Fast sensory motor control based on event-based hybrid neuromorphic-procedural system

T. Delbruck, P. Lichtsteiner

Inst. of Neuroinformatics, UNI-ETH Zurich

tobi@ini.phys.ethz.ch

Abstract—**Fast sensory-motor processing is challenging when using traditional frame-based cameras and computers. Here we show how a hybrid neuromorphic-procedural system consisting of an address-event silicon retina, a computer, and a servo motor can be used to implement a fast sensory-motor reactive controller to track and block balls shot at a goal. The system consists of a 128x128 retina that asynchronously reports scene reflectance changes, a laptop PC, and a servo motor controller. Components are interconnected by USB. The retina looks down onto the field in front of the goal. Moving objects are tracked by an event-driven cluster tracker algorithm that detects the ball as the nearest object that is approaching the goal. The ball's position and velocity are used to control the servo motor. Running under Windows XP, the reaction latency is 2.8±0.5 ms at a CPU load of <10% with a minimum observed latency of 1.8 ms. A 2 GHz Pentium M laptop can process at >1 million events per second (Meps), although fast balls only create ~30 keps. This system demonstrates the advantages of hybrid event-based sensory motor processing.**

I. INTRODUCTION

High speed vision is demanding if high frame rate traditional cameras are used. For example, at 1 kHz frame rate, a low resolution 128x128 monochrome image sensor (16k pixels) with 8 bits gray resolution (only 48 dB dynamic range) produces data at 16 Mbytes/s. A contemporary PC running at a realistic usable instruction rate of 10^9 operations per second can perform only 60 machine instructions per pixel per frame. At this frame rate useful processing is demanding even for the fastest contemporary PCs.

In this paper we describe the results of experiments in low-latency vision using an address-event transient silicon retina [1] as the input sensor, a standard PC as the processor, and a servo motor as the output. The general idea is that the retina produces only a small amount of informative low-latency data and this data is immediately useful for tracking objects. By forming a hybrid of this sensor with a computer and a servo motor, we can do experiments in fast sensory motor control that retain the advantages of both the spike event-based sensor and the flexible procedural computer. We were particularly interested in measuring latency in this hybrid architecture using a standard operating system on the computer. These preemptive multitasking operating systems like Windows or Linux are widely used on personal computers and increasingly on embedded systems and they greatly reduce development time for complex applications.

II. APPLICATION

In this example application, we built a soccer goalie robot (Fig. 1). This robot blocks balls shot at a goal using a single-axis arm. Although fast visually guided control has been studied for many years (e.g. [2]), previous work has used frame based image sensors and has often tracked objects based on distinctive color cues. Our approach uses pixel events rather than frames and these events represent scene reflectance change. This approach is inspired by the transient pathway of biological visual systems that motivates the design of the silicon retina.
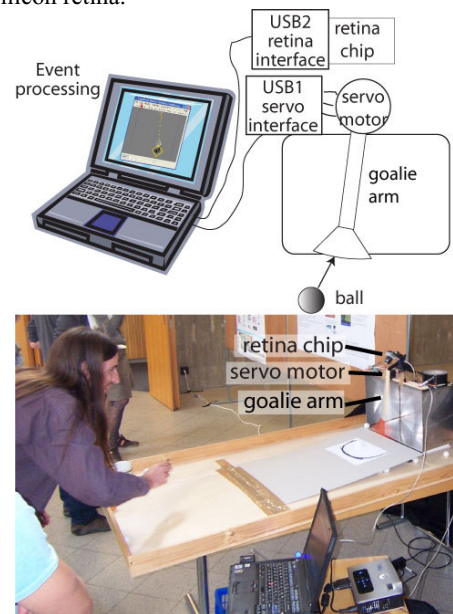

Fig. 1   System architecture and a photo of the setup, showing the placement of retina, servo motor and goal. The white balls have a diameter of 3 cm (smaller than table tennis balls) and are viewed against the gray cardboard ramp. The reflectance ratio between balls and gray area is about 1.3. The retina view extends out to just past the tape (80 cm).

