CIRCUIT CELLAR®
THE MAGAZINE FOR COMPUTER APPLICATIONS

FEATURE ARTICLE

by Hanno Sander

Vision-Guided Robotics

A Next-Generation Balancing Robot

Are you interested in building a sophisticated, vision-guided, balancing r obot that can interact with its environment? Hanno shows you how to tackle this project with a Parallax Propeller, a handy design kit, and an inexpensive camera.

It's time to build the next generation of robots. W ith today's technology, our robots should be tall enough to look us in the eye and interact with us through sight. Two years ago, I star ted experimenting with the latest microprocessor from Parallax, a camera, and a vision. In this article, I'll explain how to integrate vision technology in a project. As I describe the balancing robot design you see in Photo 1, I'll cover parallel processing, visual debugging, vision algorithms, and robot control with computer vision.

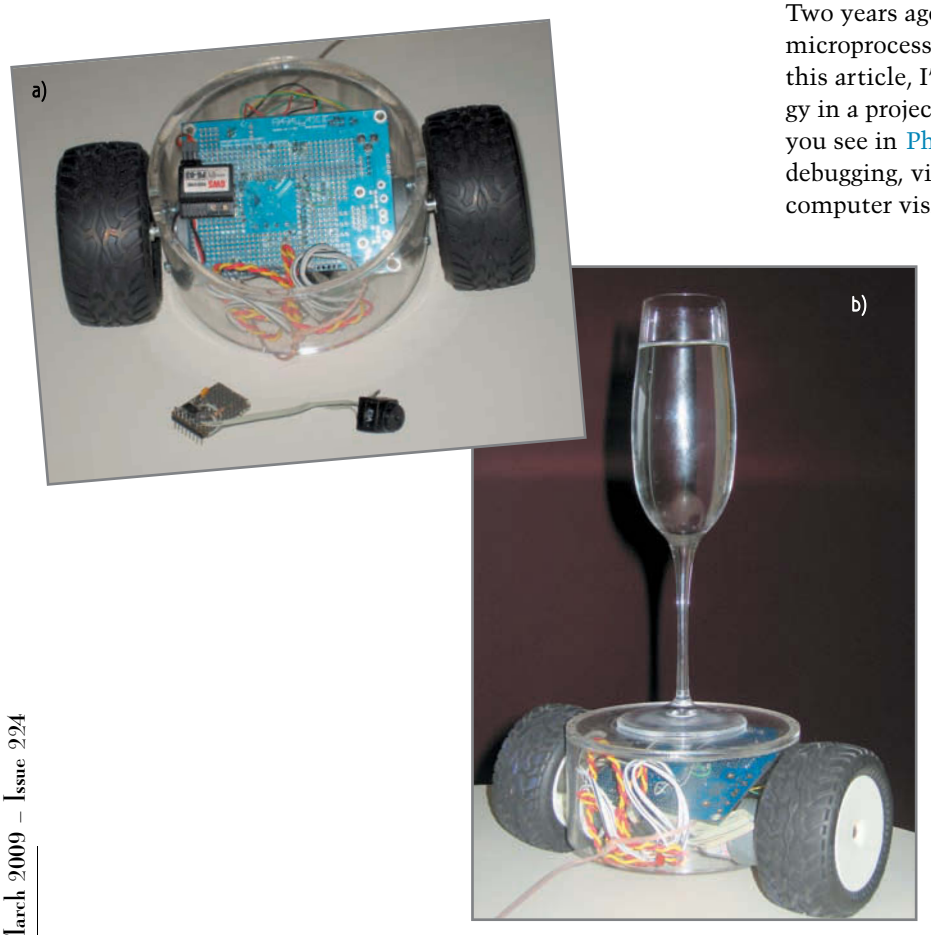

Photo 1a—This is the base of the DanceBot with wheels mounted on motors with quadrature encoders, the logic board powered by the Parallax Propeller, and the miniature camera with an ADC module. b—This autonomous balancing robot uses vision to interact with users. Here it's getting ready to balance a champagne flute for a month.

