ViSP 2.6.1: Visual Servoing Platform

The wireframe simulator

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

September 1, 2011

François Chaumette
Eric Marchand
Nicolas Melchior
Fabien Spindler

Contents

1 Introduction

Since the version 2.6.0, ViSP provides a wireframe simulator. You may ask you why we add a new simulator in ViSP? It exists indeed already a simulator based on several third party libraries like Coin, SoQt, SoXt and Qt. But it is not really easy to use and to install. Moreover, the third party libraries which are required, have problems of compatibility depending on the versions of the different softwares.

Thus, a new simulator has been added in ViSP. As shown in Figure 1, it enables to project wireframe scenes in the field of view of virtual cameras. Compared to the first simulator, it does not require any third party library. Thus, you can use it as soon as you build ViSP. Moreover, the new simulator has been written in order to be easily integrated in every projects using ViSP.

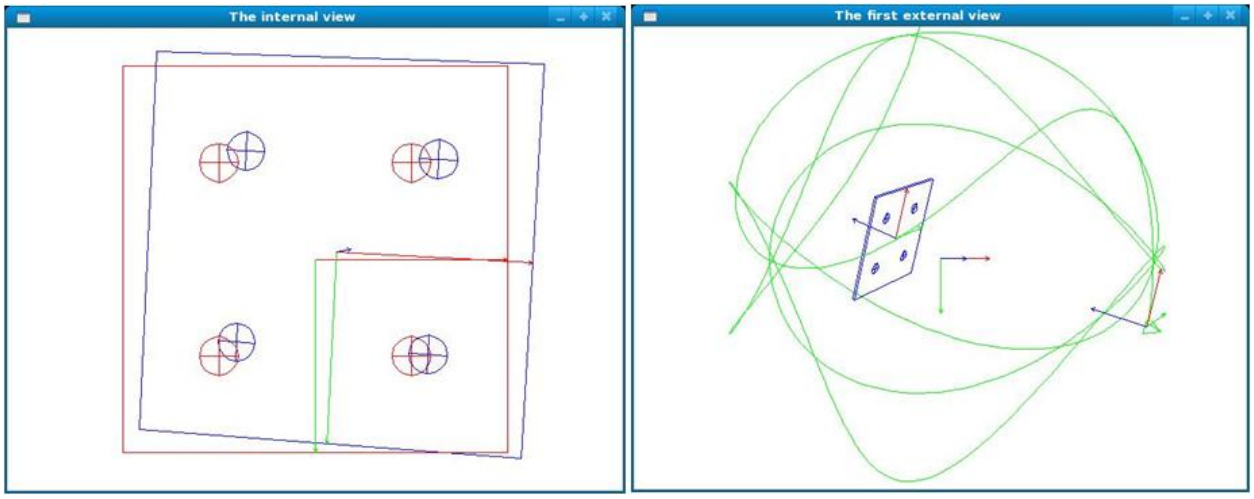

Figure 1: Screenshots of a result given by the simulator. The left window is the view of the main camera with in blue the object at the current position and in red the object at the desired position. The right window is the view of an external camera. The camera and its trajectory are displayed in green.

This document aims to explain you how to use the wireframe simulator. It will also show all its capabilities thanks to one example and three demonstrations that you can find in the demo folder of the ViSP-source code.

2 The simulator's backgrounds

In this section, the different parameters used in the simulator are presented. Indeed, you will have to manipulate several frames and homogeneous matrices which describe the displacement between two frames.

The simulator is composed by several virtual cameras. The first one is the main camera which typically provides the images used for the processing (see Figure 1 left screenshot). The second one is the main external camera. It represents a point of view to display the scene composed by the wireframe object and the main camera (see Figure 1 right screenshot). In a later section, you will see that the main external camera

has interesting capabilities. And it is possible to add other external cameras which display the scene from other points of view.

All the elements of the simulator (ie the wireframe object and the camera) have their own frame as shown in the Figure 2. In the simulator, all the frames are located regarding to the position of a world reference frame which is fixed.

