A Resolution Independent Video Language*

Jonathan Swartz
Brian C. Smith
Department of Computer Science
4130 Upson Hall
Cornell University
Ithaca, NY 14853-7501 USA
{swartz,bsmith}@cs.cornell.edu
http://www.cs.cornell.edu/Info/Projects/zeno/rivl/rivl.html

ABSTRACT

As common as video processing is, programmers still implement video programs as manipulations of arrays of pixels. This paper presents a language extension 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 a program's portability, simplifying code reuse, and reducing development time. This paper also describes a Rivl interpreter and the strategies the interpreter uses to optimize Rivl programs. These optimizations include classical programming language optimizations, such as common subexpression elimination and out of order execution, image and video specific optimizations, such as computing only those images that will affect the output, and an optimized memory manager.

1. INTRODUCTION

In order to support the development of multimedia applications, programming languages should include audio, video, and images as true data types just as characters, text, and numerical values are true data types in today's languages. The operators in a multimedia language should be format independent, so that video and images of different formats can be easily intermixed, like integers and floating point numbers in most modern programming languages. The operators in a multimedia language should be resolution independent, so that high or low resolution input data produces better or worse results, just like single and double precision floating point numbers produce more or less accurate results in numeric computations. Platform independence and code reuse will be a useful side effect of multimedia programming languages, and just as optimizing and parallel compilers can take existing programs and make them run faster, so optimizing multimedia compilers will speed up multimedia programs.

Rivl (pronounced "rival") is a language extension and optimizing system designed with these goals in mind. Rivl provides a video data type and video operators that are format and resolution independent. Format independence means that MPEG[5] video can be freely intermixed with JPEG[7], Postscript[9], and uncompressed images. Resolution independence means that operations in Rivl (e.g., "cut the first five seconds of the video clip") are well defined whether the film resolution is 16 or 30 frames per second or the image size is 100x75 or 4000x3000. These features relieve the programmer of tedious, low level details and allow a runtime system to execute a Rivl program quickly on low resolution data for crude output (e.g., for prototyping or debugging) and slowly on high resolution data for more refined output.

Rivl's approach to video manipulation has significant advantages over current approaches:

- Rivl programs are easier to read, write, and maintain than their low-level counterparts.
- Rivl code is platform independent.
- Rivl expresses video and image editing operations in a format and resolution independent way.
- Rivl separates the description of video operations from their implementation, allowing the Rivl runtime system to improve the execution efficiency of Rivl programs.

We have implemented an interpreter for Rivl as an extension to the Tcl language [11], an approach that allows us to easily embed Rivl in other applications. This paper describes the design and implementation of the Rivl language, the Rivl interpreter, and its optimizer. The rest of the paper is organized as follows. Section 2 illustrates the Rivl language through a series of examples. Section 3 discusses the Rivl interpreter and how it optimizes image and video operations. Section 4 reviews related work, and concludes with

* This work was supported by a generous donation from Xerox Corporation.

Type	Image operations	Sequence operations	Description
Input/Output	im_read	seq_read	Read image/sequence from disk
	im_write	seq_write	Write image/sequence to disk
Geometric	im_trans	seq_shift	Translate an image in space / shift a sequence in time
	im_scale[C]	seq_scale	Scale an image in space / scale a sequence in time
	im_rotate[C]		Rotate an image [C = around its center]
Assembly	im_crop	seq_crop	Crop the specified region to make a new image/sequence
	im_concat	seq_concat	Concatenate multiple images/sequences end to end
	im_overlay	seq_overlay	Overlay multiple images/sequences in place
Conversion	ims_to_seq	seq_to_ims	Convert between a list of images and a sequence
		seq_map	Apply a script to each image in the sequence
Transforms	im_fade		Fade the image by a specified percentage
	im_resample		Resample the image at a specified size
	im_blur		Apply a blur filter to the image
	im_mask		Make transparent all pixels below a certain intensity

Table 1: Image and sequence primitives

current status and future research directions.

2. THE RIVL LANGUAGE

This section illustrates Rivl programs through a series of examples. 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*[1], which represent still images, and *sequences*, which represent video segments (a timestamped set of images).

