# Proscenium: A Framework for Spatio-Temporal Video Editing

Eric P. Bennett
University of North Carolina
at Chapel Hill
bennett@cs.unc.edu

Leonard McMillan
University of North Carolina
at Chapel Hill
mcmillan@cs.unc.edu

## ABSTRACT
We present an approach to video editing where movie sequences are treated as spatio-temporal volumes that can be sheered and warped under user control. This simple capability enables new video editing operations that support complex postproduction modifications, such as object removal and/or changes in camera motion. Our methods do not rely on complicated and error-prone image analysis or computer vision methods. Moreover, they facilitate an editing approach to video that is similar to standard image-editing tasks. Central to our system is a movie representation framework called Proscenium that supports efficient queries and operations on spatio-temporal volumes while maintaining the original source content. We have adopted a graph-based lazy-evaluation model in order to support interactive visualizations, complex data modifications, and efficient processing of large spatio-temporal volumes.

## Categories and Subject Descriptors
H.5.1[**Information Interfaces and Presentation**]: Multimedia Information Systems–*video.* I.3.4[**Computer Graphics**]:Graphics Utilities–*graphics editors.* I.3.6[**Computer Graphics**]: Methodology and Techniques–*graphics data structures and data types.*

## General Terms
Design, Management, Performance

## Keywords
Video editing, multimedia framework, feature selection, feature removal, video layers, video stabilization, special effects.

## 1. INTRODUCTION
The recent introduction and rapid adoption of consumer digital video camcorders has redefined the landscape for video editing tools. We see many parallels between the ongoing evolution of digital video editing systems and the image editing systems that arose following the introduction of digital photography. Prior to digital photography there was relatively little editing of film prints per se, beyond the crops afforded by scissors and the occasional zoom provided by photo lab enlargements. Whereas today, even the most naïve user of digital photography commonly crops, resizes, adjusts the contrast of, and varies the brightness of their images. Moreover, sophisticated image processing tools, such as layer segmentation, clone brushing, and unsharp masking are increasingly being applied by recreational photographers.


*MM'03*, November 2-8, 2003, Berkeley, California, USA.


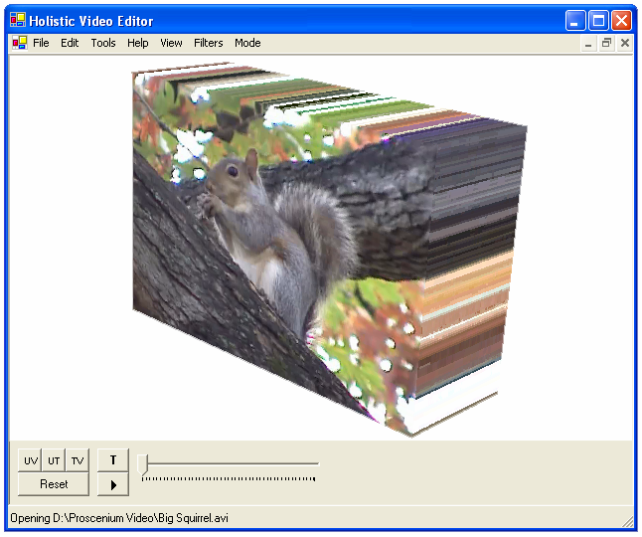


**Figure 1:** Screenshot from our application depicting an 8 second video segment as a spatio-temporal volume. Our editing system provides a user interface for sheering, modifying, restoring, and resheering such volumes.

As with digital photography, the first generation video editing tools facilitate the most simple and common of editing tasks–specifically, the cutting and pasting of video segments interspersed with transitions and titles. As the marketplace for consumer video editing evolves we believe that a new generation of tools will be developed to provide video manipulation capabilities ranging from simple touchups to advanced post-production special effects like those seen in big-budget Hollywood films.

In this paper we present Proscenium, a new video-editing paradigm that, rather than processing video clips one frame at a time, treats the entire video sequence as a spatio-temporal volume as shown in Figure 1. We have developed a new set of user-guided video editing tools that enable complex manipulations of entire video sequences using simple and intuitive user interactions. Furthermore, we have developed a movie representation framework that supports efficient queries and operations on spatio-temporal volumes while always maintaining the original fidelity of the source content. We have adopted a graph-based deferred execution model in order to support interactive manipulation and processing across these volumes.

## 2. PREVIOUS WORK

Most modern digital video editing systems are adaptations of classical analog video and film editing systems. Their primary function is to cut raw footage into a series of clips, and then assemble these clips along some timeline into a produced video. The introduction of digital processing capability and content has enabled a wide range of new inter-sequence transition effects as well as a rich set of intra-frame processing capabilities to improve image contrast and color-balance. However, the process of editing, assembling, and producing a video still follows a similar pipeline to the classical approaches pioneered for analog video.

Systems like Adobe Premiere [3], Apple Final Cut Pro [5], and Avid Media Composer [7] all adopt the classical film editing approach. They support the assembly of short clips by allowing cuts, cropping, editing, and rearranging of short video segments. Alternatively, Adobe After Effects [1] looks at making modifications primarily on only short pieces of footage. Our spatio-temporal video-editing framework allows both kinds of edits, but it is particularly well suited for manipulating and combining elements from many short video segments. We also do away with the idea of a timeline, replacing it with a more visual 3D video rendering.

