International Journal of Applied Engineering Research ISSN 0973-4562 Volume 11, Number 9 (2016) pp 6117-6120

Detection and Tracking over Image Pyramids using Lucas and Kanade Algorithm

Viacheslav Tarasenko
Department of Game and Multimedia Engineering,
PaiChai University, Daejeon, Republic of Korea.

Dong-Won Park*
Department of Game and Multimedia Engineering,
PaiChai University, Daejeon, Republic of Korea.

Abstract
Image registration finds a variety of application in computer vision, such as image matching for stereo vision, pattern recognition and motion analysis. This paper focuses on pyramid LK algorithm, which tracks starting from highest level of an image pyramid (lowest detail) and working down to lower levels (finer detail). Tracking over image pyramids allows track points of interest despite large disparity between photos.

Keywords: Lucas Kanade algorithm, Tracking and motion, Image pyramids, OpenCV library.

Introduction

General overview

In the field of computer vision, there are several methods to detect motion between images[1]. One kind of methods is the feature tracking. A feature, or a point of interest, is a point or a set of points where an algorithm can look and follow the motion through frames. There are several ways to select the features: based on brightness and colors or based on corners and edges detection. Depending on the algorithm that we choose, it will determinate in each way the features are selected[2].

To track features there are essentially two important steps. The first one is to decide which features to track, as we explained before, and the second one is the tracking in itself. One of the most known methods is the Lucas-Kanade feature tracking algorithm. We are going to change the order in the explanation, and we start explaining how the tracking works, and afterwards how the features are selected.

Problem Statement

Let's consider two gray-scaled sequential images, I and J. The two quantities $I(t) = I(x, y)$ and $J(t) = J(x, y)$ are then the gray scale value of the two images at the location $t = [x\ y]$, where *x* and *y* are the two pixel coordinates of a generic image point *t*. The image I sometimes referenced as the first image, and the image J as the second image[3].

For practical issues, the images I and J are discrete function (or arrays), and the upper left corner pixel coordinate vector is $[0\ 0]$. Let n_x and n_y be the width and height of the two images. Then the lower right pixel coordinate vector is $[n_{x-1}\ n_{y-1}]$.

The goal of this method is to find the location $v = u + d$ on the second image such as $J(u)$ and $I(u)$ are similar. The vector *d* is the image velocity at the point *u*. The next step is to choose a neighborhood where we can analyze the similarity between *u* and *v*. Thus, let's define ω_x and ω_y as two integers. We define the image velocity d as being the vector that minimizes the residual function:

$$\epsilon(d) = \epsilon(d_x, d_y) = \sum_{x=u_x-\omega_x}^{u_x+\omega_x} \sum_{y=u_y-\omega_y}^{u_y+\omega_y} (I(x,y) - J(x + d_x, y + d_y))^2$$

The residual function is applied to a window size of $(2\omega_x + 1) \times (2\omega_y + 1)$. This window is sometimes referred as integration window, and it has typical values between 2 and 7. To choose a good value we should consider two important aspects of any algorithm: accuracy and robustness. The accuracy component relates to the local sub-pixel accuracy attached to tracking. If we choose a small a value, we increase the accuracy, since it will consider only the nearest neighbors for the calculations, but it decreases the robustness, since the big motions will be ignored. The opposite problem occurs if we choose a big value for the integration window. In an idealistic situation, we should have $d_x \leq \omega_x$ and $d_y \leq \omega_y$, to cover all the possible motions. A solution for this problem is a pyramidal implementation of the classical Lucas Kanade algorithm[3].