Table 1 lists some of the Rivl primitives for manipulating images and sequences. The table is divided into five classes:

- *Input/output*. Currently supported formats include pgm/ppm[8], JPEG[7], MPEG[5], and Postscript[9].
- *Geometric*. The image operations in this class move and resize the image; the sequence operations speed up, slow down, and temporally shift the sequences.
- *Assembly*. This class provides "cut and paste" type operations on images and sequences.
- *Conversion*. This class provides functions to convert between images and sequences, and to map image operations over the frames of a sequence.
- *Transforms*. Image transforms used in this paper.

1. The Rivl image type is unrelated to the Tcl/Tk[11] canvas image type.

Sections 2.1 and 2.2 contain examples that clarify the use of image primitives and sequence primitives, respectively.

2.1. Image Operations

Consider the following Rivl fragment:

```
   set  image2  [im_scaleC  $image1  [expr  1  -
$p]]
   set image3 [im_rotateC $image2 [expr 360
* $p]]
```

Program 1: "Whirlpool" effect

`image1` is a Rivl image and `p` is a floating point value between 0 and 1. The first line calls `im_scaleC` to shrink `image1` about its center by a factor of 1-`p` and assigns the result to `image1`. The second line calls `im_rotateC` to rotate `image1` about its center by 360*`p` and again stores

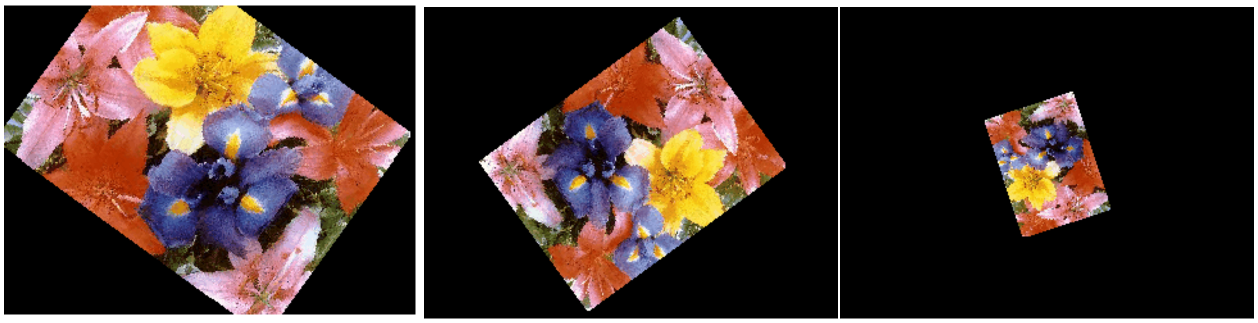


Figure 1: Output from program 1 for p = 0.1, 0.4, 0.7

the result in `image1`. Figure 1 shows the effect of this fragment with several values of `p`.

The repeated use of `set` in this fragment is cumbersome. To remedy this problem, we borrowed an idiom from Scheme: any operator with the character “!” appended destructively modifies its first argument. Taking advantage of this notation, we can rewrite program 1 as:

```
   im_scaleC! image1 [expr 1 - $p]]
   im_rotateC! image1 [expr 360 * $p]
```

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 Tcl implements pass-by-reference, a point we discuss in a related paper [14].

More complex effects can be constructed using Tcl constructs for looping, branching, procedure creation, and recursion. The Rivl program in figure 2 creates a fractal (Sierpinski’s Gasket) from an arbitrary image.

