A Real-Time Video Stabilizer Based on Linear-Programming[*]

Moshe Ben-Ezra   Shmuel Peleg   Michael Werman
Institute of Computer Science
The Hebrew University of Jerusalem
91904 Jerusalem, Israel
Email: {moshe, peleg, werman}@cs.huji.ac.il

Abstract

Real-time video stabilization is computed from point-to-line correspondences using linear-programming. The implementation of the stabilizer requires special techniques for (i) frame grabbing, (ii) computing point-to-line correspondences, (iii) linear-program solving and (iv) image warping. Timing and real-time profiling are also addressed.

1 Introduction

Real time video stabilization can be performed robustly on a regular PC. The stabilization system described here works on a single CPU PC, at a frame rate of 10-15 320×240 color frames per second,

The stabilization minimizes an L_1 metric ($\sum |a_i - b_i|$) of point to line distances using linear programming [3]. The advantages of this approach with respect to accuracy and robustness are discussed in [2]. In this paper we describe the real-time implementation issues of this video stabilization.

1.1 Steps in the Stabilization

Image stabilization is performed using motion computation from point-to-line correspondences. The motion parameters are recovered from these correspondences by minimizing on L_1 error measure using linear programming. The algorithm has the following steps:

Point selection - A set of points from the first image is selected. These points are used for point-to-line correspondences and should have the following properties: (i) Uniform distribution across the image. (ii) Be on strong edges with different orientations.

Line correspondence - For each point in the first image finding one or two possible lines in the second image.

Linear Programming - Solving for the motion parameters by minimizing an L_1 error measure is done using linear-programming. Special considerations are required for real time performance.

Warping - The stabilized image is computed by warping the input image according to the recovered motion parameters. This step involves all the pixels and therefore it needs to be very efficient.

Two important aspects of any frame rate system include frame grabbing and real-time profiling, these aspects are also discussed.

2 Frame Grabbing

At first glance it seems that all frame grabbers do the same job - they get a frame from the camera and place it in the computer's memory for processing. However, several factors related to the frame grabber have tremendous affect on the systems performance Among them are:

2.1 Double Buffering

Grabbing a frame takes a long time ($1/30 - 1/25$ sec). To save this time the grabber should work concurrently with the computer's CPU. To do that the grabber should use double buffering. The frame grabber continuously grabs frames into one of two alternating buffers while the computer CPU is processing the previously grabbed frame. To avoid busy-waiting the grabber should be able to signal the computer upon frame grabbing completion. The scheme is given by:

[*]This research was funded by DARPA through the U.S. Army Research Labs under grant DAAL01-97-R-9291. It was also supported by Esprit project 26247 - Vigor

```
1. Get Frame (i=0) into "current buffer"
2. loop:
3.     Wait (non-busy) for Get Frame(i) to complete
4.     *Concurrently* Get Frame(i+1) into "other buffer"
5.     Process Frame(i)
6.     Toggle buffers
7. end loop
```

2.2 Internal Memory

A frame grabber that works concurrently with the computer must save the video data as it is being digitized. This will often clash with the CPU memory access. To avoid this clash the frame grabber should have internal memory and either a smart DMA controller or an internal bus which allows the CPU to access one buffer while the grabber is accessing the other buffer.

2.3 Compression

Some frame grabbers are designed mainly for video editing. These grabbers usually compress the video input on the fly. If such grabbers are used for frame rate application, it is important to verify that the frames are not compressed by the hardware and then decompressed by the software driver using most of the CPU time!

2.4 Color Modes

Color frame grabbers are designed to provide one or more color video formats. It is important to set the grabbing format to the format that best match the software requirement. For example, when an application requires brightness it is best to set grabber output to YUV, and then directly use the Y (brightness) channel. The driver is also important, for example, if the driver always converts the grabber output into RBG format it causes redundant conversion $YUV \rightarrow RGB \rightarrow brighness(Y)$.

