CHAPTER 9

Image Parts and Segmentation

Parts and Segments

This chapter focuses on how to isolate objects or parts of objects from the rest of the image. The reasons for doing this should be obvious. In video security, for example, the camera mostly looks out on the same boring background, which really isn't of interest. What is of interest is when people or vehicles enter the scene, or when something is left in the scene that wasn't there before. We want to isolate those events and to be able to ignore the endless hours when nothing is changing.

Beyond separating foreground objects from the rest of the image, there are many situations where we want to separate out parts of objects, such as isolating just the face or the hands of a person. We might also want to preprocess an image into meaningful *super pixels*, which are segments of an image that contain things like limbs, hair, face, torso, tree leaves, lake, path, lawn and so on. Using super pixels saves on computation; for example, when running an object classifier over the image, we only need search a box around each super pixel. We might only track the motion of these larger patches and not every point inside.

We saw several image segmentation algorithms when we discussed image processing in Chapter 5. The routines covered in that chapter included image morphology, flood fill, threshold, and pyramid segmentation. This chapter examines other algorithms that deal with finding, filling and isolating objects and object parts in an image. We start with separating foreground objects from learned background scenes. These background modeling functions are not built-in OpenCV functions; rather, they are examples of how we can leverage OpenCV functions to implement more complex algorithms.

Background Subtraction

Because of its simplicity and because camera locations are fixed in many contexts, *background subtraction* (aka *background differencing*) is probably the most fundamental image processing operation for video security applications. Toyama, Krumm, Brumitt, and Meyers give a good overview and comparison of many techniques [Toyama99]. In order to perform background subtraction, we first must "learn" a model of the background.

Once learned, this *background model* is compared against the current image and then the known background parts are subtracted away. The objects left after subtraction are presumably new foreground objects.

Of course "background" is an ill-defined concept that varies by application. For example, if you are watching a highway, perhaps average traffic flow should be considered background. Normally, background is considered to be any static or periodically moving parts of a scene that remain static or periodic over the period of interest. The whole ensemble may have time-varying components, such as trees waving in morning and evening wind but standing still at noon. Two common but substantially distinct environment categories that are likely to be encountered are indoor and outdoor scenes. We are interested in tools that will help us in both of these environments. First we will discuss the weaknesses of typical background models and then will move on to discuss higher-level scene models. Next we present a quick method that is mostly good for indoor static background scenes whose lighting doesn't change much. We will follow this by a "codebook" method that is slightly slower but can work in both outdoor and indoor scenes; it allows for periodic movements (such as trees waving in the wind) and for lighting to change slowly or periodically. This method is also tolerant to learning the background even when there are occasional foreground objects moving by. We'll top this off by another discussion of connected components (first seen in Chapter 5) in the context of cleaning up foreground object detection. Finally, we'll compare the quick background method against the codebook background method.

Weaknesses of Background Subtraction

Although the background modeling methods mentioned here work fairly well for simple scenes, they suffer from an assumption that is often violated: that all the pixels are independent. The methods we describe learn a model for the variations a pixel experiences without considering neighboring pixels. In order to take surrounding pixels into account, we could learn a multipart model, a simple example of which would be an extension of our basic independent pixel model to include a rudimentary sense of the brightness of neighboring pixels. In this case, we use the brightness of neighboring pixels to distinguish when neighboring pixel values are relatively bright or dim. We then learn effectively two models for the individual pixel: one for when the surrounding pixels are bright and one for when the surrounding pixels are dim. In this way, we have a model that takes into account the surrounding *context*. But this comes at the cost of twice as much memory use and more computation, since we now need different values for when the surrounding pixels are bright or dim. We also need twice as much data to fill out this two-state model. We can generalize the idea of "high" and "low" contexts to a multidimensional histogram of single and surrounding pixel intensities as well as make it even more complex by doing all this over a few time steps. Of course, this richer model over space and time would require still more memory, more collected data samples, and more computational resources.

Because of these extra costs, the more complex models are usually avoided. We can often more efficiently invest our resources in cleaning up the *false positive* pixels that result when the independent pixel assumption is violated. The cleanup takes the form of image processing operations (`cvErode()`, `cvDilate()`, and `cvFloodFill()`, mostly) that eliminate stray patches of pixels. We've discussed these routines previously (Chapter 5) in the context of finding large and compact[*] *connected components* within noisy data. We will employ connected components again in this chapter and so, for now, will restrict our discussion to approaches that assume pixels vary independently.

Scene Modeling

How do we define background and foreground? If we're watching a parking lot and a car comes in to park, then this car is a new foreground object. But should it stay foreground forever? How about a trash can that was moved? It will show up as foreground in two places: the place it was moved to and the "hole" it was moved from. How do we tell the difference? And again, how long should the trash can (and its hole) remain foreground? If we are modeling a dark room and suddenly someone turns on a light, should the whole room become foreground? To answer these questions, we need a higher-level "scene" model, in which we define multiple levels between foreground and background states, and a timing-based method of slowly relegating unmoving foreground patches to background patches. We will also have to detect and create a new model when there is a global change in a scene.

In general, a scene model might contain multiple layers, from "new foreground" to older foreground on down to background. There might also be some motion detection so that, when an object is moved, we can identify both its "positive" aspect (its new location) and its "negative" aspect (its old location, the "hole").

In this way, a new foreground object would be put in the "new foreground" object level and marked as a positive object or a hole. In areas where there was no foreground object, we could continue updating our background model. If a foreground object does not move for a given time, it is demoted to "older foreground," where its pixel statistics are provisionally learned until its learned model joins the learned background model.

For global change detection such as turning on a light in a room, we might use global frame differencing. For example, if many pixels change at once then we could classify it as a global rather than local change and then switch to using a model for the new situation.

A Slice of Pixels

Before we go on to modeling pixel changes, let's get an idea of what pixels in an image can look like over time. Consider a camera looking out a window to a scene of a tree blowing in the wind. Figure 9-1 shows what the pixels in a given line segment of the image look like over 60 frames. We wish to model these kinds of fluctuations. Before doing so, however, we make a small digression to discuss how we sampled this line because it's a generally useful trick for creating features and for debugging.

[*] Here we are using mathematician's definition of "compact," which has nothing to do with size.

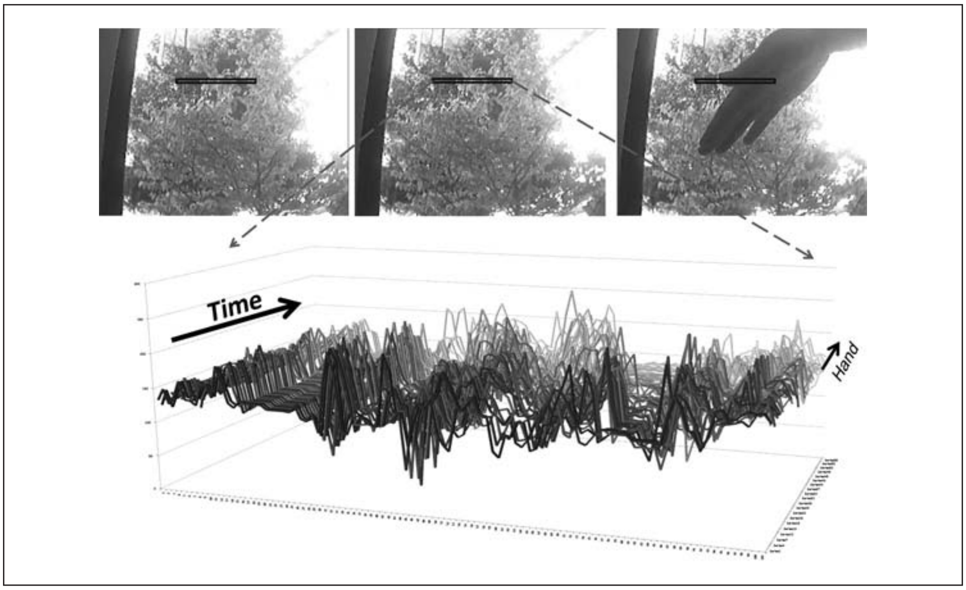

Figure 9-1. Fluctuations of a line of pixels in a scene of a tree moving in the wind over 60 frames: some dark areas (upper left) are quite stable, whereas moving branches (upper center) can vary widely

OpenCV has functions that make it easy to sample an arbitrary line of pixels. The line sampling functions are cvInitLineIterator() and CV_NEXT_LINE_POINT(). The function prototype for cvInitLineIterator() is:

```
int cvInitLineIterator(
    const CvArr*    image,
    CvPoint         pt1,
    CvPoint         pt2,
    CvLineIterator* line_iterator,
    int             connectivity  = 8,
    int             left_to_right = 0
);
```

The input image may be of any type or number of channels. Points pt1 and pt2 are the ends of the line segment. The iterator line_iterator just steps through, pointing to the pixels along the line between the points. In the case of multichannel images, each call to CV_NEXT_LINE_POINT() moves the line_iterator to the next pixel. All the channels are available at once as line_iterator.ptr[0], line_iterator.ptr[1], and so forth. The connectivity can be 4 (the line can step right, left, up, or down) or 8 (the line can additionally step along the diagonals). Finally if left_to_right is set to 0 (false), then line_iterator scans from pt1 to pt2; otherwise, it will go from the leftmost to the rightmost point.* The cvInitLineIterator() function returns the number of points that will be

* The left_to_right flag was introduced because a discrete line drawn from pt1 to pt2 does not always match the line from pt2 to pt1. Therefore, setting this flag gives the user a consistent rasterization regardless of the pt1, pt2 order.

iterated over for that line. A companion macro, `CV_NEXT_LINE_POINT(line_iterator)`, steps the iterator from one pixel to another.

Let's take a second to look at how this method can be used to extract some data from a file (Example 9-1). Then we can re-examine Figure 9-1 in terms of the resulting data from that movie file.

Example 9-1. Reading out the RGB values of all pixels in one row of a video and accumulating those values into three separate files

```
// STORE TO DISK A LINE SEGMENT OF BGR PIXELS FROM pt1 to pt2.
//
CvCapture*     capture = cvCreateFileCapture( argv[1] );
int            max_buffer;
IplImage*      rawImage;
int            r[10000],g[10000],b[10000];
CvLineIterator iterator;

FILE *fptrb = fopen("blines.csv","w"); // Store the data here
FILE *fptrg = fopen("glines.csv","w"); // for each color channel
FILE *fptrr = fopen("rlines.csv","w");

// MAIN PROCESSING LOOP:
//
for(;;){
    if( !cvGrabFrame( capture ))
          break;
    rawImage = cvRetrieveFrame( capture );
    max_buffer = cvInitLineIterator(rawImage,pt1,pt2,&iterator,8,0);
    for(int j=0; j<max_buffer; j++){

        fprintf(fptrb,"%d,", iterator.ptr[0]); //Write blue value
        fprintf(fptrg,"%d,", iterator.ptr[1]); //green
        fprintf(fptrr,"%d,", iterator.ptr[2]); //red

        iterator.ptr[2] = 255;  //Mark this sample in red

        CV_NEXT_LINE_POINT(iterator); //Step to the next pixel
    }
    // OUTPUT THE DATA IN ROWS:
    //
    fprintf(fptrb,"/n");fprintf(fptrg,"/n");fprintf(fptrr,"/n");
}
// CLEAN UP:
//
fclose(fptrb); fclose(fptrg); fclose(fptrr);
cvReleaseCapture( &capture );
```

We could have made the line sampling even easier, as follows:

```
int cvSampleLine(
    const CvArr* image,
    CvPoint      pt1,
    CvPoint      pt2,
    void*        buffer,
    int          connectivity = 8
);
```

This function simply wraps the function `cvInitLineIterator()` together with the macro `CV_NEXT_LINE_POINT(line_iterator)` from before. It samples from `pt1` to `pt2`; then you pass it a pointer to a `buffer` of the right type and of length $N_{\text{channels}} \times \max(|\text{pt2}_x - \text{pt2}_x| + 1, |\text{pt2}_y - \text{pt2}_y| + 1)$. Just like the line iterator, `cvSampleLine()` steps through each channel of each pixel in a multichannel image before moving to the next pixel. The function returns the number of actual elements it filled in the `buffer`.

We are now ready to move on to some methods for modeling the kinds of pixel fluctuations seen in Figure 9-1. As we move from simple to increasingly complex models, we shall restrict our attention to those models that will run in real time and within reasonable memory constraints.

Frame Differencing

The very simplest background subtraction method is to subtract one frame from another (possibly several frames later) and then label any difference that is "big enough" the foreground. This process tends to catch the edges of moving objects. For simplicity, let's say we have three single-channel images: `frameTime1`, `frameTime2`, and `frameForeground`. The image `frameTime1` is filled with an older grayscale image, and `frameTime2` is filled with the current grayscale image. We could then use the following code to detect the magnitude (absolute value) of foreground differences in `frameForeground`:

```
    cvAbsDiff(
        frameTime1,
        frameTime2,
        frameForeground
    );
```

Because pixel values always exhibit noise and fluctuations, we should ignore (set to 0) small differences (say, less than 15), and mark the rest as big differences (set to 255):

```
    cvThreshold(
        frameForeground,
        frameForeground,
        15,
        255,
        CV_THRESH_BINARY
    );
```

The image `frameForeground` then marks candidate foreground objects as 255 and background pixels as 0. We need to clean up small noise areas as discussed earlier; we might do this with `cvErode()` or by using connected components. For color images, we could use the same code for each color channel and then combine the channels with `cvOr()`. This method is much too simple for most applications other than merely indicating regions of motion. For a more effective background model we need to keep some statistics about the means and average differences of pixels in the scene. You can look ahead to the section entitled "A quick test" to see examples of frame differencing in Figures 9-5 and 9-6.

Averaging Background Method

The averaging method basically learns the average and standard deviation (or similarly, but computationally faster, the average difference) of each pixel as its model of the background.

Consider the pixel line from Figure 9-1. Instead of plotting one sequence of values for each frame (as we did in that figure), we can represent the variations of each pixel throughout the video in terms of an average and average differences (Figure 9-2). In the same video, a foreground object (which is, in fact, a hand) passes in front of the camera. That foreground object is not nearly as bright as the sky and tree in the background. The brightness of the hand is also shown in the figure.

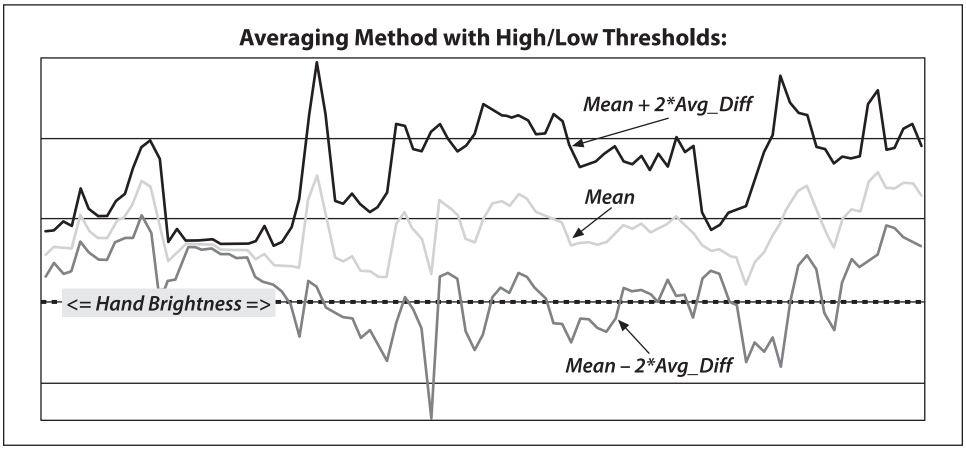

Figure 9-2. Data from Figure 9-1 presented in terms of average differences: an object (a hand) that passes in front of the camera is somewhat darker, and the brightness of that object is reflected in the graph

The averaging method makes use of four OpenCV routines: `cvAcc()`, to accumulate images over time; `cvAbsDiff()`, to accumulate frame-to-frame image differences over time; `cvInRange()`, to segment the image (once a background model has been learned) into foreground and background regions; and `cvOr()`, to compile segmentations from different color channels into a single mask image. Because this is a rather long code example, we will break it into pieces and discuss each piece in turn.