2.2. Sequence Operations

A *sequence*, the Rivl abstraction for video, can be thought of as a set of time-stamped images. Like image commands, sequence commands can be composed to express new operations. For instance, a common video editing operation is assembly, when two sequences are connected and written. The following Rivl fragment assembles the first 10 seconds of the sequence raiders.mpg and the sequence bobo.mpg, writing the result to out.mpg (all files are MPEG format):

```
   set raiders [seq_read raiders.mpg]
   set bobo [seq_read bobo.mpg]
   seq_crop! raiders 0.0 10.0
   seq_write [seq_concat $raiders $bobo]
out.mpg
```

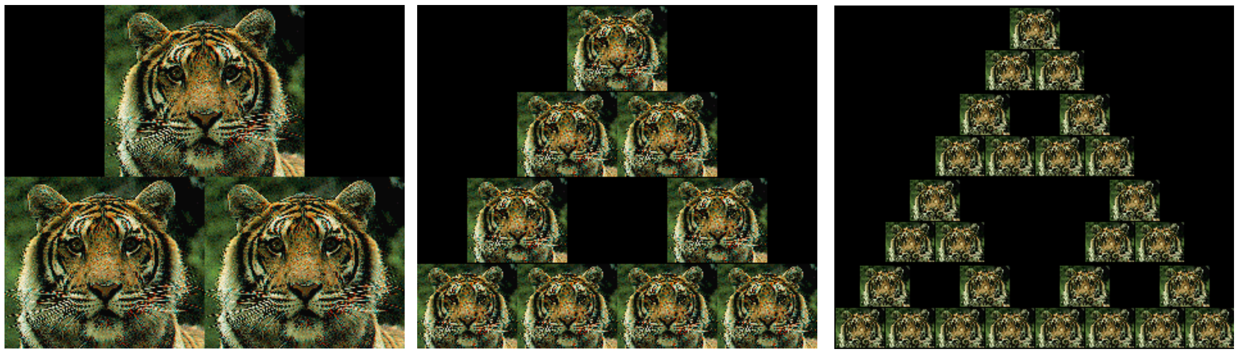
An important primitive in Rivl is `seq_map`. `seq_map` applies image effects to sequences, executing a given script for each image of a sequence and combining the resulting images into a new sequence. `Seq_map` is similar to `map` in Scheme. For example, consider the command

```
   seq_map $clip {im_resample %1 100 75}
```

`Seq_map` evaluates the template command (`im_resample %1 100 75`) on each image in `clip`, substituting the current image wherever `%1` appears in the template. The results are gathered and returned as a new sequence. Thus, this command returns a new sequence containing 100x75 (“thumbnail”) versions of the images in `clip`.

Sometimes, rather than applying the same operation on each image in a sequence, it is desirable to vary the operation over time. For example, consider the operation of fading a sequence to black. 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`



```
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

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

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

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

The Rivl language extension thus provides a powerful notation for programming with video. Rivl's high level semantic description of video operations also allows the interpreter to optimize the execution of Rivl programs. The next section describes these optimizations.

3. THE RIVL INTERPRETER

This section discusses the implementation of the Rivl interpreter. In the first two subsections we discuss the efficient implementation of image and sequence operations. In the third subsection we discuss memory allocation issues for video computing and describe Rivl's custom memory management system.

3.1 Implementation of Image Computing

There are two ways to optimize still 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 type of optimization is *lazy evaluation*, also known as *demand-*

```
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 $mid1 $transition] $mid2]

    seq_concat $begin $middle $end
}
```

Figure 3: Procedure to connect two sequences with an arbitrary transition

driven execution[10]. 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.

The Rivl interpreter alternates between two modes of operation: *graph-construction* mode and *graph-evaluation* mode. In *graph-construction* mode, the interpreter evaluates Rivl programs, recording and storing operations in a directed acyclic graph (DAG) whose edges correspond to images and whose nodes correspond to primitive operations (e.g. scale or overlay). This process is typically very fast, since image operations are recorded but not executed. In effect, the DAG represents a dynamic instruction trace of the Rivl program's execution.

Consider the following program, which overlays a scaled and rotated version of the image tiger.jpg onto the image flowers.jpg:

