Imaging with Spike Camera

Zeng Fu
School of Mathematical Sciences
Peking University
1900010644@pku.edu.cn

Liying Wang
School of EECS
Peking University
2000012958@stu.pku.edu.cn

Yuhan Xing
Yuanpei College
Peking University
2000017797@stu.pku.edu.cn

Songkun Zhan
School of EECS
Peking University
2000013139@stu.pku.edu.cn

Abstract

This project aims to calculate and generate images, which should be clear, stable and of high quality, from the given continuous spike data sequence. It is required that the output images have twice the resolution of the spike sequence. To obtain better result, we compared three methods: texture from latencies (TFL), texture from playback (TFP) and optical flow. We also tried a series of optimizations for the second and the third methods, respectively. Finally, the optical flow method yielded relatively ideal results.

1. Introduction

Spike camera captures light and accumulates the converted luminance intensity at each pixel. It records the pulses generated by incident lights at a speed of 20000-40000 frames per second. A spike is fired when the accumulated intensity exceeds the dispatch threshold. The spike stream generated by the camera indicates the luminance variation. Analyzing the patterns of the spike stream makes it possible to reconstruct the picture of any moment, which enables the playback of high speed movement. If we collect all the spikes fired at the same time and draw points according to the x- and y-addresses of the spikes on a clear screen, the scene structure and the object movement can be illustrated as time goes by. This is how the dynamic vision sensor (DVS) camera works.

However, the spike sequence only represents changes of luminance intensity in the temporal dimension, but can't reflect the difference of luminance intensity between a pixel and pixels around it (i.e., changes in the spacial dimension). Therefore, the texture is missing without analog-to-digital converter (ADC) data being transmitted along with the spike. Another deficiency of the DVS is that it has difficulty in processing still objects, because the luminance intensity does not change much if there is no object moving, leading to no spike to be fired. [1]

On the basis of previous work, we tried and optimized three methods – TFL, TPL and optical flow – to generate high-quality images from the given spike sequence. We read five related papers: "Spike Camera and Its Coding Methods" [1], "Reconstructing Clear Image for High-Speed Motion Scene with a Retina-Inspired Camera" [2], "Motion Estimation for Spike Camera Data Sequence via Spike Interval Analysis" [3], "Large Displacement Optical Flow: Descriptor Matching in Variational Motion Estimation" [4], "Kernel Regression for Image Processing and Reconstruction" [5]. Firstly, the spike sequence is interpolated to get images with a resolution of $500 \times 800 \times 4000$. Then, we adopt some methods, which we consider intuitive and viable, to process the images. After comparing the effects of these methods, we decide on the optical flow method. Based on the original version of flow, we make optimizations and obtain significant improvements.

The rest of the report is organized as follows. Section 2 introduces two basic methods. Section 3 presents some preliminary optimizations we try. Section 4 introduces optical flow and presents our optimizations. Section 5 shows experimental results and Section 6 concludes the projects.

2. Two basic methods

2.1. Texture from latencies (TFL)

The main idea of TFL is using the reciprocal of the interval between two adjacent spikes to represent the greyscale

of all points in the interval. Let $i_1, i_2, ..., i_n$ be the sequence of time then spikes are fired. Assuming $i_0 = 0$, then for each i such that $i_{k-1} < i \leqslant i_k$

Following the principle of TFL, the computed values are not perfectly matched with the original texture data, but the variation is in accordance with the original one. TFL can reconstruct the outline of the texture but not clear details, and there is much noise in the generated images. [1]

2.2. Texture from playback (TFP)

To reduce the noise in the images, we try to take advantage of the information from adjacent frames to denoise a certain frame. Given that the spikes are dispatched with a very high frequency, if we play back the spikes, the historical pictures are able to be illustrated. In TFP, there is a sliding time window collecting the spikes in a specific period. [1] The greyscale value at time i is computed as:

$$g(i) = \frac{1}{min(i + hwl, T_{max}) - max(1, i - hwl) + 1} \times \sum_{j=max(1,i-hwl)}^{m} in(i + hwl, T_{max}) f(j)$$

Thereinto, hwl is a variable parameter representing half the length of sliding window, and T_{max} is the time o fthe last frame.

3. Preliminary optimizations

3.1. Noise reduction for single frames

We notice that there are obvious noise points in not only the generated video, but also every single frame. Using the method in the paper [5], we denoise each frame respectively. Since the original data is binary, satisfactory results may not be acquired through direct denoising. We decide to first utilize TFL to convert the binary data into greyscale values, and do noise reduction for single frames based on this.

It it worth noting that the data-adapted kernel presented in reference paper has a considerable computational cost, but can't achieve significant improvements. Thus we make a compromise to adopt the non-adaptive kernel. To be specific, we adopt the 2-D Gaussian Kernel shown in Fig.1.

This method had a good effect on denoising within a single frame, but it is not the focus of this project. Under the task of video denoising, we expect to make the best of information from frames in adjacent times.

