Chapter 7

Video Processing

In this chapter we will discuss how to use video to get motion analysis. These techniques can be used in applications such as video surveillance, robot navigation, traffic surveillance and other real-time video analysis. Motion analysis is also used for video compression algorithms because it facilitates the process of finding redundancies between frames. In this study guide we will limit the scope to 2D images (2D motion estimation).

The area of motion detection can be classified in three main groups [1]:

- Motion detection analyses a video sequence to find if anything has moved within a certain time frame. Although a simple approach can be implemented, problems such as small objects moving randomly make practical applications limited to those running under controlled environments (static camera and backgrounds, steady lighting etc).
- Object detection tries to locate a certain object as well as its speed and direction. This usually uses more complex techniques and may also involve areas of pattern recognition. However there are many special cases where it is possible to uniquely identify an object using simple cues (for example colour distribution) and still compute the kinematics of the object in question.
- Derivation of 3D object properties from two or more 2D projections acquired by two or more cameras.

In a digital image, motion analysis can be based on individual pixels. Given two consecutive frames in a video sequence, four different cases occur:

- a pixel p remains static
- a pixel p moves to a new position
- a pixel p disappears from the field of the camera
- a pixel p changes its value due to lighting changes

Some basic assumptions have to be made in order to simplify the method, so applications can in real-time. Such assumptions might include, for example, maximum velocity for the objects, limited acceleration of objects, and mutual correspondence (no occlusion, no object rotation, no illumination changes, etc).

7.1 Optical Flow

Optical flow algorithms main objective is to find the velocity and the direction of moving objects. We show in this section that the process of finding the vectors' properties is not trivial due to certain ambiguities that arise from the limited 2D view of a scene. Also, the optical flow does not necessarily reflects the *true* motion of objects. For example, a homogeneous sphere rotating in front of a camera may appear to be static. All the pixels are moving, but because the values are the same or at least very similar to the previous frame, no velocity can be computed. Another classic example is the barber shop pole, in which the optical flow vectors might point up, when in fact the pixels are moving along the horizontal line.

The luminance variation of a video sequence is $\psi(x, y, t)$, where the pixel is at (x, y) at time t ([2]). The equation for optical flow is usually represented by:

$$\frac{\delta\psi}{\delta x}\upsilon_x + \frac{\delta\psi}{\delta y}\upsilon_y + \frac{\delta\psi}{\delta t} = 0 \tag{7.1}$$

The velocity vector (υ_x, υ_y) is the gradient vector of $\psi(x, y, t)$. In practise a few problems arise at this point. The first is that intensity (for all pixels) has to be considered constant, otherwise there is no reliable way of differentiating actual movement from illumination changes.

The second problem is that the equation above has no single solution (one equation to two unknowns) and additional constraints need to be imposed.

The third problem is called the *aperture problem*. This problem is related to the fact that regions along edges tend to have similar pixel intensities. Figure 7.1 shows that the optical flow for point 1 cannot be uniquely determined, while for point 2 it can.

Finally the forth problem is that in regions where the pixels have constant brightness the flow is indeterminate.

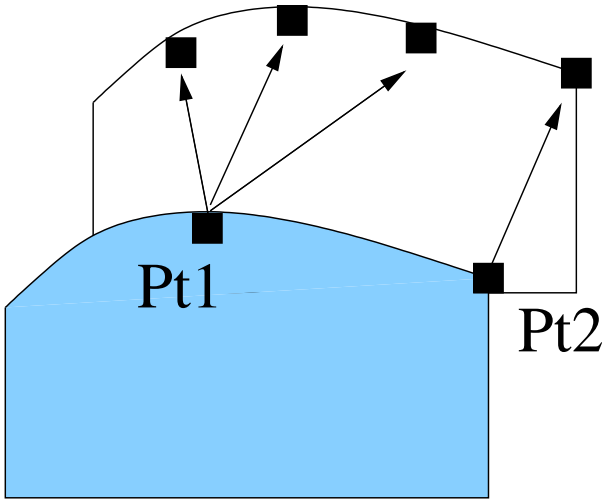

Figure 7.1: Aperture problem.

A practical example of the problems described above is illustrated by the barber's shop sign (figure 7.2). The actual flow, i.e., the place where the real pixels are moving to, is different than the optical flow result. Such scenarios deceive even our human vision into believing that the movement of the stripes is downwords, even though we know it is the wrong direction.