First, we create pointers for the various scratch and statistics-keeping images we will need along the way. It will prove helpful to sort these pointers according to the type of images they will later hold.

```
//Global storage
//
//Float, 3-channel images
//
IplImage *IavgF,*IdiffF, *IprevF, *IhiF, *IlowF;
IplImage *Iscratch,*Iscratch2;

//Float, 1-channel images
//
IplImage *Igray1,*Igray2, *Igray3;
IplImage *Ilow1,  *Ilow2, *Ilow3;
IplImage *Ihi1,   *Ihi2,  *Ihi3;

// Byte, 1-channel image
//
IplImage *Imaskt;

//Counts number of images learned for averaging later.
//
float Icount;
```

Next we create a single call to allocate all the necessary intermediate images. For convenience we pass in a single image (from our video) that can be used as a reference for sizing the intermediate images.

```
// I is just a sample image for allocation purposes
// (passed in for sizing)
//
void AllocateImages( IplImage* I ){

  CvSize sz = cvGetSize( I );

  IavgF     = cvCreateImage( sz, IPL_DEPTH_32F, 3 );
  IdiffF    = cvCreateImage( sz, IPL_DEPTH_32F, 3 );
  IprevF    = cvCreateImage( sz, IPL_DEPTH_32F, 3 );
  IhiF      = cvCreateImage( sz, IPL_DEPTH_32F, 3 );
  IlowF     = cvCreateImage( sz, IPL_DEPTH_32F, 3 );
  Ilow1     = cvCreateImage( sz, IPL_DEPTH_32F, 1 );
  Ilow2     = cvCreateImage( sz, IPL_DEPTH_32F, 1 );
  Ilow3     = cvCreateImage( sz, IPL_DEPTH_32F, 1 );
  Ihi1      = cvCreateImage( sz, IPL_DEPTH_32F, 1 );
  Ihi2      = cvCreateImage( sz, IPL_DEPTH_32F, 1 );
  Ihi3      = cvCreateImage( sz, IPL_DEPTH_32F, 1 );
  cvZero( IavgF );
  cvZero( IdiffF );
  cvZero( IprevF );
  cvZero( IhiF );
  cvZero( IlowF );
  Icount    = 0.00001; //Protect against divide by zero

  Iscratch  = cvCreateImage( sz, IPL_DEPTH_32F, 3 );
  Iscratch2 = cvCreateImage( sz, IPL_DEPTH_32F, 3 );
  Igray1    = cvCreateImage( sz, IPL_DEPTH_32F, 1 );
  Igray2    = cvCreateImage( sz, IPL_DEPTH_32F, 1 );
  Igray3    = cvCreateImage( sz, IPL_DEPTH_32F, 1 );
  Imaskt    = cvCreateImage( sz, IPL_DEPTH_8U,  1 );
  cvZero( Iscratch );
  cvZero( Iscratch2 );
}
```

In the next piece of code, we learn the accumulated background image and the accumulated absolute value of frame-to-frame image differences (a computationally quicker proxy* for learning the standard deviation of the image pixels). This is typically called for 30 to 1,000 frames, sometimes taking just a few frames from each second or sometimes taking all available frames. The routine will be called with a three-color channel image of depth 8 bits.

```
// Learn the background statistics for one more frame
// I is a color sample of the background, 3-channel, 8u
//
void accumulateBackground( IplImage *I ){

   static int first = 1;                 // nb. Not thread safe
   cvCvtScale(  I, Iscratch, 1, 0 );     // convert to float
   if( !first ){
      cvAcc( Iscratch, IavgF );
      cvAbsDiff( Iscratch, IprevF, Iscratch2 );
      cvAcc( Iscratch2, IdiffF );
      Icount += 1.0;
   }
   first = 0;
   cvCopy( Iscratch, IprevF );

}
```

We first use `cvCvtScale()` to turn the raw background 8-bit-per-channel, three-color-channel image into a floating-point three-channel image. We then accumulate the raw floating-point images into `IavgF`. Next, we calculate the frame-to-frame absolute difference image using `cvAbsDiff()` and accumulate that into image `IdiffF`. Each time we accumulate these images, we increment the image count `Icount`, a global, to use for averaging later.

Once we have accumulated enough frames, we convert them into a statistical model of the background. That is, we compute the means and deviation measures (the average absolute differences) of each pixel:

```
void createModelsfromStats() {

    cvConvertScale( IavgF,  IavgF,( double)(1.0/Icount) );
    cvConvertScale( IdiffF, IdiffF,(double)(1.0/Icount) );

    //Make sure diff is always something
    //
    cvAddS( IdiffF, cvScalar( 1.0, 1.0, 1.0), IdiffF );
    setHighThreshold( 7.0 );
    setLowThreshold( 6.0 );
}
```

* Notice our use of the word "proxy." Average difference is not mathematically equivalent to standard deviation, but in this context it is close enough to yield results of similar quality. The advantage of average difference is that it is slightly faster to compute than standard deviation. With only a tiny modification of the code example you can use standard deviations instead and compare the quality of the final results for yourself; we'll discuss this more explicitly later in this section.

In this code, cvConvertScale() calculates the average raw and absolute difference images by dividing by the number of input images accumulated. As a precaution, we ensure that the average difference image is at least 1; we'll need to scale this factor when calculating a foreground-background threshold and would like to avoid the degenerate case in which these two thresholds could become equal.

Both setHighThreshold() and setLowThreshold() are utility functions that set a threshold based on the frame-to-frame average absolute differences. The call setHighThreshold(7.0) fixes a threshold such that any value that is 7 times the average frame-to-frame absolute difference above the average value for that pixel is considered foreground; likewise, setLowThreshold(6.0) sets a threshold bound that is 6 times the average frame-to-frame absolute difference below the average value for that pixel. Within this range around the pixel's average value, objects are considered to be background. These threshold functions are:

```
void setHighThreshold( float scale )
{
   cvConvertScale( IdiffF, Iscratch, scale );
   cvAdd( Iscratch, IavgF, IhiF );
   cvSplit( IhiF, Ihi1, Ihi2, Ihi3, 0 );
}

void setLowThreshold( float scale )
{
   cvConvertScale( IdiffF, Iscratch, scale );
   cvSub( IavgF, Iscratch, IlowF );
   cvSplit( IlowF, Ilow1, Ilow2, Ilow3, 0 );
}
```

Again, in setLowThreshold() and setHighThreshold() we use cvConvertScale() to multiply the values prior to adding or subtracting these ranges relative to IavgF. This action sets the IhiF and IlowF range for each channel in the image via cvSplit().

Once we have our background model, complete with high and low thresholds, we use it to segment the image into foreground (things not "explained" by the background image) and the background (anything that fits within the high and low thresholds of our background model). Segmentation is done by calling:

```
// Create a binary: 0,255 mask where 255 means foreground pixel
// I      Input image, 3-channel, 8u
// Imask  Mask image to be created, 1-channel 8u
//
void backgroundDiff(
  IplImage *I,
  IplImage *Imask
) {
  cvCvtScale(I,Iscratch,1,0); // To float;
  cvSplit( Iscratch, Igray1,Igray2,Igray3, 0 );

  //Channel 1
  //
  cvInRange(Igray1,Ilow1,Ihi1,Imask);
```

```
   //Channel 2
   //
   cvInRange(Igray2,Ilow2,Ihi2,Imaskt);
   cvOr(Imask,Imaskt,Imask);

   //Channel 3
   //
   cvInRange(Igray3,Ilow3,Ihi3,Imaskt);
   cvOr(Imask,Imaskt,Imask)

   //Finally, invert the results
   //
   cvSubRS( Imask, 255, Imask);
}
```

This function first converts the input image I (the image to be segmented) into a floating-point image by calling cvCvtScale(). We then convert the three-channel image into separate one-channel image planes using cvSplit(). These color channel planes are then checked to see if they are within the high and low range of the average background pixel via the cvInRange() function, which sets the grayscale 8-bit depth image Imaskt to max (255) when it's in range and to 0 otherwise. For each color channel we logically OR the segmentation results into a mask image Imask, since strong differences in any color channel are considered evidence of a foreground pixel here. Finally, we invert Imask using cvSubRS(), because foreground should be the values out of range, not in range. The mask image is the output result.

For completeness, we need to release the image memory once we're finished using the background model:

```
void DeallocateImages()
{
   cvReleaseImage( &IavgF);
   cvReleaseImage( &IdiffF );
   cvReleaseImage( &IprevF );
   cvReleaseImage( &IhiF );
   cvReleaseImage( &IlowF );
   cvReleaseImage( &Ilow1 );
   cvReleaseImage( &Ilow2 );
   cvReleaseImage( &Ilow3 );
   cvReleaseImage( &Ihi1 );
   cvReleaseImage( &Ihi2 );
   cvReleaseImage( &Ihi3 );
   cvReleaseImage( &Iscratch );
   cvReleaseImage( &Iscratch2 );
   cvReleaseImage( &Igray1 );
   cvReleaseImage( &Igray2 );
   cvReleaseImage( &Igray3 );
   cvReleaseImage( &Imaskt);
}
```

We've just seen a simple method of learning background scenes and segmenting foreground objects. It will work well only with scenes that do not contain moving background components (like a waving curtain or waving trees). It also assumes that the lighting remains fairly constant (as in indoor static scenes). You can look ahead to Figure 9-5 to check the performance of this averaging method.

Accumulating means, variances, and covariances

The averaging background method just described made use of one accumulation function, cvAcc(). It is one of a group of helper functions for accumulating sums of images, squared images, multiplied images, or average images from which we can compute basic statistics (means, variances, covariances) for all or part of a scene. In this section, we'll look at the other functions in this group.

The images in any given function must all have the same width and height. In each function, the input images named image, image1, or image2 can be one- or three-channel byte (8-bit) or floating-point (32F) image arrays. The output accumulation images named sum, sqsum, or acc can be either single-precision (32F) or double-precision (64F) arrays. In the accumulation functions, the mask image (if present) restricts processing to only those locations where the mask pixels are nonzero.

Finding the mean. To compute a mean value for each pixel across a large set of images, the easiest method is to add them all up using cvAcc() and then divide by the total number of images to obtain the mean.

```
void cvAcc(
  const Cvrr*  image,
  CvArr*       sum,
  const CvArr* mask = NULL
);
```

An alternative that is often useful is to use a *running average*.

```
void cvRunningAvg(
  const CvArr* image,
  CvArr*       acc,
  double       alpha,
  const CvArr* mask = NULL
);
```

The running average is given by the following formula:

$$\text{acc}(x,y) = (1-\alpha)\cdot\text{acc}(x,y) + \alpha\cdot\text{image}(x,y) \quad \text{if } \text{mask}(x,y) \neq 0$$

For a constant value of α, running averages are not equivalent to the result of summing with cvAcc(). To see this, simply consider adding three numbers (2, 3, and 4) with α set to 0.5. If we were to accumulate them with cvAcc(), then the sum would be 9 and the average 3. If we were to accumulate them with cvRunningAverage(), the first sum would give $0.5 \times 2 + 0.5 \times 3 = 2.5$ and then adding the third term would give $0.5 \times 2.5 + 0.5 \times 4 = 3.25$. The reason the second number is larger is that the most recent contributions are given more weight than those from farther in the past. Such a running average is thus also called a *tracker*. The parameter α essentially sets the amount of time necessary for the influence of a previous frame to fade.

Finding the variance. We can also accumulate squared images, which will allow us to compute quickly the variance of individual pixels.

```
void cvSquareAcc(
  const CvArr* image,
  CvArr*       sqsum,
  const CvArr* mask = NULL
);
```

You may recall from your last class in statistics that the variance of a finite population is defined by the formula:

$$\sigma^2 = \frac{1}{N}\sum_{i=0}^{N-1}(x_i - \overline{x})^2$$

where $\bar{x}$ is the mean of x for all N samples. The problem with this formula is that it entails making one pass through the images to compute $\bar{x}$ and then a second pass to compute σ^2. A little algebra should allow you to convince yourself that the following formula will work just as well:

$$\sigma^2 = \left(\frac{1}{N}\sum_{i=0}^{N-1}x_i^2\right) - \left(\frac{1}{N}\sum_{i=0}^{N-1}x_i\right)^2$$

Using this form, we can accumulate both the pixel values and their squares in a single pass. Then, the variance of a single pixel is just the average of the square minus the square of the average.

Finding the covariance. We can also see how images vary over time by selecting a specific *lag* and then multiplying the current image by the image from the past that corresponds to the given lag. The function `cvMultiplyAcc()` will perform a pixelwise multiplication of the two images and then add the result to the "running total" in `acc`:

```
void cvMultiplyAcc(
  const CvArr* image1,
  const CvArr* image2,
  CvArr*       acc,
  const CvArr* mask = NULL
);
```

For covariance, there is a formula analogous to the one we just gave for variance. This formula is also a single-pass formula in that it has been manipulated algebraically from the standard form so as not to require two trips through the list of images:

$$\mathrm{Cov}(x,y) = \left(\frac{1}{N}\sum_{i=0}^{N-1}(x_i y_i)\right) - \left(\frac{1}{N}\sum_{i=0}^{N-1}x_i\right)\left(\frac{1}{N}\sum_{j=0}^{N-1}y_j\right)$$

In our context, x is the image at time t and y is the image at time $t - d$, where d is the lag.

We can use the accumulation functions described here to create a variety of statistics-based background models. The literature is full of variations on the basic model used as our example. You will probably find that, in your own applications, you will tend to extend this simplest model into slightly more specialized versions. A common enhancement, for example, is for the thresholds to be adaptive to some observed global state changes.

Advanced Background Method

Many background scenes contain complicated moving objects such as trees waving in the wind, fans turning, curtains fluttering, et cetera. Often such scenes also contain varying lighting, such as clouds passing by or doors and windows letting in different light.

A nice method to deal with this would be to fit a time-series model to each pixel or group of pixels. This kind of model deals with the temporal fluctuations well, but its disadvantage is the need for a great deal of memory [Toyama99]. If we use 2 seconds of previous input at 30 Hz, this means we need 60 samples for each pixel. The resulting model for each pixel would then encode what it had learned in the form of 60 different adapted *weights*. Often we'd need to gather background statistics for much longer than 2 seconds, which means that such methods are typically impractical on present-day hardware.

To get fairly close to the performance of adaptive filtering, we take inspiration from the techniques of video compression and attempt to form a *codebook** to represent significant states in the background.† The simplest way to do this would be to compare a new value observed for a pixel with prior observed values. If the value is close to a prior value, then it is modeled as a perturbation on that color. If it is not close, then it can seed a new group of colors to be associated with that pixel. The result could be envisioned as a bunch of blobs floating in RGB space, each blob representing a separate volume considered likely to be background.

In practice, the choice of RGB is not particularly optimal. It is almost always better to use a color space whose axis is aligned with brightness, such as the YUV color space. (YUV is the most common choice, but spaces such as HSV, where V is essentially brightness, would work as well.) The reason for this is that, empirically, most of the variation in background tends to be along the brightness axis, not the color axis.

