Advanced Visual Geometry

Lab 1: Homography-based visual odometry

1 Content of this lab

The goal of this lab is to perform Augmented Reality (AR) from a sequence of images.
The only information that we have is that the camera is observing a mostly planar environment, where we want to display the 3D model of a robot (AR), see Fig. 1 for an overview.

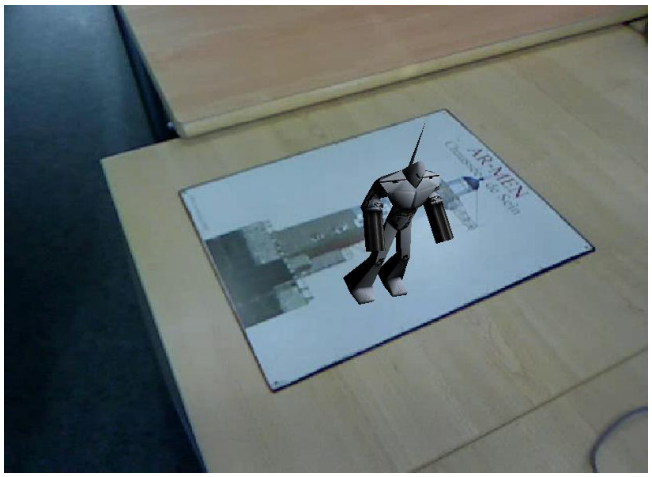

Figure 1: Current image with 3D robot AR display

AR needs a 3D transform between the camera and the world frame. By doing homography-based visual odometry (VO), we can do this pose estimation with regards to a reference 3D plane.
The goal is thus to load the video file, and go through the images with all the necessary steps to perform AR. Two libraries will be used to do so:

- OpenCV 3 to read images from files or camera, perform low-level image processing[1]. We use version 3 as it includes the homography matrix decomposition.
- ViSP to manipulate 3D points, camera pose and use vectors and matrices[2]

The actual classes that are used are detailed in Appendix A.
This lab is available on GitHub. The package can be downloaded through git and compiled with the sequence:

```
git clone https://github.com/oKermorgant/ecn_visualodom.git
cd ecn_visualodom
mkdir build
cd build
cmake ..
make
```

[1]Open source Computer Vision, http://opencv.org
[2]Visual Servoing Platform, http://visp.inria.fr

The video should be downloaded from my website in the lab directory:

```
wget http://pagesperso.ls2n.fr/~kermorgant-o/files/armen.mp4
```

It may be useful to use an IDE like Qt Creator, to do so you have to launch it from the command line then:

- Open project and load the `CMakeLists.txt` file
- Set the `build` folder as the build folder

1.1 Structure of the `ecn_visualodom` folder

The folder has the classical structure of a C++ project:

```
ecn_visualodom
├── include
│   └── ecn_visualodom
│       ├── ar_display.h ..... augmented reality code
│       ├── video_loop.h ..... load and read video file
│       └── visual_odom.h ..... VO class header
├── src
│   └── visual_odom.cpp ..... VO class to be modified
├── armen.mp4 ..... video file
├── CMakeLists.txt ..... CMake file
└── vo.cpp ..... main file to be modified
```

Figure 2: Files used by the package

The only files to be modified are `visual_odom.cpp` that is the visual odometry class and performs visual odometry between two consecutive frames, and `vo.cpp` that is the main C++ file and uses visual odometry to estimate the current pose of the camera with regards to the world frame.

1.2 Expected work

The goal is to implement an homography-based visual odometry and test it on the `armen.mp4` video. The main steps are already written in the code as comment sections. The `VisualOdom` class will be doing most of the job, except for the initial pose estimation. Dedicated OpenCV of ViSP classes or functions will also do most of the job and the goal of the lab is also to get used to them.
The next section details the homography-based visual odometry algorithm and its most important steps.