```
set tiger [im_read tiger.jpg]
im_scaleC! tiger 0.8
im_rotateC! tiger 288.0
set flowers [im_read flowers.jpg]
im_blur! flowers 5.0
im_overlay $tiger $flowers
im_write out.jpg
```

Figure 4 shows the graph created by this program. Both `im_scaleC` and `im_rotateC` are implemented with a pair of translations surrounding an `im_scale` or `im_rotate`; the translations move the origin of the scale and rotate operations to the center of the image.

A call to `im_write` triggers the *graph-evaluation* mode. In principle, the Rivl interpreter traverses the graph from inputs to output, computing intermediate images until the output image is computed. But before computing the images, the interpreter perform several optimizations. Two optimizations, *graph restructuring* and *result-region calculation*, are described below.

Graph restructuring. The first 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, figure 4 contains six adjacent affine transformations. Rivl collapses these nodes into a single affine transform (figure 5a). This optimization improves both the computation speed and the quality of the final image by reducing the number of times the image is resampled.

Result-region calculation. The second optimization, introduced by Shantzis[10], is to compute only those regions of each intermediate image that affect the final result. In our example program, only a small portion of `flowers` is visible in the final result (figure 5b, on right). It is sufficient to read in and blur only this portion.

Rivl calculates 3 regions for intermediate images (i.e. for every edge in the DAG): the *have* region, the *need* region, and the *result* region. The *have* region at an edge is the region of pixels provided by the edge's left node(s). The *need* region at an edge is the region of pixels needed by the edge's right node(s). Finally, the *result* region is the intersection of have and need. This intersection contains all the pixels that both have defined values and affect the final output image. Only the pixels inside the result region need to be

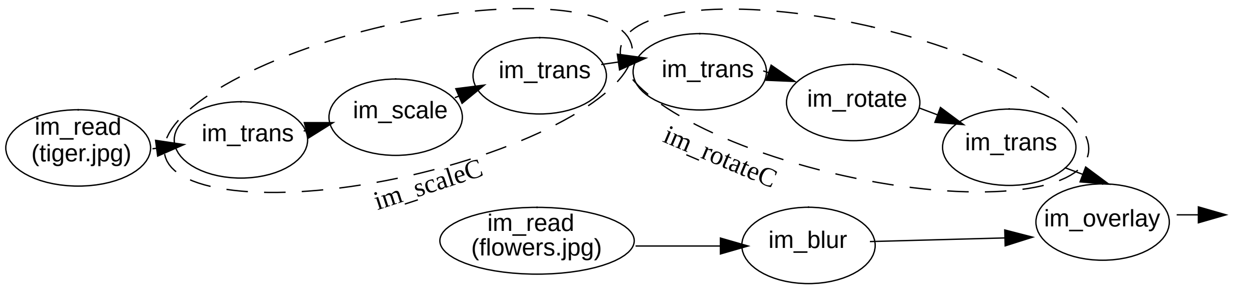


Figure 4: Sample image graph

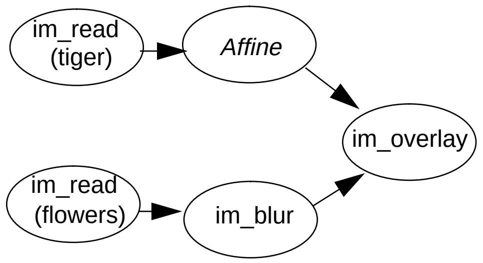


Figure 5a: Restructured image graph

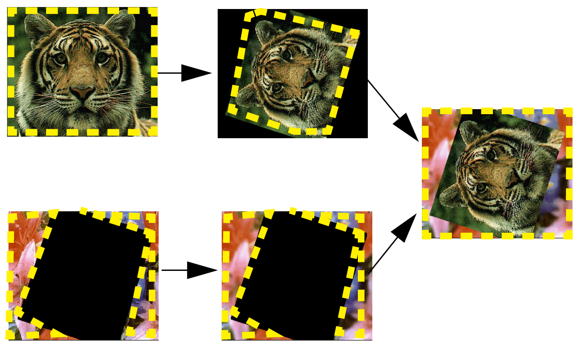


Figure 5b: Result regions

computed. Figure 5b shows the regions computed for each intermediate image in our example graph. In particular, only a small region is calculated for each of the two lower images.

Following these optimizations, the Rivl interpreter computes the graph's result image, writes the image to disk, and returns to processing commands in graph-construction mode.

3.2 Implementation of Sequences

Our implementation of sequences borrows many ideas from our implementation of images, since many still-image optimizations prove especially beneficial for sequence computing.

We will use scrolling titles as a sample task to motivate this section. The following program adds scrolling credits to the last 40 seconds of a 240 second (4 minute) movie:

