12

Machine Vision

The broad subject of machine vision has many levels of complexity. The simplest is the use of a single photosensitive detector to locate the boundary of a brightness change, so that, for example, a factory vehicle carrying parts can follow the edge of a line painted on the floor using "if it's bright, steer left; if it's dark, steer right."

At the other end of the scale is a high-resolution color vision system in which the computer must recognize some object by its shape or texture, even though it might be partially obscured.

Some of the associated mathematical and computational techniques are concerned with improving the "quality" of the appearance an image, while others relate to extraction of data from the image such as the finding of edges and other features.

12.1 VISION SENSORS

In Chapter 2, we met a hierarchy of optical sensors that can be ranked in order of increasing complexity as follows.

12.1.1 Single-Point, Binary

This is "pair" consisting of a single LED and a single phototransistor:

Essentials of Mechatronics, by John Billingsley

- A *reflective opto switch* to detect a dark mark on a light background or vice versa.
- A *slotted opto switch*, where the sensors are mounted to face each other and indicate when there is an obstruction in the slot.

12.1.2 Single-Point, Analog

A single photocell measuring brightness is a popular sensor for a "Micromouse," a robot finding its way through a maze, where brightness can be used as a crude measure of distance from a wall.

A single sensor can be given the attributes of a linescan device by scanning it, such as with the use of a spinning mirror.

An optically based sensor that has had wide adoption as a "quick fix" aid to navigation is the Sick sensor. This uses just such a spinning mirror to scan with a laser beam. The additional factor is that the beam is pulsed. High-frequency circuitry measures the time of flight of the return journey to and from the point of contact. In this way, a map is obtained of the range from the sensor as measured in the scanning plane.

There is much to criticize with this sensor, mainly because of its serial output format. It was originally designed simply as a safety device to ensure that nobody entered the proximity of a dangerous object such as an industrial robot, so the output of image data was intended as a diagnostic tool. The basic scanning rate is 40 scans per second, with maximum resolution representing samples at quarter-degree intervals. Even at 500 kHz baud rate, however, the serial output cannot keep up with the highest scan speed at the highest resolution.

12.1.3 Linescan Devices

These are a linear array of sensors, giving data for one line of an image. As in the fax machine, a two-dimensional image is built up by the object moving past the array.

12.1.4 Framescan Devices

A two-dimensional image is captured in one hit. There may be a single frame of data, relating to a rectangular array of pixels, or a stream of frames constituting a "movie."

12.2 ACQUIRING AN IMAGE

For the single-pixel or the linescan sensor, simple bit-level input will be similar whether the system is built around a single-chip microcontroller or a

PC. It is when we wish to acquire a full two-dimensional image that we are faced with a confusion of choices.

12.2.1 DirectX and VFW

For minimum effort, it is easy to purchase a low-cost Webcam and plug it into a USB port on a PC. The driver software that comes with it will enable you to see moving images on the screen, and freeware packages will let you communicate face-to-face with your friends.

There are many cards on the market that can tune a television signal or receive a "composite video" signal (the yellow socket on the VCR). They "stream" the data onto the computer screen, but again we must break into the entertainment-directed technology if we are to make serious use of the signal.

We must answer the problem of putting image data where you can attack it with analysis software.

Close to the hardware level, the "driver" inputs bytes of data and packs them into an array. It then signals software at the next level to indicate that a frame of data is ready, while data bytes continue to be packed into a second frame. An early standard for using such data is called Video for Windows (VFW). An OCX control for Visual Basic can be written to capture data at this level. Details of such an OCX, including the source code, can be found on the Web at www.essmech.com/12/2/1.htm.

As soon as you place this control in your VB form, you can access its properties and methods. One of these is SnaptoArray (I admit that it has more than two syllables!), which will copy the next frame of image data that arrives into an array that you name. What you do with the image is then up to you.

