Modular parallelization framework for multi-stream video

Tim Lenertz

ULB, LISA department  
Avenue Franklin D. Roosevelt 50  
1050 Brussels, Belgium  
+32 494 92 70 47

tim.lenertz@ulb.ac.be

2nd Author

2nd author's affiliation  
1st line of address  
2nd line of address  
Telephone number, incl. country code

2nd E-mail

3rd Author

3rd author's affiliation  
1st line of address  
2nd line of address  
Telephone number, incl. country code

3rd E-mail

SUBMITTED to ACM MULTIMEDIA 2016 OPEN SOURCE SOFTWARE COMPETITION

**ABSTRACT**

A flow-based software framework specialized for 3D and video is presented, which in particular handles automatic parallelization of multi-stream video processing. The workflow is decomposed into a set of filter units which become nodes in a graph. Execution with support for time windows on node inputs is handled by the framework, as well as low-level manipulation of multi-dimensional data.

**CCS Concepts**

• **Computer systems organization~Data flow architectures**     
• *Computing methodologies~Image manipulation*     
• Computing methodologies~Computer vision     
• *Software and its engineering~Middleware*

**Keywords**

video; view synthesis; FTV; parallelization; multi-stream; 3D

# INTRODUCTION

In video and image processing, algorithms are often developed in an ad hoc manner. An existing solution is used as test bed, and incremental improvements are made to it in order to improve the visual quality of the output.

This is for instance the case in FTV[[1]](#footnote-1), where the goal is to synthesize virtual-viewpoint images of a scene that is captured by multiple real cameras and possibly depth sensors.

This leads to lack of a software architecture which would encapsulate different parts of the view synthesis process, making it difficult to extend a method to use cases beyond stereo video, and to optimize the algorithms.

For this reason, a software framework, currently named *mf*, is developed which provides a generic implementation of elements common to FTV-related algorithms, including a data flow structure into which *filter* modules that process video frames are inserted. A major feature is the support for automatic parallelization of the workflow, and partial activation of multi-stream input.

Figure 1 shows an example of such a flow graph of an FTV algorithm which will be elaborated in section 4. The diamond-shaped nodes are filters implemented as independent modules.

/Users/timlenertz/Desktop/demograph.pdf

Figure . Flow graph of demonstration FTV algorithm

# GOALS

One aim in the design of the framework is to clearly separate between incrementally developed *experimental* code, and the *accurate* code of the generic, reusable framework components.

It should also be *modular*, in the sense that external code, which does not reference any of the framework functionality, can easily be fitted into a plug-in and used as a node in the flow graph.

The framework tries to remain as versatile as possible, and provide a greatest common denominator of the functionality required for FTV applications. For example, *filter* nodes may have many inputs which get dynamically activated or deactivated, and may receive bidirectional time windows of their inputs. For example the “blender” on the example in Figure 1 activates its input branches in function of a virtual camera position. Enough versatility should be provided to make it possible for complex algorithms to be decomposed into the graph structure.

Parallelization is handled by the framework itself, both on an *inner* (code used by the filters) and *outer* (coordination of nodes) level.

Also, the framework is written in modern C++, using techniques like RAII[[2]](#footnote-2) and static polymorphism, and is extensively tested.

# FEATURES

The framework is currently built in a bottom-up approach with the goal of getting the required components to develop a FTV view synthesis method. It consists of low-level *core* components for multi-dimensional data manipulation and queuing, the *flow graph system* which built on top of this, and additionally a *media toolkit*, which is a loose collection of useful components for 3D and video applications.

## N-d array

The data processed in 3D and video algorithms often has the form of *n*-dimensional homogeneous arrays, whereby types can be scalars, vectors or tuples and may be null-able. For instance, a masked image is a 2D array of null-able RGB 3-vectors, and a point cloud is a 1D array of XYZRGB tuples. A strided *n*-d array type is provided at the core level of the framework, with the basic functionality like slicing, sectioning and iteration.

Copy-less type casts exist which make is possible for example to pass an ndarray<1,xyzrgb> object to a function which expects an ndarray<1,xyz>, or even an ndarray<2,float>, without copying data or breaking the encapsulation, or unnecessarily templatizing the function.

A wrapper masked\_elem<T> exists which adds a binary mask flag and makes the type null-able (unless it already is so), and it can also be cast away similarly. Having this support at the core level removes difficulties in application code caused by differing conventions for marking null values. For example, separate binary mask images or special “background color” values.

Type erasure of lower dimensions is supported: For example components which work on whole video frames may safely receive video data in the form of an ndarray<1,frame> instead of ndarray<3,rgb>. Code complexity is reduced, but the framework still assures type safety and proper memory alignment.

