The Utah Raster Toolkit

John W. Peterson
Rod G. Bogart
and
Spencer W. Thomas

University of Utah, Department of Computer Science[1]
Salt Lake City, Utah

Abstract

The Utah Raster Toolkit is a set of programs for manipulating and composing raster images. These tools are based on the Unix concepts of pipes and filters, and operate on images in much the same way as the standard Unix tools operate on textual data. The Toolkit uses a special run length encoding (RLE) format for storing images and interfacing between the various programs. This reduces the disk space requirements for picture storage and provides a standard header containing descriptive information about an image. Some of the tools are able to work directly with the compressed picture data, increasing their efficiency. A library of C routines is provided for reading and writing the RLE image format, making the toolkit easy to extend.

This paper describes the individual tools, and gives several examples of their use and how they work together. Additional topics that arise in combining images, such as how to combine color table information from multiple sources, are also discussed.

1. Introduction

Over the past several years, the University of Utah Computer Graphics Lab has developed several tools and techniques for generating, manipulating and storing images. At first this was done in a somewhat haphazard manner - programs would generate and store images in various formats (often dependent on particular hardware) and data was often not easily interchanged between them. More recently, some effort has been made to standardize the image format, develop an organized set of tools for operating on the images, and develop a subroutine library to make extending this set of tools easier. This paper is about the result of this effort, which we call the *Utah Raster Toolkit.* Using a single efficient image format, the toolkit provides a number of programs for performing common operations on images. These are in turn based on a subroutine library for reading and writing the images.

2. Origins

The idea for the toolkit arose while we were developing an image compositor. The compositor (described in detail in section 4.1) allows images to be combined in various ways. We found that a number of simple and independent operations were frequently needed before images could be composited together.

With the common image format, the subroutine library for manipulating it, and the need for a number of independent tools, the idea of a "toolkit" of image manipulating programs arose. These tools are combined using the

---

[1]Originally presented at the third Usenix Workshop on Graphics, Monterey California, November 1986

Unix shell, operating on images much like the standard Unix programs operate on textual data. For example, a Unix user would probably use the sequence of commands:

```
cat /etc/passwd | grep Smith | sort -t: +4.0 | lpr
```

to print a sorted list of all users named "Smith" in the system password file. Similarly, the Raster Toolkit user might do something like:

```
cat image.rle | avg4 | repos -p 0 200 | getfb
```

to downfilter an image and place it on top of the frame buffer screen. The idea is similar to a method developed by Duff [3] for three dimensional rendering.

3. The RLE format

The basis of all of these tools is a Run Length Encoded image format [8]. This format is designed to provide an efficient, device independent means of storing multi-level raster images. It is not designed for binary (bitmap) images. It is built on several basic concepts. The central concept is the *channel*. A channel corresponds to a single color, thus there are normally separate red, green and blue channels. Up to 255 color channels are available for use; one channel is reserved for coverage ("alpha") data. Although the format supports arbitrarily deep channels, the current implementation is restricted to 8 bits per channel. An RLE file is treated as a byte stream, making it independent of host byte ordering.

Image data is stored in an RLE file in a scanline form, with the data for each channel of the scanline grouped together. Runs of identical pixel values are compressed into a count and a value. However, sequences of differing pixels are also stored efficiently (i.e, not as a sequence of single pixel runs).

3.1. The RLE header

The file header contains a large amount of information about the image. This includes:

- The size and position on the screen,
- The number of channels saved and the number of bits per channel (currently, only eight bits per channel is supported),
- Several flags, indicating: how the background should be handled, whether or not an alpha channel was saved, if picture comments were saved,
- The size and number of channels in the color map, (if the color map is supplied),
- An optional background color,
- An optional color map,
- An optional set of comments. The comment block contains any number of null-terminated text strings. These strings are conventionally of the form "name=value", allowing for easy retrieval of specific information.

3.2. The scanline data

The scanline is the basic unit that programs read and write. It consists of a sequence of operations, such as *Run*, *SetChannel*, and *Pixels*, describing the actual image. An image is stored starting at the lower left corner and proceeding upwards in order of increasing scanline number. Each operation and its associated data takes up an even number of bytes, so that all operations begin on a 16 bit boundary. This makes the implementation more efficient on many architectures.

