Practical – Point-based virtual visual servoing

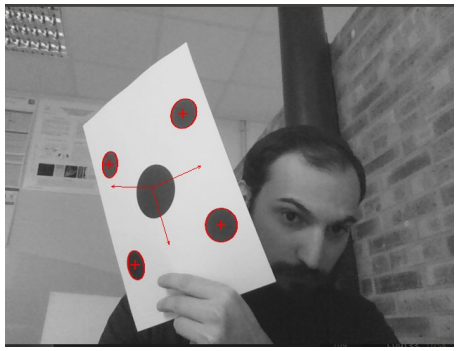
This practical aims to track an object, in images acquired by a camera, and to simultaneously estimate its pose, knowing a *3D model* of it. The tools that are going to be used are a digital camera (Kinect) for the hardware, and the C++ language, CMake and ViSP library for the software side. ViSP stands for Visual Servoing Plaform and comes with its documentation. This is a framework for vision-based control of a robot and visual tracking using several features. CMake is a C++ project configuration tool to be used with many IDEs (e.g. qtCreator).

To configure the practical programming project:
Run CMake, select the practical source and build directories and then configure and generate.

Point-based VVS is similar to the pose computation by optimization of an object from points. To do this without ambiguity, fours points need to be used.

Part 1: Image acquisition, dot detection and tracking

1. Image acquisition
   a. Image acquisition is done using a *MyFreenectDevice* object. After declaring it, open the device using the *open* method. Start to acquire an image using the *getVideo* method, taking a *vpImage* object as a parameter, still on the same object.
   b. Image display is done using a *vpDisplayX* object. Declare and instantiate it with four parameters: the image to display, the horizontal and vertical positions of the window and the name of the window. Then, use the *vpDisplay::display* static method to prepare the image display and the *vpDisplay::flush* method to flush the display and effectively display the image.
   c. Loop on these acquisition and display stages to display the camera video flow.



2. Dot detection and tracking. A dot is a white/black disk printed on a black/white paper sheet. The *vpDot2* object of ViSP detects and tracks a dot over images.
   a. To select the original position of the dot in the image, use the *initTracking* method of the *vpDot2* object. Set the display option of the dot detection result on the image using the *setGraphics* method, before calling *initTracking*.
   b. Use the dot tracking in the acquisition and display loop, calling the *track* method of *vpDot2*, to test its robustness with respect to motion and various orientations.

Part 2: Initial pose computation

1. Let us first create a *vpCameraParameters* object taking as parameters, the intrinsic parameters of the camera: α_u, α_v, u_0, v_0. This is needed for the virtual visual servoing process since the camera projection model is involved in the framework.
2. After points are selected, an initial pose ${}^{c}\mathbf{M}_{o}$ of the object, in the camera frame, has to be computed. A way is to linearly compute it using a Lagrangian method, for instance. Add the four points to a *vpPose* object (see the item list below to know how to

proceed) and compute the pose with its *computePose* method, with parameters *vpPose::LAGRANGE* and the ${}^{c}\mathbf{M}_{o}$ matrix. To create the set a four points to be used by *vpPose*, use the following procedure:

a. Create an array of four *vpPoint*
b. Set the world coordinates of each (the coordinates in the object frame)
c. Then, from the dots centers of gravity, expressed in the image frame, convert them to the normalized plane using the *vpPixelMeterConversion::convertPoint* static method. Then, set the normalized coordinates *x*, *y* of each *vpPoint*.
d. To finish, use the *addPoint* method of the *vpPose* object to effectively add every *vpPoint*.

3. Display the initial frame in red using the following method of the vpPose object: *display(I, cMo, cam, 0.05, vpColor::red)*.

Part 3: Point-based virtual visual servoing

1. Define a *vpRobotCamera* object. This is like a virtual free flying robot. Set its initial pose with the last computed pose using its *setPosition* method.
2. Definition of the task. The task is a *vpServo* object, and needs to be configured to move the virtual robot.
   a. The *setServo* method of the task has to be used to set the servoing frame: the camera frame itself, that is *vpServo::EYEINHAND_CAMERA*. Then, set the interaction matrix type, first, at *vpServo::CURRENT*, and computed by pseudo-inverse `vpServo::PSEUDO_INVERSE` .
   b. The visual features are points. Then, define four desired features *vpFeaturePoint*, of which *x* and *y* normalized coordinates are the one from the result of dots tracking, using the static *vpFeatureBuilder::create*. Since the current interaction matrix is used, the depth corresponding to desired points is meaningless.
   c. Knowing the points coordinates in the object frame (see the array of four *vpPoint*), the current set of features is the projection of the object four points in the image, for the current pose ${}^{c}\mathbf{M}_{o}$. So first apply a frame change on each *vpPoint* using the *changeFrame* method. Then, use the *projection* method to apply the perspective projection of a point and create or update the current feature with *vpFeatureBuilder::create*.
3. Update the features in a loop included in the dots tracking loop and use the task object to compute the control law. The obtained 6-vector **v** is the update vector for the pose. Then, update the *robot camera* pose with the setVelocity method, taking **v** as parameter. Finally, get the new ${}^{c}\mathbf{M}_{o}$ from the *robot camera* using the *getPosition* method and loop.
4. Display the object frame in images with *vpPose::displayFrame*, previously used.