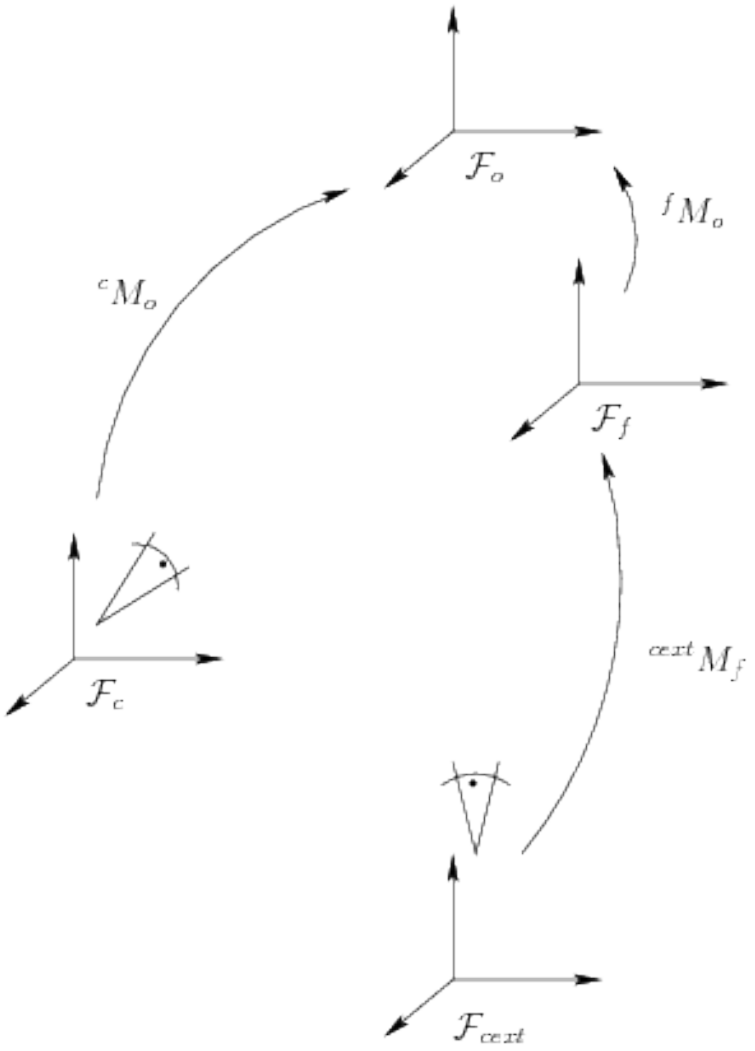

Figure 2: Scheme of the virtual scene. The different frames are the world reference frame $\mathcal{F}_f$, the object frame $\mathcal{F}_o$, the main camera frame $\mathcal{F}_c$ and an external camera frame $\mathcal{F}_{cext}$

Thus, the main frames are the following :

1. The world frame $\mathcal{F}_f$: This frame can be considered as a reference frame also called fix frame. All the other frame locations are computed regarding to $\mathcal{F}_f$. Its position is fixed and set during the initialisation. At the begining the world frame is at the same location as the first position of the object frame.

2. The object frame $\mathcal{F}_o$: This is the frame linked to the object. Later you will see that it is possible to move the object regarding to the $\mathcal{F}_f$.

3. The main camera frame $\mathcal{F}_c$: This is the frame linked to the main camera. To precise, this is the virtual camera which represents the real camera used during a visual servoing. In its field of view appears the desired and the current position of the object.

4. The external camera frame $\mathcal{F}_{cext}$: This is the frame linked to an external virtual camera. To precise, these cameras show the current scene, ie the current object location and the main camera (if they are in the field of view of course).

The gap between two frames is represented by several homogeneous matrices. The main homogeneous matrices are the following:

1. ${}^{f}M_o$: The homogeneous matrix which represents the pose of the object frame regarding to the world reference frame.

2. ${}^{c}M_o$: The homogeneous matrix which represents the pose of the object frame regarding to the main camera's frame.

3. ${}^{cd}M_o$: The homogeneous matrix which represents the desired pose of the object frame regarding to the main camera's frame.

4. ${}^{cext}M_f$: The homogeneous matrix which discribes the gap between one external camera frame and the world frame.

