Visual Odometry Pipeline for Augmented Reality

Krzysztof Lis

January 8, 2017

1 Introduction

The goal of our program is to track the camera's pose using visual odometry methods and display a virtual object in a stable position in the real world. Our intention is to integrate this method into the open source Augmented Unreality plugin [3] and make it available for AR application developers. This use case is associated with the following requirements:

- Proper scale and orientation is needed to show the virtual object in the correct place.
- There are no constraints on the camera pose.
- The algorithm must run on-line and in real time.

2 Algorithm

The program performs the following steps in each iteration.

2.1 Initialization phase: ArUco pose estimation

In the initialization phase, when no 3D landmarks are known yet, we determine the camera pose using ArUco[1] fiducial markers and their detection method implemented in in OpenCV[4]. This pose is used to triangulate the first set of 3D landmarks. When at least $kpt_{init} = 24$ landmarks are created, the initialization is over and the fiducial marker tracker is deactivated.

This approach requires the fiducial marker pattern to be visible during initialization. However, it provides the correct scale and orientation, which is important in the augmented reality use case as the user expects to see the virtual object of a reasonable size and properly oriented (for instance standing on the table). (function `ProcessFrameAruco` in `AURTracker.cpp`)

2.2 Keypoint tracking

All keypoints (candidates and landmarks) are tracked from frame to frame using the Lucas-Kanade optical flow algorithm. If a point goes out of view (reported position is outside of the frame borders), it is discarded. (function `StepTracking` in `AURTracker.cpp`)

2.3 Pose estimation using landmarks

The visible landmarks with known 3D positions are used to estimate camera pose. We use the P3P algorithm to find the potential camera poses and RANSAC to find the pose supported by the biggest number of keypoints. (function `StepPoseEstimation` in `AURTracker.cpp`)

2.4 Removal of outliers

In this step we try to eliminate landmarks whose 3D position is incorrect. For each keypoint, we store a *reputation* value, initially equal to $rep_{max} = 50$. In each frame, if the point is labeled as an inlier by RANSAC, its reputation increases by 1, up to rep_{max}, while the reputation of outliers is reduced by 1. If it reaches 0, the keypoint is removed. (function `StepPoseEstimation` in `AURTracker.cpp`)

2.5 Candidate point triangulation

Candidate points are triangulated when the distance between the current camera pose and the camera pose at the time when the point was first detected, exceeds $d_{triang} = 30\text{cm}$. We have observed that triangulating landmarks earlier yields less accurate pose estimation. The 3D position of the point is determined using triangulation between the current frame and the frame where the point was first detected. Afterwards, the point is added to the landmark pool. (function `StepTriangulationAttempt` in `AURTracker.cpp`)

2.6 Generation of new keypoints

New points are generated if at least one of the following conditions is met:

- There are fewer than $kpt_{min} = 64$ currently tracked keypoints.
- The existing keypoints are not distributed around the center of the image. To check that, we calculate the vector from the center of the image to the center of mass of all tracked keypoints, divide its coordinates by image width and height respectively, and check whether its length exceeds a threshold:

  $$((\frac{1}{w}\frac{1}{N}\sum_i p_x) - 0.5)^2 + ((\frac{1}{h}\frac{1}{N}\sum_i p_h) - 0.5)^2 > thr^2_{offcenter},$$

  where: w - image width, h - image height, $thr_{offcenter} = 0.2$ - distance threshold.

  (function `ShouldCreateNewPoints` in `AURTracker.cpp`)

Potential points to track are found using the Harris corner detector. Corners which are closer than $kpt_{dist} = 64$ pixels away from other corners or currently tracked keypoints are discarded. The remaining corners are added to the candidate keypoint pool and their initial positions in the image as well as the camera pose at the time they were detected, are stored. This initial camera pose is determined using either ArUco markers in the initialization phase or landmarks in operation phase. (function `StepKeypointGeneration` in `AURTracker.cpp`)

2.7 Augmented reality

The estimated camera pose is used to draw a virtual object on the video frame. In the example program, that is a cube, which - if the tracking is correct - should stay in the point $(0, 0, 0)$, which is in the center of the fiducial marker board. The cube should allow the user to see if the camera pose is correct.

3 Installation

The program is implemented using C++ and OpenCV. For convenience, the OpenCV binaries are included with the project. To compile or run the program, please copy the libraries from `./opencv/lib` to `/usr/local/lib`.

