Marker Detection and Tracking for Augmented Reality Applications

Oliver Toole
School of Electrical Engineering
Stanford University
Email: toolebox@stanford.edu

Dave Dolben
School of Computer Science
Stanford University
Email: ddolben@stanford.edu

***Abstract*—This paper explores a simple method for detecting and tracking "fiducial" markers in a webcam video stream. The system first uses SIFT feature matching to detect when a marker is present in a frame of the video stream. Then, the detected keypoints are given to a KLT optical flow tracker, which tracks the keypoints frame-by-frame as they move through the video. In this paper, we will describe the method in detail, and present our results and analysis.**

I. Introduction

Detecting and tracking markers is a useful process in augmented reality. This technique gives augmented reality applications a simple way to to estimate the position and orientation, in 3-space, of an object in a video stream. From there, it is trivial to overlay 3-dimensional content to the video stream in real-time, in a way that makes it appear consistent with the scene.

This project explores one method of detecting and tracking these so-called "fiducial markers" in a webcam feed, using the OpenCV computer vision library. Our method first uses SIFT keypoint matching to detect the marker in the video stream. This gives us a set of keypoints in the video frame, and their corresponding locations on the marker. This is enough information to compute a homography between the clean image of the marker and the marker's location in the video frame.

After acquiring keypoints in the video frame, we use a KLT optical flow tracking algorithm to track the motion of these keypoints frame-to-frame. By maintaining the correspondence between our tracked keypoints and those on the clean marker image, we can compute a new homography for every frame. This allows us to track the orientation of the marker as it moves in the video.

II. Other Methods

Our method is by no means the only way of accomplishing this goal. One piece of example code discovered online uses a contour-based detection algorithm. It binarizes the image and computes contours using OpenCV, then checks to see if any of the contours are convex rectangles. From there, it computes the correlation coefficient between the normalized patch of the video frame and a clean marker image, and uses this as a similarity measure. If it is above a certain threshold, it is considered a match.

We will compare this method (using the sample code for a library called "ARma" discovered online at http://xanthippi.ceid.upatras.gr/people/evangelidis/arma/) to ours in a later section.

III. Detection

The detection step uses SIFT keypoint matching to find the marker in the video stream. We first pre-compute keypoints and descriptors for the clean marker image using OpenCV. Then, we compute keypoints and descriptors for the current video frame. We use OpenCV's built-in keypoint matching functionality to find matches between the sets of keypoints. Finally, we run OpenCV's implementation of RANSAC to find a homography between the two sets of keypoints. If the number of inliers (consistent with the homography) is above a threshold, we consider it a match.

This gives us a set of keypoints in the video frame that are located on the marker. It also gives us correspondences with keypoints in the clean marker image, which we keep in order to be able to compute a homography at each successive frame. Now that we have correspondences between keypoints, we can discard the SIFT descriptors, since they will no longer be needed. From here, our algorithm passes the keypoints and correspondences on to the tracker.

IV. Tracking

The bulk of this project was learning and re-implementing the KLT optical flow tracking algorithm. The goal of the tracking step is, given two frames from a video and a set of keypoints in the first frame, find the locations of those same keypoints in the second frame. Our system implements the algorithm as described in the three papers by Lucas, Kanade and Tomasi. We use OpenCV for its image and math functionality, but the core of the algorithm is our own implementation.

Let us formalize the problem. We have two (grayscale) images, I and J. $u = \begin{bmatrix} u_x \\ u_y \end{bmatrix}$ is a point in the first image. The goal is to find point $v = u + d$ on image J such that $I(u)$ and $J(v)$ are similar. As for the similarity measure itself, it's based on a window around each point, of size $(2w_x + 1) \times (2w_y + 1)$. We define the *residual function* ϵ as simply the squared difference between corresponding gray values in the two images, summed over the window around the point:

$$\epsilon(\mathbf{d}) = \sum_{x=u_x-w_x}^{u_x+w_x} \sum_{y=u_y-w_y}^{u_y+w_y} (I(x,y) - J(x+d_x, y+d_y))^2 \tag{1}$$

We want to find the point $\mathbf{v}$ in J that minimizes this error term.