PARALLAX PROPELLER

I started my project after my dad gave me a Parallax Propeller for Christmas. Parallax is best known for its Boe-Bot robot and BASIC Stamps, but its Propeller chip is quickly becoming popular with engineers and hobbyists because of its power and simplicity. Its eight identical processors (cogs) share common resources, such as global memory and an I/O por t, but each can run its own program. It was perfect for my application because I needed to sample many different sensors at different rates, perform intensive filter calculations,

Sensors	Preprocess	Logic	Actuators
Camera	Location of person	Find person and set target position	
Quadrature encoder	Position, velocity	Fuzzy PID loop to balance and achieve target position	Drive two motors
Gyroscope Accelerometer	Tilt, rate of turn		

Figure 1—The DanceBot gets information from its environment through its sensors: a camera, a quadrature encoder, a gyroscope, and an accelerometer. It processes this data to find its dance partner, current position, and tilt. Fuzzy logic is used to balance and to maintain a set distance from its partner by driving the wheel motors.

process captured video, and control motors. I focused on parts of the problem and later integrated everything by assigning different algorithms to their own cogs.

The Propeller has eight 32-bit processors running at up to 80 MHz. It has shared global resources, including 32 KB of RAM, 32 KB of ROM, and 32 I/O pins. It has dedicated resources per processor, including 2 KB of RAM and two general counters, and video output. The Propeller operates at 3.3 VDC (each pin can sink up to 40 mA). It is available on Parallax's web site for $12.99 per chip and $29.99 per ProtoBoard.

ViewPort

Parallax offers a free tool to load programs written in assembly or a high-level language called Spin to the Propeller. This is fine for getting started writing simple programs, but I quickly determined that I needed a more powerful debugging tool to develop and configure my vision-powered balancing robot—an application I now call ViewPort.

I started by dedicating one of the eight cogs to continuously share data stored in the Propeller's memory with a PC application. This enabled me to monitor and change variables while the other seven cogs ran at full speed. When I added a module that sampled all 32 I/O pins at 80 MHz, other designers became interested in using my application to debug their integration code—and ViewPort was born. Since then, I've added other capabilities to the ViewPort application to turn it into a complete debugging package. You can download a free 30-day trial of ViewPort at http://mydancebot.com.

THE DanceBot

After watching friends demonstrate their iRobot Roomba robotic vacuum cleaners at a party, I wondered if I could build a balancing robot that could dance—not just with me, but with anyone in any environment. I wanted to build a robot that could dance with people and seem almost human.

Balancing robots make a great platform for mobile robots. They are highly maneuverable, have great traction, and move more smoothly and naturally than other designs. They can turn on a dime, navigate precisely, and are a pleasure to watch while they keep their balance. Unfortunately, building a robot that balances and maintains position robustly in any environment is not easy.

The first lesson I learned was that unlike the inverted pendulum problem, a true balancing robot requires two control loops: one control loop to keep the robot from falling and another to keep the robot from losing its position. This combination also lets you move it programmatically. A significant milestone for building a balancing robot involves taking a simple step, accelerating to a set speed, travelling, and then decelerating to a stop. The DanceBot uses what's known as a "hybrid fuzzy logic cascading PID controller" to precisely carry out this and other advanced maneuvers. In the algorithm, the inputs to the PID controllers are first processed by a fuzzy logic engine to make the control algorithm more robust and easier to tune. The PID controllers, which correct the error between a measured variable and its setpoint by calculating a corrective action, are arranged in a cascade with the output of one used as the setpoint in the second.

Second, while it's possible to determine tilt by optically measuring the distance to the floor, this technique isn't robust. The DanceBot measures rate of turn using a ceramic gyroscope and integrates this signal to calculate tilt. Fusing the calculated tilt value with measurements from an accelerometer with a Kalman filter yields an accurate tilt reading with no drift. This combination lets the DanceBot stay balanced in any environment.