2.5 Hardware and Software used

The stabilizer was implemented on a $300M_h$ PC equipped with a Matrox Meteor frame grabbing card. The frame grabber was controlled using Matrox MIL software library using asynchronous mode and double buffering. The operating system used was Windows-Nt4.0

3 Point Selection

Selected points should be distributed uniformly over the image, they should be on strong edges, and these edges should include both vertical and horizontal orientations. The following algorithm is used for point selection. It is based on a heuristic that vertical edges are easily detected by horizontal search and vice versa.

1. An initial set of $M \times N$ points is selected on a regular fixed grid across the image. These points will be referred to as "black" or "white", corresponding to a checkerboard pattern.
2. All "black" points search a narrow horizontal rectangle for the strongest vertical edge in that rectangle. All "white" points search a narrow vertical rectangle for the strongest horizontal edge in that rectangle.
3. A subset of the $2K$ best points (strongest gradients, K white, K black) are selected from the $M \times N$ points using two adaptive thresholds one for "black" and one for "white". Each threshold is increased if more than $K + \epsilon$ points passed it and decreased if less than $K + \epsilon$ passed it. Note that a large $\frac{M \times N}{2K}$ ratio will select only the best but they may be distributed non-uniformly across the image.

The test values were: $M = N = 9, K = 30, \epsilon = 3$ which yields an initial set of 81 points. After initialization of several frames, the threshold is stabilized and as long as consecutive frames are similar (which is an assumption of the stabilization process) approximately $54 - 66$ points are used for the stabilization. An illustration of the search pattern and of the detected points is given in Figure 1

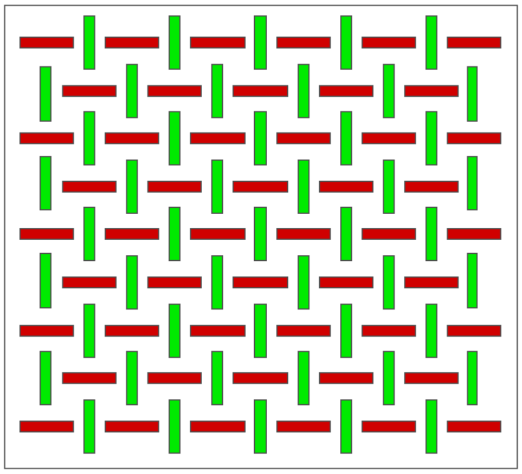
4 Point to Line Correspondence

Finding point to line correspondences consists of two steps (i) computing a similarity surfaces. (ii) Detecting lines (ridges) within the these surfaces.

4.1 Computing Similarity Matrices

For each selected point (x_i, y_i) in Frame F_q, a similarity matrix S_i, of size $R \times R$, between F_q and F_{q+1} is computed using SSD (sum of square difference).:

$$S_i(m,n) = \sum_{k=-v}^{+v} \sum_{l=-v}^{+v}$$
$$(F_q(x_i + k, y_i + l) - F_{q+1}(x_i + m + k, y_i + n + l))^2$$

a)


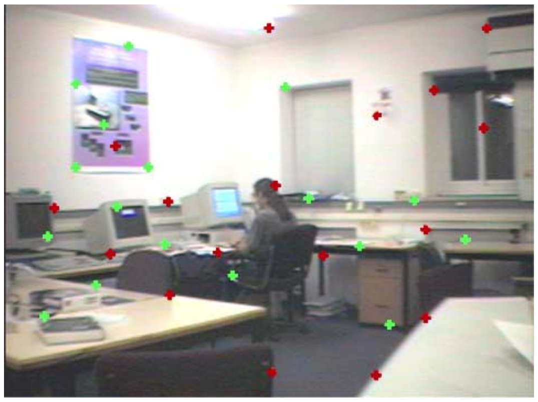
b)


Figure 1. Search pattern for selected points. a) Horizontal / vertical search areas. Strongest horizontal edge is sought for each vertical rectangle, and strongest vertical edge for each horizontal rectangle. b) The selected points in a real image.