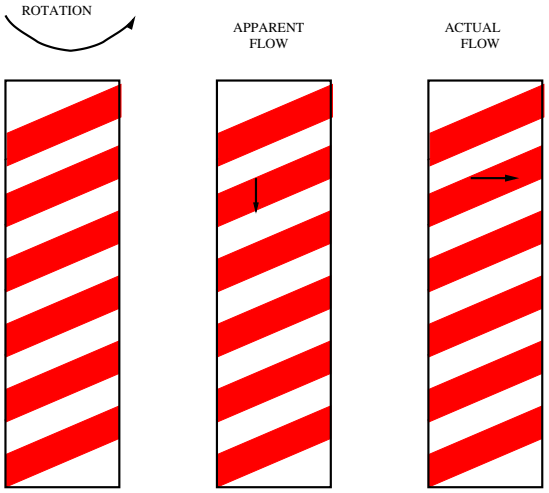

Figure 7.2: Determining optical flow in a barber's shop sign.

7.1.1 Block Matching

The previous section states that pixel motion estimation is an ill-defined problem because there might be several pixels in the same frame with identical properties. A standard approach to deal with this problem is to use a set of pixels, usually a square block. This technique is called *block-matching*. The simplest approach is to divide the frame into equal sized blocks and search for the new position of an entire block in the next frame. This approach makes the following assumptions:

- The objects in the frame are rigid
- The illumination changes are small

If the search for the position of the block is over the entire frame, it is unfeasible in practice to run block matching algorithms in real-time. Consider for example that a 512x512 image is split into blocks of 16x16 pixels. The total number of operations required to match two frames would be approximately $275 * 10^9$. If a sequence had a few frames per second, it is easy to conclude that it would be impossible to run such an application in real-time. One solution is to minimise the search area by limiting it to areas that are close to each original block.

A matching criteria for blocks belonging to different frames needs to be adopted. A simple criteria could be similarity based on the minimum square error. This criteria could partially compensate problems with occlusions and illumination changes, e.g., an object moves to a shaded area of the environment. On the other hand, for large regions of the frames where the same pattern is repeated, there is no reliable way to find the next match. In figure 7.3, the blocks with the images of the grass will not be matched correctly. Matching criteria that takes deformation and rotation are computationally more expensive.

$$error = \sqrt{\sum (f(x,y) - f'(x',y'))^2} \tag{7.2}$$

where: $f(x,y)$ is the sub-image for a block from a frame and $f'(x',y')$ is the sub-image of a block of the next frame. The block with the minimum error is chosen as the best matching.

A sample of a block matching result using this criteria can be seen on figure 7.3 for the sequence *football*. In this sequence the player are moving randomly and the camera is trying to

follow the ball. Notice that there is a homogeneous flow to the right on the background, while the players' blocks are moving in other directions.

Figure 7.3: A block matching algorithm in action.

In order to minimise the number of operations, one can make the assumption that no object will travel beyond a certain velocity. Between two consecutive frames there one can assure that no blocks will move more than a certain distance. To make the algorithm even faster, other search strategies can be used. For example, the three-step search algorithm uses search areas of different sizes. In each step the areas are smaller than the previous, which makes the search very fast. The only drawback is that in some images a local minimum can be found, so there is no guaranty that the minimum error criteria is fulfilled (see figure 7.4).

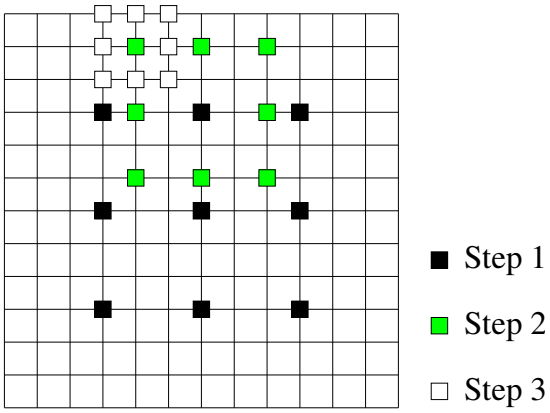

Figure 7.4: Three steps approach for block matching.