The DanceBot is controlled like a car: it requires two channels of information (see Figure 1). Channel 1, speed, controls how fast the robot should travel. Channel 2, turn rate, controls how quickly the robot should turn about its own axis. The DanceBot manages the speeds of its two motors to stay balanced and to achieve the position orientation and velocity goals given by its higher level planner. Unlike a car, the robot is capable of turning in place. At first, I controlled my robot with a remote control, but I quickly realized that it would be much more fun if it could interact with others as well—just by watching what they were doing. The first step to guide the robot with vision was to build a frame grabber.

FRAME GRABBER

The DanceBot's vision is controlled by a small grayscale Electronics123.com C-Cam-2A miniature video camera. It is just 16 × 16 × 16 mm, uses less than 100 mW, and costs less

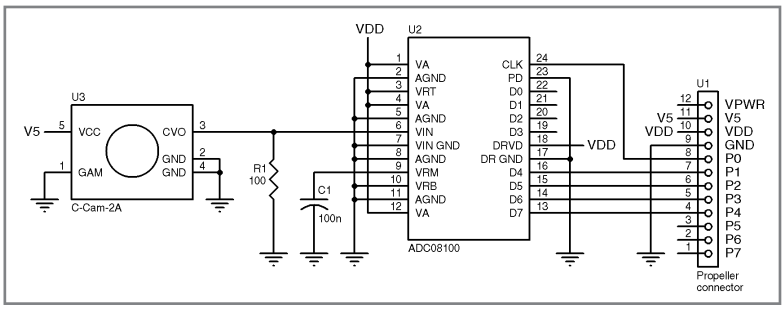

Figure 2—This is the frame grabber hardware. The C-Cam-2A outputs an NTSC composite signal in pin 3. This is digitized by the ADC08100 whose output D4..D7 is fed to the Parallax Propeller.

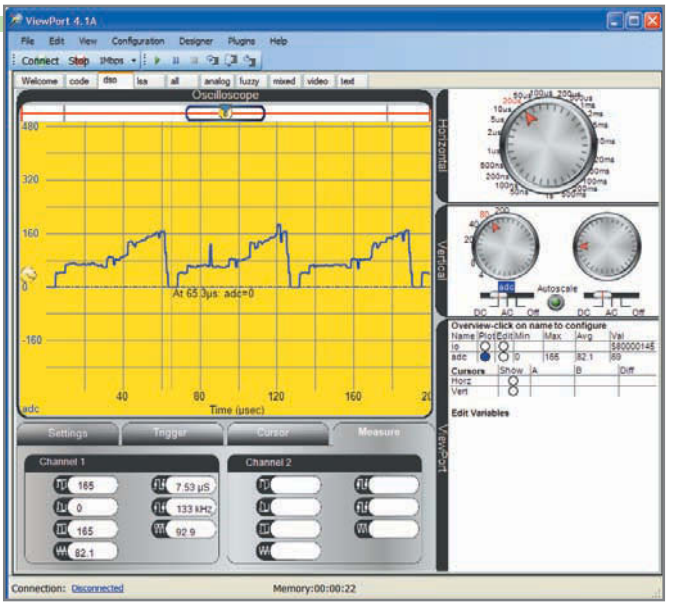

Photo 2—ViewPort shows raw NTSC signal from the camera, as digitized by the ADC.

than $20. It has five pins, three of which provide ground and 5-V power, a gamma mode, and an output. The output signal consists of a 1-V_{PP} composite video signal when terminated with a 75-Ω resistor to ground. To watch the camera's output, you can simply plug it into the composite input of your TV. It's that simple! Understanding what the camera sees is a bit harder, so I'll take it one step at a time.

First, you have to digitize the analog signal. To sample slower waveforms with the Propeller, you would typically use delta-sigma modulation with a capacitor and a resistor. But because you need to resolve the individual pixels in a frame, you need a faster solution.

