Plasma: A Scripting Language for Processing Media Streams

Tao Zhu, Pavel Korshunov, Bing Liu and Wei Tsang Ooi
Department of Computer Science,
National University of Singapore,
Singapore.

ABSTRACT

Media streaming has found applications in many domains such as education, entertainment, communication and video surveillance. Many of these applications require non-trivial manipulations of media streams, beyond the usual capture/playback operations supported by typical multimedia software and tools. To support rapid development of such applications, we have designed and implemented a scripting language called Plasma. Plasma treats media streams as first-class objects, and caters to the characteristic differences between stored media files and live media streams. In this paper, we illustrate the design and features of Plasma through several small examples. We also describe two example applications that we developed on top of Plasma. These two applications demonstrated that using Plasma, complex applications that compose, mix, and filter multimedia streams can be written with relatively little effort.

1. INTRODUCTION

Many years of research in the area of media compression, middleware technology, and inter-networking has enabled daily use of streaming video and audio over the Internet. Many academic and research institutions are routinely capturing and web-casting lectures. Academic conferences and workshops, such as SIGCOMM and IPTPS have begun to broadcast their technical sessions live over the Internet. At the same time, video conferencing is becoming common in both home (e.g. iChat, MSN Messenger) and institutions (e.g. Access Grid[1]). These developments over the last few years indicate that streaming media is slowly becoming an important part of every day's computing experience.

The increasing usage of streaming video and audio causes proliferation of servers and players to capture, transmit, and playback streaming media. These software tools, however, seldom go beyond the simple functionalities of capturing and transmitting, and receiving and playing back. Non-trivial manipulations of media streams such as composition, mixing, or filtering are seldom supported. These operations on media streams are important in many scenarios. We describe some of these scenarios next.

- In a typical live web-cast production environment with multiple cameras, a director is responsible for producing the stream sent to public audience.[2] The director controls what the audience sees by switching between interesting camera views, adding titles, inserting logos, inserting rolling credits and opening sequence, or composing the output stream using video streams from different cameras. All these operations require manipulations of live video streams.
- In Access Grid, difference conference venues might have different conference setup and hence cannot communicate with each other. Access Grid allows creation of *network services*,[3] which transcode video and audio among sites to enable interoperability. Complex network services could allow filtering of inactive ("blue screen") video streams, or composition of multiple streams from a single site to enhance viewing experience.
- Some existing IP-based, distributed, video monitoring systems capture video using network cameras or video sensors, and transmit the video over the network for analysis and monitoring.[4] One approach to scale up such system to large number of video sources is to filter out uninteresting video streams. We[5] proposed an approach where low quality video streams are sent to proxy for analysis and only when interesting events are detected, high quality streams are sent to human for viewing. Processing of live video streams is needed in such system. Composition of video streams using live video feeds from cameras and recorded video from archives is useful in video monitoring system as well. For example, a user might want to compare two intresting video side-by-side.

- In video jockeying, a VJ can choose different video files from recorded video or cameras, and apply mixing and filtering to create special effects, reacting to the beat of music.

Despite the increasing needs to process video streams, there is a lack of tools to support such needs. Most available tools and libraries support only limited processing operations on video streams and are hard to extend (e.g. OpenMash[6]). On the other hand, many tools for processing stored video exist. These include many scripting languages, designed for rapid prototyping of stored multimedia processing application (e.g. VideoScript,[7] Rivl[8]). These languages, however, do not support live multimedia streams.

Extending existing scripting languages designed for processing stored media files to process live media streams is not straightforward. Live media has different properties from stored media. Firstly, an abstraction for a live media stream should expose the network conditions associated with the media stream, such as packet loss rate. One useful application for such exposure is that it allows scriptable adaptation strategies based on network conditions. Secondly, processing a stored media file involves only storage devices and display devices. On the other hand, a live media stream has a network source and destination, possibly consisting of a group of receivers. The source of the media stream is no longer limited to storage devices, but could be camera, television or other video sources. Finally, temporal manipulations, such as concatenating two video files and deleting certain scene from a video, are not meaningful for live media streams. On the other hand, spatial manipulations, such as creating a picture-in-picture effect or adding logos or sub-titles to live media streams are common operations.

To address the need for an easy-to-use tool for manipulating live media streams, we have designed and implemented a new scripting language called *Plasma* (Programming LAnguage for Streaming MediA). With the differences between stored media and live media in mind, our language is designed with the following features:

- Exposure of underlying network conditions to programmers;
- Integration with audio/video devices as sources and sinks; and
- Focus on spatial composition operations.

These features differentiate Plasma from existing languages in the literature.

Although designed with specific features to support live media streams, Plasma supports manipulation of stored media like many existing languages. Supporting both live and stored media in Plasma enables mixture of both media types easily. Such mixing operations are useful in many contexts. For instance, a web-cast operator might want to insert a pre-recorded opening sequence prior to broadcasting live streams. Others might want to insert pre-recorded advertisement segments in between sessions while web-casting seminars. Another trivial and commonly used operation is to receive live media streams and record it to disk as stored media.

