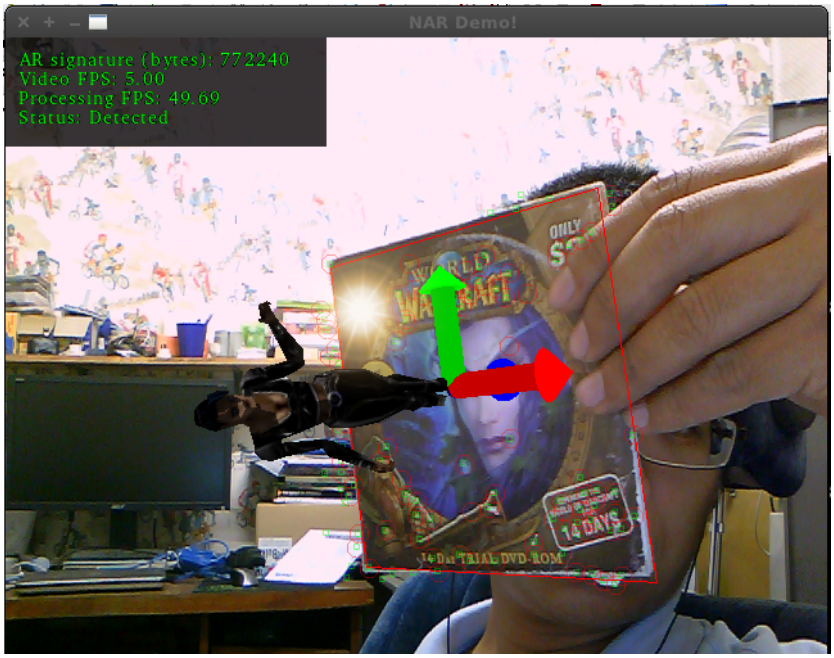
Nghia's Augmented Reality (NAR) demo
System documentation (work in progress)

Nghia Ho

5th Feb 2012



1 Introduction

I wrote NAR Demo as an improvement to MarkerlessARDemo. MarkerlessARDemo was my first attempt at markerless augmented reality (AR). At the time I wanted to learn how an AR system worked. One of the main problems with MarkerlessARDemo was its reliant on CUDA, which I happened to be learning at the time. I have since then removed any GPU dependent code and made numerous changes. Overall NAR Demo works better, has less ambiguous parameters to tune, and cleaner code. I want to make NAR Demo easy enough for a beginner to poke around and understand the system. I hope someone will find it useful in their adventures into the AR field.

In this document I will be describing how NAR works and what makes it tick. I'll be focusing on the computer vision aspect of NAR and not on the code. The aim is to give you a good idea

of how things work without bogging you down with the nitty gritty implementation details. For that you can just look at the code.

I assume the reader has some familiarity with common algorithms found in computer vision, such as

- Gaussian blurring
- Homography
- K-means clustering
- Optical flow

This document is organised as follows, Section 2 will give an overview of the system architecture, section 3 talks about the keypoint detection process, section 4 the feature descriptor used, section 5the detection and pose estimation process, section 6 talks about how the AR object is learnt for detection, and some final words in section 7 to wrap things up.

2 System architecture

NAR is implemented as a multi-threaded system. This is required to achieve real-time frame rates, or close to it. This comes at a small cost of latency/lag between when the frame arrives and when it gets processed. Figure 1 shows a flow chart of the threading system used. The work flow is typical of what you might find in a markerless augmented reality system, that is

1. Detect suitable keypoint features in the image
2. Extract a feature vector at each keypoint
3. Match the features
4. Filter out the bad matches
5. Perform pose estimation on the good matches

Step 1 is done in KeyPointThread, step 2 inside ExtractFeatureThread and steps 3 to 5 are inside the NAR::DoWork function. There are a total of 8 threads, VideoThread + KeyPoint-Thread + ExtractThread + a thread that calls the NAR::DoWork function. The system will run in real-time provided each thread runs at least or faster than the incoming video frame per second, and provided you have a multi-core CPU. I'll briefly mention the first two stages in the threading pipeline, with the rest in greater detail in the later sections.

2.1 VideoThread

This thread's job is simply to grab images from the webcam/camera and send it off to the main NAR thread.