Image Pyramids

Image pyramid representation

Considering an image I of size $n_x \times n_y$ and $I^0 = I$ as the "zero-th" level image. This image is essentially the highest resolution image (the raw image). The pyramid representation is then built in a recursive fashion: compute I^1 from I^0, then compute I^2 from I^1, and so on... In a general view, I^L is computed based on I^{L-1} where *L* is the pyramidal level. The recursive algorithm is then constructed as it follows:

$$I^L(x, y) = \frac{1}{4} I^{L-1}(2x, 2y) +$$
$$+\frac{1}{8}\left(I^{L-1}(2x-1,2y) + I^{L-1}(2x+1,2y) + I^{L-1}(2x,2y-1) + I^{L-1}(2x,2y+1)\right) +$$

$$+\frac{1}{16}\Big(I^{L-1}(2x-1,2y-1)+I^{L-1}(2x+1,2y+1)$$
$$+I^{L-1}(2x-1,2y-1)+I^{L-1}(2x+1,2y+1)\Big)$$

The value L_m is the height of the pyramid (picked heuristically). Practical values of I^L are 2,3,4. For typical image sizes, it makes no sense to go above a level 4. For example, for an image *I* of size 640x480, the images I^1, I^2, I^3 and I^4 are of respective sizes 320x240, 160x120, 80x60 and 40x30. Going beyond level 4 does not make much sense in most cases. The central motivation behind pyramidal representation[5] is to be able to handle large pixel motions (larger than the integration window sizes ω_x and ω_y). Therefore the pyramid height (L_m) should also be picked appropriately according to the maximum expected optical flow in the image.

Pyramidal Feature Tracking

Recall the goal of feature tracking: for a given point *u* in image I, find its corresponding location $v = u + d$ in image J, or alternatively find its pixel displacement vector *d*.

For $L = 0,\ldots,L_m$, devine $u^L = [u_x^L \;\; u_y^L]$, the corresponding coordinates of the point *u* on the pyramidal images I^L. Following our definition of the pyramid representation equations, the vectors u^L are computed as follows:

$u^L = \frac{u}{2^L}$.

The division operation is applied to both coordinates independently (so will be the multiplication operations appearing in subsequent equations). Observe that in particular, $u^0 = u$.

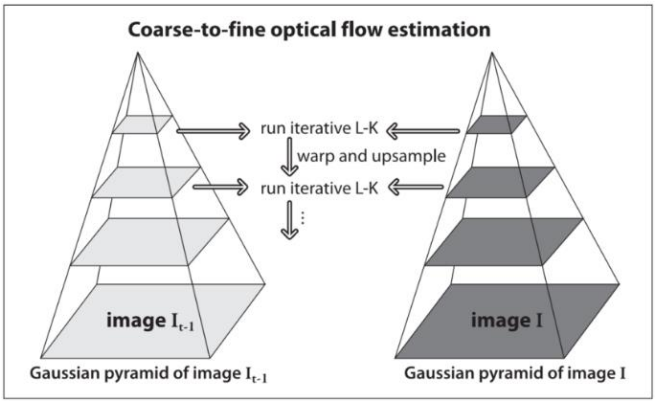

Figure 1: Pyramid Lucas Kanade optical flow: the motion estimate from the preceding level is taken as the starting point for estimating motion at the next layer down

The overall pyramidal tracking algorithm proceeds as follows: first, the optical flow is computed at the deepest pyramid level L_m. Then, the result of that computation is propagated to the upper level $L_m - 1$ in a form of an initial guess for the pixel displacement (at level $L_m - 1$). Given that initial guess, the refined optical flow is computed at level $L_m - 1$, and the result is propagated to level $L_m - 2$ and so on up to the level 0 (the original image).

The clear advantage of a pyramidal implementation is that each residual optical flow vector d^L can be kept very small while computing a large overall pixel displacement vector *d*. Assuming that each elementary optical flow computation step can handle pixel motions up to d_{max}, then the overall pixel motion that the pyramidal implementation can handle becomes

$d_{max\,final} = (2^{L_m+1} - 1) \cdot d_{max}$.

For example, for a pyramid depth of $L_m = 3$, this means a maximum pixel displacement gain of 15. This enables large pixel motions, while keeping the size of the integration window relatively small[6].