The next detail is how to model the "blobs." We have essentially the same choices as before with our simpler model. We could, for example, choose to model the blobs as Gaussian clusters with a mean and a covariance. It turns out that the simplest case, in

* The method OpenCV implements is derived from Kim, Chalidabhongse, Harwood, and Davis [Kim05], but rather than learning-oriented tubes in RGB space, for speed, the authors use axis-aligned boxes in YUV space. Fast methods for cleaning up the resulting background image can be found in Martins [Martins99].

† There is a large literature for background modeling and segmentation. OpenCV's implementation is intended to be fast and robust enough that you can use it to collect foreground objects mainly for the purposes of collecting data sets to train classifiers on. Recent work in background subtraction allows arbitrary camera motion [Farin04; Colombari07] and dynamic background models using the mean-shift algorithm [Liu07].

which the "blobs" are simply boxes with a learned extent in each of the three axes of our color space, works out quite well. It is the simplest in terms of memory required and in terms of the computational cost of determining whether a newly observed pixel is inside any of the learned boxes.

Let's explain what a codebook is by using a simple example (Figure 9-3). A codebook is made up of boxes that grow to cover the common values seen over time. The upper panel of Figure 9-3 shows a waveform over time. In the lower panel, boxes form to cover a new value and then slowly grow to cover nearby values. If a value is too far away, then a new box forms to cover it and likewise grows slowly toward new values.

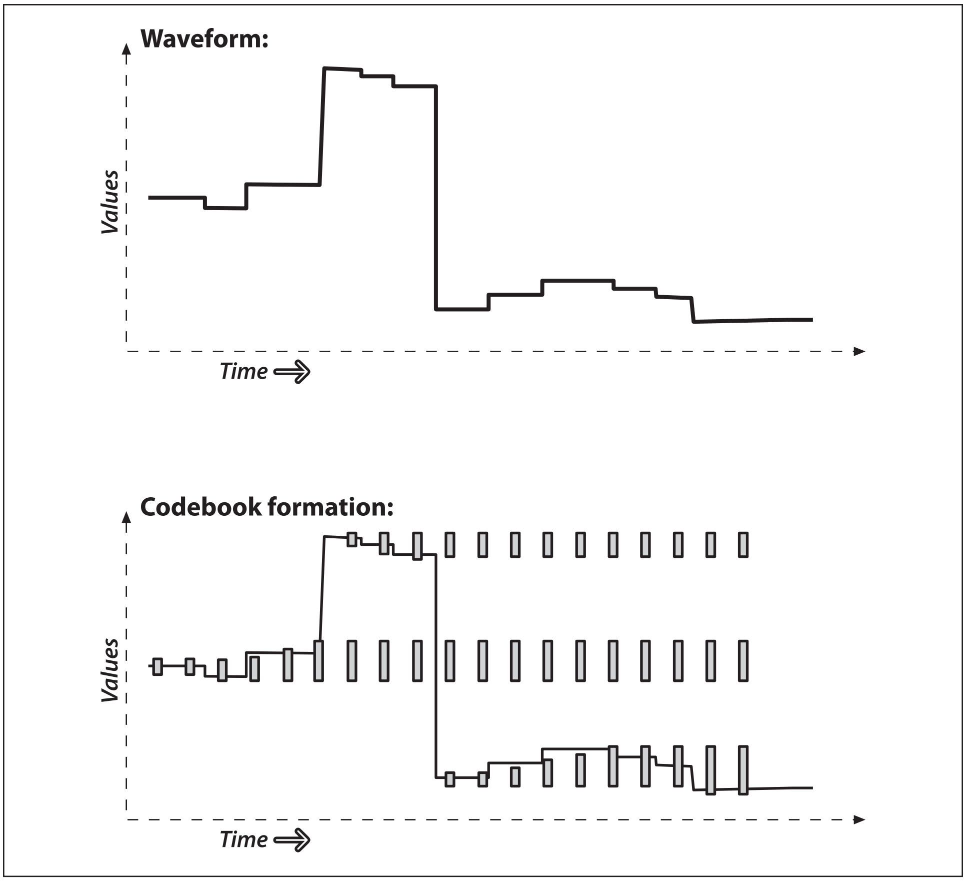

Figure 9-3. Codebooks are just "boxes" delimiting intensity values: a box is formed to cover a new value and slowly grows to cover nearby values; if values are too far away then a new box is formed (see text)

In the case of our background model, we will learn a codebook of boxes that cover three dimensions: the three channels that make up our image at each pixel. Figure 9-4 visualizes the (intensity dimension of the) codebooks for six different pixels learned from the data in Figure 9-1.* This codebook method can deal with pixels that change levels dramatically (e.g., pixels in a windblown tree, which might alternately be one of many colors of leaves, or the blue sky beyond that tree). With this more precise method of modeling, we can detect a foreground object that has values between the pixel values. Compare this with Figure 9-2, where the averaging method cannot distinguish the hand value (shown as a dotted line) from the pixel fluctuations. Peeking ahead to the next section, we see the better performance of the codebook method versus the averaging method shown later in Figure 9-7.

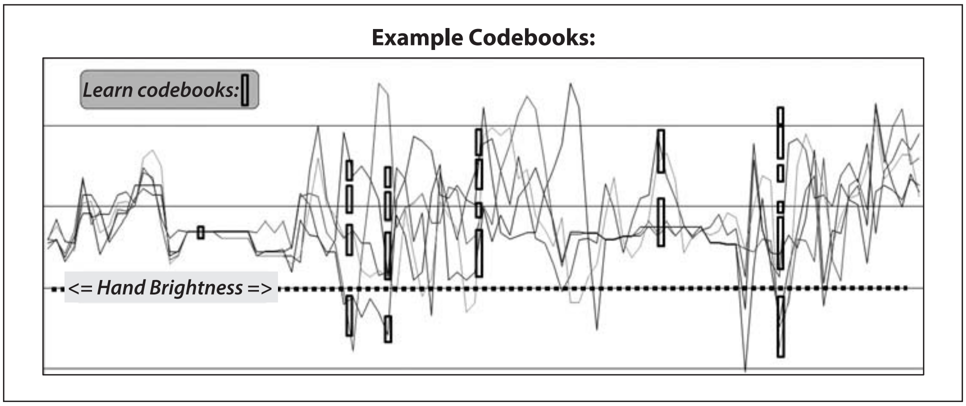

Figure 9-4. Intensity portion of learned codebook entries for fluctuations of six chosen pixels (shown as vertical boxes): codebook boxes accommodate pixels that take on multiple discrete values and so can better model discontinuous distributions; thus they can detect a foreground hand (value at dotted line) whose average value is between the values that background pixels can assume. In this case the codebooks are one dimensional and only represent variations in intensity

In the codebook method of learning a background model, each box is defined by two thresholds (`max` and `min`) over each of the three color axes. These box boundary thresholds will expand (`max` getting larger, `min` getting smaller) if new background samples fall within a learning threshold (`learnHigh` and `learnLow`) above `max` or below `min`, respectively. If new background samples fall outside of the box and its learning thresholds, then a new box will be started. In the *background difference* mode there are acceptance thresholds `maxMod` and `minMod`; using these threshold values, we say that if a pixel is "close enough" to a `max` or a `min` box boundary then we count it as if it were inside the box. A second runtime threshold allows for adjusting the model to specific conditions.

A situation we will not cover is a pan-tilt camera surveying a large scene. When working with a large scene, it is necessary to stitch together learned models indexed by the pan and tilt angles.

* In this case we have chosen several pixels at random from the scan line to avoid excessive clutter. Of course, there is actually a codebook for every pixel.

Structures

It's time to look at all of this in more detail, so let's create an implementation of the codebook algorithm. First, we need our codebook structure, which will simply point to a bunch of boxes in YUV space:

```
typedef struct code_book {
   code_element **cb;
   int numEntries;
   int t;         //count every access
} codeBook;
```

We track how many codebook entries we have in numEntries. The variable t counts the number of points we've accumulated since the start or the last clear operation. Here's how the actual codebook elements are described:

```
#define CHANNELS 3
typedef struct ce {
   uchar learnHigh[CHANNELS]; //High side threshold for learning
   uchar learnLow[CHANNELS];  //Low side threshold for learning
   uchar max[CHANNELS];       //High side of box boundary
   uchar min[CHANNELS];       //Low side of box boundary
   int t_last_update;         //Allow us to kill stale entries
   int stale;                 //max negative run (longest period of inactivity)
} code_element;
```

Each codebook entry consumes four bytes per channel plus two integers, or CHANNELS × 4 + 4 + 4 bytes (20 bytes when we use three channels). We may set CHANNELS to any positive number equal to or less than the number of color channels in an image, but it is usually set to either 1 ("Y", or brightness only) or 3 (YUV, HSV). In this structure, for each channel, max and min are the boundaries of the codebook box. The parameters learnHigh[] and learnLow[] are the thresholds that trigger generation of a new code element. Specifically, a new code element will be generated if a new pixel is encountered whose values do not lie between min - learnLow and max + learnHigh in each of the channels. The time to last update (t_last_update) and stale are used to enable the deletion of seldom-used codebook entries created during learning. Now we can proceed to investigate the functions that use this structure to learn dynamic backgrounds.

Learning the background

We will have one codeBook of code_elements for each pixel. We will need an array of such codebooks that is equal in length to the number of pixels in the images we'll be learning. For each pixel, update_codebook() is called for as many images as are sufficient to capture the relevant changes in the background. Learning may be updated periodically throughout, and clear_stale_entries() can be used to learn the background in the presence of (small numbers of) moving foreground objects. This is possible because the seldom-used "stale" entries induced by a moving foreground will be deleted. The interface to update_codebook() is as follows.

```
////////////////////////////////////////////////////////////
// int update_codebook(uchar *p, codeBook &c, unsigned cbBounds)
// Updates the codebook entry with a new data point
```

```
//
// p            Pointer to a YUV pixel
// c            Codebook for this pixel
// cbBounds     Learning bounds for codebook (Rule of thumb: 10)
// numChannels  Number of color channels we're learning
//
// NOTES:
//      cvBounds must be of length equal to numChannels
//
// RETURN
//   codebook index
//
int update_codebook(
  uchar*    p,
  codeBook& c,
  unsigned* cbBounds,
  int       numChannels
){
   unsigned int high[3],low[3];
   for(n=0; n<numChannels; n++)
   {
      high[n] = *(p+n)+*(cbBounds+n);
      if(high[n] > 255) high[n] = 255;
      low[n] = *(p+n)-*(cbBounds+n);
      if(low[n] < 0) low[n] = 0;
   }
   int matchChannel;

   // SEE IF THIS FITS AN EXISTING CODEWORD
   //
   for(int i=0; i<c.numEntries; i++){
      matchChannel = 0;
      for(n=0; n<numChannels; n++){
         if((c.cb[i]->learnLow[n] <= *(p+n)) &&
         //Found an entry for this channel
         (*(p+n) <= c.cb[i]->learnHigh[n]))
            {
               matchChannel++;
            }
      }
      if(matchChannel == numChannels) //If an entry was found
      {
         c.cb[i]->t_last_update = c.t;
         //adjust this codeword for the first channel
         for(n=0; n<numChannels; n++){
            if(c.cb[i]->max[n] < *(p+n))
            {
               c.cb[i]->max[n] = *(p+n);
            }
            else if(c.cb[i]->min[n] > *(p+n))
            {
               c.cb[i]->min[n] = *(p+n);
            }
         }
         break;
```

```
            }
        }
    . . . continued below
```

This function grows or adds a codebook entry when the pixel `p` falls outside the existing codebook boxes. Boxes grow when the pixel is within `cbBounds` of an existing box. If a pixel is outside the `cbBounds` distance from a box, a new codebook box is created. The routine first sets `high` and `low` levels to be used later. It then goes through each codebook entry to check whether the pixel value `*p` is inside the learning bounds of the codebook "box". If the pixel is within the learning bounds for all channels, then the appropriate `max` or `min` level is adjusted to include this pixel and the time of last update is set to the current timed count `c.t`. Next, the `update_codebook()` routine keeps statistics on how often each codebook entry is hit:

```
    . . . continued from above

        // OVERHEAD TO TRACK POTENTIAL STALE ENTRIES
        //
        for(int s=0; s<c.numEntries; s++){

           // Track which codebook entries are going stale:
           //
           int negRun = c.t - c.cb[s]->t_last_update;
           if(c.cb[s]->stale < negRun) c.cb[s]->stale = negRun;

        }

    . . . continued below
```

Here, the variable `stale` contains the largest *negative runtime* (i.e., the longest span of time during which that code was not accessed by the data). Tracking stale entries allows us to delete codebooks that were formed from noise or moving foreground objects and hence tend to become stale over time. In the next stage of learning the background, `update_codebook()` adds a new codebook if needed:

```
    . . . continued from above

        // ENTER A NEW CODEWORD IF NEEDED
        //
        if(i == c.numEntries) //if no existing codeword found, make one
        {
           code_element **foo = new code_element* [c.numEntries+1];
           for(int ii=0; ii<c.numEntries; ii++) {
             foo[ii] = c.cb[ii];
           }
           foo[c.numEntries] = new code_element;
           if(c.numEntries) delete [] c.cb;
           c.cb = foo;
           for(n=0; n<numChannels; n++) {
              c.cb[c.numEntries]->learnHigh[n] = high[n];
              c.cb[c.numEntries]->learnLow[n] = low[n];
              c.cb[c.numEntries]->max[n] = *(p+n);
              c.cb[c.numEntries]->min[n] = *(p+n);
           }
```

```
            c.cb[c.numEntries]->t_last_update = c.t;
            c.cb[c.numEntries]->stale = 0;
            c.numEntries += 1;
         }
```

. . . continued below

Finally, update_codebook() slowly adjusts (by adding 1) the learnHigh and learnLow learning boundaries if pixels were found outside of the box thresholds but still within the high and low bounds:

. . . continued from above

```
         // SLOWLY ADJUST LEARNING BOUNDS
         //
         for(n=0; n<numChannels; n++)
         {
            if(c.cb[i]->learnHigh[n] < high[n]) c.cb[i]->learnHigh[n] += 1;
            if(c.cb[i]->learnLow[n] > low[n]) c.cb[i]->learnLow[n] -= 1;
         }
         return(i);
      }
```

The routine concludes by returning the index of the modified codebook. We've now seen how codebooks are learned. In order to learn in the presence of moving foreground objects and to avoid learning codes for spurious noise, we need a way to delete entries that were accessed only rarely during learning.

