Improving feature tracking using motion sensors

Jean-Baptiste Boin
Stanford University
jbboin@stanford.edu

Abstract

A reliable and fast tracking algorithm is the core of a mobile augmented reality application. However, being on a computing power limited platform adds constraints that can lead to reduced performance (e.g. frame rate). In this paper, we present a new method of fast tracking by using the motion sensors that are available in most mobile platforms sold nowadays. They allow us to estimate the position of features in consecutive frames, so that we get an increase of around 45% of frame rate in the tracking of a textured 2D target. Our method is also a proof of concept that shows that estimating feature descriptors at each frame is not a necessary step in feature tracking, and staying at an abstract level by using geometric relations can yield enough information to get accurate results.

1. Introduction

Porting state-of-the-art computer vision algorithms to mobile devices is a difficult, but desirable task. Indeed, some applications require mobility, like augmented reality. However, running an application in real time may be tricky given the limited resources available on a mobile device. This is why the algorithms may need to be downscaled or simplified.

A typical augmented reality pipeline is composed of object recognition, object localization and object tracking. In this project, we will be focusing on the third part, the tracking of a target in real-time. We will be focusing on feature-based tracking, and our goal will be to make it more efficient by using the motion sensors in the mobile device (gyroscopes, accelerometers, magnetometer). Our approach will be compared to a pure vision-based feature matching scenario, which will be our baseline. Since we want to be able to track our target in real-time, we need to run the tracking algorithm for each frame and it should complete before the next frame is acquired. This shows how achieving an efficient tracking is critical to the performance of an augmented reality application. If the baseline can be efficient enough for simple situations, it is possible that some optimizations are needed if we need to deal with more complex objects (e.g. several 2D targets or one 3D target). In that case, an increase in the performance would allow for more advanced applications. For this project, we will try to develop and evaluate such an improvement in simple situations.

2. Context of our work

2.1. Review of previous work

While we did a bibliographic search on the subject, we realized that using motion sensors to aid tracking is not something that has been extensively researched on. There could be mostly two explanations for that. Either the use of motion sensors does not improve considerably the performance of tracking a target, which would jeopardize our whole project, or there has not been sufficient time yet to have real improvements using this additional measurement. Indeed, if accelerometers and gyroscope have been around for some time, they were not as tightly integrated as they are now, and using them required to use external devices that needed to be calibrated with the camera. It is also possible that the increase of computing power in recent handheld devices could only recently allow us to run the algorithms that we used in this project fast enough, or that the motion sensors were still not reliable for such applications until recently. Whatever the reason is, most papers do not give a central role to these motion sensors in the tracking of an object.

For example, in [1], the authors used mostly a visual based tracker, with two different modes, direct or incremental, depending on whether they would compare the current frame to a reference image or to the previous frame, and the only use of the sensor tracker is to validate the values given by the visual tracking. In an older paper [2], high frequency data from the sensors is fused with the low frequency data given by a visual system using an extended Kalman filter, showing that the fusion improves the accuracy given by each sensor alone.

Wagner et al. [3] could get some interesting results for tracking natural images using a simplified version of SIFT (called PhonySIFT) or Ferns. Using a motion model that

predicts the motion of the next from the motion of the previous frames, they also develop a patch based method that uses normalized cross correlation to get the position of the new patch in the vicinity of where the model predicted it to be. Our contribution bears some similarities to this approach, except that we tried to run our algorithm at the abstract level of the keypoints and not going down to the pixel level after the keypoints are detected for a given frame.

SURFTrac developed by Ta et al. [4] tracks features by detecting higher-level keypoints (based on SURF), but without extracting descriptors. We will develop a similar approach here, but with lower-level keypoints based on ORB.

2.2. Main contributions

Unlike most projects presented for this class, this one is not an implementation of a computer vision algorithm written by other authors. Instead, we are trying to answer a more open-ended research question: can we efficiently track a target using a feature-based method and information given by the motion sensors of the device? Because of the limited time we had, we could not aim at implementing a perfect algorithm, and we preferred to give a proof of concept of what we could achieve. In this report, we will justify the different choices we made for the project and in the end analyze them in the light of the results obtained. We will also hint at the different improvements that could be brought to our current method to improve the results.

Our main contribution in this project is to make use of the motion sensors of the device to get a fast tracking algorithm. We also develop a way of tracking features without having to extract computationally expensive descriptors, making use of a priori estimates of how features will move between frames.