The ADC08100 is a 20- to 100-Msps, 8-bit ADC. With a clock signal, it will output the digital equivalent of its input voltage on its eight digital outputs. We'll use one of the Propeller's 16 hardware counters to clock the ADC at 10 MHz and read the result from the Propeller's I/O port (see Figure 2).

At this point, your robot is ready to take its first peak at the world—one scan line at a time (see Photo 2).

Listing 1 is a short program that uses ViewPort to trigger and display the NTSC waveform generated by the camera.

The program starts by configuring the Propeller's clock to run at 80 MHz and including the three objects you need. The `vp` commands register is a component that will quickly sample the state of the I/O port and configure the ViewPort interface. Finally, the `Freq.Synth` call generates a 10-MHz clock to drive the ADC. Photo 2 shows the oscilloscope with a timescale of 10 µs/division. The waveform represents a horizontal trigger followed by a color burst and 50 µs of data. A pixel's brightness is proportional to the signal's value.

To complete your frame grabber object, your algorithm must detect the horizontal and vertical sync marks and then compress the pixel data into memory. Vertical and horizontal sync marks differ in the amount of time the signal stays at the lowest level. After detecting a vertical sync

Listing 1—This Spin program sets the Propeller's clock to run at 80 MHz and includes several objects: "Conduit" to graph the ADC's waveform via ViewPort on the PC, "QuickSample" to sample the Propeller's I/O pins, and "Synth" to generate the 10-MHz clock signal for the ADC. Running this on the Propeller enables you to display the NTSC waveform, as sampled by your frame grabber on the PC.

```
CON
      _clkmode          = xtal1 + pll16x
      _xinfreq          = 5_000_000
OBJ
      vp    :         "Conduit" 'transfers data to/from PC
      qs    :         "QuickSample"      'samples INA up to 80Mhz
      Freq  :         "Synth"
pub demoADC|a,frame[1600+6] 'frame stores 1600 samples+configuration
      vp.register(qs.sampleINA(@frame,1 ))
      vp.config(string("var:io,adc(decode=io[0..7])"))
      vp.config(string("dso:view=adc,trigger=adc<15,timescale=50us"))
      vp.share(0,0)
      Freq.Synth("A",8, 10_000_000)
      repeat
```

mark, the code initializes a new frame and processes one video line at a time. For each line, it detects the horizontal sync, skips past the color burst, and then samples the ADC's value every five instructions—for a line length of 240 pixels. To fit a complete video frame into the Propeller's global memory, I store 4 bits of brightness information for each pixel. This data is accessible by all eight cogs on the Propeller. In the DanceBot, one cog is dedicated to run this program continuously to sample video from the camera at 30 fps with a resolution of 240 pixels × 200 lines × 4 bits/pixel.

Listing 2 is an example program that uses the VideoCapture object. This program configures the clock and imports some objects and then starts the video cog to capture frames. Then, it configures ViewPort to display the streamed video. The Spin code draws a thick black line in the middle of the frame by setting parts of the array to 0. Photo 3 shows the project's first view of the world.

REAL-TIME TRACKING

You know how to create the infrastructure to digitize video from a camera into the Propeller's memory using an ADC and one of the Propeller's eight cogs. You have 24 KB of visual data updated 30 times per second. Now you need a filter that can analyze the video and give you just two variables to control the robot.

Start by implementing a filter that identifies the location of the brightest spot in each frame. It's easy to search for the maximum value in your array of pixel brightness values. Just remember that you're working on 4-bit pixel values compressed into 32-bit longs (A 32-bit long on the Propeller is the basic unit of memory space) (see Listing 3).

This filter processes one pixel every five instructions. Because the filter processes only data, not sync marks or color bursts, you can process video at 40 fps. So the filter can easily keep up with the frame grabber object and update the position of the brightest spot in real time.

To integrate this code with the rest of your DanceBot, simply keep this filter code running in its own cog. The cog will continually filter the data provided by the frame grabber and write the x,y location of the brightest spot into the main memory.