Optical Flow

The core of the tracking algorithm is computing the optical flow of a point between two frames. We want to find the displacement vector $\mathbf{d}$ that minimizes the error term $\epsilon(\mathbf{d})$. The minimum of the error term occurs when its derivative with respect to $\mathbf{d}$ is zero, namely

$$\frac{\partial \epsilon(\mathbf{d}_{opt})}{\partial \mathbf{d}_{opt}} = [0 \quad 0] \tag{2}$$

We can find this point by taking the derivative of the expression for ϵ:

$$\begin{aligned}\frac{\partial \epsilon(\mathbf{d})}{\partial \mathbf{d}} = -2 \sum_{x=u_x-w_x}^{u_x+w_x} \sum_{y=u_y-w_y}^{u_y+w_y} (I(x,y) \\ - J(x+d_x, y+d_y)) \begin{bmatrix} \frac{\partial J}{\partial x} & \frac{\partial J}{\partial y} \end{bmatrix}\end{aligned} \tag{3}$$

If we use the first-order Taylor expansion for $J(x+d_x, y+d_y)$, we can simplify the equation:

$$\begin{aligned}\frac{\partial \epsilon(\mathbf{d})}{\partial \mathbf{d}} = -2 \sum_{x=u_x-w_x}^{u_x+w_x} \sum_{y=u_y-w_y}^{u_y+w_y} (I(x,y) \\ - J(x,y) - \begin{bmatrix} \frac{\partial J}{\partial x} & \frac{\partial J}{\partial y} \end{bmatrix} \mathbf{d}) \begin{bmatrix} \frac{\partial J}{\partial x} & \frac{\partial J}{\partial y} \end{bmatrix}\end{aligned} \tag{4}$$

Now, if we recognize that this expression basically consists of the image gradient $\nabla I = \begin{bmatrix} \frac{\partial J}{\partial x} & \frac{\partial J}{\partial y} \end{bmatrix}$ and the difference between the two frames $\delta I = I(x,y) - J(x,y)$, we can express this in a more clean notation:

$$\frac{\partial \epsilon(\mathbf{d})}{\partial \mathbf{d}} = -2 \sum_{x=u_x-w_x}^{u_x+w_x} \sum_{y=u_y-w_y}^{u_y+w_y} (\delta I - \nabla I^T \mathbf{d}) \nabla I^T \tag{5}$$

Rewriting the equation in a sightly different format, we arrive at

$$\frac{1}{2} \left[\frac{\partial \epsilon(\mathbf{d})}{\partial \mathbf{d}} \right]^T = \sum_{x=u_x-w_x}^{u_x+w_x} \sum_{y=u_y-w_y}^{u_y+w_y} ((\nabla I)^2 \mathbf{d} - \delta I \nabla I) \tag{6}$$

Finally, if we push the sum into the parentheses and let $G = \sum_x \sum_y (\nabla I)^2$ and $\mathbf{b} = \sum_x \sum_y \delta I \nabla I$, we get

$$\frac{1}{2} \left[\frac{\partial \epsilon(\mathbf{d})}{\partial \mathbf{d}} \right]^T = G\mathbf{d} - \mathbf{b} \tag{7}$$

Now, we are trying to minimize ϵ by finding the zero of its derivative, so essentially we are trying to solve

$$G\mathbf{d}_{opt} = \mathbf{b} \tag{8}$$

Solving this equation for $\mathbf{d}$ should give us the displacement vector that we want!

Iterative Optical Flow

Unfortunately, this algorithm, as written, only works when the pixel displacement $\mathbf{d}$ is very small. Ideally, we want to be able to solve for optical flow for larger pixel displacements in order to make this algorithm robust. We will address this in two ways: first, we will introduce an iterative version of the optical flow algorithm (as presented in Tomasi & Kanade 1991). Second, we will introduce an image pyramid approach (as presented in Suhr 2009).

In our iterative implementation of the optical flow algorithm, we simply start with an initial guess at the displacement vector, and iterate the above method until our guess converges. Formally, we use k as our iteration index, and let $\mathbf{d}^k$ be our k-th guess, with $\mathbf{d}^0 = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$. Now, at each step, we want to minimize the error term

$$\begin{aligned}\epsilon(\eta^k) = \sum_{x=u_x-w_x}^{u_x+w_x} \sum_{y=u_y-w_y}^{u_y+w_y} (I(x,y) \\ - J(x + d_x^{k-1} + \eta_x^k, y + d_y^{k-1} + \eta_y^k))^2\end{aligned} \tag{9}$$

To actually perform the iteration, we compute the intensity difference within the window

$$\delta I_k = I(x,y) - J(x + d_x^{k-1}, y + d_y^{k-1}) \tag{10}$$

the $\mathbf{b}_k$ term

$$\mathbf{b}_k = \sum_{x=u_x-w_x}^{u_x+w_x} \sum_{y=u_y-w_y}^{u_y+w_y} \delta I_k \nabla I \tag{11}$$

and the iterative displacement

$$\eta^k = G^{-1} \mathbf{b}_k \tag{12}$$

Then, we set $\mathbf{d}^k = \mathbf{d}^{k-1} + \eta^k$ and move on to the next iteration. We can break out of the iterative loop once η^k falls below a threshold, telling us that we have converged.