The idea of filter graphs has been previously used in many forms in the area of multimedia processing [19]. Conceptually, the idea is that individual processing components can be ordered and arranged so that data flows from the input of a filter graph to the output; passing through the interconnected components that lie along that path. Each component (often called a filter) may modify data before passing it along to the next filter. The data flows in the form of buffers that comprise the pixel data of one or more frames. Each filter is only responsible to process data in a standardized manner without knowledge of exactly what filters are directly connected to it. This allows them to be easily interconnected in any order, similar to the Decorator design pattern specification [13].

Apple's QuickTime [6] framework and Microsoft's DirectShow [17] framework implement multimedia filter graphs for video and audio. Specifically, DirectShow treats video data as a stream that flows in buffers of entire frames of pixels from the graph's input to the graph's output. The Berkeley Continuous Media Toolkit [16] also implements a powerful filter graph in the form of a scripting protocol. Proscenium also features filter graphs, but extends the idea to include bi-directional data flow. Proscenium is also designed to be dynamically configurable for ease in combining and manipulating filters as specified by the user.

There have also been notable efforts towards providing 3D editing systems with interfaces analogous to standard 2D image editors. Ideas from Adobe Photoshop [2], a commercially popular image-editing tool, are, in fact, frequently applied to both 3D and video applications because of its ease of use, flexibility, and rich set of tools. Recently systems have been proposed to extend the same rich image-editing environment to 3D volumetric [8] and point-based models [23]. We also provide a 3D extension to the classical image-editing approach, but our work focuses on a particular class of 3D data called spatio-temporal volumes, which is our working abstraction for video sequences.

The concept of displaying video data as a three dimensional volume was explored by Fels et al [12] and extended by Klein et al [14][15]. Their work demonstrated the wide range of visualizations that could be achieved by allowing slices to be taken out of videos when treated as spatio-temporal volumes. Their primary image manipulation tool was a "cutting plane". Proscenium expands the range of processing that can be applied to spatio-temporal volumes, in particular we provide the ability to distort or warp the data volume in order to facilitate object alignment and editing operations.

Video stabilization is a key component in Proscenium's editing process. Buhler et al [10] demonstrated how foreground and background stabilization could be used to generate novel videos with refined camera and object motions. Their work relied on extensive off line analysis for tracking features and combining images from a source sequence. Proscenium, on the other hand, depends on user provided guidance to aid in the stabilization process. This helps to overcome many of the problems associated with automatic techniques such as when features are occluded, or if the source images lack sufficient detail for robust tracking. Furthermore, our user aided approach allows us to separately stabilize different visual elements of the scene. Thus, through a series of successive edits we can modify each element.

Recent work in video matting has also influenced the design of our system. Chaung et al [11] presented a combination of user-guided and automatic techniques for constructing garbage mattes and trimaps. Their method relied on the use of optical flow methods for maintaining temporal coherence. If the optical flow did not yield a sufficient result, new user specified mattes could be substituted at any frame. They also discuss their method of background estimation by choosing the nearest temporally aligned pixel to replace a foreground pixel that avoids many parallax issues. Proscenium implements similar functionality, but relies on user-guided stabilization followed by a statistical analysis of the aligned region to establish background and foreground mattes. Proscenium uses a very similar background extraction technique in its edge-filling filter, but integrates it with stabilization for non-static camera applications.

The overall workflow of Proscenium was influenced by the layer concepts of Wang and Adelson [21]. They developed the notion that general planes of motion in a video should be edited independently. Proscenium edits use video stabilization to make one particular layer static through time, and therefore easier to modify, before changes are applied.

## 3. SPATIO-TEMPORAL VIDEO EDITING

The goal of spatio-temporal video editing is to enable new interactive tools for manipulating video with similar flexibility and ease of use as current 2D image editing applications. This is accomplished using all of the frames of the original video footage. Achieving interactive response over such large volumes of source material requires special design considerations and support infrastructure.

Making precise image edits to videos is complicated by the fact that users are very sensitive to temporal artifacts even as small as a single pixel. Maintaining temporal consistency is therefore a common problem in video editing. If temporal consistency is not properly handled when dealing with object removal or background replacement, the viewer can become aware of "ghost-like" outlines in areas where the video was modified. A video editing system must therefore support operators that are aware of

both their local spatial (intra-frame) and temporal (inter-frame) contexts. This further motivates our requirement that the whole video "volume" should be processed simultaneously rather a frame at a time.

In standard image editing systems, graphic artists are provided with a refined set of tools and techniques that are both subtle and powerful. Most of these tools are very straightforward, and do not rely on complex analysis algorithms or otherwise rely on a great deal of automation. The real power of editing tools comes from the human in the loop who chooses where and when tools are applied as well as resolves ambiguities when necessary. However, the straightforward application of traditional image-editing techniques to video sequences would be tedious, particularly if applied one frame at a time. To the extent that the frames are similar, it is conceivable that edits applied to one frame could be propagated to the remainder, thus, better leveraging the editing efforts. Therefore, in order to allow a more traditional editing approach, we provide tools to locally align specific image regions, thus making them more similar to each other before editing begins. In order to support local alignments it is often necessary to significantly skew and distort nearby frames of the video volume. However, this global modification of each frame allows us to keep the area being edited static through time. This permits edits to be applied simultaneously to many frames.

