Pipe Line Filtering

Dmitry V. Yurin
Moscow State University, Faculty of Computational Mathematics and Cybernetics,
Lab. of Mathematical Methods of Image Processing.
yurin_d@inbox.ru

Abstract

A unified template C++ library is proposed for image filtering by local environment. The library provides fast and easy implementation of wide class of image processing procedures such as smoothing, sharpening, edge and corner detection, differential invariants calculation, image resampling, texture analysis etc. The resulting code is computationally effective, a large memory saving is obtained due to image processing line by line, no intermediate images are created. It makes easy to combine filters and transfer output of filters to input of others. The program code for complex filters directly reflects filtering algorithm graph structure. It is easy to adjust the system for using any image containers which support reading/writing operation line by line. The system is self adjusting and has self testing and debugging capabilities. The library proposed is especially useful for processing of large images.

Keywords: *Image processing, filtering, local environment, programming, memory saving, edge detection, feature points, texture descriptors, differential invariants, image resampling, graph, DAG balancing.*

1. INTRODUCTION

This paper is devoted to technical aspects of programming of image filtering. It is supposed that the class of filters under consideration satisfies the condition: each pixel value on output image(s) depends on pixels on input image(s) in the same relative coordinates and pixels in small environment of such coordinates. The term “relative coordinates” implies ratio x/width and y/height.

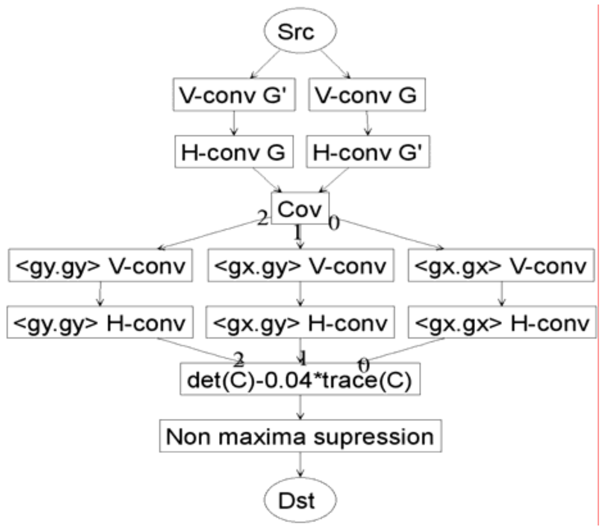

Figure 1: Harris feature detector composed of elementary filters.

A great number of frequently used procedures in image processing satisfy this condition, including image smoothing, sharpening, feature points and edge detection, texture analisys, image resampling (due to the term of relative coordinates introduced above), color spaces conversion, etc. [1].

When implementing filtering by local environment the following issues should be taken in consideration:

- ✓ the image pixel type may be different (8-16 bit unsigned integer, float, etc.), but it is desired to do some operations with certain types, for example, it is desired to perform convolutions with Gaussian and its derivatives kernels with float point types, but range filtering with integer types;
- ✓ the image should be extended over its bounds by size of local environment used by filter to produce output for all the image area (otherwise during sequential filtrations the image size will be diminished step by step); image extension is usually implemented via the image reflection relative to it’s bounds;
- ✓ a lot of filters are compound ones and the result of filtering is a some combination of simple or elementary filters results, for example, it is preferable to implement even a simple Gaussian smoothing (due to performance reasons) as a sequential vertical and horizontal convolution. More complex filters such as [2] edge detection, feature points detection (Figure 1.) and differential invariants calculations [3], especially in scale space [4,5], have a very complex structure composed of elementary filters.

The issues listed above result in temporary image creation for each intermediate result – for extended size images, for conversion pixel type, for results of elementary filters which should be combined and filtered again. This results in allocation of many image containers in memory or hard disk per each image in process. It is unacceptable when the image is large (typical image size for modern digital cameras is 30 Mb, and 0.1-10 Gb in satellite imaging).

2. SYSTEM STRUCTURE

The base principles of the proposed system are:

- avoid creation of intermediate images;
- save memory if possible;
- program code must reflect filter structure as closely as possible (see Figure 1.);
- easy filter design and combining;
- develop code that can be reused as frequently as possible;
- not significant program deceleration;
- auto adjusting;
- debugging features.

The system proposed is intended for realization of image processing algorithms which can be described as graph

$G=(V,E)$ like in (Figure 1). The vertexes or nodes $u\in V$ of graph G are filters (rectangles or ellipses in Figure 1), the arrows or directed edges $e\in E: e=(u,v)$ represent images which are the result of work of filter $u\in V$ (arrow tail) and are the input for filter $v\in V$ (arrow head). The arrows do not come from/in filter directly, but come from filter's output ports (`Out`), and come in input ports (`In`).

