Real-Time Gray-Scale Video Processing Using a Moment-Generating Chip

R. L. ANDERSSON

Abstract—**A system for performing visual processing on gray-scale images in real time (60 Hz) has been constructed. The custom VLSI moment generator chip computes area, center of gravity, orientation, and size. An image preprocessor allows separate moments to be computed for separate regions. A standard set of buses allows new processing elements to be easily added. The system is expected to find application in real-time sensor-based electronic assembly and in automated inspection and registration tasks. A simple application to a real-time visually servoed robot task is summarized.**

I. Introduction

THE IMPLEMENTATION of a large class of potential robot applications is currently not feasible due to the lack of high-speed three-dimensional image-processing systems. Current assembly-task design relies on a person's manipulative skill, their ability to detect and correct for uncertainty in the environment, and the ability to concurrently inspect the work in progress. The latter two abilities, and perhaps even the first, rely on the ability of the person to sense the environment using visual and tactile perception. To automate many tasks, we must in effect emulate a person's perceptual skills.

Robotic vision systems must operate in "real time." By a real-time system, I mean one that does its job in a period of time that is at least partially determined by external constraints, not just a system that is fast. The need for real-time operation is often created by physical processes, such as when the solder has melted or when the object has moved out of reach of a robot, and by scheduling constraints such as assembly lines. The inability of a robot system to meet such constraints is an outright bar to the usability of the system.

Given that a system can do a specific job in the required time, there are important economic reasons for improving the operating rate. A system that runs twice as fast costs half as much per unit operation.

In the near term, we would like to build systems that are capable of processing simplified scenes at the rate necessary to use the data for real-time robotic control. Tasks often involve observing some feature in an image and causing some physical device to come to a specific position in relation to it for "handling." Examples include picking up objects and registration tasks. Sometimes, the position of the controlled device may not be accurately known, so it may be desirable to simultaneously monitor the position of both object and end effector. An alternative is to put the camera on the robot and measure the relative displacement directly.

In the long term, we would like to construct systems capable of extracting information useful for manipulation and inspection from gray-scale images of three-dimensional scenes in real time. Such systems will be an important component of larger systems for monitoring an entire three-dimensional workspace for collisions, for verifying machine operation, for assuring quality, and for performing detailed assembly jobs.

A. Current Vision Systems

Commercial vision systems primarily use binary processing, typically by thresholding and run length compressing the image to reduce the amount of data before storing it in a fairly general-purpose processor. Once read in, the data is processed for many tenths of a second before some decision is made. The canonical basis for these schemes is [6]; there are many current commercial imitators.

Another class of fast computer vision algorithms is "image processing" algorithms. These algorithms take an image as input and produce an image as output. Most often, the processing is done by a convolution operator used to smooth the data or find edges. Special-purpose hardware may be built to perform the processing in pipelined fashion on the video stream (for an example, see [3]). From the perspective of robotics applications, such algorithms are not directly useful, as they do not reduce the amount of information which needs to be processed, although they may simplify subsequent feature extraction operations. Reducing the amount of data to be processed without eliminating essential image content is the fundamental problem in processing images in real time.

Advanced research-oriented systems (under the heading of "image understanding") typically read a gray-scale image into a frame buffer before processing it for several seconds, minutes, or even hours. Although these systems are slowly and steadily advancing in capability, their formidable processing requirements have severely limited their application to the robotics environment.

B. Systems Approach

The approach described here trades a lot of the flexibility of the image understanding methods for real-time operation. In the future, we would like to be able to adapt more algorithms from the world of image understanding to real-time processing.

A typical system configuration is shown in Fig. 1. The modules fall into two categories: image processors in which an

Manuscript received November 1, 1984; revised May 6, 1985.

The author is with AT&T Bell Laboratories, Room 4G608, Crawford's Corner Road, Holmdel, NJ 07733, USA.

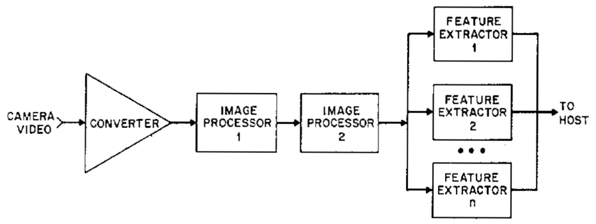


Fig. 1. Typical series/parallel system.

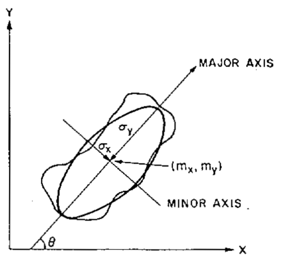


Fig. 2. Parameters of blob extracted by moments.

image is input and another image output; or "feature extractors" in which an image is input and some set of parameters output. The image processors are connected serially, whereas the feature extractors are in parallel—similarly, perhaps, to the human brain.

Each individual processing module is implemented on a single Multi-Bus board (Intel Corporation). The modules are unified by two types of buses (not including the Multi-Bus) whose signal locations and functions have been standardized. One bus contains a video signal digitized to eight bits of gray scale with 256 × 240 pixels per frame, 60 frames per second. The second bus is used to return a computer-selected video stream from one of the modules to the digitizer board for output on a monitor. Additional boards may be easily incorporated by designing them to utilize the busses correctly.

II. Moment Generator

Moments have been in use in computer vision for some time [7], [8], and their use in physics and statistics goes back much farther. The equation defining the moments $M^{m,n}$ of an intensity array $a_{i,j}$ is

$$M^{m,n}=\sum_{i,j} a_{i,j} i^m j^n, \tag{1}$$

where $m + n (m, n \geq 0)$ is the order of the moment, i is the column, and j is the row.

The zero through second-order moments are sufficient to find the area, center of gravity, angle to major axis, and standard deviation along major and minor axes for an object, approximating the object as an ellipse, as shown in Fig. 2. These quantities are directly useful in picking up an object, or in guiding further visual processing, for example. Second and higher-order moments may be combined to form "invariants" which are used to characterize an object for purposes of discriminating among members of some set of objects.

The amount of time required to compute gray-scale moments has hindered their use. On a VAX 11/780 with floating-point accelerator, a direct calculation of the zero through second-order moments of a 256 × 256 image takes 6.5 s. Straightforward hardware implementations of moment calculations require large numbers of multipliers, accumulators, registers, and supporting logic. Hybrid electrooptical approaches are possible [2], but suffer the same problems, in terms of accuracy, stability, and dynamic range, that are typical of analog computers. The moment generator system is intended to make the computation of moments easy enough for use as a new primitive for image examination.

The moment computation has been integrated onto a VLSI chip capable of computing a single zero through second-order moment of a gray-scale image in real time. Since there are six such moments, the moment processor module contains six chips. A number of techniques used to make the chip possible will be discussed below.

A. Power-Vector Generation

We consider moment generation as a dot product

$$M^{m,n}=\sum_{t} a_t p_t^{m,n}=(\bar{a}, \bar{p}^{m,n}), \tag{2}$$

where the elements of the vectors are in the same order as a normal TV scan, $t = i + 256j$. The element $p_{i,j}$ of $\bar{p}$ will be referred to interchangably with p_{i+256j}.

The equation defining $\bar{p}^{m,n}$ is

$$p_{i,j}^{m,n}=i^m j^n. \tag{3}$$

The element $p_t^{0,0}$ is one for all t. The first order moments require a counter for either x or y, depending on the moment. Apparently, a second order $\bar{p}$ requires two counters and a multiplier. We can write the next value of each second order $\bar{p}$ as a function of the previous one,

$$p_{i+1,j}^{2,0}=p_{i,j}^{2,0}+2i+1 \tag{4}$$

$$p_{i+1,j}^{1,1}=p_{i,j}^{1,1}+j \tag{5}$$

$$p_{i,j+1}^{0,2}=p_{i,j}^{0,2}+2j+1 \tag{6}$$

with special cases for top of screen and left margin. We can build an iterative $\bar{p}$ generator composed of a single counter, a shifter, an adder, some "and" gates, and a small control programmable logic array (PLA).

B. Bit Decomposition

We can decompose $\bar{a}$ as

$$\bar{a}=2^7\bar{a}_7+2^6\bar{a}_6+\cdots+\bar{a}_0. \tag{7}$$

If we substitute equation (7) into (2) and distribute, we obtain

$$M^{m,n}=2^7(\bar{a}_7, \bar{p}^{m,n})+2^6(\bar{a}_6, \bar{p}^{m,n})+\cdots+(\bar{a}_0, \bar{p}^{m,n}). \tag{8}$$

Computation of the dot products in (8) requires only 1 × n bit multiplication, which may be implemented by n "and" gates, where n is the number of bits in $\bar{p}$.

