Development of Generic Algorithms for Random Access Machine Vision

Adrian Trenaman and David Vernon

National University of Ireland, Maynooth, Co. Kildare, Ireland.
Email: trenaman,dvernon@cs.may.ie
WWW home page: http://www.cs.may.ie

Abstract. In this paper we describe our investigations into random access machine vision sensors. We begin with a brief description of a new CMOS imaging sensor with random access capabilities, and then detail the design of a software toolbox to yield the best results from this technology. Four applications of random access vision are presented: two industrial inspection tasks demonstrate the efficacy of the random-access approach, while real-time tracking of features in one and two dimensions demonstrates the utility of real-time image information. This paper then forms a ground-up description of a new approach to industrial machine vision.

1 Random Access Machine Vision

This paper describes the development of random access machine vision algorithms, work carried out as part of the ESPRIT Project 20577 *RAMAP*, described previously in [2]. Random access vision sensors, also known as *active pixel* sensors, provide access to individual pixels at high speed, and are not bound to the CCIR frame rate of 25 frames a second. Many industrial machine vision tasks utilise only a small percentage of the image information available: it is proposed that by furnishing the developer with tools to access only those pixels of interest in the image then the cycle time from data-acquisition to information-extraction will be significantly reduced.

The active pixel sensor developed in the RAMAP project provides the developer with a high-speed, high-resolution acquisition device. To achieve the highest acquisition speeds, acquisition must be ordered appropriately to reduce the number of slow address settings. This has lead us to develop high-level acquisition primitives to yield optimal acquisition speeds. These acquisition primitives are described in Section 2. The data obtained through these primitives can then be be analysed using one of a set of *analysis* primitives described in in Section 3. We provide demonstrations of the primitives in action in Section 4, applying the algorithms to visual inspection of ball-grid-arrays and chip leads, and tracking of features with one and two degrees of freedom. Our initial experiments were carried out using a simulated random access device, however recent results with the sensor itself are very encouraging.

eration in the design of the acquisition primitives was whether a signal should carry its own location information for use in further processing. Consider a line extracted from an image, and stored as an array of intensities. Subsequent processing reveals an threshold crossing, thus indicating the presence perhaps of an edge, between the 7^{th} and 8^{th} intensity reading. The question now is: where is the corresponding position in the image? Determining this involves knowing from where the signal was extracted. However, the problem is further complicated when one must consider that the signal may have been extracted via sub-sampling at discrete intervals or through an elaborate algorithm such as a DDA line extraction algorithm for arbitrary lines. Further, diagonal lines are sampled with non-uniform inter-pixel spacing. The solution we choose is to allow the extraction of two kinds of signal: one with *location information*, the other without. We define the following:

```
typedef int Intensity;
typedef Intensity *Signal;

typedef struct TaggedIntensity {
  int x, y;
  Intensity val;
} TaggedIntensity;

typedef TaggedIntensity *TaggedSignal;
```

Thus, a `Signal` is an array of `Intensitys` (which are just integer values), while a `TaggedSignal` is an array of `TaggedIntensitys`, where each `TaggedIntensity` is a `struct` containing an `Intensity` and two integer co-ordinates indicating the X and Y co-ordinates of the pixel it refers to.

The RAMAP acquisition, processing and analysis primitives can thus have three different variants, depending on whether they work directly on the sensor, a `Signal` or a `TaggedSignal`. Primitives working on either of the latter are suffixed with either `S` or `TS`.

2.3 Pixel Acquisition

The following primitives provide pixel access to the sensor:

```
Intensity getPixelS(int x, int y);
TaggedIntensity getPixelTS(int x, int y);
```

Both are optimised in that they only set the Y register if necessary, that is, if the current value of the Y address register is correct then no call to `yyy()` will be made.