3. Technical solution

3.1. Summary

The main architecture of the system that we built is given in figure 1. It is made of two states. The first state, which also corresponds to our baseline, is used as a bootstrapping step to precisely identify the position of our target. It compares each processed frame to the reference image of the target, contained in the database, and is relatively slow. This step is developped in the next sub-section.

As an output of this first step we get a good guess of the homography that maps the reference image to our camera frame, which means that we know precisely where the outline of the target is in our image. The second step is the actual tracking. Our current model is iterative and in this step we find the best affine transformation that maps the target between two consecutive frames. If this transformation search fails, then we go back in the bootstrapping step. This second step is made of three distinct parts:

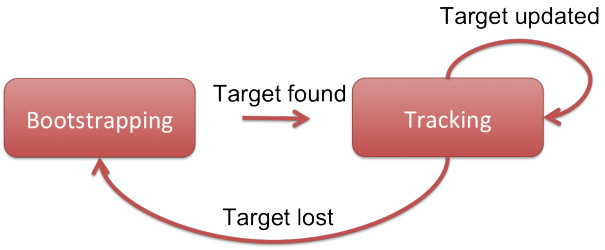

Figure 1. Architecture of the system.

- Estimation of the transformation of the keypoints from the previous frame, using the new rotation matrix given by the motion sensors
- ORB keypoint detection on the current frame
- RANSAC loop to find the best match between our estimates and the actual keypoints

We will elaborate on these parts in the following subsections.

At the end of this step, we can estimate the transformation (homography) between the previous frame and the current frame. This allows us to update the positions of the corners of the target.

If the previous step fails, which happens when we cannot properly map the keypoints between consecutive frames or if motion blur in one frame prevents the detection of enough keypoints, then we go back to the bootstrapping step to get a new clean estimate of the frame position.

3.2. Bootstrapping step (baseline)

This step is based on a typical image matching algorithm. At initialization of our application, we first extract 2000 ORB keypoints on our reference image as well as the corresponding descriptors. Then, at runtime, we extract 500 ORB keypoints and descriptors on the current frame. We then match each of the descriptors extracted to the reference descriptors, using the BFMatcher (brute-force) in OpenCV. After removing the matches with a high distance, we run a geometric verification with RANSAC. If this step still returns more than 20 geometrically consistent inliers, then we consider the bootstrapping successful and we can move to the tracking step.

The number of keypoints extracted was chosen empirically. We extracted more keypoints/descriptors in our reference image because this step is only done once, while we extracted fewer of them in each frame at runtime.

Figure 2 shows the output of this step. More explanation for the implementation is given in the code as well as in section 4.2.

A time analysis of an average cycle gave us more insight on the time-consuming steps in this baseline, and allowed

us to come to the conclusion that the descriptor extraction is one of the most consuming steps, which is why we removed it in our main tracking step. We will elaborate more on that in part 4.1.2.

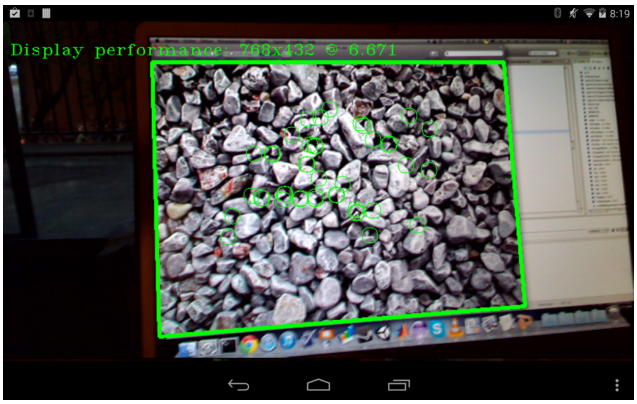

Figure 2. Example of the output of the bootstrapping step. The green circles are the features we extracted that match with the reference image of the target and that are geometrically consistent

We also call that step baseline because it is the basic algorithm that we are trying to improve through our tracking. This algorithm by itself could already give a very reliable tracking for a 2D target, but it would not be very fast, which is something we want to focus on for this project.

3.3. Tracking step

The main time gain in that step compared to the baseline comes from the fact that we do want to restrict the pixel operations as much as possible, and to stay at a higher level of abstraction, by dealing only with keypoints. This is why the ORB keypoint detector is the only pixel operation that will be done in that step.

