ObjTrack

Group 15
Anson Chan asc@ece.cmu.edu
Andrew Y Ng ayn@ece.cmu.edu
(Sabi Varon svaron@ece.cmu.edu)

Final Report

18-551
Digital Communication and Signal Processing System Design

Object Tracking via Optical Flow in Video

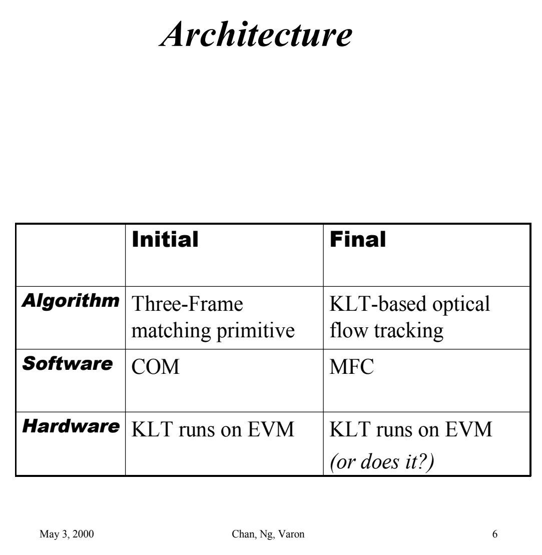
Anson Chan, Andrew Ng, (Sabi Varon)
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213
{asc,ayn,(svaron)}@ece.cmu.edu

Abstract

This project aimed to implement algorithms for tracking moving object in an AVI compressed video sequence. The system uses KLT (Kanade-Lucas-Tomasi Tracker) to select and track the good features on the object specified by the user. In addition to tracking, our system uses the tracked points and the change of velocity vectors to do object distribution analysis on the selected region. And a perspective-transformed diagram of the selected region shows a histogram of the distribution of the ball. It's particularly useful for some sport videos when determining where of the ball hits the ground is of an interest. (i.e tennis, volleyball).

Originally, we divided up the tasks up into three major parts, the KLT, software, and EVM porting. And we split up the work among the members. We were actually ahead of schedule before spring break, meaning that Andrew and Anson had the working software by spring break. The software was tested thoroughly by Andrew during spring break, and was handed to Sabi to take care of the EVM coding before

April, which gave him a month to finish it, which was plenty of time for just porting a few functions which was originally written in ANSI C anyway. However, we last found out that Sabi was unable to port the code to EVM, and that he hadn't started working on it until late April. Therefore we really regret to give him full responsibility on his part. However, since Anson and Andrew did everything (including the homework) before the EVM part, we felt that Sabi should contribute at least a little in the project.



Architecture

	Initial	**Final**
Algorithm	Three-Frame matching primitive	KLT-based optical flow tracking
Software	COM	MFC
Hardware	KLT runs on EVM	KLT runs on EVM *(or does it?)*

May 3, 2000 Chan, Ng, Varon 6

Initially we proposed using the *Three-Frame Matching Primitive* for doing tracking, this method also uses the KLT tracking algorithm to compute the optical flow, however it uses 3 frames for doing the chaining of the object motion. Supposedly this method is more accurate due to the trilinear constraints, however the paper did not provide enough details when we actually wanted to implement the algorithm.

We were going to use COM, Microsoft's *Component Object Model*, to do the software. We thought that it would enable integration easier and so forth. However, only one of our members were familiar with COM, therefore it was not feasible using it. We therefore switched to pure MFC application, which is simple enough for at least two members to program in.

KLT Feature Tracking Method

Feature tracking is a widely researched topic in the Computer Vision community. The most commonly used methods for feature tracking employ image correlation or sum of squared difference (SSD) techniques. With small inter-frame displacements, a window can be tracked by optimizing some matching criterion with respect to translation and linear image deformation. Our current system tracks features on the frame using the method proposed by Tomasi.et.al (KLT system). The KLT (Kanade Lucas Tomasi) feature tracking system identifies and tracks features by monitoring a measure of feature dissimilarity. Features with good texture are selected and are tracked using an affine tracking model, which can compensate for translation and linear warping. The affine model is computed numerically using Newton Raphson minimization technique. Translation gives more reliable results than affine changes when the inter-frame motion is small, but affine changes are necessary to compare frames with large motions to determine dissimilarity. We tested the tracker on a few sport sequences and observed that features which were identified were tracked satisfactory given that the ball move less than 20 pixels away in the subsequent frame.