RAMAP Primitive Operation	Time
xxx()	1.1μs
yyy()	1.7μs
zzz()	1.9μs

Table 1. Typical timings for fundamental acquisition primitives

2 Acquisition Primitives for Random Access Machine Vision

In this section we describe the high level acquisition primitives used in random access sensing. The most basic acquisition primitives are presented in Section 2.1. In Section 2.2 we describe the fundamental data structures used in our software toolbox, and in Section 2 we describe our high-level acquisition primitives. Each primitive is described briefly and its signature is given in the form of a C-style function heading.

2.1 Basic Acquisition Primitives

The fundamental acquisition primitives are `xxx()`, `yyy()` and `zzz()`, where `yyy()` sets the Y address line, `xxx()` sets the X address line, and `zzz()` performs the acquisition. The following constraints are imposed: if the Y line is already set to the correct value, then it need not be set for consecutive acquisitions on the same line. If the Y line is set to a new value, then the X line must be set again before the acquisition. The length of time required to execute each of the fundamental primitives in software are given in Table 1.

The cost of setting the Y line is almost twice the time to set the X line, so this scheme favours strategies that minimise the number of Y address settings.

To ascertain the penalty of the slow addressing on the Y line, an analysis was performed of pixel acquisition strategies while performing a 2D convolution over an input image. Caching of previous pixel values was incorporated in the analysis, and in all cases it was found that true random access was highly inefficient. Pixel accesses must be ordered to minimise the number of slow acquisitions: an optimal acquisition strategy was identified, based on scan-line algorithms borrowed from the computer graphics literature [1]. These scan-line algorithms allow the optimum pixel acquisition strategy to be derived automatically for arbitrary lines and rectangles, and can be extended to support circles and arbitrary shapes described by polygons. The conclusion from this study revealed the need for higher level optimised acquisition strategies, described in the next two sections.

2.2 Data Primitives

A *signal* is a subsection of the image extracted through a random access primitive: these signals can be either one or two-dimensional. An important consid-

2.4 Line Acquisition

The following primitives allow acquisition of lines from the sensor. Note that as well as the arbitrary line extraction functions (taking two end points) we have provided optimised extraction algorithms for extraction along the horizontal and vertical direction.

```
int getLineS   (int x0, int y0, int x1, int y1, Signal s);
int getHLineS  (int x0, int x1, int y, int step, Signal s);
int getVLineS  (int x, int y0, int y1, int step, Signal s);
int getLineTS  (int x0, int y0, int x1, int y1, TaggedSignal s);
int getHLineTS (int x0, int x1, int y, int step,
                TaggedSignal s);
int getVLineTS (int x, int y0, int y1, int step,
                TaggedSignal s);
```

The parameters `x`, `y`, `x0`, `y0`, `x1` and `y1` have their usual meanings. Each of the functions return the length of the line acquired.

2.5 Window Acquisition

The following are provided to allow the extraction of windows of arbitrary size from the sensor:

```
void getWindowS   (int x0, int y0, int x1, int y1,
                   Intensity **win);
void getWindowTS  (int x0, int y0, int x1, int y1,
                   TaggedIntensity **win);
```

3 Processing & Analysis Primitives

For signal processing and analysis we provide image smoothing operations in the form of convolution with Gaussian signals, and general edge detection algorithms using either thresholds or convolution with Laplacian-Gaussian filters. These are typical machine vision techniques, and are described more fully in [3]

3.1 Smoothing Through Convolution

We perform signal smoothing to reduce the effects of noise by convolving the signal with a suitable Gaussian signal. We generate the signal using the following function:

```
void generateGaussianS(float sigma, Signal gau, int *n);
```

where `sigma` controls the width of the Gaussian, `gau` is a preallocated `Signal`, and `n` is set to the length of the signal. The formula used to generate the Gaussian is

$$G(t) = e^{-t^2/\sigma^2} \tag{1}$$

Where t ranges from 0 to the length of the Gaussian signal. The length of a Gaussian signal for a given `sigma` can be determined by using the function

```
int lengthGaussian(float sigma);
```