Each operation is identified by an 8 bit opcode, and may have one or more operands. Single operand operations fit into a single 16 bit word if the operand value is less than 256. So that operand values are not limited to the range 0..255, each operation has a *long* variant, in which the byte following the opcode is ignored and the following word is taken as a 16 bit quantity. The long variant of an opcode is indicated by setting the bit 0x40 in the opcode (this allows for 64 opcodes, of which 6 have been used so far.)

The current set of opcodes include:

SkipLines	Increment the *scanline number* by the operand value, terminating the current scanline.
SetColor	Set the *current channel* to the operand value.
SkipPixels	Skip over pixels in the current scanline. Pixels skipped will be left in the background color.
PixelData	Following this opcode is a sequence of pixel values. The length of the sequence is given by the operand value.
Run	This is the only two operand opcode. The first operand is the length (*N*) of the run. The second operand is the pixel value, followed by a filler byte if necessary[2]. The next *N* pixels in the scanline are set to the given pixel value.
EOF	This opcode has no operand, and indicates the end of the RLE file. It is provided so RLE files may be concatenated together and still be correctly interpreted. It is not required, a physical end of file also indicates the end of the RLE data.

3.3. Subroutine Interface

Two similar subroutine interfaces are provided for reading and writing files in the RLE format. Both read or write a scanline worth of data at a time. A simple "row" interface communicates in terms of arrays of pixel values. It is simple to use, but slower than the "raw" interface, which uses arrays of "opcode" values as its communication medium.

In both cases, the interface must be initialized by calling a setup function. The two types of calls may be interleaved; for example, in a rendering program, the background could be written using the "raw" interface, while scanlines containing image data could be converted with the "row" interface. The package allows multiple RLE streams to be open simultaneously, as is necessary for use in a compositing tool, for example. All data relevant to a particular RLE stream is contained in a "globals" structure. This structure essentially echoes the information in the RLE header, along with current state information about the RLE stream.

4. The tools

4.1. The image compositor - Comp

`Comp` implements an image compositor based on the compositing algorithms presented in [6]. The compositing operations are based on the presence of an alpha channel in the image. This extra channel usually defines a mask which represents a sort of a cookie-cutter for the image. This is the case when alpha is 255 (full coverage) for pixels inside the shape, zero outside, and between zero and 255 on the boundary. When the compositor operates on images of the cookie-cutter style, the operations behave as follows:

A **over** B (the default)	The result will be the union of the two image shapes, with A obscuring B in the region of overlap.
A **atop** B	The result shape is the same as image B, with A obscuring B where the image shapes overlap. (Note that this differs from "over" because the portion of A outside the shape of B will not be in the result image.)
A **in** B	The result is simply the image A cut by the shape of B. None of the image data of B will be in the result.
A **out** B	The result image is image A with the shape of B cut out.
A **xor** B	The result is the image data from both images that is outside the overlap region. The overlap region will be blank.
A **plus** B	The result is just the sum of the image data. This operation is actually independent of the

[2]E.g., a 16 bit pixel value would not need a filler byte.

alpha channels.

The alpha channel can also represent a semi-transparent mask for the image. It would be similar to the cookie-cutter mask, except the interior of the shape would have alpha values that represent partial coverage; e.g. 128 is half coverage. When one of the images to be composited is a semi-transparent mask, the following operations have useful results:

Semi-transparent A **over** B
: The image data of B will be blended with that of A in the semi-transparent overlap region. The resulting alpha channel is as transparent as that of image B.

A **in** Semi-transparent B
: The image data of A is scaled by the mask of B in the overlap region. The alpha channel is the same as the semi-transparent mask of B.

If the picture to be composited doesn't have an alpha channel present, comp assumes an alpha of 255 (i.e., full coverage) for non-background pixels and an alpha of zero for background pixels.

Comp is able to take advantage of the size information for the two images being composited. For example, if a small picture is being composited over a large backdrop, the actual compositing arithmetic is only performed on a small portion of the image. By looking at the image size in the RLE header, comp performs the compositing operation only where the images overlap. For the rest of the image the backdrop is copied (or not copied, depending on the compositing operation). The "raw" RLE read and write routines are used to perform this copy operation, thus avoiding the cost of compressing and expanding the backdrop as well.

4.2. Basic image composition - Repos and Crop

`Repos` and **`crop`** are the basic tools for positioning and arranging images to be fed to the compositor. Crop simply throws away all parts of the image falling outside the rectangle specified. Repos positions an image to a specific location on the screen, or moves it by an incremental amount. Repos does not have to modify any of the image data, it simply changes the position specification in the RLE header. In order to simplify the code, the Raster Toolkit does not allow negative pixel coordinates in images.

