelihood framework for determining moving edges. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 11(5):499–511, May 1989.

[3] P.-J. Huber. *Robust Statistics*. Wiler, New York, 1981.

[4] B.D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In *Int. Joint Conf. on Artificial Intelligence, IJCAI'81*, pages 674–679, 1981.

[5] M. Ozuysal, P. Fua, and V. Lepetit. Fast keypoint recognition in ten lines of code. In *IEEE Int. Conf. on Computer Vision and Pattern Recognition*, 2007.

[6] C. Tomasi and T. Kanade. Detection and tracking of point features. Technical Report CMU-CS-91-132, Carnegie Mellon University Technical Report, April 1991.