At the end of each frame, we must compute

$$M^{m,n}=2^7F_7+2^6F_6+\cdots+F_0 \tag{9}$$

where

$$F_k=(\bar{a}_k, \bar{p}^{m,n}). \tag{10}$$

Equation (9) can be evaluated only once per frame (60 times per second) using Horner's method of polynomial evaluation. The calculation is performed by the host processor which controls the system.

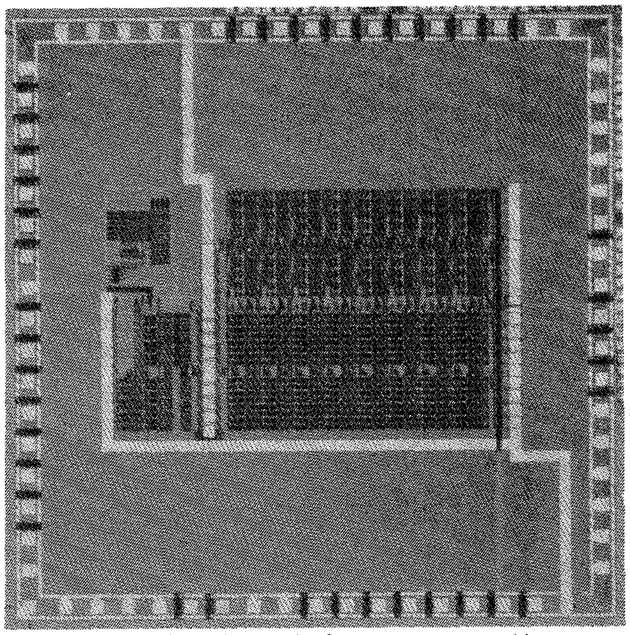


Fig. 3. Photomicrograph of moment generator chip.

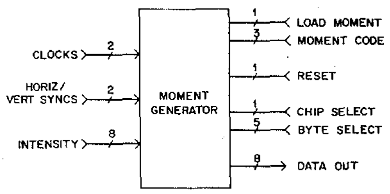


Fig. 4. Primary input/output paths for moment generator chip.

The F_k accumulators are identical, simplifying the layout of the chip. The decomposition used to obtain fast operation will be seen to provide significant flexibility.

III. Moment Generator Chip

The techniques described above allow a moment generating IC to be contructed. Internally, the chip (Fig. 3) contains a programmable logic array (PLA) for control and the $\bar{p}$ generator on the left side, and a row of eight F_k accumulators across the middle and to the right. On the far right is an output multiplexer. The chip was designed using the MULGA symbolic layout system [9]. The chip is fabricated in a TTL compatible 5 V 2.5 μm CMOS process and contains 10 214 transistors. The moment generator is packaged in a 40-pin ceramic DIP.

A high-level view of the chips is shown in Fig. 4. On one side, the chip connects to a digitized video source. The video is comprised of an eight-bit gray-scale value and a two-bit sync code. Data is input synchronously with respect to a two-phase clock. On the other side, the chip connects to a processor or other logic. Five address lines and one enable line are used to select one of 32 bytes on the chip for presentation on an "open collector" output bus. The processor may also reset the entire F_k accumulator array, or load a moment select code into an onboard latch.

Many devices have been fabricated with an acceptable yield. Moment generators have been tested to run at a 115 ns period, and may run even faster. The clock frequency must not drop below 500 Hz or the dynamic registers will fail.

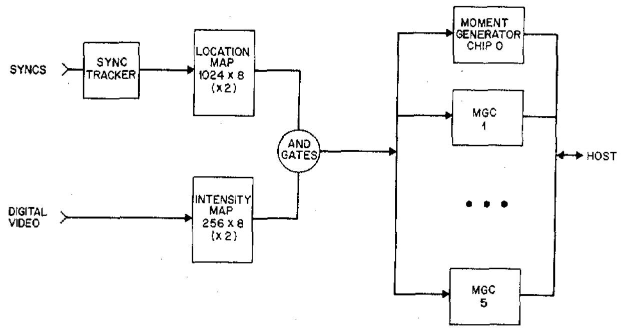


Fig. 5. Block diagram of moment generator board.

