6. References

[1] *Avid VideoShop*. Avid Technology Inc. Tewksbury, MA.
[2] *Data Explorer* software package. IBM.
[3] J. D. Foley, et. al., Computer Graphics: Principles and Practice, second edition, Reading, Mass. Addison-Wesley, 1990.
[4] Holzmann, Gerald J. *Popi*. AT&T Bell Laboratories. Murray Hill, NJ.
[5] D. Le Gall, MPEG: a video compression standard for multimedia applications, Communications of the ACM, April 1991, Vol. 34, Num.4, pp. 46-58.
[6] Matthews, James, Peter Gloor, and Fillia Makedon. "VideoScheme: A Programmable Video Editing System for Automation and Media Recognition." ACM Multimedia '93 Proceedings, pp. 419-426.
[7] W. B. Pennebaker, JPEG still image data compression standard, Van Nos and Reinhold, New York, 1992.
[8] Shantzis, Michael. "A Model for Efficient and Flexible Image Computing." SIGGRAPH '94 Proceedings, pp. 147-154.
[9] Ousterhout, John K. *Tcl and the Tk Toolkit*. Addison-Wesley, Massachusetts, 1994.
[10] Rasure and Kubica, "The Khoros Application Development Environment", Experimental Environments for Computer Vision and Image Processing, editor H.I Christensen and J.L Crowley, World Scientific 1994.
[11] Silicon Graphics Inc. *ImageVision Library.* Silicon Graphics Inc., Mountain View, CA.

sequences: a sequence frame is only computed if it appears in the output sequence. Thus, if we apply effects to all the frames in a sequence, and then cut most of the sequence, only the frames that remain will need to be processed.

- *Graph modifications.* The second optimization modifies the graph so its output is equivalent, but the computation is more efficient. Such modifications include combining or swapping adjacent nodes. For example, an affine transformation (rotation, scale, translation) is characterized by a 3x3 matrix. Any adjacent affine nodes can be combined into a single node by multiplying their matrices together.

  Graphs can be optimized for speed of computation or quality of output. This option allows Rivl users to prototype operations at low quality and then compute the result off-line at high quality once the program is debugged.

- *Smart caching.* Since the graph allows us to determine the exact order of image computations in advance, we know how many times an image will be used, and when it is safe to throw it away. Thus Rivl can keep an image in memory for the optimal length of time. Furthermore, if there are too many images in memory at once, we can page images to disk using an optimal virtual memory policy because we know every future image reference. An implementation of the latter idea is in progress.

4. Experiences

On the whole, our experience with Tcl as a platform for Rivl has been favorable. Tcl's extensibility makes it easy to build new video processing commands from Rivl. Tcl's interpreted nature allows for instant feedback on the results of new video effects, facilitating experimentation and fine-tuning. Access to Tk is another advantage. We are currently working on a Tk video editor that will use an embedded Rivl interpreter for video processing.

We encountered a few Tcl-related problems in our work. The most bothersome syntactic issue was the lack of transparent pass-by-reference. Recall that most Rivl commands come in two flavors: destructive (the argument is modified) and non-destructive (the result is returned). To implement destructive operations in Tcl, the name of the variable must be passed in without the leading $. The result is an inconsistency in syntax, as the following equivalent lines of code show:

```
im_scale! image 0.5
set image [im_scale $image 0.5]
```

Predictably, a common error is to mistakenly insert or omit a dollar sign.

5. Related and Future Work

Many commercial packages are available that provide software libraries of image manipulation functions. Some use demand-driven execution to achieve similar optimizations as those mentioned in section 4. These include the Pixar system described by Shantzis[8], and Silicon Graphics' ImageVisionLibrary[11]. Holzmann's Popi[4] allows image transformations to be specified with concise expressions at run-time, a mechanism that permits rapid prototyping of new image effects. We are adapting this idea to Rivl.

