Programmable Transitions for Video Stream Editing

Alexandre Hardy*
Academy for Information Technology
University of Johannesburg

Abstract

Video editing applications provide a facility to transition from one video stream to another, or to filter a video stream in some way. New transitions are usually developed using a custom API for the particular package. In this article we present a shading language for specifying transitions and filters on video streams. Video editing is performed by constructing a tree based on the transitions. The shading language is compiled to a virtual machine, but can still be executed efficiently. We provide several examples of transitions which have been successfully applied using our video stream editing software.

CR Categories: D.3.2 [Language Classifications]: Specialized application languages; I.3.3 [Computer Graphics]: Picture/Image Generation I.4.0 [Image Processing and Computer Vision]: Image processing software

Keywords: Stream processing, video editing, video shaders

1 Introduction

Video editing software is well established. Such software is characterized by non-linear editing and a plethora of transitions and effects that can be applied to the video streams. If additional effects or transitions are required, an SDK is used to develop those transitions. This process requires specialized knowledge of the SDK and tools to compile the plug-in.

We propose a custom video editing language, inspired by the shading language used in the REYES [Cook et al. 1987] architecture to efficiently implement transitions and filtering on video streams. The shading language operates on streams in such a way that the details of frames and video streams are largely hidden from the user. The shading language is compiled on demand by the video editing software, and executed to produce a video stream. Thus no additional software packages, compilers or SDK's are needed.

We construct a tree to specify video editing operations instead of the conventional timeline formulation. The tree structure allows multiple transitions to be applied within a segment of the output video.

In the next section we discuss work related to shader programming and models for video editing. Section 3 and following sections present the video stream editing model and discuss the language features. We also discuss efficiency of the implementation and conclude with several examples of transitions and effects that have been used effectively in our software.

*e-mail: alexandre.hardy@gmail.com

2 Previous work

Most video editing software is based on storyboards or timelines [Casares et al. 2002] although tree structures have also been used [Ueda and Miyatake 1996]. Girgensohn et al. [Girgensohn et al. 2000] automatically determine video segments, and determine clip lengths to provide a desired final video length to simplify video editing. CVEPS [Meng and Chang 1996] allows editing to be done in the compressed-domain of video streams, but transitions are not always simple to express. Several transitions are discussed, but no general approach to transitions is given. VideoScheme [Matthews et al. 1994] uses scheme to allow users to manipulate movies and individual frames. Frames and movies are built-in types, but the user is still required to manually choose frames and operations. The provided interface is not stream based and requires extensive work on the part of the user to implement transitions. Davis [Davis 2003] advocates computational media where frames and pixels are physical entities used in a computation to deliver media. Our software is a realization of this approach to a minor extent.

Shade trees [Cook 1984] introduced programmable shading to the computer graphics literature, which developed into the shader system used in the REYES [Cook et al. 1987] architecture. The shading language is important not only for the programmability introduced, but also for the efficiency of the shading language. The idea of a shading language has become pervasive to the extent that hardware [Olano and Lastra 1998] has been developed to execute these programs. Since then GPU's have been employed to solve more general stream computing problems [Buck et al. 2004; Bolz et al. 2003]. Ansari [Ansari 2003] showed that GPU's can be employed to implement transitions and video filters, but the use of the standard GPU shading languages requires shader programming knowledge and additional infrastructure for video decoding and encoding support. Our software supports these aspects directly and hides the details from the user.

None of the approaches listed in this section have supported a language designed to work on video streams, where most of the technical aspects of video are hidden from the implementer of transitions. Although software exists that allows filters to be programmed (FXBench from `http://www.burgers-transition-site.de`), such software generally does not support transitions, or more general video effects.

3 Stream Editing Architecture

We have attempted to create a programming environment for video transitions that is both flexible and simple. Simplicity is achieved by restricting all operations to frames within the video sequence. Any transition only has access to the frames from the streams that it must transition between at the current point in time. Most transitions can be described even more simply, by describing how one new pixel is produced from pixels of the provided frames. The programming language thus specifies how to compute a pixel color, given the current frames of the input stream. This approach hides all frame and video stream details from the transition programmer.

The breakdown of a programmed transition is illustrated in figure 1.

