Design of the Utah RLE Format

Spencer W. Thomas

University of Utah, Department of Computer Science

Abstract

The Utah RLE (Run Length Encoded) format is designed to provide an efficient, device independent means of storing multi-level raster images. Images of arbitrary size and depth can be saved. The design of the format is presented, followed by descriptions of the library routines used to create and read RLE format files.

1. Introduction

The Utah RLE (Run Length Encoded) 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 that of a *channel*. A channel corresponds to a single color, thus there are normally a red channel, a green channel, and a blue channel. Up to 255 color channels are available for use; one channel is reserved for "alpha" data. Although the format supports arbitrarily deep channels, the current implementation is restricted to 8 bits per channel.

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 (not as a sequence of single pixel runs).

The file header contains a large amount of information about the image, including its size, the number of channels saved, whether it has an alpha channel, an optional color map, and comments. The comments may be used to add arbitrary extra information to the saved image.

A subroutine interface has been written to allow programs to read and write files in the RLE format. Two interfaces are available, one that completely interprets the RLE file and returns scanline pixel data, and one that returns a list of "raw" run and pixel data. The second is more efficient, but more difficult to use, the first is easy to use, but slower.

The Utah RLE format has been used to save images from many sources, and to display saved images on many different displays and from many different computers.

2. Description of RLE Format

All data in the RLE file is treated as a byte stream. Where quantities larger than 8 bits occur, they are written in PDP-11 byte order (low order byte first).

The RLE file consists of two parts, a header followed by scanline data. The header contains general information about the image, while the scanline data is a stream of operations describing the image itself.

2.1. The Header

<table>
<tr><td colspan="2">Magic number</td></tr>
<tr><td colspan="2">xpos</td></tr>
<tr><td colspan="2">ypos</td></tr>
<tr><td colspan="2">xsize</td></tr>
<tr><td colspan="2">ysize</td></tr>
<tr><td>flags</td><td>ncolors</td></tr>
<tr><td>pixelbytes</td><td>ncmap</td></tr>
<tr><td>cmaplen</td><td>red bg</td></tr>
<tr><td>green bg</td><td>blue bg</td></tr>
<tr><td colspan="2">color map entry 0</td></tr>
<tr><td colspan="2">color map entry 1</td></tr>
<tr><td colspan="2"></td></tr>
</table>

Figure 2-1: RLE file header

The header has a fixed part and a variable part. A diagram of the header is shown in Figure 2-1. The magic number identifies the file as an RLE file. Following this are the coordinates of the lower left corner of the image and the size of the image in the X and Y directions. Images are defined in a first quadrant coordinate system (origin at the lower left, X increasing to the right, Y increasing up.) Thus, the image is enclosed in the rectangle

$$[xpos, xpos+xsize-1] X [ypos, ypos+ysize-1].$$

The position and size are 16 bit integer quantities; images up to 32K square may be saved (the sizes should not be negative).

A flags byte follows. There are currently four flags defined:

- ClearFirst — If this flag is set, the image rectangle should first be cleared to the background color (q.v.) before reading the scanline data.
- NoBackground — If this flag is set, no background color is supplied, and the ClearFirst flag should be ignored.
- Alpha — This flag indicates the presence of an "alpha" channel. The alpha channel is used by image compositing software to correctly blend anti-aliased edges. It is stored as channel -1 (255).
- Comments — If this flag is set, comments are present in the variable part of the header, immediately following the color map.

The next byte is treated as an unsigned 8 bit value, and indicates the number of color channels that were saved. It may have any value from 0 to 254 (channel 255 is reserved for alpha values).

The *pixelbits* byte gives the number of bits in each pixel. The only value currently supported by the software is 8 (in fact, this byte is currently ignored when reading RLE files). Pixel sizes taking more than one byte will be packed low order byte first.

The next two bytes describe the size and shape of the color map. *Ncmap* is the number of color channels in the color map. It need not be identical to *ncolors*, but interpretation of values of *ncmap* different from *0*, *1*, or *ncolors* may be ambiguous, unless *ncolors* is *1*. If *ncmap* is zero, no color map is saved. *Cmaplen* is the log base 2 of the length of each channel of the color map. Thus, a value for *cmaplen* of 8 indicates a color map with 256 entries per channel.

Immediately following the fixed header is the variable part of the file header. It starts with the background color. The background color has *ncolors* entries; if necessary, it is filled out to an odd number of bytes with a filler byte on the end (since the fixed header is an odd number bytes long, this returns to a 16 bit boundary).