3.2. Weighting in the temporal dimension

The TFP mentioned in **Section 2.1** is relatively simple. For each frame in a fixed-length window before and after a certain frame, we give equal weights. But i reality, we

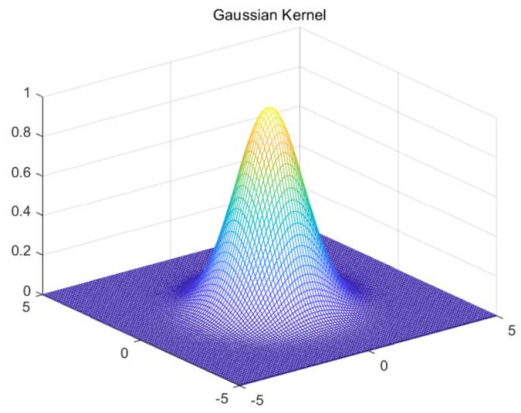

Figure 1. 2-D Gaussian Kernel

expect that the nearer a frame is from the target frame, the bigger role it plays in the estimation. In order to penalize the points which have a location offset from a particular time, and to reduce adverse effects they have on the estimation, we present a sliding window with weights on the temporal dimension on the basis of TFP. The kernel function we choose is 1-D Gaussian Kernel. The function *calcKer* in the code package is used to calculate 1-D Gaussian Kernel vector, while the function *weightSpikeMatrix* calls *calcKer* to generate appropriate kernel and do the weighted computation.

The sliding window with time weights plays a good denoising effect. Meanwhile, it effectively reduces the blur caused by equal weights. But essentially, it is still TFP. When the window is too long, it will face the same problems as the original TFP.

3.3. Iteration

Each method presented above has its own advantages and disadvantages: TFL does not average in the time dimension, resulting in sharp images. But noise in single frames and the whole video cannot be ignored. Original TFP or the one with time weights can generate videos that have smooth transition between adjacent frames, while the adverse effect is that the picture appears to be a certain degree of blur. To combine the advantages of each method, we do combinations and iterations of them. Experiments show that when using the iterative process as shown in Fig.2 and iterating 2 or 3 times, good effects can be achieved.

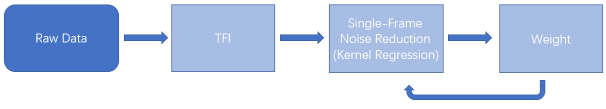

Figure 2. Iteration

3.4. 3-D Kernel regression

We try to refer to the method presented in the paper [5], directly extending the 2-D kernel regression to the 3-D form.

If we stack the frames in time order, then we can get a "3-D" image, as is shown in Fig.3. Intuitively, like in the 2-D form, this image has its corner, edge and flat areas, and we expect the 3-D kernel to contract at the local corner texture so as to avoid the interference caused by points of great differences, to elongate along the directions of the local edge texture, and to fully spread in the flat area to make use of as much information as possible to denoise.

This extension is mathematically achievable and we have written the corresponding code (the function *kernel_regression* in the code package). However, the computational cost of 3-D kernel is so huge that we have to give it up.

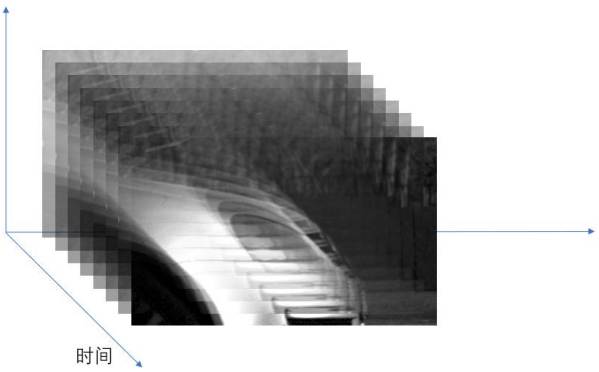

Figure 3. 3-D image

4. Optical flow and optimizations

Optical flow is an optimization algorithm based on the sliding window. As time goes by, objects in adjacent frames are displaced, so the same point corresponds to different physical objects in different frames. That's why sliding window results in blurry images.

We first calculate the displacement of objects in adjacent frames. According to the displacement vector, we can get the position of every point at a new time, and compute the weighted average in the time dimension. Below is the calculation formula, where $\vec{x} = (x, y)$ is the coordinate of the point, $\vec{v}(i, j, x)$ is the displacement variable. And the function *mex_of* in the code package gives the specific code.

$$g(i,\vec{x}) = \frac{1}{min(i+hwl, T_{max} - max(1, i-hwl) + 1} \times \sum_{j=max(1,i-hwl)}^{min(i+hwl,T_{max})} f(j, \vec{x} + \vec{v}(i,j,x))$$

4.1. Preprocessing of the video

When calculating the displacement, we find that if we use unprocessed videos, the displacement vectors are quite inaccurate. The reason is that the function *mex_of* has certain requirements on the quality of the video. Accordingly, we apply some methods to process the video before calculating the displacement vectors, then a more accurate result can be achieved.