Lucas Kanade Method

Lucas Kanade algorithm background

The Lucas Kanade (LK) algorithm, as originally proposed in 1981, was an attempt to produce dense results[9]. Yet because the method is easily applied to a subset of the points in the input image, it has become an important sparse technique. The LK algorithm can be applied in a sparse context because it relies only on local information that is derived from some small window surrounding each of the points of interest. This is in contrast to the intrinsically global nature of the Horn and Schunck algorithm. The disadvantage of using small local windows in Lucas Kanade is that large motions can move points outside of the local window and thus become impossible for the algorithm to find. This problem led to development of the "pyramidal" LK algorithm, which tracks starting from highest level of an image pyramid (lowest detail) and working down to lower levels (finer detail). Tracking over image pyramids allows large motions to be caught by local windows[10].

General Idea behind LK algorithm

The basic idea of the LK algorithm rests on three assumptions.

i. *Brightness constancy*. A pixel from the image of an object in the scene does not change in appearance as it (possibly) moves from frame to frame. For grayscale images (LK can also be done in color), this means we assume that the brightness of a pixel does not change as it is tracked from frame to frame.

ii. *Temporal persistence or "small movements".* The image motion of a surface patch changes slowly in time. In practice, this means the temporal increments are fast enough relative to the scale of motion in the image that the object does not move much from frame to frame.

iii. *Spatial coherence*. Neighboring points in a scene belong to the same surface, have similar motion, and project to nearby points on the image plane.

We now look at how these assumptions, which are illustrated in Figure 2, lead us to an effective tracking algorithm. The first requirement, brightness constancy, is just the requirement that pixels in one tracked patch look the same over time:

$f(x, t) \equiv I(x(t), t) = I(x(t + dt), t + dt)$

That's simple enough, and it means that our tracked pixel intensity exhibits no change over time $\frac{\delta f(x)}{\delta t} = 0$.

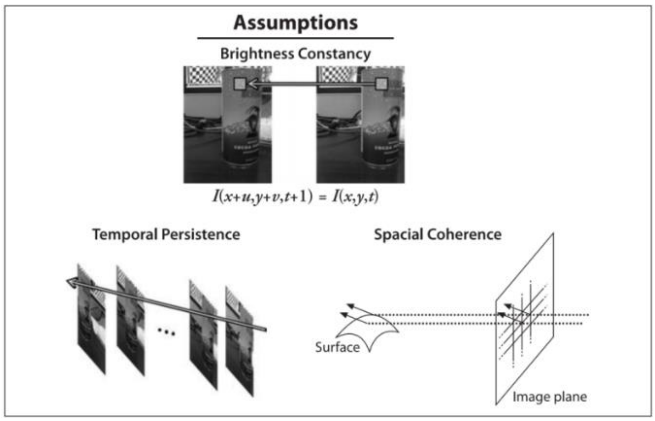

Figure 2: Assumptions behind Lucas-Kanade optical flow: for a patch being tracked on an object in a scene, the patch's brightness doesn't change; motion is slow relative to the frame rate; and neighboring points stay neighbors.

The second assumption, temporal persistence, essentially means that motions are small from frame to frame. In other words, we can view this change as approximating a derivative of the intensity with respect to time (i.e., we assert that the change between one frame and the next in a sequence is differentially small). When motion is detected with a small aperture, you often see only an edge, not a corner. But an edge alone is in sufficient to determine exactly how (i.e., in what direction) the entire object is moving.

We use last optical flow assumption for help. If a local patch of pixels moves coherently, then we can easily solve for the motion of the central pixel by using the surrounding pixels to set up a system of equations. Hence, the recommended technique is first to solve for optical flow at the top layer and then to use the resulting motion estimates as the starting point for the next layer down. We continue going down the pyramid in this manner until we reach the lowest level. Thus we minimize the violations of our motion assumptions and so can track faster and longer motions. This more elaborate function is known as pyramid Lucas Kanade optical flow.