Following the background color is the color map, if present. Color map values are stored as 16 bit quantities, left justified in the word. Software interpreting the color map must apply a shift appropriate to the application or to the hardware being used. This convention permits use of the color map without knowing the original output precision. The channels of the map are stored in increasing numerical order (starting with channel 0), with the entries of each channel stored also in increasing order (starting with entry 0). The color map entries for each channel are stored contiguously.

Comments, if present, follow the color map. A 16 bit quantity giving the length of the comment block comes first. If the length is odd, a filler byte will be present at the end, restoring the 16 bit alignment (but this byte is not part of the comments). The comment block contains any number of null-terminated text strings. These strings will conventionally be of the form "name=value", allowing for easy retrieval of specific information. However, there is no restriction that a given name appear only once, and a comment may contain an arbitrary string. The intent of the comment block is to allow information to be attached to the file that is not specifically provided for in the RLE format.

2.2. The Scanline Data

The scanline data 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.

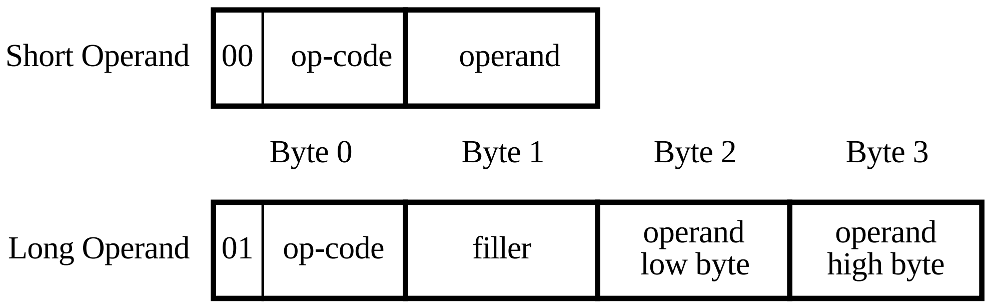

Figure 2-2: RLE file operand formats

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 two single operand formats are shown pictorially in Figure 2-2.

The individual operations will now be discussed in detail. The descriptions are phrased in terms of the actions necessary to interpret the file. Three indices are necessary: the *current channel*, the *scanline number*, and the *pixel index*. The *current channel* is the channel to which data operations apply. The *scanline number* is just the Y position of the scanline in the image. The *pixel index* is the X position of the pixel within the scanline. The operations are:

SkipLines	Increment the *scanline number* by the operand value. This operation terminates the current scanline. The *pixel index* should be reset to the xpos value from the header.
SetColor	Set the *current channel* to the operand value. This operation does not have a long variant. Note that an operand value of 255 will be interpreted as a −1, indicating the alpha channel. All other operand values are positive. The *pixel index* is reset to the xpos value.
SkipPixels	Skip over pixels in the current scanline. Increment *pixel index* by the operand value. 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. If the length of the sequence is odd, a filler byte is appended. Pixel values are inserted into the scanline in increasing X order. The *pixel index* is incremented by the sequence length.
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[1]. The next *N* pixels in the scanline are set to the given pixel value. The *pixel index* is incremented by *N*, to point to the pixel following the run.
EOF	This opcode has no operand, and indicates the end of the RLE file. It is provided so that RLE files may be concatenated together and still be correctly interpreted. It is not required, a physical end of file will also indicate the end of the RLE data.

2.3. 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 a list 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.

The globals structure echoes the format of the RLE header. The fields are described below:

dispatch	The RLE creation routines are capable of writing various types of output files in addition to RLE. This value is an index into a dispatch table. This value is initialized by *sv_setup*.
ncolors	The number of color channels in the output file. Up to this many color channels will be saved, depending on the values in the channel bitmap (see below).
bg_color	A pointer to an array of *ncolors* integers containing the background color.
alpha	If this is non-zero, an alpha channel will be saved. The presence or absence of an alpha channel has no effect on the value in *ncolors*.
background	Indicates how to treat background pixels. It has the following values: 0 — Save all pixels, the background color is ignored. 1 — Save only non-background pixels, but don’t set the "clear screen" bit. This indicates "overlay" mode, a cheap form of compositing (but see note below about this.) 2 — Save only non-background pixels, clear the screen to the background color before restoring the image.

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

xmin, xmax, ymin, ymax
: Inclusive bounds of the image region being saved.

ncmap
: Number of channels of color map to be saved. The color map will not be saved if this is 0.

cmaplen
: Log base 2 of the number of entries in each channel of the color map.

cmap
: Pointer to an array containing the color map. The map is saved in "channel major" order. Each entry in the map is a 16 bit value with the color value left justified in the word. If this pointer is NULL, no color map will be saved.

comments
: Pointer to an array of pointers to strings. The array is terminated by a NULL pointer (like *argv* or *envp*). If this pointer is NULL or if the first pointer it points to is NULL, comments will not be saved.

fd
: File (FILE *) pointer to be used for writing or reading the RLE file.

bits
: A bitmap containing 256 bits. A channel will be saved (or retrieved) only if the corresponding bit is set in the bitmap. The alpha channel corresponds to bit 255. The bitmap allows an application to easily ignore color channel data that is irrelevant to it.

The globals structure also contains private data for use by the RLE reading and writing routines; data that must be maintained between calls, but that applies to each stream separately.

2.4. Writing RLE files

To create a run-length encoded file, one first initializes a globals structure with the relevant information about the image, including the output file descriptor. The output file should be open and empty. Then one calls *sv_setup*:

```
sv_setup( RUN_DISPATCH, &globals );
```

This writes the file header and initializes the private portions of the global data structure for use by the RLE file writing routines.

The image data must be available or expressible in a scanline order (with the origin at the bottom of the screen). After each scanline is computed, it is written to the output file by calling one of *sv_putrow* or *sv_putraw*. If a vertical interval of the image has no data, it may be skipped by calling sv_skiprow:

```
/* Skip nrow scanlines */
sv_skiprow( &globals, nrow );
```

If the image data for a scanline is available as an array of pixel values, *sv_putrow* should be used to write the data to the output file. As an example, let us assume that we have a 512 pixel long scanline, with three color channels and no alpha data. We could call *sv_putrow* as follows:

```
rle_pixel scandata[3][512], *rows[3];
int i;

for ( i = 0; i < 3; i++ )
    rows[i] = scandata[i];
sv_putrow( rows, 512, &globals );
```

Note that *sv_putrow* is passed an array of pointers to vectors of pixels. This makes it easy to pass arbitrarily many, and to specify values of *rowlen* different from the size of (e.g.) the *scandata* array.

The first element of each row of pixels is the pixel at the *xmin* location in the scanline. Therefore, when saving only part of an image, one must be careful to set the *rows* pointers to point to the correct pixel in the scanline.

If an alpha channel is specified to be saved, things get a little more complex. Here is the same example, but now with an alpha channel being saved.

```
rle_pixel scandata[3][512],
          alpha[512], *rows[4];
int i;

rows[0] = alpha;
for ( i = 0; i < 3; i++ )
    rows[i+1] = scandata[i];
sv_putrow( rows+1, 512, &globals );
```

The *sv_putrow* routine expects to find the pointer to the alpha channel at the −1 position in the rows array. Thus, we pass a pointer to *rows[1]* and put the pointer to the alpha channel in *rows[0]*.

Finally, after all scanlines have been written, we call *sv_puteof* to write an *EOF* opcode into the file. This is not strictly necessary, since a physical end of file also indicates the end of the RLE data, but it is a good idea.

Here is a skeleton of an application that uses *sv_putrow* to save an image is shown in Figure 2-3. This example uses the default values supplied in the globals variable *sv_globals*, modifying it to indicate the presence of an alpha channel.

Using *sv_putraw* is more complicated, as it takes arrays of *rle_op* structures instead of just pixels. If the data is already available in something close to this form, however, *sv_putraw* will run much more quickly than *sv_putrow*. An *rle_op* is a structure with the following contents:

opcode	The type of data. One of *ByteData* or *RunData*.
xloc	The X location within the scanline at which this data begins.
length	The length of the data. This is either the number of pixels that are the same color, for a run, or the number of pixels provided as byte data.
pixels	A pointer to an array of pixel values. This field is used only for the *ByteData* opcode.
run_val	The pixel value for a *RunData* opcode.

Since there is no guarantee that the different color channels will require the same set of *rle_ops* to describe their data, a separate count must be provided for each channel. Here is a sample call to *sv_putraw*:

```
int nraw[3];    /* Length of each row */
rle_op *rows[3];/* Data pointers */
sv_putraw( rows, nraw, &globals );
```

A more complete example of the use of *sv_putraw* will be given in connection with the description of *rle_getraw*, below.

Calls to *sv_putrow* and *sv_putraw* may be freely intermixed, as required by the application.

2.5. Reading RLE Files

Reading an RLE file is much like writing one. An initial call to a setup routine reads the file header and fills in the *globals* structure. Then, a scanline at a time is read by calling *rle_getrow* or *rle_getraw*.

The calling program is responsible for opening the input file. A call to *rle_get_setup* will then read the header information and fill in the supplied globals structure. The return code from *rle_get_setup* indicates a variety of errors, such as the input file not being an RLE file, or encountering an EOF while reading the header.

Each time *rle_getrow* is called, it fills in the supplied scanline buffer with one scanline of image data and returns the Y position of the scanline (which will be one greater than the previous time it was called). Depending on the setting of the *background* flag, the scanline buffer may or may not be cleared to the background color on each call. If it is not (*background* is 0 or 1), and if the caller does not clear the buffer between scanlines, then a "smearing" effect will be seen, if some pixels from previous scanlines are not overwritten by pixels on the current scanline. Note that if *background* is 0, then no background color was supplied, and setting *background* to 2 to try to get automatic buffer clearing will usually cause a segmentation fault when *rle_getrow* tries to get the background color through the *bg_color* pointer.

```
#include <svfb_global.h>

main()
{
        rle_pixel scanline[3][512], alpha[512], *rows[4];
        int y, i;

        /* Most of the default values in sv_globals are ok */
        /* We do have an alpha channel, though */
        sv_globals.sv_alpha = 1;
        SV_SET_BIT( sv_globals, SV_ALPHA );

        rows[0] = alpha;
        for ( i = 0; i < 3; i++ )
                rows[i+1] = scanline[i];

        sv_setup( RUN_DISPATCH, &sv_globals );

        /* Create output for 512 x 480 (default size) display */
        for ( y = 0; y < 480; y++ )
        {
                mk_scanline( y, scanline, alpha );
                sv_putrow( rows, 512, &sv_globals );
        }
        sv_puteof( &sv_globals );
}
```

Figure 2-3: Example of use of sv_putrow

Figure 2-4 shows an example of the use of *rle_getrow*. Note the dynamic allocation of scanline storage space, and compensation for presence of an alpha channel. A subroutine, *rle_row_alloc*, is available that performs the storage allocation automatically. It is described below. If the alpha channel were irrelevant, the macro *SV_CLR_BIT* could be used to inhibit reading it, and no storage space would be needed for it.

The function *rle_getraw* is the inverse of *sv_putraw*. When called, it fills in the supplied buffer with raw data for a single scanline. It returns the scanline *y* position, or 2^{15} to indicate end of file. It is assumed that no image will have more than $2^{15}-1$ scanlines. A complete program (except for error checking) that reads an RLE file from standard input and produces a negative image on standard output is shown in Figure 2-5.

The functions *rle_row_alloc* and *rle_raw_alloc* simplify allocation of buffer space for use by the rle routines. Both use a supplied globals structure to determine how many and which channels need buffer space, as well as the size of the buffer for each scanline. The returned buffer pointers will be adjusted for the presence of an alpha channel, if it is present. Buffer space for pixel or *rle_op* data will be allocate only for those channels that have bits set in the channel bitmap. The buffer space may be freed by calling *rle_row_free* or *rle_raw_free*, respectively.

3. Comments, issues, and directions

Some comments on the file format and current subroutine implementation:

- The background color for the alpha channel is always 0.
- All channels must have the same number of bits. This could be a problem when saving, e.g., Z values, or if more than 8 bits of precision were desired for the alpha channel.
- Pixels are skipped (by *sv_putrow*) only if all channel values of the pixel are equal to the corresponding background color values.
- The current Implementation of *sv_putrow* skips pixels only if at least 2 adjacent pixels are equal to the background. The SkipPixels operation is intended for efficiency, not to provide cheap compositing.

```
/* An example of using rle_getrow */
/* Scanline pointer */
rle_pixel ** scan;
int i;

/* Read the RLE file from stdin */
rle_get_setup( &globals );

/* Allocate enough space for scanline data, including alpha channel */
/* (Should check for non-zero return, indicating a malloc error) */
rle_row_alloc( &globals, &scan );

/* Read scanline data */
while ( (y = rle_getrow( &globals, stdin, scan ) <= globals.sv_ymax )
        /* Use the scanline data */;
```

Figure 2-4: Example of rle_getrow use.

- Nothing forces the image data to lie within the bounds declared in the header. However, *rle_getrow* will not write outside these bounds, to prevent core dumps. No such protection is provided by *rle_getraw*.
- Images saved in RLE are usually about 1/3 their original size (for an "average" image). Highly complex images may end up slightly larger than they would have been if saved by the trivial method.

We have not yet decided how pixels with other than 8 bits should be packed into the file. To keep the file size down, one would like to pack *ByteData* as tightly as possible. However, for interpretation speed, it would probably be better to save one value in each (*pixelbits*+7)/8 bytes.

Some proposed enhancements include:

- A "ramp" opcode. This specifies that pixel values should be linearly ramped between two values for a given number of pixels in the scanline. This opcode would be difficult to generate from an image, but if an application knew it was generating a ramp, it could produce significant file size savings (e.g. in Gouraud shaded images).
- Opcodes indicating that the current scanline is identical to the previous, or that it differs only slightly (presumably followed by standard opcodes indicating the difference). Detection of identical scanlines is easy, deciding that a scanline differs slightly enough to warrant a differential description could be difficult. In images with large areas with little change, this could produce size savings[2]

The subroutine library is still missing some useful functions. Some proposed additions are:

- Conversion from "raw" to "row" format, and back. One could then view *sv_putrow* as being a "raw" to "row" conversion followed by a call to *sv_putraw*, and *rle_getrow* as a call to *rle_getraw* followed by "row" to "raw" conversion.
- A function to merge several channels of "raw" data into a single channel. For example, this would take separate red, green, and blue channels and combine them into a single RGB channel. This would be useful for RLE interpretation on devices that do not easily support the separate channel paradigm, while preserving the efficiency of the "raw" interface. It could also be used to increase the efficiency of a compositing program.

The Utah RLE format has developed and matured over a period of about six years, and has proven to be versatile and useful for a wide variety of applications that require image transmittal and storage. It provides a compact, efficiently interpreted image storage capability. We expect to see continued development of capabilities and utility, but expect very little change in the basic format.

---

[2] This suggestion was inspired by a description of the RLE format used at Ohio State University.

```
#include <stdio.h>
#include <svfb_global.h>
#include <rle_getraw.h>

main()
{
    struct sv_globals in_glob, out_glob;
    rle_op ** scan;
    int * nraw, i, j, c, y, newy;

    in_glob.svfb_fd = stdin;
    rle_get_setup( &in_glob );
    /* Copy setup information from input to output file */
    out_glob = in_glob;
    out_glob.svfb_fd = stdout;

    /* Get storage for calling rle_getraw */
    rle_raw_alloc( &in_glob, &scan, &nraw );

    /* Negate background color! */
    if ( in_glob.sv_background )
        for ( i = 0; i < in_glob.sv_ncolors; i++ )
            out_glob.sv_bg_color[i] = 255 - out_glob.sv_bg_color[i];

    /* Init output file */
    sv_setup( RUN_DISPATCH, &out_glob );

    y = in_glob.sv_ymin;
    while ( (newy = rle_getraw( &in_glob, scan, nraw )) != 32768 ) {
        /* If > one line skipped in input, do same in output */
        if ( newy - y > 1 )
            sv_skiprow( &out_glob, newy - y );
        y = newy;
        /* Map all color channels */
        for ( c = 0; c < out_glob.sv_ncolors; c++ )
            for ( i = 0; i < nraw[c]; i++ )
                switch( scan[c][i].opcode ) {
                case RRunDataOp:
                    scan[c][i].u.run_val = 255 - scan[c][i].u.run_val;
                    break;
                case RByteDataOp:
                    for ( j = 0; j < scan[c][i].length; j++ )
                        scan[c][i].u.pixels[j] =
                            255 - scan[c][i].u.pixels[j];
                    break;
                }
        sv_putraw( scan, nraw, &out_glob );
        /* Free raw data */
        rle_freeraw( &in_glob, scan, nraw );
    }
    sv_puteof( &out_glob );

    /* Free storage */
    rle_raw_free( &in_glob, scan, nraw );
}
```

Figure 2-5: Program to produce a negative of an image

4. 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.

Table of Contents

List of Figures