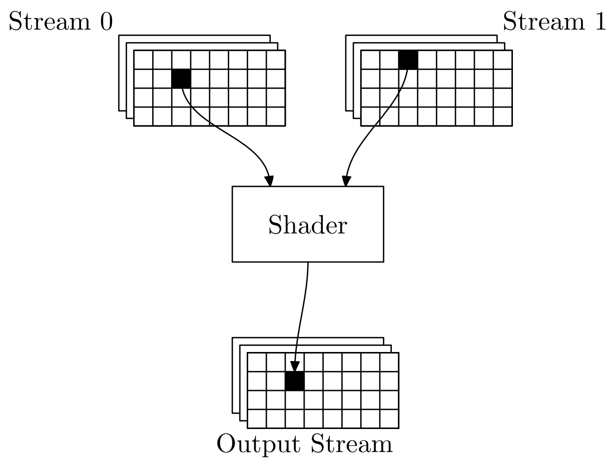

Figure 1: *Shader architecture.*

The video editing system decodes video streams and provides the frames necessary for a video shader to execute. The shader is executed on a per pixel basis. That is, the shader is executed for each pixel in the output frame. The underlying virtual machine is responsible for building frames from the pixels computed by the shader, and for providing access to input frames. The video editing system produces a video stream from the individual frames. As a consequence the shader program cannot modify the speed of the video stream, nor can it reverse the direction of the video stream. No changes to the timing of the video stream are possible with the shading program.

The video is specified by building a tree of operations on streams. Initial streams can be specified by shaders which produce information procedurally, or by specifying a source stream (our implementation is limited to Ogg Theora files). Our implementation also supports static images which are continually reproduced to form a stream. In the future we plan to add text streams for traditional applications such as sub-titling and end credits.

The initial streams serve as input to shader programs, where the output of the shader program is used to create a new video stream. The output of these streams can then serve as input to other video shader programs to produce further streams. The compiler for the stream specification prevents cycles from occurring in the tree structure.

3.1 Specifying Streams

The stream tree hierarchy is specified with a script file. Each line of the script file is of the form

$$\textit{variable} = \textit{value};$$

The file usually begins with variables specifying the output width, height, aspect, frame rate and target bit rate. Streams are specified in a similar manner:

$$\textit{variable} = \textit{type}(\textit{parameters});$$

where parameters are constants or the names of previously declared streams. The stream type `stream` specifies an Ogg Theora file, and `image` specifies a PNG file to use as input. Any other type is a reference to a shader program that will be executed to produce output. The parameters of the shader program can be previously declared streams.

Each stream has several modifiers that can be applied to the stream, these modifiers include:

- `fps`–To re-specify the speed of the stream. This modifier can be used to speedup or slow down the stream. Since shaders cannot modify the speed of a stream, this parameter provides an important control over the speed of the entire stream.
- `width`, `height`–This modifier specifies the output width and height of a stream produced by a shader program. By default the system uses the output width and height. If lower quality is acceptable, then the width and height can be changed. The width and height directly influence the execution speed of the shader program, so these modifiers may be useful for obtaining extra performance during the execution of a script.
- `delay` – The time at which the stream starts. Before the start time, black frames are created. The `delay` modifier could be used by an interactive graphical program to specify the point on the timeline where a particular stream was placed.
- `from`, `to` – These modifiers are used to select a specific section of the video stream.

The combination of stream modifiers and shader programs allow most common transition effects to be implemented.

3.2 Shader Program Features

Each shader program is written in a language similar to C. The language has built-in support for integer, floating point and vector (color) types. An additional parameter type is supported (`frame`) which is used to identify input video streams.

A shader program consists of a number of functions, one of which must have the same name as the file in which it resides. This function returns a `stream` to indicate that it produces the video stream for the shader. Only one such function may be present in any shader file. The usual conditional, assignment and arithmetic operations and language constructs are supported. Arrays are restricted to static arrays.

Our implementation has no graphical interface, but the shader language has been designed to support graphical interfaces. Each parameter can be described by what text should be shown for selecting the parameter, default values and the type of parameter (for specific GUI controls).

The shader program is compiled to a virtual instruction set which includes some common functions for video processing. For example, a few colorspace conversion instructions are available.

During compilation, the maximum stack size required is estimated under the assumption that values will not be pushed onto the stack continuously in a loop. This assumption holds for a typical C like program. Thus the number of push instructions provides the maximum likely stack size required, although the program in unlikely to use all the stack entries. Registers are also allocated to each parameter and local variable. We have not yet implemented an optimizing compiler, so variables cannot share registers. The compiler computes the number of registers required and stores this information in the virtual machine.