The feature tracker works in realtime at around 3Hz for a 320 x 240 video.

We obtained a fully implemented KLT C library from Stanford University [1]. There are two main functions we used. SelectGoodFeature and TrackFeature. Both of these functions are computation intensive which involve 2D convolution, and quite a few loops for iterative minimization and sorting. Here's how the KLT algorithm works. After the user select an elliptical region, the GUI sends the images to the function SelectGoodFeature and identify N "good and trackable" points within the region. To identify "goodness", first, gradients are computed from the resulting image by convolving with derivative of a Gaussian of sigma. These gradients are used to select the features. Generally, pixels throughout the elliptical region are considered. The goodness of each pixel is measured as the minimum eigenvalue of the 2x2 gradient matrix around the window. After all the pixels have been considered, their goodness are sorted in descending order. Only those

whose value are greater than the "min_eigenvalue" defined by the tracking context are selected as "good and trackable points". Then the KLT will try to find these points in the next image in order to track the object.

To find the object in the second frame, he function TrackFeature creates a multi-resolution image pyramid and subsampled between each level. At each resolution, tracking is accomplished by a iterative minimization between the intensities of the two windows. Five possible reason to cause the tracking to stop, each "good" feature identified in the first image will fall into one of these group.

The feature moves by no more than the "min_displacement" defined by the following criterias:

1. the tracking context. Successfully tracked
2. The determinant of the 2x2 gradient matrix is too small.
3. The number of iterations exceeds limit.
4. The feature is out of bound (happen when the object is close to the border).
5. The average intensity difference between images is too large.

Only the first group of points is tracked, our GUI will display red dots to show it's successfully tracked.

More Details of Kanade-Lucas-Tomasi (KLT) Feature Tracking Algorithm

Feature tracking is an essential element in many computer vision systems. Most 3-d reconstruction from motion or stereo systems requires relatively high quality (reliable and accurate) feature correspondence information to generate a coarse model of the world. For our project, we emphasize the tracking of an object through the video. Someone might ask why we use this algorithm rather a simpler correlation filter to find where the object is in the subsequent frame. Keep in mind that this algorithm is more than just a object tracker, it is a feature tracker meaning that it is able to handle any translational or rotational or sizing (zooming) changes and all features can move to different directions and be tracked successfully. The Kanade-Lucas-Tomasi algorithm for feature tracking is an excellent choice for one reason: KLT both reliably tracks features across frames and determines whether a feature is still "good" enough to continue tracking.

The general idea of the KLT tracking algorithm proceeds as follows:

GUI selects region, then a feature selector initially selects features.

These features are then tracked through the image sequence using a Newton-Raphson method in order to minimize sum-of-squared difference between the feature windows in two successive frames.

After the new location of the feature is determined in the next frame, the quality of this feature is evaluated and bad features are dropped.

Using the remaining features, repeat the process starting from step 2 for the next image in the sequence. If not enough features remain; new features can be added using the original feature selector.