```
    set credits [ims_to_seq [im_mask
[im_read credits.ps]]]
    seq_map! credits {im_trans %1 0.0 [expr -
%p*8000]}
    seq_scale! credits 40.0
    seq_shift! credits [expr 200.0]
    set raiders [seq_read raiders.mpg]
    set outseq [seq_map “$credits $raiders”
{im_overlay %1 %2}]
    seq_write $outseq out.mpg
```

Program 2: Adding scrolling credits to the end of a movie

The titles are stored as a Postscript program that generates a long image (640x8000). The `im_mask` function makes the titles' background transparent. The program converts the image to a one-second sequence (`credits`), scrolls `credits` upwards over time using `seq_map`, and scales and shifts `credits` to the desired time range (200-240 sec). The final `seq_map` overlays the titles onto the raiders movie, and the result is written to the file out.mpg.

Like image commands, sequence commands are stored in a graph (called the *sequence graph*) until the sequence is computed (e.g. in response to a `seq_write` command).

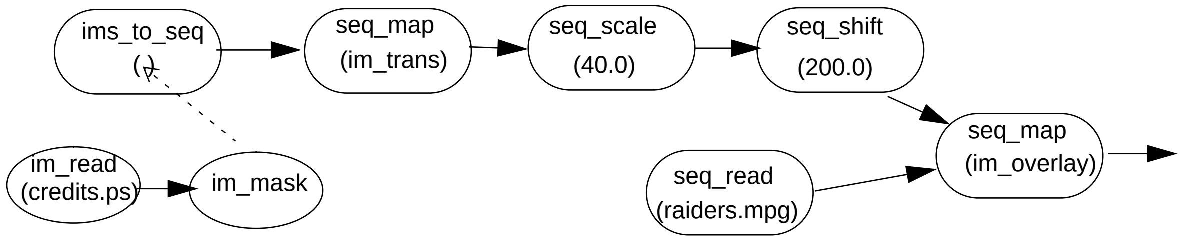


Figure 6: Sequence graph for scrolling titles

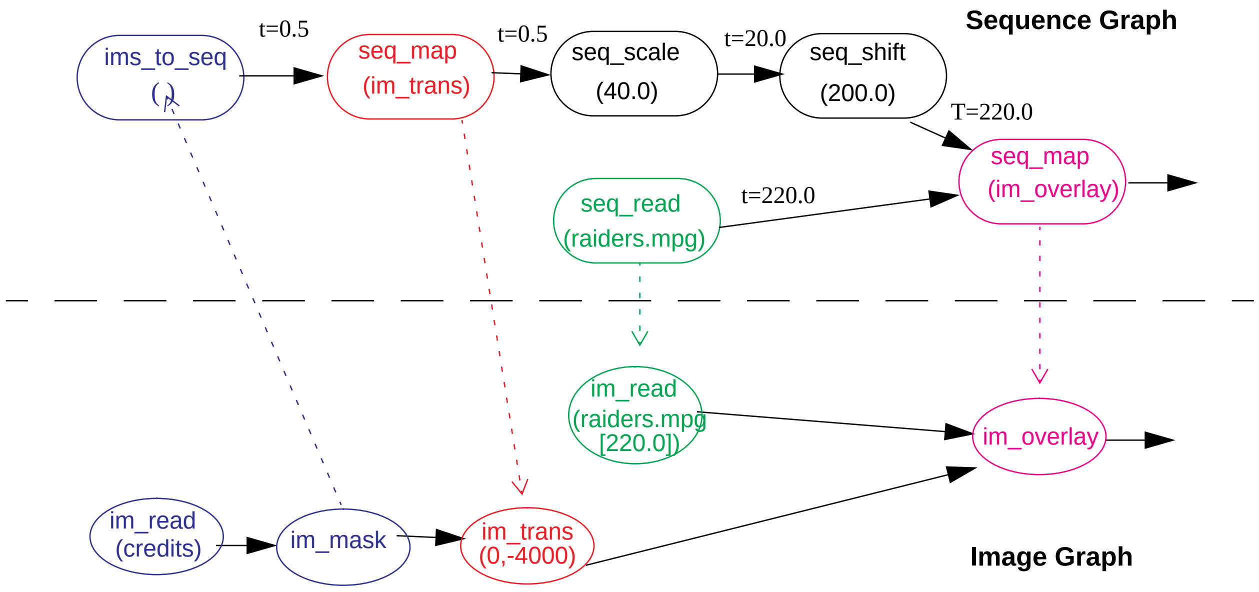


Figure 7: Generating the image graph for t = 220.0

Figure 6 shows the sequence graph for program 2, in which each node corresponds with one line of the program. The sequence graph is used to generate a set of image graphs that correspond to the sequence's individual frames.

Suppose we want to compute the frame in the output sequence at time `T`. We perform two passes over the sequence graph, a *backward pass* and a *forward pass*. In the backward pass, we compute a *timestamp* for each edge. An edge's timestamp indicates the time value at that edge that influences output frame `T`. As we traverse the graph, each `seq_scale` and `seq_shift` node we encounter potentially alters the timestamp. The top of figure 7 shows the timestamps computed for `T`=220.0.

In the forward pass, we build an image graph corresponding to output frame T. As we traverse the sequence graph, each `seq_read` and `seq_map` node we encounter adds node(s) to the image graph. `seq_read` uses the timestamp to determine which frame to read; `seq_map` uses the timestamp when substituting values for `%p` and `%t`. The bottom of figure 7 shows the image graph computed for `T` = 220.0.

To compute the whole sequence, we repeat the image graph generation algorithm for all relevant output times `T`, at increment of 1/`fps`, where `fps` is the desired frame rate (in frames/sec) of the output sequence. For program 2, T ranges from 0.0 to 240.0. The resulting graphs are merged into a single compound image graph, as shown in figure 8.