4.3. Changing image orientation and size - Flip, Fant and Avg4

Two tools exist for changing the orientation of an image on the screen. **`Flip`** rotates an image by 90 degrees right or left, turns an image upside down, or reverse it from right to left. **`Fant`** rotates an image an arbitrary number of degrees from -45 to 45. In order to get rotation beyond -45 to 45, flip and fant can be combined, for example:

```
flip -r < upright.rle | fant -a 10 > rotated.rle
```

rotates an image 100 degrees. Fant is also able to scale images by an arbitrary amount in X and Y. A common use for this is to stretch or shrink an image to correct for the aspect ratio of a particular frame buffer. Many frame buffers designed for use with standard video hardware display a picture of 512x480 pixels on a screen with the proportions 4:3, resulting in an overall pixel aspect ratio of 6:5. If the picture **`kloo.rle`** is digitized or computed assuming a 1:1 aspect ratio (square pixels), the command:

```
cat kloo.rle | fant -s 1.0 1.2 | getfb
```

correctly displays the image on a frame buffer with a 6:5 aspect ratio. Fant is implemented using a two-pass subpixel sampling algorithm [4]. This algorithm performs the spatial transform (rotate and/or scale) first on row by row basis, then on a column by column basis. Because the transformation is done on a subpixel level, fant does not introduce aliasing artifacts into the image.

`Avg4` downfilters an RLE image into a resulting image of 1/4th the size, by simply averaging four pixel values in the input image to produce a single pixel in the output. If the original image does not contain an alpha channel, avg4 creates one by counting the number of non-zero pixels in each group of four input pixels and using the count to produce a coverage value. While the alpha channel produced this way is crude (only four levels of coverage) it is enough to make a noticeable improvement in the edges of composited images. One use for avg4 is to provide anti-aliasing for rendering programs that perform no anti-aliasing of their own. For example, suppose **`huge.rle`** is a 4k x 4k pixel image rendered without anti aliasing and without an alpha channel. Executing the commands:

```
cat huge.rle | avg4 | avg4 | avg4 > small.rle
```

produces an image **small.rle** with 64 (8x8) samples per pixel and an alpha channel with smooth edges.

Images generated from this approach are as good as those produced by direct anti-aliasing algorithms such as the A-buffer [2]. However, a properly implemented A-buffer renderer produces images nearly an order of magnitude faster.

4.4. Color map manipulation - Ldmap and Applymap

As mentioned previously, RLE files may optionally contain a color map for the image. **Ldmap** is used to create or modify a color map within an RLE file. Ldmap is able to create some standard color maps, such as linear maps with various ramps, or maps with various gamma corrections. Color maps may also be read from a simple text file format, or taken from other RLE files. Ldmap also performs *map composition*, where one color map is used as in index into the other. An example use for this is to apply a gamma correction to an image already having a non-linear map.

Applymap applies the color map in an rle file to the pixel values in the file. For example, if the color map in **kloo.rle** contained a color map with the entries (for the red channel):

```
Index   Red color map
    0:      5
    1:      7
    2:      9
```

Then a (red channel) pixel value of zero would be displayed with an intensity of five (assuming the display program used the color map in **kloo.rle**). When kloo.rle is passed through applymap,

```
cat kloo.rle | applymap > kloo2.rle
```

pixels that had a value of zero in **kloo.rle** now have a value of five in **kloo2.rle**, pixel values of one would now be seven, etc. When displaying the images on a frame buffer, **kloo2.rle** appears the same with a linear map loaded as **kloo.rle** does with its special color map loaded.

One use for these tools is merging images with different compensation tables. For example, suppose image **gam.rle** was computed so that it requires a gamma corrected color table (stored in the RLE file) to be loaded to look correct on a particular monitor, and image **lin.rle** was computed with the gamma correction already taken into consideration (so it looks correct with a linear color table loaded). If these two images are composited together without taking into consideration these differences, the results aren't correct (part of the resulting image is either too dim or washed out). However, we can use applymap to "normalize" the two images to using a linear map with:

```
cat gam.rle | applymap | comp - lin.rle | ldmap -l > result.rle
```

4.5. Generating backgrounds

Unlike most of the toolkit programs which act as filters, **background** just produces output. Background either produces a simple flat field, or it produces a field with pixel intensities ramped in the vertical direction. Most often background is used simply to provide a colored backdrop for an image. Background is another example of a program that takes advantage of the "raw" RLE facilities. Rather than generating the scanlines and compressing them, background simply generates the opcodes required to produce each scanline.