Learning with moving foreground objects

The following routine, clear_stale_entries(), allows us to learn the background even if there are moving foreground objects.

```
///////////////////////////////////////////////////////////////////
//int clear_stale_entries(codeBook &c)
// During learning, after you've learned for some period of time,
// periodically call this to clear out stale codebook entries
//
// c   Codebook to clean up
//
// Return
// number of entries cleared
//
int clear_stale_entries(codeBook &c){
  int staleThresh = c.t>>1;
  int *keep = new int [c.numEntries];
  int keepCnt = 0;
  // SEE WHICH CODEBOOK ENTRIES ARE TOO STALE
  //
  for(int i=0; i<c.numEntries; i++){
     if(c.cb[i]->stale > staleThresh)
        keep[i] = 0; //Mark for destruction
     else
     {
        keep[i] = 1; //Mark to keep
        keepCnt += 1;
```

```
      }
    }
    // KEEP ONLY THE GOOD
    //
    c.t = 0;          //Full reset on stale tracking
    code_element **foo = new code_element* [keepCnt];
    int k=0;
    for(int ii=0; ii<c.numEntries; ii++){
      if(keep[ii])
      {
        foo[k] = c.cb[ii];
        //We have to refresh these entries for next clearStale
        foo[k]->t_last_update = 0;
        k++;
      }
    }
    // CLEAN UP
    //
    delete [] keep;
    delete [] c.cb;
    c.cb = foo;
    int numCleared = c.numEntries - keepCnt;
    c.numEntries = keepCnt;
    return(numCleared);
  }
```

The routine begins by defining the parameter staleThresh, which is hardcoded (by a rule of thumb) to be half the total running time count, c.t. This means that, during background learning, if codebook entry i is not accessed for a period of time equal to half the total learning time, then i is marked for deletion (keep[i] = 0). The vector keep[] is allocated so that we can mark each codebook entry; hence it is c.numEntries long. The variable keepCnt counts how many entries we will keep. After recording which codebook entries to keep, we create a new pointer, foo, to a vector of code_element pointers that is keepCnt long, and then the nonstale entries are copied into it. Finally, we delete the old pointer to the codebook vector and replace it with the new, nonstale vector.

Background differencing: Finding foreground objects

We've seen how to create a background codebook model and how to clear it of seldom-used entries. Next we turn to background_diff(), where we use the learned model to segment foreground pixels from the previously learned background:

```
////////////////////////////////////////////////////////////
// uchar background_diff( uchar *p, codeBook &c,
//                         int minMod, int maxMod)
// Given a pixel and a codebook, determine if the pixel is
// covered by the codebook
//
// p            Pixel pointer (YUV interleaved)
// c            Codebook reference
// numChannels  Number of channels we are testing
// maxMod       Add this (possibly negative) number onto
//              max level when determining if new pixel is foreground
// minMod       Subract this (possibly negative) number from
//              min level when determining if new pixel is foreground
//
// NOTES:
// minMod and maxMod must have length numChannels,
// e.g. 3 channels => minMod[3], maxMod[3]. There is one min and
//      one max threshold per channel.
//
// Return
// 0 => background, 255 => foreground
//
uchar background_diff(
  uchar*    p,
  codeBook& c,
  int       numChannels,
  int*      minMod,
  int*      maxMod
) {
  int matchChannel;

  // SEE IF THIS FITS AN EXISTING CODEWORD
  //
  for(int i=0; i<c.numEntries; i++) {
     matchChannel = 0;
     for(int n=0; n<numChannels; n++) {
        if((c.cb[i]->min[n] - minMod[n] <= *(p+n)) &&
           (*(p+n) <= c.cb[i]->max[n] + maxMod[n])) {
           matchChannel++; //Found an entry for this channel
        } else {
           break;
        }
     }
     if(matchChannel == numChannels) {
        break; //Found an entry that matched all channels
     }
  }
  if(i >= c.numEntries) return(255);
  return(0);
}
```

The background differencing function has an inner loop similar to the learning routine update_codebook, except here we look within the learned max and min bounds plus an offset threshold, maxMod and minMod, of each codebook box. If the pixel is within the box plus maxMod on the high side or minus minMod on the low side for each channel, then the matchChannel count is incremented. When matchChannel equals the number of channels, we've searched each dimension and know that we have a match. If the pixel is within a learned box, 255 is returned (a positive detection of foreground); otherwise, 0 is returned (background).

The three functions update_codebook(), clear_stale_entries(), and background_diff() constitute a codebook method of segmenting foreground from learned background.

Using the codebook background model

To use the codebook background segmentation technique, typically we take the following steps.

1. Learn a basic model of the background over a few seconds or minutes using `update_codebook()`.
2. Clean out stale entries with `clear_stale_entries()`.
3. Adjust the thresholds `minMod` and `maxMod` to best segment the known foreground.
4. Maintain a higher-level scene model (as discussed previously).
5. Use the learned model to segment the foreground from the background via `background_diff()`.
6. Periodically update the learned background pixels.
7. At a much slower frequency, periodically clean out stale codebook entries with `clear_stale_entries()`.

A few more thoughts on codebook models

In general, the codebook method works quite well across a wide number of conditions, and it is relatively quick to train and to run. It doesn't deal well with varying patterns of light—such as morning, noon, and evening sunshine—or with someone turning lights on or off indoors. This type of global variability can be taken into account by using several different codebook models, one for each condition, and then allowing the condition to control which model is active.

Connected Components for Foreground Cleanup

Before comparing the averaging method to the codebook method, we should pause to discuss ways to clean up the raw segmented image using connected-components analysis. This form of analysis takes in a noisy input mask image; it then uses the morphological operation *open* to shrink areas of small noise to 0 followed by the morphological operation *close* to rebuild the area of surviving components that was lost in opening. Thereafter, we can find the "large enough" contours of the surviving segments and can optionally proceed to take statistics of all such segments. We can then retrieve either the largest contour or all contours of size above some threshold. In the routine that follows, we implement most of the functions that you could want in connected components:

- Whether to approximate the surviving component contours by polygons or by convex hulls
- Setting how large a component contour must be in order not to be deleted
- Setting the maximum number of component contours to return
- Optionally returning the bounding boxes of the surviving component contours
- Optionally returning the centers of the surviving component contours

The connected components header that implements these operations is as follows.

```
////////////////////////////////////////////////////////////////////
// void find_connected_components(IplImage *mask, int poly1_hull0,
//                            float perimScale, int *num,
//                            CvRect *bbs, CvPoint *centers)
// This cleans up the foreground segmentation mask derived from calls
// to backgroundDiff
//
// mask          Is a grayscale (8-bit depth) “raw” mask image that
//               will be cleaned up
//
// OPTIONAL PARAMETERS:
// poly1_hull0   If set, approximate connected component by
//                 (DEFAULT) polygon, or else convex hull (0)
// perimScale    Len = image (width+height)/perimScale. If contour
//                 len < this, delete that contour (DEFAULT: 4)
// num           Maximum number of rectangles and/or centers to
//                 return; on return, will contain number filled
//                 (DEFAULT: NULL)
// bbs           Pointer to bounding box rectangle vector of
//                 length num. (DEFAULT SETTING: NULL)
// centers      Pointer to contour centers vector of length
//                 num (DEFAULT: NULL)
//
void find_connected_components(
  IplImage* mask,
  int       poly1_hull0 = 1,
  float     perimScale  = 4,
  int*      num         = NULL,
  CvRect*   bbs         = NULL,
  CvPoint*  centers     = NULL
 );
```

The function body is listed below. First we declare memory storage for the connected components contour. We then do morphological opening and closing in order to clear out small pixel noise, after which we rebuild the eroded areas that survive the erosion of the opening operation. The routine takes two additional parameters, which here are hardcoded via #define. The defined values work well, and you are unlikely to want to change them. These additional parameters control how simple the boundary of a foreground region should be (higher numbers are more simple) and how many iterations the morphological operators should perform; the higher the number of iterations, the more erosion takes place in opening before dilation in closing.* More erosion eliminates larger regions of blotchy noise at the cost of eroding the boundaries of larger regions. Again, the parameters used in this sample code work well, but there's no harm in experimenting with them if you like.

```
// For connected components:
// Approx.threshold - the bigger it is, the simpler is the boundary
//
```

* Observe that the value CVCLOSE_ITR is actually dependent on the resolution. For images of extremely high resolution, leaving this value set to 1 is not likely to yield satisfactory results.

```
#define CVCONTOUR_APPROX_LEVEL  2

// How many iterations of erosion and/or dilation there should be
//
#define CVCLOSE_ITR  1
```

We now discuss the connected-component algorithm itself. The first part of the routine performs the morphological open and closing operations:

```
void find_connected_components(
  IplImage *mask,
  int poly1_hull0,
  float perimScale,
  int *num,
  CvRect *bbs,
  CvPoint *centers
) {

  static CvMemStorage*   mem_storage = NULL;
  static CvSeq*          contours    = NULL;

  //CLEAN UP RAW MASK
  //
  cvMorphologyEx( mask, mask, 0, 0, CV_MOP_OPEN,  CVCLOSE_ITR );
  cvMorphologyEx( mask, mask, 0, 0, CV_MOP_CLOSE, CVCLOSE_ITR );
```

Now that the noise has been removed from the mask, we find all contours:

```
  //FIND CONTOURS AROUND ONLY BIGGER REGIONS
  //
  if( mem_storage==NULL ) {
    mem_storage = cvCreateMemStorage(0);
  } else {
    cvClearMemStorage(mem_storage);
  }

  CvContourScanner scanner = cvStartFindContours(
    mask,
    mem_storage,
    sizeof(CvContour),
    CV_RETR_EXTERNAL,
    CV_CHAIN_APPROX_SIMPLE
  );
```

Next, we toss out contours that are too small and approximate the rest with polygons or convex hulls (whose complexity has already been set by CVCONTOUR_APPROX_LEVEL):

```
  CvSeq* c;
  int numCont = 0;
  while( (c = cvFindNextContour( scanner )) != NULL ) {

    double len = cvContourPerimeter( c );

    // calculate perimeter len threshold:
    //
    double q = (mask->height + mask->width)/perimScale;

    //Get rid of blob if its perimeter is too small:
    //
    if( len < q ) {
       cvSubstituteContour( scanner, NULL );
    } else {

      // Smooth its edges if its large enough
      //
      CvSeq* c_new;
      if( poly1_hull0 ) {

        // Polygonal approximation
        //
        c_new = cvApproxPoly(
          c,
          sizeof(CvContour),
          mem_storage,
          CV_POLY_APPROX_DP,
          CVCONTOUR_APPROX_LEVEL,
          0
        );

      } else {

         // Convex Hull of the segmentation
         //
         c_new = cvConvexHull2(
           c,
           mem_storage,
           CV_CLOCKWISE,
           1
         );
      }
      cvSubstituteContour( scanner, c_new );
      numCont++;
    }
  }
  contours = cvEndFindContours( &scanner );
```

In the preceding code, CV_POLY_APPROX_DP causes the Douglas-Peucker approximation algorithm to be used, and CV_CLOCKWISE is the default direction of the convex hull contour. All this processing yields a list of contours. Before drawing the contours back into the mask, we define some simple colors to draw:

```
// Just some convenience variables
const CvScalar CVX_WHITE = CV_RGB(0xff,0xff,0xff)
const CvScalar CVX_BLACK = CV_RGB(0x00,0x00,0x00)
```

We use these definitions in the following code, where we first zero out the mask and then draw the clean contours back into the mask. We also check whether the user wanted to collect statistics on the contours (bounding boxes and centers):

```
// PAINT THE FOUND REGIONS BACK INTO THE IMAGE
//
cvZero( mask );
IplImage *maskTemp;
```

```
// CALC CENTER OF MASS AND/OR BOUNDING RECTANGLES
//
if(num != NULL) {

  //User wants to collect statistics
  //
  int N = *num, numFilled = 0, i=0;
  CvMoments moments;
  double M00, M01, M10;
  maskTemp = cvCloneImage(mask);
  for(i=0, c=contours; c != NULL; c = c->h_next,i++ ) {

     if(i < N) {
       // Only process up to *num of them
       //
       cvDrawContours(
         maskTemp,
         c,
         CVX_WHITE,
         CVX_WHITE,
         -1,
         CV_FILLED,
         8
       );

       // Find the center of each contour
       //
       if(centers != NULL) {

           cvMoments(maskTemp,&moments,1);
           M00 = cvGetSpatialMoment(&moments,0,0);
           M10 = cvGetSpatialMoment(&moments,1,0);
           M01 = cvGetSpatialMoment(&moments,0,1);
           centers[i].x = (int)(M10/M00);
           centers[i].y = (int)(M01/M00);
       }

       //Bounding rectangles around blobs
       //
       if(bbs != NULL) {
          bbs[i] = cvBoundingRect(c);
       }
       cvZero(maskTemp);
       numFilled++;
     }
     // Draw filled contours into mask
     //
     cvDrawContours(
       mask,
       c,
       CVX_WHITE,
       CVX_WHITE,
       -1,
       CV_FILLED,
       8
     );
  }                              //end looping over contours
  *num = numFilled;
  cvReleaseImage( &maskTemp);
}
```

If the user doesn't need the bounding boxes and centers of the resulting regions in the mask, we just draw back into the mask those cleaned-up contours representing large enough connected components of the background.

```
     // ELSE JUST DRAW PROCESSED CONTOURS INTO THE MASK
     //
     else {
       // The user doesn't want statistics, just draw the contours
       //
       for( c=contours; c != NULL; c = c->h_next ) {
         cvDrawContours(
         mask,
         c,
         CVX_WHITE,
         CVX_BLACK,
         -1,
         CV_FILLED,
         8
       );
     }
   }
```

That concludes a useful routine for creating clean masks out of noisy raw masks. Now let's look at a short comparison of the background subtraction methods.

A quick test

We start with an example to see how this really works in an actual video. Let's stick with our video of the tree outside of the window. Recall (Figure 9-1) that at some point a hand passes through the scene. One might expect that we could find this hand relatively easily with a technique such as frame differencing (discussed previously in its own section). The basic idea of frame differencing was to subtract the current frame from a "lagged" frame and then threshold the difference.

Sequential frames in a video tend to be quite similar. Hence one might expect that, if we take a simple difference of the original frame and the lagged frame, we'll not see too much unless there is some foreground object moving through the scene.* But what does "not see too much" mean in this context? Really, it means "just noise." Of course, in practice the problem is sorting out that noise from the signal when a foreground object does come along.

* In the context of frame differencing, an object is identified as "foreground" mainly by its velocity. This is reasonable in scenes that are generally static or in which foreground objects are expected to be much closer to the camera than background objects (and thus appear to move faster by virtue of the projective geometry of cameras).

To understand this noise a little better, we will first look at a pair of frames from the video in which there is no foreground object—just the background and the resulting noise. Figure 9-5 shows a typical frame from the video (upper left) and the previous frame (upper right). The figure also shows the results of frame differencing with a threshold value of 15 (lower left). You can see substantial noise from the moving leaves of the tree. Nevertheless, the method of connected components is able to clean up this scattered noise quite well* (lower right). This is not surprising, because there is no reason to expect much spatial correlation in this noise and so its signal is characterized by a large number of very small regions.