To project the wireframe scene into the image plan, a simple perspective projection model is used. The camera parameters are partially taken into account. Indeed, it is possible to define the p_x and p_y pixel ratio camera parameters. The image plane of the camera is located one meter before the virtual camera. Figure 3 illustrates the perspective projection.

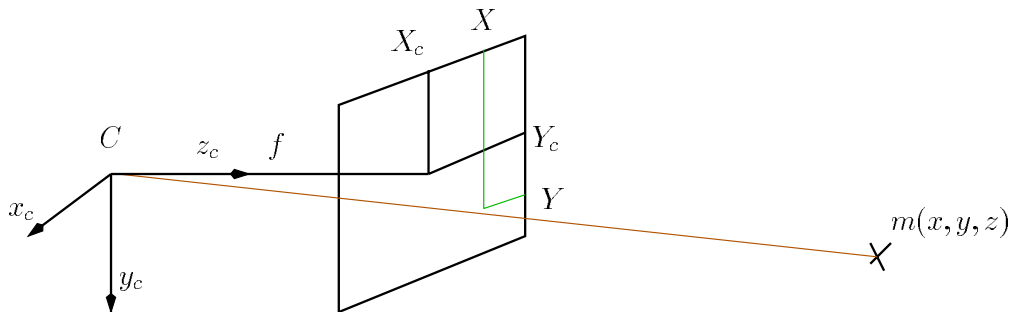

Figure 3: Scheme which present the principle of the perspective projection. The 3D point $m(x, y, z)$ is projected into the image plane which is 1 meter far from the camera. X_c and Y_c are the coordinates of the optical center in the image plan. X and Y are the coordinates of the projection of the 3D point m.

3 Simulator basic usage

In this section, you will find the basics you need to introduce the wireframe simulator into your own program. One example in ViSP presents the main capabilities of the simulator. You can find its source code in the example folder which comes with the ViSP source code. There are also three demonstrations which present three visual servoing applications using the simulator. You can find there source codes in the demo folder. It is advised to execute the example and the demonstrations to see the simulator behavior and capabilities.

3.1 How to initialise the simulator

The wireframe simulator is programmed as a C++ class in ViSP and named vpWireFrameSimulator. Thus, the first thing you have to do is to declare an instance of the class. As it was decided to separate the simulator part from the display part you will have to declare several images vpImage<vpRGBa> or vpImage<unsigned char>. The advantage of this strategy is that the simulator is compatible with all the display in ViSP.

```
vpImage<vpRGBa> Iint(480,640,255); //The main camera which provides the internal view
vpImage<vpRGBa> Iext1(480,640,255);  //The main external camera
                                     //which provides an external view of the scene.

vpWireFrameSimulator sim;
```

The simulator projects the different elements of the scene into the views of the cameras. In the main camera, the object at the current and the desired positions is displayed. Depending on the function you use to initialize the scene, you can decide to change the object used to display the desired position or decide to avoid the desired position display. In the external cameras, the object at the current position and the main camera are displayed. For the moment, the file format used to load the 3D model of the object is a specific format called ".bnd". You can find already defined objects in the data folder located in the ViSP build directory (VISP_BUILD_DIR/data/wireframe-simulator).

Thus, you have now to initialise the scene you want to display. If you want to display the desired position in the main camera view, you have to use the `initScene()` method as shown here.

```
sim.initScene(vpWireFrameSimulator::PLATE, vpWireFrameSimulator::D_STANDARD);
```

The first argument corresponds to the 3D model used to display the current position of the object. And the second argument corresponds to the 3D model used to represents the desired position.

If you do not want to display the desired position of the object, you have to use the same method as shown here.

```
sim.initScene(vpWireFrameSimulator::PLATE);
```

As said before, it exists several predefined objects whose list is given in Table 1.