The signal can then be convolved using either of the `convolveSignal()` functions:

```
void convolveSignalsS (Signal inputSignal,
                       int inputSignalLength,
                       Signal filter, int filterLength,
                       Signal outputSignal, int *outputSignalLength

void convolveSignalsTS(TaggedSignal inputSignal,
                       int inputSignalLength,
                       Signal filter, int filterLength,
                       TaggedSignal outputSignal,
                       int *outputSignalLength);
```

A signal can also be convolved using with a Laplacian-Gaussian filter signal to yield the 2^{nd} derivative of the smoothed signal. The Laplacian-Gaussian filter is easily generated using:

```
void generateLapGaussianS(float sigma, Signal lapgau, int *n);
```

The formula used to calculate each point of the Laplacian-Gaussian curve is

$$L(t) = e^{-t^2/\sigma^2}(1 - 2t^2/\sigma^2) \tag{2}$$

Where t ranges from 0 to the length of the Laplacian-Gaussian signal. The length of a Laplacian-Gaussian filter for a particular `sigma` can be found using:

```
int lengthLapGaussian(float sigma);
```

3.2 Edge Detection Through Convolution

We define an edge as those parts of the signal where the 2^{nd} derivative of the signal crosses zero, i.e., the points of inflexion of the signal. Every edge is described by a `struct` called a `transition` which contains both location information and a flag indicating the direction of the transition, either positive or negative. The constants

```
#define POSITIVE_TRANSITION 1
#define NO_TRANSITION 0
#define NEGATIVE_TRANSITION -1
```

are defined accordingly. The definition of a transition is:

```
typedef struct transition {
  float x;
  float y;
  int direction;
} transition;
```

We can calculate the 2^{nd} derivative of the signal by convolving with a Laplacian-Gaussian filter, such as that created using `generateLapGaussianS()`. The form of the edge detection functions for `Signals` and `TaggedSignals` are:

```
int getEdgesS (Signal inputSignal, int inputSignalLength,
               Signal lapGau, int lapGauLength,
               transition *e,
               int eLength);

int getEdgesTS(TaggedSignal inputSignal, int inputSignalLength,
               Signal lapGau, int lapGauLength,
               transition *e,
               int eLength);
```

The functions take a preallocated array of transitions, `e`, whose length is specified by `eLength`. On leaving the function, `e` will contain a number of transitions describing the position of the edges in the object. The function returns the number of transitions found.

3.3 Automatic Threshold Detection

Suitable thresholds for edge detection can be found by using a edge detection algorithm to calculate the edges in a signal, and choosing the average pixel value at these edges as a threshold. We provide the following function to create appropriate thresholds:

```
int detectThresholdS (Signal inputSignal, int inputSignalLength,
                      Signal lapGau, int lapGauLength);

int detectThresholdTS(TaggedSignal inputSignal, int inputSignalLength,
                      Signal lapGau, int lapGauLength);
```

Both functions find thresholds by convolving a user defined Laplacian-Gaussian (generated by `generateLapGaussianS()`) signal with either a `Signal` or a `TaggedSignal` and finding the average intensity of the edge pixels.

3.4 Edge Detection with Threshold

An alternative and far more efficient way to calculate the edges is to use a threshold (perhaps created automatically in a calibration phase using `detectThreshold()`) and use the threshold crossings to determine the positions of the edges. The following functions perform this:

```
int getTransitions  (int *x0, int *y0, int x1, int y1,
                     transition *t,
                     int max_transitions,
                     int required_transitions,
                     unsigned char threshold,
                     int sub_pixel_accuracy);

int getTransitionsTS(TaggedSignal inputSignal,
                     int inputSignalLength,
                     transition *t,
                     int tLength,
                     unsigned char thresh,
                     int sub_pixel_accuracy);
```

The first accesses the sensor directly, and scans for transitions (threshold crossings) between the points $(x0, y0)$ and $(x1, y1)$. A preallocated array of transitions, `t`, of length specified by `max_transitions` is sent to hold the results, along with a number `required_transitions` dictating the number of transitions actually required. When this number is reached the algorithm terminates and `x0` and `y0` contain the location of the last pixel read. This function has the advantage that not necessarily all pixels in the line will be accessed: only those until the required number of transitions has been found. The function returns the number of transitions found.