A. Multiple Regions

Taking the moments of an entire picture is not generally useful, since the moments of multiple objects are smeared together. Moments of each object must be taken individually.

Because the F_k accumulators are identical and independent, we can use the same chip to compute the moments of more than one region at a time, assigning each F_k accumulator to a single region (possibly more than one per region, however). Since the number of F_k accumulators is the same as the number of bits in the picture intensity, we must sacrifice bits of precision per region to accommodate additional regions.

Suppose we would like to compute the moments of three regions simultaneously, one region to one bit of intensity resolution, another region to five bits of intensity, and the last region to two bits of intensity resolution. Equation (9) would then be performed three times,

$$M_1^{m,n} = F_7 \tag{11}$$

$$M_2^{m,n} = 2^4F_6 + 2^3F_5 + 2^2F_4 + 2F_3 + F_2 \tag{12}$$

$$M_3^{m,n} = 2F_1 + F_0. \tag{13}$$

The intensity data applied to the chips must be specially formatted. A preprocessor works independently on the intensity for each F_k bit. A block diagram of the moment generator board, including the preprocessor, is shown in Fig. 5.

The intensity map converts intensity values to the desired precision and alignment. For example, the map may perform binary thresholding, intensity windowing, nonlinear response correction, or any combination of the above. The intensity map may be used to correct for scene illumination problems or changes.

The location map defines the region of activity of each F_k; a bit is on if that F_k is to be activated at that position on the screen. To reduce the size of the location map, and simplify the host's job, regions are quantized into 8 $\times$ 8 pixel blocks. For example, Fig. 6 shows a location map for computing the moments of three regions simultaneously to different intensity resolutions.

The ability to do multiple-moment calculations simultaneously is a consequence of the design of the moment generator IC and of the preprocessor. We desire as many F_k bits as possible, since we can track more objects, or get more accurate results, with each additional bit. We can get more F_k bits by adding more chips. If a lower frame rate per object is acceptable, we could process a different set of objects each frame.

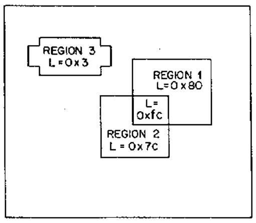


Fig. 6. Location map for finding moments of three different regions simultaneously.

To track a large number of objects, or to deal with a complex background or overlapping objects, a more complex preprocessor would be needed that could perform the necessary segmentation.

B. Region Finding

Implicit in the simultaneous computation of the moments of multiple regions is the *a priori* knowledge of where the regions are. This implies some type of region-finding process before multiregion tracking may begin. One method of doing this is to read a frame into the host's main memory, and perform a conventional region-finding operation on it.

A second method is to use moments themselves to find the regions. In this paradigm, the moments of the image as a whole are found and the principal axis identified. The image is then bisected perpendicular to the major axis, and the process is repeated. The bisection is terminated when the centers of gravity of two halves are close to the line of division. The intermediate representation is somewhat akin to quad-trees.

This algorithm has not been tried, but similar algorithms have been reported [5]. A substantial amount of experimentation will be required to get the right parameters and to make the process work.

C. Higher-Order Moments

Second-order moments describe location and orientation, in effect approximating the object by an ellipse. The orientation is degenerate; second-order moments will not tell us which end of a wrench has a handle on it. Similarly sized and shaped objects are not easily discriminated. Higher-order moments contain more information. If we compute all the moments, we know the entire shape of the object.

The $\bar{p}$ generator and register widths could be expanded to generate higher-order moments, but I do not think the increased functionality justifies the increased complexity.

I will describe a way of computing third-order moments using second-order hardware. Even higher-order moments may be calculated by recursive descent, at the expense of exponential time. Fifth and higher-order moments are probably best computed by the host.

We want to calculate

$$M^{m,n} = \sum_{i,j} a_{i,j} i^m j^n \tag{14}$$

where $m + n = 3$. Equation (14) may be rewritten as

$$M^{m,n} = \sum_i i(N_i^{m-1,n}) \tag{15}$$

for

$$N_i^{m,n} = \sum_j a_{i,j} i^m j^n. \tag{16}$$

Moments with $m = 0$ can factor j out symmetrically. We can partition i as

$$i = 2^7 i_7 + 2^6 i_6 + \cdots + i_0. \tag{17}$$

Substituting (17) into (15) yields

$$\begin{aligned} M^{m,n} = {} & N_0^{m-1,n}(2^7 0_7 + 2^6 0_6 + \cdots + 0_0) \\ & + N_1^{m-1,n}(2^7 1_7 + 2^6 1_6 + \cdots + 1_0) + \cdots \\ & + N_{255}^{m-1,n}(2^7 255_7 + 2^6 255_6 + \cdots + 255_0). \end{aligned} \tag{18}$$

We can redistribute as

$$\begin{aligned} M^{m,n} = {} & 2^7(N_0^{m-1,n}0_7 + N_1^{m-1,n}1_7 + \cdots + N_{255}^{m-1,n}255_7) \\ & + 2^6(N_0^{m-1,n}0_6 + N_1^{m-1,n}1_6 + \cdots + N_{255}^{m-1,n}255_6) \\ & + \cdots + (N_0^{m-1,n}0_0 + N_1^{m-1,n}1_0 + \cdots \\ & + N_{255}^{m-1,n}255_0). \end{aligned} \tag{19}$$

I define the inside of the parentheses as

$$Q_k^{m,n} = \sum_i \omega_i N_i^{m,n} \tag{20}$$

for k between 0 and 7, with $\omega_i = i_k$ so that

$$M^{m,n} = 2^7 Q_7^{m-1,n} + 2^6 Q_6^{m-1,n} + \cdots + Q_0^{m-1,n}. \tag{21}$$

We can compute (21) easily if we can do (20). Substituting (16) into (20) yields

$$Q_k^{m,n} = \sum_i \omega_i \left(\sum_j a_{i,j} i^m j^n \right) \tag{22}$$

$$Q_k^{m,n} = \sum_{i,j} \omega_i a_{i,j} i^m j^n. \tag{23}$$

The ω_k define a region for each $Q_k^{m,n}$. The region corresponding to the Q_7 term is the right half of the screen. The region corresponding to the Q_6 term is the right half of each half of the screen. The region corresponding to the Q_5 term is the right half of all the quarters of the screen, and so on down to the Q_0 term which is every other column.

Eight moment calculations may be weighted and summed to form the third-order moment. At worst, this will take eight frame times. If the region is small, or we are willing to sacrifice precision, or both, we can obtain the third-order moment even faster by computing the moments of several Q terms each frame time.

Basically, the technique is an extension of the assembly of binary moments to form gray-scale moments, as may be seen by examination of (9) and (21). The regions defined by ω_i in (23) are generated in hardware using a small amount of additional circuitry in the moment generator preprocessor.

D. Advanced Hardware Systems

One possible preprocessor is described in the earlier section on multiple regions; however, it is not the only possibility. The properties of moments can be exploited to achieve additional effects. The following are some ideas on what might be done in particular cases.

If we want to track more than eight objects, several sets of chips could be placed on one board. A hardware segmenter (region finder) can be used instead of a map to separate various objects. Some provision would have to be made for culling the small "noise" regions.

If the source of the data operates at a low frame rate, or if the resolution and thus the pixel rate is very low (say from a tactile sensor), a frame could be temporarily stored in a buffer and then run past a single chip several times in succession at a higher rate.

On the other hand, if we would like to process higher-resolution video (i.e., 512 × 512), then two sets of chips may be used. In the horizontal direction, each chip processes every other pixel. The two 60 Hz fields that comprise a 30 Hz frame are already interlaced conveniently. We compute two sets of moments each field for two fields, then mathematically combine the results to obtain the (exact) higher resolution moments at 30 Hz.

IV. Experiment

This section describes an experiment demonstrating the moment generator and robot system catching rolled ping-pong balls. Although this operation may not be immediately useful (except possibly to ping-pong ball manufacturers), it is intended to be illustrative of the techniques used for real-time tracking of objects, for example, components on conveyor belts, robots, and for processing data in real-time from cameras mounted on robots, and so on.

A. Hardware Setup

Three computers provide the computing power; one performs the vision processing, one controls the robot, and the third performs auxiliary debugging functions. All three computers contain a Motorola 68000 processor (SUN board, Pacific Microcomputers, Inc. PM68K), a SKY Computers, Inc. SKYFFP-M 5100 floating-point processor, and 1 MB of memory. The processors are connected by means of the S/Net [1], a high-speed interprocessor connect well suited for this type of task. An operating system provides multiprocessing and multitasking support for real-time programming [4]. A variety of UNIX-like functions are provided, including a multiprocessor version of a UNIX pipe for communication. A VAX host acts as a file server and software development machine.

An overhead vidicon TV camera connected to the moment generator system views a black table surface in the reach of a Unimation PUMA 260 robot. Ping-pong balls are rolled from the side of the table opposite the robot towards the robot. The robot carries a special purpose gripper for catching ping-pong balls not unlike a baseball glove.

The PUMA 260 robot is controlled by a 68000 processor, which replaces the DEC LSI-11 in the Unimation controller. By programming the 68000 appropriately, we achieve control of the robot kinematics. Servoing of the individual joints is performed by an existing microprocessor per joint.

A second TV camera watches obliquely from the side; it is connected to a commercial frame grabber, and stores a series of images for debugging purposes. Position and velocity profiles are also output on this display.

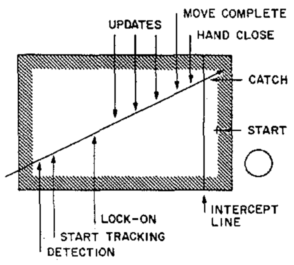


Fig. 7. Ball-catching sequence.

B. Algorithm

The system goes through an explicit series of events to catch each ball, as outlined in Fig. 7. The plan is to catch the ball at the intersection of its trajectory with a fixed line to the rear of the table surface (which may be reached by the robot).

Initially, the entire field of view of the camera except for the very bottom (where the robot is) is sensitive to incoming balls. When a ball first comes into the field of view, it is ignored for six frames, so that it may become totally visible. The program tracks the ball at a 60 Hz rate for 0.2 s (12 frames), and least squares fits a straight line to the trajectory. The fitted straight line is intersected with the intercept line to determine the catch point. The robot starts moving towards the catch point rapidly enough to get there in advance of the ball.

The vision system then "locks on" to the ball by building a small location map around it. Subsequent balls will be ignored until the catch sequence is complete. The ball is tracked as the robot moves, and each 0.1 s an update to the robot's target location is generated based on the ball's motion during the previous 0.2 s. This allows the system to compensate for nonlinear motion on the part of the ball, normally due to the table not being level and being warped, and also due to initial spin on the ball. The ball may even be made to bounce off obstacles during this phase and still be caught.

As the ball comes nearer to the hand, a number of events occur more or less asynchronously. When the ball is predicted to be too close to the hand, or when the area of the ball changes too much, visual tracking is discontinued, since otherwise the robot hand may be tracked, which causes very bad effects. At the catch point, the robot stops moving, and allows the ball to roll into the gripper. Without this strategy, and even sometimes with it, the robot hand knocks away a ball it would have otherwise caught. The hand is commanded to close before the ball gets to the hand because of the finite actuator delay. Otherwise, the ball can bounce back out of the hand before it shuts. Students of baseball should be familiar with this effect, as it indicates more damping in the hand would help. The relative timing of these events depends on some input parameters and the speed of the ball.

Once the ball is caught, it is thrown back along a fixed trajectory. It is possible to throw the ball in the direction from which it came, but this makes the possibility of injury to the operator much larger.

This algorithm is not necessarily optimal, but it is a reasonable one. Some balls that are theoretically catchable are missed due to incompleteness in the algorithm. The program works best on balls with constant speed and direction over the central part of the table.

A counterexample is a ball with a lot of backspin. The algorithm sticks with its original intercept time once it has computed it. If the ball slows down, the robot decides to "come out" and catch the ball. In some cases, the robot is not able to reach the position where it wants to catch the ball and in effect gives up, despite the fact that the ball continues rolling right on past the gripper. This problem is somewhat difficult to fix, largely because of the necessity of re-timing the robot motion, and also because of the interaction of the asynchronously scheduled events described previously, which might have already occurred and have to be undone.

C. Implementation

The ping-pong ball catcher is implemented by a three-processor complex, as described previously. The robot processor is the master; it starts the other two, connects to the user terminal, and serves as a switching point.