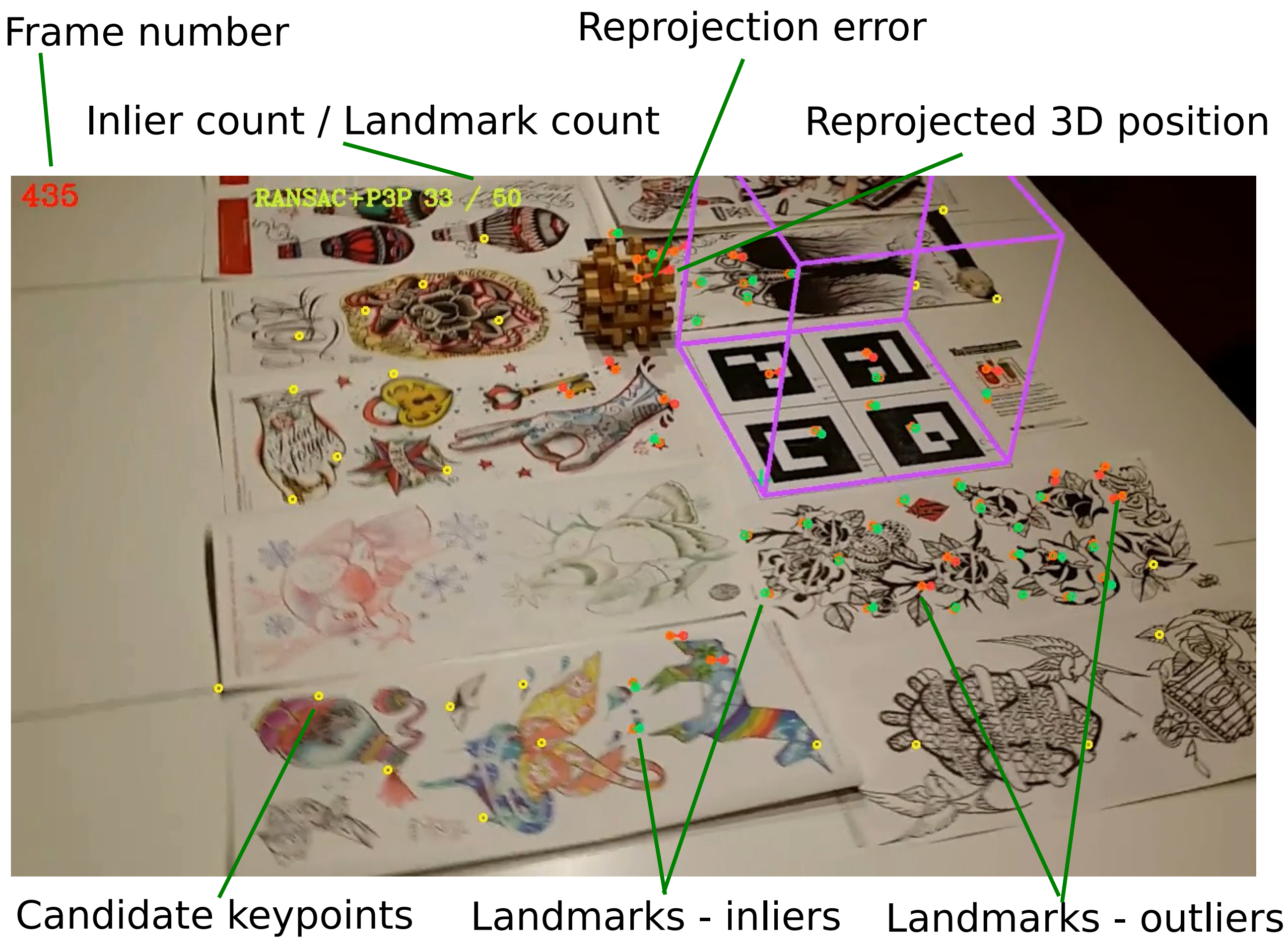

Figure 1: View of the example program. Markings: candidate keypoints - yellow dots, landmark inliers - green dots, landmark outliers - orange dots, reprojected 3D positions of landmarks - red dots, reprojection error - red lines. The cube is reprojected using the current camera pose and should be standing on the outer margin of the square containing fiducial markers.

The code can is built using CMake:

```
cd build
cmake ..
make -j8
```

The program reads a config file which contains the name of the input video and camera calibration file. There are two example inputs available. To run the program, execute:

```
build/aur_odometry default.xml
```

or

```
build/aur_odometry table.xml
```

4 Example videos

Example output videos are in `exaple_videos` directory. The notation used in the output video is explained in Figure 1.

References

[1] S. Garrido-Jurado, R. Mu noz Salinas, F.J. Madrid-Cuevas, and M.J. Marín-Jiménez. Automatic generation and detection of highly reliable fiducial markers under occlusion. *Pattern Recognition*, 47(6):2280 – 2292, 2014.

[2] Krzysztof Lis. Augmented Unreality - augmented reality plugin for Unreal Engine 4. `https://github.com/adynathos/AugmentedUnreality`.

[3] Krzysztof Lis. Quadrotor pilot training using augmented reality. `https://adynathos.net/projects/drone-pilot/DroneTrainingAR.pdf`.

[4] OpenCV - Detection of ArUco Markers. `http://docs.opencv.org/3.1.0/d5/dae/tutorial_aruco_detection.html`.