4.6. Converting full RGB images to eight bits - to8 and tobw

Although most of the time we prefer to work with full 24 bit per pixel images, (32 bits with alpha) we often need the images represented with eight bits per pixel. Many inexpensive frame buffers (such as the AED 512) and many personal color workstations (VaxStation GPX, Color Apollos and Suns) are equipped with eight bit displays.

To8 converts a 24 bit image to an eight bit image by applying a dither matrix to the pixels. This basically trades spatial resolution for color resolution. The resulting image has eight bits of data per pixel, and also contains a special color map for displaying the dithered image (since most eight bit frame buffers still use 24 bit wide color table entries).

Tobw converts a picture from 24 bits of red, green, and blue to an eight bit gray level image. It uses the standard television YIQ transformation of

$$graylevel = 0.35 \times red + 0.55 \times green + 0.10 \times blue.$$

These images are often preferred when displaying the image on an eight bit frame buffer and shading information is more important than color.

5. Interfacing to the RLE toolkit

In order to display RLE images, a number of programs are provided for displaying the pictures on various devices. These display programs all read from standard input, so they are conveniently used as the end of a raster toolkit pipeline. The original display program, **getfb**, displays images on our ancient Grinnell GMR-27 frame buffer. Many display programs have since been developed:

getcx	displays images on a Chromatics CX1500
getiris	displays an image on an Iris workstation (via ethernet)
getX	for the X window system
getap	for the Apollo display manager
gethp	for the Hewlett-Packard Series 300 workstation color display
rletops	converts an RLE file to gray-level PostScript output[3]

The **getX**, **getap** and **gethp** programs automatically perform the dithering required to convert a 24 bit RLE image to eight bits (for color nodes) or one bit (for bitmapped workstations). Since all of these workstations have high resolution (1Kx1K) displays, the trade of spatial resolution for color resolution produces very acceptable results. Even on bitmapped displays the image quality is good enough to get a reasonable idea of an image's appearance.

Two programs, **painttorle** and **rletopaint** are supplied to convert MacPaint images to RLE files. This offers a simple way to add text or graphic annotation to an RLE image (although the resolution is somewhat low).

6. Examples

A typical use for the toolkit is to take an image generated with a rendering program, add a background to it, and display the result on a frame buffer:

```
rlebg 250 250 250 -v | comp image.rle - | getfb
```

What follows are some more elaborate applications:

6.1. Making fake shadows

In this exercise we take the dart (Figure 6-1a) and stick it into the infamous mandrill, making a nice (but completely fake) shadow along the way.

First the image of the dart is rotated by 90 degrees (using **rleflip -l** and then stretched in the X direction (using **fant -s 1.3 1.0**) to another image we can use as a shadow template, figure 6-1b. Then we take this image, **dart_stretch_rot.rle**, and do:

```
rlebg 0 0 0 175 \
   | comp -o in - dart_stretch_rot.rle > dart_shadow.rle
```

This operation uses the stretched dart as a cookie-cutter, and the shape is cut out of a black image, with a coverage mask (alpha channel) of 175. Thus when we composite the shadow (figure 6-2a) over the mandrill image, it lets 32% of the mandrill through ((255 - 175) / 255 = 32%) with the rest being black.

---

[3] **Rletops** was used to produce the figures in this paper.

pics/dart_and_strtch.ps

Figure 6-1: **a:** Original dart image. **b:** Rotated and stretched dart.

pics/dart_shadow_and_monkey.ps

Figure 6-2: **a:** Dart shadow mask. **b:** Resulting skewered baboon.

Now all we have left to do is to composite the dart and the shadow over the mandrill image, and the result is figure 6-2b. Note that since the original dart image was properly anti-aliased, no aliasing artifacts were introduced in the final image.

6.2. Cutting holes

In this example we start out out with a single bullet hole, created with the same modelling package that produced the dart.[4] First the hole (which originally filled the screen) is downfiltered to a small size, with

[4]The bullet hole was modeled with cubic B-splines. Normal people probably would have painted something like this...

```
avg4 < big_hole.rle | avg4 | avg4 > smallhole.rle
```

pics/bullet_holes_and_shot_turb.ps

Figure 6-3: **a:** A set of bullet holes. **b:** A shot-up turbine blade.