The kernel size for the SSD is $(2v+1)\times(2v+1)$. The similarity matrix S is the displacement search area - its size R is the maximum detectable displacement. For numerical stability S is divided by $(2v+1)^2$ The test values were: $v = 3 \rightarrow kernel\ size = 7\times 7,\ \ R = 21\times 21$. Note that the number of pixels that were used is $(21+3)\times(21+3)$ to get a complete kernel cover at the edges of S.

4.2 Lines from Similarity Matrices

Two lines are detected for each similarity matrix S, using the weighted Hough transform [4]. Each line L in the Hough transform space is designated by the angle between L and $X - Axis$ and by the distance of L from the origin. Since the size of S is $R\times R$, the range of possible distances is $[-R/2..R/2]$ and the angular resolution is $180/2R$ degrees. These boundaries allow the setting of a lookup table for a fast Hough-transform. The weights used for the Hough transform are the values of similarity matrix S. The first detected line is the line with the largest bin in the Hough space which is the strongest ridge in the similarity matrix S. The search for a second line starts with an angle shift of 90^o from the first line, this ensures perpendicular lines in the case where the similarity matrix is symmetric. Figure 3 shows a similarity matrix, the detected lines, the approximated surface and an Hough space example.

5 Motion parameter recovery using Linear-Programming

Stabilization is done by computing a global 2D motion model and warping the image backwards according to the computed model. Common linear models include;

translation - two parameter model: 2D horizontal and vertical motion.

similarity - four parameter model: translation, scale and rotation (Z-axis)

affine - six parameter model: similarity and shear

homography - eight parameter model, true projective transformation of a plane.

For simplicity we will demonstrate in this paper motion recovery using linear programming from point-to-point correspondences and a similarity model. Point-to-line correspondences which are more robust since they are less sensitive to aperture effect, point-to-lines correspondences as well as homography model recovery can be found in [2].

Motion parameters are recovered from point-to-point correspondence by solving the system $p' = Mp$ where p, p' are corresponding points (in homogeneous coordinates) and M is the recovered model. Usually the model is recovered from an over determined system by minimizing the error in the least-square sense. This approach is very sensitive to outliers such as moving objects within the scene. Linear-programming's contribution to this scheme include:

Global optimum with no initial guess - Linear programming is guaranteed to find the global optimum and no initial guess is required.

L_1 **- minimization** - which is more robust to outliers than the least-square minimization.

additional linear constraints - with linear programming it is easy to constrain the recovered parameters to predefined limits.

Linear programming is also useful for a special type of correspondence - point-to-polygon correspondence [1]

5.1 The Linear Program

A linear program in standard form is given by:

$$\begin{aligned}&Min: c^t x\\&\text{subject to :}\\&Ax = b\\&x \geq 0\end{aligned}$$

Where: x is the variable vector (the unknown), $x_i \geq 0$. x are constant weight vector, $Ax = b$ are the constraints. $c^t x$ is called the objective function.

The constraint $x_i \geq 0$ can be bypassed by substituting every occurrence of x_i with $(x_i^+ - x_i^-)$, $x_i^+, x_i^- \geq 0$

We can always make sure that the constraint $Ax = b$ is met by adding slack variables $z, z_i^+, z_i^- \geq 0 : Ax + (z^+ - z^-) = b$.

Since all variables including the slack variables z are non-negative, and if:

$$\begin{aligned}&\forall i, z_i^+ = 0 \vee z_i^- = 0 \qquad (1)\\&\text{then}\\&(z_i^+ + z_i^-) = |z|\end{aligned}$$

Enforcing the condition 1 is difficult. However if the objective function is: $min : \sum_i (Z_i^+ + Z_i^-)$ then for this objective function the condition is always met (at the optimum). Our new linear programming system becomes:

$$\begin{aligned}&min : \sum_i (z_i^+ + z_i^-)\\&\text{subject to :}\\&Ax + (z^+ - z^-) = b\\&x \geq 0\\&z^+, z^- \geq 0\end{aligned}$$

which is the known solution for solving an over-constraint system $Ax = b$ using the L_1 metric. Note that when the minimum is found the z variables contain the residual errors which can be used for segmentation.

5.2 Example

The recovery of similarity model from point-to-point correspondence is given by:

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} \begin{bmatrix} a & b & e \\ -b & a & f \\ 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \qquad (2)$$

and the linear program is given by:

$$\begin{aligned}&min : \sum_i (z_i^+ + z_i^-)\\&\text{subject to :}\\&+(a^+ - a^-)x + (b^+ - b^-)y + (e^+ - e^-) + (z^+ - z^-) = x'\\&-(b^+ - b^-)x + (a^+ - a^-)y + (f^+ - f^-) + (z^+ - z^-) = y'\\&a^+, a^-, b^+, b^-, e^+, e^-, f^+, f^- \geq 0\\&z^+, z^- \geq 0\end{aligned}$$

5.3 Implementation issues

Since the constraints $Ax + (z^+ - z^-) = b$ are easily met by setting $x = 0, z = b$, which is also a "basic feasible solution (BFS)" a single phase simplex algorithm can be used to solve the problem. Since finding the first BFS (and, in general, determining if the problem is feasible and bounded) can be as hard as finding a solution, providing a BFS can reduces time approximately by half.

For a small number of constraints, simplex implementation which uses a dense matrix (i.o. sparse matrix) is more efficient. This matrix is allocated once and reused. The efficiency of the simplex algorithm for dense matrices can be dramatically increased by using floating point MMX for pivoting (not yet implemented).

The current implementation of the simplex is a self-written "by-the-book" primal algorithm with dense matrix. The only optimization used was loop-unrolling.

6 Warping

Warping is done over the full image. In-order to do this step efficient MMX is used. To avoid holes in the target image, backward warping is done using bi-linear interpolation. (bi-quadratic is too time consuming).

7 Stabilization Examples

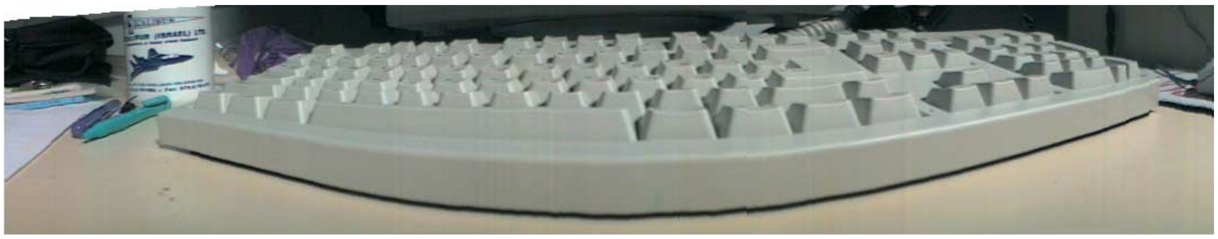
Constructing a panoramic image from a sequence of images requires a good stabilization of every frame. Figure 2 shows two panoramas that were created in real-time using the image stabilizer. Both panoramas were created in a difficult environment.

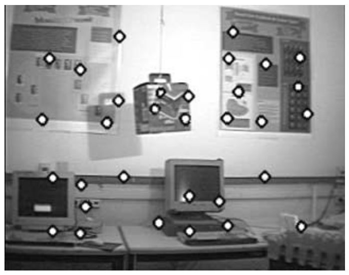
8 Profiling

An essential part of the optimization process of a real-time system is profiling. Optimization should be performed only when real performance data is obtained, as there are

Code	Parent Code	Description	Total Time (sec)	Percentage	Average time (msec)	Number of calls
0	-1	Everything Else	33.86	9.81%	1.36	1
1	0	Next Frame	64.72	18.75%	15.56	4160
2	0	Locate Points	3.12	0.90%	0.75	4156
3	0	Hough Lines	76.52	22.17%	18.41	4156
4	0	Solve LP	73.72	21.36%	17.74	4156
5	0	Warp	78.85	22.85%	18.97	4156
6	0	Display	14.32	4.15%	3.44	4160

Table 1. *Profiling results. Elapsed time was: 345.107 sec, 12 Fps.*

a)


b)