2 Homography-based visual odometry

The basis of the visual odometry from homography is to compute an homography from two views of the same plane. If the equation of the plane is written $\mathbf{X}^T\mathbf{n} = d$ in the first camera frame, then the homography matrix is $\mathbf{H} = \mathbf{R} + \mathbf{t}\mathbf{n}^T/d$ where $(\mathbf{t}, \mathbf{R})$ express the transform from camera 1 to camera 2 as shown in Fig. 2.

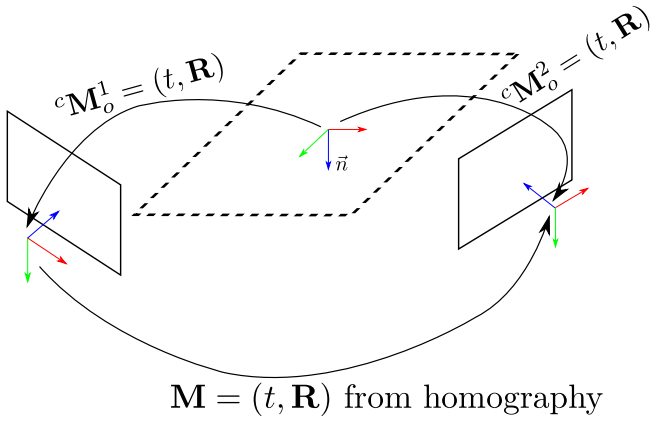

Figure 3: Two views of a given 3D plane links 3 transforms.

However, in order to display any 3D model with AR, an initial pose ${}^c\mathbf{M}_o$ has to be estimated for the initial camera. We first recall the relative odometry before giving clues about the initial pose estimation.

2.1 Relative odometry from homography

The core algorithm is to compute the relative odometry, that is the transform between the two camera frames. It corresponds to the method `VisualOdom::process`. The algorithm to follow is:

1. Convert image to gray level
2. Detect and compute keypoint descriptors in the new image
3. Match keypoints with the ones from the previous image
4. With RANSAC, compute homography and also get matches inliners
5. Decompose homography to get $(\mathbf{t}, \mathbf{R}, \mathbf{n})$ candidates
   A method in OpenCV will give us all candidates (usually 4) under the form of 3 vectors of dimension 4: translations, rotations and normal to the plane. Candidates are stored under a `vector` of `Homography`, a custom structure used to regroup $(\mathbf{t}, \mathbf{R}, \mathbf{n})$ (see Section A.2).
6. Prune candidates that give a negative Z for at least 1 point
   The corresponding loop is already in the code but the condition to consider a candidate to be valid or not has to be defined.

7. There still may be two candidates, keep the one that has the normal $\mathbf{n}$ closer to the estimated normal

The last step shows that a guess for the normal has to be given in order to find the best candidate homography. This can be done with the `VisualOdom::setNormalGuess` in the main code (around line 24). In our case we assume we observe a horizontal plane with a camera also placed horizontally. This is sufficient to have a corse estimate of the normal to the plane in the camera frame.
At the end of the whole sequence, we have a homography decomposition $(\mathbf{t}, \mathbf{R}, \mathbf{n})$ without the scale factor. Actually OpenCV gives us $\mathbf{t}/d$ as $\mathbf{t}$ as they cannot be separated without extra information. If we have an estimation of the value d, we can easily compute the true translation with $d.\mathbf{t}$. This d estimation is stored in the `VisualOdom` class as `d_guess`.
The final step is to prepare for the next image: the parameters $(d, \mathbf{n})$ of the plane equation have to be rewritten in the frame of camera 2.

1. Updating $\mathbf{n}$ is only a matter of frame change as we have $\mathbf{R}$.
2. Updating d can be done with only one point, so in practice we compute the new d from all the points and then take the mean value.

We now detail how to get an initial guess for d.