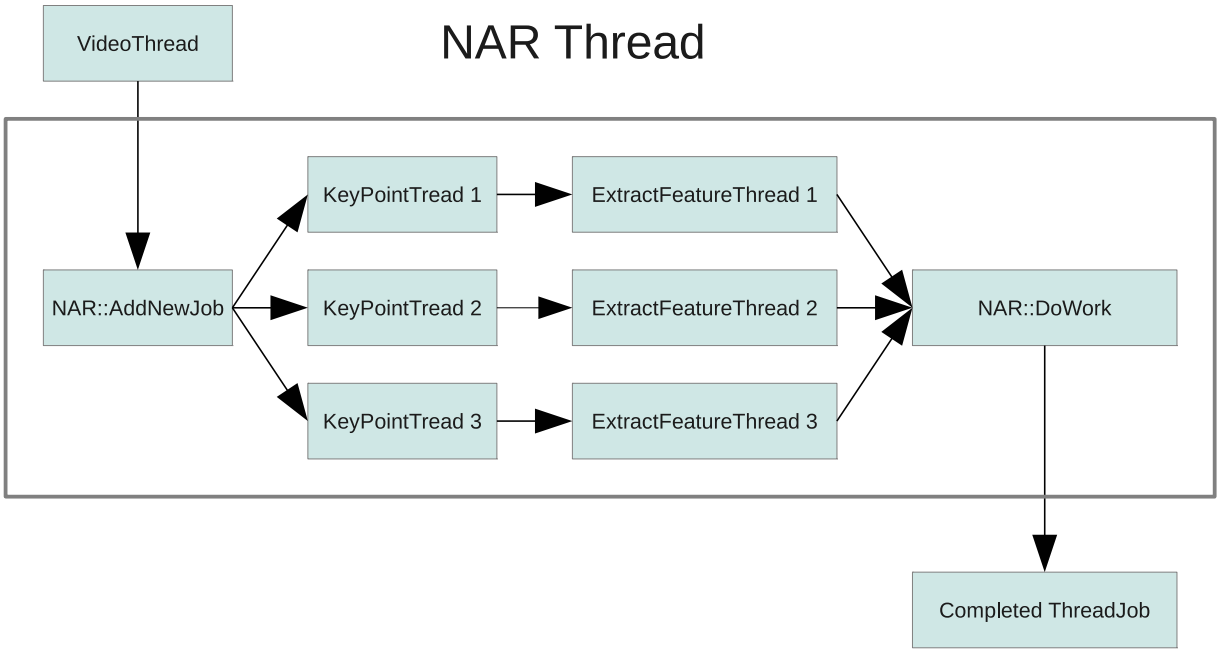


Figure 1: NAR thread flow chart

2.2 NAR::AddNewJob

The NAR thread manages the internal KeyPointThread and ExtractFeatureThread. The entry point to the NAR thread class is the NAR::AddNewJob function, which accepts an incoming images. NAR::AddNewJob creates 3 different resized images from the original image. This is done to detect features at different resolution levels. The default setting uses an image scaling factor of 0.75. The 3 images are produced using the follow equation

$$new\,size = original\,size \times 0.75^k,\ k = [0,1,2] \tag{1}$$

As an example, a 640x480 will result in following 3 images.

k	image size
0	640x480
1	480x360
2	360x270

Each resized image is sent to the respective KeyPointThread.

3 KeyPointThread

The KeyPointThread's job is to detect good features in the incoming image. A good keypoint feature should be robust to noise (a must on crappy webcams) and repeatable for a wide range of viewpoints. I chose the Difference of Gaussian (DOG), an approximation to the Laplacian,

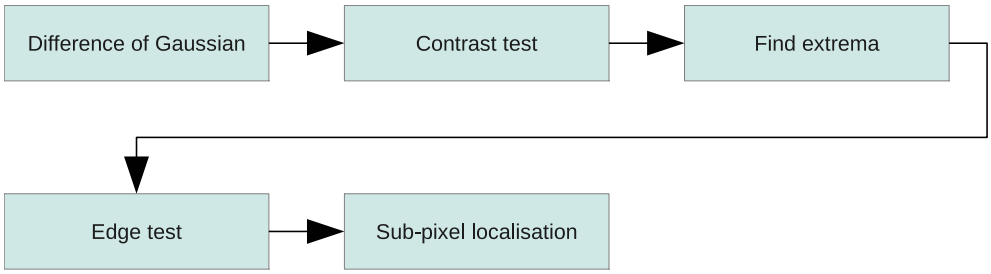


Figure 2: Keypoint detection steps

because it has these properties. I've tried FAST [5] but it seems too susceptible to noise, though having said that I have not done a thorough test. Figure 2 shows the keypoint detection steps, most of the idea is borrowed from SIFT [2].

3.1 Difference of Gaussian (DOG)

The DOG is the difference between two Gaussian blurred images, expressed as

$$DOG = G\left(image, \sigma_1\right) - G\left(image, \sigma 2\right) \tag{2}$$

where G is the Gaussian blurring function of an image by some sigma value. The two sigma values I used are

$$\begin{aligned} \sigma_1 &= 1.6 \\ \sigma_2 &= 1.6 * 1.6 = 2.56 \end{aligned} \tag{3}$$

I chose 1.6 based on SIFT's pre blurring value, the second sigma is based on a Wikipedia article (`http://en.wikipedia.org/wiki/Difference_of_Gaussians`) stating a ratio of 1.6 is a good approximation to the Laplacian. Larger sigma values will detect larger blobs but at a cost of an increase in run time. Since we are detecting a fixed size blob at different image resolutions we end up detecting large blobs at the lower resolution images. Each pixel in the DOG image is examined to see if it passes all the tests shown in figure 2.

3.2 Contrast test

The contrast test checks to see if the absolute pixel value in the DOG image is above a certain threshold, expressed as

$$|DOG(x, y)| > t \tag{4}$$

The $|.|$ is the absolute value function. The DOG values can be thought of as contrast values of blob features. A low value means low contrast, which indicates a weak blob feature, and hence will fail the test. Since this is the cheapest test it is performed first. I use a $t = 7.0$.

-3	5	6
7	-10	-1
8	9	0

0	6	-2
6	8	-9
-1	2	5

Figure 3: Example of 3x3 minima/maxima extrema pixels

3.3 Finding extrema

This step looks at the 3x3 neighbouring pixels to see if the pixel stands out, either lower or greater than all the neighbouring values. If it meets this requirement then it is an extrema. An example of a 3x3 minima/maxima extrema is shown in figure 3. The centre value is compared to the 8 neighbouring pixel values. In practice the values are floats. This is different to SIFT, where the extrema is done in scale space as well, resulting in a 3D grid comparison.

3.4 Edge test

The edge test calculates the principal curvature at DOG(x,y) and determines whether the feature is too 'edge' like using the ratio of eigenvalue values, but uses a maths trick so that the eigenvalues don't have to be explicitly computed. The full eigenvalue calculation involves nasty square roots. If a pixel position is deemed to edge like it is skipped, because they are not stable features. Figure 4 shows an image exhibiting a strong edge and the location of interest to do the edge test. I added noise to make it easier to see the white part of the image. The red circle in the Difference of Gaussian image is the location of interest.

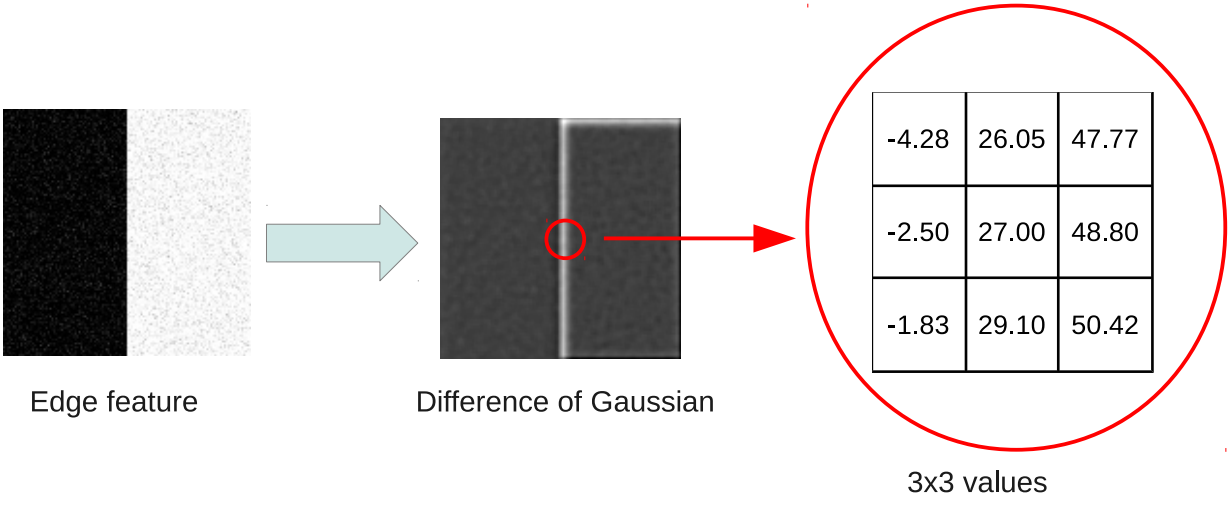


Figure 4: Edge test using 3x3 region

Let W be the 3x3 values indexed by

$$W = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \tag{5}$$

The edge test calculation for the 3x3 window is as follows

$$D_{xx} = d - 2e + f \tag{6}$$

$$D_{yy} = b - 2e + h \tag{7}$$

$$D_{x1} = (c - a)/2 \tag{8}$$

$$D_{x2} = (i - h)/2 \tag{9}$$

$$D_{xy} = (D_{x2} - D_{x1})/2 \tag{10}$$

The above quantities are the second order derivatives using finite differencing. The ratio of the eigenvalues is calculated as

$$trace = D_{xx} + D_{yy} \tag{11}$$

$$det = D_{xx}D_{yy} - D_{xy}D_{xy} \tag{12}$$

$$ratio = trace^2/det \tag{13}$$

$$ratio > threshold? \tag{14}$$

If the ratio exceeds a threshold it is deemed an edge and the pixel is skipped. The threshold used is 12.1. If *det* is less than or equal to zero then it is not an extrema location and skipped as well. This happens when the sub-pixel localistion step does not converge correctly, which is not often .The example in figure 4 has a ratio of -0.13220 and is thus not an extrema. The calculation shown is very similar to the one used to calculate Harris corner features [1], but uses the 2x2 Hessian matrix instead.

3.5 Sub-pixel localisation

The sub-pixel localisation step refines the integer (x,y) position into floating point position using gradient descent. This is important when detecting keypoints at the lower resolution images because any errors in the extrema's position are magnified when scaling back up to the original image resolution. The extrema's sub-pixel location is found using basic gradient descent with a learning rate of 0.5. The cost function that gets minimised is

$$cost = dx^2 + dy^2 \tag{15}$$

where dx and dy are the first order derivatives at DOG(x,y). The derivative at DOG(x,y) is evaluated using the following finite differencing

$$dx = DOG\left(x + 0.1, y\right) - DOG\left(x - 0.1, y\right) \tag{16}$$

$$dy = DOG\left(x, y + 0.1\right) - DOG\left(x, y - 0.1\right) \tag{17}$$

The gradient descent requires evaluating pixels at fractional position using bilinear interpolation. The gradient descent generally converges quickly. Sub-pixel localisation is only used when the input image is smaller than the original image size. I didn't see any point in doing it for the original size image since it's theoretically accurate to within a pixel already, and would add an unnecessary computation burden since more features are detected at the original image size.

4 ExtractFeatureThread

This thread extracts a feature vector at the (x,y) position found by the KeyPointThread. I used my own home brew rotation invariant binary feature patch, having a length of 64 bits (8 bytes) per keypoint. The feature extraction process is shown in figure 5.

For a given keypoint (x,y) position, an orientation is calculated using pixels within a circle of radius 15. It's actually a circle within a 32x32 patch, but the circle doesn't centre perfectly within the patch. For more information on the orientation calculation please see my blog post `http://nghiaho.com/?p=1198`.

The orientation is used to extract a 32x32 rotated patch, which is then sampled every 4 pixels to produce a down sampled 8x8 patch. In practice, the rotation and downsampling can be combined into one step for efficiency. The average intensity of the patch is then calculated and used to binarise the image by thresholding at this value. The final feature vector consists of 64 bits. The 64 bit length was carefully chosen to allow the use of the POPCNT assembly instruction found in CPUs supporting SSE4. The instruction counts the numbers of 1s in a 64 bit number in a single operation.

5 NAR::DoWork

This function is where the magic happens. It's job is to perform pose estimation to determine the 3D pose of the AR object. Figure 6 outlines the steps involved.

5.1 Feature matching

This uses the extracted features from the three ExtractFeatureThread and matches them against the AR object's feature (extracted beforehand). To speed up matching I used a hierarchical K-means tree, internally just called a K-tree. If you are familiar with K-means, the K-tree partitions the data by splitting it into 2 centroids recursively for a fixed depth. Searches are performed just like in a binary tree by comparing which centroid is closer to the query and going down the tree. The final leaf node will contain a handful of features that have to be searched linearly, but this is okay because there's only a handful of them to go through.

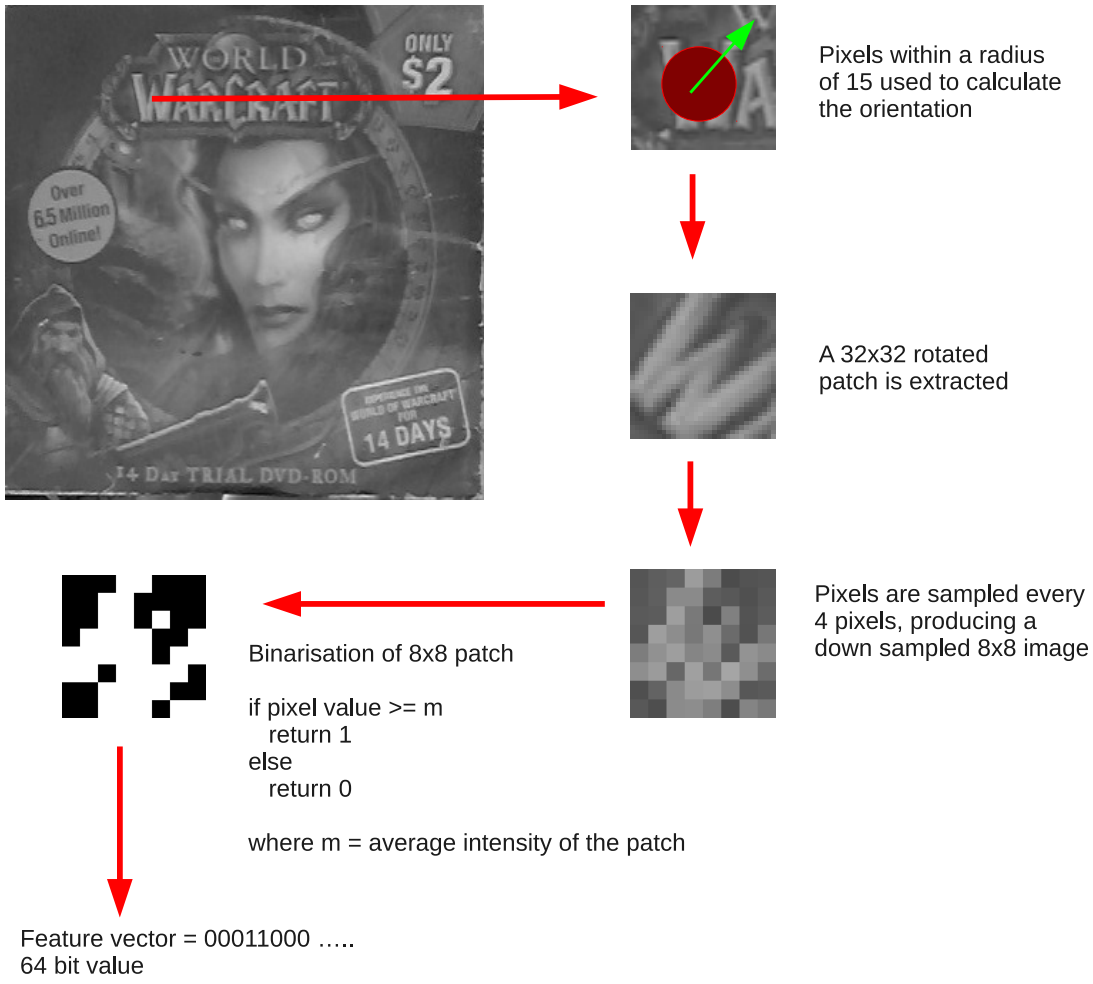


Figure 5: Extracting feature vector

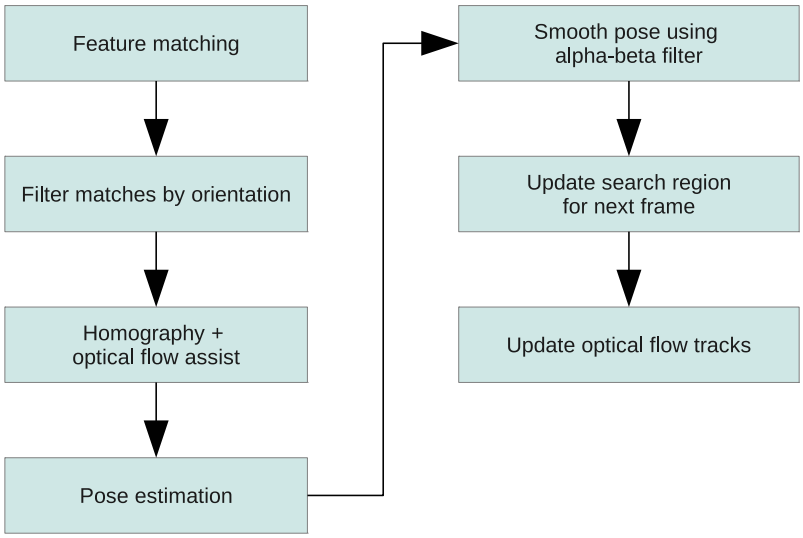


Figure 6: NAR::DoWork process

The trade off for a deeper tree is accuracy. My implementation does not traverse back up the tree to visit other potential nodes like in a standard KD-tree implementation. I've found in practice this is not required.

Some of the matches will not be one to one, that is more than one query feature can point to the same feature in the K-tree. To resolve this conflict, the one with the lowest matching score is kept. Here, a low score means a good match, using Hamming distance metric.

5.2 Filtering matches by orientation

A lot of the initial matches will be bad matches that need to be pruned out. We can take advantage of the feature's orientation value to do this. I borrowed this idea from [8]. Remember back that I use a rotation invariant feature that has an orientation associated with it. This means the relative angle between matches are

$$relative\,angle = ShortestAngle\,(query,\,match) \tag{18}$$

The ShortestAngle is a function that returns the shortest angle between A and B, and is between 0 and 180 degrees. The angles are the accumulated into a histogram of 36 bins (every 10 degrees). Matches in the top two bins are kept. I chose the top two instead of the top to allow for angular errors in the binning processing.