In figure 7.5 the same sequence is shown, but the blockmatching algorithm uses the three steps approach. Notice that the matching is not as good as in figure 7.3.

7.1.2 Block Matching in practise

Using the block matching approach with cameras in real-time requires heavy processing. So there needs to be a compromise between accuracy of the optical flow and performance, as discussed in the previous section.

The direction of the arrows can be tested when moving the camera around pointing to a scene where the objects are clearly defined, and where there is not much ambiguities between the blocks. Typical movements of the camera include translation, rotation and zooming. Note

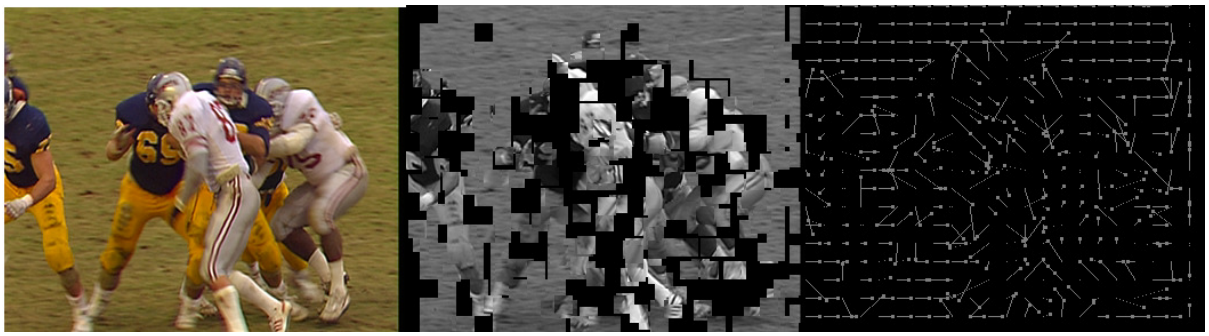

Figure 7.5: A block matching algorithm using the three steps approach.

that the arrows should mostly go on the opposite direction of the movement of the camera. When rotating, the centre would show very low optical flow, while in the edges outside the centre of the rotation there should be arrows that are normal to the radius. When zooming in or out, the arrows should point to the centre of the zoom if zooming in, and outside the centre if zooming out.

Images in figures 7.6, 7.7, 7.8 and 7.9 show my own implementation of a simple block matching optical flow. The arrows are not very accurate, as the objects not necessarily are of a size that is adequate for the block size. In these images, the resolution is 640x480 and there are 16x16 blocks (256 blocks). The application can run with a reasonable frame rate using the three steps method, but it is very slow if using the exhaustive search (it takes a few seconds per frame to process that).

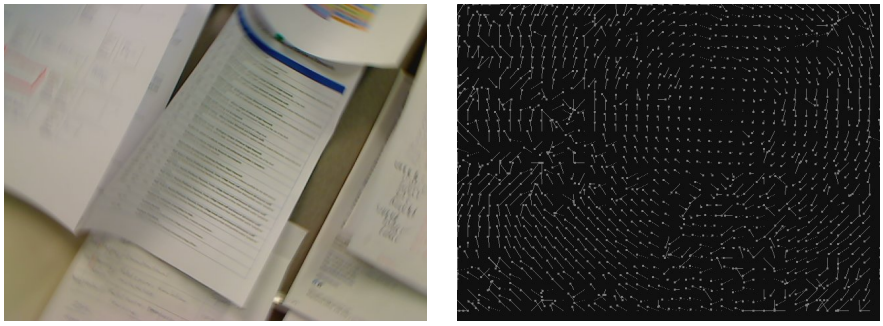

Figure 7.6: Optical flow example: camera rotates (notice the centre of the rotation looking at the arrows.

7.2 Other optical flow approaches

In OpenCV there are four functions to compute optical flow:

- cvCalcOpticalFlowBM
- cvCalcOpticalFlowLK
- cvCalcOpticalFlowPyrLK
- cvCalcOpticalFlowHS

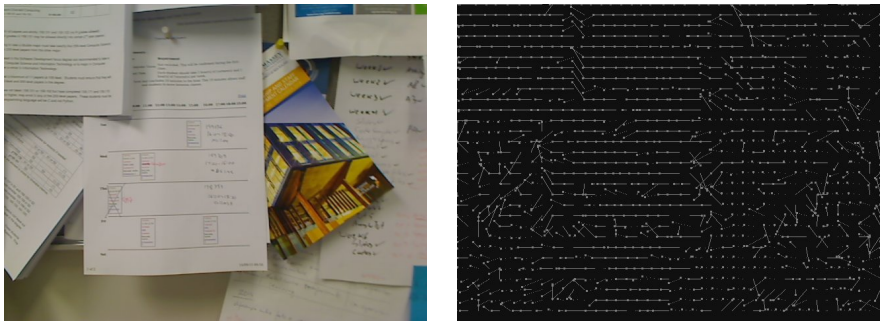

Figure 7.7: Optical flow example: camera moves to the right (most arrows point to the left).

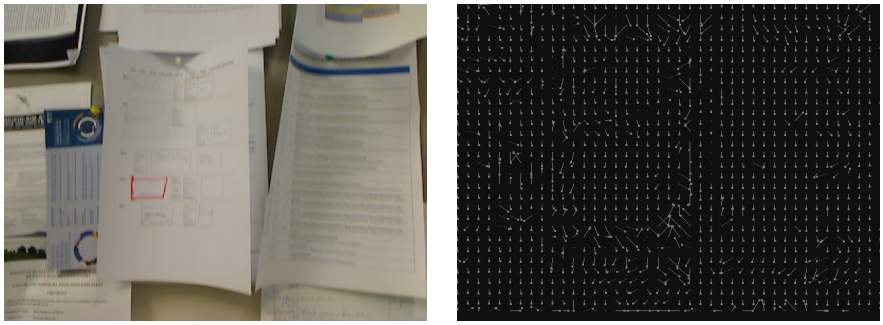

Figure 7.8: Optical flow example: camera goes up.

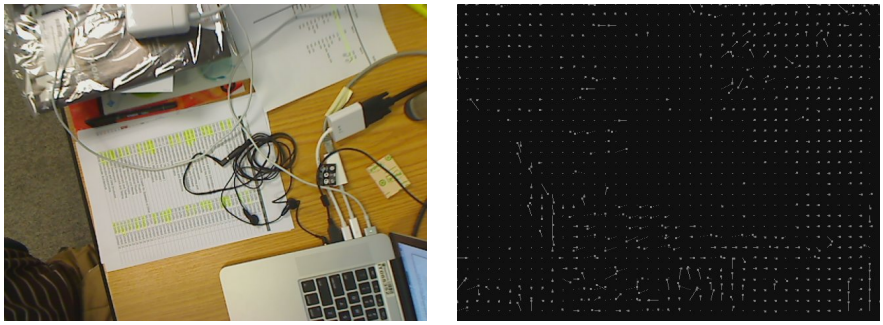

Figure 7.9: Optical flow example: camera approaches the objects (zoom).

The first function is an implementation of the Blocking Matching algorithm shown in section 7.1.1. Two are based on Lucas and Kanade algorithm [3] and the last one is based on Horn and Schunck ([4]). Both algorithms are based on intensity differential and use pixels (rather than an entire subwindow) to compute Optical Flow.

Assumptions about the nature of the movement constrain the scope of the optical flow algorithms. For example, if objects move too fast the optical flow may be incorrect. Occlusions can also create problems because the algorithm may lose track of features of the image.

7.2.1 Horn and Schunck

Horn and Schunck [4] assumptions are:

- brightness is constant for each pixel along the sequence (in time)
- smoothness of the flow is constrained

The method uses the minimization of a global energy functional (not written here), generating the following equations:

$$I_x(I_x u + I_y v + I_t) - \alpha^2 \Delta u = 0 \quad and \quad I_y(I_x u + I_y v + I_t) - \alpha^2 \Delta v = 0 \tag{7.3}$$

Where I_x, I_y and I_t are the partial derivatives on x, y,and time, and u and v are the velocities at point $(x.y)$ of the image, α is a constant.