4 Efficiency

Virtual machines are generally a lot slower to execute than native implementations. However, each pixel can be computed entirely independently of adjacent pixels. It is thus possible to compute an entire frame, each pixel computed in parallel with the others. At any point in time the state of the computation for each pixel must be stored. To improve the efficiency of the virtual machine, each virtual machine instruction is executed on an entire frame, and not on individual pixels (in a similar fashion to the REYES technique [Cook et al. 1987]). As a result most operations are performed at near native speed. The state of the computation is stored on a stack, where each entry of the stack stores some state for every pixel in the frame. For example, the `ADD` instruction of the virtual machine will add two values for every pixel of the frame. These values will most likely differ for each pixel.

At each step of execution the instruction pointer is advanced through the program. Conditional jumps would cause the instruction pointer to refer to different instructions for different parts of the frame. To avoid this situation, no conditional jumps are available. Instead a mask is introduced [Cook et al. 1987] to determine which pixels are actually executing the instructions. We simply prevent write operations for pixels that are masked out (not executing). For example, if an `if` statement is executed, then both the true part and the false part are executed for all pixels. However, the mask will be set during the true part so that only pixels satisfying the condition will change any values. The mask is then inverted for execution of the false part. A fragment of virtual machine code to execute an if statement is as follows:

```
'evaluate condition
PUSHMASK
MASK
'evaluate true statements
POPMASK
NOT
PUSHMASK
MASK
'evaluate false statements
POPMASK
```

We use floating point values to represent pixel data throughout the program. As a result we have excellent accuracy and the ability to represent colors outside of the normal display range for each stage of the video. The colors are clamped just before the final frame is encoded. Although the floating point values add to the expressive power of the shading language, the shader executes slightly more slowly as a result.

Although we process an entire frame at a time, some values in the shader are not dependent on the position in the frame. We call such values *uniform* values [Cook et al. 1987]. Values which vary based on the position in the frame are *varying* values. We need only one value to represent a uniform value, whereas for varying values we store a value for each pixel in the frame. The compiler automatically creates uniform values for constants, and the virtual machine automatically converts to varying values as necessary. Thus a shader producing a constant color will never produce a varying output. However, interactions with varying parameters automatically result in varying answers. For example, adding a varying value to a uniform value can only give a varying answer. If the value for each pixel happens to be the same, the value remains a varying value. We have no way to determine that this is the case other than examining each pixel (and this would be too expensive).

Simple shaders execute in approximately 6 times the time it takes to perform only decompression of a stream, followed by recompression of the stream. Although the performance is below what would be desired, the execution speed is still much faster than would be achieved if the virtual machine executed individually on each pixel.

We initially used bilinear interpolation for the lookup of pixels in frames, but the software implementation caused performance to deteriorate significantly.

5 Limitations

Frames are always delivered in sequence, one at a time. Thus differences between frames cannot be computed. One way to bypass this problem is to create two identical streams and to delay the one stream. An additional restriction, is that streams cannot be reused unless they have exactly the same timing. For example, if we have a video stream for which we wish to compute motion vectors, the streams have to be duplicated, we cannot take one stream and put it through two paths where one is delayed and the other is not. In this case the delayed stream will simply miss the initial frames of the stream. To partially address this problem, the CPU implementation allows access to the previous frame in the stream, but access to other frames is not supported. The GPU implementation does not support access to the previous frame.

Since new streams are instantiated to effect delays, we have additional decoding overhead. The streams are decoded multiple times. A possible future modification may allow caching of frames, but the memory overhead for delayed streams (without additional decoding) may be prohibitive.

Profiling of the video editing system reveals that the greatest amount of time is spent constructing and copying objects that represent color. Execution speed can be improved by rewriting the virtual machine instructions to avoid the use of constructors and copying of objects as much as possible. Since the virtual machine is stack based, most operations destroy items on the stack and create a new item. In many situations the last item on the stack can be replaced by the answer instead of being destroyed and recreated.

6 GPU Implementation

Although the CPU based implementation is fairly slow, GPU's offer the potential for significant speed increases. The GPU shader model is very similar to our video shader model, so a GPU implementation of our video shader is straight forward. The GPU enables several pixels to be processed in parallel, since the shader specifies only the color of a pixel based on the provided input frames. The GPU also has all the facilities required to deal with branching within the shader programs.