Matthews, Gloor, and Makedon's VideoScheme [6] combines Apple's QuickTime movie player with a Scheme-based video manipulation language. In VideoScheme the user works with objects close to the underlying implementation of video data, such as pixel arrays and frames. This low-level access gives users considerable flexibility in creating new image operations. For example, algorithms for detecting "cuts" in video can be easily built out of pixel array primitives. In contrast, Rivl's high level of abstraction allows it to exploit delayed computation for improved efficiency, and its resolution independence makes programs more portable.

Rivl is implemented with 4000 lines of C code and 500 lines of Tcl code. It has been ported to the Sun OS, HPUX, and Linux operating systems. We plan to release Rivl version 1.0 in early summer 1995.

The Rivl language is still evolving. We are extending the core set of Rivl primitives to support other types of video processing, such as image analysis and vision, text titling, and morphing. We are also building a Tk video editor on top of Rivl. The editor provides similar features to other video editors such as Avid's VideoShop[1]. Users can edit multiple sequences, perform assembly editing, and apply video effects in a convenient manner. But since the editor is based on Rivl, users can exploit the power of the Rivl interpreter to define effects and perform tasks beyond the scope of the graphical user interface.

seconds) to apply the transition effect. For example, `connectWithTransition {im_fade %1 [expr 1-%p]} $jack $jill 5` connects two sequences `jack` and `jill` with a five second fade.

3. Efficiency Considerations

This section discusses efficiency considerations involved in implementing Rivl. Efficiency is always an issue for image processing because of the large amount of data to process. The problem is compounded for video streams, which have many images for each second of footage. There are two ways to optimize image computing. First, we must make sure that individual image operations, such as scales, rotations, etc., are efficient. These issues have been addressed at length in the graphics literature, and good algorithms are readily available[3]. Second, we must be intelligent about which operations we call, in what order, to achieve our final result.

A feature of Rivl that allows us to exploit the second form of optimization is *lazy evaluation,* also known as *demand-driven execution*[8]. Rivl only computes video data when it is needed for output or display. The result is that at computation time, Rivl can plan a more intelligent computing strategy than if each command were executed immediately and independently.

Consider the following Rivl session:

```
% rivl
> set flowers [im_read flowers.jpg]
rvl_image2
> fractal! flowers 25
rvl_image9
> im_write images/newflowers.ppm
>
```

Objects such as `rvl_image2` and `rvl_image9` are actually image *proxies*. A proxy contains the dimensions of an image, but no actual pixel data. As we call image operations with proxies, Rivl builds up a *directed graph* representing the operations to be executed without actually computing the results of the image operators. In effect, the graph is a dynamic trace of the Rivl program execution. For example, the first two lines of the above session result in the graph shown in figure 4.

Every edge in the graph represents an image, and every node represents a primitive image operation. Having this graph provides us with several opportunities for optimization. Three optimizations are described below.

- *Computing only what you need.* In the process of computing an output image, we often compute several intermediate images. However, not all the pixels of an intermediate image necessarily affect the result. For example, an image might be scaled, rotated, and smoothed, only to be mostly obscured in a later overlay operation. Rivl uses an algorithm described by Shantzis [8] that guarantees that only those regions of an image which affect the output are computed. We apply the same idea to

