Nota: Algunas de las imágenes que aparecen en esta presentación provienen del libro:
Visión por Computador: fundamentos y métodos.
Arturo de la Escalera Hueso. Prentice Hall.

Intelligent Systems Lab

Sistemas de Percepción

Visión por Computador

Arturo de la Escalera
José María Armingol
Fernando García
David Martín
Abdulla Al-Kaff

uc3m | Universidad Carlos III de Madrid

Intelligent Systems Lab

Detección de movimiento Lucas-Kanade piramidal Camshift

Lucas-Kanade piramidal

- El algoritmo de Lucas-Kanade se basa en detectar la distancia en píxeles entre dos imágenes consecutivas
- Se necesita una imagen de 8bits y un único canal:
  - CV_8UC1 en niveles de gris
  - No se produzcan cambios de iluminación importantes
  - El desplazamiento entre imágenes
- ¿Cómo es posible conocer el movimiento de un píxel?
  - IMPOSIBLE, es necesario disponer de más información.
- El algoritmo de Lucas-Kanade supone que los píxeles cercanos tienen un movimiento similar, centrándose en regiones de píxeles en lugar de en un único píxel.
- Por último, es necesario determinar los puntos de interés en la imagen (esquinas) utilizando la detección de esquinas presentada por Shi-Tomasi.

Lucas-Kanade piramidal

- El algoritmo de Lucas-Kanade únicamente funciona para distancias reducidas en píxeles
- Cuando la distancia se incrementa es necesario utilizar la implementación piramidal
  - Se reduce la resolución de la imagen (píxeles más grandes) y, por tanto, se reduce la distancia entre los mismos
- Se utilizará la función **goodFeaturesToTrack** (detección de esquinas Shi-Tomasi) para detectar los puntos interés utilizados en la implementación de piramidal Lucas-Kanade.

Lucas-Kanade piramidal

Intelligent Systems Lab

goodFeaturesToTrack

Determines strong corners on an image.

C++: void **goodFeaturesToTrack**(InputArray **image**, OutputArray **corners**, int **maxCorners**, double **qualityLevel**, double **minDistance**, InputArray **mask**=noArray(), int **blockSize**=3, bool **useHarrisDetector**=false, double **k**=0.04)

Parameters:
- **image** – Input 8-bit or floating-point 32-bit, single-channel image.
- **eig_image** – The parameter is ignored.
- **temp_image** – The parameter is ignored.
- **corners** – Output vector of detected corners.
- **maxCorners** – Maximum number of corners to return. If there are more corners than are found, the strongest of them is returned.
- **qualityLevel** – Parameter characterizing the minimal accepted quality of image corners. The parameter value is multiplied by the best corner quality measure, which is the minimal eigenvalue (see **cornerMinEigenVal()**) or the Harris function response (see **cornerHarris()**). The corners with the quality measure less than the product are rejected. For example, if the best corner has the quality measure = 1500, and the `qualityLevel=0.01` , then all the corners with the quality measure less than 15 are rejected.
- **minDistance** – Minimum possible Euclidean distance between the returned corners.
- **mask** – Optional region of interest. If the image is not empty (it needs to have the type `CV_8UC1` and the same size as `image`), it specifies the region in which the corners are detected.
- **blockSize** – Size of an average block for computing a derivative covariation matrix over each pixel neighborhood. See **cornerEigenValsAndVecs()** .
- **useHarrisDetector** – Parameter indicating whether to use a Harris detector (see **cornerHarris()**) or **cornerMinEigenVal()** .
- **k** – Free parameter of the Harris detector.

The function finds the most prominent corners in the image or in the specified image region, as described in [Shi94]:

1. Function calculates the corner quality measure at every source image pixel using the **cornerMinEigenVal()** or **cornerHarris()** .
2. Function performs a non-maximum suppression (the local maximums in *3 x 3* neighborhood are retained).
3. The corners with the minimal eigenvalue less than $\texttt{qualityLevel} \cdot \max_{x,y} \texttt{qualityMeasureMap}(x,y)$ are rejected.
4. The remaining corners are sorted by the quality measure in the descending order.
5. Function throws away each corner for which there is a stronger corner at a distance less than `maxDistance`.

Lucas-Kanade pyramidal

calcOpticalFlowPyrLK

Calculates an optical flow for a sparse feature set using the iterative Lucas-Kanade method with pyramids.