Vision and other media processes are supported in later versions of Windows by a software suite called DirectX. The software developer's kit can be downloaded free from the Microsoft site—although it is several hundred megabytes in size. It includes DirectShow, which deals specifically with video streams.

The package is designed around "drag and drop" concepts, in which "filters" are linked in a "graph." A handy tool for building such graphs comes with the package. It is called Graphedt.exe.

The filters are unlike any of the filters we have met in the control sections. One example of a filter is a videocamera! The filter appears as a rectangle on the chart. In general, it has input and output "pins" that are notional, not physical.

A "video capture filter" such as a Webcam might have two output pins, CAPTURE and PREVIEW. A right click on one of the pins can show its "properties," the format of the data that can be taken from it. A typical value for a Webcam is "Major type: video—subtype RGB24."

A right click inside the rectangle itself will present the choice of FILTER PROPERTIES. In this case, the choice will open a window in which video source and video format can be set or changed.

The other option when the output pin is right-clicked is RENDER PIN. A second box will appear, with label VIDEO RENDERER, with its input pin connected to the output pin of the Webcam rectangle.

In a control bar above are the green triangle and red square for media RUN and STOP. A click on the RUN icon causes a window to appear with the moving Webcam image in it.

Of course, this is just the tip of the iceberg. There are filters for compressing video, for rendering audio, for interleaving video and audio streams in an "AVI Mux," and a file writer to record your video to disk. These are just a few of the hundred or more filters that are likely to lurk on your machine.

So, how is video captured for analysis? The analysis can be performed without capturing it at all. Instead, the analysis software is written as yet another filter that can accept the incoming video stream in real time and pass on the desired conclusions.

The few simple lines of code that are required to process an image, say, to reduce objects to their edges, have to be "topped and tailed" with a mass of "include" references and other housekeeping. However, a colleague, Mark Dunn, has contributed a template and a "wizard" that have been placed on the Website. These will enable you to construct your own image processing filters. You will also find examples that you can modify for your own purposes.

Provided you do not mind depending on one specific commercial operating system, you will find this a satisfying and rapid way to arrive at machine vision solutions. You may instead prefer to take a "bottom-up" approach.

12.2.2 Video Chips

As USB Webcams have tumbled in price, their cousins have invaded mobile telephones. There is a growing market for video subassembly modules for embedding in consumer products, for both low-resolution "fun" applications and high-resolution cameras.

The computers destined to handle these signals are far removed from the PC. They are single-chip microcontrollers, such as the *reduced instruction set computer* (RISC) ARM series. Nevertheless, once the image has been captured into an array, the analysis procedures they apply are almost identical.

12.3 ANALYZING AN IMAGE

Image data bytes flow at an immense rate, even from a low-resolution camera. In RGB24 format, one byte is used for each of the red, green, and blue components of each pixel. An image of 640×480 pixels will require $640 * 480 * 3$ bytes per frame. There will be 30 frames per second (25 in many countries outside the United States), so the data rate is 27,648,000 bytes per second. Even at a resolution of 320×240, the flow is nearly 7 megabytes per second.

It is clear that an essential feature of analysis must be data reduction.

12.3.1 Data Reduction

For many purposes, each pixel can be reduced to a binary decision, light or dark. A vision guidance project studied small green seedlings on an earthy background, and a decision "soil" or "plant" gave all the image data needed. Immediately the data size is reduced by a factor of 24.

Perhaps the largest reduction can be made by looking at just a subset of the image bits. One project concerned the visual counting of macadamia nuts. They were picked up between the blue-colored bristles of a plastic brush roller. The routine needs only to look at every fifth pixel or so to avoid missing a nut. When a nonblue pixel is found, a more intensive search can be made to locate the outline of the nut with some accuracy. Thus the initial scan only looks at 1 pixel in 25 of the image.

The ultimate data reduction in such projects is to the "answer," maybe statements such as "steer left a little" or "there were 2435 nuts."