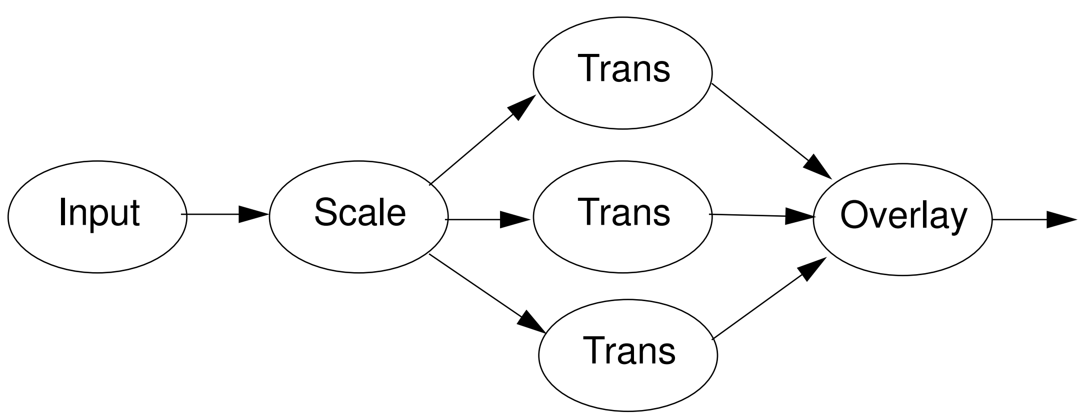

Figure 4: Graph produced by one iteration of `fractal`

translating images in the spatial dimension. `Seq_speedup` changes the speed of a movie, `seq_reverse` reverses a movie, and `seq_shift` shifts the sequence in time.

Rivl has primitives to convert between sequences and lists of images, and to apply an image operator to every image in a sequence. The operators `seq_to_ims` and `ims_to_seq` convert a sequence to a list of images and back. `Seq_map` executes a given script for each image of a sequence and assembles the resulting images into a new sequence. `Seq_map` is similar to `map` in Scheme. Consider the command

```
seq_map $clip {im_fade %1 0.5}
```

`Seq_map` substitutes the handle of the current image wherever `%1` appears in the script. Thus, this command returns a new sequence containing the images in `clip` faded by half.

Sometimes, rather than applying the exact same effect to each image, it is desirable to vary the effect over time. For example, consider the fade-to-black effect. This effect can be achieved by calling `im_fade` on each image in the sequence with a parameter that decreases over time.In this case, `seq_map` must call a procedure with a parameter that indicates the time of the image being modified. To this end, `seq_map` performs the following additional substitutions:

- `%t`: Substitute the time stamp of the current image, in seconds
- `%l`: Substitute the length of the sequence in seconds
- `%p`: Substitute the relative time of the current image: `%t` divided by `%l`

Using this mechanism, fade-to-black can be expressed

```
seq_map $clip {im_fade %1 [expr 1-
%p]}
```

If `clip` is 21 frames long, this expands into the following series of commands:

```
im_fade image1 1.00
im_fade image2 0.95
im_fade image3 0.90
...
im_fade image20 0.05
im_fade image21 0.00
```

The resulting images are concatenated into a new sequence, resulting in the desired fade-out visual effect.

When combined with sequence assembly operations, `seq_map` simplifies the expression of effects that are often used in transitions between two parts of a movie. For example, the Rivl procedure connectWithTransition, shown in figure 3, connects two sequences with a transition. The first parameter, `transition`, is a script to be passed to `seq_map`. `MovieA` & `movieB` are the two sequences to be joined, and `duration` is the time (in

```
proc connectWithTransition {transition movieA movieB duration} {
    set lengthA [seq_length $movieA]
    set lengthB [seq_length $movieB]

    # Untouched parts of first and second movie
    set begin [seq_crop $movieA 0.0 [expr $lengthA-$duration]]
    set end [seq_crop $movieB $duration $lengthB]

    # Apply timed effect to end of first movie; overlay with
    # beginning of second movie
    set mid1 [seq_crop $movieA [expr $lengthA-$duration] $lengthA]
    set mid2 [seq_crop $movieB 0.0 $duration]
    set middle [seq_overlay [seq_map $transition $mid1] $mid2]

    seq_concat $begin $middle $end
}
```

Figure 3: `connectWithTransition` procedure

Since the full Tcl language is available, Rivl procedures can be constructed that include looping, branching, and recursion. For example, suppose the following sequence of operations is applied to an image `n` times:

1. Shrink the image to half size.
2. Create three duplicates of the image. Move them "north", "southwest", and "southeast" respectively.
3. Merge the three duplicates into a single image.

In the limit of large `n`, a fractal (Sierpinski's Gasket) is created. The corresponding Rivl program is given in figure 2. The first two lines compute the translation distance `dx` and `dy` for the second step as a function of the image size. The rest of the procedure follows from the description given above. Figure 2 shows sample results for several values of `n`.

2.2 Sequences

A sequence, the Rivl abstraction for video, can be thought of as a set of time-stamped images. Table 2 lists four classes of sequence primitives in Rivl. The first class provides for sequence input/output. The second class provides operations to split and join sequences (called sequence *assembly* operations.) The third class allows scaling and translation of sequences in the time dimension. And, the final class provides a bridge between sequences and images.