When the robot processor has decided to catch a ball, it sends a "go" message to the vision processor. When a ball has been found and is being tracked, the vision processor sends back a message giving its position and velocity. The arm starts moving, and the vision processor continues to send updates each 0.1 s. The robot processor sends the frame-grabber processor a message that says "grab a frame" at various points in the catch sequence.

Eventually the ball is caught (or missed), and the robot processor and vision processor go through an asynchronous hand shake to acknowledge the end of processing for that ball. The vision processor then sends stored debugging information (such as position, velocity, speed, and eccentricity) to the robot processor, which formats and forwards it to the frame grabber for generation of graphs. The frame grabber is then told to cycle through its stored images, and the system goes on to get ready for the next ball.

Robot Control Program—Snatch: The robot control program is a myriad of details: the interaction with other processors, coordinate system transformation, asynchronous event handling, error recovery, etc. Three specific capabilities of the robot programming system that are essential to the ability to catch ping-pong balls are: the ability to start a robot motion and continue processing while the move is performed; the ability to specify a fixed time for an operation to be completed in; and the ability to modify the target of a motion while it is in progress. None of these capabilities are generally found in commercial robot programming systems.

Moment Generator Program-Ogle: Visual tracking of the ping-pong balls is performed by the program "ogle." This program uses binary thresholding and dynamically generated location maps. Ball trajectories are determined by fitting centroids to a straight line on the fly. During the update period, a moving-window technique is used to smooth the data. The program is internally controlled by a type of finite-state machine (state + cycle count) which has proven to be worthwhile.

The general sequence of operations for getting some moments computed using this system is: 1) set up the intensity and location maps; 2) tell the chips which moments to compute; 3) wait for the start of a frame; 4) clear the moment accumulators; 5) wait for the end of the frame; 6) read out the moments from the chips; and 7) do something with the moments. Operations 4) and 6) must be done during the vertical retrace interval, about 700 μs.

One way of using the moment generator system is to implement the previously described sequence of operations as some subroutine to be called when appropriate. Because of the synchronization requirements, at most 30 sets of moments can be computed per second. For initial program development and experimentation this is sufficient.

For maximal utilization of the system, the operations sequence 1)–7) is pipelined such that both operations 6) and 4) (in that order) occur during each and every retrace interval. During the computation of one set of moments, compute the location and intensity maps for the next frame are computed. So that the updating of the maps does not interfere with the current computation, there are in fact two sets of maps; one map of each type is accessible to the processor at a time, while the other set is being used in the current computation. I refer to this duplication of maps as "shadowing," also known as ping-pong or swing buffers.

A simple method of making programs that operate in this pipeline fashion uses a loop which is traversed at frame rate, changing the "shadow" (which set of maps to use) bit at each pass.

A modified finite-state machine implements the tracking sequence. In state RESET, all of the accumulation registers are unconditionally clobbered. In the WAIT state, if there is not a ball showing, we go back to RESET. If there is a ball, we check to see if we have seen it for long enough, and if not continue waiting. If so, we go to TRACK state for the first 0.2 s of tracking the ball. TRACK mainly updates various accumulators. After the 0.2 s, the initial position and velocity are sent to "Snatch," and the system goes to state REFINE.

In REFINE, the checking on whether or not to continue tracking is a bit more stringent, but the most interesting part is the dynamic generation of a location map. The position of the ball is predicted ahead slightly, due to the shadow swapping, and a rectangular patch is generated about that position into the appropriate location map, as determined by the shadow bit.

After the ball is lost or "Ogle" is commanded to "release," the system enters PRINT state, where it sends debugging information out. Extensive "tanks" of data are kept from the real-time segment of operation.

D. Results

The ping-pong ball catcher is quite reliable for balls in the range of 0.5–0.75 m/s. Some balls rolling to the side of the table away from where the robot starts are knocked away. Very slowly moving balls are intentionally ignored, and slightly faster balls are sometimes missed because they can curve dramatically between the time they get too close to the hand to see and the time they are caught. At the higher end of the scale, balls may be caught fairly reliably up to the 1 m/s. range, depending on how close they are to the robot. The gripper, a low technology device made of door hinges, also contributes to the error rate by allowing some balls to bounce out. Balls have been caught at 1.3–1.4 m/s, but the robot is currently programmed not to go for balls moving that fast, as it tends to cause the robot to crash into things, such as the table.