Shapes	Data files	Description (considered units are meters)
THREE_PTS	3pts.bnd	Three points (radius 0.02) on a plate (0.4x0.4x0.01)
CUBE	cube.bnd	A simple cube (0.125x0.125x0.125)
PLATE	plate.bnd	Four points (radius 0.02m) on a plate (0.4x0.4x0.01)
SMALL_PLATE	plate_6cm.bnd	Four points (radius 0.01m) which are separated by 0.06
RECTANGLE	rectangle.bnd	Four points (radius 0.02m) which are separated by 0.07m along x and 0.05m along y. They are on a plate (0.4x0.4x0.01)
SQUARE_10CM	square10cm.bnd	Four points (radius 0.02m) which are separated by 0.05. They are on a plate (0.4x0.4x0.01)
DIAMOND	diamond.bnd	Four points (radius 0.02) which are separated by 0.1. They are on a plate (0.4x0.4x0.01)
TRAPEZOID	trapezoid.bnd	Four points (radius 0.02) on a plate (0.4x0.4x0.01). Pt1 (-0.025,-0.05,0), Pt2 (-0.075,0.05,0), Pt3 (0.075,0.05,0), Pt4 (0.025,-0.05,0) from the center of the plate.
THREE_LINES	line.bnd	Three lines (200,0.01,0.01)
ROAD	road.bnd	A long road with poles along
TIRE	circles2.bnd	Two circles, one with a radius of 0.1 and one of 0.15
PIPE	pipe.bnd	A cylinder (radius 0.15 and length 0.25)
CIRCLE	circle.bnd	A circle (radius 0.1)
SPHERE	sphere.bnd	A sphere (radius 0.15)
CYLINDER	cylinder.bnd	A cylinder (radius 0.1 and length 0.8)
PLAN	plan.bnd	A plan (0.56x0.56x0.01) with several small squares (0.01x0.01x0.01)

Table 1: List of the available 3D objects.

Several methods enable to change the color of the current object, desired object and camera. By default the colors are blue, red and green respectively.

The next step consists in positionning the different cameras. The main camera is positionned relative to the object. Thus, you have to create two homogeneous matrix (${}^{c}M_{o}$ and ${}^{cd}M_{o}$ corresponding to the current and the desired pose of the main camera). The following code shows the methods you have to use.

```
vpHomogeneousMatrix cMo;  //The pose between the object and the main camera
vpHomogeneousMatrix cdMo; //The desired pose between the object and the main camera

//Set the initial and the desired position of the main camera.
sim.setCameraPosition(cMo) ;  // Position of the object in the camera frame
sim.setDesiredCameraPosition(cdMo);  //desired position of the object in the camera frame
```

The external cameras are positionned relative to the world reference frame. Thus, even if the object moves, the external camera don't move. As for the main camera you have to create an homogeneous matrix (${}^{cext}M_{f}$) and use the following method.

```
//The initial position of the main external camera
vpHomogeneousMatrix cextMf(0.0,0.0,1.0,vpMath::rad(0),vpMath::rad(0),vpMath::rad(0));

//Set the External camera position
sim.setExternalCameraPosition(cextMf);
```

To end the initialisation, you can set the camera parameters. All the external cameras have the same parameters. To define the camera parameters you have to use the vpCameraParameters class. In fact, all the

parameters are not taken into account. Indeed, only the p_x and p_y pixel ratio parameters are used. The u_0 and v_0 parameters which represent the optical center of the camera in the image are set in order to be in the center of the image. The parameters relative to the distorsion are not taken into account. The following lines of code show how to set the camera parameters.

```
1 //Set the parameters of the cameras (internal and external)
2 vpCameraParameters camera(1000,1000,320,240); //px,py,u0,v0
3 sim.setInternalCameraParameters(camera);
4 sim.setExternalCameraParameters(camera);
```

After this step the wireframe simulator is initialised.

3.2 How to get images from the simulator

After the initialisation stage described in section 3.1, you are able to get the views of the cameras and then to display them. The simulator does not modify directly the image. Indeed, it displays the different elements (object, main camera, ...) as overlays. There are three different methods to get the views. The first one which is `getInternalImage()`, enables to get the view of the main camera. The projection of the current view and the desired view of the object is made thanks to the parameter ${}^cM_o$ and ${}^{cd}M_o$ you set during the initialisation. As said before, it is possible to have several external cameras. Thus, there are two methods `getExternalImage()`, one for the main external camera and another to get the view of another camera which is positionned relative to the world reference frame. The following code explain how to use these methods.

```
1  //Get the view of the main camera.
2  sim.getInternalImage(Iint);
3  //Get the view of the main external camera.
4  sim.getExternalImage(Iext1);
5
6  //Set the pose of another external camera relative to the world reference frame
7  vpHomogeneousMatrix othercamMf(0.0,0.0,3.0,vpMath::rad(25),vpMath::rad(15),vpMath::rad(0));
8  //The other external camera image
9  vpImage<vpRGBa> Iext2(480,640,255);
10 //Get the view of the other external camera.
11 sim.getExternalImage(Iext2,othercamMf);
```

As you can see the positions of the main camera and the main external camera are stored in the simulator. For the other external camera, you give its position when you get the view. Thus, you can have as many external views as you want.