Like image commands, sequence commands can be composed to create procedures. For example, consider the following Rivl procedure:

```
proc preview {movieList n} {
    # Create an empty sequence
    set result [seq_new]
    foreach movie $movieList {
       # Grab segment from next movie
       set chunk [seq_crop $movie 0.0
$n]
       # Concatenate segment onto
result
       seq_concat! result $chunk
    }
    return $result
}
```

`Preview` extracts `n` seconds off the beginning of each movie in a list of movies. The call to `seq_new` creates an empty sequence for the return value. The loop repeatedly extracts the first `n` seconds from each sequence in `movieList` and appends it to `result`, which is returned.

Sequences can also be scaled and shifted in the time dimension, operations that are analogous to scaling and

seq_read filepat	Read sequence from MPEG or multiple image files
seq_write seq filepat	Write sequence to disk
seq_crop seq begin end	Crop the specified range to make a new sequence
seq_concat seq seq ...	Concatenate multiple sequences end to end
seq_overlay seq seq ...	Overlay the sequences to form one composite
seq_speedup seq factor	Speed up or slow down sequence
seq_shift seq delta	Shift sequence in time
seq_reverse seq delta	Reverse sequence in time
seq_to_ims seq	Return a list of images by sampling the sequence
ims_to_seq imagelist	Construct a sequence from the list of images
seq_map seq script	Apply a script to each image in the sequence

Table 2: Sequence Primitives

Many Rivl procedures, like `whirlpool`, apply a sequence of operations to an image. Since the operators in table 1 are all non-destructive, these procedures consist of a sequence of statements of the form

```
set im [im_effect $im params]
```

We found this notation cumbersome, so we borrowed an idiom from Scheme. Any operator with the character "!" appended destructively modifies its first argument. This rule applies to all operators listed in table 1. Taking advantage of this notation, we can write `whirlpool` as:

```
proc whirlpool {image p} {
    im_scaleC! image [expr 1 - $p]]
    im_rotateC! image [expr 360 * $p]
    return $image
}
```

Notice that the destructive operation omits the "$" in front of its first argument, whereas the non-destructive form requires the "$". This artifact is caused by the way pass-by-reference is implemented in Tcl, a point we discuss in more detail in section 4.

For consistency, we felt that user defined effects should also have destructive and non-destructive versions. To this end we provide the `rvl_proc` command, which is similar to Tcl's `proc` command but also creates a destructive form of the procedure. Thus, if we use `rvl_proc` in the definition of `whirlpool` above, two commands, called `whirlpool` and `whirlpool!`, are created.

```
rvl_proc fractal {image n} {
      set dx [expr 0.25 * [im_width $image]]
      set dy [expr 0.25 * [im_height $image]]
      for {set i 0} {$i < $n} {incr i} {
            # 1. Shrink to half size.
            im_scaleC! image 0.5

            # 2. Create and move three duplicates.
            set north [im_trans $image 0 -$dy]
            set sw [im_trans $image -$dx $dy]
            set se [im_trans $image $dx $dy]

            # 3. Merge the three duplicates.
            set image [im_overlay $north $sw $se]
      }
      return $image
}
```

Figure 2: `fractal` program and output for n= 1,2,3

2.1. Images

Table 1 lists three classes of image primitives in Rivl. The first class provides for image input/output, the second class provides geometric operations on images, such as translation, rotation, and scaling, and the third class provides miscellaneous operations including cropping, overlaying, and fading.

