10 Image Files (.rla)

The .rla (run-length encoded, version A) file format is a machine-independent format used for storing images in the Advanced Visualizer.

Sample programs are provided in the $WF_AV_DIR/samples directory. These programs read and write .rla files that are compatible with the Advanced Visualizer.

Format

As shown in figure 10-1, .rla files consist of three parts:

- a *file header* that gives basic information about the file
- an *offset table* that tells where in the file the data for each scanline starts
- the *image data* that contains a series of encoded data strings

All .rla filenames must end with the extension .rla.

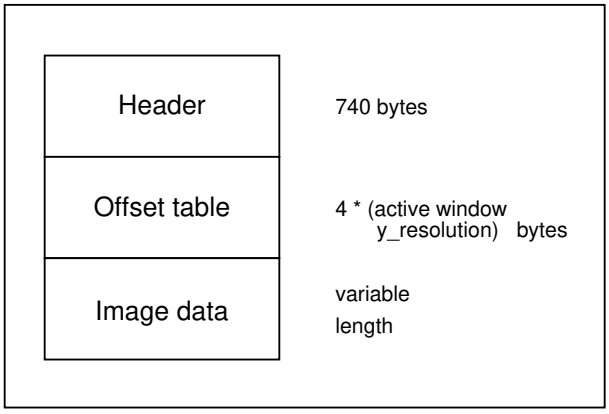

Figure 10-1. File layout (.rla)

In the .rla file format, data is stored as integer or ASCII data in the native byte order of 68000-based machines; that is, byte 0 is the high-order byte. All floating point values in the header are stored as ASCII strings. Image data must be written sequentially.

Scanlines can be read sequentially or at random using the offset table.

The file header

The *file header* contains basic information about the image, such as its size, the color space it uses, and so on.

In figure 10-2, a C programming structure is illustrated that represents the header. Each field in the header is described in alphabetical order under “Header fields” on page 10-2.

Header fields

This section describes each field in the header in alphabetical order.

Field names are shown in bold-face type. All fields are required and must be specified to ensure proper header size (740 bytes).

Arbitrary values can be used for some fields when associated information is not available. These optional fields are shown in italicized type.

active_window.left
active_window.right
active_window.bottom
active_window.top

These fields specify the left, right, bottom, and top boundaries of the non-zero portion of the stored image. These are the boundaries of the meaningful data within the image window. Image writes data for only the active window area. If data for the entire active window area is not provided, errors may result.

aspect

Describes the aspect format for the file. Formats are defined in the .imfrc file.

```
typedef struct {
    short   left, right, bottom, top;
} WINDOW_S;

typedef struct {
    WINDOW_S window;
    WINDOW_S active_window;
    short   frame;
    short   storage_type;
    short   num_chan;
    short   num_matte;
    short   num_aux;
    short   revision;
    char    gamma[16];
    char    red_pri[24];
    char    green_pri[24];
    char    blue_pri[24];
    char    white_pt[24];
    long    job_num;
    char    name[128];
    char    desc[128];
    char    program[64];
    char    machine[32];
    char    user[32];
    char    date[20];
    char    aspect[24];
    char    aspect_ratio[8];
    char    chan[32];
    short   field;
    char    time[12];
    char    filter[32];
    short   chan_bits;
    short   matte_type;
    short   matte_bits;
    short   aux_type;
    short   aux_bits;
    char    aux[32];
    char    space[36];
    long    next;
} HEADER;
Structure of header
```

Figure 10-2. C programming structure

aspect_ratio

A floating point number equal to the picture aspect ratio for the file. The picture aspect ratio is picture width divided by the picture height of the target display device (in physical units). It is used to determine the pixel aspect ratio.

The aspect ration field is useful in a case where, the image window does not match the dimensions of the format referenced by the aspect type.

aux

Specifies the auxiliary channel data as either *range* or *depth.*

aux_bits

Specifies the bit precision of each pixel's auxiliary channel. The range is 1 to 32.

aux_type

Specifies the type of auxiliary channel. Auxiliary channel types are encoded as follows:

Type 0 encodes auxiliary channel as integer data.

Type 4 encodes the auxiliary channel as 4-byte IEEE float data.

revision

A revision identifier. Current revision is *0xfffe*.

chan

Identifies the color space for the image.

This can be rgb, xyz, sampled, or raw.

chan_bits

Specifies the bit precision of each of the pixel's image (color) channel. Range is 1 to 32.

date

Specifies the date that the file was created.

Any valid character string (up to 20 characters) is acceptable. Internally, Wavefront uses the format: *MMM DD hh:mm yyyy*, such as Sep 30 12:29 1993.

desc

A description of the file's contents.

This can be any character string 128 characters or less.

frame

Specifies the frame number in a sequence of frames.

Can be any integer greater than 0.

field

Specifies that the image contains field- rendered data. A value of 1 specifies field-rendered data; 0 specifies non-field-rendered data.

filter

Specifies the name of the filter used to post-process the image data.

gamma

Specifies the gamma that was applied to the file for storage.

Must be a floating point number greater than 0. Use a value of 2.2 for writing, if in doubt.

job_num

Specifies the job number under which the file was rendered. The job number is an identifier for the user.

Can be any integer**.**

matte_bits

Specifies the bit precision of each pixel's matte channel. The range is 1 to 32.

matte_type

Specifies the type of matte channel. The matte channels are encoded as follows:

Type 0 encodes the matte channel as integer data.

Type 4 encodes the matte channel as 4-byte IEEE float data.

machine

Specifies the machine on which the file was created.

Can be any character string of 32 characters or less.

name

Specifies the filename used when the file was created.