With all these simple methods, you are able to initialise and get the views of the cameras provided by the simulator. In the next section, the advanced fonctionalities are presented.

4 Advanced functionalities

In the previous section, you get the basics to initialise the simulator and display the view of the cameras. But it exists several other functionalities for an advanced use.

4.1 Changing the point of view of the main external camera with the mouse

As you saw before, it is possible to set the pose of the main external camera relative to the world reference frame thanks to the `setExternalCameraPosition()` method. You can use it not only during the initialisation. But it is also possible to move the main external camera thanks to the mouse pointer. To enable

this functionality, you have to put the `getExternalImage()` method corresponding to the main external camera in a loop as shown in the code below.

```
while (1)
{
  vpDisplay::display(Iext1);
  sim.getExternalImage(Iext1);
  vpDisplay::flush(Iext1);
}
```

Then, during the execution of the code, use one of the three buttons of the mouse and move the pointer to rotate, zoom or translate the camera. Figure 4 illustrates the use of this functionnality.

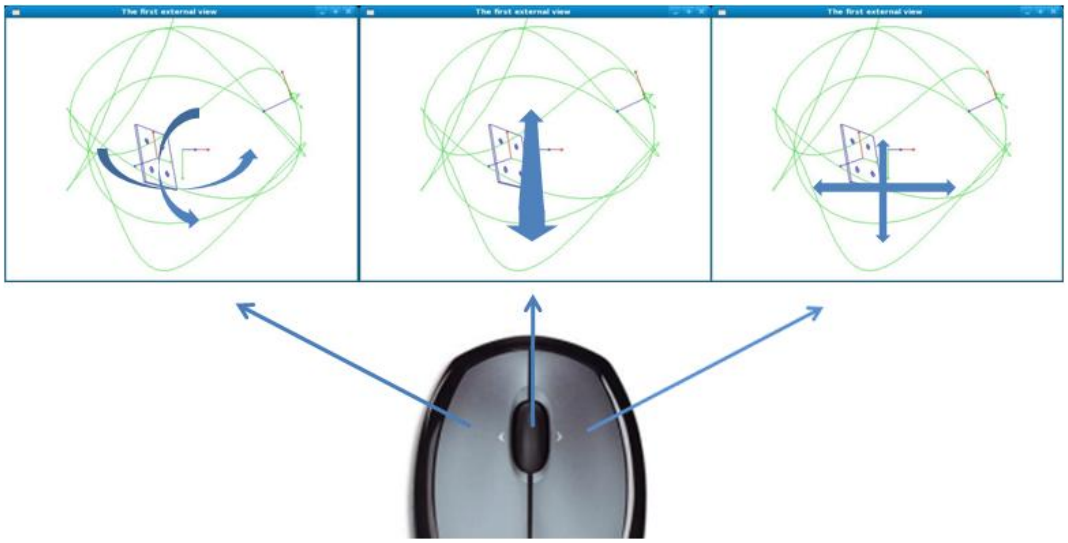

Figure 4: How to move the main external camera with the mouse.

The example and the demonstrations use this functionality. It is advised to execute them in order to understand how it works.

Moreover, the `getExternalCameraPosition()` method enables to get the pose of the main external camera relative to the world reference frame. Thus, during the first execution of your program, you can search the best position of the external camera thanks to the mouse. You get the position thanks to the method. And after you can initialise directly the position of the external camera.