The second form of the function is similar in functionality, but works on a previously acquired `TaggedSignal` and acquires up to `tLength` transitions. The number of transitions found is returned.

In both of the above functions, `sub_pixel_accuracy` is a flag determining whether the positions of the transitions should be calculated to sub-pixel-accuracy. When set to zero, the transition's location will have be accurate to the nearest pixel. When set to 1, a simple linear interpolation between the two transition pixels is used to give a threshold-crossing to sub-pixel-accuracy.

4 Using Random Access Primitives

In this section we present four applications of the random access vision toolbox. The first two demonstrate the effectiveness of pixel rather than frame-based access, and show the toolbox primitives at work in static industrial inspection tasks. The second two demonstrate the real-time aspect of the sensor through tracking features with one and two degrees of freedom. Each application is described in four parts: statement of problem, solution, results and issues arising.

4.1 Inspection of Ball Grid Arrays

Problem In this demonstrator we must locate and count the number of balls present on a ball-grid-array (BGA). The balls are metallic, and are accentuated through dark field illumination. A sample image is shown in Figure 1.

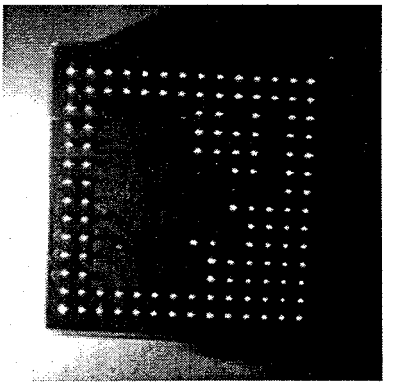


Fig. 1. Sample input for the BGA problem. The device is illuminated using strong dark-field illumination to accentuate the metallic surface of the balls.

Solution Our algorithm takes approximate positions for the top-left, top-right, bottom-left and bottom-right corners of the BGA, and scans using a given threshold for a positive-negative transition pair in the region of the these positions. The pairs found indicate the rise and fall of intensity due to the bright reflection of the corner ball. Now, having found the four corners, a line is scanned between the two left-hand side corners, and the centroids of the positive-negative transition pairs are calculated. This procedure is repeated on the right-hand side. Finally, corresponding centroids on the left and right centroid lists are used as points for further line scans, in which the positive-negative transition pairs are counted to determine the number of balls present.

Results The acquisitions and results for the algorithm are shown in Figure 2. The number of RAMAP primitive calls made are shown in Table 2.

The total acquisition time for the algorithm, using the software-timings, is thus 15.9ms, less than *half* the CCIR frame acquisition time.

Issues Due to the discrete sampling of the scan-line algorithm, it is possible that the sampled signal may only touch the side of a ball, and thus register an intensity less than the threshold for a valid peak in the signal. To illustrate this, Figure 3 shows a sample line scan from the BGA image. The first two peaks,

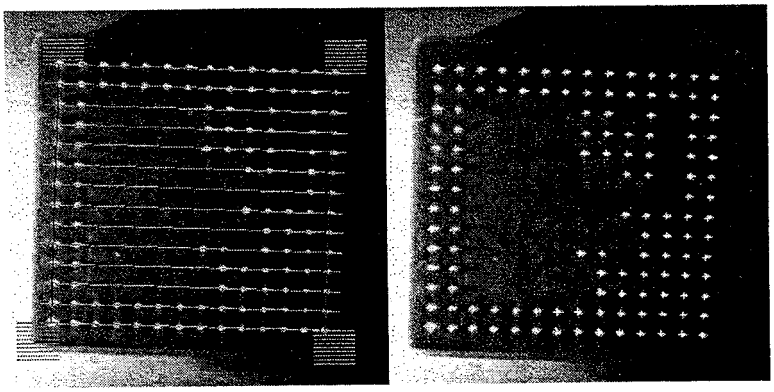


Fig. 2. Acquisition pattern and results for BGA inspection.