It is important to discriminate between processing methods that extract "facts" from the image and processing that will simply change the appearance—or processing that will change the appearance as little as possible, for that matter. Image compression such as is used for digital television is a subject in itself.

A black-and-white image is likely to contain lumps of black pixels and lumps of white ones, rather than a random scattering. An early method of data compression was *run-length* encoding, where each scanline is coded in a form that might represent "23 black, 15 white, 75 black . . . ," and so on.

But the clumping will take place in two dimensions, not just along scanlines, so methods such as LZW allow the data to be reduced in size with no loss of actual information.

The compression of color images presents a different problem. This time, image data must inevitably be lost, since only in cartoons will many adjacent pixels be identical in color, but the aim is to keep the "essence" of the appearance of the image.

One compression method is to use a "palette" of 256 colors and approximate each pixel to one of these. The approximation can be brought a little closer by the use of "dither," the alternation of two colors to get that appears to the eye as something in between.

More effective for photographs is the JPEG technique. The picture data defines the parameters of two-dimensional functions, bounded by coarse rectangular tiles of the image. These fit together to give a smooth high-resolution picture, but detail can be lost and flat areas such as sky can carry "tide marks" of color quantization. The degrees of compression and smoothing can be set as a parameter when compressing the image.

Sequences of movie images offer even further possibilities. Many "codecs" (*co*mpression–*dec*ompression filters) save only the differences between frames to the data stream, so that the background does not need to be repeated.

Every few frames, maybe 15 or so, the entire image is saved so that it is necessary to go back only to this "key frame" to reconstruct a particular image, rather than to the beginning of the film.

To a large extent, image compression is irrelevant as far as our purposes are concerned. Allowing for the problems of data size, we wish to work on an image with as much of the original detail left intact as possible.

Now we have captured a frame of image data, either in an array that we can access using subscripts in a high-level language or in a block of memory into which we construct a pointer to find the pixel we seek.

12.3.2 Smoothing a Binary Image

Whether the image is binary or grayscale, we will want to perform some sort of integration or differentiation on it to achieve a filter (in the control sense). The first operation that we will consider is smoothing, to remove spots and ragged edges.

The simplest way to do this is to consider each square block of 9 pixels. Take their average, and give that new value to the center pixel. If the image is binary, "taking the average" means that if 5 or more of the 9 pixels are light, the new pixel is light, otherwise it is dark.

You can see this in action on the Web at www.essmech.com/12/3/2.htm. Examples have been written in JavaScript, which has a syntax closely resembling that of C. More details of the implementation are given in Chapter 13.

The code that performs the smoothing is

```
function smooth() {
 var m, i, j, k, l;
 for(i = 1;i<cols;i++){              //for each point
   for(j = 1;j<rows;j++) {           //except the edges
      m = 0;                         //clear the total
      for (k = i - 1;k<=i + 1;k++) {
         for (l = j - 1;l<= j + 1;l++) {
            m = m + pic1 [k][l]; //add 9 values in 3x3
                                   //block
          }
      }
      if (m > 4) {                 //If majority are white
          pic2 [i][j] = 1;         //make pixel of pic2
                                   //light
      }else{
          pic2 [i][j] = 0;         //otherwise make it
                                   //dark
      };
   }
 }
}
```

In this example, the smoothing is applied several times and the image settles down to a shape without ragged edges.

When used on a grayscale image, this averaging technique has the effect of blurring the image. We will see another method in action later.

A disadvantage is that some special measures would be needed to process the outside boundaries of the image, since they have a row of neighbors missing.

12.3.3 Finding Edges

To find the outline of an object in the image, we must think in terms of differentiating it.