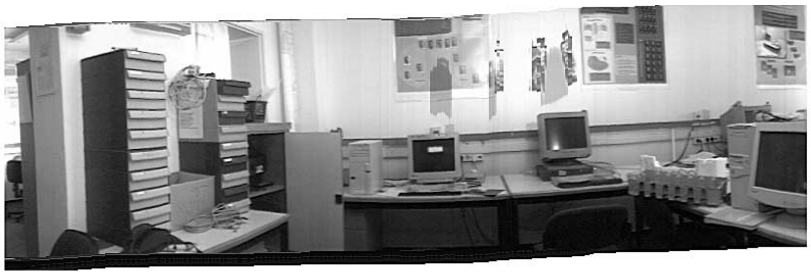
c)


Figure 2. Mosaicing examples. a) A panoramic image that was created in a very close range causing strong distortion of the input images. b) Point selected for motion computation. Four of the thirty points are located on the moving pendulum. c) Panoramic image that was created while a pendulum was swinging. The alignment was not affected by the outliers.

always surprises... A real-time system profiler should have a very low overhead and allow pin-point profiling. Common profilers are usually too heavy to be used in a real-time system (probabilistic profilers are more efficient but they require a long execution time). Table 1 shows the profiling result of the stabilization program. Appendix A describes the two pages long profiler code that was used. This profiler is both very simple and very efficient.

References

[1] M. Ben-Ezra, S. Peleg, and M. Werman. Efficient computation of the most probable motion from fuzzy correspondences. In *IEEE Workshop on Applications of Computer Vision (WACV98)*, 1998.

[2] M. Ben-Ezra, S. Peleg, and M. Werman. Real-time motion analysis with linear programming. In *Int. Conf. on Computer Vision*, 1999.

[3] H. Karloff. *Linear Programming*. Birkhäuser Verlag, Basel, Switzerland, 1991.

[4] M. D. Levine. *Vision in Man and Machine*. McGraw-Hill, NY, USA, 1985.