The GPU implementation accepts exactly the same shader language, and the same specification of video editing operations through tree structures. The transition shader is converted to a GLSL fragment program which can be executed on the GPU. The program creates a texture for each video stream, that will hold the current frame for that stream. The texture is then updated as each new frame is delivered.

For example, the fadein shader (see examples section)

```
stream fadein(frame f1, float start,
        float end) {
    if (t<start) out=vector(0.0, 0.0, 0.0);
    else if (t>end) out=lookup(f1, p);
    else {
        float f=(t-start)/(end-start);
        out=f*lookup(f1, p);
    }
}
```

is translated to (some of the GLSL shader is omitted):

```
vec3 lookup(sampler2D tex, vec3 pos) {
    if (pos.x<0.0) return vec3(0.0,0.0,0.0);
    if (pos.x>1.0) return vec3(0.0,0.0,0.0);
    if (pos.y<0.0) return vec3(0.0,0.0,0.0);
    if (pos.y>1.0) return vec3(0.0,0.0,0.0);
    return texture2D(tex, pos.xy).rgb;
}

uniform float t;
varying vec3 p;
uniform sampler2D f1;
uniform float start;
uniform float end;
void main(void) {
    vec3 out_put=vec3(0.0, 0.0, 0.0);
    if (t<start)
        out_put=vec3(0.0,0.0,0.0);
    else
        if (t>end)
            out_put=lookup(f1,p);
        else {
            float f=(t-start)/(end-start);
            out_put=f*lookup(f1,p);
        }
    gl_FragColor.rgb=out_put;
    gl_FragColor.a=1.0;
}
```

Parameters for the shader are passed through as uniform GLSL variables. Several functions (such as lookup) are provided at the start of the GLSL fragment program. The program automatically determines the uniform variables required and sets the uniform variables according to the parameters in the shade tree. Frames required as input are automatically allocated to texture units and corresponding uniform samplers in the shader are set accordingly.

Shader programs render their output to the framebuffer, which is then used to update the texture without transferring the data to main memory. A side effect of this rendering process, is that all color values are clamped at each stage of execution of the shade tree (unlike the CPU implementation). At this time, all textures are 8-bit RGB textures. We may add floating point textures in the future. Although we do not have the advantage of floating point textures, hardware accelerated bilinear filtering is available.

Once the final output frame is rendered, the program reads the frame from the framebuffer and encodes the frame.

7 Example shaders

We begin with a shader that produces a constant color:

```
stream constant(float r, float g, float b)
    describe r as "Red";
    describe g as "Green";
    describe b as "Blue";
{
    out=vector(r, g, b);
}
```

Note the use of the `describe` keyword to provide hints for a graphics interface. The variable `out` is always used to store the return value from the shader.

The next shader copies the input stream exactly (although the width and height may differ).

```
stream copy(frame f1)
{
    out=lookup(f1, p);
}
```

The variable `p` is an implicit varying variable that specifies the pixel position in the frame. The `frame f1` parameter specifies that one stream is expected as input for this shader. `lookup` is a function that looks up a pixel in a frame.

Now we begin to look at more complex shaders, `fadein` fades a video stream (which can be a picture) in from black:

```
stream fadein(frame f1, float start,
        float end) {
    if (t<start) out=vector(0.0, 0.0, 0.0);
    else if (t>end) out=lookup(f1, p);
    else {
        float f=(t-start)/(end-start);
        out=f*lookup(f1, p);
    }
}
```

The implicit uniform variable `t` is the time in seconds since the start of the output stream.

The traditional cross-fade transition is implemented as follows:

```
stream crossfade(frame f1, frame f2,
        float start, float end)
    describe start as "Start time for fade";
    describe end as "End time for fade";
{
    float factor;
    factor=(t-start);
    factor=factor/(end-start);
    if (factor<0.0) factor=0.0;
    if (factor>1.0) factor=1.0;
    out=(1.0-factor*lookup(f1, p))+
        factor*lookup(f2, p);
}
```

Chroma keying can also be implemented. In the following shader, we overlay an image (or video stream) over another video stream:

```
stream overlay(frame f1, frame f2)
{
    float blend=lookup(f2, p);
    if (blend.y<1e-3)
        out=lookup(f1, p);
    else
        out=lookup(f1, p)*(1.0-blend.y)+blend;
}
```

The `y` field of the color gives the luminance, or grayscale intensity of the pixel.