C++: void **calcOpticalFlowPyrLK**(InputArray **prevImg**, InputArray **nextImg**, InputArray **prevPts**, InputOutputArray **nextPts**, OutputArray **status**, OutputArray **err**, Size **winSize**=Size(21,21), int **maxLevel**=3, TermCriteria **criteria**=TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 30, 0.01), int **flags**=0, double **minEigThreshold**=1e-4)

Parameters:

- **prevImg** – first 8-bit input image or pyramid constructed by **buildOpticalFlowPyramid()**.
- **nextImg** – second input image or pyramid of the same size and the same type as `prevImg`.
- **prevPts** – vector of 2D points for which the flow needs to be found; point coordinates must be single-precision floating-point numbers.
- **nextPts** – output vector of 2D points (with single-precision floating-point coordinates) containing the calculated new positions of input features in the second image; when `OPTFLOW_USE_INITIAL_FLOW` flag is passed, the vector must have the same size as in the input.
- **status** – output status vector (of unsigned chars); each element of the vector is set to 1 if the flow for the corresponding features has been found, otherwise, it is set to 0.
- **err** – output vector of errors; each element of the vector is set to an error for the corresponding feature, type of the error measure can be set in `flags` parameter; if the flow wasn't found then the error is not defined (use the `status` parameter to find such cases).
- **winSize** – size of the search window at each pyramid level.
- **maxLevel** – 0-based maximal pyramid level number; if set to 0, pyramids are not used (single level), if set to 1, two levels are used, and so on; if pyramids are passed to input then algorithm will use as many levels as pyramids have but no more than `maxLevel`.
- **criteria** – parameter, specifying the termination criteria of the iterative search algorithm (after the specified maximum number of iterations `criteria.maxCount` or when the search window moves by less than `criteria.epsilon`.
- **flags** –

  operation flags:

  - **OPTFLOW_USE_INITIAL_FLOW** uses initial estimations, stored in `nextPts`; if the flag is not set, then `prevPts` is copied to `nextPts` and is considered the initial estimate.
  - **OPTFLOW_LK_GET_MIN_EIGENVALS** use minimum eigen values as an error measure (see `minEigThreshold` description); if the flag is not set, then L1 distance between patches around the original and a moved point, divided by number of pixels in a window, is used as a error measure.
- **minEigThreshold** – the algorithm calculates the minimum eigen value of a 2x2 normal matrix of optical flow equations (this matrix is called a spatial gradient matrix in [Bouguet00]), divided by number of pixels in a window; if this value is less than `minEigThreshold`, then a corresponding feature is filtered out and its flow is not processed, so it allows to remove bad points and get a performance boost.

Lucas-Kanade piramidal

- Lucas-Kanade pyramidal : OpenCV
  - Esquema de etapas:
    - Cargar las imágenes.
    - Detección de bordes.
    - Aplicar el flujo óptico para detectar las esquinas en la siguiente imagen.
    - Mostrar el estado y los errores.
    - Dibujar las esquinas y las líneas .
    - Mostrar las imágenes.
    - Liberar la memoria.
    - Finalizar el programa.

      - Probar diferentes tamaño de ventanas y número de niveles.
      - Probar con diferentes imágenes.

Lucas-Kanade piramidal

```
1   // Pyramidal_LucasKanade: Optical Flow
2   #include "opencv\cv.hpp"
3   #include <iostream>
4
5   using namespace cv;
6   using namespace std;
7
8
9   int main(int argc, char* argv[])
10  {
11      // Objects
12      Mat original01, original02, original03, original04, original05;
13      Mat colorCanvas;
14
15      // Load image from disk
16      original01 = imread("Square01.png", CV_LOAD_IMAGE_GRAYSCALE);
17      original02 = imread("Square02.png", CV_LOAD_IMAGE_GRAYSCALE);
18      original03 = imread("Square03.png", CV_LOAD_IMAGE_GRAYSCALE);
19      original04 = imread("Square04.png", CV_LOAD_IMAGE_GRAYSCALE);
20      original05 = imread("Square05.png", CV_LOAD_IMAGE_GRAYSCALE);
21      colorCanvas= imread("Square01.png", CV_LOAD_IMAGE_COLOR);
22
23      if (!original01.data || !original02.data || !original03.data
24          || !original04.data || !original05.data)
25      {
26          cout << "error loading images" << endl;
27          system("pause");
28          return 1;
29      }
```

Lucas-Kanade piramidal

```
31      /// Parameters for Shi-Tomasi algorithm
32      vector<Point2f> cornersA, cornersB;
33      double qualityLevel = 0.01;
34      double minDistance = 10;
35      int blockSize = 3;
36      bool useHarrisDetector = false; // cornerMinEigenVal then
37      double k = 0.04;
38      int maxCorners = 4;
39
40      /// Apply corner detection
41      goodFeaturesToTrack(original01,
42                          cornersA,
43                          maxCorners,
44                          qualityLevel,
45                          minDistance,
46                          Mat(),
47                          blockSize,
48                          useHarrisDetector,
49                          k);
50
51      // status: return the detected corners
52      vector<uchar> status;
53      // error: return the confidence of the detection
54      vector<float> error;
55      // play with different sizes and levels
56      int winsize = 11;
57      int maxlvl = 5;
```