The maximum speed of the robot itself is on the order of 1–2 m/s—not much faster than the ball. Faster balls could be caught by increasing the speed of the robot, or alternatively by increasing the field of view, and thus travel time, of the balls. By increasing the field of view, we decrease the accuracy of localization.

Note that people can move on the order of 10–15 m/s, and that a 100 mph fastball is moving at 45 m/sec; clearly there is a lot of range of improvement for the robot speed. Robot speed improvement is not just a matter of building a faster arm, but improving the control system as well.

V. Conclusion

Conventional systems look at an image with pixel examination as their primitive operation. Because there are so many pixels, processing takes a long time. The availability of the moment system means we can change our idea of what the primitive should be for looking at an image; rather than examining individual pixels, we can take a more holistic, gestalt view and examine whole areas of the picture. It may well be better to overkill with moments than use the gross underkill of pixel examination.

We were able to exploit the power of VLSI technology to implement a single special-purpose feature extracting element which is able to operate in real time. Moments are not the ultimate such operation and there is probably no single universal operator. Instead, we hope that a catalog of such high-performance vision processing chips will evolve. Very powerful vision processors might then be built as a handful of such chips.

The need to operate in three dimensions should govern the creation and selection of primitives for the next generation of systems. By using powerful image-examination primitives, we should eventually be able to do real-time image processing of complex three-dimensional scenes.

Acknowledgment

Many people contributed to this multi-disciplinary project; it is impossible to thank them all. Many thanks to the S/Net team, the MULGA team, and the rest of my department. J. F. Jarvis deserves special mention for providing the environment and impetus for this project to take place.

References

[1] S. R. Ahuja, "S/Net: A high speed interconnect for multiple computers," *IEEE J. Select. Areas Commun.*, vol. SAC-1, no. 5, p. 751–756, Nov. 1983.

[2] D. Casasent and D. Psaltis, "Hybrid processor to compute invariant moments for pattern recognition," *Opt. Lett.*, vol. 5, no. 9, p. 395–397, Sept. 1980.

[3] T. Fukushima *et al.*, "ISP: A dedicated LSI for gray image local operations," in *Proc. 7th Int. Conf. Pattern Recognition*, vol. 1, p. 581–584, July 1984.

[4] R. D. Gaglianello and H. P. Katseff, "Meglos: An operating system for a multiprocessor environment," in *Proc. 5th Int. Conf. Distributed Computing Systems*, May 1985.

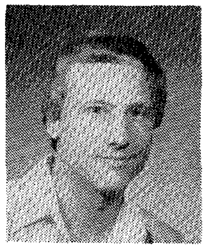
[5] L. Gibson and D. Lucas, "Spatial data processing using generalized balanced ternary," in *Proc. IEEE Comp. Soc. Conf. Pattern Recognition and Image Processing*, p. 566–571, June 1982.

[6] G. J. Gleason and G. J. Agin, "A modular vision system for sensor-controlled manipulation and inspection," in *Proc. 9th Int. Symp. Industrial Robots*, SME/RIA, p. 57–70, Mar. 1979.

[7] M. Hu, "Visual pattern recognition by moment invariants," *IRE Trans. Inform. Theory*, vol. IT-8, p. 179–187, Feb. 1962.

[8] A. P. Reeves and A. Rostampour, "Shape analysis of segmental objects using moments," in *Proc. IEEE Comp. Soc. Conf. Pattern Recognition and Image Processing*, p. 171–176, Aug. 1981.

[9] N. H. E. Weste, "Virtual grid symbolic layout," in *Proc. 18th Design Automation Conf.*, p. 225–233, June 1981.



Russell L. Andersson was born in Philadelphia, PA, on September 24, 1958. He received the B.S. and M.S. degrees in computer science from the University of Pennsylvania, Philadelphia, in 1980 and 1981, respectively.

He joined AT&T Bell Laboratories in 1981 and is currently a Member of Technical Staff in the Robotics Systems Research Department, Holmdel, NJ. He is concurrently working on his Ph.D. in robotics at the University of Pennsylvania.

Mr. Andersson is a member of Eta Kappa Nu and Tau Beta Pi.