RAMAP Primitive	Number
Set X (fast) address register	4811
Set Y (slow) address register	863
Acquire Pixel	4809

Table 2. Acquisition characteristics for the BGA demo.

though clearly indicating the presence of balls, will be missed if a high threshold is used. Rather than use expensive edge detection to solve this problem, a new threshold for each line scan is calculated using

$$thresh = (max + min)/2 + (max - min)/6 \tag{3}$$

The initial scans to locate the corner edges are carefully ordered: for example, when seeking the top-left ball, scan lines run from left to right, and then top to bottom to ensure that the ball found is genuinely the top-left ball.

4.2 Inspection of Chip Leads

Problem In this problem, we set out to identify the positions of the leads of a chip: a sample image is shown in Figure 4. In this image, a healthy lead was removed and replaced with background information to yield an image of a flawed chip.

Solution Light field illumination was used to accentuate the metallic surface of the chip leads. The algorithm is similar to that of the BGA problem in Section 4.1, however, due to the poor quality of this image, yielding no global threshold for segmentation, edge detection was performed using Laplacian-Gaussian convolution filters.

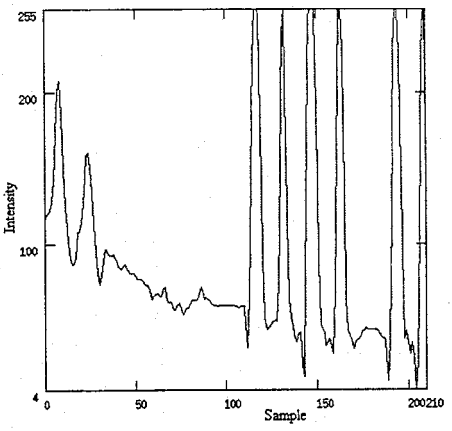


Fig. 3. Line sample from BGA image. The first two peaks may not be identified by a high threshold.

RAMAP Primitive	Number
Set X (fast) address register	4573
Set Y (slow) address register	550
Acquire Pixel	4559

Table 3. Acquisition characteristics for the chip-lead inspection demo.

Our algorithm then takes as input estimates of the 4 corners, along with a value σ defining the size of the Laplacian-Gaussian filter. Line scans to detect the corner leads are performed using the generated Laplacian-Gaussian filter. When the corners have been located, a line sample is taken on the left and right hand side of the chip, and the leads are located and counted, again using the Laplacian-Gaussian filter to locate the edges.

Results Results and acquisition pattern for the algorithm are shown in Figure 5. The number of accesses and register settings for the algorithm is shown below in Table 3. The total acquisition time for the algorithm is thus 14.6ms, again less then half the frame acquisition time under the CCIR standard.

Issues Edge detection using threshold could not be performed on this image, due to the poor lighting and non-uniform background. As a consequence of this the lines extracted by the image must be longer at either end, to accommodate the start-up of the convolution process.

Fig. 4. Sample input image for the chip lead inspection problem

Further, it was found that edge detection during the initial corner-scan produced edges in areas where there was on lead, due to the noisy back-ground. To counter this, edge detection was only performed on those scanned lines where

$$max > average * 1.3$$

where max and average are the maximum and average intensities of the signal, respectively.

4.3 Tracking Features in One Dimension

In this example we have considered how the RAMAP sensor could be best used to perform high-speed tracking on easily segmented images. Tests were performed on a 100 frame animation of a bar moving rapidly back and forward in the field of view. We present two effective tracking algorithms, *window* and *smart* tracking.