Now by invoking **repos** and **comp** several times in a row, a set of shots is built up (Figure 6-3a)

To shoot up the turbine blade, the *atop* operator of comp is used. This works much like *over*, except the bullet holes outside of the turbine blade don't appear in the resulting image (Figure 6-3b) Note how the top left corner is just grazed.

But there's still one catch. Suppose we want to display the shot-up turbine blade over a background of some sort. We want to be see the background through the holes. To do this we come up with another mask to cut away the centers of the bullet holes already on the turbine blade so we can see through them (Figure 6-4a).

pics/center_masks_and_fin_turb.ps

Figure 6-4: **a:** Cut-away masks for the bullet hole centers.
b: Finished turbine blade

Now we can see through the blade (Figure 6-4b).

6.3. Integrating digitized data

In this example, we take a raw digitized negative and turn it into a useful frame buffer image. Figure 6-5a shows the original image from the scanner. First the image is cropped to remove the excess digitized portion, then rotated into place with **`flip -r`**, resulting in Figure 6-5b.

pics/scanned_and_cropped.ps

Figure 6-5: **a:** The raw digitized negative. **b:** Cropped and rotated image

To get the image to appear as a positive on the frame buffer, a negative linear color map (stored as text file) is loaded, then this is composed with a gamma correction map of 2.2, and finally applied to the pixels with the command:

```
ldmap -f neg.cmap < neg_pahriah.rle | ldmap -a -g 2.2 \
   | applymap > pahriah.rle
```

The result, Figure 6-6 now appears correct with a linear color map loaded.

7. Future Work

Needless to say, a project such as the Raster Toolkit is open ended in nature, and new tools are easily added. For example, most of the tools we have developed to date deal with image synthesis and composition, primarily because that is our research orientation. Additional tools could be added to assist work in areas such as vision research or image processing (for examples, see [1, 7]).

Another interesting application would be a visual "shell" for invoking the tools. Currently, arguments to programs like **`crop`** and **`repos`** are specified by tediously finding the numbers with the frame buffer cursor and then typing them into a shell. The visual shell would allow the various toolkit operations to be interactively selected from a menu. Such a shell could probably work directly with an existing workstation window system such as X [5] to provide a friendly environment for using the toolkit.

8. Conclusions

The Raster Toolkit provides a flexible, simple and easily extended set of tools for anybody working with images in a Unix-based environment. We have ported the toolkit to a wide variety of Unix systems, including Gould UTX, HP-UX, Sun Unix, Apollo Domain/IX, 4.2 and 4.3 BSD, etc. The common interfaces of the RLE format and the subroutine library make it easy to interface the toolkit to a wide variety of image sources and displays.

pics/pahriah_final.ps

Figure 6-6: Final image of the old Pahriah ghost town

9. Acknowledgments

This work was supported in part by the National Science Foundation (DCR-8203692 and DCR-8121750), the Defense Advanced Research Projects Agency (DAAK11-84-K-0017), the Army Research Office (DAAG29-81-K-0111), and the Office of Naval Research (N00014-82-K-0351). All opinions, findings, conclusions or recommendations expressed in this document are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

[1] Burt, Peter J, and Adelson, Edward H.
A Multiresolution Spline With Application to Image Mosaics.
ACM Transactions on Graphics 2(4):217-236, October, 1983.

[2] Carpenter, Loren.
The A-Buffer, An Antialiased Hidden Surface Method.
Computer Graphics 18(3):103, July, 1984.
Proceedings of SIGGRAPH 84.

[3] Duff, Tom.
Compositing 3-D Rendered Images.
Computer Graphics 19(3):41, July, 1985.
Proceedings of SIGGRAPH 85.

[4] Fant, Karl M.
A Nonaliasing, Real-Time, Spatial Transform.
IEEE Computer Graphics and Applications 6(1):71, January, 1986.

[5] Gettys, Jim, Newman, Ron, and Fera, Tony D.
Xlib - C Language X Interface, Protocol Version 10.
Technical Report, MIT Project Athena, January, 1986.

[6] Porter, Thomas and Duff, Tom.
Compositing Digital Images.
Computer Graphics 18(3):253, July, 1984.
Proceedings of SIGGRAPH 84.

[7] Stockham, Thomas G.
Image Processing in the Context of a Visual Model.
Proceedings of the IEEE 60(7):828-842, July, 1972.

[8] Thomas, Spencer W.
Design of the Utah RLE Format.
Technical Report 86-15, Alpha_1 Project, CS Department, University of Utah, November, 1986.

Table of Contents