4.2. Reduction of computational cost

Since the speed of most objects does not change within a few frames, there is no need to calculate the displacement vector for every frame. Rather, it will be enough to calculate the displacement vector over a certain time interval. Such simplification greatly reduces the calculation time, and the formula becomes:

$$g(i,\vec{x}) = \frac{1}{2 \times hwl + 1} \times$$

$$[\sum_{j=1}^{hwl} f(i+j, \vec{x} + \vec{v}(i, i+l, x) \times \frac{j}{l}) +$$

$$f(i-j, \vec{x} + \vec{v}(i, i-l, x) \times \frac{j}{l}) + f(i, \vec{x})]$$

4.3. Elimination of unreal outlines

We notice that although optical flow can make objects in motion clearer, there are some unreal black outlines at the edge of the objects. After discussing and researching, we conjecture that this is because the function *mex_of* wrongly estimates the displacement of points which are going to be blocked or are just exposed. Let's consider the following situation: An object was originally moving at a certain speed. When it is gradually blocked by some other object, the algorithm will still try to calculate an "appropriate" position for it and generate the new image as if there were no block. But this out to be erroneous. To verify our conjecture, we carry out the following experiment: We design three scenarios: (1) A white rectangle moving slowly against a black background (the corresponding code is *NoWall.m* in the code package); (2) A white rectangle moving slowly against a black background and being blocked by a grey wall (*WithWall.m*); (3) A white rectangle running through the picture, moving slowly against a black background and being blocked by a grey wall (*WithWallThrough.m*). Among the three scenarios, (3) is closer to the situation described above. Three phenomena are observed in our experiments: (a) The velocities of the edges of objects are roughly accurate without block; (b) The velocities of the interior of objects will slow down or even tend to 0; (c) From before being blocked to after, the velocities of the blocked parts significantly changed, but the

change values are uncertain. From these experiments, we find that the edges are often stationary for half of the time, and make some unreasonable displacements for the other half. To solve this problem, we write a function (the function *judge* in the file *fastflow.m*) to determine whether a pixel is blocked, and remove the blocked time period from the weighting. Now the formula is:

$$g(i,\vec{x}) = \frac{1}{2\times hwl+1}\times\{judge(\vec{v}(i,i+l,x),\vec{v}(i,i-l,x))$$
$$[\sum_{j=1}^{hwl} f(i+j,\vec{x}+\vec{v}(i,i+l,x)\times\frac{j}{l})+$$
$$f(i-j,\vec{x}+\vec{v}(i,i-l,x)\times\frac{j}{l})+f(i,\vec{x})]\}$$

judge is defined as follows:

$$judge(\vec{a},\vec{b}) = \begin{cases}(1,1)^T, a=b=\vec{0}, or\, a\neq\vec{0}, b\neq\vec{0}\\ (2,0)^T, a=\vec{0}, b\neq\vec{0}\\ (0,2)^T, a\neq\vec{0}, b=\vec{0}\end{cases}$$

5. Experimental results

Using the same video, we experiment with the methods mentioned. For TFP and its optimized version, we also tried different window lengths to compared the effects. Fig.4 – Fig.14 show the results and their comparison.

6. Conclusion

In this project, we try and compare several methods to generated high quality results from the given spike sequences. We also attempt to do a series of optimizations. Through many experiments and adjustments, we draw the conclusion that optical flow is the relatively optimal method. For further work, more efficient algorithms for 3-D kernel regression and optical flow need to be explored. Moreover, how to enhance the adaptability of these methods in different situations is worth studying.

7. Reference

[1] S. Dong, T. Huang and Y. Tian. "Spike Camera and Its Coding Methods," *Data Compression Conference (DCC),* pp. 437-437, 2017, doi: 10.1109/DCC.2017.69. [2] J. Zhao, R. Xiong, J. Xie, B. Shi, Z. Yu, W. Gao, and T. Huang. "Reconstructing Clear Image for High-Speed Motion Scene with a Retina-Inspired Spike Camera," in *IEEE Transactions on Computational Imaging,* vol. 8, pp. 12-27, 2022, doi: 10.1109/TCI.2021.3136446. [3] J. Zhao, R. Xiong, R. Zhao, J. Wang, S. Ma and T. Huang, "Motion Estimation for Spike Camera Data Sequence via Spike Interval Analysis," *2020 IEEE International Conference on Visual Communications and Image Processing (VCIP),* 2020, pp. 371-374, doi: 10.1109/VCIP49819.2020.9301840. [4] T. Brox and J. Malik, "Large Displacement Optical Flow: Descriptor Matching in Variational Motion Estimation," in *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 33, no. 3, pp. 500-513, March 2011, doi: 10.1109/TPAMI.2010.143. [5] H. Takeda, S. Farsiu and P. Milanfar, "Kernel Regression for Image Processing and Reconstruction," in *IEEE Transactions on Image Processing,* vol. 16, no. 2, pp. 349-366, Feb. 2007, doi: 10.1109/TIP.2006.888330.

Figure 4. TFL

Figure 5. TFP, window length = 9

Figure 6. TFP, window length = 17

Figure 7. TFP, window length = 65

Figure 8. 2-D kernel denoising

Figure 9. weighted TFP, window length = 9

Figure 10. weighted TFP, window length = 17

Figure 11. weighted TFP, window length = 65

Figure 12. Iteration twice

Figure 13. Flow, window length = 17

Figure 14. Flow, window length = 65