The commands in table 1 can be used in Tcl procedures to create more complex visual effects. For example, consider the following Tcl procedure:

```
proc whirlpool {image1 p} {
   set image2 [im_scaleC $image1
[expr 1 - $p]]
   set image3 [im_rotateC $image2
[expr 360 * $p]]
   return $image3
}
```

This procedure takes two arguments: a Rivl image (`image1`) and a double value between 0 and 1 (`p`). The call to `im_scaleC` reduces `image1` by a factor of 1-`p`, and assigns the result to `image2`. The call to `im_rotateC` rotates `image2` about its center by an angle proportional to `p`, and stores the result in `image3`, which is returned. Figure 1 shows examples of whirlpool with several values of `p`.

im_read file	Read an image file (ppm, pgm, jpeg)
im_write image file	Write the image to a file
im_trans image dx dy	Translate an image
im_rotate[C] image theta	Rotate an image; C = around its center
im_scale[C] image s	Scale an image; C = around its center
im_crop image x1 y1 x2 y2	Crop the specified rectangle to make a new image
im_overlay image image ...	Overlay images
im_fade image factor	Darken the image according to factor
im_convolve image kernel	Convolve the image with specified kernel

Table 1: Image primitives

Figure 1: Output from `whirlpool` for p = 0.1, 0.4, 0.7

RIVL: A Resolution Independent Video Language

Jonathan Swartz
Brian C. Smith
Department of Computer Science
Cornell University
{swartz,bsmith}@cs.cornell.edu

Abstract

As common as video processing is, programmers still implement video programs as manipulations of arrays of pixels. This paper presents an extension to Tcl called Rivl (pronounced "rival") where video is a first class data type. Programs in Rivl use high level operators that are independent of video resolution and format, increasing portability of programs and allowing rapid prototyping of video effects. This paper gives several examples of still-image and video sequence programs in Rivl. It also discusses efficiency issues and experiences with Tcl as a platform for Rivl.

1. Introduction

The incorporation of video into our computing environment will change the way we interact with computers as much as the shift from alphanumeric terminals to graphical user interfaces. To realize this vision, video must become as accessible in our computing environment as text and images are today. Because video has different semantic, storage, and timing requirements, the realization of truly programmable video requires research in storage systems, transport protocols, compression methods, and algorithms.

The way we encode video algorithms today is similar to the way we expressed numerical algorithms in the days of assembly language. Video is composed of a sequence of images, each of which is represented as a two dimensional array of pixel values. In the past, floating point operations were expressed by manipulating individual bits. Today, video and image operations are expressed by manipulating pixel values. Some systems (e.g., Data Explorer[2] or Khoros[10]) provide a graphical programming environment where programs are expressed as flowcharts. Although this is an improvement, the limitations of flowcharts for expressing complex programs are well-known.

What is needed is a language that incorporates video as a first class data type, just as floating point numbers are first class data types in almost all modern programming languages. Thus, just as the floating point addition "A+B" is well-defined regardless of whether A and B are single, double, or even quad-precision floating point numbers, the operation "cut the first five seconds of the video clip" is well defined whether the film resolution is 16 or 30 frames per second, whether the format is MPEG[5] or motion JPEG[7], and whether the image size is 100x100 or 6000x4000. This paper describes one such language, called *Rivl* (pronounced "Rival"). In Rivl, video operations are expressed independent of the internal representation of video data. Just as it is the responsibility of traditional languages to map a floating-point operation onto the underlying bit manipulations, it is Rivl's responsibility to map image and video clip operations onto the underlying pixel and frame manipulations.

Rivl is currently implemented as an extension to the Tcl language [9]. This paper describes the design and implementation of Rivl, and discusses our experiences with the language. The rest of the paper is organized as follows. Section 2 illustrates the Rivl language through a series of examples. Section 3 discusses efficiency considerations for video languages. Section 4 discusses our experiences with Tcl as a language platform for Rivl. Section 5 reviews related work, and concludes with current status and future research directions.

2. Rivl: The Language

This section illustrates Rivl programs through a series of examples. The intention is to give the reader a feel for the flavor of Rivl programs and an idea of the expressive power of the language, rather than providing a reference manual or an exhaustive catalog of Rivl's functions.

Since Rivl is an extension of Tcl, Rivl programs have access to all the primitives of the Tcl language. Rivl extends Tcl with two data types: *images*, which represent still images, and *sequences*, which represent video segments (a timestamped set of images).

RIVL: A Resolution Independent Video Language

Jonathan Swartz and Brian C. Smith
Cornell University