We only need to compute the image gradient ∇I and the matrix G once before the first iteration. Then we just have to compute δI_k (and thus $\mathbf{b}_k$) at each iteration in order to find η^k. Assuming that we have performed K iterations, the final displacement vector is

$$\mathbf{d} = \mathbf{d}^K = \sum_{k=1}^{K} \eta^k \tag{13}$$

Image Pyramids

In order to compensate for larger displacements that cannot be handled by the iterative KLT algorithm, we use an image pyramid as suggested in (Suhr 2009). For each frame, we compute a pyramid of progressively downsampled versions of the image. Let I^L be the image at the L-th level of the pyramid, with $I^0 = I$. At level L in the pyramid, the image is downsampled by a factor of 2^L. With this notation, a point $\mathbf{u}$ in the original image I corresponds to point $\mathbf{u}^L = \dfrac{\mathbf{u}}{2^L}$ in the pyramid image I^L.

We introduce a displacement guess $\mathbf{g}^L$ as our initial guess for the L-th level of the pyramid. Let $\mathbf{g}^m = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$. The displacement calculated at each level of the pyramid is $\mathbf{d}^L$. Starting with $L = m$, we calculate $\mathbf{d}^L$ at each level of the pyramid, using the iterative KLT algorithm with $\mathbf{g}^L$ as an initial guess. Then, we set

$$\mathbf{g}^{L-1} = 2(\mathbf{g}^L + \mathbf{d}^L) \tag{14}$$

and move on to the next level of the pyramid. Then, when we reach $L = 0$, our value for the net displacement is

$$\mathbf{d} = \sum_{L=0}^{m} 2^L d^L \tag{15}$$

We can use the same window size for all levels of the pyramid.

This method effectively calculates displacement vectors on increasingly high-resolution versions of the image, allowing us to calculate large-scale movement on the smaller versions of the image, and refine our guess on smaller scales as we move up the pyramid. This allows the algorithm to account for larger movements using almost the same code.

Parallel Execution

The biggest difference between our implementation of KLT tracking and OpenCV's implementation is the runtime. Originally, our code ran entirely sequentially. This was unbearably slow, running at around 1 frame per second (FPS). This made it unusable for real-time video. OpenCV, on the other hand, ran at around 22 FPS.

In an attempt to fix this, we modified our KLT tracking code to make use of OpenCV's built-in parallel execution architecture. Using OpenCV's "parallel_for_" construct, we were able to squeeze some extra speed out of our code, getting it to around 11 FPS, making it usable for real-time video.

V. Results

Our initial results were promising: we were able to replicate the functionality of OpenCV's implementation of the KLT tracking algorithm at framerates that were usable for live video. Figure 1 shows an example of one of the fiducial markers that we used for tracking, and Figure 2 shows a screen capture of the algorithm tracking points on a marker (the red box represents the calculated boundary of the marker). However, we experienced issues that arose in both our and OpenCV's implementations of the tracker algorithms.

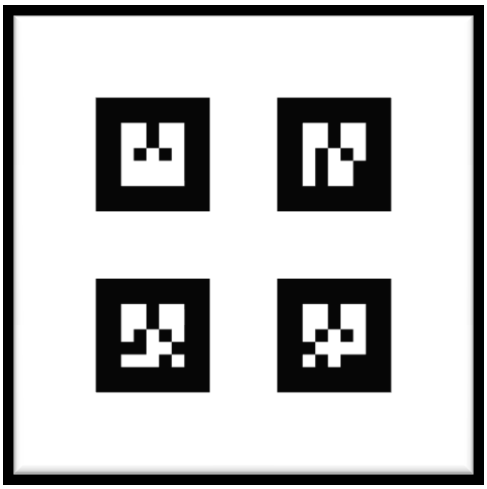

Fig. 1. Sample Fiducial Marker

Fig. 2. Keypoint Tracking with Homography

Firstly, we experienced a drift issue. For the most part, the KLT tracker did a good job of tracking the movement of individual points through time. However, there were occasions where the points would drift a little, or error would build up, and their positions relative to each other would change slightly. This happened most often on large rotations or orientation changes. This made it increasingly difficult to calculate a good homography between the clean marker image and the video frame.

Also, while unrelated to the KLT tracker itself, we discovered, unsurprisingly, that SIFT keypoint detection and matching is a very slow process. In the detection loop, our program ran at frame rates of around 2 FPS, which is enough to make the video feed look staggered and hard to look at. Once it moved into the tracking loop, as mentioned above, we were able to achieve roughly 11 FPS, which was enough for the video to feel somewhat smooth, and was fast enough to

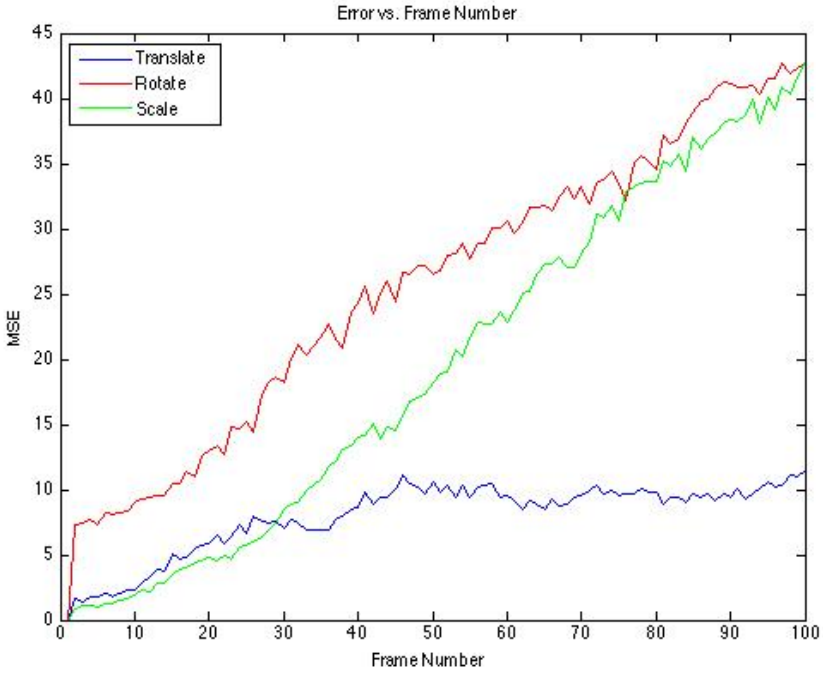

Fig. 3. Error Term

run the tracker on live video.

Finally, we discovered that, while the KLT tracker was quite robust while tracking translational displacement, it began to break down under rotation and scaling of the marker. We plotted the mean squared error term between the first and current frames under translation, rotation, and scaling. This plot can be seen in Figure 3. Unsurprisingly, the plots confirmed what we expected: translation performed well, and the other two didn't. This is a property of the KLT algorithm as described in the papers: it only attempts to reduce the error vector between frames for translational movement, and it doesn't take into account other kinds of affine transformation.

VI. Conclusion

After implementing our own KLT tracker and playing with both our and OpenCV's implementations of the tracker, we concluded that while the method was decent at achieving our goal of smoothly tracking the marker, there are likely better method that in practice work more efficiently and more robustly than the one we chose to implement. After much tweaking, the KLT tracker was simply not precise enough to accurately track the movement of the keypoints through the video feed. Small perturbations in the positions of keypoints relative to each other affected the resulting homography, making our results less than ideal. The per-frame contour-based method described in section II (based on the ARma library) performed noticeably better at tracking the marker through the motion of the video.

In conclusion, although it was an excellent learning experience to implement and experiment with the KLT tracking algorithm for marker keypoint tracking, we cannot recommend it, as we implemented it, for professional solutions.

Work Breakdown

The breakdown of work was fairly simple. We both worked on both parts, with Oliver tackling more of the coding, and Dave tackling more of the paper/poster writing.

References

[1] Bruce D. Lucas and Takeo Kanade. *An Iterative Image Registration Technique with an Application to Stereo Vision.* International Joint Conference on Artificial Intelligence, pages 674-679, 1981.

[2] Carlo Tomasi and Takeo Kanade. *Detection and Tracking of Point Features.* Carnegie Mellon University Technical Report CMU-CS-91-132, April 1991.

[3] Jianbo Shi and Carlo Tomasi. *Good Features to Track.* IEEE Conference on Computer Vision and Pattern Recognition, pages 593-600, 1994.

[4] J. K. Suhr, *Kanade-Lucas-Tomasi (KLT) Feature Tracker.* Course notes, Yonsei University, Korea, 2009. (http://web.yonsei.ac.kr/jksuhr/articles/Kanade-Lucas-Tomasi%20Tracker.pdf)

[5] J.Y. Bouguet, *Pyramidal Implementation of the Lucas Kanade Feature Tracker Description of the Algorithm.* technical report, Intel Microprocessor Research Labs, 1999.