Moreover, a kind of reinterpret\_cast operation is provided which allows for in-place processing on an ndarray with different input and output element types.

## Queues

On top of the type-erased ndarrays, FIFO ring buffer classes are implemented. Features include absolute time indices on frames, and dual-thread support (one reader and one writer) with mutual waiting for readable/writable frames and deadlock prevention.

Two variants exist, a *seekable* ring buffer where the reader may seek to another absolute time in the stream and the writer responds, and a *non-seekable* ring buffer where the stream duration does not need to be known at construction and the writer marks the end after it has written the last frame.

The circular buffer wrap-around is implemented using the operating system's virtual memory mapping functionality. No special handling is required for views that cross the buffer’s boundary.

The flow graph system makes use of these classes to implement the data flow with seekable files, as well as non-seekable real-time sources.

## Flow graph

The flow graph is a tree structure through which frames of the media stream that is being processed flow. All data in the flow graph has an *n*-d array format, described by a dimension, element type, and frame shape[[3]](#footnote-3). Nodes in the graph perform a concrete frame-wise media processing step. The flow graph system handles the coordination of multiple nodes. Lock-less parallelization and time synchronization and are handled at the framework level, independently of the node implementation.

The flow graph always contains one *sink* node, and one or multiple *source* nodes. Frames are *pulled* from the sink, and recursively from the preceding nodes up to the sources. Nodes may have more than one input, and (currently) always have one output. Inputs are connected to outputs of other nodes in a one-by-one manner, the graph can have no loops.

Each node contains a *filter* which does the concrete frame processing and may be implemented as a module external to the framework.

### Features

The format of a frame data is described by a *n*-d array dimension, element type, and shape. Input(s) and output of nodes can each have different formats. Time is represented as an integral index value. The nodes operate frame-wise and invariably write one frame to their output at each step.

However, nodes can receive a *bidirectional time window* on their inputs: On construction, the node specifies, for each input, the number of past and future frames that is needs to receive, in addition to the current frame. The system assures that nodes receive this time window for each frame. It gets truncated only near the beginning and end of the stream. Previous frames are retained, and no unnecessary copies are made. This allows for example implementation of a node which applies an image kernel filter over both time and space, on a 3D *n*-d array.

Nodes may *activate* and *deactivate* their inputs at runtime. No frames are pulled from graph branches connected to deactivated inputs. When an input is reactivated after having been deactivated, the intermediary frames are skipped. If the source node at the end of the branch is seekable, a *seek* request gets propagated towards it, and intermediary frames are never loaded or processed. With this it is possible to implement nodes with a large number of inputs, out of which only a small number is active at a time, in a scalable manner.

Finally, inter-node parallelism is supported with *asynchronous nodes*. These nodes have the same features as *synchronous nodes*, but instead of processing frames when pulled from the output, they run a separate thread and independently process frames in advance. Pull requests wait until the requested frames become available. For example, while the sink is processing frame *t*, a preceding asynchronous node may be processing *t*+ *k* at the same time. The maximal number of frames an asynchronous node may be in advance of last the frame pulled from its output is called its *prefetch duration* and can also be configured at runtime.

Any *filter* can be run on a synchronous or asynchronous node, and this can be specified at runtime. Additional node types may be added in the future. (Such as one that processes multiple frames in parallel.)

Nodes can also have parameters which are either set to a constant value, to a function of time, or to mirror a parameter of another node.

### Interface

Concrete filter classes implement three operations:

**Process**. Receives *n*-d array views to input and output data. Must write one frame to the output. For source filter, sets a flag when this was last frame in stream.

**Pre-process**. Called prior to *process*, with views not yet available. At this stage, the filter can activate or deactivate inputs.

**Setup**. Called at initialization. Preceding nodes (connected on inputs) are already set-up, and the filter can set the output frame shape in function of the input frame shapes.

Prior to *setup*, typically at construction, the filter creates its inputs and outputs, and statically defines their dimension and element type. It may create a variable number of inputs. A C++ template architecture is set up in a way that the filter receives its views casted into this format at compile-time, without having to handle any ndarray casting operations itself. The underlying nodes and queues operate solely on type-erased frames.

## Media toolkit

Currently the media toolkit mainly consists of an encapsulation of extrinsic and intrinsic camera matrices: In a *space object* hierarchy 3D objects have poses relative to each other, and classes for pin-hole camera models are implemented which support several depth projection conventions.

Also included are an interface to *OpenCV* for image processing tasks, generic *n*-dimensional kernel filter functions, and point cloud support.