Tracking features and results

Feature Selection

So far, we have described the tracking procedure that takes care of following a point *u* on an image I to another location *v* on another image J. However, we have not described means to select the point *u* on I in the first place. This step is called feature selection.

There are many kinds of local features that one can track[7,8]. It is worth taking a moment to consider what exactly constitutes such a feature. Obviously, if we pick a point on a large blank wall then it won't be easy to find that same point in the second image.

If all points on the wall are identical or even very similar, then we won't have much luck tracking that point in subsequent frames. On the other hand, if we choose a point that is unique then we have a pretty good chance of finding that point again. In practice, the point or feature we select should be unique, or nearly unique, and should be parameterizable in such a way that it can be compared to other points in another image. See Figure 3.

Returning to our intuition from the large blank wall, we might be tempted to look for points that have some significant change in them—for example, a strong derivative. It turns out that this is not enough, but it's a start. A point to which a strong derivative is associated may be on an edge of some kind, but it could look like all of the other points along the same edge.

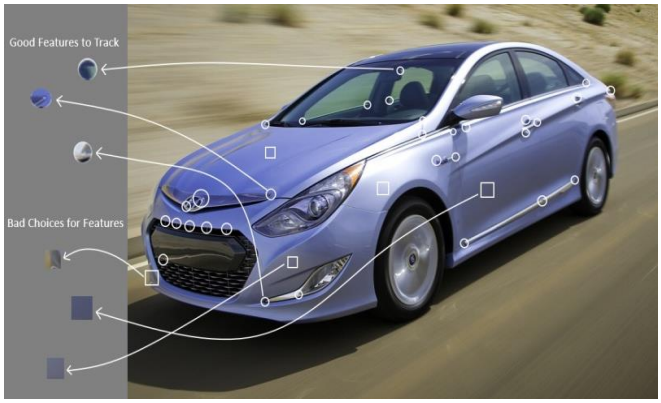

Figure 3: Good Features. The points in circles are good points to track, whereas those in boxes—even sharply defined edges—are poor choices.

However, if strong derivatives are observed in two orthogonal directions then we can hope that this point is more likely to be unique. For this reason, many track able features are called corners. Intuitively, corners—not edges—are the points that contain enough information to be picked out from one image to the next.

It is very intuitive to approach the problem of feature selection once the mathematical ground for tracking is led out. Indeed, the central step of tracking is the computation of the optical flow vector. At that step, the spatial gradient matrix G is required to be invertible, or in other words, the minimum eigenvalue of G must be large enough (larger than a threshold). This characterizes pixels that are "easy to track".

Therefore, the process of selection goes as follows[5,7,8]:

i. Compute the G matrix and its minimum eigenvalue λ_m at every pixel in the image I.
ii. Call λ_{max} the maximum value of λ_m over the whole image.
iii. Retain the image pixels that have λ_m value larger than a percentage of λ_{max} . This percentage can be 10% or 5%.
iv. From those pixels, retain the local max. pixels (a pixel is kept if its λ_m value is larger than that of any other pixel in its 3x3 neighborhood).
v. Keep the subset of those pixels so that the minimum distance between any pair of pixels is larger than a given threshold distance (e.g. 10 or 5 pixels).

After that process, the remaining pixels are typically "good to track". They are the selected feature points that are fed to the tracker.

The last step of the algorithm consisting of enforcing a minimum pair wise distance between pixels may be omitted if a very large number of points can be handled by the tracking engine. It all depends on the computational performances of the feature tracker.

Finally, it is important to observe that it is not necessary to take a very large integration window for feature selection (in order to compute the G matrix). In fact, a 3x3 window is sufficient $\boldsymbol{\omega_x = \omega_y = 1}$ for selection, and should be used. For tracking, this window size (3x3) is typically too small[11].

Subpixel Computation

It is absolutely essential to keep all computation at a subpixel accuracy level. It is therefore necessary to be able to compute image brightness values at locations between integer pixels.