The equations above can be rewritten as:

$$(I_x^2 + \alpha^2)u + I_x I_y v = \alpha^2 u_{avg} - I_x I_t \quad and \quad (I_y^2 + \alpha^2)v + I_x I_y u = \alpha^2 v_{avg} - I_x I_t \tag{7.4}$$

Laplacians are estimated in x,y directions. Both the rate of change of the brightness and smoothness have their errors minimised by finding suitable values for the optical flow (u, v) at each pixel. Because the equation is applied for a region (getting the averages for u and v), the system of linear equations above has to be solved iteratively using Jacobi method. This becomes:

$$u(i+1) = u_{avg}(i) - \frac{I_x(I_x u_{avg}(i) + I_y v_{avg}(i) + I_t)}{\alpha^2 + I_x^2 + I_y^2} \tag{7.5}$$

and

$$v(i+1) = v_{avg}(i) - \frac{I_y(I_x u_{avg}(i) + I_y v_{avg}(i) + I_t)}{\alpha^2 + I_x^2 + I_y^2} \tag{7.6}$$

So homogeneous areas have a valid vector for every pixel, as every pixel gets the average of the neighbours, making a high density vector field. This can be both an advantage and a hindrance depending on the application.

7.2.2 Lukas and Kanade

Lucas and Kanade [3] assumptions are:

- no articulation(so edges original forms are preserved, they only translate or rotate)
- the distance travelled by one edge between two sequential frames is small (typically less than one pixel).

The problem becomes finding the direction and displacement of edges. In figure 7.10 one can derive the idea for 1 dimension. To minimise the error of the difference between the curves in relation to h, Lukas and Kanade approximated the distance h to:

$$h \approx \frac{\sqrt{\sum F'(x)[G(x) - F(x)]}}{\sum F'(x)^2} \tag{7.7}$$

With these assumptions, a sub-window can be used to determine the optical flow equation for every pixel inside the sub-window:

$$I_x u + I_y v = -I_t \tag{7.8}$$

With the equation above, a system of linear equations can be built around a sub-window, and solved by least-square method. Because of the conditions to solve the system of equations

requires a certain property from the matrices, and these properties are similar to the condition to get a corner, it is possible to determine "good points" or "good features to follow" from a single image. The problem is when the background has similar sub-windows, then the points can jump from the object being tracked to the background. Although this is a smooth optical flow from one frame to the next, it is not what the method intended to do. In practice, other features (such as colours, or other tracking features) can be used to improve the accuracy and reliability of the method.

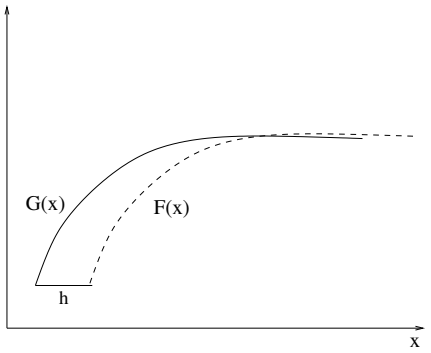

Figure 7.10: Lukas Kanade approach (adapted from [3]).

7.3 Getting frames from a WebCam with OpenCV

This is a sample of a code that acquire frames from a webcam using the C++ API (listing 7.1). A sample image is shown in figure 7.11.

This code makes use of the new c++11 chronos library to estimate the frame rates that the camera can deliver. The code includes a simple printing of the frame rates on the screen, so the user can easily see the effects of lighting conditions and the cost of further processing each frame.

The lighting conditions for the cameras in the Linux laboratory affect the frame rates directly. This is the case because the exposure is in auto mode by default for the web cameras installed in the lab. When the images are darker under poor illumination conditions, the exposure time is automatically increased in order to grab more light, and therefore the camera cannot deliver as many frames per second as when working with normal illumination conditions. It is possible to avoid this issue by turning off the auto exposure. One way to achieve that is to use guvcview, an application that can control UVC cameras.

The application guvcview can be installed directly via apt-get in any linux distribution. It can also be used simultaneously with opencv applications to control all the camera parameters that are available for this particular model. The command to achieve that is: `guvcview --control_only`.

In the listing 7.1 the OpenCV function `waitKey()` is being used to make a simple user interface. The function accepts one parameter, time in miliseconds. The application will halt and wait for the amount of time that was passed as a parameter. Normally this is a short time to allow for the frame to be displayed in the window `‘‘WebCam’’` created previously. The function returns an integer that corresponds to a certain key (pressed in the keyboard). In the case of the listing, there are two keys that if pressed will quit the program without any further processing, 'esc' and 'q'. The numbers correspond to 113 and 27. For other keys it is better to do by trial and error, as the keys's numbers might depend on architecture and OS.