4.2 Moving the object

In order to simulate a movement of the object, it is possible to set the position of the object relative to the world reference frame. At the begining, the object frame and the reference frame are at the same location. Thus, to move the object, you only have to create an homogeneous matrix (${}^{f}M_{o}$) which gives the position of the object relative to the world refrence frame. Then, use the `moveObject()` method to set the position of the object.

```
vpHomogeneousMatrix fMo(0.0,0.0,0.1,vpMath::rad(25),vpMath::rad(15),vpMath::rad(0));
sim.moveObject(fMo);
```

4.3 Displaying the main camera trajectory

As shown in the previous screenshots, it is possible to display the trajectory of the camera in the main external camera's view. This functionnality is by default. The `setDisplayCameraTrajectory()` method allows to disable it.

Each time you call the `getExternalCameraPosition()` corresponding to the main external camera, the pose of the main camera and the position of the object are saved. By default, only 1000 positions at maximum are stored. But this parameter can be changed thanks to the `setNbPtTrajectory()` method. You can also choose the color of the trajectory with the `setCameraTrajectoryColor()` method. And finally, it is possible to display the trajectory with lines or points thanks to the `setCameraTrajectoryDisplayType()`.

You can get the lists containing the main camera's positions and the object's positions thanks to the two methods : `get_cMo_History()` and `get_fMo_History()`. It can be practical if you want to display several trajectories in the same window or display the trajectory in an other external camera. Indeed, you can use the `displayTrajectory()` method to project the camera trajectory into the external camera view you want. The method requires an image, a list of main camera's positions (${}^cM_o$), a list of object position (${}^fM_o$) and the position of the world reference frame relative to the external camera (${}^cM_f$). Figure 5 shows an example of result which can be obtained.

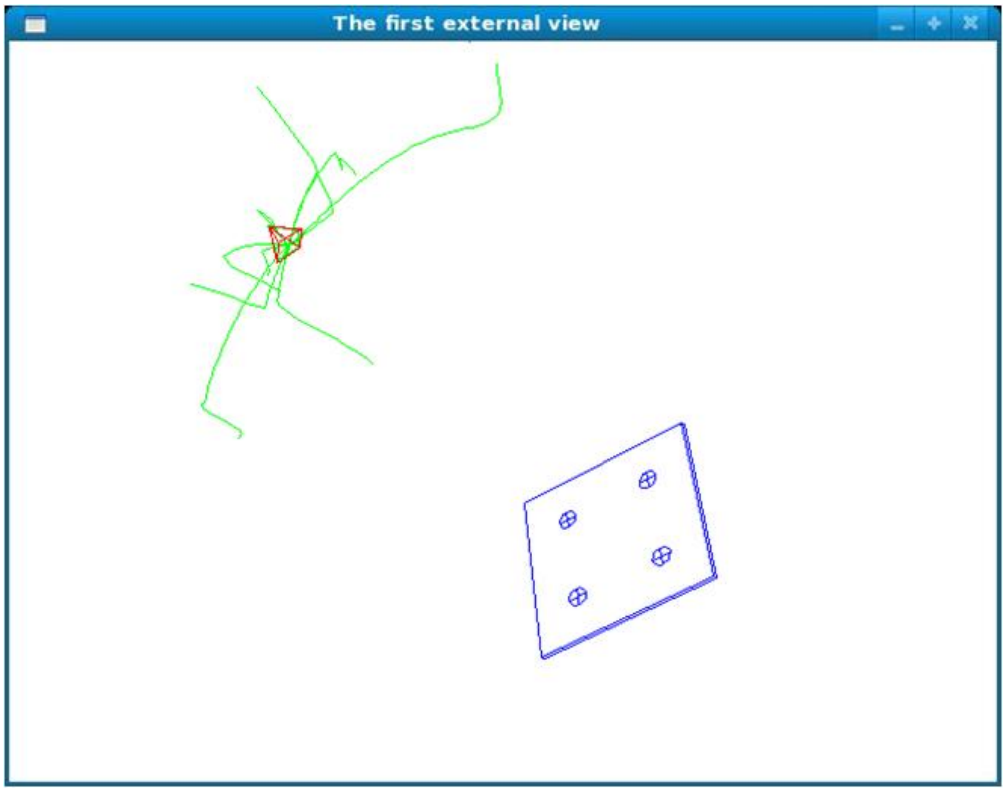

Figure 5: Example of several trajectory displayed in the same time..