ViSP 2.6.1: Visual Servoing Platform

Image manipulation

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

September 1, 2011

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

Contents

1 The image structure

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), each pixel value being of type `Type`. The image size is `height x width`.

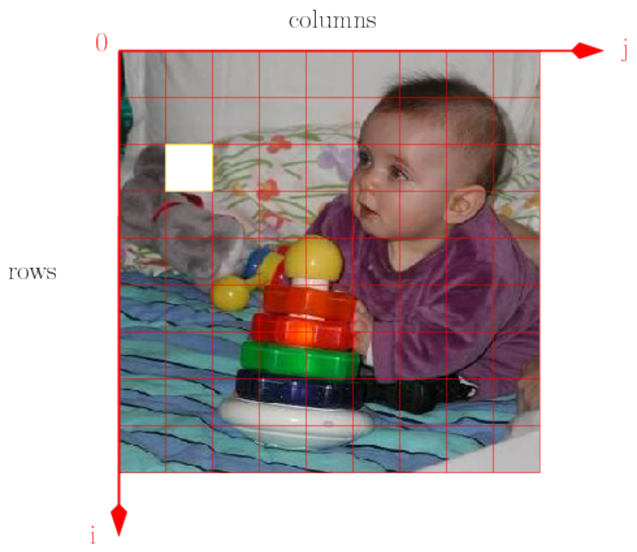

Figure 1: ViSP image representation. The pixel in white is at the coordinate (2,1).

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 2). Such a structure allows a fast access to each element of the image (see section 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.

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.

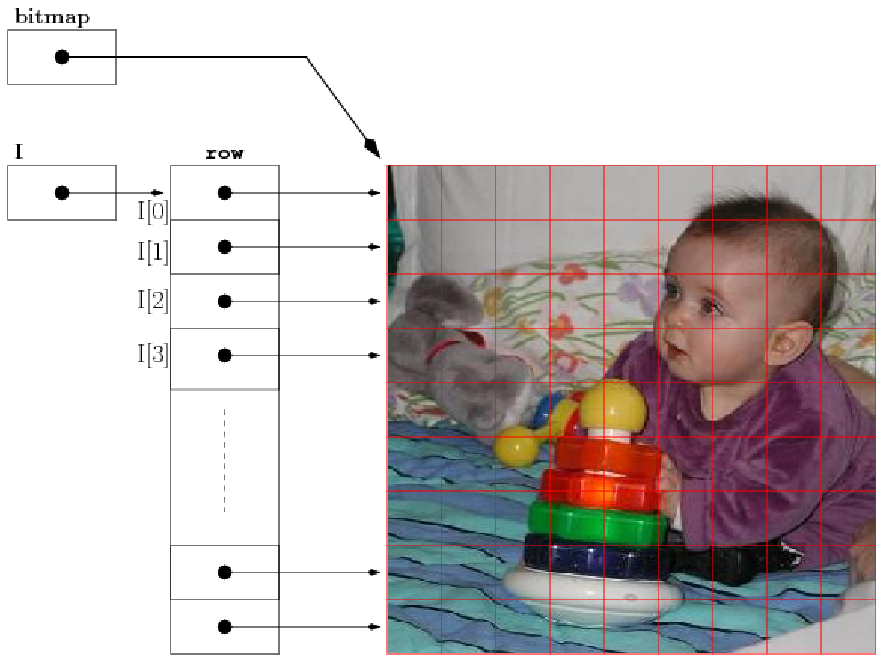

Figure 2: ViSP image data structure.

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

1.2 Image types

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 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;
```

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.

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

Then a color image can be created (see figure 4):

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

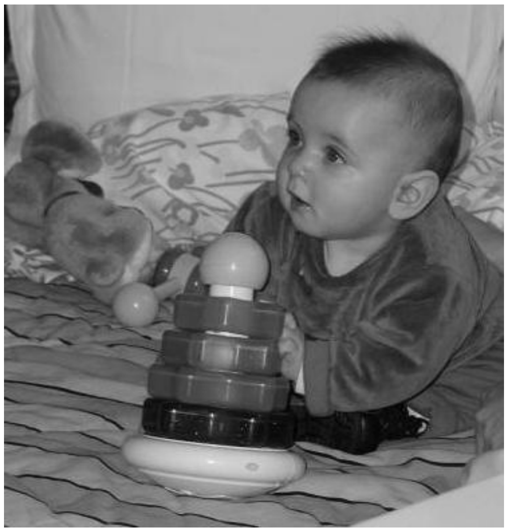

Figure 3: Example of a luminance image: `vpImage<unsigned char>`.

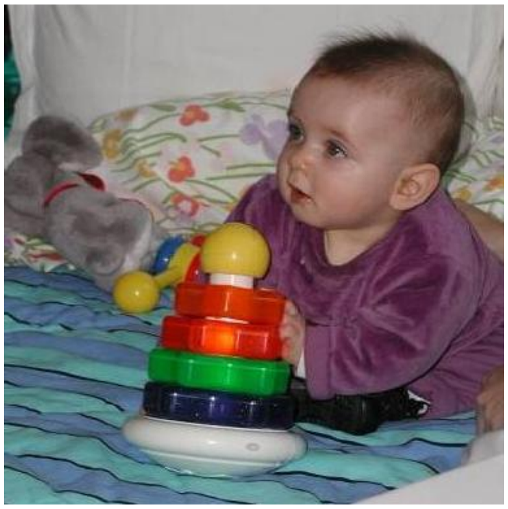

Figure 4: Example of a RGB color image: `vpImage<vpRGBa>`.

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 1.6).

1.3 Allocating and releasing images

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

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

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

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

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 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.

1.4 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 $0 \leq \mathtt{i} < \mathtt{height}$ and $0 \leq \mathtt{j} < \mathtt{width}$. 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)
```

1.5 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.
```

1.6 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).

1.7 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.
```

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 2.1, 2.2 and 2.3 depending on the filename extension.