3.3.1 Prior estimates of the previous keypoints using motion sensors

As presented in the Google Tech Talk "Sensor Fusion on Android Devices: A Revolution in Motion Processing"[5], the noise in the orientation angles and in the linear acceleration account for a very large drift once these accelerations are double-integrated, and so it makes it completely impossible to measure translation between two camera poses by using just the sensors of the device. However, it is possible to measure rotations with reasonable accuracy, and we have several tools available for this purpose.

The Android SDK allows us to access values from different sensors in the device, but it also offers interesting "composite" sensors that are based on values from these sensors.[6] These composite sensors are obtained from the values given by the raw sensors and processing them using Kalmann filters, commonly used in sensor fusion.

Among them is the "Rotation vector" sensor that uses accelerometer, gyroscope and magnetometer. It gives the rotation of the device with respect to a fixed coordinate system, East-North-Up. Our experiments showed us that the rotation values given by this sensor were already pretty accurate, and their real-time update reflected the motion of the device.

If the motion between two consecutive frames was a pure rotation and that the motion sensors gave us perfect measurements on the rotation matrix of our device, then we would be able to know exactly the position of each point of the first frame in the second frame. These points would be related by a homography, its matrix being given by:

$$H_{sens} = K R_{sens} K^{-1}$$

where R_{sens} is the relative rotation between the frames and K is the camera matrix. We found the values of K by calibrating the camera. Obviously, we had to remap the axes properly so that the axes of rotation matrix given by the motion sensors correspond to the actual coordinate system used in computer vision.

In practice, the relative rotation matrix will be slightly off, and the translation component of the motion will introduce an additional discrepancy with the previous model. But since we know that the target is planar, we are sure that the transformation of the target between the two frames should be a homography H_{real}. We can summarize this with the equation:

$$H_{real} = H_{error} H_{sens} \tag{1}$$

Given H_{sens} obtained with the sensor measurements, we can transform all the keypoints of the previous frame to get an initial estimate of the position of the new keypoints. Then, using the fact that the ORB detector applied on consecutive frames should return consistent keypoints (the overlap between the two sets of descriptors should be large), we want to find the homography H_{error} that maps the point cloud of the estimates to the point cloud of the real keypoints.

Since we decided to remove the descriptor extractor step, we can only match the keypoints by using geometric considerations, which is the next step of this algorithm.

3.4. Naive matching

As a first approximation, we assumed that the framerate was high enough to consider the error in the position estimate of the keypoints small enough.

In that case, it sounds logical to match each estimate with the nearest keypoint, as long the distance is smaller than a given threshold. If no keypoint is found nearby, it is safe to assume that this point is an outlier so we do not match it with any keypoint in the new frame.

Then, once our estimates are matched with real keypoints, we can estimate the best homography that matches the keypoints of the previous frame to these matching points. This is done using the OpenCV function `findHomography` that is based on RANSAC. If this method was usually good when there was almost no translation, as we can see with the diagram of figure 3. In this case, the estimated keypoints are close to the real ones, so the matches given by this method are the right ones.

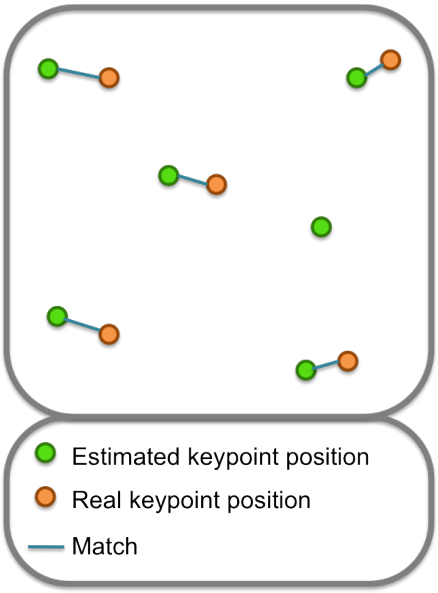

Figure 3. Example of a good case. The estimates are properly matched with the real keypoints in the new frame. One of them, which does not have a keypoint in its neighborhood, is considered as an outlier.