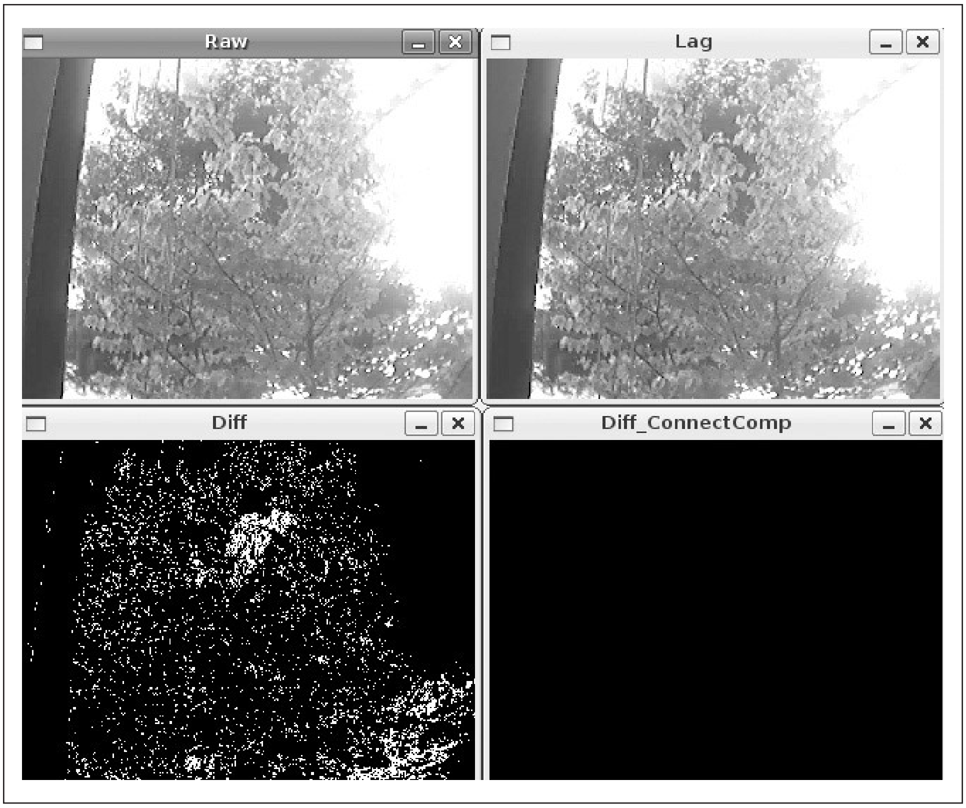

Figure 9-5. Frame differencing: a tree is waving in the background in the current (upper left) and previous (upper right) frame images; the difference image (lower left) is completely cleaned up (lower right) by the connected-components method

Now consider the situation in which a foreground object (our ubiquitous hand) passes through the view of the imager. Figure 9-6 shows two frames that are similar to those in Figure 9-5 except that now the hand is moving across from left to right. As before, the current frame (upper left) and the previous frame (upper right) are shown along

* The size threshold for the connected components has been tuned to give zero response in these empty frames. The real question then is whether or not the foreground object of interest (the hand) survives pruning at this size threshold. We will see (Figure 9-6) that it does so nicely.

with the response to frame differencing (lower left) and the fairly good results of the connected-component cleanup (lower right).

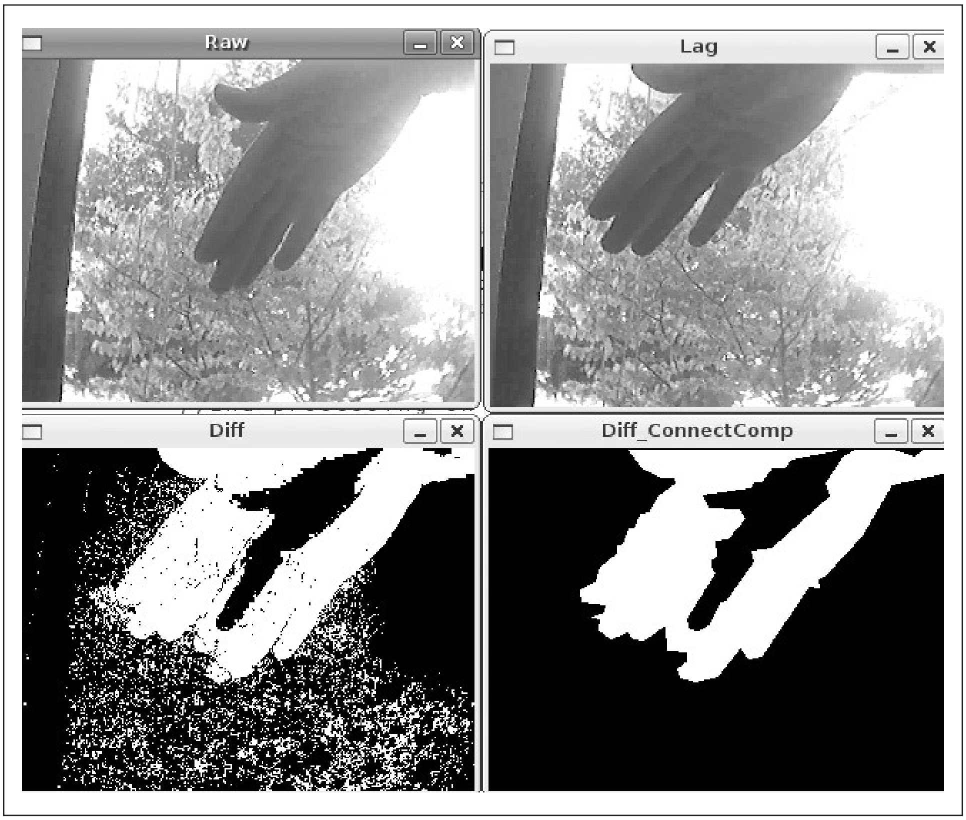

Figure 9-6. Frame difference method of detecting a hand, which is moving left to right as the foreground object (upper two panels); the difference image (lower left) shows the "hole" (where the hand used to be) toward the left and its leading edge toward the right, and the connected-component image (lower right) shows the cleaned-up difference

We can also clearly see one of the deficiencies of frame differencing: it cannot distinguish between the region from where the object moved (the "hole") and where the object is now. Furthermore, in the overlap region there is often a gap because "flesh minus flesh" is 0 (or at least below threshold).

Thus we see that using connected components for cleanup is a powerful technique for rejecting noise in background subtraction. As a bonus, we were also able to glimpse some of the strengths and weaknesses of frame differencing.

Comparing Background Methods

We have discussed two background modeling techniques in this chapter: the average distance method and the codebook method. You might be wondering which method is better, or, at least, when you can get away with using the easy one. In these situations, it's always best to just do a straight bake off* between the available methods.

We will continue with the same tree video that we've been discussing all chapter. In addition to the moving tree, this film has a lot of glare coming off a building to the right and off portions of the inside wall on the left. It is a fairly challenging background to model.

In Figure 9-7 we compare the average difference method at top against the codebook method at bottom; on the left are the raw foreground images and on the right are the cleaned-up connected components. You can see that the average difference method leaves behind a sloppier mask and breaks the hand into two components. This is not so surprising; in Figure 9-2, we saw that using the average difference from the mean as a background model often included pixel values associated with the hand value (shown as a dotted line in that figure). Compare this with Figure 9-4, where codebooks can more accurately model the fluctuations of the leaves and branches and so more precisely identify foreground hand pixels (dotted line) from background pixels. Figure 9-7 confirms not only that the background model yields less noise but also that connected components can generate a fairly accurate object outline.

Watershed Algorithm

In many practical contexts, we would like to segment an image but do not have the benefit of a separate background image. One technique that is often effective in this context is the *watershed algorithm* [Meyer92]. This algorithm converts lines in an image into "mountains" and uniform regions into "valleys" that can be used to help segment objects. The watershed algorithm first takes the gradient of the intensity image; this has the effect of forming valleys or *basins* (the low points) where there is no texture and of forming mountains or *ranges* (high ridges corresponding to edges) where there are dominant lines in the image. It then successively floods basins starting from user-specified (or algorithm-specified) points until these regions meet. Regions that merge across the marks so generated are segmented as belonging together as the image "fills up". In this way, the basins connected to the marker point become "owned" by that marker. We then segment the image into the corresponding marked regions.

More specifically, the watershed algorithm allows a user (or another algorithm!) to mark parts of an object or background that are known to be part of the object or background. The user or algorithm can draw a simple line that effectively tells the watershed algorithm to "group points like these together". The watershed algorithm then segments the image by allowing marked regions to "own" the edge-defined valleys in the gradient image that are connected with the segments. Figure 9-8 clarifies this process.

The function specification of the watershed segmentation algorithm is:

```
void cvWatershed(
  const CvArr* image,
```

* For the uninitiated, "bake off" is actually a bona fide term used to describe any challenge or comparison of multiple algorithms on a predetermined data set.

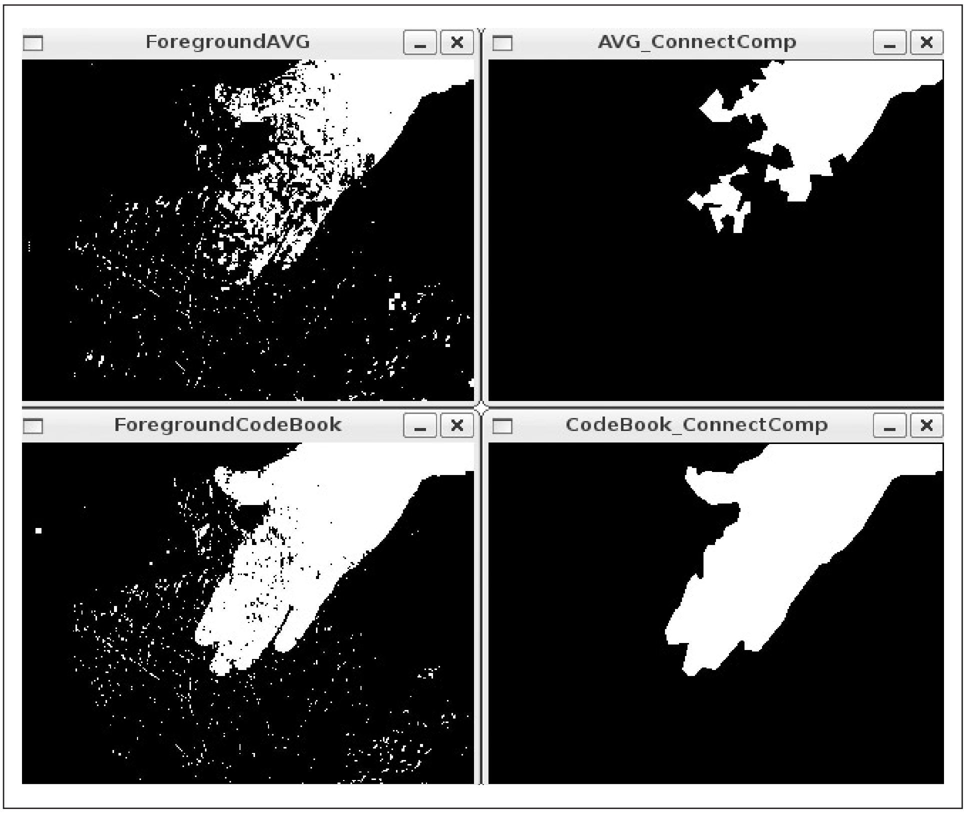

Figure 9-7. With the averaging method (top row), the connected-components cleanup knocks out the fingers (upper right); the codebook method (bottom row) does much better at segmentation and creates a clean connected-component mask (lower right)

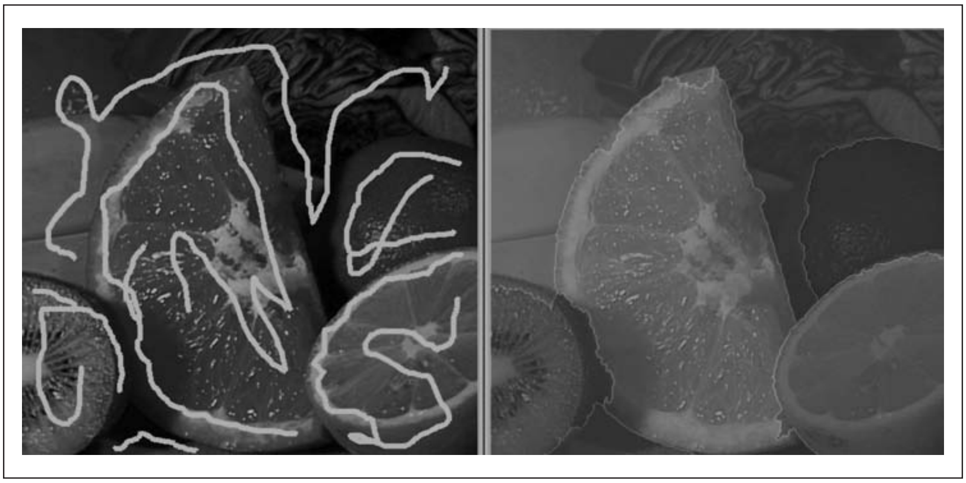

Figure 9-8. Watershed algorithm: after a user has marked objects that belong together (left panel), the algorithm then merges the marked area into segments (right panel)

```
    CvArr*       markers
);
```

Here, `image` is an 8-bit color (three-channel) image and `markers` is a single-channel integer (`IPL_DEPTH_32S`) image of the same (*x*, *y*) dimensions; the value of `markers` is 0 *except* where the user (or an algorithm) has indicated by using positive numbers that some regions belong together. For example, in the left panel of Figure 9-8, the orange might have been marked with a "1", the lemon with a "2", the lime with "3", the upper background with "4" and so on. This produces the segmentation you see in the same figure on the right.

Image Repair by Inpainting

Images are often corrupted by noise. There may be dust or water spots on the lens, scratches on the older images, or parts of an image that were vandalized. *Inpainting* [Telea04] is a method for removing such damage by taking the color and texture at the border of the damaged area and propagating and mixing it inside the damaged area. See Figure 9-9 for an application that involves the removal of writing from an image.

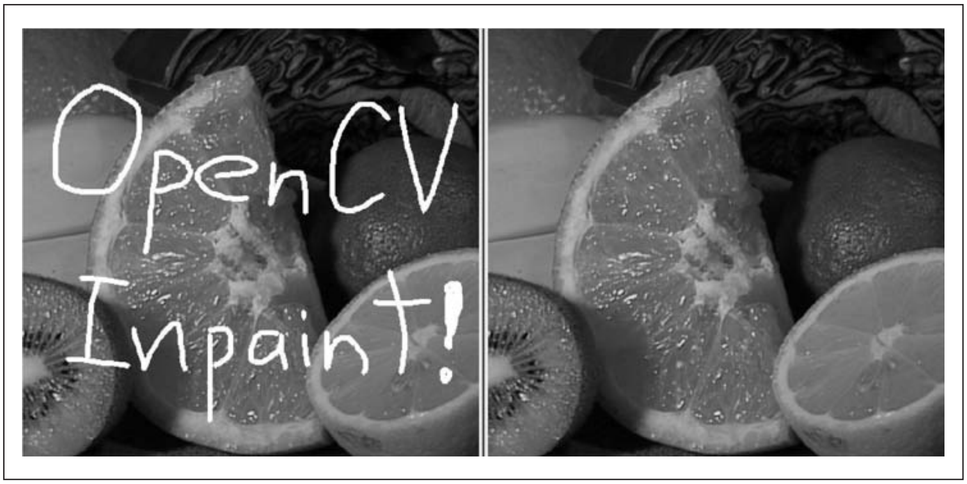

Figure 9-9. Inpainting: an image damaged by overwritten text (left panel) is restored by inpainting (right panel)

Inpainting works provided the damaged area is not too "thick" and enough of the original texture and color remains around the boundaries of the damage. Figure 9-10 shows what happens when the damaged area is too large.

The prototype for `cvInpaint()` is

```
void cvInpaint(
    const CvArr* src,
    const CvArr* mask,
    CvArr*       dst,
    double       inpaintRadius,
    int          flags
);
```

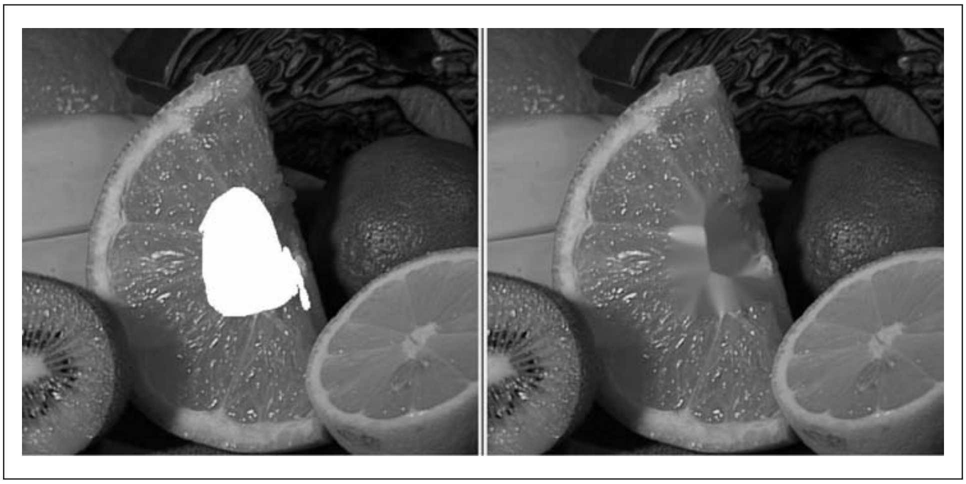

Figure 9-10. Inpainting cannot magically restore textures that are completely removed: the navel of the orange has been completely blotted out (left panel); inpainting fills it back in with mostly orange-like texture (right panel)

Here `src` is an 8-bit single-channel grayscale image or a three-channel color image to be repaired, and `mask` is an 8-bit single-channel image of the same size as `src` in which the damaged areas (e.g., the writing seen in the left panel of Figure 9-9) have been marked by nonzero pixels; all other pixels are set to 0 in `mask`. The output image will be written to `dst`, which must be the same size and number of channels as `src`. The `inpaintRadius` is the area around each inpainted pixel that will be factored into the resulting output color of that pixel. As in Figure 9-10, interior pixels within a thick enough inpainted region may take their color entirely from other inpainted pixels closer to the boundaries. Almost always, one uses a small radius such as `3` because too large a radius will result in a noticeable blur. Finally, the `flags` parameter allows you to experiment with two different methods of inpainting: `CV_INPAINT_NS` (Navier-Stokes method), and `CV_INPAINT_TELEA` (A. Telea's method).

Mean-Shift Segmentation

In Chapter 5 we introduced the function `cvPyrSegmentation()`. Pyramid segmentation uses a color merge (over a scale that depends on the similarity of the colors to one another) in order to segment images. This approach is based on minimizing the *total energy* in the image; here energy is defined by a *link strength*, which is further defined by *color similarity*. In this section we introduce `cvPyrMeanShiftFiltering()`, a similar algorithm that is based on mean-shift clustering over color [Comaniciu99]. We'll see the details of the mean-shift algorithm `cvMeanShift()` in Chapter 10, when we discuss tracking and motion. For now, what we need to know is that mean shift finds the peak of a color-spatial (or other feature) distribution over time. Here, mean-shift segmentation finds the peaks of color distributions over space. The common theme is that both the motion tracking and the color segmentation algorithms rely on the ability of mean shift to find the modes (peaks) of a distribution.

Given a set of multidimensional data points whose dimensions are (*x, y,* blue, green, red), mean shift can find the highest density "clumps" of data in this space by scanning a *window* over the space. Notice, however, that the spatial variables (*x, y*) can have very different ranges from the color magnitude ranges (blue, green, red). Therefore, mean shift needs to allow for different window radii in different dimensions. In this case we should have one radius for the spatial variables (`spatialRadius`) and one radius for the color magnitudes (`colorRadius`). As mean-shift windows move, all the points traversed by the windows that converge at a peak in the data become connected or "owned" by that peak. This ownership, radiating out from the densest peaks, forms the segmentation of the image. The segmentation is actually done over a scale pyramid (`cvPyrUp()`, `cvPyrDown()`), as described in Chapter 5, so that color clusters at a high level in the pyramid (shrunken image) have their boundaries refined at lower pyramid levels in the pyramid. The function call for `cvPyrMeanShiftFiltering()` looks like this:

```
void cvPyrMeanShiftFiltering(
  const CvArr*   src,
  CvArr*         dst,
  double         spatialRadius,
  double         colorRadius,
  int            max_level    = 1,
  CvTermCriteria termcrit     = cvTermCriteria(
      CV_TERMCRIT_ITER | CV_TERMCRIT_EPS,
      5,
      1
    )
  );
```

In `cvPyrMeanShiftFiltering()` we have an input image `src` and an output image `dst`. Both must be 8-bit, three-channel color images of the same width and height. The `spatialRadius` and `colorRadius` define how the mean-shift algorithm averages color and space together to form a segmentation. For a 640-by-480 color image, it works well to set `spatialRadius` equal to 2 and `colorRadius` equal to 40. The next parameter of this algorithm is `max_level`, which describes how many levels of scale pyramid you want used for segmentation. A `max_level` of 2 or 3 works well for a 640-by-480 color image.

The final parameter is `CvTermCriteria`, which we saw in Chapter 8. `CvTermCriteria` is used for all iterative algorithms in OpenCV. The mean-shift segmentation function comes with good defaults if you just want to leave this parameter blank. Otherwise, `cvTermCriteria` has the following constructor:

```
cvTermCriteria(
    int    type; // CV_TERMCRIT_ITER, CV_TERMCRIT_EPS,
    int    max_iter,
    double epsilon
);
```

Typically we use the `cvTermCriteria()` function to generate the `CvTermCriteria` structure that we need. The first argument is either `CV_TERMCRIT_ITER` or `CV_TERMCRIT_EPS`, which tells the algorithm that we want to terminate either after some fixed number of iterations or when the convergence metric reaches some small value (respectively). The next two arguments set the values at which one, the other, or both of these criteria should terminate the algorithm. The reason we have both options is because we can set the type to `CV_TERMCRIT_ITER | CV_TERMCRIT_EPS` to stop when either limit is reached. The parameter `max_iter` limits the number of iterations if `CV_TERMCRIT_ITER` is set, whereas `epsilon` sets the error limit if `CV_TERMCRIT_EPS` is set. Of course the exact meaning of `epsilon` depends on the algorithm.

Figure 9-11 shows an example of mean-shift segmentation using the following values:

```
    cvPyrMeanShiftFiltering( src, dst, 20, 40, 2);
```

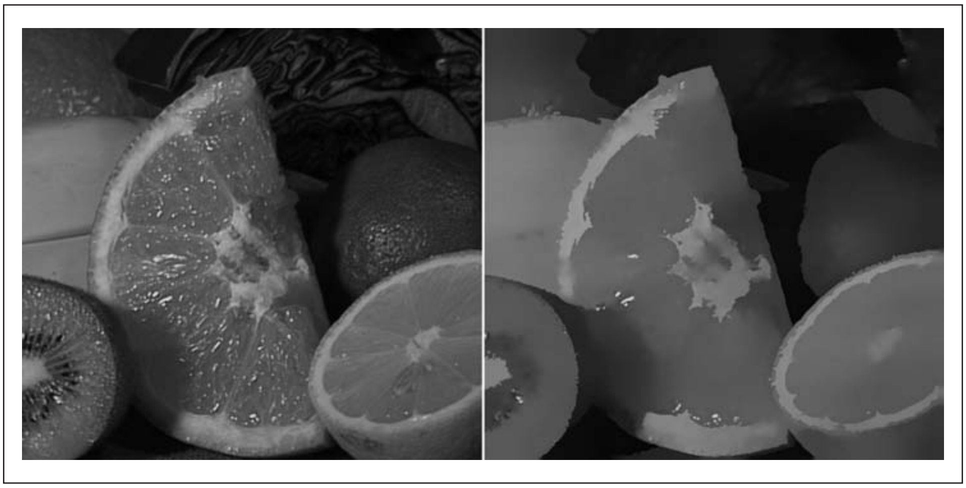
Figure 9-11. Mean-shift segmentation over scale using cvPyrMeanShiftFiltering() with parameters max_level=2, spatialRadius=20, and colorRadius=40; similar areas now have similar values and so can be treated as super pixels, which can speed up subsequent processing significantly

Delaunay Triangulation, Voronoi Tesselation

Delaunay triangulation is a technique invented in 1934 [Delaunay34] for connecting points in a space into triangular groups such that the minimum angle of all the angles in the triangulation is a maximum. This means that Delaunay triangulation tries to avoid long skinny triangles when triangulating points. See Figure 9-12 to get the gist of triangulation, which is done in such a way that any circle that is fit to the points at the vertices of any given triangle contains no other vertices. This is called the *circum-circle property* (panel c in the figure).

For computational efficiency, the Delaunay algorithm invents a far-away outer bounding triangle from which the algorithm starts. Figure 9-12(b) represents the fictitious outer triangle by faint lines going out to its vertex. Figure 9-12(c) shows some examples of the circum-circle property, including one of the circles linking two outer points of the real data to one of the vertices of the fictitious external triangle.

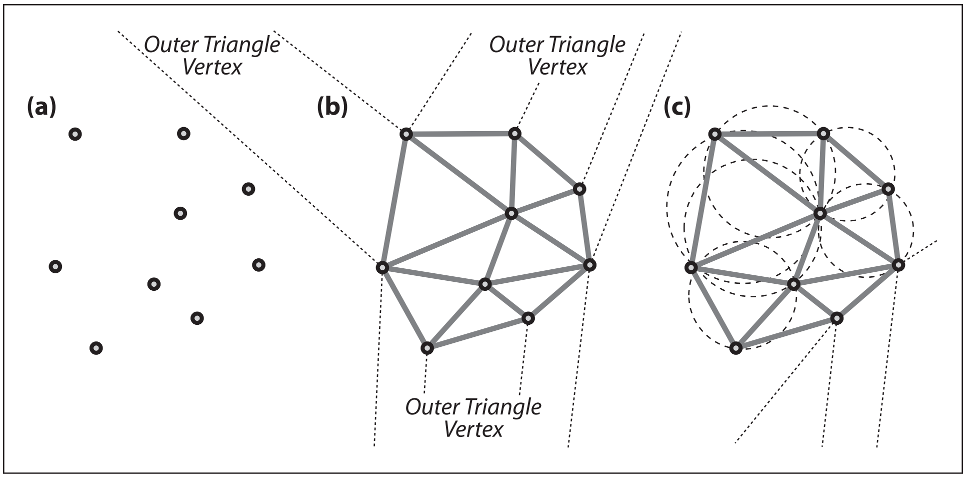

Figure 9-12. Delaunay triangulation: (a) set of points; (b) Delaunay triangulation of the point set with trailers to the outer bounding triangle; (c) example circles showing the circum-circle property

There are now many algorithms to compute Delaunay triangulation; some are very efficient but with difficult internal details. The gist of one of the more simple algorithms is as follows:

1. Add the external triangle and start at one of its vertices (this yields a definitive outer starting point).
2. Add an internal point; then search over all the triangles' circum-circles containing that point and remove those triangulations.
3. Re-triangulate the graph, including the new point in the circum-circles of the just removed triangulations.
4. Return to step 2 until there are no more points to add.

The order of complexity of this algorithm is $O(n^2)$ in the number of data points. The best algorithms are (on average) as low as $O(n \log \log n)$.

Great—but what is it good for? For one thing, remember that this algorithm started with a fictitious outer triangle and so all the real outside points are actually connected to two of that triangle's vertices. Now recall the circum-circle property: circles that are fit through any two of the real outside points and to an external fictitious vertex contain no other inside points. This means that a computer may directly look up exactly which real points form the outside of a set of points by looking at which points are connected to the three outer fictitious vertices. In other words, we can find the convex hull of a set of points almost instantly after a Delaunay triangulation has been done.

We can also find who "owns" the space between points, that is, which coordinates are nearest neighbors to each of the Delaunay vertex points. Thus, using Delaunay triangulation of the original points, you can immediately find the nearest neighbor to a new point. Such a partition is called a *Voronoi tessellation* (see Figure 9-13). This tessellation is the dual image of the Delaunay triangulation, because the Delaunay lines define the distance between existing points and so the Voronoi lines "know" where they must intersect the Delaunay lines in order to keep equal distance between points. These two methods, calculating the convex hull and nearest neighbor, are important basic operations for clustering and classifying points and point sets.

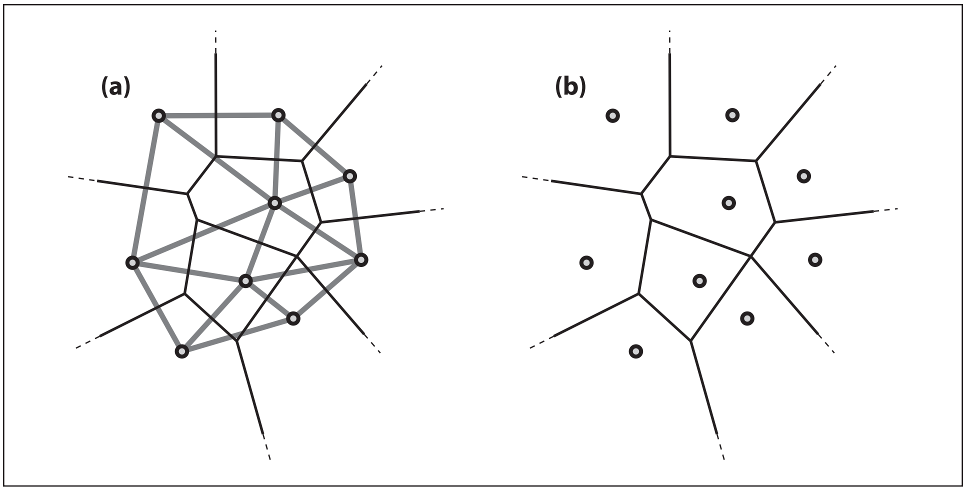

Figure 9-13. Voronoi tessellation, whereby all points within a given Voronoi cell are closer to their Delaunay point than to any other Delaunay point: (a) the Delaunay triangulation in bold with the corresponding Voronoi tessellation in fine lines; (b) the Voronoi cells around each Delaunay point

If you're familiar with 3D computer graphics, you may recognize that Delaunay triangulation is often the basis for representing 3D shapes. If we render an object in three dimensions, we can create a 2D view of that object by its image projection and then use the 2D Delaunay triangulation to analyze and identify this object and/or compare it with a real object. Delaunay triangulation is thus a bridge between computer vision and computer graphics. However, one deficiency of OpenCV (soon to be rectified, we hope; see Chapter 14) is that OpenCV performs Delaunay triangulation only in two dimensions. If we could triangulate point clouds in three dimensions—say, from stereo vision (see Chapter 11)—then we could move seamlessly between 3D computer graphics and computer vision. Nevertheless, 2D Delaunay triangulation is often used in computer vision to register the spatial arrangement of features on an object or a scene for motion tracking, object recognition, or matching views between two different cameras (as in deriving depth from stereo images). Figure 9-14 shows a tracking and recognition application of Delaunay triangulation [Gokturk01; Gokturk02] wherein key facial feature points are spatially arranged according to their triangulation.

Now that we've established the potential usefulness of Delaunay triangulation once given a set of points, how do we derive the triangulation? OpenCV ships with example code for this in the *.../opencv/samples/c/delaunay.c* file. OpenCV refers to Delaunay triangulation as a Delaunay *subdivision*, whose critical and reusable pieces we discuss next.