Problem Track the two edges of a moving black bar against a white background. The input images where chosen for their easy segmentation with threshold.

Solution We present two solutions to this problem. Both involve an initial scan where the two edges of the bar are located. After this initialisation stage, velocity profiles of the edges are estimated so that at each frame an estimate of the edge's position is known. The estimate can be calculated using zero, first or second derivative information of the previous positions.

In the first solution, tracking is performed by scanning from left to right in the vicinity of the estimated location, that is a scanning-window of length l . This *window-scanning* algorithm gave good results, however, given a good estimate for the position it will still demand at least $l/2$ acquisitions to verify this. A

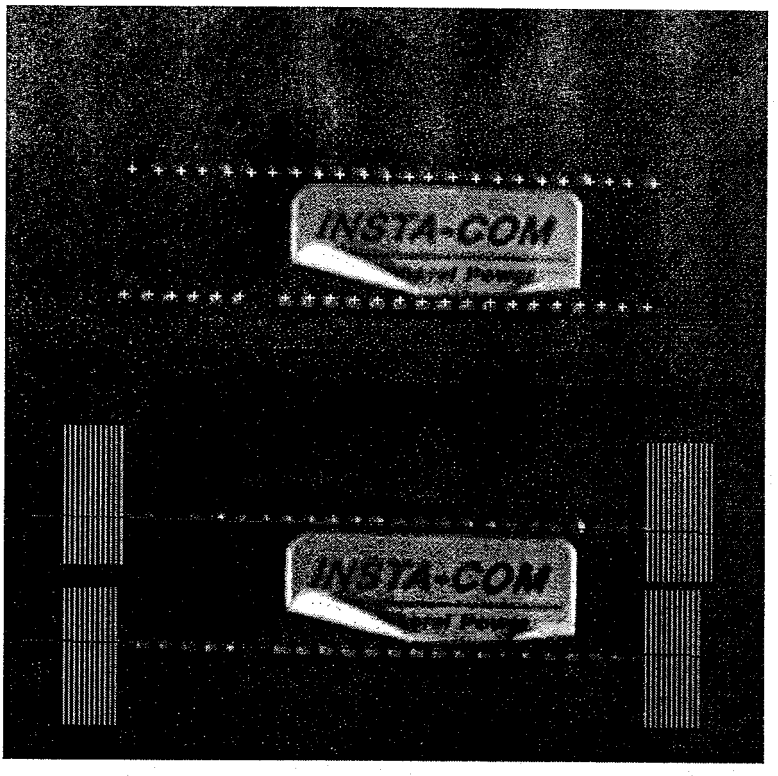


Fig. 5. Results and Acquisition Pattern for chip lead inspection demo

far better algorithm was devised which makes use of the value of the signal at the estimated location of the edge. For example, if the edge is caused by a negative transition, and the value at the estimate is greater than the threshold, then clearly scanning should proceed to the right. This approach we call *smart* scanning.

Results Both algorithms were implemented and tested to reveal the average number of accesses per frame. Acquisition pictures for the window and smart scanner are shown in Figure 6 and Figure 7 respectively.

The benefit of using first and second-derivative information in estimating the position of the edge was examined, and results are presented in Table 4. Window scanning was performed with a window of size 40. Note that these figures are for tracking *two* edges of the bar.

It is clear that tracking certainly benefits from 1^{st} derivative information, while there is no appreciable gain in using 2^{nd} derivative information. Using 1^{st} derivative information, the cycle time of the smart tracker for two edges is $33\mu s$.