You now have two channels of information updated at 30 times/second with which you can drive the two control channels of your robot—speed and direction. Use the x position of the spot to control the turning rate of the robot. If the spot is in the middle, you don't need to do anything. However, if it's on the left side of the image, the algorithm tells the robot to turn left, until the spot is in the center and the robot is facing the source of the spot. A similar technique controls the robot's speed using the vertical position of the spot. The algorithm's goal is to keep the spot's position centered in the image. So, when the spot is too low, the robot is instructed to move forward, which brings the robot closer to the spot's source. Because your camera is looking up at the spot, it will raise the spot in the image. Conversely, if the spot is too high, your robot is too

close, so it's commanded to drive backwards. Translating this algorithm into code is simple, just scale and offset the x,y location of the spot to control the robot. The complete control program for the robot is posted on the *Circuit Cellar* FTP site.

To illustrate the tracking ability of this filter, I can use ViewPort to display the streamed video with a superimposed trail showing the position returned from the filter over the last minute. Photo 4 shows the grayscale image, as seen by the camera, with a yellow trail showing the path the bright source took.

Listing 2—This Spin program sets the Propeller's clock and includes two objects to find a blob: "Conduit" to stream the video signal to the PC and "VideoCapture" to grab the frame. By continually setting videoFrame[3010] to 0 with the ~ operator, we will have an eight-pixel horizontal black line in the center of the image.

```
CON
      _clkmode        = xtal1 + pll16x
      _xinfreq        = 5_000_000
OBJ
      vp    : "Conduit"           'transfers data to/from PC
      video: "VideoCapture"       'capture video signal pub
findblob|videoFrame[6000],a,blob
      vp.register(video.start(@videoFrame,video#HIVIDEO))
      vp.config(string("start:video"))
      vp.share(@blob,@blob)
      repeat
       repeat
         videoFrame[3010]~
```

LINE FOLLOWING WITH A CAMERA

You can control the behavior of the robot by shining a bright light at the camera. This works in some environments where you can control the lighting and ensure that no other objects reflect or create light to the camera that's brighter than your flashlight. It's also an active method, where you have to power the flashlight. I'll now describe a filter that is less restrictive and uses a passive method to steer the robot.

Most line-following robots use two phototransistors to stay on a line. They're programmed to ensure that one detector is on the dark line while the other is on the lighter background. More sophisticated robots use additional detectors to detect the robot's exact position on the line to look ahead or even to recognize junctions. In this section, you'll build a filter that uses your existing frame grabber to perform line following with a camera.

Again, your frame grabber gives you too much information, so you need to design a filter that will steer a robot in the middle of a line. Tilt the robot's camera so its field of view is from below the horizon to just in front of the robot. Now, you can stream the video to ViewPort and analyze what the video of a properly programmed robot would do. It will become apparent that a good algorithm involves averaging the location of the darkest pixel in each line. This is quite robust, easy to program, and gives a good control signal to the robot. Again, integrating this filter with the rest of the DanceBot is straightforward. Just use the average position of the line to control the direction of the robot while it's moving along at a set speed (see Listing 4).

TRACK A PATTERN

You've gone from tracking an active, bright spot to following a passive line on an artificial background. Now it's time to track a passive pattern in the real world. The goal for this section is to develop an algorithm and pattern that will steer the robot in any environment. As an experiment, take a look around and imagine what type of pattern would stand out in our typical cluttered world. For most people, a bar code-like pattern of repeated black and white lines should do relatively well. Of course, this pattern won't suit everyone (e.g., Zebra lovers may need to find another pattern). But this pattern doesn't occur often, and it can be identified reasonably easily with a chain of simple vision filters.

Before you analyze the individual filters, analyze how ViewPort manages a chain of vision filters. Because the Propeller's memory is a limited resource, you can afford only to keep one image in memory at a time. To visualize the effects of different filters, segment the image array into four vision buffers: top left, top right, bottom left, and bottom right. Both the frame grabber and vision objects support both the full-size and the segmented modes. Filters operate on buffers in that they read data from one place and write their result to another. The DanceBot uses one cog to continuously process image data with a number of filters—one at a time. Configuring the filter's order, parameters, and values is high-level Spin language. Streaming all four vision buffers to ViewPort enables you to watch how each filter manipulates the video in real time.

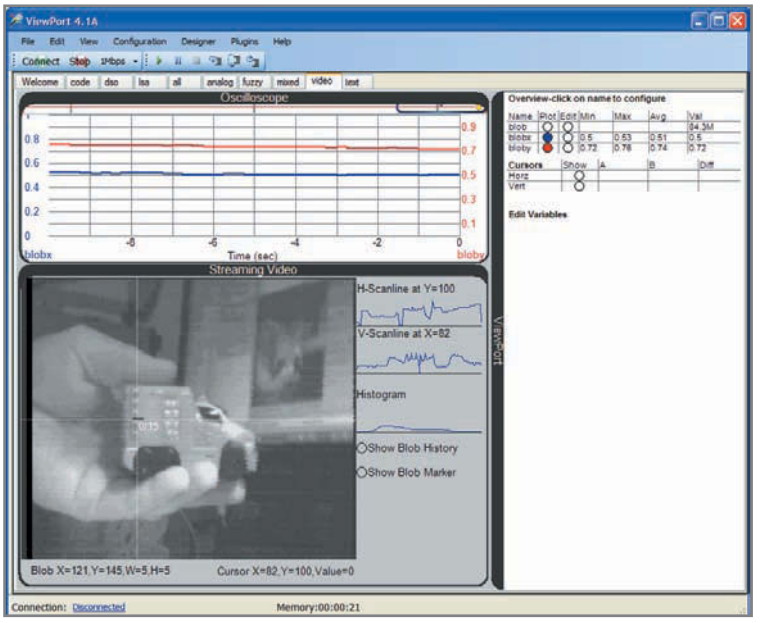

Photo 3—This is the system's first picture of a fire truck. Notice the black line at the cross hairs.

Now that you

Listing 3—This Propeller assembly function finds the brightest pixel in the video frame. Thirty-two bits of data are read using the `rdlong` command. This represents eight pixels, which are inspected one by one by rotating the bits with the `ror` instruction. The location of the brightest pixel is written to the address pointed to by `cmdPtr`.

```
doMax
  '2 ptrs :src, dnp
  '2 value:old, dn
  'setup ptrs to positions
                rdlong        val,cmdPtr
                add           cmdPtr,#4
                mov           dnp,src
                add           dnp,bytesNline
                sub           n,#15
                mov           dest,#0 'seexy
                mov           sum,#0 'max value

:loop           rdlong        old,src
                add           src,#4
                rdlong        dn,dnp
                add           dnp,#4

                mov           m,# 8
                mov           new,#0
                'input: old,dn have pixel in 0..3, new has data
                'output: old,dn rol by 4
:dodiffb        mov           tmp,old
                and           tmp,#15 'tmp=pixelvalue in mid
                mov           t1,dn
                and           t1,#15 't1=pixel down
                add           tmp,t1 'tmp=two rows
                cmp           tmp,sum wc 'c if tmp<sum
if_c            jmp           #:notMax
'                 mov           dest,m
'                 shl           dest,#8
                mov           dest,sum
                shl           dest,#16
                add           dest,n
                mov           sum,tmp 'reset max
:notMax         ror           dn,#4
                ror           old,#4
                djnz          m,#:dodiffb
                djnz          n,#:loop
                wrlong        dest,val
                jmp           #cmdLoop
```

understand how ViewPort vision filters work, build a filter to look for transition edges in your video. Looking at relative changes in value improves the robustness of your algorithm, especially when lighting conditions change. The horizontal Soebel filter is quick to implement and does a good job of detecting vertical edges. To compute it, replace each pixel with the absolute value of the difference of its horizontal neighbors.