Luckily, we were assisted and directed by Prof Chen to use an implemented C library from Stanford. The library we downloaded has about thousand lines of code, so it is quite computation intensive and took us a while to understand. Before getting into all the signal processing and mathematical details of KLT, let’s get familiar with some of the important parameters for the use of the tracker. These parameters are contained in the tracking context (tc), whose members we now list, along with a brief description of each:

```
int mindist;            /* minimum distance between
selected features */

int window_width;       /* dimensions of feature window */

int window_height;   /* dimensions of feature window */

KLT_BOOL sequentialMode;  /* whether to save most recent
image */

KLT_BOOL smoothBeforeSelecting; /* whether to smooth image
before selecting features */

KLT_BOOL writeInternalImages;   /* whether to write
internal images for later viewing */

int min_eigenvalue;      /* smallest eigenvalue allowed for
selecting */

float min_determinant;    /* min determinant for declaring
tracking failure */

float min_displacement;   /* amount of pixel shift for
stopping tracking iterations */

int max_iterations;       /* max iterations before
declaring tracking failure */

float grad_sigma;         /* sigma of gaussian for
computing gradient */
```

```
float smooth_sigma_fact;  /* sigma factor of gaussian for
smoothing image */

float pyramid_sigma_fact; /* sigma factor of gaussian for
computing image pyramid */

int nSkippedPixels;       /* used to speed up feature
selection */

int nPyramidLevels;       /* number of pyramid levels */

int subsampling;          /* amount of subsampling between
pyramid levels */
```

Procedures of our system

Step 1: GUI selects the region of the object

This is the region where the slect feature function will find the “good” features.

Step 2: Select Good Features in the first frame

`KLTSelectGoodFeatures()` takes the first frame of the video. If `tc->smoothBeforeSelecting` is set to `TRUE`, then the image is smoothed by convolving with a Gaussian of

`sigma = tc->smooth_sigma_fact * max(tc->window_width, tc->window_height);`

otherwise, the image is not smoothed. In either case, gradients are computed from the resulting image by convolving with the derivative of a Gaussian of

`sigma = tc->grad_sigma`.

These gradients (one in the *x* direction and the other in the *y* direction) are used to select the features.

Pixels throughout the selected region from step 1 are then measured as to their "goodness", which is a measure of their trackability. The parameter `tc->nSkippedPixels` can be used to speed up the process in the following way: its default value is zero, in which case every pixel within the interior is considered; if it is set to one, then every other pixel within the interior is considered; setting it to two causes every third pixel to be considered; and similarly for higher values. Since neighboring pixels generally have similar goodness values, then skipping every other one will probably not noticeably decrease performance, at the same time, it can speed up the run time.

The goodness of each pixel is measured as the minimum eigenvalue of the 2 by 2 gradient matrix computed from the `tc->window_width` by `tc->window_height` window around the pixel. After all the pixels have been considered, they are sorted in descending order according to goodness. Then, one by one the top `fl->nFeatures` features (or pixels) whose minimum eigenvalue is at least `tc->min_eigenvalue` are selected, ensuring that each new feature is at least `tc->mindist` pixels away from all the other features.

If `tc->writeInternalImages` is `TRUE`, then the smoothed image and the image derivatives are written to `"kltimg_sgfrlf.pgm"`, `"kltimg_sgfrlf_gx.pgm"`, and `"kltimg_sgfrlf_gy.pgm"`, respectively. This allows the user to more intelligently select the parameters for smoothing and differentiating.

Basically, the feature selector used a corners algorithm. Corners were detected by computing the gradient over the entire selected region, computing the matrix C for each point, and selecting the points with corresponding eigenvalue, lambda of C, (the smaller of the two eigenvalues) greater than a given threshold (`tc->min_eigenvalue`).

$$C = \begin{bmatrix} \sum g_x^2 & \sum g_x g_y \\ \sum g_x g_y & \sum g_x^2 \end{bmatrix}$$

Step 3: Track Features in subsequent frame(s)

`KLTTrackFeatures()` takes two images, frame 1 and frame 2. The tracking context has a member called *sequentialMode* which, when set to `TRUE`, causes `KLTTrackFeatures()` to store the gradients of the second image, along with its smoothed version, into the tracking context. This can greatly speed up the tracking process because when we tracked frame 2 and frame 3, we do not have to compute all the things for frame 2 again. KLT ignores its second parameter and replaces it with the previously stored image (except for the first time the function is called, in which case it must use both images). In either case (doing sequential mode or not), the resulting images are smoothed by convolving with a Gaussian of

`sigma = tc->smooth_sigma_fact * max(tc->window_width, tc->window_height).`

Then a multi-resolution image pyramid is created with `tc->nPyramidLevels` levels and `tc->subsampling` pixels subsampled between each level; smoothing before sampling is accomplished with