Lucas-Kanade piramidal

```
59      // Pyramidal Lucas Kanade optical flow
60      calcOpticalFlowPyrLK(original01, original02, cornersA, cornersB, status, error,
61          Size(winsize, winsize), maxlvl);
62
63      /// Draw corners detected
64      int radius = 2;
65      cout << "Number of cornersA detected: " << cornersA.size() << endl;
66      cout << "Optical Flow corners detected: " << cornersB.size() << endl;
67
68      // the size of cornersA and cornersB is the same even if not detected
69      for (int i = 0; i < cornersA.size(); i++)
70      {
71          circle(colorCanvas, cornersA[i], radius, Scalar(255, 0, 0), -1, 8, 0);
72          circle(colorCanvas, cornersB[i], radius, Scalar(0, 255, 0), -1, 8, 0);
73          cout << "status: " << (int)status[i];
74          cout << " error: " << error[i];
75          cout << endl;
76          line(colorCanvas, cornersA[i], cornersB[i], Scalar(0, 0, 255));
77      }
78
79      // Windows for all the images
80      namedWindow("ColorCanvas", CV_WINDOW_AUTOSIZE);
81
82      // Show image in the name of the window
83      imshow("ColorCanvas", colorCanvas);
84
85      // Function for show the image in ms.
86      waitKey(0);
87
88      // Free memory
```

```
88      // Free memory
89      original01.release();
90      original02.release();
91      original03.release();
92      original04.release();
93      original05.release();
94      colorCanvas.release();
95      destroyAllWindows();
96      // End of the program
97      return 0;
98  }
```

Camshift

- MSHIFT(Mean Shift)
  - Busca el centro de masas de los puntos de interés en una región específica.
  - Se desplaza la región, centrándose en el nuevo centro de masas iterativamente.
- CAMSHIFT(**C**ontinuously **A**daptative **M**ean Shift)
  - Se utiliza para el seguimiento de un objeto en una secuencia de imágenes. Para ello, modifica el tamaño de la región de interés a lo largo de las iteraciones.
  - Emplea el canal H del HSV para segmentar el objeto.

Camshift

CamShift

Finds an object center, size, and orientation.

C++: RotatedRect **CamShift**(InputArray **probImage**, Rect& **window**, TermCriteria **criteria**)

Python: `cv2.` **CamShift**(probImage, window, criteria) → retval, window

C: int **cvCamShift**(const CvArr* **prob_image**, CvRect **window**, CvTermCriteria **criteria**, CvConnectedComp* **comp**, CvBox2D* **box**=NULL)

Python: `cv.` **CamShift**(prob_image, window, criteria) -> (int, comp, box)

Parameters:
- **probImage** – Back projection of the object histogram. See **calcBackProject()** .
- **window** – Initial search window.
- **criteria** – Stop criteria for the underlying **meanShift()** .

Returns: (in old interfaces) Number of iterations CAMSHIFT took to converge

The function implements the CAMSHIFT object tracking algorithm [Bradski98]. First, it finds an object center using **meanShift()** and then adjusts the window size and finds the optimal rotation. The function returns the rotated rectangle structure that includes the object position, size, and orientation. The next position of the search window can be obtained with `RotatedRect::boundingRect()` .

See the OpenCV sample `camshiftdemo.c` that tracks colored objects.

Camshift

- BackProjection
  - Con objeto de emplear camshift, es necesario disponer de un conjunto de puntos que puedan pertenecer a la región de interés a la que se le realiza el seguimiento.
  - Estos puntos se pueden obtener mediante Back Projection Histogram: Esta función crea una nueva imagen que contiene la probabilidad para cada píxel de pertenecer a la región de interés (objeto).

Camshift

calcBackProject

Calculates the back projection of a histogram.

C++: void **calcBackProject**(const Mat* **images**, int **nimages**, const int* **channels**, InputArray **hist**, OutputArray **backProject**, const float** **ranges**, double **scale**=1, bool **uniform**=true)

C++: void **calcBackProject**(const Mat* **images**, int **nimages**, const int* **channels**, const SparseMat& **hist**, OutputArray **backProject**, const float** **ranges**, double **scale**=1, bool **uniform**=true)

Parameters:

- **images** – Source arrays. They all should have the same depth, `CV_8U` or `CV_32F` , and the same size. Each of them can have an arbitrary number of channels.
- **nimages** – Number of source images.
- **channels** – The list of channels used to compute the back projection. The number of channels must match the histogram dimensionality. The first array channels are numerated from 0 to `images[0].channels()-1` , the second array channels are counted from `images[0].channels()` to `images[0].channels() + images[1].channels()-1`, and so on.
- **hist** – Input histogram that can be dense or sparse.
- **backProject** – Destination back projection array that is a single-channel array of the same size and depth as `images[0]` .
- **ranges** – Array of arrays of the histogram bin boundaries in each dimension. See **calcHist()** .
- **scale** – Optional scale factor for the output back projection.
- **uniform** – Flag indicating whether the histogram is uniform or not (see above).

The functions `calcBackProject` calculate the back project of the histogram. That is, similarly to `calcHist` , at each location `(x, y)` the function collects the values from the selected channels in the input images and finds the corresponding histogram bin. But instead of incrementing it, the function reads the bin value, scales it by `scale` , and stores in `backProject(x,y)` . In terms of statistics, the function computes probability of each element value in respect with the empirical probability distribution represented by the histogram. See how, for example, you can find and track a bright-colored object in a scene: