Tracking feature points in a video

This chapter is about reading, writing, and processing video sequences. The objective is to be able to analyze a complete video sequence. As an example, in this recipe, you will learn how to perform temporal analysis of the sequence in order to track feature points as they move from frame to frame.

How to do it...

To start the tracking process, the first thing to do is to detect the feature points in an initial frame. You then try to track these points in the next frame. Obviously, since we are dealing with a video sequence, there is a good chance that the object on which the feature points are found has moved (this motion can also be due to camera movement). Therefore, you must search around a point's previous location in order to find its new location in the next frame. This is what accomplishes the `cv::calcOpticalFlowPyrLK` function. You input two consecutive frames and a vector of feature points in the first image; the function returns a vector of new point locations. To track points over a complete sequence, you repeat this process from frame to frame. Note that as you follow the points across the sequence, you will unavoidably lose track of some of them such that the number of tracked feature points will gradually reduce. Therefore, it could be a good idea to detect new features from time to time.

We will now take benefit of the framework we defined in the previous recipes and we will define a class that implements the `FrameProcessor` interface introduced in the *Processing the video frames* recipe of this chapter. The data attributes of this class include the variables that are required to perform both the detection of feature points and their tracking:

```
class FeatureTracker : public FrameProcessor {

  cv::Mat gray;         // current gray-level image
  cv::Mat gray_prev;      // previous gray-level image
  // tracked features from 0->1
  std::vector<cv::Point2f> points[2];
  // initial position of tracked points
  std::vector<cv::Point2f> initial;
  std::vector<cv::Point2f> features;  // detected features
  int max_count;     // maximum number of features to detect
  double qlevel;    // quality level for feature detection
  double minDist;   // min distance between two points
  std::vector<uchar> status; // status of tracked features
  std::vector<float> err;    // error in tracking

  public:
```

```
FeatureTracker() : max_count(500), qlevel(0.01), minDist(10.) {}
```

Next, we define the `process` method that will be called for each frame of the sequence. Basically, we need to proceed as follows. First, feature points are detected if necessary. Next, these points are tracked. You reject points that you cannot track or you no longer want to track. You are now ready to handle the successfully tracked points. Finally, the current frame and its points become the previous frame and points for the next iteration. Here is how to do this:

```
void process(cv:: Mat &frame, cv:: Mat &output) {

  // convert to gray-level image
  cv::cvtColor(frame, gray, CV_BGR2GRAY);
  frame.copyTo(output);

  // 1. if new feature points must be added
  if(addNewPoints())
  {
    // detect feature points
    detectFeaturePoints();
    // add the detected features to
    // the currently tracked features
    points[0].insert(points[0].end(),features.begin(),features.end());
    initial.insert(initial.end(),features.begin(),features.end());
  }

  // for first image of the sequence
  if(gray_prev.empty())
    gray.copyTo(gray_prev);

  // 2. track features
  cv::calcOpticalFlowPyrLK(gray_prev, gray, // 2 consecutive images
  points[0], // input point positions in first image
  points[1], // output point positions in the 2nd image
  status,    // tracking success
  err);      // tracking error
```

```
  // 3. loop over the tracked points to reject some
  int k=0;
  for( int i= 0; i < points[1].size(); i++ ) {

    // do we keep this point?
    if (acceptTrackedPoint(i)) {
      // keep this point in vector
      initial[k]= initial[i];
      points[1][k++] = points[1][i];
    }
  }

  // eliminate unsuccesful points
  points[1].resize(k);
  initial.resize(k);

  // 4. handle the accepted tracked points
  handleTrackedPoints(frame, output);

  // 5. current points and image become previous ones
  std::swap(points[1], points[0]);
  cv::swap(gray_prev, gray);
}
```

This method makes use of four utility methods. It should be easy for you to change any of these methods in order to define a new behavior for your own tracker. The first of these methods detects the feature points. Note that we already discussed the `cv::goodFeatureToTrack` function in the first recipe of Chapter 8, *Detecting Interest Points*:

```
// feature point detection
void detectFeaturePoints() {

  // detect the features
  cv::goodFeaturesToTrack(gray, // the image
```

```
    features,   // the output detected features
    max_count,  // the maximum number of features
    qlevel,     // quality level
    minDist);   // min distance between two features
}
```

The second method determines whether new feature points should be detected:

```
// determine if new points should be added
bool addNewPoints() {

  // if too few points
    return points[0].size()<=10;
}
```

The third method rejects some of the tracked points based on a criteria defined by the application. Here, we decided to reject points that do not move (in addition to those that cannot be tracked by the `cv::calcOpticalFlowPyrLK` function):

```
// determine which tracked point should be accepted
bool acceptTrackedPoint(int i) {

  return status[i] &&
  // if point has moved
  (abs(points[0][i].x-points[1][i].x)+(abs(points[0][i].y-points[1][i].y))>2);
}
```

Finally, the fourth method handles the tracked feature points by drawing all of the tracked points with a line that joins them to their initial position (that is, the position where they were detected the first time) on the current frame:

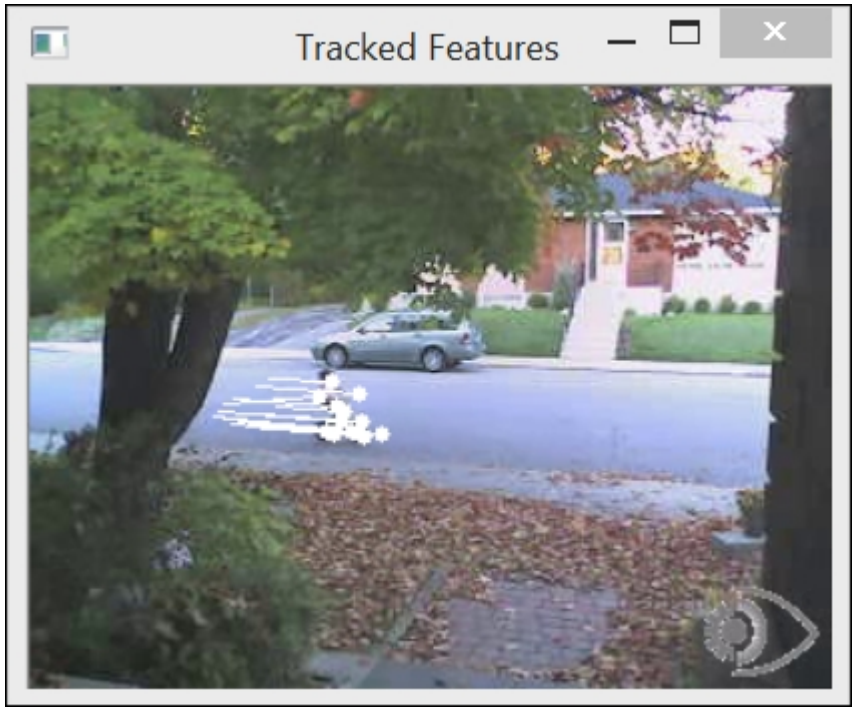
```
// handle the currently tracked points
void handleTrackedPoints(cv:: Mat &frame, cv:: Mat &output) {

  // for all tracked points
  for(int i= 0; i < points[1].size(); i++ ) {
```

```
      // draw line and circle
      cv::line(output,
        initial[i],  // initial position
        points[1][i],// new position
        cv::Scalar(255,255,255));
        cv::circle(output, points[1][i], 3, cv::Scalar(255,255,255),-1);
    }
  }
```

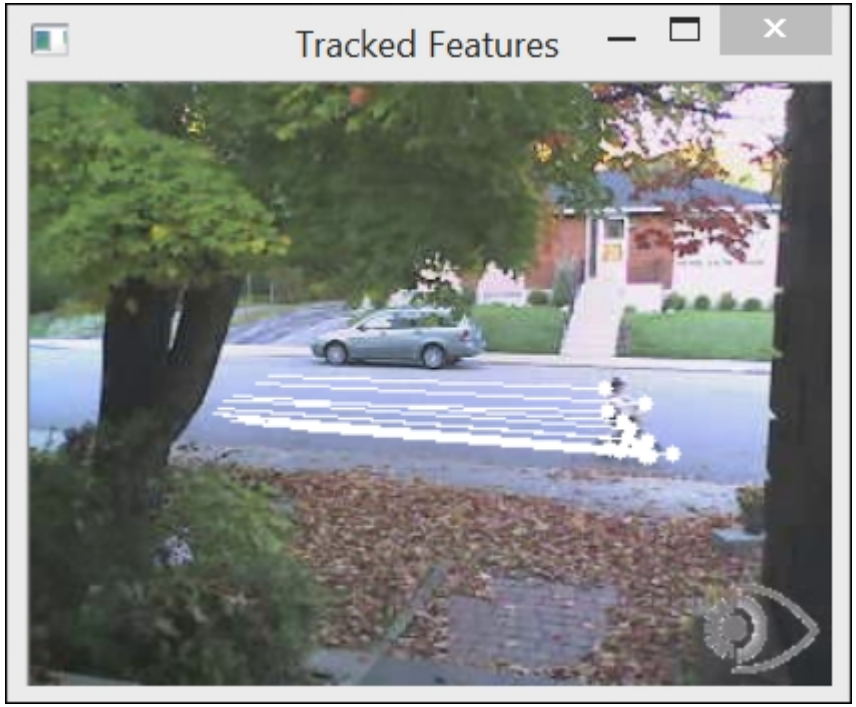
A simple main function to track feature points in a video sequence would then be written as follows:

```
int main()
{
  // Create video procesor instance
  VideoProcessor processor;

  // Create feature tracker instance
  FeatureTracker tracker;
   // Open video file
   processor.setInput("../bike.avi");

  // set frame processor
  processor.setFrameProcessor(&tracker);

  // Declare a window to display the video
  processor.displayOutput("Tracked Features");

  // Play the video at the original frame rate
  processor.etDelayetDelay(1000./processor.getFrameRate());

  // Start the process
  processor.run();
}
```

The resulting program will show you the evolution of the moving tracked features over time. Here are, for example, two such frames at two different instants. In this video, the camera is fixed. The young cyclist is, therefore, the only moving object. Here is the result that is obtained after a few frames have been processed:



A few seconds later, we obtain the following frame:



How it works...

To track feature points from frame to frame, we must locate the new position of a feature point in the subsequent frame. If we assume that the intensity of the feature point does not change from one frame to the next one, we are looking for a displacement *(u,v)* as follows:

$$I_t(x,y) = I_{t+1}(x+u, y+v)$$

Here, I_t and I_{t+1} are the current frame and the one at the next instant, respectively. This constant intensity assumption generally holds for small displacement in images that are taken at two nearby instants. We can then use the Taylor expansion in order to approximate this equation by an equation that involves the image derivatives:

$$I_{t+1}(x+u, y+v) \approx I_t(x,y) + \frac{\partial I}{\partial x}u + \frac{\partial I}{\partial y}v + \frac{\partial I}{\partial t}$$

This latter equation leads us to another equation (as a consequence of the constant intensity assumption that cancels the two intensity terms):

$$\frac{\partial I}{\partial x}u+\frac{\partial I}{\partial y}v=-\frac{\partial I}{\partial t}$$

This well-known constraint is the fundamental **optical flow** constraint equation. This constraint is exploited by the so-called Lukas-Kanade feature-tracking algorithm that also makes an additional assumption that the displacement of all points in the neighborhood of the feature point is the same. We can, therefore, impose the optical flow constraint for all of these points with a unique *(u,v)* unknown displacement. This gives us more equations than the number of unknowns (2), and therefore, we can solve this system of equations in a mean-square sense. In practice, it is solved iteratively and the OpenCV implementation also offers us the possibility to perform this estimation at a different resolution in order to make the search more efficient and more tolerant to larger displacement. By default, the number of image levels is `3` and the window size is `15`. These parameters can obviously be changed. You can also specify the termination criteria, which define the conditions that stop the iterative search. The sixth parameter of `cv::calcOpticalFlowPyrLK` contains the residual mean-square error that can be used to assess the quality of the tracking. The fifth parameter contains binary flags that tell us whether tracking the corresponding point was considered successful or not.

The preceding description represents the basic principles behind the Lukas-Kanade tracker. The current implementation contains other optimizations and improvements that make the algorithm more efficient in the computation of the displacement of a large number of feature points.

See also

- Chapter 8, *Detecting Interest Points*, has a discussion on feature point detection.
- The classic article by B. Lucas and T. Kanade, *An Iterative Image Registration Technique with an Application to Stereo Vision* in *Int. Joint Conference in Artificial Intelligence, pp. 674-679, 1981*, describes the original feature point tracking algorithm.
- The article by J. Shi and C. Tomasi, *Good Features to Track in IEEE Conference on Computer Vision and Pattern Recognition*, pp. 593-600, 1994, describes an improved version of the original feature point tracking algorithm.