Each filter $v\in V$ can contain arbitrary number of input and output ports denoted as $v.|Out|$ and $v.|In|$ respectively. The set of all filter ports is denoted as $v.\{Out\}$ and $v.\{In\}$. There are special types of filters – source (S) and target or destination (T), shown as ellipses in Figure 1, these filters contain 0 input or output ports respectively. Each input port must be connected to one and only one filter output port. Each output port of any filter may be connected to arbitrary non zero number of some filters input ports.

2.1 System Overview

The proposed template C++ library consists of base class FilterBase and a lot of its derivatives, class System, which provides framework adjustment and operation, and some other classes. Each class Filter includes as a member zero or more classes InPort and OutPort, through which image reading and writing are performed. Complex filtering schemes are realized by connection of InPort to OutPort terminals of Filter classes. Such filters can be implemented as classes derived from FilterComplex. Each InPort object does not contain any pixel data, but only the requirements on input data (the environment size) and a pointer to OutPort object to which this InPort object is connected to. Class OutPort contains class StripBuffer, where a portion of image (a few image rows) is located. This is the only place where intermediate images data exist. As many as required InPort objects can be connected to the OutPort object and get access to the same data in the StripBuffer object (see section 2.2).

Input for framework described is object(s) derived from class FilterBase with 0 InPort and >0 OutPort members (Source filters). Output(s) are similiar objects containing 0 OutPort and >0 InPort member objects (Target or Destination filters).

Typical program code contains the following mandatory fragments:

Filters creation, including source, destination elementary and/or complex filters; generally, elementary filters are multipurpose, their customizations are fulfilled by proper functors:

```
Filter1<float> f1(..filter parameters..);
MyFunctor fun(..functor parameters..);
Filter2<float,MyFunctor> f2(fun);
```

Filters connection according to desired filtering scheme (graph), for example, connection of filter f1 OutPort 2 to filter f3 InPort 0 looks as:

```
f1.out(2).ConnectTo(&f3.in(0));
```

System engine creation and initialization:

```
FiltersSystem sys("test.dot");
sys.Assign(&src,&dst);
```

The last step is filtering in essence:

```
sys.Run();
```

It should be mentioned that there may be more than one source and one target filters, for example RGB images may be realized as 3 grayscale sources/targets or by one source (target) with 3 OutPorts (InPorts). In this case function Assign() receives iterators to STL collections of sources and targets. Moreover, source images can be generated line by line virtually (for example, noise input image) and target(s) may not be an image, for example, it may be a list of feature points detected.

2.2 InPort OutPort and StripBuffer

Each class InPort contains members: a pointer to OutPort class the InPort class is connected to and local environment size Env required by the filter on this input. Structure Env contains 4 integer members: l(eft), r(ight), b(efore) f(orward) – the number of pixels from the central pixel required to calculate filter output.

Each OutPort class contains a list of pointers to InPort classes the OutPort class is connected to and a StripBuffer class where the results of filtering on OutPort channel are written to.

Class StripBuffer is organized as 1D array, where grayscale image strip lines are placed line by line. To add string to buffer it is necessary to request iterator to image line which is currently processing (call beginUpdate() function), write data to this iterator and than call endUpdate() function. When calling beginUpdate(), the buffer contents are shifted by line via memmove() function to erase data not required for further processing and to get free space for receiving a new image line. When calling endUpdate() function, for each pixel in this line the buffer calls post processing functor assigned, reflects this line ends to the left by Env.l and to the right by Env.r pixels and, if required (at first and last lines of the whole virtual image), makes copies of some lines from buffer to buffer to support image extension beyond the top bound (by Env.b pixels) and beyond the bottom bound (by Env.f pixels). After begin/end update transaction is finished, image strip in buffer becomes accessible for reading for InPort's connected to OutPort, where this buffer is located. Access to image data is performed by requesting reference to image object, which contains pointer inside buffer and image info. Point (0,0) corresponds to the left end of line being currently processed, the available image portion from this line is described by Env structure, filter can safely read pixels in requested environment (including first and last current line pixels) they are certainly in buffer.

It should be mentioned that after image string y is placed to buffer, the string (y-Env.f) is ready for reading. That is each buffer produce a ***delay*** in pipelining. Otherwise, each filter one time should process input and output lines with the same relative numbers in the images. In other words, the value y/height is ***invariant*** for all the filter's inputs and outputs at each time – by system construction. It is always forced by system that Env.f ≥ Env.b to make all delays constant during filtering. The problems of delay balancing are solved automatically and the algorithms used are considered in the next section.

3. SYSTEM OPERATION

The Assign() method of the System object performs the following to properly initialize the framework:

1) check that all the targets can be distinguished from sources;
2) check the absence of loops in the filter graph;
3) save graph diagram (if necessary) to hard disk in DiGraph format;
4) check that all the sources can be distinguished from targets in back arrows direction;
5) check the absence of not connected InPorts/OutPorts;
6) propagate image size from sources through graph of filters and accumulate requests from filters connected to each OutPort on required environment size and meet all the requests;
7) if there are filters with more than one InPort object in the graph, then adjust the delays produced by buffers via increasing some buffers to balance directed acyclic graph (DAG) of filtering system.