However, the most blatant failure cases happen as soon as the translation becomes non-negligible, especially in forward or backward motion. An example of pure forward motion is given in figure 4, with or without matches. The radial directions are also showed. In that case, the real keypoints have moved radially compared to the estimated keypoints (the estimation does not take into accounr the forward motion). Because of that, there can be some large discrepancies especially as we go further from the center, and it is very likely to match an estimate to a keypoint that is not the right one. In that case, we will tend to neglect the radial movement and this behavior was actually visible in our implementation. As soon as we moved towards the target or further from the target, the outline would stay more or less at the same size, as if there. This was not an acceptable behavior and it led us to try to find a new method to match the point clouds between frames.

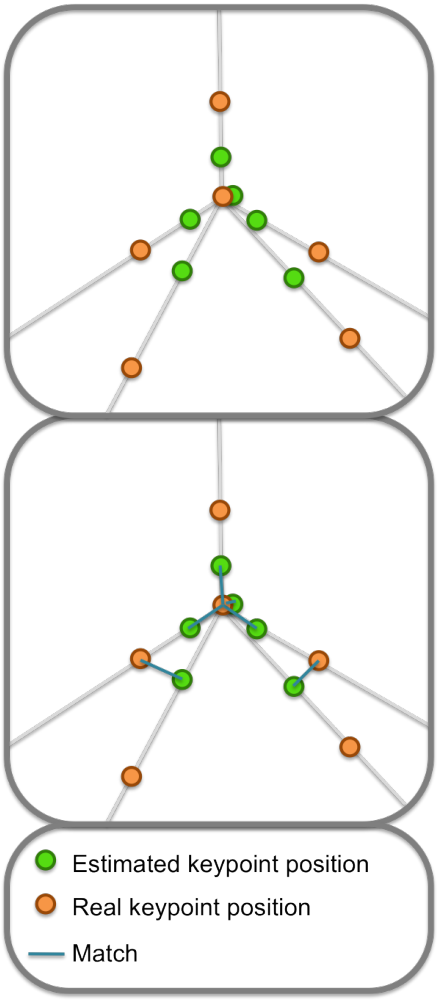

Figure 4. Example of a failure case. Between the two frames, the camera has gone forward, which means that the detected keypoints are further along the radial directions compared to their position on the previous frame. If we match each estimate with the nearest keypoint, this leads to wrong matches for almost all the pairs, except for the central keypoint. Clearly, this will yield a wrong homography between the frames.

3.5. RANSAC loop using an affine transformation model

This led us to try a new method for matching the point clouds. By saving the positions of keypoints and estimates in `.txt` files, we exported a typical sequence to Matlab so that we could experiment on how to find the best way to transform these keypoints. Typical techniques to find a transformation matching two point clouds are covered by iterative techniques of point set registration. Among those algorithms we tried the ICP algorithm (iterative closest point matching). An easy to use library implementing this algorithm, LIBICP, is given by [7]. However we did not get very good results with ICP because of the high number of out-

liers that we have. Indeed, ORB keypoints are not always that consistent in terms of response in different frames, so the top 500 keypoints in one frame may not correspond to the top 500 keypoints in the next one, which adds some difficulty to our problem.

In order to tackle that, we adopted a new RANSAC scheme. The observations made during the "naive matching" of the previous part (especially the failure cases) showed that it was important to consider the scaling part of the transformation to apply. By looking at data, it became apparent that the homography H_{error} of equation 1 could be quite accurately approximated by a translation, rotation and scaling. Adding shearings to this family of transformations, we get all the affine transformations. Our assumption here is that H_{error} will usually be well described by an affine transformation. In figure 5, you can see an example that shows the affine relation between two point clouds in consecutive frames.

The RANSAC loop will then do the following:

- We select 3 points in our estimates (an affine transformation is fully described by 3 points)
- We find the nearest current keypoints to each of these 3 estimates
- We find the affine transformation that maps the estimates to the keypoints
- We count the number of inliers by applying that affine transformation to all the estimates, counting a point as inlier if its transformation has a keypoint within a given radius.

Finally, the best transformation is kept and will be used as an approximation of H_{error}, which will then give us the full homography using H_{sens} and equation 1.

Since speed is our main concern here, it is important to see what will be our bottleneck in this algorithm. At each RANSAC loop, when we count the inliers, we will search the distance with the nearest keypoint for each of our estimates. If we use brute-force matching, considering that we have around the same number N of keypoints in both frame, which will usually be the case, this number being 500, then this requires in the order of N^2 operations. Considering that we will run the RANSAC loops many times, this is very likely to become a problem. But we can use the fact that the keypoints of the current frame will not change for all iterations of the RANSAC loop (although the estimates will change at each iteration) to make our search more efficient. This 2D framework is the perfect one for techniques like hashing, quadtrees or kd-trees that are commonly used for that. They usually save time for each query, going from N (brute-force) to $\log N$. Applied to each of our estimate, we end up doing $N \log N$ operations instead of N^2, which is

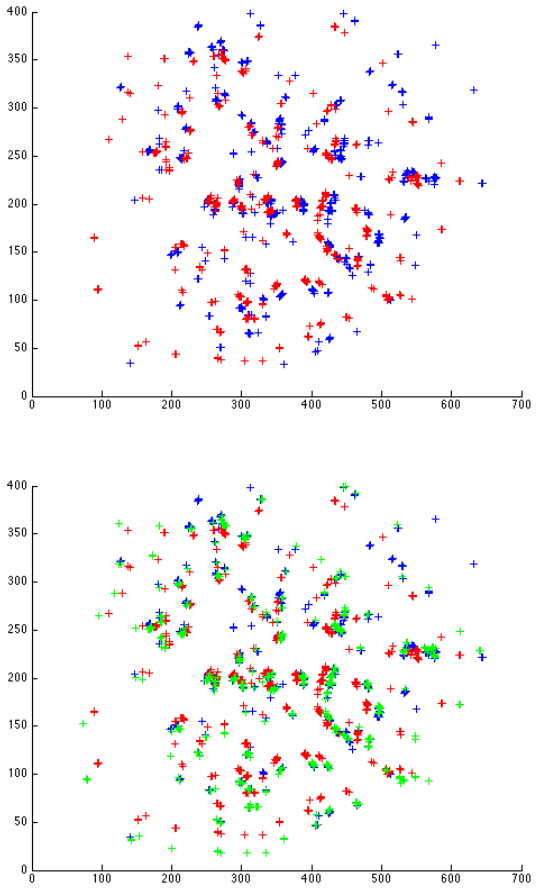

Figure 5. Example of affine transformation between two frames related by a forward motion (actual data). *Top.* The blue points correspond to the real keypoints while the red points correspond to the estimates. *Bottom.* Same as above, but we also added the green points, that correspond to the best affine transformation that maps the red points to the blue points. This single example gives some justification to what we choose to do: even though a few points are clear outliers, we can see the general trend: clusters of green points will correspond to clusters of blue points, which gives us a high number of inliers.

enough to considerably speed up the process. In our implementation, we used kd-trees, adapting the implementation on the Rosetta Code Wiki [8] to our case.

Even with the k-d tree, the number of loops needed to get a good estimate of the best affine transformation is still a bottleneck, and we cannot afford to count the inliers for many iterations. This is why we refine our RANSAC scheme by getting rid of transformations that are clearly not possible in our case. In general, the corrective affine transformation will be a rather small correction, so we will expect the shear and scaling factors to be respectively close to 0 and 1.

We can decompose an affine transformation matrix A as

follows.

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \end{bmatrix}$$

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} 1 & m \\ 0 & 1 \end{bmatrix} \begin{bmatrix} s & 0 \\ 0 & s \end{bmatrix}$$

$$\begin{bmatrix} A_{13} \\ A_{23} \end{bmatrix} = \begin{bmatrix} t_x \\ t_y \end{bmatrix}$$

We can then get the shear and scaling factors m and s from the matrix values with:

$$m = \frac{A_{12} + A_{21}}{A_{11}}$$

$$s = \sqrt{A_{11}^2 + A_{12}^2}$$

If we enforce the constraint that m should be close to 0, and s close to 1, then we can discard all the transformations that are clearly not possible. This process goes really fast, because it only deals with 3 points, so we can re-iterate many times until we get a good transformation candidate. With this constraint, it turns out that in most cases, the only transformations that remain will be very strong candidates, so we do not have to run many full RANSAC iterations before getting a good transformation. In order to gain some time, we only run 20 full iterations.

In some extreme cases, for example when the overlap between the two sets of keypoints is very small, or when one frame suffers from motion blur, it is possible that there are no good transformation candidates. In that case, we would loop forever as we would not be able to complete these 20 full iterations. This is why we also give a higher bound on the number of iterations (full or not) that we try. Once we reach that limit (set to 10,000), if we did not complete the 20 full iterations, then we consider that we are in a failure case and we start the bootstrapping again.

This method is not perfect but we managed to keep it quite fast, and it succeeds for most frames as will be discussed in the next section.

4. Implementation

4.1. Experiments and results

4.1.1 Quantitative and qualitative results

Because of the nature of this algorithm, we only evaluated the accuracy of the tracking qualitatively, while quantitative experiments were made to assess the performance (speed) of our algorithm compared to the baseline.

In terms of qualitative results, trying our prototype shows a few structural problems in the current state. If the baseline gives excellent result in terms of tracking, since it always refers to the original target image, our iterative approach can easily accumulate drift since it only refers to the previous frame. When most keypoints are the same between consecutive frames, which happens when we lock the camera towards the same area, or if we translate our camera slowly enough, then there are no failure cases and the drift is quite limited (although it is increasing in time). If we translate our camera quickly however, the affine transformation estimation can be quite bad and the position of the estimated outline changes suddenly (to a wrong value). These are corner cases of our implementation that would need to be dealt with in future versions. We will give hints in the conclusion at how we could remove some of these problems.

Evaluating the cycle time of processing a frame was a bit tricky because the processing time can vary a lot depending of what the camera is pointed at (especially the texture of the object). To simplify things, we decide to judge accurately a case where the camera was standing still in front of the target, with the target filling most of the screen. We stood still for some time and evaluated the average frame rate during this time. This gave us the following values:

- Baseline: frame rate of **6.8 FPS** (or cycle time of 147 ms) .
- Our approach: frame rate of **9.9 FPS** (or cycle time of 101 ms).

We get a decrease in the cycle time of 31%, which is to be compared to the analysis made in the next subsection.

Even though the tracking is still not perfect, it is still very satisfying to see that we can get a considerable improvement in the speed (increase of the frame rate by 45%). It is important to bear in mind that the current failure cases of our algorithm do not happen that often, given the frame rate at which we are running our application. Because of the iterative nature, even succeeding 95% of the time means that we will fail once every two seconds and never be able to recover after that. These issues will also need to be tackled, outside of the scope of this project.

4.1.2 Justification of choices and time gains

This subsection serves as a justification of our choices for the algorithm, mostly the fact that we only wanted to run one operation at pixel level (keypoint detection). By setting up timers that we start and stop at fixed points in our code, we could break down the proportion of time used for each stage in a typical cycle of the baseline algorithm (bootstrapping step). The values in figure 6 were taken by pointing at the target in a fixed position during around 30 seconds, which, at 6 frame per second, gives an average on around 180 similar frames. This can be considered as enough to get a good idea of the time used for each stage of the bootstrapping step.

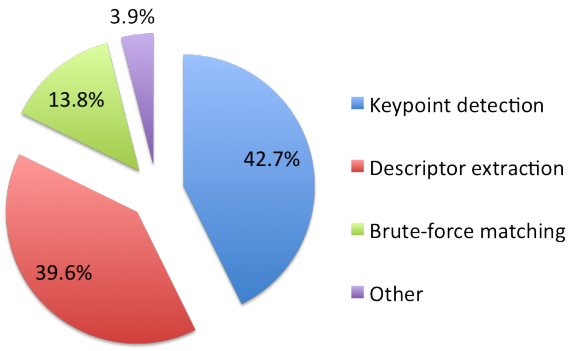

Figure 6. Breakdown of the time needed for each stage for our baseline.

There can be some variations in different frames, and these variations can be quite large in extreme cases, for instance when there is motion blur and only a handful of keypoints are detected. In that case, it is obvious that the keypoint detection is faster, and at an even larger scale the descriptor extraction and matching are also much faster. However, in most cases when this application would be used, these values are quite representative.

The first idea that we had when we first thought about how we could use the estimates that we had on the keypoints with the motion sensors was to speed up the matching stage by only trying to match estimates to the keypoints in the neighborhood. This could considerably decrease the matching time, especially since we ran a brute-force matching as a baseline, but even if we could almost make it instantaneous, we would not be able to get a decrease in the cycle time of more than 15%, which would not really speed up our tracking much. Since we are doing a feature tracking, it seems that a keypoint detector was definitely necessary on the whole image so that we could get the positions of new features. By elimination, the only step that could be removed was the descriptor extraction, which is what we decided to do.

It is interesting to note that our final algorithm has a cycle time of 70% the one of the baseline. This means that the parts we removed (descriptor extraction and matching) and that accounted for around 45% of the cycle time were replaced by a transformation estimation algorithm that takes about 15% of that cycle time, which divides this step by 3 and accounts for our improvement in speed.

4.2. Disclaimer and instructions for running the prototype

Our algorithm makes the assumption that the camera is **calibrated**. In our case, we ran a calibrating algorithm and hardcoded the values we got on our Tegra Note. If this algorithm is tried on another kind of device, it will probably fail because the camera parameters are likely to be quite different. This prototype should be run on another Tegra Note device, where the camera parameters are probably quite similar to the ones we developed on. It could be interesting to see how the algorithm performs when the values of the camera parameters are slightly different, but we did not study the influence of this parameter. Since these values are only used to get an initial estimate of the position of the new features using the motion sensors, and the following steps are used to refine this estimation using the positions of the new features, we would expect that having estimates that are slightly off would not matter much since our algorithm handles imperfect estimates. Still, this claim was not tested so we cannot be sure of that.

When the application is started, it is in bootstrapping mode. If it is pointed at the target (file included with the report), then a green outline should appear at the borders of this target. Green circles corresponding to the features that are properly matched should be seen briefly in this outline. We then enter the second step. In this step, the outline is constantly updated to match the new position of the target. The visual elements apart from this outline are:

- Red circles: positions of the features detected in the current frame
- Cyan circles: prior estimates of the new position of the features detected in the previous frame (using motion sensors purely)
- Yellow circles: final estimates of the new position of the features detected in the previous frame (related to the cyan circles by an affine transformation)

The cyan and yellow circles will usually be very close from each other since the error that we correct with the affine transformation is usually quite low, and the yellow circles should be also very close to a subset of the red circles. Errors usually happen when different features are detected in two consecutive frames, which can be avoided by moving slowly enough.

It is frequent to accumulate some drift, and in this case **touching the screen resets the system** so that we can bootstrap it again and get a new correct position of the outline. To test the baseline, one can either continuously tap on the screen or move his finger on the screen to continuously reset the. The “FPS” indicator is based on the durations of the operations for the last 25 frames processed, so we can consider its values stable in either mode (bootstrapping or tracking) after staying 25 frames in the same mode (a few seconds are enough).

It is important to note that the application runs almost completely in native mode. We first used the JNI but realized that there was some considerable overhead for each frame that decreased the frame rate. In the new version,

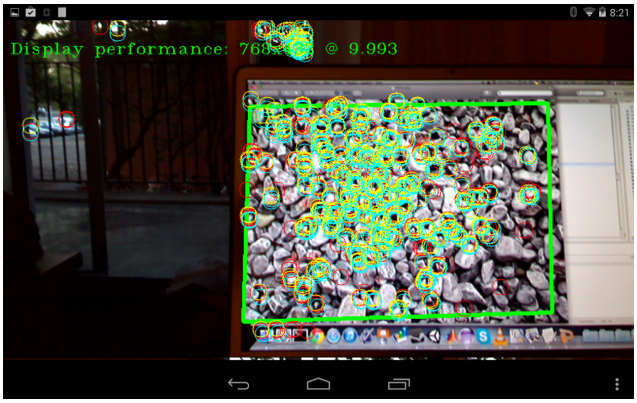

Figure 7. Screenshot of our application in tracking mode.

we only use the JNI to update the current rotation matrix. Indeed, the values of the rotation sensor, which is a virtual sensor given by the sensor fusion of magnetometer, accelerometer and gyroscope, are only accessible on the Java side. All the image processing is done in C++. The application does not run at full resolution, but at 768×432, which allowed us to get better frame rates at a limited cost in resolution loss.

5. Conclusion and future improvements

5.1. Next steps

As mentioned above, the main weaknesses of our target tracking algorithm is its iterative structure, and the fact that at its current state it fails too often to be actually used for AR applications.