`sigma = tc->subsampling * tc->pyramid_sigma_fact.`

Gradients are computed at each level of the pyramid by convolving with the derivative of a Gaussian of

`sigma = tc->grad_sigma.`

In the feature list generated from step 2, each feature that is not lost is tracked beginning with the coarsest resolution and ending with the finest resolution, with each resolution providing the starting point for the subsequent resolution. At each resolution, tracking is accomplished by a ***Newton-Raphson*** iterative minimization between the intensities of the two windows, one window in each image. There are four conditions that cause the iterations to stop (only in the first case is the tracker successful):

the feature moves by no more than `tc->min_displacement`. It means the feature has been successfully tracked.

the determinant of the 2-by-2 gradient matrix is less than `tc->min_determinant`. It indicates that the feature has been lost due to the 2 by 2 gradient matrix having a small determinant

the number of iterations exceeds `tc->max_iterations`. It means that the feature has been lost because the number of iterations exceeded the maximum allowable.

the feature is out of bounds (i.e., it is within `tc->borderx` or `tc->bordery` of the border of the image) It means that the feature has been lost because it was out of bounds (i.e., it was too close to the image border).

This translational tracker is actually a simple optimization problem, where the sum of square differences measure (SSD) between the windows surrounding the two corresponding points in the two frames is to be minimized. Intuitively, when this measure is minimized we have the best match of features.

We define the SSD error function, epsilon as:

$$\varepsilon(\bar{x},\bar{d}) = \iint_{w(\bar{x})} [J(\bar{\xi}) - I(\bar{\xi} - \bar{d})]^2 \, d\bar{\xi}$$

where I and J are the two successive images, x is the position of the feature window, d is the translation, and the integral is over the area of the feature window. To find the translation of the feature, we minimize the error function by setting the derivative to zero:

$$\frac{\partial \varepsilon}{\partial \bar{d}} = \iint_{w(\bar{x})} [2[J(\bar{\xi}) - I(\bar{\xi} - \bar{d})]\nabla I] d\bar{\xi}$$

Taking the first order Taylor series expansion, this can be approximated as:

$$[\iint_{w(\bar{x})} \nabla I \nabla I^T d\bar{\xi}]\bar{d} = \iint_{w(\bar{x})} [[I(\bar{\xi}) - J(\bar{\xi})]\nabla I] d\bar{\xi}$$

Which reduces to the following linear system:

$$A\bar{d} = \bar{e}$$

where A is equal to the double integral on the left and e is equal to the double integral on the right.

Using this equation, we iterate using Newton-Raphson to lower the error introduced by the Taylor series approximation.

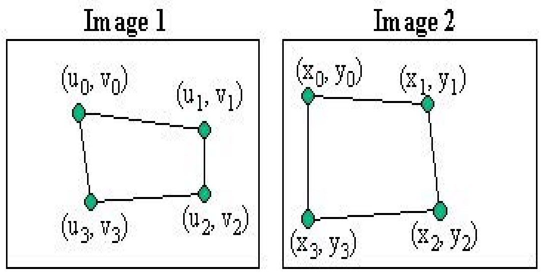
The translational tracker tracks all the features by performing the above computations for each feature between each successive pair of images and updating each feature location. A more detailed description of translational tracking can be found in Tomasi's paper of "Good Features to Track".

Step 4: Repeat Tracking Features for the subsequent pair (of Step 3) of images

If all the features were lost due to whatever reason, we will let the user to select the object manually again and we started the process of selecting and tracking features again.

Perspective transform for distribution histogram

After the object is successfully tracked through the entire video, our system is able to show where the object hits the user-defined region. For example, a user wants to track a tennis ball in a full-court-tennis-game video and is interested to see a hit-distribution of the ball on the tennis court. Our system is able to pinpoint the points when the ball bounces off the ground by detecting the changes of velocity vectors. However, since the camera can be in any direction when shooting the tennis game, a perspective transform is necessary for the ball distribution histogram displaying as if the camera is shooting from directly above the tennis court.



The vertices of the quadrilaterals in both the source (Image 1) and destination image (Image 2) are then used to compute an initial projective mapping, ***M***, that satisfies the following equation,

$$Mx = b$$

, where *x* is column vector representing a point in the source image, and b is a column vector representing a point in the destination image. To compute the forward mapping matrix ***M***, there are eight equations with eight unknowns (m_0 - m_7) for the vertices numbered cyclically $k = 0, 1, 2, 3$:

$$x_k = \frac{m_0 u_k + m_1 v_k + m_2}{m_6 u_k + m_7 v_k + 1} \Rightarrow$$

$$u_k m_0 + v_k m_1 + m_2 - u_k x_k m_6 - v_k x_k m_7 = x_k$$

$$y_k = \frac{m_3 u_k + m_4 v_k + m_5}{m_6 u_k + m_7 v_k + 1} \Rightarrow$$
$$u_k m_3 + v_k m_4 + m_5 - u_k y_k m_6 - v_k y_k m_7 = y_k$$

This can be rewritten as a linear system:

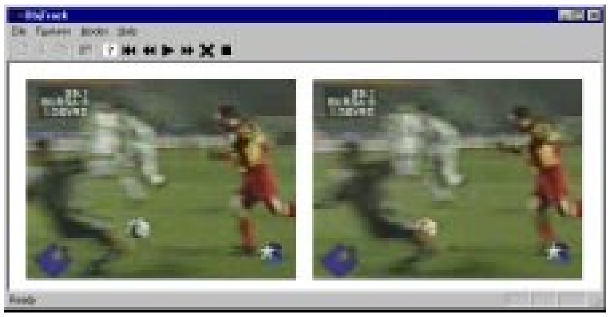
$$\begin{pmatrix} u_0 & v_0 & 1 & 0 & 0 & 0 & -u_0x_0 & -v_0x_0 \\ u_1 & v_1 & 1 & 0 & 0 & 0 & -u_1x_1 & -v_1x_1 \\ u_2 & v_2 & 1 & 0 & 0 & 0 & -u_2x_2 & -v_2x_2 \\ u_3 & v_3 & 1 & 0 & 0 & 0 & -u_3x_3 & -v_3x_3 \\ 0 & 0 & 1 & u_0 & v_0 & 1 & -u_0y_0 & -v_0y_0 \\ 0 & 0 & 1 & u_1 & v_1 & 1 & -u_1y_1 & -v_1y_1 \\ 0 & 0 & 1 & u_2 & v_2 & 1 & -u_2y_2 & -v_2y_2 \\ 0 & 0 & 1 & u_3 & v_3 & 1 & -u_3y_3 & -v_3y_3 \end{pmatrix} \begin{pmatrix} m_0 \\ m_1 \\ m_2 \\ m_3 \\ m_4 \\ m_5 \\ m_6 \\ m_7 \end{pmatrix}$$
$$= \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

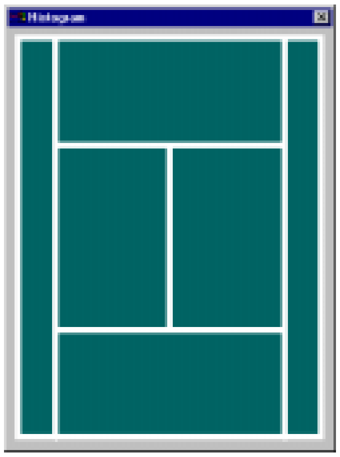
After we calculate what m0-m7 are, we have linear functions to direct map the coordinates for the perspective transform.

User Interface

We created a user-friendly interface to our tracking application in Windows. The application is MFC based executable, and it allows both automatic and manual tracking of features. It also allow the user to select a region where s/he wants features to be extracted, this initializes the searching algorithm (KLT). The application also pops up a 2-D bird-view graphical representation of the tennis court, and a hit-histogram can be generated according to hit counts of areas in the tennis court. Currently, the tracking algorithm is unable to track a tennis ball in compressed video format (i.e.: avi), this is due to compression artifacts (such as interlacing) and also to the fact that the tennis ball is too small and it moves too fast.

Below is a few screeshots:





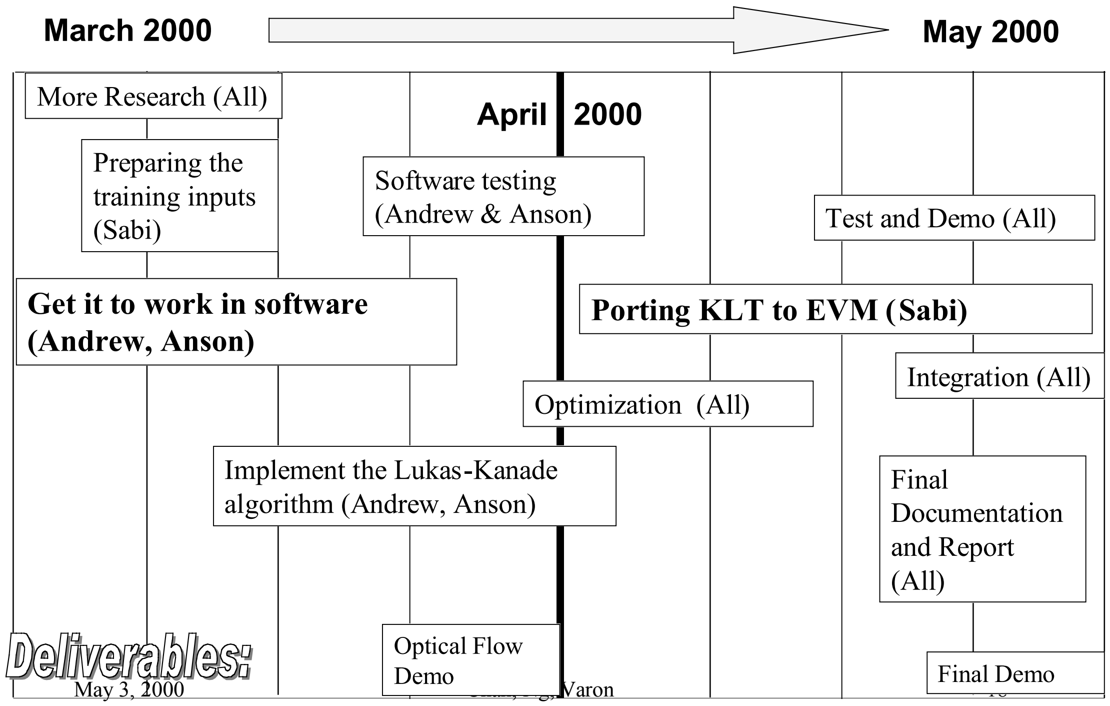
Conclusion

Object tracking can provide useful application such as analyzing performance of tennis players, tracking of ball in a ball game, etc. Object tracking using the KLT tracker is very accurate, nonetheless it is computation-intensive. This prevents us from being able to track in real-time. Also, video compression artifacts such as interlacing makes tracking more difficult. Also, in some cases, such as tennis, the ball moves too fast and is too small for the tracker to handle.

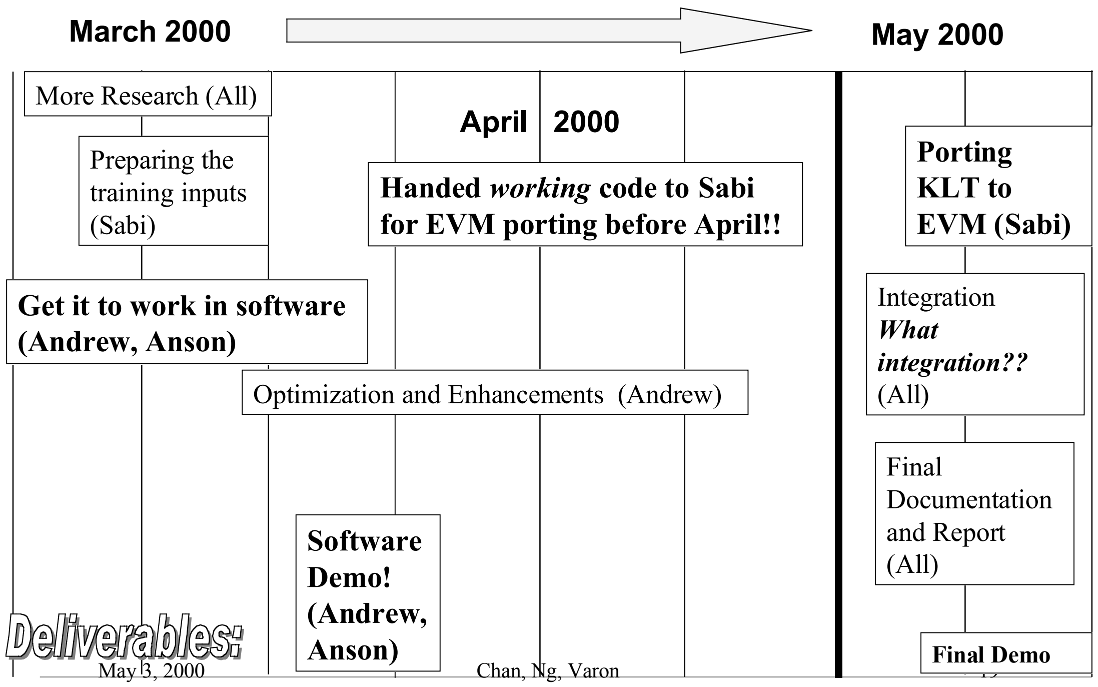
Future work on this project could be to improve the tracking algorithm, for example, maybe implementation Kalman Filtering instead of using the KLT tracker. Kalman filtering guarantees the minimum mean-square-error (MMSE) estimate, and it is very easy to compute. Also, the fact that the Kalman filtering is an iterative filtering approach makes it attractive to this application. Also, trying to feed in uncompressed video (maybe in YUV format) to ObjTrack and see if it would be able to track the tennis ball or other small fast-moving object would give us more insights as to what are the bottlenecks of the architecture.

From past experience in group project, we learned that it was a good idea to split up the work clearly, so that each member would be fully responsible for his or her part. In this project, it was proven that this is not always a good idea, if one or more of your members are not up to the level of complexity of the project. We regret not being able to get the EVM part to work, it was a lesson of organizational behavior, or “group think”. I guess in the future it would always be wise to underestimate one’s ability, that we wouldn’t put ourselves at risk at any time. The software version of ObjTrack, however, totally meet the initial goals we set up for this project. We were able to accurately track the object, and we also added enhancements to it. The perspective transform is extremely accurate because we actually went ahead to compute the matrix inverse by doing the LU decomposition. All in all, we have all learned much from this project, and it was by all means a very present experience.

Roadmap and Deliverables as ***Planned***


March 2000
May 2000
More Research (All)
April 2000
Preparing the training inputs (Sabi)
Software testing (Andrew & Anson)
Test and Demo (All)
Get it to work in software (Andrew, Anson)
Porting KLT to EVM (Sabi)
Integration (All)
Optimization (All)
Implement the Lukas-Kanade algorithm (Andrew, Anson)
Final Documentation and Report (All)
Deliverables:
Optical Flow Demo
Final Demo
May 3, 2000
Chan, Ng, Varon

Actual Timeline


March 2000
May 2000
More Research (All)
April 2000
Preparing the training inputs (Sabi)
Handed *working* code to Sabi for EVM porting before April!!
Porting KLT to EVM (Sabi)
Get it to work in software (Andrew, Anson)
Integration *What integration??* (All)
Optimization and Enhancements (Andrew)
Final Documentation and Report (All)
Software Demo! (Andrew, Anson)
Deliverables:
Final Demo
May 3, 2000
Chan, Ng, Varon

References

[1] Birchfield, Stan. "KLT: An Implementation of the Kanade-Lucas-Tomasi Feature Tracker". 1998. http://vision.stanford.edu/~birch/klt/ (2 February 2000).

[2] 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.

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