5.3 Homography with optical flow assist

After filtering the matches by the orientation, they are filtered further using the homography. RANSAC is used to find the best homography between the matches. However, I've observed that using matches from the DOG features only are not enough to produce a stable homography. It tends to jitter even when the AR object is stationary. To remedy this, additional matches from optical flow tracks are added. The optical flow tracks are initialised in the previous frame

and tracked in the current frame using the popular Lucas-Kanade optical flow tracker [3] found in OpenCV. The extra matches improve stability a lot.

5.4 Pose estimation

By this stage we should have filtered out most, if not all, of the bad matches and are only left with good matches. The pose is estimated from these matches using the Robust Planar Pose (RPP) algorithm by [6]. This algorithm does not incorporate any outlier filtering, hence why it's very important to give it clean data. The algorithm finds the best pose estimate given 2D to 3D correspondences. The 3D points here are the features detected in the AR object, with z=0. The x,y positions are normalised in a way such that (0,0) is at the centre of the image and the image width is length 1.0, the height is scaled so that the aspect ratio does not change. The pose estimated is relative to this normalised coordinate space.

5.5 Smoothing pose using alpha-beta filter

This step takes the estimated pose from the RPP and smooths it using an alpha-beta filter (`http://en.wikipedia.org/wiki/Alpha_beta_filter`). It is similar to a Kalman filter but less complex and less parameters to tune, the alpha and beta values. The alpha controls how responsive the filter is to new pose input (position and rotation) and the beta controls the response to new velocity input. The alpha/beta gains range from [0,1]. In general. the lower the gains are the smoother the pose estimates but the less responsive it is to new input. I don't actually use velocity because I found it unreliable, especially when there is motion blur. My alpha-beta filter update equation is

$$r_t = X_t - \hat{X}_{t-1} \tag{19}$$

$$\hat{X}_t = \hat{X}_{t-1} + \alpha r_t \tag{20}$$

where $\hat{X}_t$ is the current smoothed out pose estimate. $\hat{X}_{t-1}$ is the previous estimate, X_t is the raw pose input, r_t is the residual, and α is the gain. The pose is a 6D vector (x, y, z, yaw, pitch, roll). The smoothing is used to reduce jittering of the AR object. This step is not required if the pose estimation step mentioned previously is very stable.

5.6 Update search region

Being able to predict where the AR object might be in the next frame is important for good performance. It allows us to speed up feature matching and reduce bad matches because we don't have to process every pixel. Instead of using an active search region to predict where the AR object is in the next frame, say using results from the alpha-beta filter (or Kalman filter), I use a simpler approach instead.

My approach assumes the AR object is mostly stationary and will restrict keypoint detection to a region in the image, formed by a bounding box around the rectangular contour of the AR object in the previous frame. This works well if the movement is not too large, because the search region will shift with the AR object as it moves. There is an option in the system to increase the search region by padding the borders to cater for larger movements. An illustrative example is shown in figure 7.

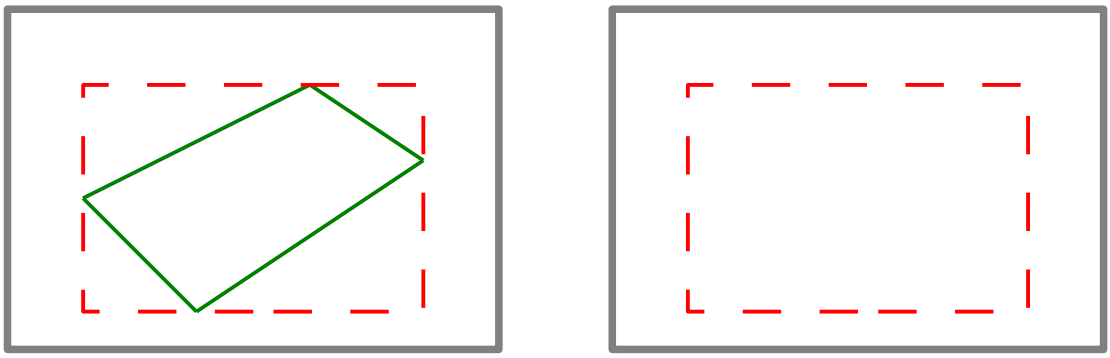


Figure 7: Simple search region used by NAR

5.7 Update optical flow tracks

This step will add optical flow tracks or reset the optical flow tracks depending on the situation. Optical flows are reseted often to prevent them from going 'stale' due to drifts or bad tracking, which will mess up pose estimation. When that happens new fresh optical flow tracks are added. The heuristics for this process is illustrated in the flow chart in figure 8.