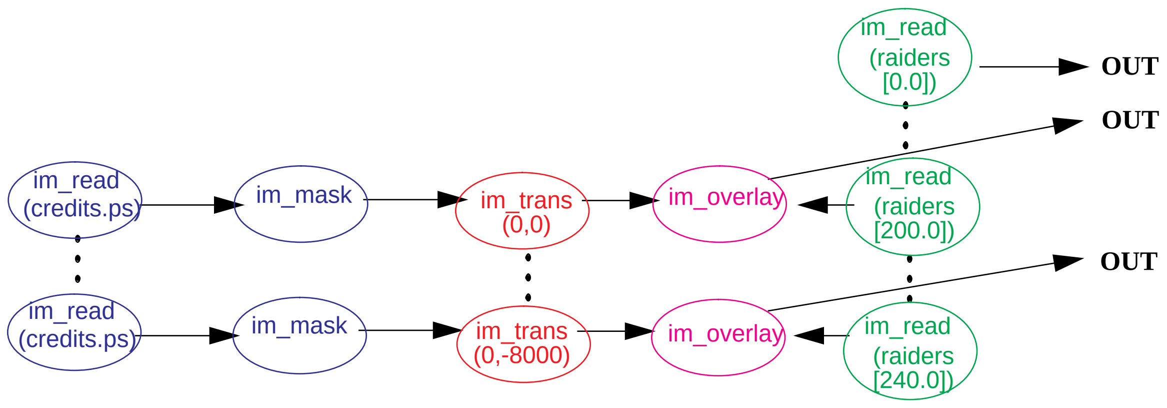


Figure 8: Image graph for entire sequence

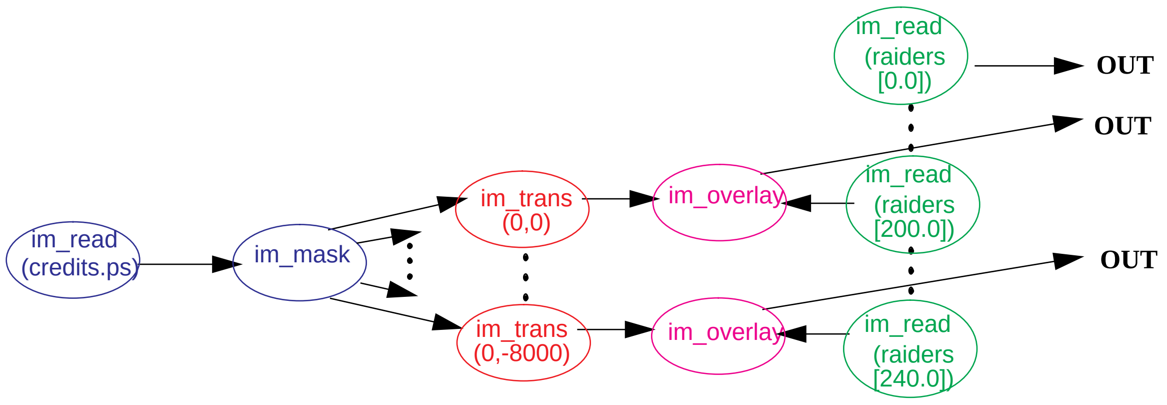


Figure 9: Image graph for entire sequence with shared subgraph

The optimizations of section 3.1 are used to process the compound image graph to produce the output images, along with two additional optimizations: *image subgraph reuse* and *direct-transfer detection*.

Image subgraph reuse. In figure 8 the subgraph containing `im_read(credits.ps)` and `im_mask` is replicated many times. It is more efficient to use a single subgraph with multiple output edges, as shown in figure 9. In this way the pixels of credit.ps are read and masked only once, and the various `im_trans` nodes share a common input. In general, Rivl detects and merges redundant image subgraphs whenever possible, a form of common subexpression elimination[1].

Direct-transfer detection. In our example, the first 200 seconds of `raiders.mpg` appear unchanged in the output. An obvious optimization is to avoid unnecessary decompression and compression by copying the compressed data directly to the output.

In formats such as MPEG, direct copying is not always possible on every frame, since MPEG sequences contain frames that are encoded as differences from other frames and cannot be decoded in isolation. However, MPEG streams are often divided into *groups of pictures* (*GOPs*), usually 15 to 30 frames long, that are independent from other GOPs. When reading and writing MPEG, Rivl transfers groups of pictures directly whenever possible.

3.3 Memory Management

In addition to optimizing image and sequence calculation, the Rivl interpreter contains a custom memory management module to cache previously computed images and cope with very large images.

To understand the utility of caching images, consider the

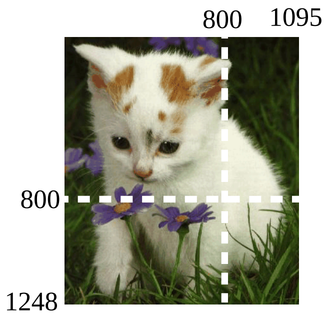


Figure 10a: Dividing large image into pages

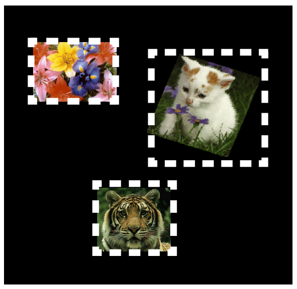


Figure 10b: Representing sparse image with 3 pages

evaluation of the graph in figure 9. The output of the `im_mask` node is used many times; it is advantageous to cache this image. The Rivl memory manager detects this case, freeing each image only when it is no longer needed in the current graph evaluation.

Another issue is whether to store images after a graph evaluation ends. Interactive applications of Rivl often require repeated evaluations of a slightly changing graph. Because the language restricts the way image graphs can be modified, the image associated with an edge remains accurate for the lifetime of the graph and can be cached. Unfortunately, we have no special knowledge about which images to cache for future graph evaluations. In principle, the user can access any edge that was ever created. Mistakenly discarding data is nonfatal, since we can always recompute the data, but such mistakes hurt performance.