2.2 Initial plane parameter from assumed translation

The scale factor is given by the size of the 3D model to be displayed. In practice this amounts to fix the initial translation ${}^c\mathbf{t}_o = (-0.1, -0.1, 0.7)$. This translation can be used to have a first estimation of the plane parameter d after the first homography has been computed. Indeed, in this case we have the normal to the plane and we can just set $d = \mathbf{n}^T {}^c\mathbf{t}_o$. This value can then be used to rescale the relative transform that was extracted from the homography. In the code, this corresponds to the line `if(d_guess == 0)`, where we do not yet have an estimate on d.

2.3 Overall algorithm

By combining the relative transform from homography and the initial guess from known translation, after the second image we can have the whole information:

1. Get unscaled transform from initial homography, this gives us $(\mathbf{t}, \mathbf{R})$ between the cameras and the normal $\mathbf{n}$ to the plane in the first camera frame.
2. This transform can be scaled from the known initial translation ${}^c\mathbf{t}_o$
3. Get the rotation matrix corresponding to the Z-axis of the plane aligning with the $\mathbf{n}$ axis of the camera frame (`VisualOdom::getRotationFromNormal`).
4. Build the full transform ${}^c\mathbf{M}_o(0)$ for the first image (variable `cMo0`)

After this initialization, the scale factor will be updated at each iteration from the previous value and the current camera pose just has to be updated from the previous one and the relative transform found by the homography.
In practice, we may experience a scale drift and perform all homographies with regards to the initial image.

A Main classes and tools

A.1 ViSP classes

This library includes many tools for linear algebra, especially for 3D transformations. The documentation is found here: `http://visp-doc.inria.fr/doxygen/visp-daily/classes.html`.
The main classes from ViSP (at least in this lab) are:

- `vpMatrix` represents a classical matrix, can then be transposed, inversed (or pseudo-inversed), multiplied with a vector, etc.
- `vpColVector` is a column vector with classical mathematical properties.
- `vpHomogeneousMatrix` is a 4×4 3d transform matrix. They can be multiplied to compose transforms.

These class can be declared and used as follows:

```
vpMatrix M(2,6);          // matrix of dim. 2x6
M[0][0] = M[1][1] = 1;    // element-wise assignment
vpColVector v(6);         // column vector of dim. 6
v[0] = 1;                 // element-wise assignment
M*v;                      // matrix-vector multiplication
```

A.2 The `Homography` structure

This structure is a simple object that regroups values for $(\mathbf{t}, \mathbf{R}, \mathbf{n})$. From an instance `H`, they can be accessed with `H.t`, `H.R` and `H.n`. `t` and `n` are `vpColVector` while `R` is a `vpRotationMatrix`.

A.3 The `VisualOdom` class

This class does most of the job (at least when you add the correct missing code). It is initialized from the camera parameters and a flag telling if the homography should be computed relative to the initial image or relative to the previous image (inducing scale drift).
The existing methods are:

- `bool process(cv::Mat &, vpHomogeneousMatrix &M)`
  Main method, processes a new image and returns the corresponding transform. Returns True if everything went fine.
- `void getRotationFromNormal(vpRotationMatrix &)`: give the rotation matrix between the plane frame and the camera frame. Can only be used if at least one homography was computed.

The current estimate of the homography plane is defined from `n_guess` and `d_guess`.
Most of the variables are already defined, the attributes of the class can be read in `visual_odom.h`.
In particular, these useful variables are already available as class attributes:

- Stored image `im1`, image that can be used for gray-level conversion `img`

- Keypoints (`kp1, kp2`) and their descriptors (`des1, des2`)
- Keypoint detector and descriptor: `akaze`
- Matcher and matches (`matcher, matches`)
- Calibration matrices: `Kcv` for OpenCV functions, `K` and `Ki` (inverse) for ViSP matrix multiplication
- Homography matrix (`Hp`, OpenCV format) and RANSAC mask for inliers (`mask`)
- Vector of homography structures `H` used to analyze the candidates and prune them if needed.