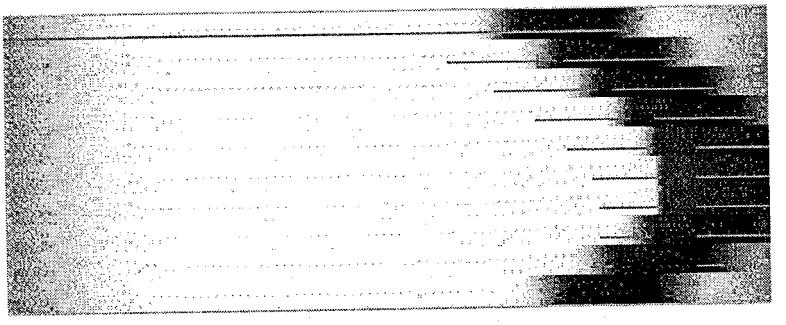


Fig. 6. Ten frames showing the acquisition pattern for the simple window tracker.

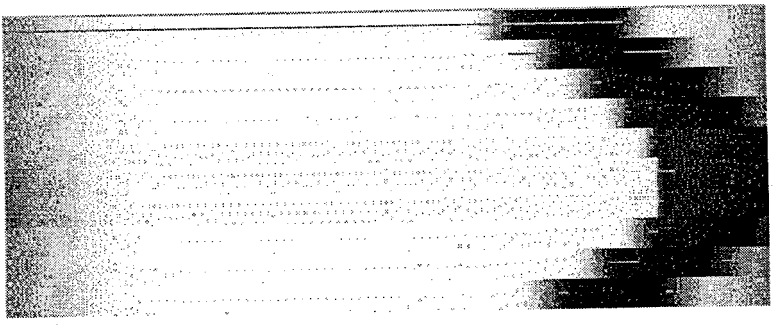


Fig. 7. Ten frames showing the acquisition pattern for the smart tracker

Experience with the real sensor The linear tracking demonstrator was implemented with the real RAMAP sensor, using the setup illustrated in Figure 8. The rotating drill bit moves a bar, which when viewed from above appears to move in simple harmonics motion from left to right and back again. The angular velocity of the drill bit was measured with a tachometer. For 10,000 track cycles, the window tracker (using a window size of 20) gave an average cycle time of 0.36ms, providing accurate position data at a rate of 2.8KHz. The angular velocity of the bit was 1100 RPM, the bar oscillating with period 18.3 Hz. Only 25 of 10000 scans failed, recovery performed by re-scanning the entire sample line for the edge. The *smart* scanner performed poorly with this setup, because

Location Estimation Method	Window Tracker	Smart Tracker
Previous Position	86	38
Previous Position & 1^{st} Derivative	85	11
Previous Position & 2^{nd} Derivative	85	11

Table 4. Results from the Window and smart trackers

the tracked object contains two very close edges, one positive and one negative. In the event that the position estimate lies outside these two edges, the smart tracker cannot function correctly, as was the case with this setup.

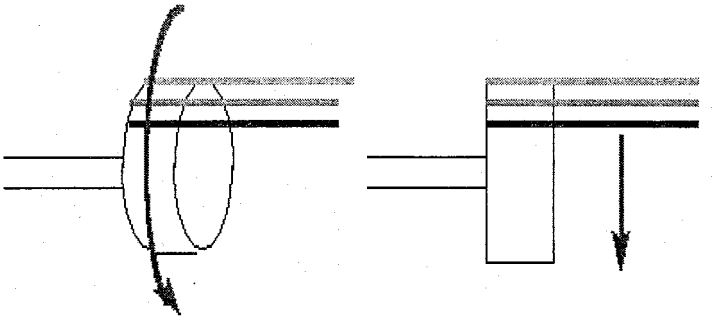


Fig. 8. A bar attached to a rotating drill bit, when viewed from above, appears to move back and forth in simple harmonic motion

4.4 Tracking Features in Two Dimensions

Problem Track a circular dark object against a bright background

Solution To get an initial estimate of the location of the blob, we sub-sample the image using a user-specified sub-sample width. This process yields a point somewhere within the blob. Linear scanning proceeds vertically up and down and horizontally left and right from this point, to acquire four edge points. By casting a horizontal from the left to right edge, and a vertical from top to bottom, we can use the midpoints of these lines to locate the centroid of the blob. This process is shown in Figure 9.