To address this issue, Rivl provides an `im_priority` command to allow applications to set the *priority* of an image. The memory manager discards low priority images and keeps high priority images in memory. For instance, a video editor built using Rivl calls `im_priority` to raise the priority of displayed images, so that the results of special effects can be quickly viewed. We are looking into algorithms and heuristics for automatic priority adjustment. For example, images generated by expensive operations, and images that have been referenced repeatedly in the past, are candidates for high priority.

The initial implementation of Rivl treated images as indivisible memory buffers. Unfortunately, this representation performed poorly for large images. The Rivl memory manager divides large images into non-overlapping *pages* of manageable size (figure 10a). Pages are handled as independent entities by the memory manager, allowing an image to be cached in parts. In addition, large images with considerable blank space are efficiently represented by a set of non-contiguous pages (figure 10b).

To illustrate the utility of Rivl's memory management policy, we consider the execution of the scrolling titles program (program 2) under a standard memory model in which the entire image is read into a virtual memory buffer for the duration of the program. Assuming a 256 color image, this requires 5 MB of storage. In contrast, Rivl accomplishes the task as follows:

1. The 640x8000 title region is divided into ten equally sized pages (given Rivl's current maximum page size.)
2. Rivl allocates, loads, and masks each page of data only when necessary. The results of each call to `im_mask` are cached for future requests.
3. Rivl discards pages as soon as they are no longer needed.

The memory footprint is 1 MB, enough for Rivl to hold two pages at once.

In summary, the Rivl interpreter uses a variety of strategies to optimize execution of Rivl programs. They are:

- *Graph restructuring*: Combining or reordering nodes in the graph for speed
- *Result-region calculation*: Computing only the parts of an image that affect the output
- *Direct transfer detection*: Copying compressed data directly to the output when possible
- *Image subgraph reuse*: Sharing common subexpressions in the image graph
- *Image caching*: Caching images if they are needed later in the graph evaluation