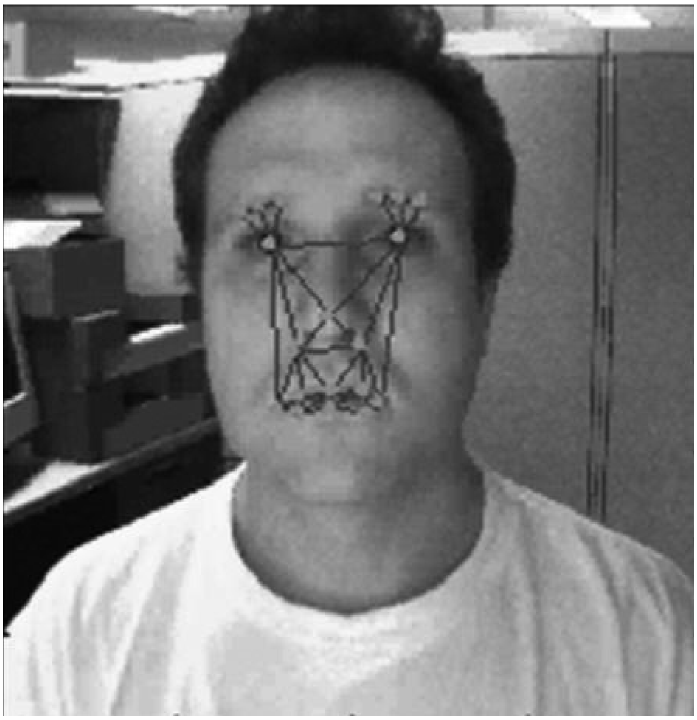

Figure 9-14. Delaunay points can be used in tracking objects; here, a face is tracked using points that are significant in expressions so that emotions may be detected

Creating a Delaunay or Voronoi Subdivision

First we'll need some place to store the Delaunay subdivision in memory. We'll also need an outer bounding box (remember, to speed computations, the algorithm works with a fictitious outer triangle positioned outside a rectangular bounding box). To set this up, suppose the points must be inside a 600-by-600 image:

```
// STORAGE AND STRUCTURE FOR DELAUNAY SUBDIVISION
//
CvRect        rect = { 0, 0, 600, 600 };  //Our outer bounding box
CvMemStorage* storage;                    //Storage for the Delaunay subdivsion
storage = cvCreateMemStorage(0);          //Initialize the storage
CvSubdiv2D*   subdiv;                     //The subdivision itself
subdiv = init_delaunay( storage, rect);   //See this function below
```

The code calls init_delaunay(), which is not an OpenCV function but rather a convenient packaging of a few OpenCV routines:

```
//INITIALIZATION CONVENIENCE FUNCTION FOR DELAUNAY SUBDIVISION
//
CvSubdiv2D* init_delaunay(
  CvMemStorage* storage,
  CvRect rect
) {
  CvSubdiv2D* subdiv;
  subdiv = cvCreateSubdiv2D(
    CV_SEQ_KIND_SUBDIV2D,
    sizeof(*subdiv),
    sizeof(CvSubdiv2DPoint),
    sizeof(CvQuadEdge2D),
    storage
  );
  cvInitSubdivDelaunay2D( subdiv, rect ); //rect sets the bounds
  return subdiv;
}
```

Next we'll need to know how to insert points. These points must be of type float, 32f:

```
CvPoint2D32f fp;     //This is our point holder

for( i = 0; i < as_many_points_as_you_want; i++ ) {

    // However you want to set points
    //
    fp = your_32f_point_list[i];

    cvSubdivDelaunay2DInsert( subdiv, fp );
}
```

You can convert integer points to 32f points using the convenience macro `cvPoint2D32f(double x, double y)` or `cvPointTo32f(CvPoint point)` located in *cxtypes.h*. Now that we can enter points to obtain a Delaunay triangulation, we set and clear the associated Voronoi tessellation with the following two commands:

```
cvCalcSubdivVoronoi2D( subdiv );  // Fill out Voronoi data in subdiv
cvClearSubdivVoronoi2D( subdiv ); // Clear the Voronoi from subdiv
```

In both functions, `subdiv` is of type `CvSubdiv2D*`. We can now create Delaunay subdivisions of two-dimensional point sets and then add and clear Voronoi tessellations to them. But how do we get at the good stuff inside these structures? We can do this by stepping from edge to point or from edge to edge in `subdiv`; see Figure 9-15 for the basic maneuvers starting from a given edge and its point of origin. We next find the first edges or points in the subdivision in one of two different ways: (1) by using an external point to locate an edge or a vertex; or (2) by stepping through a sequence of points or edges. We'll first describe how to step around edges and points in the graph and then how to step through the graph.

Navigating Delaunay Subdivisions

Figure 9-15 combines two data structures that we'll use to move around on a subdivision graph. The structure `cvQuadEdge2D` contains a set of two Delaunay and two Voronoi points and their associated edges (assuming the Voronoi points and edges have been calculated with a prior call to `cvCalcSubdivVoronoi2D()`); see Figure 9-16. The structure `CvSubdiv2DPoint` contains the Delaunay edge with its associated vertex point, as shown in Figure 9-17. The quad-edge structure is defined in the code following the figure.

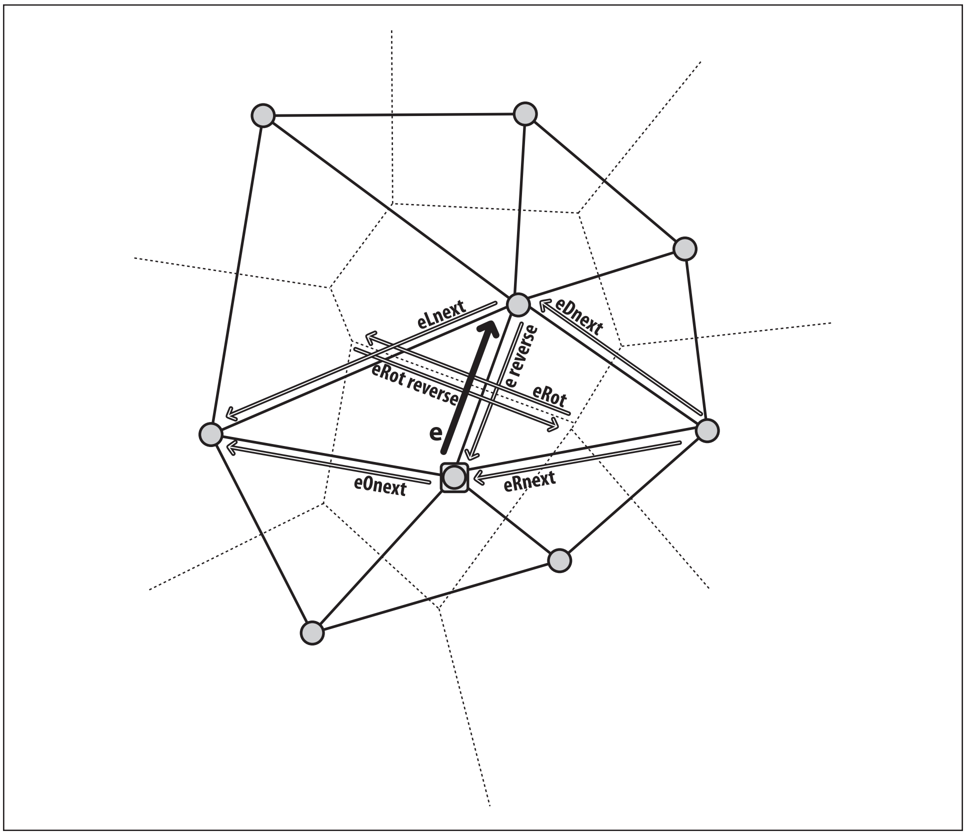

*Figure 9-15. Edges relative to a given edge, labeled “**e**”, and its vertex point (marked by a square)*

```
// Edges themselves are encoded in long integers. The lower two bits
// are its index (0..3) and upper bits are the quad-edge pointer.
//
typedef long CvSubdiv2DEdge;

// quad-edge structure fields:
//
#define CV_QUADEDGE2D_FIELDS()        /
    int flags;                        /
    struct CvSubdiv2DPoint* pt[4];    /
    CvSubdiv2DEdge  next[4];

typedef struct CvQuadEdge2D {
    CV_QUADEDGE2D_FIELDS()
} CvQuadEdge2D;
```

The Delaunay subdivision point and the associated edge structure is given by:

```
#define CV_SUBDIV2D_POINT_FIELDS() /
    int            flags;        /
    CvSubdiv2DEdge first;        //*The edge “e” in the figures.*/
    CvPoint2D32f   pt;
```

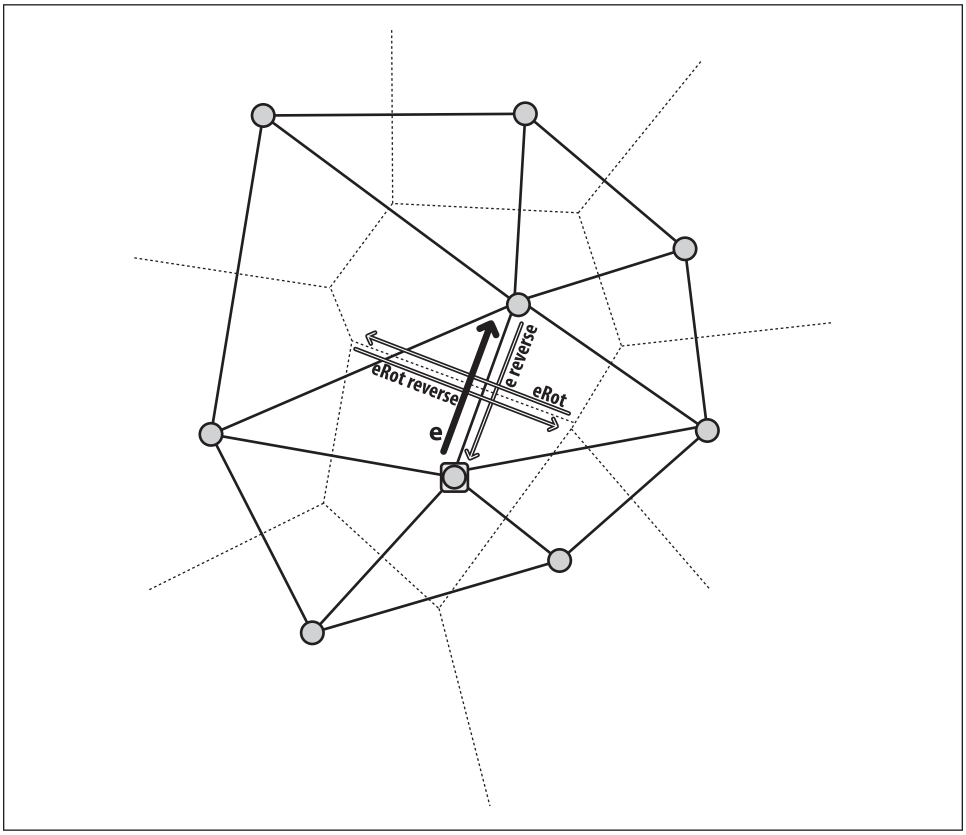

Figure 9-16. Quad edges that may be accessed by cvSubdiv2DRotateEdge() include the Delaunay edge and its reverse (along with their associated vertex points) as well as the related Voronoi edges and points

```
#define CV_SUBDIV2D_VIRTUAL_POINT_FLAG (1 << 30)

typedef struct CvSubdiv2DPoint
{
    CV_SUBDIV2D_POINT_FIELDS()
}
CvSubdiv2DPoint;
```

With these structures in mind, we can now examine the different ways of moving around.

Walking on edges

As indicated by Figure 9-16, we can step around quad edges by using

```
CvSubdiv2DEdge cvSubdiv2DRotateEdge(
  CvSubdiv2DEdge edge,
  int            type
);
```

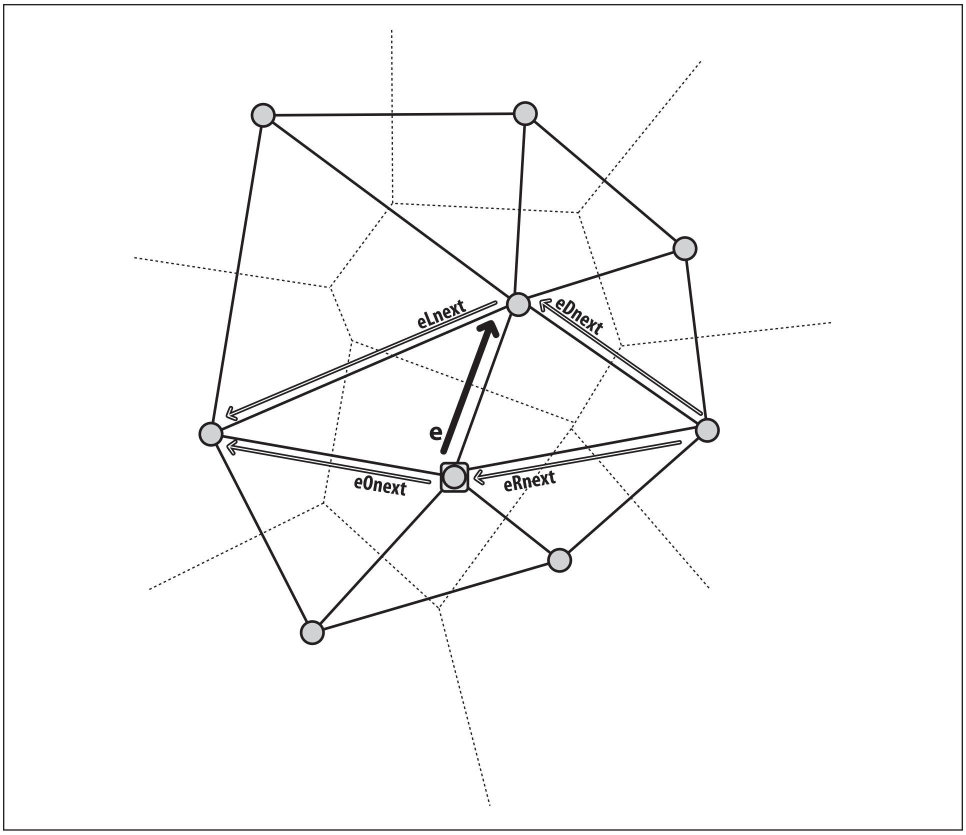

Figure 9-17. A CvSubdiv2DPoint vertex and its associated edge e along with other associated edges that may be accessed via cvSubdiv2DGetEdge()

Given an `edge`, we can get to the next edge by using the `type` parameter, which takes one of the following arguments:

- `0`, the input edge (`e` in the figure if `e` is the input edge)
- `1`, the rotated edge (`eRot`)
- `2`, the reversed edge (reversed `e`)
- `3`, the reversed rotated edge (reversed `eRot`)

Referencing Figure 9-17, we can also get around the Delaunay graph using

```
CvSubdiv2DEdge cvSubdiv2DGetEdge(
  CvSubdiv2DEdge edge,
  CvNextEdgeType type
);
#define cvSubdiv2DNextEdge( edge )      /
  cvSubdiv2DGetEdge(                    /
    edge,                               /
    CV_NEXT_AROUND_ORG                  /
  )
```

Here `type` specifies one of the following moves:

`CV_NEXT_AROUND_ORG`
: Next around the edge origin (`eOnext` in Figure 9-17 if `e` is the input edge)

`CV_NEXT_AROUND_DST`
: Next around the edge vertex (`eDnext`)

`CV_PREV_AROUND_ORG`
: Previous around the edge origin (reversed `eRnext`)

`CV_PREV_AROUND_DST`
: Previous around the edge destination (reversed `eLnext`)

`CV_NEXT_AROUND_LEFT`
: Next around the left facet (`eLnext`)

`CV_NEXT_AROUND_RIGHT`
: Next around the right facet (`eRnext`)

`CV_PREV_AROUND_LEFT`
: Previous around the left facet (reversed `eOnext`)

`CV_PREV_AROUND_RIGHT`
: Previous around the right facet (reversed `eDnext`)

Note that, given an edge associated with a vertex, we can use the convenience macro `cvSubdiv2DNextEdge( edge )` to find all other edges from that vertex. This is helpful for finding things like the convex hull starting from the vertices of the (fictitious) outer bounding triangle.