A. Silicon retina vision sensor

In this section we summarize the properties of the temporal contrast silicon retina vision sensor. [1] (Fig. 2 and Table 1). Each address-event [3] that is output from the chip is the address k of a pixel and means that the log intensity at pixel k changed by an amount T since the last event from that pixel. T is a global event threshold. In addition, one bit of the address encodes the sign of the change (ON or OFF). This representation of "change in log intensity" generally encodes scene reflectance change. Because this computation is based on a compressive logarithmic transformation in each pixel, it also allows for wide dynamic range operation (120 dB, compared with e.g. 60 dB for a high quality traditional image sensor). This wide dynamic range means that the sensor can be used with uncontrolled natural lighting. The asynchronous response property also means that the events have the timing precision of the pixel response rather than being quantized to the traditional frame rate. Thus the "effective frame rate" is typically several kHz. The temporal redundancy reduction greatly reduces the output data rate for scenes in which most pixels are not changing. The unique design of the pixel also allows for unprecedented uniformity of response. The mismatch between pixel contrast thresholds is a modest 2.1% contrast. The event threshold can be set to 10% contrast, allowing the device to sense real-world contrast signals rather than only artificial high contrast stimuli. The vision sensor also has integrated digitally-controlled biases that greatly reduce chip-to-chip variation in parameters and temperature sensitivity. And finally, the system we built has a USB2.0 interface that delivers time-stamped address-events to a host PC. This combination of features has meant that we have had the possibility of developing algorithms for using the sensor output and testing them easily in a range of real-world scenarios.
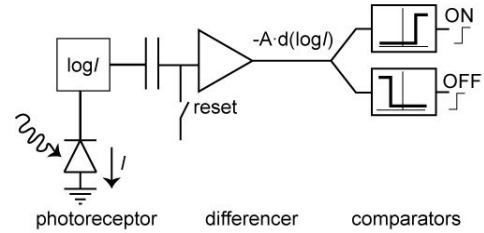
Table 1   Tmpdiff128 transient vision sensor specifications

|	
Functionality	Asynchronous temporal contrast
Pixel size um (lambda)	40x40 (200x200)
Fill factor (%)	9.4% (PD area 151µm²)
Fabrication process	4M 2P 0.35um
Pixel complexity	26 transistors (14 analog), 3 capacitors
Array size	128x128
Interface	15-bit word-parallel AER; USB2.0 time-stamped address-event interface
Power consumption	Chip : 24mW @ 3.3V. USB system : 80 mA.
Dynamic range	120dB ; 2 lux to > 100 klux scene illumination with f/1.2 lens
Response latency	15µs @ 700mW/m²
Max Events/sec	~1M events/sec
Event threshold matching	2.1% scene contrast
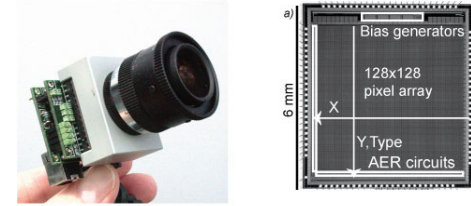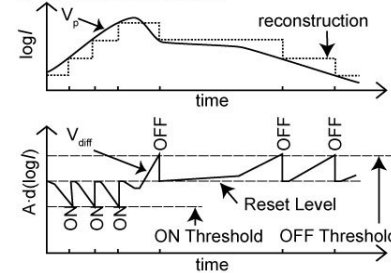
Fig. 2   Summarizes characteristics of Tmpdiff128 transient vision sensor. a) shows the vision sensor with its lens and USB2.0 interface; b) shows a die photograph labeled with the row and column from a pixel that generates an event with x,y,type output; c) shows an abstracted schematic of the pixel that responds with events to fixed-size changes of log intensity; d) illustrates how the ON and OFF events are internally represented and output in response to an input signal. Figure adapted from [1].

B. Tracking algorithm

The balls are tracked using the cluster tracker algorithm described in [1, 4] (Fig. 3). Each "cluster" models a moving object as a source of events. Visible clusters are indicated by the boxes in Fig. 3. Events that fall within the cluster move the cluster position, and a cluster is only considered supported ("visible") when it has received a threshold number of events. Clusters that lose support for a threshold period are pruned. Overlapping clusters are merged after each event packet. All cluster parameters (e.g. position, velocity) are updated by using a mixing factor that mixes the old value with the new measurement using fixed factors. Thus the time constant is not constant, but rather gets shorter as the event rate increases and there is more evidence; likewise lack of evidence (events) increases the time constant.

The key advantages of the cluster tracker are: 1) There is no frame correspondence problem. 2) Only pixels that generate

846

events need to be processed and the cost of this processing is dominated by the search for the nearest existing cluster, which is typically a cheap operation because there are few clusters. 3) Memory cost is low because there is no frame memory, only cluster memory, and each cluster requires only about 100 bytes memory.

In the application described here the clusters have a fixed size, although modifications that allow size modeling based on perspective or data are possible [1, 4].