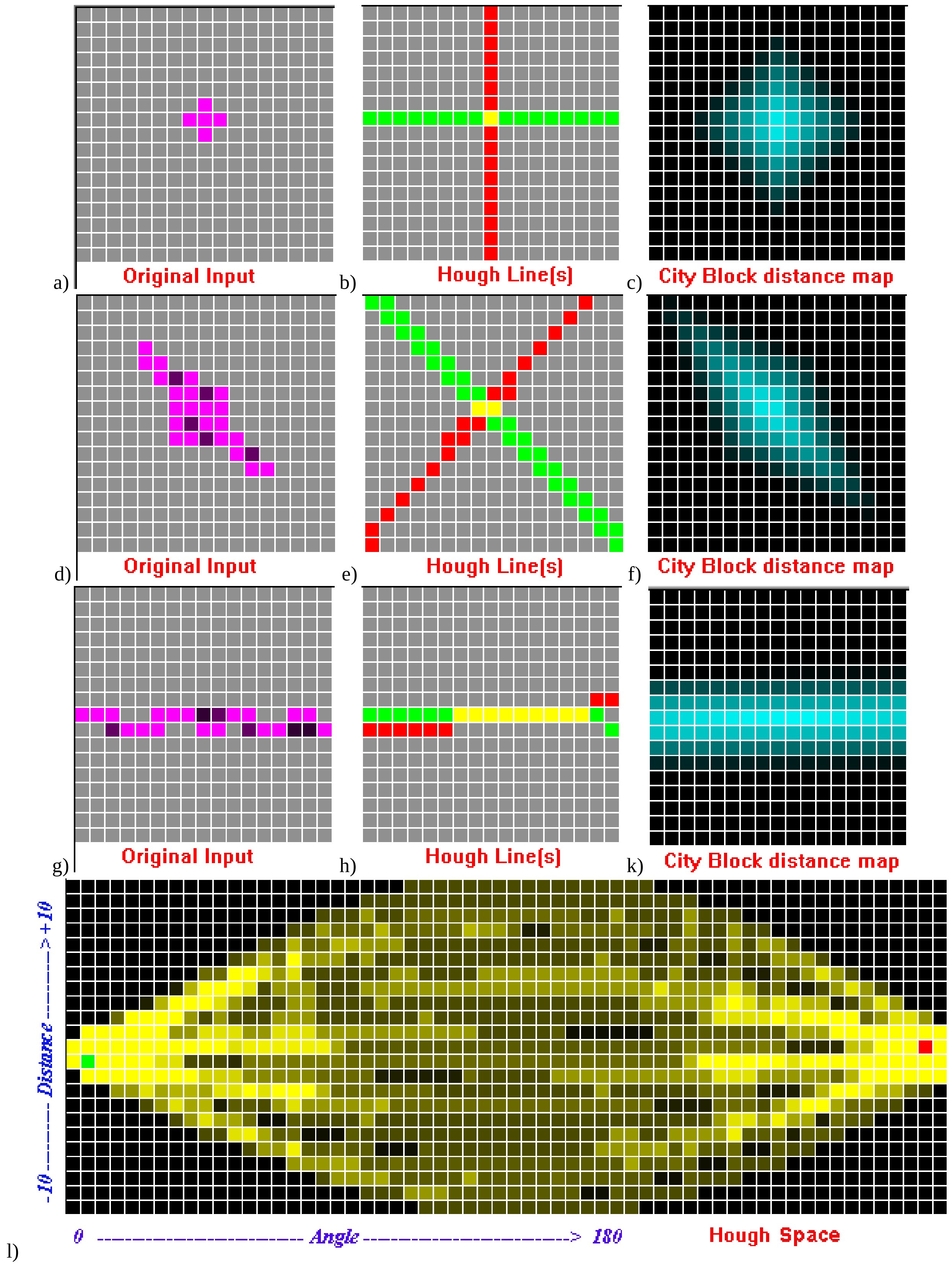


Figure 3. Basic classes of line detection a) Isolated point similarity surface matrix. b) Detected lines for (a). c) Approximated surface of (a) d) Oval shaped similarity surface matrix. e) Detected lines for (d). f) Approximated surface of (d) g) Line shaped similarity surface matrix. h) Detected lines for (g). k) Approximated surface of (g) l) Hough space. Red and Green dots are the detected lines locations.