- *Image subdivision*: Dividing large images into manageable pieces

4. 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 3. These include the Pixar system described by Shantzis[10], and Silicon Graphics' ImageVisionLibrary[13]. 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 primitives. We are adapting this idea to Rivl. None of the above mentioned systems provide language support for motion video.

Some systems (e.g., Data Explorer[2] or Khoros[12]) provide a graphical programming environment where image "programs" are expressed as flowcharts. Although this way of expressing image operations is an improvement over pixel manipulation, the limitations of flowcharts for expressing complex programs are well-known. Furthermore, the support for motion video operations in these systems is limited or non-existent.

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. Rivl has been used to build a simple video editor. Rivl and its editor can be found at http://www.cs.cornell.edu/Info/Projects/zeno/rivl/rivl.html.

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, computer vision, and morphing. With the right primitives, we hope to build a rapid prototyping environment for exploring video content processing.

We are also building a parallel implementation of the Rivl interpreter using workstation clusters. In this implementation, a Rivl program will run quickly using low resolution images on a small cluster, slowly using high resolution images on a small cluster, and quickly using high resolution images on a large cluster. The interpreter will automatically parallelize the Rivl program, using both coarse grained parallelism (one image / one process) and fine grained parallelism (one image / multiple processes).

5. REFERENCES

[1] Aho, Sethi and Ullman, Compilers: Principles, Techniques, and Tools, Reading, Mass. Addison-Wesley, 1988, pp. 592-595

[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] Poskanzer, Jef. *The Extended Portable Bitmap Toolkit (PBMPLUS)*.

[9] *Postscript*. Adobe Systems Incorporated, Mountain View, CA.

[10] Shantzis, Michael. "A Model for Efficient and Flexible Image Computing." SIGGRAPH '94 Proceedings, pp. 147-154.

[11] Ousterhout, John K. *Tcl and the Tk Toolkit*. Addison-Wesley, Massachusetts, 1994.

[12] 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.

[13] Silicon Graphics Inc. *ImageVision Library*. Silicon Graphics Inc., Mountain View, CA.

[14] Swartz, Jonathan and Smith, Brian C. *RIVL: A Resolution Independent Video Language*. Submitted to the 1995 Tcl/Tk Workshop, July 1995, Toronto, CA. http://www.cs.cornell.edu/Info/Projects/multimedia/rivl/tcl-tk-95.ps