More detailed steps for the cluster tracker are outlined as follows. For each packet of events:

1. For each event, find the first cluster than contains the event. (This will also be the oldest cluster if several contain the event because the cluster list is ordered by creation.)
   a) If the event is within the cluster, add the event to the cluster by pushing the cluster a bit towards the event and updating the last event time of the cluster. The new cluster location $\vec{x}_{n+1}$ is given by mixing the old location $\vec{x}_n$ with the event location $\vec{e}$ using an adjustable mixing factor $\alpha \approx 0.01$:
   $$\vec{x}_{n+1} = (1-\alpha)\vec{x}_n + \alpha\vec{e} \qquad (1)$$
   Other parameters like cluster velocity are also updated by mixing, but with different mixing factors.
   b) If the event is not in any cluster, seed a new cluster if there are spare unused clusters to allocate. (We typically use 10 potential clusters in this application.)
2. Iterate over all clusters, pruning out those clusters that have not received sufficient support. A cluster is pruned if it has not received an event for a "support" time, typically 5 ms in this application.
3. Iterate over all clusters to merge clusters that belong to the same object. This merging operation is necessary because new clusters can be formed when an object increases in size or changes aspect ratio. This iteration continues until there are no more clusters to merge and proceeds as follows: For each cluster that touches another cluster, merge the two clusters into a new cluster and discard the previous clusters. The new cluster takes on the history of the older cluster and the location of the new cluster is the weighted average of the locations of the source clusters, where the weights are given by the number of events in each source cluster. (This weighting greatly reduces the jitter in the cluster location caused by merging.) Continue this iteration over all clusters until there are no more clusters to merge.

A cluster is not marked as "visible" until it receives a certain number of events (typically 30 in this application).

The goalie robot uses the nearest cluster (lowest in the image) that is moving towards the goal as the ball object.

The ball cluster's location and velocity measurement are used to position the servo to intercept the ball. No account is presently taken of the goalie's arm dynamics and thus the controller is strictly proportional. A gain and offset parameter are used to adjust for optics and alignment. To reduce gear wear, the servo motor is only enabled when an approaching ball is detected. Thus the goalie "relaxes" between shots.
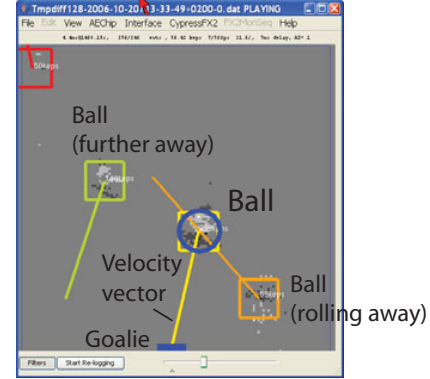

Fig. 3   Shows 4.4 ms of data (256 events) from the retina view showing four tracked balls. The closest ball rolling towards the goal (and being blocked) is marked with a circle; other balls are tracked but ignored. The velocity vectors of each ball are also shown. The goalie servo is moved to the bar shown. The balls generate average event rates of 3-30 keps. The mean event rate for this packet was 56 keps.

C.   USB interfaces and servo control

The system runs inside a large Java software infrastructure for AER (INI-AE-Biasgen) that has >200 classes and >30k non-comment lines of code (NCLOC). The cluster tracker and goalie, however, consist of only 1k NCLOC.

For both the retina and the servo controller we used the Java interface to the excellent Thesycon USB driver development kit for Windows (www.thesycon.de). The servo commands are sent to the microcontroller in a separate writer thread that takes commands placed in a queue by the retina event processing thread. This writer thread decouples the servo communication from the event processing so that the retina processing can proceed independently from the relatively slow writes to the full-speed (12 Mbps) USB1.0 servo interface. Likewise, the retina events are captured in 128-event transfers and processed in a high priority thread that runs independently from the GUI or rendering threads. The USB interface threads were set to high priority, with highest priority given to the servo writing thread. Java's maximum priority is equivalent to Windows TIME_CRITICAL priority [5].

We used a Futaba S9253 servo to move the goalie arm (http://www.futaba-rc.com). This fast hobby digital servo, accepts pulse-width modulation (PWM) input at several hundred Hz and is rated to rotate 60° with no load in 60 ms. It can move the arm across the goal in about 100 ms but is underdamped with the goalie arm as the load. We used a Silicon Labs C8051F320 USB1.0 microcontroller (www.silabs.com) to interface between the PC and the servo

847

motor. The microcontroller accepts commands over a USB bulk endpoint [6] that specify up to two servo motor position signals that it loads into internal counter-timer registers that program the PWM output width. The small servo motor is also powered directly by the 5V USB VBUS. The servo arm is constructed from a paint stirrer stick with a plastic "hand" stuck on its end. The goal and ramp are raised about 4 cm to allow the hand to protrude into the gap in order to better cover the corners of the goal (Fig. 1).