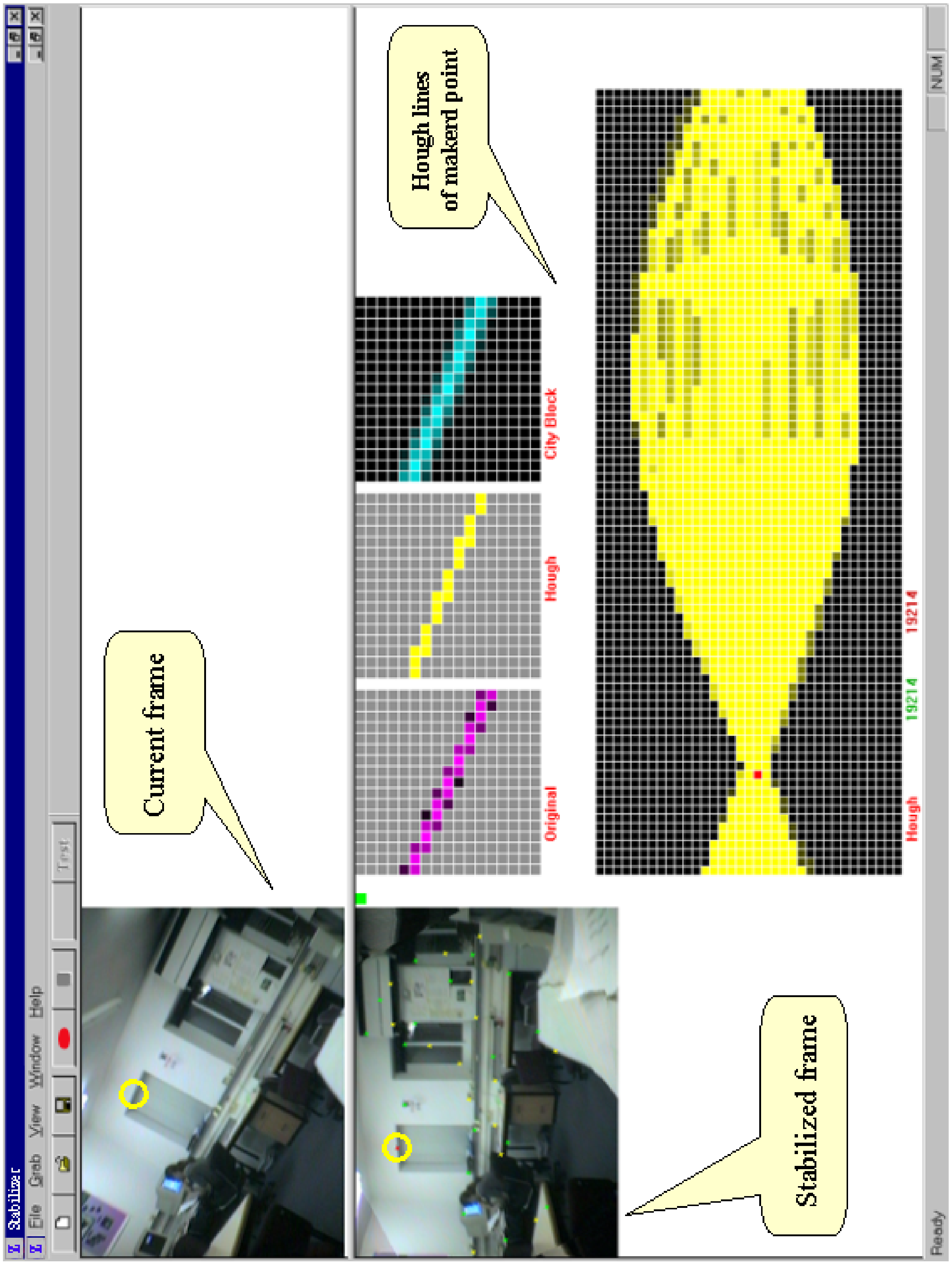


Figure 4. Application layout screen. Top left: direct un-stabilized image, one of the selected points is marked in yellow ring. Bottom left: stabilized image. Bottom right: Similarity matrix, Hough lines, City-Block distance approximation matrix and Hough space for the marked point.

A A two pages long profiler

The real-time profiler records entry and exit time intervals which are set by the programmer. These intervals can be nested, in fact all intervals are nested within the first interval which is recorded by the profiler itself. The profiler provides the following information:

code - the numeric code of the interval.

parent - the numeric code of the parent interval

description - the name/description of the interval

Total Time - Accumulated Time (without sibling intervals time) for the interval.

percentage - percentage of the elapsed time.

average time - average time per call.

calls - total number of entries to the interval (not including return from siblings).

The data is recoded using the START_PROFILE, END_PROFILE, ENTER(code), LEAVE(code) macros (windows version) and the recorded data is analyzed using a stack. Source listing follows.

A.1 Timing Macros (PC WINDOWS)

The following macros should be added to the program to be profiled. profiling starts at START_PROFILING and stoppes a STOP_PROFILING. ENTER(code) and LEAVE macros are used to bound sections timing. ENTER and LEAVE can be nested.

```
/* ============================================
 * ============= Timing Macros (PC WIN) =========
 * ============================================*/

#ifdef PROFILE

FILE *logfp, *recfp;
static _timeb tbuf;

#define START_PROFILE { \
    recfp = fopen("recording.dat","wb");\
        assert(recfp != NULL); \
    ENTER(0); \
}

#define STOP_PROFILE  { \
    LEAVE(0); \
    fclose(recfp); \
}

#define ENTER(x) { \
    _ftime(&tbuf); \
    fprintf(recfp,"E %d %d %d\n",(x),tbuf.time, tb uf.millitm); \
}

#define LEAVE(x) { \
    _ftime(&tbuf); \
    fprintf(recfp,"L %d %d %d\n",(x),tbuf.time, tb uf.millitm); \
}
```

```
#else

#define START_PROFILE
#define STOP_PROFILE
#define ENTER(x)
#define LEAVE(x)

#endif
```