# DEMONSTRATION PROGRAM

As stated the goal of free-viewpoint video (FTV) is to synthesize virtual-viewpoint images of a scene that is captured by multiple real cameras and possibly depth sensors. This typically involves a reconstruction step where scene geometry is estimated from the captured images, followed by a view synthesis step which generates the virtual view.

Numerous approaches to this have been developed [1][2][3], and a call for evidence for new FTV technologies was published by MPEG in 2015 [4]. MPEG uses the programs DERS[[4]](#footnote-4) and VSRS[[5]](#footnote-5) as reference software evaluate the results of new algorithms. They use an algorithm based on stereo matching and image warping.

A small demonstration program was written using the framework which generates a video with a moving virtual camera based on the *Poznan Blocks* sequence [5] (see Figure 2). The input data consists of 9 fixed position camera feeds, along with depth sensor data. The depth images are of low quality and have blurred edges and other artifacts.

The algorithm replicates the one employed by VSRS in its most basic form, except that now more than two input views are taken, and the flow graph system is used. The goal of the demonstration program is not to reach high-quality output (much better FTV results exist already), but to have a program which is written from scratch and uses almost all of the framework’s features.



Figure . Demonstration program output

## Algorithm

Figure 1 shown earlier depicts the flow graph and the filters that this demonstration program consists of. The *main* function of the program only constructs and runs the graph, all the concrete processing steps are implemented as filter modules. Note that the “blender” node has 9 inputs, only two of the branches are shown on the figure.

In two camera images depicting the same scene from different viewpoints, points from one *source* image can be mapped to points from the other *destination* image knowing its depth in either image, and the homography matrix which can be calculated from the camera parameters. The process of copying each pixel in an image into its corresponding location on a new destination image is called *image warping*, and is an elementary method for view synthesis.

The “warp” filter performs a pixel-wise forward warping of the input depth maps to the virtual camera view. The used 4×4 homographic matrices also include the depth reprojection, and are obtained using the camera model classes from the media toolkit.  
No low level matrix operations are performed by the filter code. Figure 3 shows a frame of one of the depth image inputs after the warping operation. Virtual shadows appear in areas that were occluded in the *source* image viewpoint, and small holes appear due to the limited resolution of pixel-wise warping. The missing values are represented using the masked\_elem wrapper.

The “depth filter” then applies a median kernel filter on the holes of the resultant depth map. The “reverse warp” filter performs a pixel-wise reverse warping of the input image into the virtual camera, taking the *destination* depth map as input. It also uses pixel splatting to fill in small holes.

Its outputs on the 9 branches are images with the same virtual viewpoint, taken from the different source views and hence with different occlusions. Figure 4 shows the output of one of the “reverse warp” nodes. The “blender” node calculates a pixel-wise mean of the images, and discards masked pixels in any input image.

It uses only 3 of its 9 inputs at each frame. The inputs whose real camera positions are closest to the virtual camera position (which is varied in time) are activated. The other inputs are deactivated. The mean is weighted in function of the camera’s distances to the virtual camera.

The “reverse warp” nodes are set to be *asynchronous*. Because of this the program gets automatically parallelized into 4 threads, with three separate threads for the branches connected to the active blender inputs. The point of synchronization of two threads is restricted to the ring buffer between two nodes. When an input is reactivated after having been inactive, the branches need to jump to the new time index. This *seek* operation is implicitly propagated through the branch, up to the source nodes which change the file position. Intermediary frames are never loaded or processed by the “warp” or “depth filter” nodes.

The “result filter” uses a past and future time window, and runs a 3D kernel filter (temporal and on the image space) to further remove small holes. For example, flickering pixels are removed that way. Finally, the “sink” writes the output to a video file.

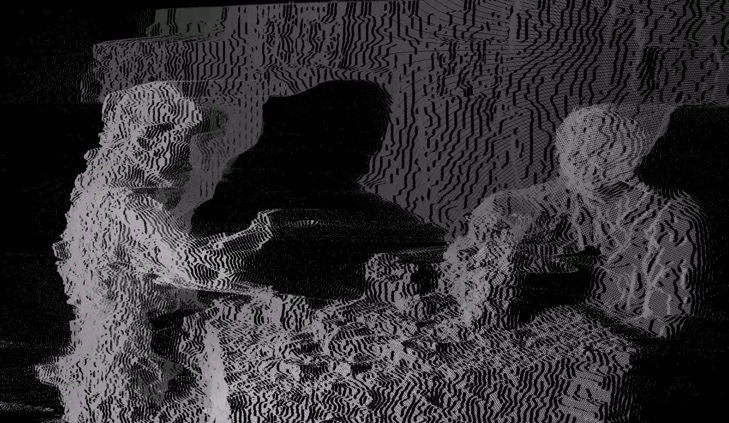


Figure 3. Depth image after warping



Figure 4. Image after reverse warping

# USAGE

The current version of the framework is available for download at <http://timlenertz.github.io/mf/>. The site also contains source code documentation and build instructions.