The other important movement types are `CV_NEXT_AROUND_LEFT` and `CV_NEXT_AROUND_RIGHT`. We can use these to step around a Delaunay triangle if we're on a Delaunay edge or to step around a Voronoi cell if we're on a Voronoi edge.

Points from edges

We'll also need to know how to retrieve the actual points from Delaunay or Voronoi vertices. Each Delaunay or Voronoi edge has two points associated with it: `org`, its origin point, and `dst`, its destination point. You may easily obtain these points by using

```
CvSubdiv2DPoint* cvSubdiv2DEdgeOrg( CvSubdiv2DEdge edge );
CvSubdiv2DPoint* cvSubdiv2DEdgeDst( CvSubdiv2DEdge edge );
```

Here are methods to convert `CvSubdiv2DPoint` to more familiar forms:

```
CvSubdiv2DPoint ptSub;                          //Subdivision vertex point
CvPoint2D32f    pt32f = ptSub->pt;              // to 32f point
CvPoint         pt    = cvPointFrom32f(pt32f);  // to an integer point
```

We now know what the subdivision structures look like and how to walk around its points and edges. Let's return to the two methods for getting the first edges or points from the Delaunay/Voronoi subdivision.

Method 1: Use an external point to locate an edge or vertex

The first method is to start with an arbitrary point and then locate that point in the subdivision. This need not be a point that has already been triangulated; it can be any point. The function cvSubdiv2DLocate() fills in one edge and vertex (if desired) of the triangle or Voronoi facet into which that point fell.

```
CvSubdiv2DPointLocation cvSubdiv2DLocate(
  CvSubdiv2D*        subdiv,
  CvPoint2D32f       pt,
  CvSubdiv2DEdge*    edge,
  CvSubdiv2DPoint** vertex = NULL
);
```

Note that these are not necessarily the closest edge or vertex; they just have to be in the triangle or facet. This function's return value tells us where the point landed, as follows.

CV_PTLOC_INSIDE

The point falls into some facet; *edge will contain one of edges of the facet.

CV_PTLOC_ON_EDGE

The point falls onto the edge; *edge will contain this edge.

CV_PTLOC_VERTEX

The point coincides with one of subdivision vertices; *vertex will contain a pointer to the vertex.

CV_PTLOC_OUTSIDE_RECT

The point is outside the subdivision reference rectangle; the function returns and no pointers are filled.

CV_PTLOC_ERROR

One of input arguments is invalid.

Method 2: Step through a sequence of points or edges

Conveniently for us, when we create a Delaunay subdivision of a set of points, the first three points and edges form the vertices and sides of the fictitious outer bounding triangle. From there, we may directly access the outer points and edges that form the convex hull of the actual data points. Once we have formed a Delaunay subdivision (call it subdiv), we'll also need to call cvCalcSubdivVoronoi2D(subdiv) in order to calculate the associated Voronoi tessellation. We can then access the three vertices of the outer bounding triangle using

```
CvSubdiv2DPoint* outer_vtx[3];
for( i = 0; i < 3; i++ ) {
  outer_vtx[i] =
    (CvSubdiv2DPoint*)cvGetSeqElem( (CvSeq*)subdiv, I );
}
```

We can similarly obtain the three sides of the outer bounding triangle:

```
CvQuadEdge2D* outer_qedges[3];
for( i = 0; i < 3; i++ ) {
  outer_qedges[i] =
    (CvQuadEdge2D*)cvGetSeqElem( (CvSeq*)(my_subdiv->edges), I );
}
```

Now that we know how to get on the graph and move around, we'll want to know when we're on the outer edge or boundary of the points.

Identifying the bounding triangle or edges on the convex hull and walking the hull

Recall that we used a bounding rectangle `rect` to initialize the Delaunay triangulation with the call `cvInitSubdivDelaunay2D( subdiv, rect )`. In this case, the following statements hold.

1. If you are on an edge where both the origin and destination points are out of the `rect` bounds, then that edge is on the fictitious bounding triangle of the subdivision.
2. If you are on an edge with one point inside and one point outside the `rect` bounds, then the point in bounds is on the convex hull of the set; each point on the convex hull is connected to two vertices of the fictitious outer bounding triangle, and these two edges occur one after another.

From the second condition, you can use the `cvSubdiv2DNextEdge()` macro to step onto the first edge whose `dst` point is within bounds. That first edge with both ends in bounds is on the convex hull of the point set, so remember that point or edge. Once on the convex hull, you can then move around the convex hull as follows.

1. Until you have circumnavigated the convex hull, go to the next edge on the hull via `cvSubdiv2DRotateEdge(CvSubdiv2DEdge edge, 0)`.
2. From there, another two calls to the `cvSubdiv2DNextEdge()` macro will get you on the next edge of the convex hull. Return to step 1.

We now know how to initialize Delaunay and Voronoi subdivisions, how to find the initial edges, and also how to step through the edges and points of the graph. In the next section we present some practical applications.

Usage Examples

We can use `cvSubdiv2DLocate()` to step around the edges of a Delaunay triangle:

```
void locate_point(
  CvSubdiv2D*   subdiv,
  CvPoint2D32f fp,
  IplImage*    img,
  CvScalar     active_color
) {
  CvSubdiv2DEdge e;
  CvSubdiv2DEdge e0 = 0;
  CvSubdiv2DPoint* p = 0;
  cvSubdiv2DLocate( subdiv, fp, &e0, &p );
```

```
    if( e0 ) {
      e = e0;
      do // Always 3 edges -- this is a triangulation, after all.
      {
        // [Insert your code here]
        //
        // Do something with e ...
         e = cvSubdiv2DGetEdge(e,CV_NEXT_AROUND_LEFT);
      }
      while( e != e0 );
    }
  }
```

We can also find the closest point to an input point by using

```
CvSubdiv2DPoint* cvFindNearestPoint2D(
  CvSubdiv2D*  subdiv,
  CvPoint2D32f pt
);
```

Unlike cvSubdiv2DLocate(), cvFindNearestPoint2D() will return the nearest vertex point in the Delaunay subdivision. This point is not necessarily on the facet or triangle that the point lands on.

Similarly, we could step around a Voronoi facet (here we draw it) using

```
void draw_subdiv_facet(
  IplImage *img,
  CvSubdiv2DEdge edge
) {

  CvSubdiv2DEdge t = edge;
  int i, count = 0;
  CvPoint* buf = 0;

  // Count number of edges in facet
  do{
      count++;
      t = cvSubdiv2DGetEdge( t, CV_NEXT_AROUND_LEFT );
  } while (t != edge );

  // Gather points
  //
  buf = (CvPoint*)malloc( count * sizeof(buf[0]))
  t = edge;
  for( i = 0; i < count; i++ ) {
      CvSubdiv2DPoint* pt = cvSubdiv2DEdgeOrg( t );
      if( !pt ) break;
      buf[i] = cvPoint( cvRound(pt->pt.x), cvRound(pt->pt.y));
      t = cvSubdiv2DGetEdge( t, CV_NEXT_AROUND_LEFT );
  }

  // Around we go
  //
  if( i == count ){
      CvSubdiv2DPoint* pt = cvSubdiv2DEdgeDst(
                            cvSubdiv2DRotateEdge( edge, 1 ));
      cvFillConvexPoly( img, buf, count,
         CV_RGB(rand()&255,rand()&255,rand()&255), CV_AA, 0 );
        cvPolyLine( img, &buf, &count, 1, 1, CV_RGB(0,0,0),
                  1, CV_AA, 0);
      draw_subdiv_point( img, pt->pt, CV_RGB(0,0,0));
  }
  free( buf );
}
```

Finally, another way to access the subdivision structure is by using a `CvSeqReader` to step though a sequence of edges. Here's how to step through all Delaunay or Voronoi edges:

```
void visit_edges( CvSubdiv2D* subdiv){

  CvSeqReader  reader;                     //Sequence reader
  int i, total = subdiv->edges->total;     //edge count
  int elem_size = subdiv->edges->elem_size; //edge size

  cvStartReadSeq( (CvSeq*)(subdiv->edges), &reader, 0 );

  cvCalcSubdivVoronoi2D( subdiv ); //Make sure Voronoi exists

  for( i = 0; i < total; i++ ) {

    CvQuadEdge2D* edge = (CvQuadEdge2D*)(reader.ptr);

    if( CV_IS_SET_ELEM( edge )) {

      // Do something with Voronoi and Delaunay edges ...
      //
      CvSubdiv2DEdge voronoi_edge = (CvSubdiv2DEdge)edge + 1;
      CvSubdiv2DEdge delaunay_edge = (CvSubdiv2DEdge)edge;

      // …OR WE COULD FOCUS EXCLUSIVELY ON VORONOI…

      // left
      //
      voronoi_edge = cvSubdiv2DRotateEdge( edge, 1 );

      // right
      //
      voronoi_edge = cvSubdiv2DRotateEdge( edge, 3 );
    }
    CV_NEXT_SEQ_ELEM( elem_size, reader );
  }
}
```

Finally, we end with an inline convenience macro: once we find the vertices of a Delaunay triangle, we can find its area by using

```
double cvTriangleArea(
  CvPoint2D32f a,
  CvPoint2D32f b,
  CvPoint2D32f c
)
```

Exercises

1. Using `cvRunningAvg()`, re-implement the averaging method of background subtraction. In order to do so, learn the running average of the pixel values in the scene to find the mean and the running average of the absolute difference (`cvAbsDiff()`) as a proxy for the standard deviation of the image.

2. Shadows are often a problem in background subtraction because they can show up as a foreground object. Use the averaging or codebook method of background subtraction to learn the background. Have a person then walk in the foreground. Shadows will "emanate" from the bottom of the foreground object.

    a. Outdoors, shadows are darker and bluer than their surround; use this fact to eliminate them.

    b. Indoors, shadows are darker than their surround; use this fact to eliminate them.

3. The simple background models presented in this chapter are often quite sensitive to their threshold parameters. In Chapter 10 we'll see how to track motion, and this can be used as a "reality" check on the background model and its thresholds. You can also use it when a known person is doing a "calibration walk" in front of the camera: find the moving object and adjust the parameters until the foreground object corresponds to the motion boundaries. We can also use distinct patterns on a calibration object itself (or on the background) for a reality check and tuning guide when we know that a portion of the background has been occluded.

    a. Modify the code to include an autocalibration mode. Learn a background model and then put a brightly colored object in the scene. Use color to find the colored object and then use that object to automatically set the thresholds in the background routine so that it segments the object. Note that you can leave this object in the scene for continuous tuning.

    b. Use your revised code to address the shadow-removal problem of exercise 2.

4. Use background segmentation to segment a person with arms held out. Investigate the effects of the different parameters and defaults in the `find_connected_components()` routine. Show your results for different settings of:

    a. `poly1_hull0`

    b. `perimScale`

    c. `CVCONTOUR_APPROX_LEVEL`

    d. `CVCLOSE_ITR`

5. In the 2005 DARPA Grand Challenge robot race, the authors on the Stanford team used a kind of color clustering algorithm to separate road from nonroad. The colors were sampled from a laser-defined trapezoid of road patch in front of the car. Other colors in the scene that were close in color to this patch—and whose connected

component connected to the original trapezoid—were labeled as road. See Figure 9-18, where the watershed algorithm was used to segment the road after using a trapezoid mark inside the road and an inverted "U" mark outside the road. Suppose we could automatically generate these marks. What could go wrong with this method of segmenting the road?

Hint: Look carefully at Figure 9-8 and then consider that we are trying to extend the road trapezoid by using things that look like what's in the trapezoid.

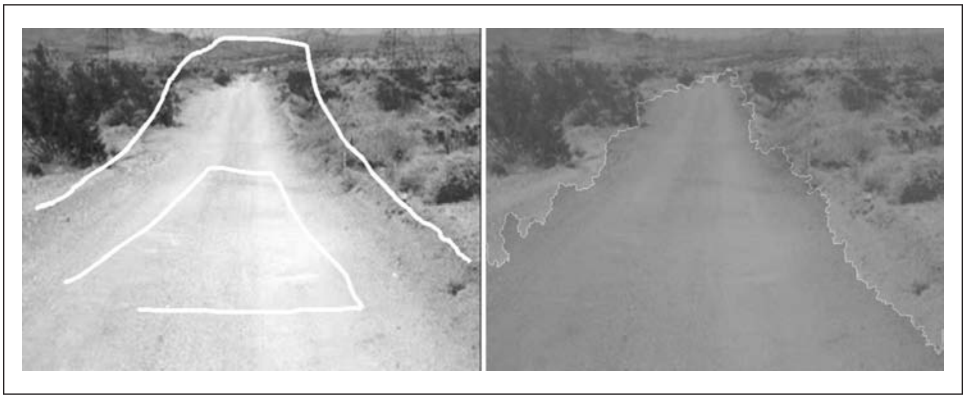
Figure 9-18. Using the watershed algorithm to identify a road: markers are put in the original image (left), and the algorithm yields the segmented road (right)

6. Inpainting works pretty well for the repair of writing over textured regions. What would happen if the writing obscured a real object edge in a picture? Try it.
7. Although it might be a little slow, try running background segmentation when the video input is first pre-segmented by using `cvPyrMeanShiftFiltering()`. That is, the input stream is first mean-shift segmented and then passed for background learning—and later testing for foreground—by the codebook background segmentation routine.
    a. Show the results compared to not running the mean-shift segmentation.
    b. Try systematically varying the `max_level`, `spatialRadius`, and `colorRadius` of the mean-shift segmentation. Compare those results.
8. How well does inpainting work at fixing up writing drawn over a mean-shift segmented image? Try it for various settings and show the results.
9. Modify the *.../opencv/samples/delaunay.c* code to allow mouse-click point entry (instead of via the existing method where points are selected at a random). Experiment with triangulations on the results.
10. Modify the *delaunay.c* code again so that you can use a keyboard to draw the convex hull of the point set.
11. Do three points in a line have a Delaunay triangulation?

12. Is the triangulation shown in Figure 9-19(a) a Delaunay triangulation? If so, explain your answer. If not, how would you alter the figure so that it is a Delaunay triangulation?
13. Perform a Delaunay triangulation by hand on the points in Figure 9-19(b). For this exercise, you need not add an outer fictitious bounding triangle.

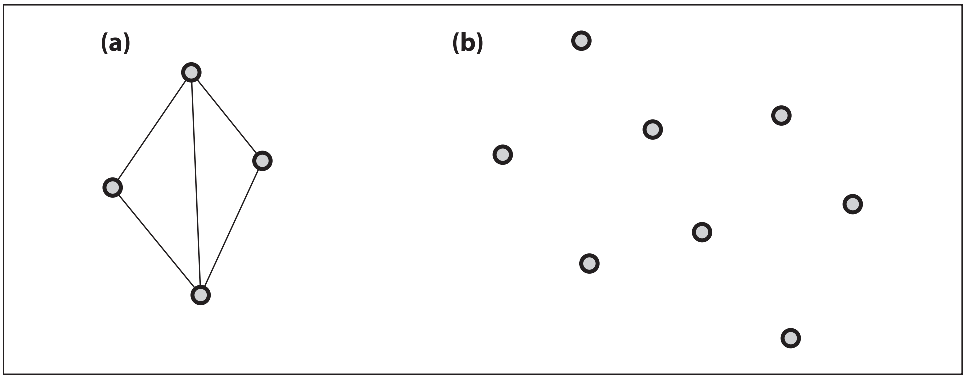

Figure 9-19. Exercise 12 and Exercise 13