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") ;
```

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") ;
```

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") ;
```

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.

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>
```

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 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;

  // Load a grey image from the disk. Klimt.ppm image is part of the ViSP
  // image data set available from http://www.irisa.fr/lagadic/visp/download.html
  std::string filename = "./Klimt.ppm";
  vpImageIo::read(I, filename) ;

#ifdef VISP_HAVE_GTK
  // Create a display using GTK
  vpDisplayGTK display;

  // For this grey level image, open a GTK display at position 100,100
  // in the screen, and with title "GTK display"
  display.init(I, 100, 100, "GTK display") ;

  // Display the image
  vpDisplay::display(I) ;

  // Display in overlay a red cross at position 100,10 in the
  // image. The lines are 20 pixels long
  ip.set_i( 200 );
  ip.set_j( 200 );
  vpDisplay::displayCross(I, ip, 20, vpColor::red, 3) ;

  // Display in overlay a horizontal red line
  ip1.set_i( 10 );
  ip1.set_j( 0 );
  ip2.set_i( 10 );
  ip2.set_j( I.getWidth() );
  vpDisplay::displayLine(I, ip1, ip2, vpColor::red, 3) ;

  // Display in overlay a vertical green dot line
  ip1.set_i( 0 );
  ip1.set_j( 20 );
  ip2.set_i( I.getWidth() );
  ip2.set_j( 20 );
  vpDisplay::displayDotLine(I, ip1, ip2, vpColor::green, 3) ;

  // Display in overlay a blue arrow
  ip1.set_i( 0 );
  ip1.set_j( 0 );
  ip2.set_i( 100 );
  ip2.set_j( 100 );
  vpDisplay::displayArrow(I, ip1, ip2, vpColor::blue, 8, 4, 3) ;

  // Display in overlay some circles. The position of the center is 200, 200
  // the radius is increased by 20 pixels for each circle
  for (unsigned i=0 ; i < 5 ; i++) {
    ip.set_i( 200 );
    ip.set_j( 200 );
    vpDisplay::displayCircle(I, ip, 20*i, vpColor::white, false, 3) ;
  }

  // Display in overlay a rectangle.
  // The position of the top left corner is 300, 200.
  // The width is 200. The height is 100.
  ip.set_i( 280 );
  ip.set_j( 150 );
  vpDisplay::displayRectangle(I, ip, 270, 30,vpColor::purple, false, 3) ;

  // Display in overlay a yellow string
  ip.set_i( 300 );
  ip.set_j( 160 );
  vpDisplay::displayCharString(I, ip,
             "ViSP is a marvelous software",
             vpColor::black) ;
  //Flush the display : without this line nothing will appear on the screen
  vpDisplay::flush(I);

  // Create a color image
  vpImage<vpRGBa> Ioverlay ;
  // Updates the color image with the original loaded image and the overlay
  vpDisplay::getImage(I, Ioverlay) ;

  // Write the color image on the disk
  filename = "./Klimt.overlay.ppm";
  vpImageIo::write(Ioverlay, filename) ;

  // If click is allowed, wait for a mouse click to close the display
  std::cout << "\nA click to close the windows..." << std::endl;
  // Wait for a blocking mouse click
  vpDisplay::getClick(I) ;

  // Close the display
  vpDisplay::close(I);
#endif

  return 0;
}
```

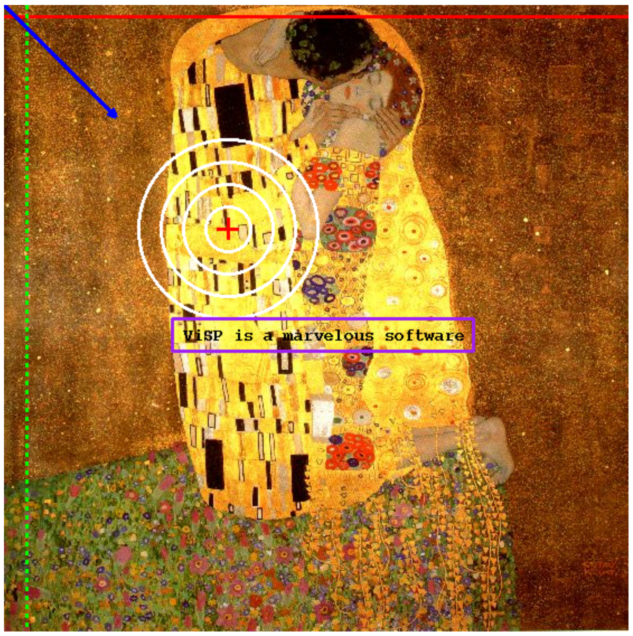

Figure 5: Displayed image by the example presented in section 3.2

4 Image Acquisition

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

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 ;
} ;
```

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.

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;
}
```

4.2.2 Interface to firewire camera: apple quicktime

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

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
}
```

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
}
```

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 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) ;
  }
}
```

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
}
```