Listing 7.1: WebCam with OpenCV C++ API

```
#include <stdio.h>
#include <chrono>
#include <ctime>
#include <iostream>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc/imgproc.hpp>

using namespace cv;
using namespace std;
using namespace chrono;

/*******************************
* compile with:
* g++ -std=c++11 camera_with_fps.cpp -o camera_with_fps `pkg-
    config --cflags --libs opencv`
*******************************/

Mat frame;//, image;
int main(int argc, char** argv)
{
  VideoCapture cap;
  cap.open(0);
  if (!cap.isOpened())
     {
         cout << "Failed to open camera" << endl;
         return 0;
     }
   cout << "Opened camera" << endl;
   namedWindow("WebCam", 1);
   cap.set(CV_CAP_PROP_FRAME_WIDTH, 640);
   cap.set(CV_CAP_PROP_FRAME_HEIGHT, 480);
   cap >> frame;
   printf("frame size %d %d \n",frame.rows, frame.cols);
   int key=0;
   double fps=0.0;
   while (1){
     system_clock::time_point start = system_clock::now();
       cap >> frame;
       if( frame.empty() )
       break;

       char printit[100];
       sprintf(printit,"%2.1f",fps);
       putText(frame, printit, cvPoint(10,30), FONT_HERSHEY_PLAIN,
           2, cvScalar(255,255,255), 2, 8);
       imshow("WebCam", frame);
```

```
46          key=waitKey(1);
47          if(key==113 || key==27) return 0;//either esc or 'q'
48        system_clock::time_point end = system_clock::now();
49        double seconds = std::chrono::duration_cast<std::chrono::
              microseconds>(end - start).count();
50        fps = 1000000/seconds;
51        cout << "frames " << fps << " seconds " << seconds << endl;
52      }
53  }
```

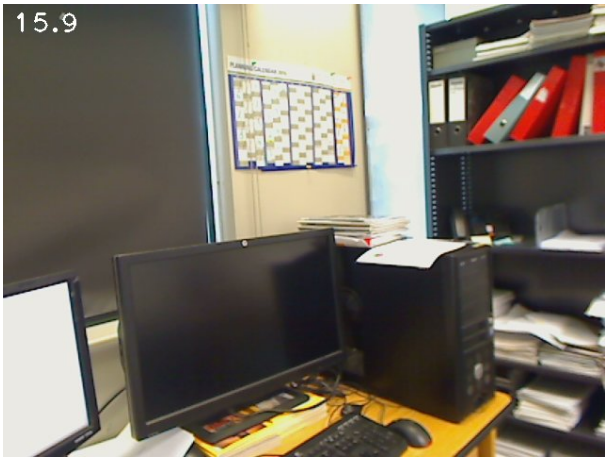

Figure 7.11: Image from camera with fps. Notice that the frame rates are estimated in real-time and can change with the conditions.

7.4 Reading

Read section 10.6 of Gonzales & Woods. Read the original articles about optical flow cited in the references (both articles are available on the Web).

7.5 Exercises

1. Download the sequences (in YUV format) from Stream. Considering that you know the size of the frames and that the yuv format uses three bytes per channel in this order, devise an algorithm to read and show the yuv sequence frame by frame.

2. Write a simple version of the Block Matching algorithm using a similarity criteria to find the block's position on the other frame. Draw the vector field using an arrow for each block, indicating direction and velocity. You need to limit the size (indicating velocity) of each arrow in order to see the field properly.

3. Run the sample LucasKanade implemented in OpenCV using the web camera. Use markers on each fingertip and see if the algorithms is robust enough to follow each finger independently. What happens when the marker encounter a deeper edge from the background?

Bibliography

[1] M. Sonka, V. Hlavac, and R. Boyle, *Image Processing, Analysis and Machine Vision*. PWS Publishing, 1999.

[2] Y. Wang, J. Ostermann, and Y.-Q. Zhang, *Video Processing and Communications*. Prentice Hall, 2002.

[3] B. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," in *Proc. of 7th International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 674–679, 1981.

[4] B. K. Horn and B. G. Schunck, "Determining optical flow," *Artificial Intelligence*, no. 17, pp. 185–203, 1981.