The framework currently only supports Linux and Darwin (OS X). Some low-level components (virtual memory mapping, thread synchronization) depend on the operating system and are currently only implemented for these targets. A compiler with full support for C++14 is required, and the framework is developed and tested on *LLVM/Clang* 3.8. Standard GNU *Makefiles* are used for building.

# SIMILAR SYSTEMS

*RabbitFire* [6] is similar data-flow framework with the primary purpose of handling parallelization at the framework level, and it is also intended for video processing. It reaches very high efficiency by using lockless thread synchronization, also supports intra-node parallelism and input time windows. However, it currently does not support asynchronous nodes, and runs only on Windows.

*Intel Thread Building Blocks* [8] is a more general-purpose solution for data flow graphs. It is based on a notion of message passing instead of a stream of frames.

*GStreamer* [7] is an open-source modular data-flow system with heterogeneous media formats. It is mostly intended for conversion and playback of different video and audio formats, and not specific for frame-wise processing.

On the Apple platform, *Quartz Composer* is a graph-based system which is integrated into the operating system, and mainly used for visual GUI effects and screensavers. It is not primarily intended for use with custom application-specific plugins.

# FUTURE WORK

Support for nodes with multiple outputs was removed because of complications with input activation and deactivation. An alternative idea is to implement a *multiplexer* node class which does not process data, but only copies it to its outputs, and handles the time synchronization issues.

A *thin node* class is in development which does not allocate an own buffer, but instead does in-place processing. It is an optimization which allows trivial pixel-wise operations like color conversions to be performed more efficiently and without wasting memory. Such nodes could then be inserted implicitly into the graph when connecting inputs and outputs with incompatible frame types. (For example, one which converts color formats.)

Another possible addition are *parallel nodes*, which process multiple frames at the same time, providing for intra-node parallelism. An adjustment of the underlying ring buffer system is needed for this. Parallel nodes would be stateless, and their *process* procedures would no longer be called in sequential order. Possibly *OpenCL* support can also be integrated at the framework level.

*Shared data* which can be accessed by multiple nodes, regardless of their current time, and with appropriate mutex protection, could for instance be useful for scene reconstruction methods where multiple nodes continuously refine a scene representation.

A plugin system may be developed where filters written in external code can be loaded dynamically at runtime. The graph and the parameters of its nodes and filters could then be described in an external file which is parsed at runtime without any recompilation.

# REFERENCES

1. Lee, C., Tabatabai A. and Tashiro K. 2015. Free viewpoint video (FVV) survey and future research direction. *APSIPA Transactions on Signal and Information Processing.* 15, 4 (Oct 2015).
2. Smolic A. 2011. 3D video and free viewpoint video - From capture to display. *Pattern Recognition*. 44 (2011), 1958-1968.
3. Slabaugh G., Culbertson B., Malzbender T. and Schafer R. 2001. A Survey of Methods for Volumetric Scene Reconstruction from Photographs. *International Workshop on Volume Graphics*. (Jun 2001)
4. MPEG. Call for Evidence of Free-Viewpoint Television: Super-Multiview and Free Navigation – update. *ISO, Coding of Moving Pictures and Audio. N15733*. (Oct 2015)
5. Marek Domański, Adrian Dziembowski, Agnieszka Kuehn, Maciej Kurc, Adam Łuczak, Dawid Mieloch, Jakub Siast, Olgierd Stankiewicz, Krzysztof Wegner. Poznan Blocks - a multiview video test sequence and camera parameters for Free Viewpoint Television. *ISO/IEC JTC1/SC29/WG11 MPEG 2014/M32243*. San Jose, USA, (Jan 2014)
6. NEXTWave Technologies SPRL. RabbitFire™. <http://www.nextwave-techs.com/rabbitfire.html>
7. GStreamer Team. GStreamer. <https://gstreamer.freedesktop.org>
8. Intel®. Thread Building Blocks. <https://www.threadingbuildingblocks.org>

1. Free-viewpoint video [↑](#footnote-ref-1)
2. Resource acquisition is initialization [↑](#footnote-ref-2)
3. e.g. for 2D images, its width and height [↑](#footnote-ref-3)
4. Depth Estimation Reference Software [↑](#footnote-ref-4)
5. View Synthesis Reference Software [↑](#footnote-ref-5)