Optical flow tracks are reset if the count exceeds 16. The count is incremented if the system detects significant movement from the AR object. This is based on my observation that optical flow works well when tracking an object that doesn't move too fast, before motion blur kicks in, but starts to degrade quickly in the opposite case. When the object is moving fast the tracked optical flow vectors can track incorrectly. Optical flow tracks are also reseted if more than a 30% of the original numbers of track have been lost over time (due to tracking failure).

6 Learning the AR object for recognition

Up until now I've been focusing on the low level steps involved in detecting and estimating the pose of the AR object. I will now discuss how the AR object is learnt and features extracted. This is done when the program starts up and is a fast process.

My method for learning the AR object is inspired by [4]. The author presents a method where by a planar object is warped with random affine transforms, producing virtually unlimited appearances from different viewpoint. The idea is to learn the object by observing it from as much view as possible. At each view simple features are extracted. They used a DOG at 3 scale levels to detect keypoints and random binary comparisons to produce a 300 bit feature vector. The feature vector is not rotation, scale, or viewpoint invariant but is illumination invariant.

My approach is similar but instead of warping with a random affine, my affine is parameterised by three rotation angles: yaw, pitch, roll (always 0). Sscale is done implicitly by resizing the image. The steps to derive the 2D affine matrix are

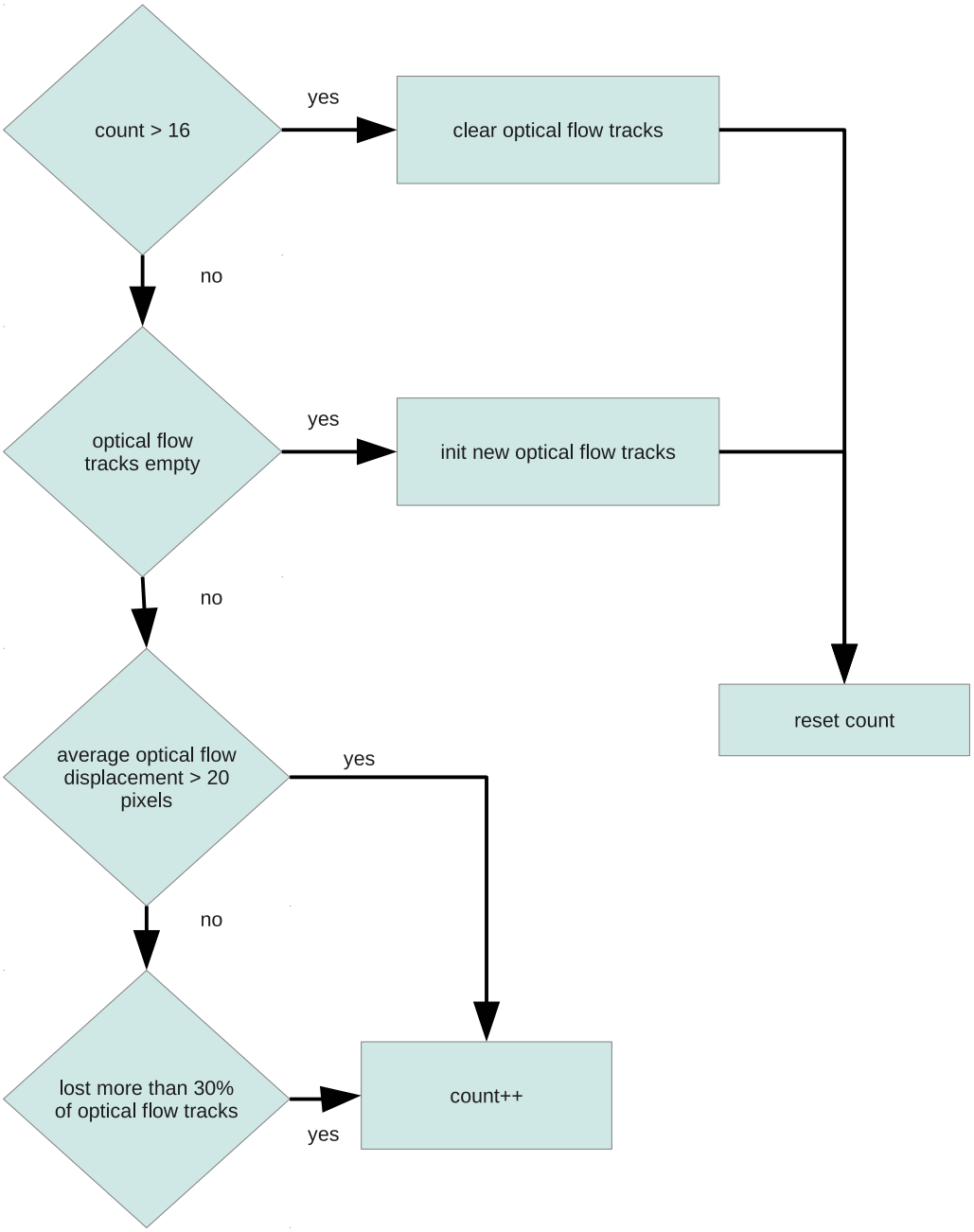


Figure 8: Optical flow update

$$R_{4x4} = MakeRotationMatrix\,(yaw, pitch, roll) \tag{21}$$

$$T = T_{centre} R_{4x4} T_{origin} \tag{22}$$

$$A_{2x3} = \begin{bmatrix} X & X & - & X \\ X & X & - & X \\ - & - & - & - \\ 0 & 0 & 0 & 1 \end{bmatrix}_T \tag{23}$$

where $MakeRotationMatrix$ is a function that returns a 4x4 rotation matrix from yaw, pitch, roll, the matrix is padded with zeros where required. T_{origin} is a 4x4 matrix that shifts the origin to the centre of the image, T_{centre} is a 4x4 matrix that undoes the shift by T_{origin}. A_{2x3} is the 2x3 affine matrix, by taking values marked X in matrix T.

The three rotation angles are sampled at fixed intervals as follows, all values are in degrees:

angle	start	end	angle step
yaw	0	60	10
pitch	0	60	10
roll	-	-	-

I prefer this method over random values because I know I'll get the same results every time and it samples the viewpoint space uniformly. An example of affine warping is shown in figure 9.

Figure 9: Example of affine warping parameterised by yaw, pitch, roll

For each warped image, DOG keypoints are detected and feature vectors extracted just as described previously in the KeyPointThread and ExtractFeatureThread. However there is one minor difference. The orientation calculation for the patch, to get the rotation invariance, has

an angle ambiguity. There are two valid angles, the second one being out by 180 degrees. So for each patch two binary vectors are stored.

7 Some final words