In this paper, we highlight the design and features of Plasma, and show how useful, complex manipulations of media streams can be done easily in our language. We first describe the basic abstractions and operations supported by Plasma in Section 2. Section 3 further elaborates on the features and usefulness of Plasma through several examples. We describe the implementation of Plasma briefly in Section 4, and two applications written using Plasma in Section 5. Section 6 compares Plasma with existing tools and languages in the literature, and Section 7 concludes.

2. DESIGN OF PLASMA

2.1. Plasma as a Scripting Language

We faced a design decisions at the beginning of the project, in deciding whether to implement Plasma as a software library, a system-level programming language or a scripting language. We chose to implement Plasma as a scripting language, partly because of the many advantages of scripting. A scripting language allows rapid prototyping and normally has succinct syntax (and thus shorter code). Furthermore, scripts written can be run across different platforms without recompilation, as long as the language interpreter is ported.

Another main reason that drives us towards using a scripting language is that we would like to exploit the strengths of the Tcl/Tk language, in particular the strength of Tk as a GUI building tool. As a scripting language

for building GUI programs, Tk excels at two things: (i) specifying layouts of widgets and (ii) binding of GUI events to callbacks. By borrowing the syntax of Tk, Plasma allows spatial arrangement and composition of different media streams easily. By supporting events and callbacks, Plasma allows callbacks that manipulate media streams to be written when certain events (e.g., network congestion) are triggered. Due to the ubiquity of Tk as the GUI toolkit for many scripting languages (including Perl, Ruby, Python and of course, Tcl), we believe that the programming model and syntax used in Tk are familiar to many programmers. For these reasons, we designed the syntax of Plasma to closely resemble those of Tk, and implemented Plasma as an extension to the Tcl scripting language, the same language Tk is built upon.

2.2. Abstractions in Plasma

Plasma supports two main types of abstract objects, *media objects* and *events*. A media object is a unified abstraction for live media streams, stored media files or analog media sources. A media object can be either a video or an audio object, and it can recursively contain other media objects. As an example, consider a composite media object representing a web-cast seminar. This object consists of a video object and an audio object. The video object can, in turn, contains two other video objects, composed as a picture-in-picture arrangement, one showing the speaker, the other showing the slides.

An event object, which can be created and bound to a callback by the programmer, represents an interesting phenomenon that occurs at some time instance. Common GUI events such as mouse and keyboard inputs are supported, as well as timer events. Most interestingly, programmers can define events that are based on network conditions (e.g., when network is congested) or content of the video streams (e.g., when motion is detected). Network-based events allow customized adaptations based on network conditions, while content-based events allow *continuous query* on media streams to be written. Such content-based events could be useful for building video surveillance applications.

2.3. Plasma and Indiva

The media object abstraction in Plasma can represent not only pre-recorded and live streams, but analog media devices such as cameras, cable boxes and VCR as well. To interact with such devices, Plasma uses a middleware called Indiva.[9] We briefly present Indiva below.

Indiva (INfrastructure for DIstributed Video and Audio) is a middleware over a *distributed audio/video environment*. Such an environment typically consists of audio/video equipments and computers used to capture media signals, control equipments and process media streams. The media signals and media streams are transmitted over audio/video routing networks and IP networks. Indiva provides a layer of abstractions and a simple set of APIs over such an environment, hiding the implementation details of the environment from applications.

Indiva uses a UNIX file system metaphor for managing software processes, hardware devices and media data in a distributed audio/video environment. These resources are named and organized into a hierarchical name space, much like UNIX file system. File name extensions indicate the type of the resource (e.g., .cam indicates that a resource is a camera). Indiva relies on a centralized server, called *Indiva manager*, to store the name space and meta-data for the managed resources. Applications connect to the Indiva manager, and issue command to the manager to manipulate the underlying environment (e.g. show a particular video on to a projector screen).

Indiva provides what Plasma needs to interact with analog media devices. From Plasma point of views, an analog media device is just another media object that can be manipulated in a unified way as, say, an MPEG video file. Internally, Plasma relies on Indiva to capture media signals from these devices, and transmit them as media streams. In the next section, we will see examples on how Plasma uses Indiva to access remote cameras.

3. EXAMPLES

In this section, we present several examples to illustrate the features and power of the Plasma scripting language.

```
1  media .m indiva://imgr.nus.edu/cs/room420/speaker.cam
2  .m output capture.mpg
```

Figure 1. A Plasma script for capturing video from a remote camera and saving it to local disk.

3.1. Example 1: Capture from Camera to Disk

Our first example, shown in Figure 1, captures a video from a remote camera and saves it to disk as an MPEG file.

Line 1 of Figure 1 creates a media object representing a camera, specified by a given URL. The line consists of three tokens. The first token, `media` is a Plasma command for creating a media object. The second token, `.m`, is the name given to the media object created. The way Plasma creates a media object is similar to how Tk creates a widget. We retain the naming convention of widgets in Tk, by preceding all object names with a period '`.`'. The rest of the line specifies parameters for creation of the media object. In this example, we create a media object from a camera. We specify the camera using its corresponding name as managed by the Indiva middleware. The URL, with scheme `indiva://`, also specifies the name of the Indiva manager (`imgr.nus.edu`) and the path of the camera in the Indiva name space (`cs/room420/speaker.cam`).

After Line 1 creates the media object, Line 2 saves the object to local disk as a file named `capture.mpg`. Note that the syntax here again follows the convention of Tk. In an object-oriented style, we first write the name of the object we want to operate on (`.m`), followed by the operation (`output`), followed by parameters to the operation. In this case, the parameter is the name of the file `capture.mpg`.

Even though this simple Plasma script has only two lines of code, The execution of the script is rather complex. Firstly, Plasma recognizes that the URL is an Indiva URL, and contact the Indiva manager to request for a media stream from the camera. Indiva launchs the necessary software processes to capture, encode and transmit the stream to the host where the script is executed. The Plasma interpreter opens a socket to receive the media stream for processing. In Line 2, Plasma recognizes that the stream is to be saved as an MPEG video based on the filename extension. Plasma then reads the video stream from the socket, trans-codes it to MPEG (if necessary) and writes the resulting video to a file called `capture.mpg`.

Each of Plasma commands supports a number of configuration options that can be used to overwrite the default behavior of the commands. For instance, the option `-fps` and `-bps` can be used to configure the frame-rate and bit-rate of the media object.

To stop capturing, we can simply delete the object `.m`.

3.2. Example 2: Spatial Composition

Temporal manipulations for live media streams are not as important as spatial manipulations. For this reason, one of the guiding principles in designing Plasma is that it should support easy spatial composition operations. Figure 2 shows one such example. In this scenario, the user would like to view two video streams side-by-side, the first is a current live view of a surveillance camera pointing at the door of a computer lab. The second is a surveillance video from an archive. The user possibly wants to compare what has been stolen or moved in a lab. While it is possible to play these two video streams at the same time as two separate video streams, composing it into a single video has its advantage. For instance, the user can save both views into a single video file using the `output` operation introduced in the previous example.

In Figure 2, the first three lines uses the `media` command to create three different media objects. Unlike the others, the first media object, called `.out`, is created without any parameters. This command creates a media object that acts as empty container. This container is analogous to a `frame` widget in Tk. Line 2 and 3 of Figure 2 create two media objects, one from a camera, specified by an Indiva URL, and another from an archived stream from some RTSP server. The syntax of Line 2 and 3 is similar to that of Example 1, except that, in this example,

```
1  media .out
2  media .out.door indiva://imgr.nus.edu/cs/lab0432/door.cam
3  media .out.suspect rtsp://archive.nus.edu/2005/6/20/door.mpg
4. mpack .out.door .out.suspect -side left
5. .out play
```

Figure 2. A Plasma script for arranging two media streams side-by-side.

the name of the object is prefixed by `.out`, the container object. This naming convention means that the object `.out.door` and `.out.suspect` are created as children of `.out`. This convention is again borrowed from Tk.

After we created two video streams, contained within a new media object, we need to specify how these two streams should be spatially arranged. Plasma uses a command called `mpack`, which behaves similarly to the `pack` command in Tk. Line 4 of Figure 2 illustrates how this command is used. The `mpack` command takes in a list of media objects as parameters, and arranges the objects according to the options specified. In this example, the two video streams are arranged side-by-side, from left to right.

Finally, Line 5 of this example calls the operation `play` of object `.out`. This operation has the following effects. The Plasma interpreter first creates the two video streams by contacting the Indiva manager and RTSP server, and opens two sockets to receive the two streams. The two video streams are then decoded, spatially composed into a new video stream (called `.out`). The new video stream is played out onto the screen. Figure 3 shows a sample output video from the script.

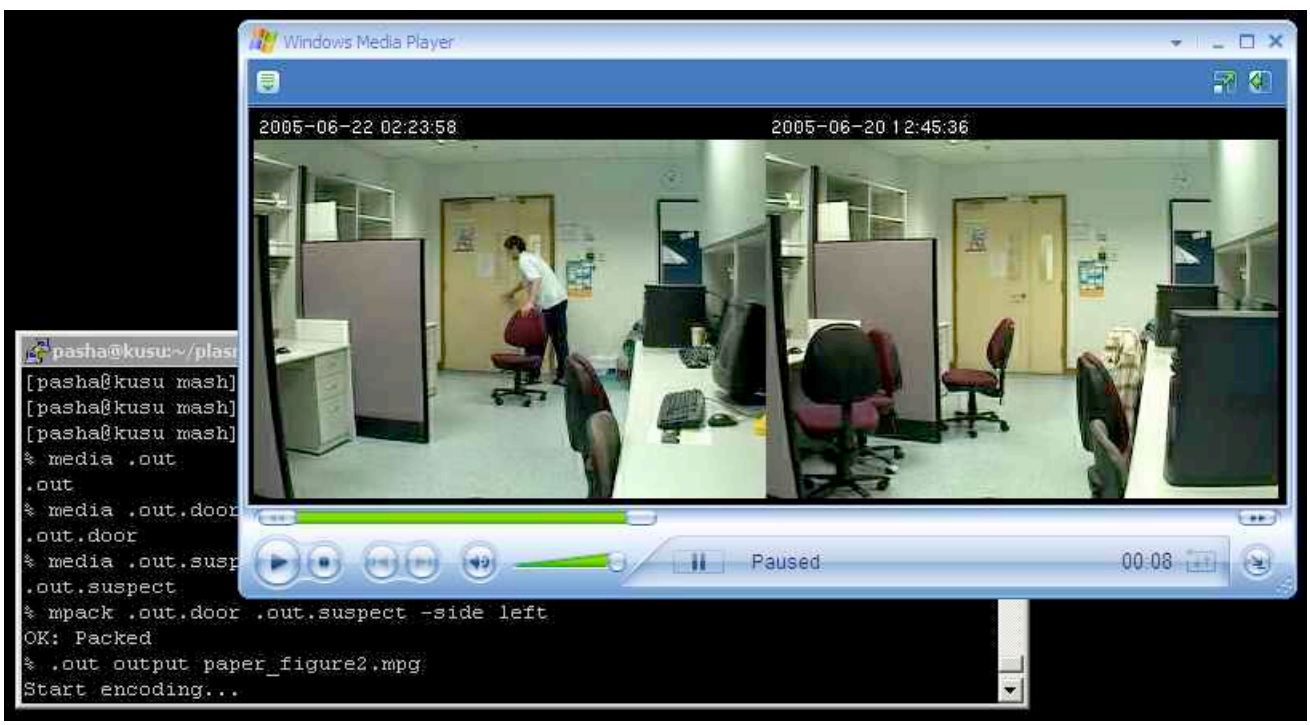

Figure 3. Example output from Plasma where two videos are arranged side-by-side.

This example illustrates that Plasma borrows heavily from the syntax of Tk for spatial arrangement and for

specifying parent-child relationship among objects. Plasma, however, extends the semantic of these command to meet its own requirements. For instance, one common operation in spatial composition is to overlay one video on top of another, but the `pack` command from Tk does not allow two widgets to overlap. The `mpack` command in Plasma allows this arrangement, by checking if the container object is empty. If the container is non-empty, the child media object is overlayed on top of the parent as a new layer. Figure 4 illustrates this operation by showing how picture-in-picture effects can be achieved in Plasma.

```
1  media .out indiva://imgr.nus.edu/cs/lab0432/door.cam
2  media .out.suspect rtsp://archive.nus.edu/2005/6/20/door.mpg
3. .out.suspect scale 0.3
4. mpack .out .out.suspect -anchor sw
```

Figure 4. A Plasma script for arranging two media streams as a picture-in-picture.

The code shown in Figure 4 is similar to that shown in Figure 2, except that the container media object `.out` contains a live camera view, and the child media object `.out.suspect` contains a view of an archived video. When we call `mpack` in Line 4, Plasma overlays the child video over the container video. The option `-anchor sw` tells Plasma to put the child video on the south-west corner of the container. The command `scale` on Line 3 simply reduces the width and height of the video to 30% of its original size so that the container video is not obstructed. Figure 5 shows a sample output video from this script.

3.3. Example 3: Events

The next example shows the event-driven aspect of Plasma. We show how events are created and bound to callbacks in this example.

Figure 6 shows a Plasma script that transmits video stream captured from a surveillance camera in a computer lab, whenever motions are detected in the room. The first three lines in this script create an event with command `mevent`. The command `mevent` has three arguments – the first argument is the name of the event, the second is a list of parameters needed to define the event, and the third argument is an expression which is evaluated periodically to check if the event should be triggered. The name of the event is surrounded by a pair of triangular brackets, following the naming convention of events in Tk. The event created here is triggered by detection of motion in the input video (parameterized by variable `obj`). The parameterization of the video object allows the same event to be applied to different video objects in Plasma.

Line 4 of the script above creates a video object, captured from a surveillance camera. Line 5 to 10 then bind this video object to the motion detection event and specify the callback function of the event. In the callback, a new video stream named `.out` with the same URL as the camera is created (Line 7) and is transmitted to multicast session 224.4.4.4 at port 44444 (Line 8).

The event expression (Line 2) is evaluated periodically. By default, it is evaluated every frame. Thus, in the example above, the event is triggered for every frame and the callback is executed for every frame, as long as there is motion. The conditional statement in Line 6 of Figure 6 ensures that a new video stream is created when motion is detected for the first time only, not every frame!

Finally, Line 11 to 18 of the example stop transmission of the multicast stream when motion no longer exists in the source video.

3.4. Example 4: Exposing Underlying Network Conditions

One of the differentiating feature of Plasma is that it exposes the network statistics associated with a media stream to the programmer. This exposure allows programmer to make application-level decisions based on the network conditions. While finer grain adaptations such as rate control or adaptive FEC are best left to the media

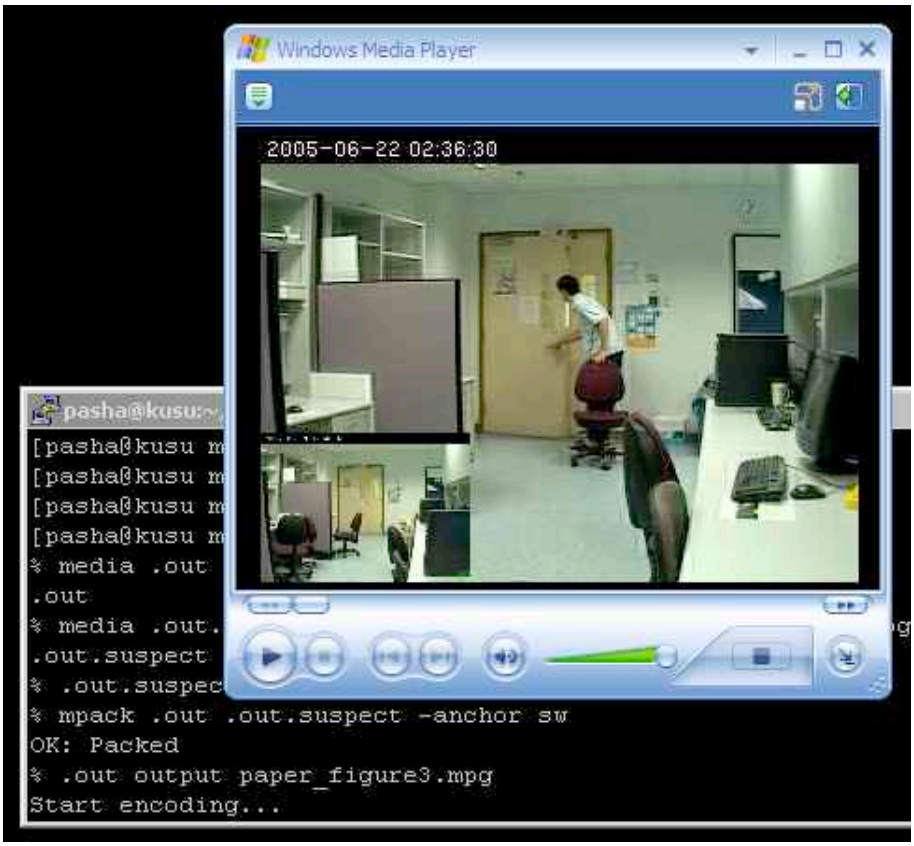

Figure 5. Example output from Plasma where two videos are arranged as a picture-in-picture.

streaming engine instead of controlled through scripts, in certain scenarios, such fine grain adaptations are not possible. An example is video streams that are transmitted from network cameras. In the next example, shown in Figure 7, we illustrate how coarse grain adaptation can be achieved using Plasma.

Figure 7 illustrates the use of `lossrate` command of a media stream object. The command returns a number between 0 and 1 indicating the weighted average packet loss rate of the stream. Using this command, the script defines two events `HighLoss` and `LowLoss`, which correspond to loss rate higher than 10% and no higher than 10% respectively (Line 1 to 6). The callbacks of these two events configure the network camera (at IP address 192.168.0.4) to send its video at lower or higher frame rate, depending on the state the network is in.

4. IMPLEMENTATION

The core functionality of Plasma is implemented in C and C++, with Tcl binding to provide the scripting interface. A library `libplasma` is provided as a Tcl dynamically loadable library. This library allows users to launch a Tk shell, load the library, and use Plasma commands in an interactive manner. Of course, a user can also run a Plasma script as a batch file by passing the name of the script file as argument to the intepreter.

Plasma makes heavy use of existing open source libraries, including FFMPEG for video decoding and encoding, LAME for audio decoding and encoding, FreeType and Glyph Keeper for text rendering, Dali for image processing, SDL for audio and video display and OpenMash for interfacing with Indiva.

5. APPLICATIONS

To demonstrate and verify the ease-of-use of Plasma as a scripting language, we have developed two applications on top of Plasma. The first application is called VJOnline, which is an application for video jockey to add effects

```
mevent <HasMotion> {obj} {
   [$obj has_motion]
}
media .camera indiva://imgr.nus.edu/cs/lab0432/door.cam
mbind <HasMotion> .camera {
   if {![mexists .out]} {
      media .out [.camera url]
     .out output rtp://224.4.4.4:44444/
  }
}
mevent <NoMotion> {obj} {
    ![$obj has_motion]
}
mbind <NoMotion> .camera {
   if {[mexists .out]} {
      mremove .out
   }
}
```

Figure 6. A Plasma script for transmitting data from camera as RTP stream upon detecting motion.

```
mevent <HighLoss> {obj} {
  [$obj lossrate] > .1
}
mevent <LowLoss> {obj} {
  [$obj lossrate] <= .1
}
media .camera http://192.168.0.4/mpeg4/1/media.amp
.camera configure -fps 30
mbind <HighLoss> .camera {
  .camera configure -fps 20
}
mbind <LowLoss> .camera {
  .camera configure -fps 30
}
```

Figure 7. A Plasma script that changes frame rate from a network camera depending on loss rate.

and composite video displays on the fly. The second application is called SLIME. SLIME is a video editor for layout and composition of video from multiple media sources. In the rest of this section, we elaborate on the functionality of these two applications and show how Plasma allows these applications to be built quickly.

5.1. VJOnline

VJOnline is a tool for a VJ (video jockey) or web-cast director, to open different video and audio sources, manipulate them with preset effects, and broadcast them to the audience over the Internet. VJOnline supports (i) transition effects such as fade in, pop in, slide in, and curtain in, (ii) picture-in-picture effects, and (iii) text marquee effects. A screenshot of VJOnline is shown in Figure 8.

Since the video effects and processing operations are implemented in Plasma, it takes only about 300 lines of code to implement VJOnline. Out of these, about 240 lines of code are for media processing, and the rest are Tk code for constructing the GUI. Figure 8 shows a screenshot of the user interface of VJOnline.

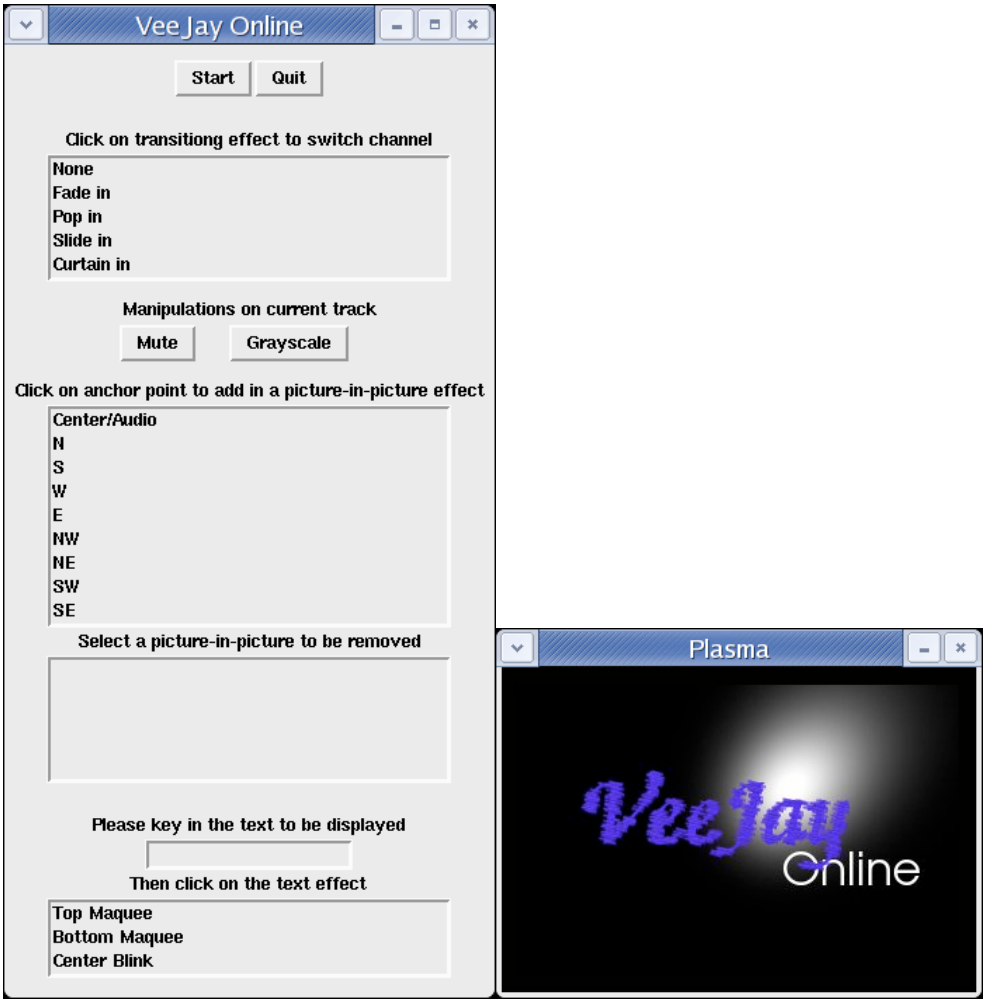

Figure 8. Screenshot of VJOnline

5.2. SLIME

The name SLIME stands for Stored/LIve Media Editor. It is an application that is designed to provide intuitive, drag-and-drop interface for composing video. SLIME is designed with one of the three grand multimedia

challenges in mind – *"making video editing as easy as word processing"*.[10] SLIME has not fully achieve this goal as it focuses on spatial editing only. As far as spatial composition is concerned, however, we design SLIME user interface to be similar to common drawing interface found in desktop software such as Microsoft PowerPoint and Adobe Photoshop.

A typical user of SLIME opens up a set of video clips, and drag-and-drops the clips onto a scratch canvas. Each clip is represented as an image showing a snapshot of a frame in the video. On the scratch canvas, the user composes the output video by moving, resizing, and cropping the video clips through clicking and dragging their corresponding image. Similar to common drawing software, the user can click anywhere on the canvas, and type in text strings, which will be overlayed on top of the resulting video. Similarly, a user can overlay another image (such as logo) on top of the composed video. The user can select individual object or group of objects on canvas, and change their properties (how long the overlay text will appear, apply additional special effects, etc.). After the user is satisfied with the composed video, she can then playback the composed video, or save the video to a file. A screenshot of SLIME is shown in Figure 9.

Figure 9. Screenshot of SLIME

Despite the complexity of SLIME, the application is written in only about 8000 lines of Tcl/Tk code. SLIME uses libraries such as vTcl for GUI constructions, TkZinc for drawing and manipulating objects on canvas, and external program such as Mplayer for playback. Most interestingly, SLIME relies on Plasma for processing the video. When a user wants to save or playback a video, SLIME analyzes the layout and properties of objects on its scratch canvas, and generates the equivalent Plasma script, which is then passed to Plasma interpreter to generate the output video. Thus, no actual video processing code is written as part of SLIME (except to extract thumbnails and snapshot image for the canvas).

During the initial development phase of SLIME, a preliminary version of Plasma which supports only stored media is used. At a later stage of the project, when support for live media streams is stable enough in Plasma, SLIME switches to using this newer version. Interestingly, SLIME suddenly supports streaming video as one of the video sources, allowing composition of video consisting of both live and stored media, with minimal changes to its code. This experience demonstrates the generality of abstractions over media streams provided by Plasma.

6. RELATED WORK

The complex nature of multimedia applications has spurred many toolkits and libraries to simplify the software development process since mid-90's. Much research effort has been put into development of toolkits, middleware, and programming frameworks for distributed multimedia applications. These efforts include Berkeley CMT,[11] MASH[12*], InfoPipes,[13] and NMM.[14] These toolkits typically provide abstractions in the form of sources, sinks, filters, and connections, allowing programmers to connect the different components to build a distributed multimedia application. Other efforts such as QCompiler[15] and PLASMA[16] [†] focus on building adaptive distributed multimedia applications. In contrast, applications built with our scripting language are not distributed – they are meant to be run on a single host and communicate with others servers through standard or open protocols such as HTTP and RTSP. Further, Plasma focuses on processing of stored media and live media streams, with emphasis on spatial composition. Therefore Plasma compliments the design goals of these distributed multimedia toolkits.

Many scripting languages for processing media exist as well. Some examples include VuSystem,[17] Rivl,[8] VideoScript[7] and AviSynth.[18] These languages focus on processing stored video only, and do not allow easy manipulation of video streams.

SMIL[19] provides a comprehensive set of functionalities for spatial-temporal layout and transitional effects on multimedia data. Unlike Plasma, which is imperative, SMIL is declarative in nature. Further, even though SMIL supports adaptive content, the adaptivity is limited to systems parameters (e.g., CPU speed) while Plasma allows a much more flexible adaptation, for instance based on dynamic network conditions or user-defined events.

MedsMan[20] is a system for querying live media streams and provides description languages and an SQL-like querying language for managing live media streams. It allows generation of *feature stream* – sequence of features (and associated metadata) resulting from analysis on media stream. Similar to Plasma, MedsMan allows user-defined events that are triggered based on results of continuous queries. In the case of MedsMan, the queries are executed on the feature streams. Plasma, however, does not support feature streams. On the other hand, MedsMan focuses on querying only and does not support spatial compostion and is not network-aware.

7. CONCLUSION

This paper presents a new scripting language (built upon Tcl/Tk) called Plasma. The rational behind designing and developing this new language is that, existing libraries and toolkits lack sufficient support for non-trivial manipulations of video and audio streams, a feature that we believe is increasingly critical. With target applications such as video conferencing, web-casting, and video surveillance in mind, we designed Plasma to treat media streams as first class citizens in the language – Plasma exposes the underlying network statistics of live streams, allows control of video and audio devices (which can produce streams) and enables easy spatial compositions of streams.

Plasma is still under development. Besides continuing development work to optimize performances and extending Plasma with useful operations, we are interested in implementing generic streams in Plasma. Generic streams would be useful in applications such as video surveillance or sensor networks, where time sequence of data (e.g. strings, numbers) are received and transmitted. Besides allowing continuous queries and definition of events over these data, one interesting new ability of Plasma if such generic stream is supported, is composition of data streams with media streams for visualization purposes.

[*] and its later incarnation, OpenMash (http://www.openmash.org)

[†] not to be confused with our language, Plasma

REFERENCES

1. L. Childers, T. Disz, R. Olson, M. Papka, R. Stevens, and T. Udeshi, "Access Grid: immersive group-to-group collaborative visualization," in *Proceedings of the 4th International Immersive Projection Technology Workshop*, (Ames, IA), June 2000.
2. T. Yu, D. Wu, K. Mayer-Patel, and L. A. Rowe, "dc: a live webcast control system," in *Proceedings of the SPIE Multimedia Computing and Networking (MMCN), Vol. 4312*, (San Jose, California), Jan. 2001.
3. H. Gao, R. L. Stevens, and M. E. Papka, "The design of network services for advanced collaborative enviornments," in *Proceedings of the 3rd Workshop on Advanced Collaborative Enviornments (WACE)*, (Seattle, WA), June 2003.
4. M. Valera and S. Velastin, "Intelligent distributed surveillance systems: a review," *IEE Proceedings on Vision, Image and Signal Processing* **152**, pp. 192 – 204, Apr. 2005.
5. P. Korshunov and W. T. Ooi, "Critical video quality for distributed automated video surveillance," in *Proceedings of the 13th ACM International Conference on Multimedia*, pp. 151–160, (New York, NY, USA), 2005.
6. OpenMash Consortium. http://www.openmash.org.
7. VideoScript. http://www.videoscript.com.
8. J. Swartz and B. C. Smith, "A resolution independent video language," in *Proceedings of the 3rd ACM International Conference on Multimedia*, pp. 179–188, (San Francisco, CA), November 1995.
9. W. T. Ooi, P. Pletcher, and L. Rowe, "Indiva: a middleware for managing distributed media environment," in *Proceedings of ACM/SPIE Multimedia Computing and Networking (MMCN), Vol. 5305*, pp. 211–224, (San Jose, CA), January 2004.
10. L. A. Rowe and R. Jain, "ACM SIGMM retreat report on future directions in multimedia research," *ACM Transactions on Multimedia Computing, Communications and Applications* **1**(1), pp. 3–13, 2005.
11. K. Mayer-Patel and L. Rowe, "Design and performance of the Berkeley Continuous Media Toolkit," in *Proceedings of ACM/SPIE Multimedia Computing and Networking (MMCN), Vol. 3020*, pp. 194–206, (San Jose, CA), January 1997.
12. S. McCanne, E. Brewer, R. Katz, L. Rowe, E. Amir, Y. Chawathe, A. Coopersmith, K. Mayer-Patel, S. Raman, A. Schuett, D. Simpson, A. Swan, T. L. Tung, D. Wu, and B. C. Smith, "Toward a common infrastructure for multimedia-networking middleware," in *Proceedings of 7th. Intl. Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, pp. 39–49, (St. Louis, Missouri), May 1997.
13. A. P. Black, J. Huang, R. Koster, J. Walpole, and C. Pu, "Infopipes: an abstraction for multimedia streaming," *Multimedia System* **8**(5), pp. 406–419, 2002.
14. M. Lohse, M. Repplinger, and P. Slusallek, "An open middleware architecture for network-integrated multimedia," in *Proceedings of Joint International Workshops on Interactive Distributed Multimedia Systems and Protocols for Multimedia Systems (IDMS/PROMS)*, *Lecture Notes in Computer Science* **2515**, pp. 327–338, Springer, 2002.
15. D. Wichadakul, X. Gu, and K. Nahrstedt, "A programming framework for quality-aware ubiquitous multimedia applications," in *Proceedings of the 10th ACM International Conference on Multimedia*, pp. 631–640, 2002.
16. O. Layaida and D. Hagimont, "PLASMA: a component-based framework for building self-adaptive applications," in *Proceedings of Embedded Multimedia Processing and Communications*, pp. 185–196, (San Jose, CA), Jan. 2005.
17. C. Lindblad, D. Wetherall, and D. L. Tennenhouse, "The VuSystem: a programming system for visual processing of digital video," in *Proceedings of the 2nd ACM International Conference on Multimedia*, pp. 307–314, (San Francisco, CA), Oct. 1994.
18. AviSynth. http://www.avisynth.org.
19. L. Rutledge, "SMIL 2.0: XML for Web multimedia," *IEEE Internet Computing* **5**(5), pp. 78–84, 2001.
20. B. Liu, A. Gupta, and R. Jain, "MedSMan: a streaming data management system over live multimedia," in *Proceedings of the 13th ACM International Conference on Multimedia*, pp. 171–180, 2005.