Positional transforms are also possible, by offsetting the lookup position in the incoming stream. The following shader slides a stream into the visible frame using a quadratic function to slow the speed down as the frame becomes fully visible.

```
stream slidein(frame f1, float start,
    float end)
{
    float ofs=(t-start)/(end-start);
    if (t<start) out=vector(0.0, 0.0, 0.0);
    else {
        if (t>end) ofs=1.0;
        ofs=2*ofs-ofs*ofs;
        ofs=1.0-ofs;
        out=lookup(f1,p+vector(0.0,ofs,0.0));
    }
}
```

We can also scale streams to create a selection screen (for an interactive video):

```
stream boxes(frame f1, frame f2, frame f3,
    frame f4)
{
    out=vector(0.0, 0.0, 0.0);
    if ((p.vx>0.1)&&(p.vx<0.45)&&
        (p.vy>0.1)&&(p.vy<0.45))
        out=lookup(f1, (p-
            vector(0.1, 0.1, 0.0))/0.35);
    if ((p.vx>0.55)&&(p.vx<0.9)&&
        (p.vy>0.1)&&(p.vy<0.45))
        out=lookup(f2, (p-
            vector(0.55, 0.1, 0.0))/0.35);
    if ((p.vx>0.1)&&(p.vx<0.45)&&
        (p.vy>0.55)&&(p.vy<0.9))
        out=lookup(f3, (p-
            vector(0.1, 0.55, 0.0))/0.35);
    if ((p.vx>0.55)&&(p.vx<0.9)&&
        (p.vy>0.55)&&(p.vy<0.9))
        out=lookup(f4, (p-
            vector(0.55, 0.55, 0.0))/0.35);
}
```

The `lookup` function accepts a floating point vector for the pixel position, so scaling the vector `p` provides an appropriate position. This shader places 4 streams in a grid arrangement with spaces between. The field `vx` and `vy` are synonyms for `r` and `g` respectively which are used to emphasize that `p` is used for positional purposes.

Motion blur can be implemented with the following shader:

```
stream motionblur(frame f1, frame f2,
    frame f3, frame f4, frame f5,
    frame f6, frame f7, frame f8)
{
    out=lookup(f1, p);
    out=0.5*out+0.5*lookup(f2, p);
    out=0.5*out+0.5*lookup(f3, p);
    out=0.5*out+0.5*lookup(f4, p);
    out=0.5*out+0.5*lookup(f5, p);
    out=0.5*out+0.5*lookup(f7, p);
    out=0.5*out+0.5*lookup(f8, p);
}
```

Each of the frames should come from the same stream, where each stream is delayed by some amount.

Rotation of a video stream is accomplished with:

```
stream rotate(frame f1, float speed)
{
    float x=p.vx-0.5;
    float y=p.vy-0.5;
    float nx=x*cos(t*speed)-y*sin(t*speed);
    float ny=y*cos(t*speed)+x*sin(t*speed);
    vector l=vector(nx+0.5, ny+0.5, 0.0);
    out=lookup(f1, l);
}
```

We conclude with an example specification of a script that fades a title in, then fades the title out while sliding a video stream in at the same time:

```
    fps=25;
    width=512;
    height=512;
    aspect=1.0;
    bitrate=12000;
    shadows=stream("shadows.ogg") delay=3.0;
    title=image("shadow.png");
    fi=fadein(title, 0.0, 2.0);
    fo=fadeout(title, 2.0, 4.0) delay=2.0;
    fio=concat(fi, fo, 2.0);
    slide=slidein(shadows, 3.0, 4.0) delay=3.0;
    out=overlay(slide, fio, 0.5)
        from=0.0 to=20.0;
```

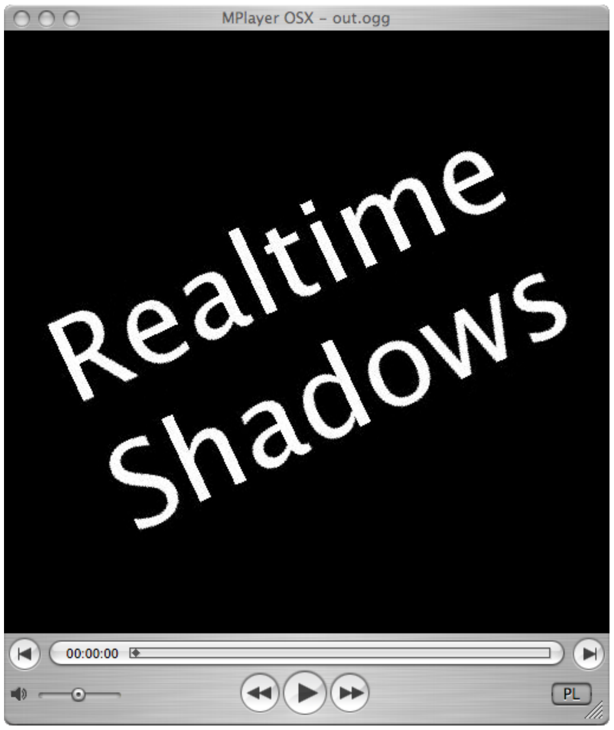

Figure 2: *Rotation of stream by shader.*

Examples of the output of these scripts are shown in figures 2, 3 and 4. Please see the accompanying videos for these examples.

8 Shade Tree Example

In this section we construct a more elaborate example to demonstrate how the shade tree is created. Consider a presenter of a news program. The presenter reads the news, while a snapshot of the video insert is showed in the top right corner. Text is provided at the bottom of the frame that describes the story that is being reported on or the person that is speaking. The picture zooms in on the insert to provide the full report.

We now construct the tree that will achieve this task. First the insert is loaded with

```
insert=stream("insert.ogg");
```

Next the insert is scaled and placed with a custom shader

```
pinsert=zoom(insert, 0.65, 0.05, 0.3,
    0.3, 5.0, 5.8);
```

The presenter is loaded with

```
presenter=stream("presenter.ogg");
```

Now the insert is overlaid over the presenter

```
reportpart=overlay(presenter, pinsert);
```

The report text is loaded

```
titletext=image("report.png");
```

and overlaid over the video:

```
out=overlay(reportpart, titletext)
```

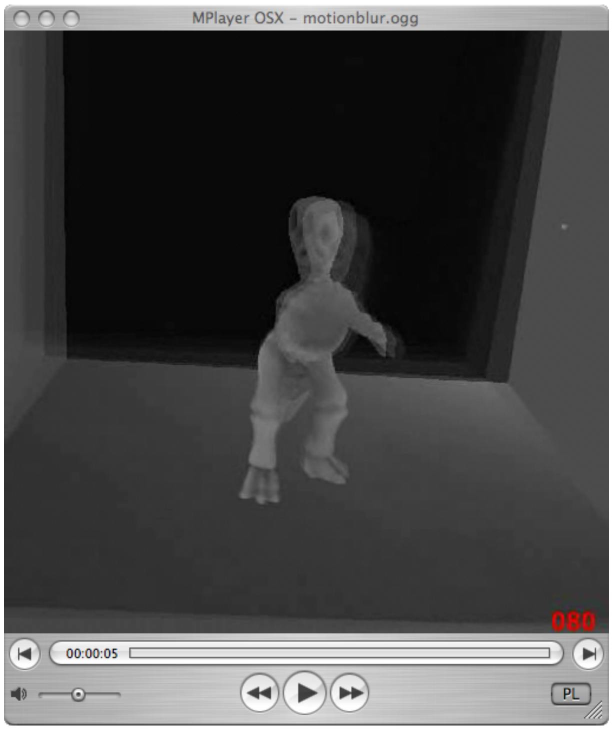

Figure 3: *Motion blur.*

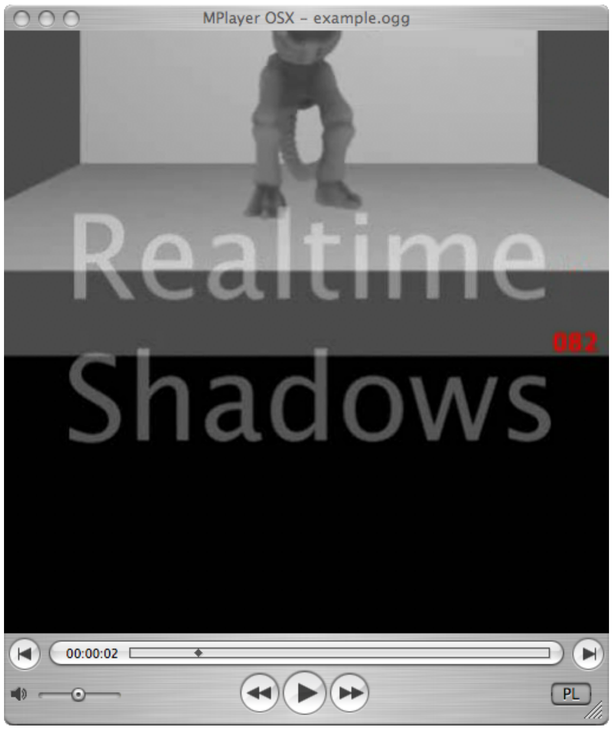

Figure 4: *Fade out and slide in.*

```
from=0 to=20.0;
```

We need the `zoom` shader to perform the editing operation:

```
vector movescale(frame f, float x,
   float y, float w, float h)
{
   if (p.vx<x) out=vector(0.0,0.0,0.0);
   else if(p.vy<y) out=vector(0.0,0.0,0.0);
   else if(p.vx>x+w) out=vector(0.0,0.0,0.0);
   else if(p.vy>y+h) out=vector(0.0,0.0,0.0);
   else out=lookup(f, (p-vector(x, y, 0.0))*
      (vector(1.0/w, 1.0/h, 0.0)));
}

stream zoom(frame f, float x, float y,
   float w, float h, float start, float end)
{
   float z=(t-start)/(end-start);
   if (z<0.0) z=0.0;
   if (z>1.0) z=1.0;
   out=movescale(f, x*(1.0-z), y*(1.0-z),
       z+(1.0-z)*w, z+(1.0-z)*h);
}
```

The final tree is illustrated in figure 5, and example output of the video editing application is shown in figure 6.

9 Results

The CPU and GPU implementations of the video editing system were executed on an AMD Athlon 2600 system with a Radeon 9600 video card. The number of registers, maximum stack size, and number of instructions for several shaders are listed in table 1 for the CPU implementation. For most shaders the stack size and instruction count is quite low. The stack size and registers provide an upper bound on the memory that can be used by the shader. In the worst case, all registers and stack entries are varying. The upper bound of memory usage is $(\textit{stacksize} + \textit{registers}) \times \textit{framesize}$. For example, the upper bound for a 512×512 frame produced by the rotate script is $(23 + 10) \times 512 \times 512 \times 3 \times 4 = 99Mb$ for a floating point frame with red, green and blue components.

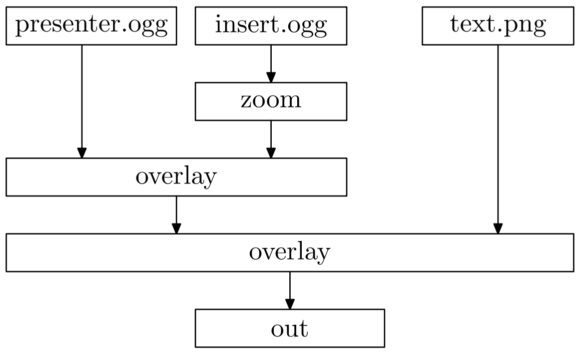

Figure 5: *A shade tree for a news program.*

Next we compared several video editing operations on the CPU and GPU. Results are given in table 2. Neither the CPU or GPU implementations can manage realtime editing on the test system. Reading the framebuffer is quite an expensive task, so the copying of a video stream to a large video stream (as in the resize example) is faster on the CPU than on the GPU. When the video resolution is lower and processing is more of a factor, the GPU implementation outperforms the CPU implementation.

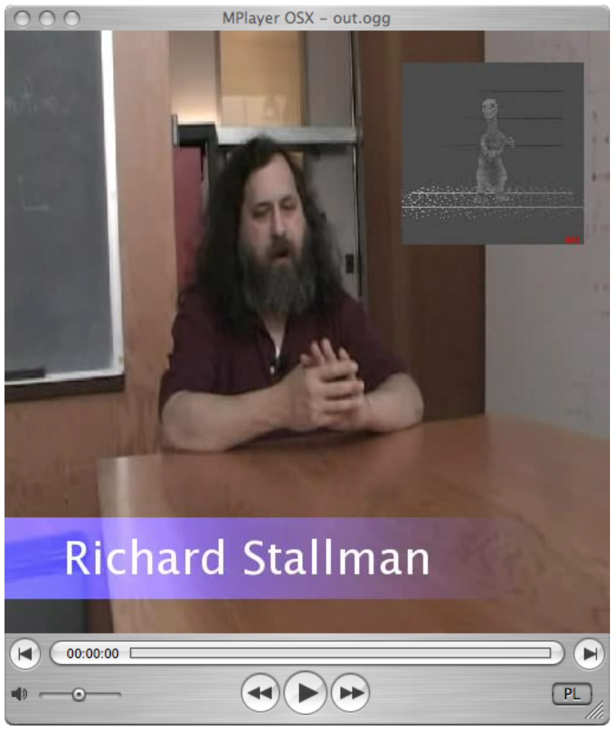

Figure 6: *Example output of the news caster script.*

Shader	Stack	Registers	Instructions
fadein	18	7	93
fadeout	20	7	105
concat	7	6	37
overlay	12	7	67
slidein	25	7	134
rotate	23	10	121
boxes	63	7	357

Table 1: *Registers, stack size and instruction count for various shaders.*

10 Conclusions and future work

We have presented a tool for the implementation of video transitions and filters. The examples have demonstrated the simplicity of the system and the ability to implement many common transitions. The shaders are applied by constructing a tree of video streams, which allows multiple transition effects to be implemented in a single segment of video.

The potential of using shade trees for video editing must still be investigated thoroughly. A comprehensive user study will demonstrate the expressive power of using shade trees for video editing, and may highlight the usability aspects of the programmable transition shaders.

Subtrees provide a definition for producing a stream. A further future avenue of research will investigate the use of templates to specify video streams by constructing subtrees from the templates.

References

ANSARI, M. Y., 2003. Video image processing using shaders. ATI ATI Sponsored Session. GDC 2003.

Task	Duration	CPU	GPU
Resize to 1024 × 1024	20	218	552
Fadein, Fadeout, Slidein	20	552	191
Rotation (512 × 512)	10	74	64
Boxes (512 × 512)	25	922	268

Table 2: *Execution time for various video editing tasks. Times are measured in seconds. Duration is the duration of the output stream.*

BOLZ, J., FARMER, I., GRINSPUN, E., AND SCHRÖODER, P. 2003. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, ACM, New York, NY, USA, 917–924.

BUCK, I., FOLEY, T., HORN, D., SUGERMAN, J., FATAHALIAN, K., HOUSTON, M., AND HANRAHAN, P. 2004. Brook for GPUs: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, ACM, New York, NY, USA, 777–786.

CASARES, J., LONG, A. C., MYERS, B. A., BHATNAGAR, R., STEVENS, S. M., DABBISH, L., YOCUM, D., AND CORBETT, A. 2002. Simplifying video editing using metadata. In *DIS '02: Proceedings of the 4th conference on Designing interactive systems*, ACM, New York, NY, USA, 157–166.

COOK, R. L., CARPENTER, L., AND CATMULL, E. 1987. The Reyes image rendering architecture. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 95–102.

COOK, R. L. 1984. Shade trees. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 223–231.

DAVIS, M. 2003. Editing out video editing. *IEEE MultiMedia 10*, 2, 54–64.

GIRGENSOHN, A., BORECZKY, J., CHIU, P., DOHERTY, J., FOOTE, J., GOLOVCHINSKY, G., UCHIHASHI, S., AND WILCOX, L. 2000. A semi-automatic approach to home video editing. In *UIST '00: Proceedings of the 13th annual ACM symposium on User interface software and technology*, ACM, New York, NY, USA, 81–89.

MATTHEWS, J. W., MAKEDON, F., AND GLOOR, P. 1994. VideoScheme: A research, authoring, and teaching tool for multimedia. In *Educatinal Media and Hypermedia*, 385–390.

MENG, J., AND CHANG, S.-F. 1996. CVEPS - a compressed video editing and parsing system. In *MULTIMEDIA '96: Proceedings of the fourth ACM international conference on Multimedia*, ACM, New York, NY, USA, 43–53.

OLANO, M., AND LASTRA, A. 1998. A shading language on graphics hardware: the pixelflow shading system. In *SIGGRAPH '98: Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, ACM, New York, NY, USA, 159–168.

UEDA, H., AND MIYATAKE, T. 1996. Automatic scene separation and tree structure GUI for video editing. In *MULTIMEDIA '96: Proceedings of the fourth ACM international conference on Multimedia*, ACM, New York, NY, USA, 405–406.