In order to compute image brightness at subpixel locations we propose to use bilinear interpolation[12].

Let *L* be a generic pyramid level. Assume that we need the image value $\mathbf{I^L}(\mathbf{x}, \mathbf{y})$ where *x* and *y* are not integers. Let $\mathbf{x_0}$ and $\mathbf{y_0}$ be the integer parts of *x* and *y*(larger integers that are smaller than *x* and *y*). Let $\boldsymbol{\alpha_x}$ and $\boldsymbol{\alpha_y}$ be the two reminder values (between 0 and 1) such that:

$x = x_0 + \alpha_x,$

$y = y_0 + \alpha_x$

Then $I^L(x, y)$ may be computed by bilinear interpolation from the original image brightness values:

$$I^L(x, y) = (1-\alpha_x)(1-\alpha_y)I^L(x_0, y_0) + \alpha_x(1-\alpha_y)I^L(x_0+1, y_0) + +(1-\alpha_x)\alpha_y I^L(x_0, y_0+1) + \alpha_x\alpha_y I^L(x_0+1, y_0+1)$$

Sentinel framework is launching game application, sending messages to it, and waits for the game to produce output in log file. An obvious solution would be using threads. However, Python provides high-level entities for asynchronous communication, called greenlets. More specifically the framework is using gevent - a coroutine-based networking library that uses greenlets to provide high-level asynchronous API on top on libevent event loop.

Conclusion

Since the Lucas-Kanade algorithm was proposed in 1981, image alignment has become one of the most widely used technique s in computer vision. Applications range from optical flow, tracking and layered motion, to mosaic construction, medical image registration, and face coding.

Numerous algorithms have been proposed and a wide variety of extensions have been made to the original formulation. We presented an overview of image alignment, describing most of the algorithms and their extensions in a consistent framework. We concentrate on the pyramidal algorithm, an efficient algorithm that successfully used for tracking points of interest despite large disparity between photos.

Acknowledgements

The content in this paper includes the content of the master's thesis submitted at PaiChai University in 2014. The corresponding author is Dong-Won Park.

References

[1] Jan Erik Solem, Programming Computer Vision with Python: Tools and algorithms for analyzing images. O'Reilly (2012).

[2] J.R. Parker. Algorithms for Image Processing and Computer Vision. Wiley (2010).

[3] Bruce D. Lucas, Takeo Kanade, An Iterative Image Registration Technique with an Application to Stereo Vision. IJCAI (1981).

[4] Sang-Wook Lee, "A Image-based 3-D shape Reconstruction using Pyramidal Volume Intersection" Journal of information and communication convergence engineering 10(1), pp. 127-135, (2006).

[5] Jean-Yves Bouguet, Pyramidal Implementation of the Lucas Kanade Feature Tracker. Description of the algorithm. Intel Corporation (2001).

[6] Simon Baker, Iain Matthews, Lucas-Kanade 20 Years On: A Unifying Framework. Carnegie Mellon University Robotics Institute (2002).

[7] Jianbo Shi, Carlo Tomasi, Good Features to Track. IEEE Conference on Computer Vision and Pattern Recognition CVPR94 Seattle (1994).

[8] Carlo Tomasi, Takeo Kanade, Detection and Tracking of Point Features. Technical Report CMU-CS-91-132 (1991).

[9] Gary Bradski and Adrian Kaehler. Learning OpenCV: Computer Vision with the OpenCV Library. O'Reilly (2008).

[10] Alex Rav-Acha, Shmuel Peleg "Lucas Kanade without iterative warping". ICIP (2006).

[11] Hyo-Seok Seo and Oh-Young Kwon. "Accelerating the Retinex Algorithm with CUDA" Journal of information and communication convergence engineering 8(3), pp. 323-327, (2010).

[12] Daniel Lélis Baggio. Mastering OpenCV with Practical Computer Vision Projects. PACKT Publishing (2011).