That brings us to end of the document. We've covered the threading pipeline, keypoint detection, feature extraction, feature matching and pose estimation. Some sections I've skimmed through while others have been given more time. This choice was made based on whether I felt a topic was better explained with figures and equations, or adequately explained with a few passing sentences, and have the reader find information about it somewhere else. I chose to expand on the keypoint detection step because the sub topics are often found in computer vision but lack adequate resources for beginners eg. papers, tutorials, blogs. I might continue to keep expanding the topics if there is a demand for them and maybe treat them as independent tutorial blogs.

So far I've only focused on the technical aspect of the AR system. However, the most important aspect from a user point of view is the actual performance of the system. The performance includes

1. Real-time run speed
2. Detection robustness of the AR object in difficult views
3. Tracking robustness of the AR object during motion
4. Quality of pose estimation, jitter reduction

The four points are general and not application specific but are typical of the challenges facing any AR system. On a desktop computer it is quite easy to achieve real-time speed but the trend these days is to get AR systems working on mobile devices [9] and even towards tracking of multiple objects [7]. Due to the limited processing power of smart phone, as compared to a desktop computer, they have to utilise clever tricks and efficient algorithms to get the job done. Because of this, expect a lot of cool innovate algorithms stemming from this.

Anyway, I hope I've explain how NAR works well enough. Now go fourth and build your own AR system!

References

[1] C. Harris and M. Stephens. A combined corner and edge detector. In *Proc. Fourth Alvey Vision Conference*, pages 147–151, 1988.

[2] David G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, November 2004.

[3] Bruce D. Lucas and Takeo Kanade. An iterative image registration technique with an application to stereo vision (darpa). In *Proceedings of the 1981 DARPA Image Understanding Workshop*, pages 121–130, April 1981.

[4] P. Fua M. Ozuysal and V. Lepetit. Fast keypoint recognition in ten lines of code. In *Conference on Computer Vision and Pattern Recognition*, 2007.

[5] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In *European Conference on Computer Vision*, volume 1, pages 430–443, May 2006.

[6] G. Schweighofer and A. Pinz. Robust pose estimation from a planar target. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 28(12):2024 –2030, dec. 2006.

[7] Simon Taylor and Tom Drummond. Multiple target localisation at over 100 fps. In *British Machine Vision Conference*, September 2009.

[8] D. Wagner, G. Reitmayr, A. Mulloni, T. Drummond, and D. Schmalstieg. Pose tracking from natural features on mobile phones. In *Mixed and Augmented Reality, 2008. ISMAR 2008. 7th IEEE/ACM International Symposium on*, pages 125 –134, September 2008.

[9] D. Wagner, G. Reitmayr, A. Mulloni, T. Drummond, and D. Schmalstieg. Real-time detection and tracking for augmented reality on mobile phones. *Visualization and Computer Graphics, IEEE Transactions on*, 16(3):355 –368, may-june 2010.