III. RESULTS

The robot achieves a success rate of 80-90% in blocking balls that are shot with >150 ms time to impact. Misses increase as the limits of servo acceleration are reached. The cluster tracker algorithm is effective for ignoring distracters (Fig. 3). Events are processed for ball tracking at a rate of 1.4 Meps (700 ns/event) on a 2.1 GHz Pentium M laptop running Windows XP, Java server JVM version 1.5.

When events are processed at the event packet level as delivered by the hardware, latency is reduced to the buffer acquisition time. A single ball that produces events at 30 keps causes a 128-event USB packet every 4 ms. Paradoxically, more retina activity actually reduces this latency, but this is only because the buffers are filled more rapidly.
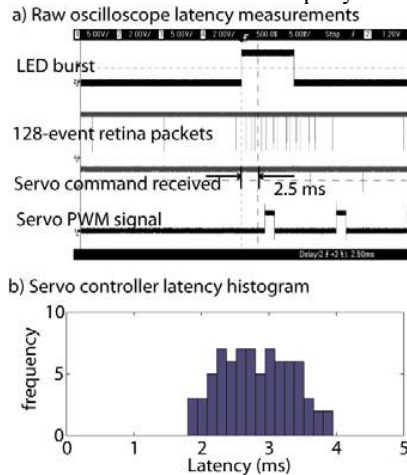

Fig. 4   Shows latency measurements. a) Oscilloscope traces showing response latency in LED burst experiment for a single trial. Top trace shows the LED burst. Second trace shows 128-event transfers of events from retina to PC. Third trace show when servo command is received by the main loop on the microcontroller. Last trace shows resulting PWM output to servo motor.
b) shows histogram of latencies; mean 2.8±0.5 ms, median 2.8 ms.

To study latency, we set up an artificial stimulus consisting of a flashing LED that could be activated in bursts, thus mimicking an instantaneously-appearing ball. We then programmed the servo motor microcontroller to turn on an output pin when it received a servo motor command. The time delay between the beginning of the LED flashing and the microcontroller output is a measure of response latency that leaves out the latency of the random PWM phase and the servo motor (Fig. 4). The latency is 2.8±0.5 ms. This latency was achieved by setting the device interrupt polling interval to 1 ms in the device's USB descriptor [6]; using the default polling interval of 10 ms resulted in substantially higher median latency of 5.5 ms that varied approximately bi-modally between 3-10 ms.

IV. CONCLUSION

The main achievement of this work is the concrete demonstration of a spike-event driven hybrid neuromorphic-procedural system for low latency object tracking and sensory motor processing. Secondary achievements are refinements of existing object tracking algorithms and the development of a reusable, convenient USB servo-motor interface. The goalie robot can successfully block balls even when these are low contrast white-on-gray objects and there are many background distracters. Progress in neuromorphic engineering will result from combining powerful neuromorphic sensors with the flexibility, familiarity, wide availability and low cost of procedural computation. End users of these sensors will drive sensor improvement and the development of more complex neuromorphic processing for parts of the problems that stress available processor capability.

REFERENCES

[1]   P. Lichtsteiner, C. Posch, and T. Delbruck, "*A 128×128 120dB 30mW Asynchronous Vision Sensor that Responds to Relative Intensity Change*," in ISSCC Dig. of Tech. Papers, Visuals Supplement, San Francisco, 2006, vol., pp. 508-509 (27.9).

[2]   R. L. Andersson, *A robot ping-pong player: experiment in real-time intelligent control*: MIT Press, 1988.

[3]   M. Mahowald, *An Analog VLSI System for Stereoscopic Vision*. Boston: Kluwer, 1994.

[4]   M. Litzenberger, C. Posch, D. Bauer, P. Schön, B. Kohn, H. Garn, and A. Belbachir, "*Embedded Vision System for Real-Time Object Tracking using an Asynchronous Transient Vision Sensor*," in IEEE Digital Signal Processing Workshop 2006, Grand Teton, Wyoming, 2006, vol.

[5]   S. Oaks and H. Wong, *Java Threads*: O'Reilly, 2004.

[6]   J. Axelson, *USB Complete*: Lakeview Research, 2001.

848