We also adopt a strategy of dynamically combining simple editing tools to form more complex tools. Many video editing packages provide specialized tools to accomplish very specific tasks. Learning the subtleties of these tools can be time-consuming. Spatio-temporal video editing proposes a more flexible and interactive solution with fewer base level tools. In addition, we provide the capability for combining a series of editing actions to compose more advanced functions. These resulting tools are neither fully automated nor completely ignorant, but allow the artist the maximum of flexibility while having the computer provide feedback to the user and handle the busy work.

In the interest of flexibility, we place no constraints on the type of source footage that can be edited. Certain edits are greatly simplified when the source footage has specific characteristics (such as extracting a background from a static camera shot, or stabilizing a segment with a fixed in-frame subject, or constructing a panorama from a rotating camera). However, our simple tools for manipulating spatio-temporal volumes can be composed and then reapplied to accomplish all of these tasks, even when the source material is far from ideal.

The large volumes of data required to process each edit when working with dynamically warped uncompressed video can be overwhelming, so lazy evaluation (also referred to as deferred execution) is used at every step of our editing process. Until the changes are committed, all apparent modifications made to a video do not modify the source materials. Instead, each operation is represented as filter in a process graph. Visualization of editing operations executes a series of filter calls that map each output pixel to the set of source pixels that determine its value and perform the desired combinations. Since it is expensive to maintain copies of the original and modified volumes, lazy evaluation and judicious caching are employed to provide interactivity. This need to maintain accurate mapping functions at each step of the process necessitates accurate sampling and reconstruction in our spatio-temporal video editing framework. We provide support for source image interpolation and filtering that is transparent to the end user.

## 3.1 Data Representation

Each source video segment in our system is conceptually represented as a three dimensional array addressed in spatial dimensions by $u$ and $v$ and in time by frame number, $t$. We provide both discrete and continuous access to the spatio-temporal volume through separate methods. Discrete addressing is akin to standard array access, whereas continuous access allows for fractional addressing and implies interpolation. The quality of interpolation is a property of the volume, and can be established by either specific (Nearest neighbor, Bilinear, Bicubic, etc.) or generic (Low quality, High quality, etc.) hints. Source image filtering (minifying access) occurs at a higher level, and can make use of either access method.

We also provide the capability to dynamically re-map the parameterization of the video model using a general 2D projective transform defined for each value of $t$. These mapping functions allow a wide range of spatial modifications including frame-to frame translations, rotations, scales, skews, and any combination of these. The ability to dynamically re-map the frame-to-frame parameterizations enables our capability to align local regions of the volume while leaving the source images in place. These projective transformations are invertible, thus allowing the mapping from either source to destination or vice versa. In this paper we refer to application volume queries using parameters $(x,y,t)$ and source video parameters as $(u,v,t)$, where $u$ and $v$ are projective functions of $x, y$, indexed by $t$ as described.

Currently all spatio-temporal volumes use a common color space for storing pixels: unsigned bytes with RGBA (0-255) values. This is encapsulated in a Color class. The alpha component serves as either a traditional blending element [18] or as a pixel-specific auxiliary variable for Boolean or more complex operations. Importantly, (0,0,0,0) is reserved as being an "empty" value, meaning that nothing was found at a requested pixel.

## 3.2 Filter Graphs

The underlying objects that make lazy evaluation possible are the filters in the filter graph. In the Proscenium system, each of these filters is referred to as a PFilter.

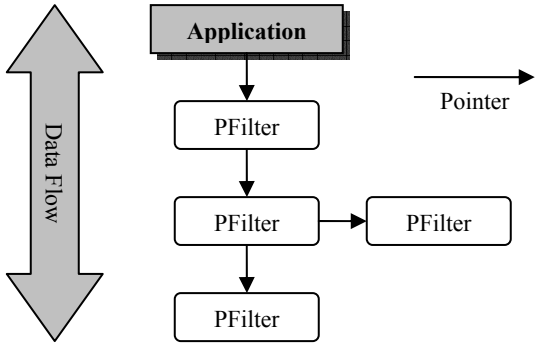


**Figure 2:** Diagram illustrating the interconnections of PFilters in a Proscenium filter graph. Each filter is only aware of those filters that comprise its inputs.

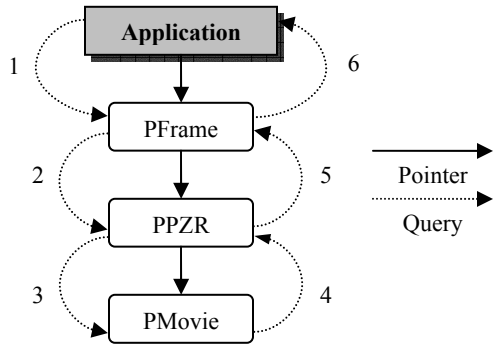


**Figure 3:** The flow diagram above illustrates the flow of pixel queries through an example filter graph. Data requests by the application traverse the graph until the holder of the data is reached, and then the results are passed back up the chain, possibly being modified along the way. This graph stabilizes a video and then crops its edges, creating the effect of a new camera motion by combining filters.

Proscenium's filter graph is specifically tuned for lazy evaluation. Whereas many implementations [6][17] treat pixel data as large buffers, Proscenium never processes with anything larger than a single pixel at a time. Proscenium's graph is fully bi-directional, instead of being a traditional directed acyclic graph that forces data to flow from the input of the system to the output. Bi-directionality allows the application at the output of the graph to request only the pixels it needs for interactive display. The filter graph is also designed to insert and remove PFilters at run-time based on user interaction without rebuilding the entire graph.

All actions begin with a request from the application that it wants a pixel at some (x,y,t). This request is sent to the output of the filter graph and not the input, which decides what actions to take. The most basic action is to say nothing was found and return an "empty" pixel. It can also return a constant, such as a background color. Finally, it can request pixel data from any of the filters at its inputs, modify that color, and return it to the requestor. This ordering of events is crucial to achieve the functionality in the filters described later. There is the added performance bonus that the final PFilter (the first queried) may be able to return a value without querying the other PFilters, because it foregoes the trouble of having to traverse through the entire graph.

All requests are initiated with a coordinate using either a discrete or continuous address as previously described. By asking for pixel color values one at a time, there is no need to define an internal sampling standard, because all sampling is handled at the application level.

## 3.3 PFilter Specification

Bi-directional data flow across PFilters is enforced by having each fulfill the requirements of a basic interface. They must be able to describe their size in width, height, and number of frames. They must internally know if they are a discrete or continuous filter. Most importantly, they must be able to handle reading and writing of individual pixels, which is done by defining input filters and filter functions. If not overridden the defaults return the values obtained from their input PFilters.

The pixelWidth and pixelHeight are the measurements in pixels of the viewable area of each frame. Proscenium currently assumes that these are constant across all frames of a sequence, so smaller images must be padded with "empty" pixels. The number

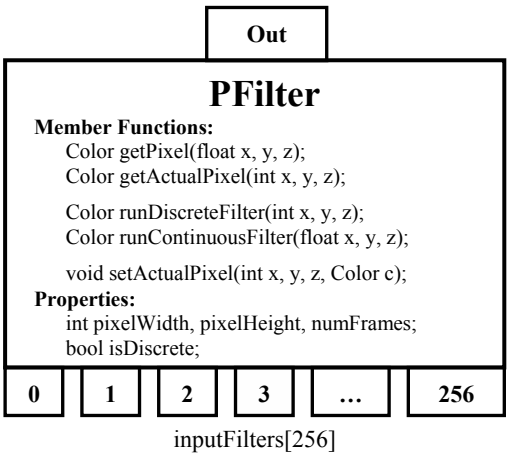


**Figure 4:** Each PFilter object supports multiple input ports and a single output. By combining the results from different input filters, video sequences can be modulated and combined. The methods shown comprise the standard interface for all PFilters.

of frames (numFrames) is a discrete quantity that makes the assumption that the frame rate is constant, but variable rates can be added through transitional PFilters.

Pixel values are queried with the discrete getActualPixel(int x,y,z) function or the continuous getPixel(float x,y,z). These functions only know the requested pixel coordinate and return the color at that pixel; nothing else. Unless these functions are overridden they pass along the query to the private functions runDiscreteFilter(int x,y,z) and runContinuousFilter(float x,y,z) after providing bounds checking. These are the most commonly overridden functions when creating a new filter. If a PFilter is defined as discrete (by bool isDiscrete) a call to the continuous getPixel will use tri-linear interpolation to resolve the color.

Inside each PFilter is an array called inputFilters[] which associates a PFilter pointer with an integer index. The filter graph is constructed by setting these values. By convention, the PFilter associated with the number zero is its main data path, carrying the video output of the filters below, while the others are designed for specific purposes for each filter type. For data flow reasons PFilters only know what PFilters are their inputs and do not know their outputs. Many PFilters may share the output of a PFilter, but only one PFilter can be assigned to each input of a PFilter.

# 4. SPATIO-TEMPORAL OPERATORS

The best way to understand how spatio-temporal video editing works is by looking at its supporting components. These example PFilters represent only a small subset of the types of filters that have been designed. However, interesting permutations are possible with just a few.

Most PFilters can be categorized into a few useful groups. First are those PFilters that alter the color value of the pixel that was passed into it through its input, such as for color or contrast adjustment. Next are those filters that pull their value from pixels that may not be the pixel directly aligned with it in its input PFilter. The PBackground filter described in this section performs an analysis of all of the pixels in the movie that share a particular x and y coordinate. Finally are those filters that alter the overall shape of the individual frames as they pass through the PFilter, such as with video stabilization and video framing.

## 4.1 Simple Color Correction

To demonstrate a simple discrete filter, PCorrect adjusts the blue values of its input pixel and sends it to the output with the red, green, and alpha unchanged. The isDiscrete variable is set to true so that all processing will pass through the runDiscreteFilter virtual function.

```
class PCorrect : public PFilter
{
public:
     PCorrect() { isDiscrete = true; blueAdjust = 20; }

     Color runDiscreteFilter(int x, int y, int z)
     {
          Color inColor = inputFilters[0]->getActualPixel(x,y,z);
          int tempBlue = inColor.B + blueAdjust;
          if(tempBlue > 255)tempBlue = 255;
          if(tempBlue < 0)tempBlue = 0;
          return Color::FromArgb(inColor.A,
                    inColor.R,inColor.G,tempBlue);
     }
     int blueAdjust = 0;
};
```

This PFilter has no internal storage, so its changes must be propagated down to its input. In the process, the color correction must be run in reverse, so that when it progresses through the filter in the forward direction at a later time, it will be filtered, and the desired color will result again. This is easily accomplished by overriding another function:

```
void setActualPixel(int x, int y, int z, Color newColor)
{
     int tempBlue = newColor.B - blueAdjust;
     if(tempBlue > 255) tempBlue = 255;
     if(tempBlue < 0) tempBlue = 0;
     Color alteredColor = Color::FromArgb(newColor.A, newColor.R,
          newColor.G, tempBlue);
     inputFilters[0]->setActualPixel(x, y, z, alteredColor);
}
```

## 4.2 Video Framing

With PFrame, the true extent of a movie can be hidden by manipulating the PFilter's pixelWidth, pixelHeight, and numFrames properties. This PFilter serves two purposes. First, it acts as a zoom. Videos are sampled during interaction, meaning that removing data on the edges allows greater detail to be shown for the remaining portion. If the PFrame is left in place it will continue to act as a crop, but if it is removed all of the occluded data on the sides is still present.

This effect is achieved by substituting new values for pixelWidth, pixelHeight, and numFrames. The filter then becomes responsible for handling the fact that the origin may no longer be at (0,0,0). Finally, it must reverse this operation when a pixel is written so that the write it transmits to its input PFilter will be the original coordinate and not the offset coordinate.

To simplify this example, only the x-dimension will be framed, but the same technique applies to all three dimensions. myOffset is the horizontal distance from the origin that the frame begins, and myWidth is its horizontal size.

```
Color runDiscreteFilter(int x, int y, int z)
{

     if ((x >= 0) && (x < myWidth))
          return inputFilters[0] > getActualPixel(x + offX, y, z);
     else return Color::FromArgb(0,0,0,0);        // Empty Pixel
}

void setActualPixel(int x, int y, int z, Color newColor)
{    inputFilters[0]->setActualPixel(x + offX, y, z, newColor); }
```

The use of the "empty" pixel is important, as it indicates the presence of an area outside the video. Bounds checking is explicitly done here because it is crucial to make sure occluded pixels cannot pass through.

## 4.3 Background Restoration

The PBackground filter returns a color based on a function of the matching (x,y) coordinates in every frame. Therefore, if this function were solved for every (x,y) pair in the video a new image of the background would result. In this example, the median of each color channel combined into a new color is the statistic used for estimating the background color. Alternatively, background filters are also possible that select the modes of the color component distributions. Furthermore, in this background filter implementation, those pixels with an alpha below some threshold are not considered.

The runDiscreteFilter function disregards the z value, and takes the mode of the pixels with the matching (x,y) and returns that color. No setActualPixel method is provided because this output is not directly related to any frame's input pixel.

```
Color runDiscreteFilter(int x, int y, int z)
{
     Color tempColor, finalColor;
     for(int i=0;i<inputFilters[0]->numFrames;i++) {
          tempColor = inputFilters[0]->getActualPixel(x,y,i);
          if(tempColor.A == 255)
                Sort the Red, Green, and Blue values into buckets
     }
     return Color::FromArgb(255, Median of Red, Blue, and Green);
}
```

Another background restoration filter has been designed called the edge-filling filter which returns the nearest opaque pixel that is temporally aligned to a transparent pixel. This technique is particularly useful in videos with translating or rotating cameras when attempting to build a panorama.

## 4.4 Caching

Performing the operations of a large number of interconnected PFilters can become very compute intensive. At some point caching becomes a convenient method to speed up operations. The PCache class is a configurable cache that complements our model of lazy evaluation. When a PCache is added onto the end of a filter graph it does not immediately cache all the pixels. Instead, it waits to be queried about a pixel before retrieving its value from its input PFilter. This accelerates subsequent accesses.

A cache exists as a block of RGBA or monochrome pixels exactly the same size as the PCache's input video. Each frame is separately stored as a bitmap, and pointers to each frame are stored in an array for quick access. This allows frames to be deleted or inserted without regenerating the entire data structure.

The cache is initially filled with an arbitrary reserved value, which is referred to as the "unsolved" color. When the cache receives a getActualPixel it checks its personal data structure, and upon finding the "unsolved" value at that coordinate, it queries its input. It takes the return of that function, updates its own data structure, and then returns it back up the filter graph. A "locked" PCache is one that will never perform a lookup even upon finding the "unsolved" color.

Our PCache functions have the following policies. First is that changes made by setActualPixel affect the data in the PCache, but are not propagated to its input terminals. Therefore, when a PCache receives an incoming setActualPixel request, its internal data structure is modified so that subsequent calls to the PCache return the new value. There is also a difficulty in alerting PCaches that they are invalidated by upstream changes to their inputs. Because all requests for pixel data flow in the opposite direction, there is no direct way for a PFilter to notify later PCaches that its

data changed. They will either go on unaware of the inconsistency or rely on the application to invalidate a portion of the PCache back to its unsolved state.

The PCaches have usefulness outside of accelerating the main data path in the filter graph. Of the example filters so far, none have used more than the first entry in inputFilters[]. PCaches placed on one of the other PFilter inputs can be an efficient way for that PFilter to cache reoccurring operations. For example, the PBackground filter returns the same value for each frame with the only variation depending on the (x,y) value requested. Therefore, by creating a locked PCache with only one frame, the solved pixel values can be stored in an efficient external data structure. This PCache then also acts as a complete image of the background.

## 4.5 Video Files

In order to process video, the filter graph must at some point contain the raw source video footage. The raw data is provided as yet another PFilter derived class. The source data class, called PMovie, is derived from the PCache class. It is essentially an unchanged PCache that defaults to being in the "locked" state, and therefore causes no input data lookups. It also adds member functions to load source video and bitmaps files into the frames.

For output to video files on disk, the mechanism actually is another PFilter called PAVIOut. It is placed on the end of the filter graph where the application would normally make its queries so that it uses the same data that the user sees on screen. This filter configures a PCache, and then initiates calls to query all pixels in the volume. This structure is then fed into the video encoder, which, in turn, produces the output file.

## 4.6 Video Stabilization

The ability to stabilize an object is a key tool in the spatio-temporal video editing repertoire. Almost all editing operations use stabilization in one way or another. In order to visualize what stabilization does, first imagine the spatio-temporal video cube. Suppose this video has an object in it translating from right to left. In order to stabilize this object, each frame must be shifted by an increasingly larger amount to the right in order to have the object fall in the same general pixel area. This results in sheering the volume. However, the moving object now falls in a static rectangular box in the volume, making it easier to select or edit.

Video stabilization is not limited to translating objects. By applying projective transforms it is also possible to remove the effects of scaling, rotation, shearing and changes in planar perspective. This can cause the entire volume to take on many interesting shapes, but the area under stabilization will appear to remain constant through time.

In a normal workflow, an object is first stabilized. If needed, subsequent "fine-tuning" stabilizations can be applied, and their transforms composed. It is then possible to edit or modify the stabilized objects throughout the warped portion of the volume. Finally the stabilization can be removed. These edits can then be propagated to the original source material. This allows what would otherwise be very complex edits over transforming objects to be done simultaneously over many frames.

Imagine having a moving object in a scene that you want to re-color. You could stabilize the object so that the orientation and shape of the object did not change. Then you could use a "temporal paintbrush" which works like a normal paint brush in a 2D paint application but makes changes to many adjacent frames. Because the stabilized area is consistent in all frames (independent of its shape and orientation in the original footage) any edit would immediately have temporal continuity when the stabilization was removed.

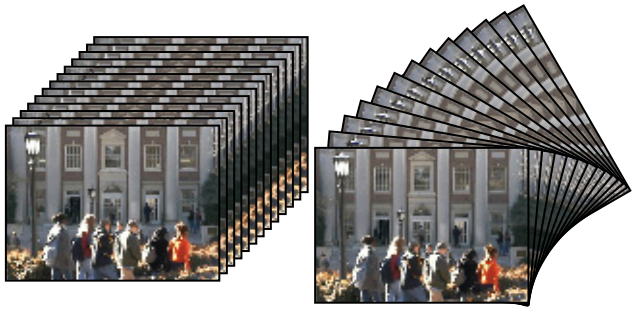

**Figure 5:** Proscenium supports arbitrary projective warps and sheers of the volume. This facility enables objects moving within the field of view to be stabilized.

The Proscenium video stabilization system involves two major components: a mechanism for determining the transform matrices of the frames in relation to each other and a filter supporting bi-directional data flow in the filter graph for editing and viewing of the video in its stabilized form.

For Proscenium, the stabilizations are performed using projective transforms, which are established via user-defined inter-frame correspondences. It is not necessary to specify a correspondence in every frame of a video segment. Instead, the user can select corresponding points at appropriate intervals in the video, and those points are linearly interpolated in the intervening frames. When only a single correspondence is specified, frame-to-frame translations are computed. When two correspondences are provided a similitude transformation is found (a translation and rotation with a global scale). With three corresponding points affine transformations of the images are computed. Four correspondences specify a unique projective transformation. When more than 4 correspondences are provided the closest projective transformation (in the least squares sense) is computed. All of the coefficients for the transforms can be found using linear methods as described in [22]. We solve for these systems in real-time using Gauss-Jordan Elimination [20]. By convention, the first frame of a movie will remain stationary, with all other frames generating their correspondences in relation to that frame.

Once the correspondence matrices have been calculated, pixel access must be considered. As mentioned previously, Proscenium is designed to only know a frame's width and its height, and it handles non-conforming frames by padding their sides. Unfortunately, frames that have been rotated will have extra area filled in with "empty" pixels. Two problems result from this method: more memory space is needed and pixel data loss can result from rotation and scaling. This would affect Proscenium's ability to remove stabilization at a later time and maintain fidelity.

The problems are solved through the use of a PFilter called PPZR (Proscenium Pan/Zoom/Rotate). By having all changes affected by a PFilter, there is no need to store the new data state, and pixels can be read from and written to the underlying video data with on-the-fly transformations. The setActualPixel and getActualPixel functions handle this work transparently.

**Figure 6:** Two original frames from a 3 second DV resolution video clip.

The transform matrices are loaded into the PPZR, and it then solves to find the coordinates of the four corners of each frame. These values determine the rectangular bounding box of the new volume. Proscenium believes the upper-left hand corner of each frame is at (0,0), so offsets are implemented to reorient PPZR's transformed coordinate space. The pixelWidth and pixelHeight of the PFilter are substituted with the values of the new extents.

PPZR handles getActualPixel requests by returning discrete pixel data without attempting interpolation or blending. It receives a discrete pixel coordinate which it multiplies by the inverse transform matrix for that frame, and then rounds to an integer value and returns that color or "empty" if outside the volume.

setActualPixel is implemented in a similar fashion. It takes the target coordinate and multiplies it by the inverse matrix and then sets the pixel color in the source material. However, this often does not result in the expected effect. This is because whereas each discrete coordinate input to getActualPixel corresponds to just a single pixel, setActualPixel can have a one-to-many, one-to-one, or even a one-to-none relationship. This often occurs when scaling is involved in a stabilization. There are many ways around the problem of changing the color of an entire area (the example marks an entire area to "empty"). One such way involves taking the corners of the bounding polygon in the transformed coordinates and changing them back to the original coordinates. Filling the new polygon in the source material will then be sure to change all pixels that fall in the transformed boundary.

## 5. RESULTS

We next demonstrate the Proscenium framework and the concepts of spatio-temporal video editing by creating a sample video editing application with the following features:

1. View and play a spatio-temporal volume with slices removed
2. Dynamically build a filter graph of the PFilters so far described
3. Dynamically remove filters from the filter graph
4. Perform video stabilization (PPZR) with a graphical interface

To demonstrate this functionality, we show the steps involved in the removal of a Frisbee being thrown in a video clip where the camera loosely tracks its path from right to left. Removal of shadows or other objects can be done by repeating this technique.

The included timing data demonstrates the system's interactivity, but it is influenced by visualization complexity. The times are from a sample run calculated on a uniprocessor Pentium IV at 2.4 GHz with 1 GB of RAM using a 120 frame 720x480 video. The majority of computation is the interface calling getActualPixel() for graphical display data forcing lazy evaluation. Thus, the more pixels in the interactive volume texture, the longer the delay. To account for this relation, timing data for requesting display pixels is measured in time per-pixel, and calculations altering the data set or rendering final output are given as total processing time.


**Figure 7:** Sheered volume after Frisbee has been stabilized. Three visualizations of the volume are shown above. The top-left image shows the sheered volume at a given time. The right image shows a fixed column through time and the bottom image shows a fixed scan line where the Frisbee's path has been stabilized.

The difficulty of removing the Frisbee from the video is two-fold because neither the Frisbee nor the background is stationary over the course of the video. Each element will be stabilized separately and edits will be made in each stabilized state. Once the edits are complete, stabilization will be removed, but the edits will remain.

The first step involves stabilizing the video around the Frisbee. Initially loading the video takes 3.3s, and display is .92µs/pixel. Once the Frisbee is mid-air, correspondence points are placed on it in every few frames. Because the Frisbee does not make any sudden shifts in its velocity or position, the linear interpolation of the stabilization engine is sufficient with only a few specified correspondence points. The transform matrices for each frame are calculated (.01s) and applied through the PPZR filter. The display of the wider view takes 1.53µs/pixel.

The next step involves selecting a liberal axis-aligned bounding box around the Frisbee. The box also has a time dimension in addition to width and height, which allows selection of a subset of all the frames. This bounding box is executed by adding a PFrame with the extents of the bounding box. Now display takes 1.77µs/pixel due to the added PFilter. A separate operation then sets all of the pixels in the source footage that lie in the PFrame to the "empty" color value(.08s). This cuts a hole in the video, erasing all pixels in the PFrame. At this point, all temporary filters are removed and the display is redrawn at .92µs/pixel.

**Figure 8:** A subsequent sheering of the volume with the background stabilized is shown above. A constant time slice is shown in the upper-left, a fixed column is shown in the upper-right, and the bottom is a fixed scan line through the aligned volume.

**Figure 9:** A frame from the original sequence is shown on the left and the corresponding frame after removal of the Frisbee is on the right.

The desired effect is to fill in the background in areas where the pixels of the Frisbee were removed. The PBackground filter would seem ideal for this situation, but the background is not static over time. The solution is to stabilize the background. Correspondence points are placed on a similar background object in a number of frames. The PPZR filter takes these transforms and redisplays the data in a new stabilized form at 1.53μs/pixel.

The PBackground filter is now ready to operate. However, because it is a PFilter, it must be removed in order to go back to the original video. Thus, the holes must be permanently filled. In order to do this, the original data will be queried for which pixels are "empty", and those pixels will be filled with the appropriate PBackground data from the transformed space (64.7s).

Once the changes are made, all of the PFilters are removed, restoring the original camera motion. The resulting video looks similar to the original, but with the Frisbee removed without loss of temporal coherency. The video can be exported (29.2s) having been maintained at DV resolution throughout the editing process. The total RAM used in the edit was the original uncompressed movie size plus 25-50% of garbage collected scratch space.

The most important result is that the user is in complete control of the editing at every stage of the process. This alleviates the need for special computer vision techniques and scene dependent heuristics by returning control to the artist. Our system provides tools to enable visualizations and complex edits that would not otherwise be possible. The computer takes out the busy work by applying changes to many frames, and leaves the artist in control.

## 6. FUTURE WORK

Our current implementation has several limitations. In particular, our editing approach implicitly treats each object being stabilized as a planar surface using projective transforms. This is a common assumption of most layer-based models, but it cannot accurately align arbitrary 3D shapes. Future work could include complex alignment tools such as morphing or plane-plus-parallax warping.

Our current implementation assumes a standard uniprocessor PC. However, we are optimistic that our lazy-evaluation model can be adapted to parallel and distributed implementations because of its loosely coupled implementation. Processing could also be done using the graphics processing unit on a graphics accelerator to increase the speed of calculation of videos.

## 7. CONCLUSIONS

We have presented Proscenium, a filter-graph framework for processing video as a spatio-temporal volume. Treating videos as spatio-temporal volumes enables new editing capabilities, which we have demonstrated by implementing a prototype video editing system. By providing the user with the capability to sheer, modify, restore, and resheer these volumes, we are able to provide an intuitive image-editing-like user interface that allows tools to be applied throughout the volume while interacting with only a small visible portion of the data. Our filter-graph framework allows operations on large volumes with minimal memory footprint.

## 8. REFERENCES


**[1]** Adobe After Effects 5.5. Adobe Systems Inc.http://www.adobe.com.

**[2]** Adobe Photoshop 7.0. Adobe Systems Inc. http://www.adobe.com.

**[3]** Adobe Premiere 6.5. Adobe Systems Inc. http://www.adobe.com.

**[4]** Alias|Wavefront Maya 4.5. Alias|Wavefront. http://www.alias.com.

**[5]** Apple Final Cut Pro 3. Apple Computer, Inc, http://www.apple.com.

**[6]** Apple QuickTime 6. Apple Computer, Inc, http://www.apple.com.

**[7]** Avid Media Composer. Avid Technology, Inc, http://www.avid.com.

**[8]** AVS/Express 6.0, Advanced Visual Systems, Inc, http://www.avs.com.

**[9]** Bolles, R.C., Baker, H.H., and Marimont, D.H. Epipolar-Plane Image Analysis: An Approach to Determining Structure from Motion. International Journal of Computer Vision, 1, 1, pp. 7-55, 1987.

**[10]** Buehler, C., Bosse, M., and McMillan, L., Non-Metric Image-Based Rendering for Video Stabilization. In Proceedings of CVPR 2001, pp. 609-614, 2001.

**[11]** Chuang, Y.-Y., Agarwala, A., Curless, B., Salesin, D.H., and Szeliski, R. Video Matting of Complex Scenes. In Proceedings of SIGGRAPH 2001, ACM Press, pp. 243-248, 2001.

**[12]** Fels, S., Lee, E., and Mase, K. Techniques for Interactive Video Cubism. In Proceedings of ACM Multimedia 2000, pp. 368–370, 2000.

**[13]** Gamma, E., Helm, R., Johnson, R., and Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, Boston, pp. 175-184, 1995.

**[14]** Klein, A., Sloan, P.P., Finkelstein, A., and Cohen, M.F. Video Cubism, Microsoft Research Technical Report MSR-TR-2001-45, 2001.

**[15]** Klein, A., Sloan, P.P., Finkelstein, A., and Cohen, M.F. Stylized Video Cubes, ACM SIGGRAPH Symposium on Computer Animation 2002, July, 2002.

**[16]** Mayer-Patel, K. and Rowe, L.A. Design and Performance of the Berkeley Continuous Media Toolkit, Multimedia Computing and Networking 1997, Proc. SPIE 3020, pp. 194-206, 1997.

**[17]** Microsoft DirectShow (DirectX 9.0). Microsoft Corporation, http://www.microsoft.com/directx.

**[18]** Porter, T. and Duff T. Compositing Digital Images. Computer Graphics, In Proceedings of SIGGRAPH 1994, ACM Press, pp. 253-259, 1994.

**[19]** Pratt, W.K., Developing Visual Applications; XIL: An Imaging Foundation Library, Prentice Hall, ISBN 0-13-461948-X, 1997.

**[20]** Press, W.H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. Numerical Recipes in C: The Art of Scientific Computing. Cambridge University Press, Cambridge, pp. 36-41, 1992.

**[21]** Wang, J.Y.A. and Adelson, E.A. Layered Representation for Motion Analysis. In Proceedings of CVPR93, pp. 361-366, 1993.

**[22]** Wolberg, G. Digital Image Warping, Wiley-IEEE Computer Society Press, ISBN: 0-8186-8944-7, 1995.

**[23]** Zwicker, M., Pfister, H., Van Baar, J., and Gross, M. Pointshop 3D: An Interactive System for Point-Based Surface Editing. In Proceedings of SIGGRAPH, 2002.