Can be any character string of 128 characters or less.

next

Specifies the byte offset relative to the beginning of the file at which the next header record/offset table begins. This is zero for single-image frames. This identifies the location of the next sub-image, which typically will be a swatch (if preset).

num_aux

Specifies the number of auxiliary channels in the file. Auxiliary channel information could include distance from the eye or the direction of normal.

Zero is the usual value.

num_chan

Specifies the number of image channels in the file.

There are typically three image channels: red, green, and blue.

num_matte

Specifies the number of matte channels in the image file. Normally there is one matte channel, however, there may be as many matte channels as image channels, if the transparency information is spectrally based.

program

Specifies the name of the program that created the image.

Can be any character string of 64 characters or less.

red_pri
green_pri
blue_pri
white_pt

Specifies the chromaticities of the red, green, and blue primaries, and the white point. Each field contains the x, y chromaticity as two floating point numbers separated by space.

Typical default values are the NTSC standard values listed:

Primary	x	y
red	0.670	0.080
green	0.210	0.710
blue	0.140	0.330
white	0.310	0.316

The printf format "%7.4f %7.4f" is typically used in writing these fields.

space

Unused expansion space. This must be all zeros.

storage_type

Specifies the type of image channels. The image channel types are encoded as follows:

Type 0 encodes the image channel as 4-byte IEEE float data.

Type 4 encodes the image channel as floating point numbers.

time

Specifies the amount of time required to create the image file. This is typically the rendering time.

user

Identifies the user who ran the creating program.

Can be any character string of 32 characters or less.

window.bottom
window.left
window.right
window.top

These four fields specify the bottom, left, right, and top boundaries of the complete image. The left corner is normally at left = 0, bottom = 0. A valid image window must have right greater than left and top greater than bottom.

How data is encoded

Each scanline of an image is comprised of 1 more channels, where each channel represents a different type of information. Typically, a scanline consists of four channels: red, green, blue, and matte.

To reduce file size, the channels of each scanline are encoded separately, thus compressing the image horizontally.

There is no fixed length for encoded channels. The length is determined by how well the encoding algorithm can compress the channel of data. Because the encoded channels are of different lengths, a mechanism is needed to locate the start of data for an arbitrary scanline. The offset table serves this purpose.

The offset table

The offset table identifies where in the file data starts for an arbitrary scanline. It points to the beginning of the red channel record, since this is the start of the encoded line. The offset table has one entry for each scanline in the active image window.

The offset value is from the beginning of the file. It is determined using the standard C *ftell* routine when the file is created.

Each encoded channel consists of a *length* and a series of *runs*.

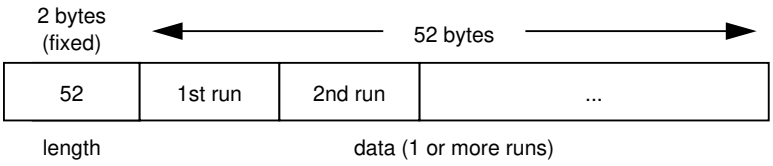

Figure 10-3. Sample encoding for the red channel of a scanline

Length and runs

The *length* is a 2-byte integer that gives the length of the encoded channel data in bytes. *Runs* are series of pixels that may or may not be encoded.

The encoding scheme for each run is very basic. Repeats of a value can be encoded into a repeat count and a repeat value (only repeats of three or more are enclosed).

The encoding algorithms differ slightly depending on the precision being stored—1 to 16 bits (32-bit data is not encoded).

Byte data (8 bits or less)

An 8-bit run begins with a one-byte count. If the count is negative, the following "count" bytes consist of unencoded channel data. If the count is positive, the next byte is repeated "count+1" times.

The following example run contains six identical values.

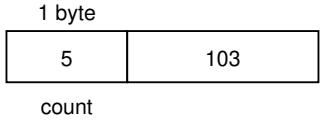

Figure 10-4. Sample encoded run

9 to 16 bits

For channel data of length between 9 and 16 bits, the situation is a little more complex. Each pixel datum is stored in a 2-byte integer (short). When encoding, the algorithm for encoding byte data is applied to the most significant byte of each datum for the length of the active x resolution. Then the algorithm is applied again to the least significant byte of each datum. Thus each channel of short data is encoded into two sets of runs. This example illustrates the algorithm:

The original data to be encoded is:

```
0x00fe 0x00fe 0x00fe 0x00ff 0x0100
0x0100 0x0100 0x0100 0x0101 0x0101
```

The encoded run is:

```
3 0x00 5 0x01          for the most significant bytes

2 0xfe -1 0xff         for the least significant bytes
3 0x00 -2 0x0101
```

The most significant bytes of the first four pixel values are 0x00. This satisfies the condition for an encoded run and 0x0300 is output. The next six most significant bytes of the pixel values are 0x01. Another encoded run is output: 0x0501. Now the least significant bytes are checked. The least significant bytes of the first three pixel values are 0xfe. An encoded run is output: 0x02fe. The least significant byte of the next pixel value is not repeated, thus an unencoded run is output: -1 0xff. The next four least significant bytes of the pixel values are 0x00, producing an encoded run: 0x0300. The least significant bits of the final two

pixel values do not satisfy the criteria for an encoded run (that is, must be at least three identical), therefore an unencoded run is produced: -2 0x01.

The complete channel record is:

```
0x000c0300050102fe –1 ff0300 –2 0101.
```

Greater than 16 bits

A similar algorithm encodes channel data of length between 17 and 32 bits of precision.

Floating point numbers (32 bits)

The algorithm for storing floating point channel data stores unencoded runs of 32-bit IEEE numbers.