Next, you need to design a filter that will identify regions with multiple strong transitions. The algorithm I chose keeps a running total of the last eight edges by adding the strength of the next edge and subtracting the last. This running total is saved to the appropriate location in the buffer.

Your last filter finds the location of the maximum in the buffer you just calculated. You can use the same algorithm that you developed to track a bright spot. By chaining these three filters together, you find the location of the maximum running total of vertical transitions. In other words, you can

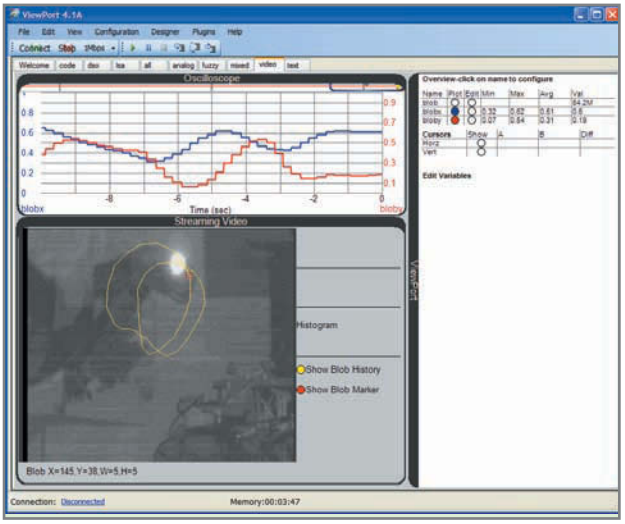

Photo 4—The DanceBot is tracking the path of a flashlight in real time.

find a black/white striped pattern.

To make your pattern visible at different distances, repeat the pattern at various scales. When you place this pattern on your belt, you can start dancing with your robot. Stepping closer to the robot makes the image of your pattern move up in the robot's field of vision. This causes the robot to move backwards and maintain a set distance from you. Stepping to one side of the robot causes the pattern to move horizontally, which commands the robot to turn and face you. While dancing with my robot, I discovered some interesting behavior that I hadn't planned on. When I jumped up or crouched down, the robot would change its set distance to me. When I turned around, thereby covering the pattern, the robot also turned around. It no longer detected the pattern and went into search mode where it turned on its axis.

FIND A BEER BOTTLE

I believe all robot vision articles should include the beer-finding problem. Finding a specially colored beer bottle with a color camera is quite doable; however, you're limited to a grayscale camera. You're also not guaranteed that the beer bottle will be the brightest object in the room. There's no line to follow. The beer bottle doesn't have a distinctive pattern. The only available trait is the actual shape of the beer bottle.

Use the correlation algorithm to find the shape of the beer bottle in a typical cluttered environment, u. This algorithm uses brute force to match a desired template to all possible locations in the image. The location of the best match is the location of the beer bottle. The degree of match at any given point is the sum of absolute differences between the pixels of the template and the corresponding pixels of the area to match. To improve the robustness of the algorithm against changes in brightness and contrast, I preprocess the template and every possible match with an auto-level algorithm.

This algorithm does a good job of finding beer bottles. Photo 5 shows the target identified in a complex environment. However, the image of the bottle must be a close match to the template (i.e., its scale and orientation must be

Listing 4—This Propeller assembly function sums the horizontal position of the darkest pixel to steer the robot along a black line. For each horizontal line, it uses t3 to track the location of the minimum brightness and adds this to t2 at the end of the line.

```
doLowest
              rdlong    t1,cmdPtr 't1 is the address of the result variable.
                                  We'll write the x position of the line here.
              add       cmdPtr,#4
              mov       t2,#0                       't2 is sum of pos


              mov       n,linesNpanel               'loop over panel
:loopLines     mov       x,longsNline               'loop over line
              sub       x,#2
              mov       val,#15                     'reset val
:loop        rdlong      old,src                    'loop over longpixels
              add       src,#4
              mov       m,#8
              mov       new,#0

:limit         mov       tmp,old
              and       tmp,#15
               cmp       tmp,val wc                 'c set if v1<v2
'     if_c     add       new,#15
'     if_c    mov       val,tmp
if_c          add        new,#15
if_c          sub        val,#1
if_c          mov        t3,x          't3 is pos of lowest item ror new,#4
               ror       old,#4
              djnz      m,#:limit

'              wrlong     new,dest
              add        dest,#4
              djnz      x,#:loop                    'loop over longpixels
              add       t2,t3
              add       src,#8
              add       dest,#8
              djnz      n,#:loopLines               'loop over lines
              wrlong     t2,t1
              jmp        #cmdLoop
```

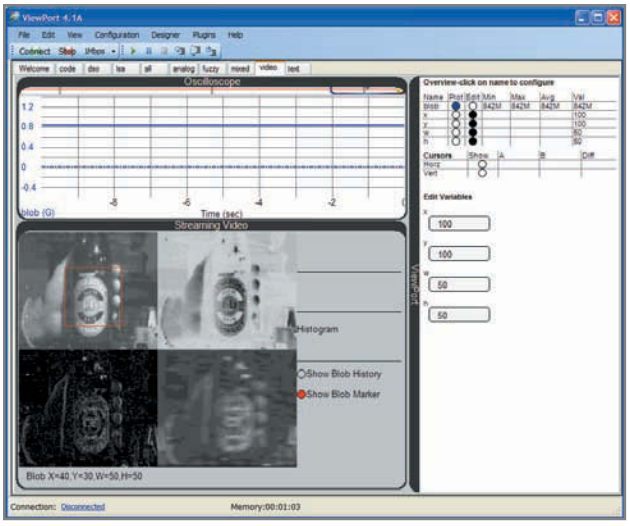

Photo 5—The DanceBot is finding the location of a beer bottle.

identical). With additional processing power, multiple templates could be searched in succession or parallel to more robustly identify the bottle. Once identified, a faster, less processor-taxing algorithm could be used to steer the robot to the beer bottle in real time.

WRAP UP

I've had a lot of fun building the vision-guided DanceBot with the Parallax Propeller and ViewPort. The Propeller's unique architecture of eight identical cogs made it easy to split my goal of guiding a balancing robot with vision into manageable pieces (see Photo 1b).

Depending on the performance required, I could write the code in the high-level, object-oriented Spin language or dive down to assembly to write completely deterministic code. The logic to configure and control the robot ended up being programmed in Spin, while the frame grabber and vision filters are programmed in assembly. Even though resources are limited, it's possible to carry out advanced vision processing with it. In today's age of multilevel architectures relying on outside libraries, drivers, and operating systems, it was a breath of fresh air to program the entire chain, from decoding the NTSC waveform to controlling the robot. Visually debugging the DanceBot with ViewPort greatly simplified its development by showing me exactly what was going on with my robot. It acted like a black box when the robot fell down, and showed me what the camera and filters were processing when I was teaching it to dance. It should be straightforward to adapt the code and filters presented in this article to other robots. Good luck!

Hanno Sander (hanno@mydancebot.com) has been working with computers since he programmed a lunar lander game for the z80 when he was six. Since then, he graduated from Stanford University with a degree in computer science and then started his corporate career as an Internet entrepreneur. Hanno moved to New Zealand in 2005 to spend time with his growing family and develop sophisticated, yet affordable, robots—starting with the DanceBot. His technical interests include computer vision, embedded systems, industrial control, control theory, parallel computing, and fuzzy logic.

PROJECT FILES

To download code, go to ftp://ftp.circuitcellar.com/pub/Circuit_Cellar/2009/224.

SOURCES

C-Cam-2A Miniature video camera
Electronics123.com, Inc. | www.electronics123.com

ADC08100 ADC
National Semiconductor Corp. | www.national.com

Propeller
Parallax, Inc. | www.parallax.com