The first 3 items are performed by topological sorting [6]. This procedure is based on depth first search (DFS):

```
$G \leftarrow \varnothing$ , $A \leftarrow \varnothing$
foreach $u \in S$
        DFS_Visit( $u$ );
Save .dot file if required
```

Starting sequentially from all sources, the graph is built by recovering all graph nodes (filters) including targets. Each node processing finishes (BLACK) in reversed topological order and the node is inserted on the top of the nodes list of graph G. If there are cycles in graph, DFS find back edges (Lemma 22.11) and item 2 is performed simultaneously. The DFS_Visit algorithm is modified (lines 2-6) to remove duplicated edges, because filter graph can be a multigraph. One filter may produce more than one image and transfer more than one image to another filter, that is there are more than one edge from one node to another.

```
DFS_Visit(FilterBase* $u$ )
1     $A \leftarrow A \cup u$  //white node found, make gray
2     $Q \leftarrow \varnothing$   //empty queue
3     foreach $v \in u.\{Out\}$
5             if $v \notin Q$  //remove multiple edges
6                     $Q \leftarrow Q \cup v$
7     foreach $v \in Q$   /investigate /edge (u,v)
8             if $v \in A \wedge v \in G$
9                     Back edge found, inform user!
10            else
11                    DFS_Visit(v)
12    G.push_front(u);  //finish processing u
```

Now all nodes of graph G are collected and initialization steps 1-3 are accomplished. To perform steps 4-5 it is sufficient to investigate that all inputs and outputs of each filter (nodes in G) are connected to nodes that are already in G, otherwise inform user what filter and port is bad and refer to the saved digraph (debugging property).

Then initialization step 6 is: in topologically sorted order (TS) process each node of G. In this order for each node virtual images sizes on input are known, and it is possible to calculate output images sizes. At these points the possibility is given to the filter to calculate and set the required environment size if it depends on images sizes.

The last initialization step is adjusting of delays to fulfill invariance restrictions (see the end of section 2.2). This task can be performed by increasing delays via increasing `Env`.f for some branches in filtering (pipelining) graph. For this purpose the initial buffer assignment algorithms from [7] have been adopted.

```
foreach $u \in V$  in TS order
 if  $u \in S$ d[u] ← 0;
 else calcDelay(u)
foreach $u \in V$  in TS order
 initStripBuffers(u)
```

Where funcvtion **`calcDelay(u)`** is:

```
foreach  $v \in u.\{In\}$
 w ← d[v]+u.In.Env.f/v.Out.H
 d[u] ← max(d[u],w)
foreach  $v \in u.\{In\}$
 w ← u.In.Env.f/v.Out.H
 b[(v,u.In)] ← d[u]-d[v]-w
```

and function **`initStripBuffers(u)`** is

```
env ← {0,0,0,0}
foreach  $v.In \in u.\{Out\}$
 en ← v.In.Env
 f← u.Out.H*b[(u,v.In)]
 en.f← round(en.f+f+0.5)
 env ← max(env,en);
 u.Out.initStripBuffer(env);
```

The Run() method of the System object finds out the filter that has all input ports data ready and all output ports ready to receive next string and executes it. By construction destination filters are always ready to receive.

4. CONCLUSION

The library developed demonstrates high performance and very large memory saving which allows to process huge images. The filter developing becomes easy and fast.

5. ACKNOWLEDGEMENTS

The research is done under financial support of RFBR, grants 06-01-00789-a, 08-07-00469-a.

6. REFERENCES

[1] Pratt, W.,K., 2001. Digital Image Processing, John Wiley & Sons, Inc., New York, 3d edition..

[2] Canny, J., 1986. A computational approach to edge detection. In IEEE Trans. PAMI, V. 8. P. 34—43.

[3] Schmid, C., Mohr, R., 1997. Local grayvalue invariants for image Retrieval. In IEEE Trans. PAMI, V. 19, No. 5, P. 530—534.

[4] Dufournaud, Y., Schmid, C., Horaud, R., 2000. Matching images with different resolutions. In In Proc. CVPR, V. 1, P. 612—618.

[5] Mikolajczyk, K., Schmid, C., 2004. Scale & Affine Invariant Interest Point Detectors. In International Journal of Computer Vision. V. 60, No. 1, P. 63—86.

[6] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C., 2002. Introduction to Algorithms. The MIT Press. Cambridge, Massachusetts, London, England, 2nd edition.

[7] Chatterjee M., Banerjee S., Pradhan D.,K., 2000. Buffer Assignment Algorithms on Data Driven ASICs In IEEE Trans. on computers. V. 49, No.1. P.16—32.

About the authors

Dmitry V. Yurin, PhD, is a senior scientist at Institute of Computing for Physics and Technology and at Laboratory of Mathematical Methods of Image Processing, Chair of Mathematical Physics, Faculty of Computational Mathematics and Cybernetics, Moscow Lomonosov State University. His contact email is yurin_d@inbox.ru