Results The algorithm relies on a good estimate of the blob position, i.e. an estimate that lies within the surface area of the blob. If the estimate is outside the blob, then the linear track will fail. In this event, the entire image is sub-sampled again to restart the tracking process. The number of RAMAP primitives used per cycle is shown in Table 5.

The results in Table 5 yield acquisition times of 905.1μs, 854.4μs and 975.4μs respectively.

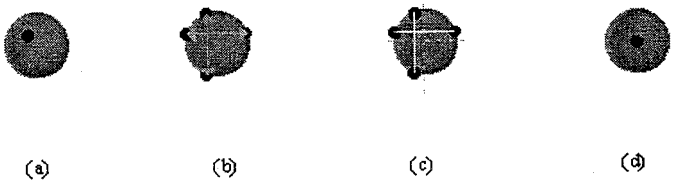


Fig. 9. Given a point within the blob, two line scans can be used to calculate a better estimate of the centroid.

Location Estimation Method	xxx()	yyy()	zzz()
Previous Position	265.7	63.5	265.7
Previous Position + 1^{st} Derivative	250.0	61.4	250.0
Previous Position + 1^{st} & 2^{nd} Derivative	289.3	63.12	289.3

Table 5. Average number of primitives called per cycle over a 100 frame movie.

Issues Due to the rapid motion of the blob, later images in the sequence exhibited a smearing around the blob in the direction of motion. This blurring effectively raised the intensity of the blob, and so, though the position estimate was correct, the original threshold could not detect the presence of the blob. Consequently, on many frames the tracking algorithm could not verify or refine the centroid.

We introduce a second local sub-sample scan for the blob within the region of the estimate, to seek a point within the area of the blob. Acquisition times using this algorithm are 768.3μs, 879.6μs and 691.5μs.

The cause of the problem, image blur due to motion, may be less of an issue when tests are performed on the RAMAP sensor itself. The existing test-set was extracted using CCD technology with the CCIR standard. The high-speed acquisition of the RAMAP sensor may reduce or eliminate the problem of blur: this has yet to be seen. In the event that blur cannot be eliminated, it may be possible to use the gradient of the blurred signal to direct search towards the blob, and thus turn the blur to our advantage. This is left for future work.

5 Conclusions

This paper has described the design of a toolbox of algorithms for use with random access vision sensors. Four applications of the toolbox were presented, each of which demonstrated the efficacy of the random access approach: the number of pixels required to solve each of the problems was small compared to the number of pixels available, and by acquiring only those pixels of interest we reduce significantly the time for data acquisition. The data acquisition bottleneck is a severe problem for high-speed industrial vision tasks, and this paper proposes that random access sensing technology may be the solution.

References

1. J. Foley, A. Van Dam, S. Feiner, J. Hughes, and R. Philips. *Introduction to Computer Graphics.* Addison-Wesley, 1994.
2. Adrian Trenaman, David Vernon, and David Barry. An analysis of strategies to reduce computational complexity and processing time in industrial optical data processing and analysis. In *Proceedings of OEPE '96.* Optical Engineering Society of Ireland, 1996.
3. David Vernon. *Machine Vision - Automated Visual Inspection and Robot Vision.* Cambridge University Press, 1991.

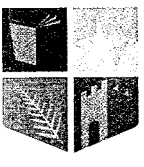
David Vernon (Ed.)

OESI–IMVIP '98

Optical Engineering Society of Ireland & Irish Machine Vision and Image Processing Joint Conference

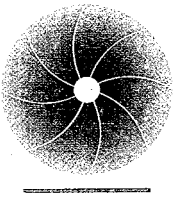
PROCEEDINGS

National University of Ireland, Maynooth
9th-10th September, 1998



NUI MAYNOOTH
Ollscoil na hÉireann Má Nuad

Cumann Innealtóireacht Optúla



THE OPTICAL ENGINEERING SOCIETY OF IRELAND