A.2 Section codes and names

The following table defines the names of each section for the profiler.

```
/* ==============================================
 * ============= File: names.h ==================
 * ==============================================*/

#define CODES 5

char *name[] = {
/* 0 */ "Code 0",
/* 1 */ "Code 1",
/* 2 */ "Code 2",
/* 3 */ "Code 3",
/* 4 */ "Code 4",
/* 5 */ "Code 5",
};
```

A.3 profiler

The following is the off line profiler report program. It reads the recoding file from the standard input and produces a small report to the standart output.

```
/* ==============================================
 * =============File: Profiler.c ================
 * ==============================================*/

/* Simple Real-Time Profiler
 * =========================
 * Usage: profile < recording.dat
 *
 * written by moshe ben ezra - the hebrew university of jerusalem
 * www.cs.huji.ac.il/˜moshe
 */

#include <stdio.h>
#include "names.h"

#define STACK_SIZE  100
#define MAX_CODES 50

main()
{
  static int stack[STACK_SIZE], sp;     /* Nested Calls stack and stack pointer */

  double
    ts, te,                             /* Start time, End time      */
    t1, t2,                            /* Interval time var s       */
    sum[MAX_CODES],                    /* sum time intervals        */
    dt;                                /* Temporary delta time var  */

  long
    sec,msec,                          /* Time from input file      */
    calls[MAX_CODES],                  /* Call counters             */
    parent[MAX_CODES];                 /* Parent code               */

  int idx,i;                           /* Function index, loop i    */
  char code[2];                        /* Enter Exit code           */

  /*
   * Initialization
   */

  for(i=0; i<MAX_CODES; i++){
    sum[i] = calls[i] = parent[i] = 0;
  }

  for(i=0; i<STACK_SIZE; i++){
    stack[i] = 0;
  }

  ts = 0;
  sp = 1;                              /* 'Enter' always leaves the  */
  parent[0] = -1;                      /* the previsous level        */

  /*
   * process data file
   */

  while(scanf("%s%d%d%d\n",&code,&idx,&sec,&msec) == 4){
    t2 = (double) sec + (double) msec / 1000.0;         /* current time               */
    te = t2;                                            /* update end time            */
    switch (code[0]){

      /*==================*
       * Enter new section
       *==================*/
    case 'E':
      if (ts == 0) ts = t1 = t2;        /* At init prev interval = 0 */
      /*
       * leave prev level
       */
      dt = t2 - t1;                    /* Update Previous level interval   */
      sum[stack[sp-1]] += dt;          /* time (but not call counter)      */
      /*
       * enter new level
       */
      if (idx != 0)
    parent[idx] = stack[sp-1];         /* Save parent code                */
      stack[sp++] = idx;               /* Save section index              */
      t1 = t2;                         /* Update start time               */
      break;

      /*======================*
       * Leave current section
```

```
       *======================*/
    case ’L’:
      if (idx != stack[sp-1]){          /* Coherence check                */
        fprintf(stderr,"Stack Mismatch !\n");
        exit(1);
      }

      dt = t2 - t1;                     /* Update time and calls counter   */
      sum[stack[sp-1]] += dt;
      calls[stack[sp-1]] ++;

      /*
       * Return to previous level
       */
      sp --;
      t1 = t2;                          /* Start new interval */
      break;
    default:
      fprintf(stderr,"Code Mismatch !\n");
      exit(1);
    }
  }

  dt = te - ts;                         /* Elapsed time */

  /*
   * Outpout simple report
   */
  printf("Duration: %g sec\n",dt);
  for(i=0; i<= CODES; i++)
    printf("Code: %2d  Parent: %2d (%-20s): %5.2f sec.  %6.2f%%      Avr: %5.2f msec  calls: %d\n",
       i,parent[i],name[i],sum[i],sum[i]/dt*100.0,sum[i]*1000.0/calls[i],calls[i]);
}
```