In the world of discrete samples, or pixels, differentiating becomes "differencing," taking the difference between a value and its neighbor. We can easily edit the code of the last example to be

```
function diffx() {
 var i, j, k, l;
 for(i = 1;i<cols;i++){       //for each point
   for(j = 1;j<rows;j++) {   //except the edges
      if ((pic[i+1][j] - pic[i][j]) > 0) {
          //If the pixel to the right is brighter
          pic2 [i][j] = 1;    //make pixel of pic2 light
      }else{
          pic2 [i][j] = 0;    //otherwise make it dark
      };
   }
 }
}
```

So, what does it do? We find a sort of negative shadow, where the lefthand edge is outlined in bright pixels while the rest are dark. This is certainly differentiating the image, but not in a way that is generally useful.

To try it for yourself, open the previous "smoothing" example and copy and paste this new function into the code window below the "smooth" function. Then add

```
diffx();
pic1=pic2;
showpic1();
```

below the rest of the code and it is ready to run.

We could embroider the code to replace the "greater than" sign ">" with a "not equal" sign "<>" to get shadows on both left and right edges, but we would also have to OR a test on the vertical difference if we wish to have an outline all round the object. But there is a more methodical way to do it.

We can think in terms of *convolution*, a process where one array of values is applied to filter another by multiplying and adding corresponding elements, planting a result, and then moving the pointer of the first array to the next pixel.

In this differencing case, the array of the filter is just [–1,1], or perhaps we should write it as [0,–1,1] since it is then clear that the "result" must be written to the central pixel position.

So, if we start with a row of pixels

```
0  0  0  0  1  0  1  1  1  1  1  0  0  0
```

and apply the filter

```
[ 0  -1  1]
```

we will first get the calculation

```
0  0  0  1  -1  1  0  0  0  0  -1  0  0
```

which results in a new string of pixels

```
0  0  0  1  0  1  0  0  0  0  0  0  0
```

when we set anything less than 1 to be a zero pixel.

Each row of pixels will be processed independently to give the full image.

If we want to detect both sides of the object, however, we should instead be looking at the second derivative, or the second difference. Now, convolving the filter with itself, we get

```
[-1  2  -1]
```

meaning "twice this pixel, minus the left and right neighbors."

When we apply it to

```
0  0  0  0  1  0  1  1  1  1  1  0  0  0
```

we get numbers

```
0  0  0  -1  2  -2  1  0  0  0  1  -1  0  0
```

which become pixels

```
0  0  0  0  1  0  1  0  0  0  1  0  0  0
```

We have succeeded in finding the edges, plus the isolated pixel "speckle." The new image is not shifted to the right or left, as it would be if using the previous filter.

So, can this convolution method help us to process the image in two dimensions?

12.3.4 Convolution and Array Filters

Consider the filter

```
 0 -1  0
-1  4 -1
 0 -1  0
```

This is the sum of the previous filter, padded out with a row of zeros top and bottom, added to its vertical counterpart. We can look at it pragmatically to deduce that the new pixel will be light only if the present pixel is light and not surrounded top and bottom, left and right by other bright pixels.

We must define our coefficients to be an array

```
filt=new Array(3);
filt[0]=new Array( 0,-1, 0);
filt[1]=new Array(-1, 4,-1);
filt[2]=new Array( 0,-1,-0);
```

and these coefficients can then be used in the routine

```
function filter() {
var m, i, j, k, l
  for(i = 1;i<cols;i++){      //for each point
    for(j = 1;j<rows;j++) {  //except edge
       m = 0;
       for (k = 0;k<=2;k++) {
          for (l = 0;l<= 2;l++) {
             m = m + pic1 [i+k-1][j+l-1]*filt[k][l];
          }
       }
       if (m >=1) {          //If total is positive
          pic2 [i][j] = 1;   //set pixel of pic2 to red
       }else{
          pic2 [i][j] = 0;   //set pixel of pic2 to black
       };
    Label(m,i,j);
    }
  }
}
```

See it in action at www.essmech.com/12/3/4.htm.

A host of filters are based on the use of such a 3×3 array of coefficients. We could have used this technique for the first smoothing example by defining the array to be

```
0.2  0.2  0.2
0.2  0.2  0.2
0.2  0.2  0.2
```

Only if 5 pixels in the array of 9 are bright will the total reach the value that we have set for the threshold. We could indeed try other values than 0.2. The value 0.25 would let us set the criterion at 4 bright pixels.

But returning to edge finders, our criterion could be that a bright pixel would survive as an edge unless surrounded completely by other bright pixels. The filter would be

```
-1  -1  -1
-1   8  -1
-1  -1  -1
```

Perhaps we want the edge to be marked in the "sea" that surrounds the "island":

```
1   1  1
1  -8  1
1   1  1
```

The choices are endless. You can try out any you think of on the Web.

I must repeat that these operations will change the appearance of an image, reducing it to spots with the appearance of lines at the edges of any "blobs," but a lot more has to be done before a line can considered as a "path" around the object.

12.3.5 Smoothing Grayscale Images

To see an array filter in action on a grayscale image, see the first Web example at www.essmech.com/12/3/5.htm.

The filter in this case is

```
 1,  -2,   1
-2,   4,  -2
 1,  -2,   1
```

giving the result that would be obtained if first the horizontal "second difference" operator [–1, 2, –1] were applied, followed by its vertical counterpart.

It is clear that by taking a second difference, it has eliminated the "graded" background that would give problems if a single threshold had to be set.

In control terms, the convolution that we are performing is termed a *finite impulse response* filter. The distance of the influence of any sample is limited, in this case to its neighbor. Broader filters could be set up, 5×5 or maybe 7×7, but the computational effort increases with the square of their size. Alternatively smaller filters could be used repetitively, so that the influence will "spread out" one pixel at a time.

There is another approach. In control theory, we saw that a lowpass filter (see, e.g., Fig. 12.1) would smooth a time series. Such a filter might take the form

```
for i = 0 to n
   xslow = xslow + (x(i) -xslow) / k
   x(i) = xslow
next i
```

where k determines the time constant.

But this "smears" the waveform to the right and would similarly smear an image. In real time, we can process a time series in only one direction, but here we have a captive image. We can follow up the left-to-right smoothing with another smoothing right-to-left that will exactly cancel out the smearing, while leaving the blurring in place. This approach, using a two-way filter, is shown in Figure 12.2.

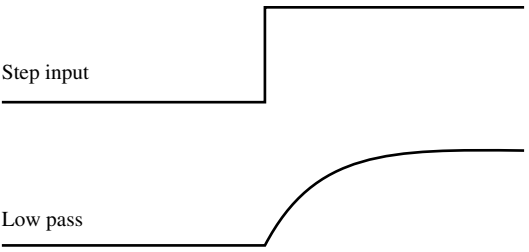

Figure 12.1 *Lowpass filter applied to a step.*

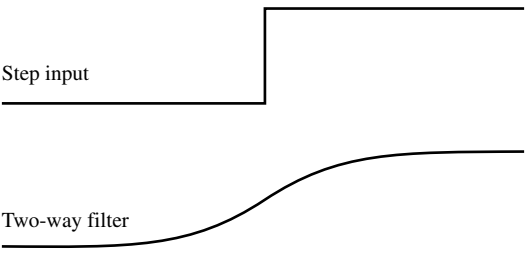

Figure 12.2 *Two-way filter applied to a step.*

Having blurred the image horizontally, we get the final effect by applying a similar vertical filter. See the result in the second Web example at www.essmech.com/12/3/5.htm.

12.3.6 Sharpening a Grayscale Image

In control theory, we used a *highpass* filter to approximate to differentiating a signal. We saw that we could construct such a filter by first making a lowpass filter, then subtracting the smoothed version from the original signal. The same principle applies to sharpening an image.

We have just seen how to use an *infinite impulse response* filter, our simple lowpass filter, for smoothing an image. We apply it left-to-right, right-to-left, top-to-bottom, and finally bottom-to-top.

When we subtract the smoothed version from the original image, we have a sharpened image in which the edges are enhanced. If this difference image requires more contrast, we can multiply the values to enhance it.

See the images shown in Figure 12.3 in action on the Web page at www.essmech.com/12/3/6.htm.

12.3.7 Edge Tracing

In the mid-1980s, we had attached a primitive stepper motor robot to a computer. A lens and a simple photocell were then added to the gripper of the robot to give a single point of vision. The robot could be moved to scan the photocell over the view, thus building up an image. It was somewhat slow and unwieldy, but it made a student project.

Could the image be scanned some swifter way? If the vision point could be driven to follow an edge in the scene, we might be able to trace out the boundaries of objects, inspecting a very few pixels within the whole scene. Today the image is captured in a flash, but analyzing the image in a logical and economic way still has the same virtues of speed.

First, the spot must be driven across the scene to detect the first change in brightness. When it has found an edge, it can start to track it. A comparison of brightness against a threshold gives a binary decision for each spot, black or white.

The spot has eight notional directions of travel, defined by eight points of the compass. Suppose that the present direction is west and the present spot is white. The spot moves one step north. If the new spot is black, the edge has been crossed, so the spot moves back southwest. If the new spot is white again, the boundary has been followed one step west. The cycle can repeat for as long as the boundary leads west and the spot continues to "stitch" along it.

Suppose, however, then another black spot is found on the "back step". Then the boundary might have curved to the south. The direction of travel is turned 45° anticlockwise and another backward step is taken, now due south.

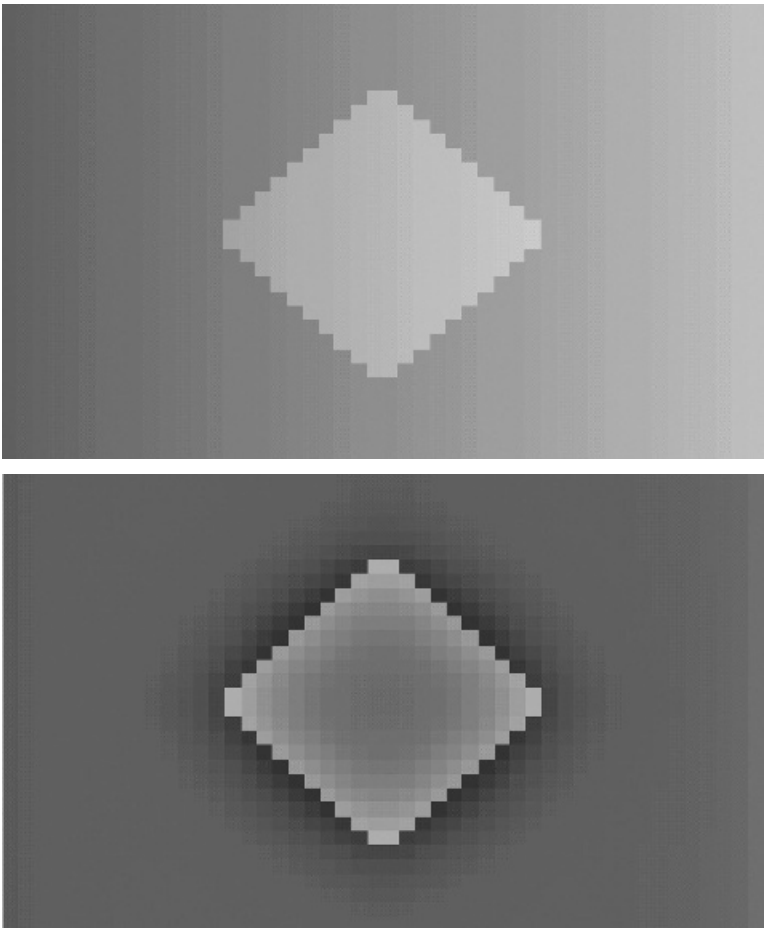

Figure 12.3 *Screen grab of edge enhancement.*

If there is a change to white, the stitching can continue, now in the new direction; otherwise yet another anticlockwise turn and backward step are taken. Eventually the movement must find an edge again, even if it has to complete the semicircle back to a previous point.

Similarly, if a forward step fails to find a change, the turn is clockwise and another step is taken.

Each time a change is detected, the coordinates are noted, giving a sequence of points that track in order around the boundary. In this case, the path will track anticlockwise around a white object, or clockwise around a black one.

An image created by edge stitching is shown in Figure 12.4.

The outline of the algorithm in QBasic is as follows:

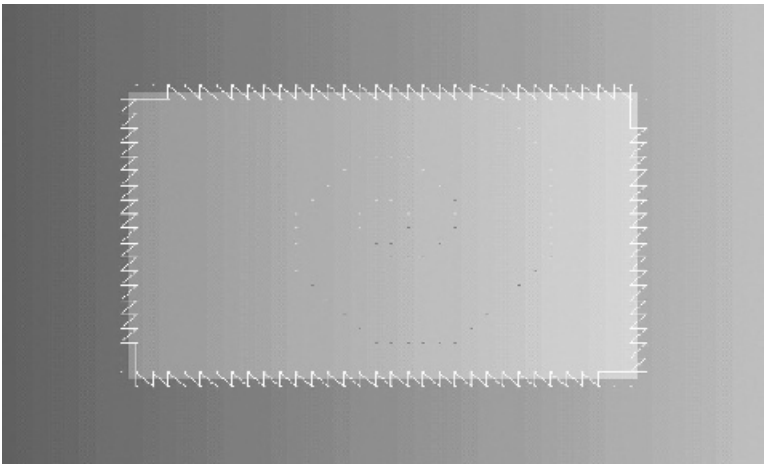

Figure 12.4 *Illustration of edge stitching.*

```
DO                          'This is the search algorithm
 IF here = white THEN       'if first point is white
   here = look(d)           'look at a second point in
                            direction d
   IF here = white THEN     'if it's the same, white, then
     turn 1                 'turn clockwise for next move
   ELSE                     'otherwise you've crossed a
                            'threshold
     notepoint              'so mark it
   END IF
 ELSE                       'if first point was black,
   here = look(d - 3)       'look at a second point in
                            'direction d-3
   IF here = black THEN     'if it's the same, black, then
      turn -1               'turn anticlockwise
   ELSE                     'otherwise you've crossed a
                            'threshold
      notepoint             'so mark it
   END IF
 END IF
LOOP UNTIL beenthere > 0  'keep going until you hit an
                           'old marked point
```

From this fundamental principle, a number of additions are needed to make the routine work.

If the starting point is not near an edge, the routine will just go round in small circles forever. The first modification is to count the number of steps

since the last boundary crossing. If this exceeds four, the direction is not allowed to change until after two steps, then three, and so on. The length of a "straight" increases by one every eight steps. Now, after the first semicircle, the search expands in a spiral.

The second modification enables the program to adapt the threshold to find subtle shades. Two variables hold the lightest and the darkest values found so far. The threshold level is set midway between these levels.

To adapt to local changes, the "lightest" value is reduced by a small amount at each step while the "darkest" is increased. They will ramp until they hit the values being found locally, while variations in the level will keep them apart. If the "gap" becomes too small, boundary crossings are ignored, so that the search spirals out to find a more prominent edge. If the search arrives at a point already tagged, then it is ended.

That describes the details of the technique, but what does it achieve?

The boundary is revealed as an ordered sequence of points, forming a *Freeman chain*.

12.3.8 Analyzing Boundaries

The sequence of points found by the "stitching" method can equally be regarded as a chain of vectors joining one point to the next. Each vector has a length and a direction. If the vectors are added in pairs or more, the "ragged" nature of an oblique edge will be smoothed.

We can take the sum of the lengths of the vectors from the starting point to obtain the distance s moved around the perimeter, and against this we can plot the angle ψ of the vector, the direction of the perimeter at that point, to get an $s-\psi$ (s-psi) curve.

With this curve the shape data can be reduced to a few hundred bytes of data, say, 256 or 512, representing the tangent directions at equal intervals around the perimeter. By comparing this shape data against templates of the same length, we can recognize the shape.

Object recognition might at first seem a daunting task. Even if the size of the object is known, it can be rotated to any angle and be located anywhere in the picture. If you are searching in 0.5 mbyte of data by a correlation method, the number of computing operations is huge. Given an $s-\psi$ plot, most of the task is already done.

From start to finish the angle will change by 2π. This plot will be the same, wherever the object is in the picture. However, the object could be lying at a different angle. In this case the plot will still be the same, if regarded as a cyclic function, but will have a constant added to the angle value. If the object is "flipped," the function will be reversed.

In each case, the task of matching the unknown object against a template is a simple case of examining a few hundred data points. We still have to consider the match as a correlation, shifting the starting point around the template, unless we can find a strategy for determining a starting point. Even so, the computing load is relatively modest.

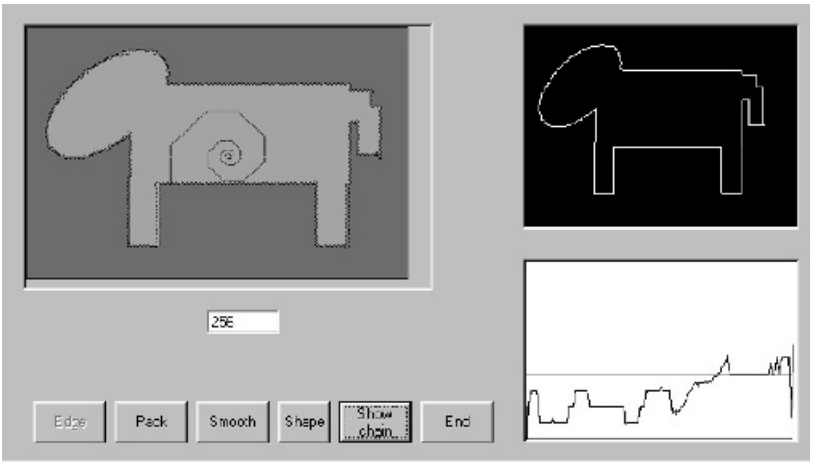

Figure 12.5 *Boundary tracing and s–ψ curve.*

Since ψ is plotted against the proportion of the distance traveled around the perimeter, size does not matter. Objects of the same shape will have the same data, however big they are.

An example on the Website shows a shape being traced. The $s-\psi$ curve is then generated, smoothed, and reduced to a fixed length. At each stage, the shape is reconstructed so that it can be seen how far the smoothing might distort the shape.

See http://www.EssMech.com/12/3/8.htm and Figure 12.5.

Of course, clues other than shape can be used for a comparison, once the binary decision has been made to discriminate between the pixels of the object and the background:

- The area of the object can be found by counting pixels.
- By searching for boundaries within the object, any "holes" can be counted.
- By testing the "width" of the object as it is rotated, the ratio between maximum and minimum can be found.

This is not the only format for shape data. By taking moments, the center of gravity can be found. Now, by tracing out along radii from the center of gravity, radius length can be found as a function of radius angle. However, a curve may have a reentrant "hook" so that a radius can cut it in more than one place. The plot of radius versus angle is then no longer single-valued and is therefore difficult to list as a computer function.

This is just a glimpse of the vast range of possibilities that are opening up in vision sensing. Any more would go beyond the essentials.