Using multithreading, we could envision getting rid of the drift by running the baseline algorithm in parallel on a different core. In that case, one core could give us faster results subject to drift while the other could run a slower algorithm (baseline) that would not be subject to drift. In a more simple architecture, we could decide to run the baseline once every second for instance, which would not considerably slow down the average frame rate. With the values given above, running the baseline exactly once per second would still allow us to achieve an average frame rate of 9.45 FPS.

To tackle the second problem, the robustness of the tracking to fast translations, it seems important to make use of the observation that our algorithm currently fails when there is no (or very little) overlap between the set of keypoints of consecutive frames. On the other hand, we noticed that our 500 strongest keypoints are usually localized in the center of the frame, due to a vignetting effect. This explains why a fast translation in the central area of the image is likely to give a failure case. Basically, in our current algorithm, we only try to match the keypoints in a limited area to keypoints in another limited area, which is not safe if the translation is large enough. The best way to remove this effect would be to match to the keypoints of the current frame not only the estimates of keypoints of the previous frame (500 points that are localized), but instead the estimates of the keypoints of the full target image (2000 points that are scattered on the target). In that case, the tunnel vision effect given by the fact that we only evaluate a small part of the image would not be as much of a problem. Among the ideas that we have to increase our algorithm, this one sounds like the most promising and it is definitely the next direction that we will go towards in the future.

Other ways to make the RANSAC loop more efficient could be to include more heuristics to discard some affine transformations early enough. Some ideas include:

- using the scale and orientation of keypoints between two consecutive frames to see if the random estimates we select are likely to match with their nearest keypoint ; this should work if these values are consistent enough between consecutive frames
- using the accelerometer values to check consistent translation values

In that case, it could be interesting to also come back to the claim that H_{error} can be approximated as an affine transformation, and instead to implement a full homography transformation. Although it needs 3 matches instead of 2, if we have good enough heuristics to discard bad homographies early, we might be able to sustain this more complex model.

Finally, a last idea comes from the fact that we can afford to run only a few RANSAC loops. In that case, if a transformation is acceptable, the last resort would be to extract descriptors for the 3 tentative matches (6 descriptors) and see if they match. If they do, then we are pretty sure that the proposed transformation is a valid one. This operation may be costly but could still be interesting since it will be limited to very few points.

5.2. Last words

Although it was a small scale one, it was really interesting to implement a full tracking algorithm, from the recognition of the object to the actual positioning of the outline of the target on the viewfinder. This project required to learn many different aspects of the intricacies of developing in OpenCV and Android using the native code but the results turned out quite satisfying for its scope. We are planning to push this code further, starting by trying a few of the proposed improvements above, so that we could turn it into a full augmented reality application. Capabilities of the current mobile hardware are already quite stunning and will keep getting better, which is already by itself a very motivating thought given the applications that will be smartly making use of these.

6. Thanks

I would like to thank Roland Angst and David Chen for their support and precious suggestions along this project.

References

[1] S. Gammeter, A. Gassmann, L. Bossard, T. Quack, L. Van Gool, "Server-side object recognition and client-side object tracking for mobile augmented reality," *CVPR*, 2010.

[2] S. You, U. Neumann, "Fusion of Vision and Gyro Tracking for Robust Augmented Reality Registration," *Proceedings of IEEE VR2001*, pp. 71-78, 2001.

[3] D. Wagner, G. Reitmayr, A. Mulloni, T. Drummond, D. Schmalstieg, "Real-Time Detection and Tracking for Augmented Reality on Mobile Phones," *Visualization and Computer Graphics*, 2009.

[4] D.-N. Ta, W.-C. Chen, N. Gelfland, K. Pulli, "SURF-Trac: Efficient Tracking and Continuous Object Recognition using Local Feature Descriptors, " *CVPR*, 2009.

[5] D. Sachs, "Sensor Fusion on Android Devices: A Revolution in Motion Processing," 2010, accessible at http://youtu.be/C7JQ7Rpwn2k.

[6] Android SDK documentation, "Composite sensors" page, accessible at https://source.android.com/devices/sensors/composite_sensors.html.

[7] A. Geiger, P. Lenz, R. Urtasun, "Are we ready for Autonomous Driving? The KITTI Vision Benchmark Suite," *CVPR*, 2012 ; LIBICP code accessible at http://www.cvlibs.net/software/libicp/.

[8] Rosetta code, "K-d tree," page accessible at http://rosettacode.org/wiki/K-d_tree.