`writeoct(int store, CUBE *scene, char *ofn[]);`

> Write the octree stored in `scene` to `stdout`, assuming the header has already been written. The flags in `store` determine what will be included. Normally, this variable is one of `IO_ALL` or `(IO_ALL & ~IO_FILES)` corresponding to writing a normal or a frozen octree, respectively.

Here is the main call for reading in an octree:

`readoct(char *fname, int load, CUBE *scene, char *ofn[]);`

> Read the octree file `fname` into `scene`, saving scene file names in the `ofn` array. What is loaded depends on the flags in `load`, which may be one or more of `IO_CHECK, IO_INFO, IO_SCENE, IO_TREE, IO_FILES` and `IO_BOUNDS`. These correspond to checking file type and consistency, transferring the information header to `stdout`, loading the scene data, loading the octree structure, assigning the scene file names to `ofn`, and assigning the octree cube boundaries. The macro IO_ALL includes all of these flags, for convenience.

Picture File Format (.pic suffix)

Radiance pictures differ from standard computer graphics images inasmuch as they contain real physical data, namely radiance values at each pixel. To do this, it is necessary to maintain floating point information, and we use a 4-byte/pixel encoding described in Chapter II.5 of *Graphics Gems II* [Arvo91,p.80]. The basic idea is to store a 1-byte mantissa for each of three primaries, and a common 1-byte exponent. The accuracy of these values will be on the order of 1% (+/-1 in 200) over a dynamic range from 10^-38 to 10^38.

Although *Radiance* pictures *may* contain physical data, they do not *necessarily* contain physical data. If the rendering did not use properly modeled light sources, or the picture was converted from some other format, or custom filtering was applied, then the physical data will be invalid. Table 6 lists programs that read and write *Radiance* pictures, with pluses next to the X-marks indicating where physical data is preserved (or at least understood). Specifically, if the picture file read or written by a program has an "X+", then it has maintained the physical validity of the pixels by keeping track of any exposure or color corrections in the appropriate header variables, described below.

Basic File Structure

Radiance picture files are divided into three sections: the information header, the resolution string, and the scanline records. All of these must be present or the file is incomplete.

Information Header

The information header begins with the usual "`#?RADIANCE`" identifier, followed by one or more lines containing the programs used to produce the picture. These commands may be interspersed with variables describing relevant information such as the view, exposure, color correction, and so on. Variable assignments begin on a new line, and the variable name (usually all upper case) is followed by an equals sign ('='), which is followed by the assigned value up until the end of line. Some variable assignments override previous assignments in the same header, where other assignments are cumulative. Here are the most important variables for *Radiance* pictures:

`FORMAT`
: A line indicating the file's format. At most one `FORMAT` line is allowed, and it must be assigned a value of either "`32-bit_rle_rgbe`" or "`32-bit_rle_xyze`" to be a valid *Radiance* picture.

`EXPOSURE`
: A single floating point number indicating a multiplier that has been applied to all the pixels in the file. `EXPOSURE` values are cumulative, so the original pixel values (i.e., radiances in watts/steradian/m^2) must be derived by taking the values in the file and dividing by all the `EXPOSURE` settings multiplied together. No `EXPOSURE` setting implies that no exposure changes have taken place.

`COLORCORR`

A color correction multiplier that has been applied to this picture. Similar to the `EXPOSURE` variable except given in three parts for the three primaries. In general, the value should have a brightness of unity, so that it does not affect the actual brightness of pixels, which should be tracked by `EXPOSURE` changes instead. (This variable is also cumulative.)

`SOFTWARE`

The software version used to create the picture, usually something like "`RADIANCE 3.04 official release July 16, 1996`".

`PIXASPECT`

The pixel aspect ratio, expressed as a decimal fraction of the height of each pixel to its width. This is not to be confused with the image aspect ratio, which is the total height over width. (The image aspect ratio is actually equal to the height in pixels over the width in pixels, *multiplied* by the pixel aspect ratio.) These assignments are cumulative, so the actual pixel aspect ratio is all ratios multiplied together. If no `PIXASPECT` assignment appears, the ratio is assumed to be 1.

`VIEW`

The *Radiance* view parameters used to create this picture. Multiple assignments are cumulative inasmuch as new view options add to or override old ones.

`PRIMARIES`

The CIE (x,y) chromaticity coordinates of the three (RGB) primaries and the white point used to standardize the picture's color system. This is used mainly by the **ra_xyze** program to convert between color systems. If no `PRIMARIES` line appears, we assume the standard primaries defined in `src/common/color.h`, namely "`0.640 0.330 0.290 0.600 0.150 0.060 0.333 0.333`" for red, green, blue and white, respectively.

As always, the end of the header is indicated by an empty line.

Resolution String

All *Radiance* pictures have a standard coordinate system, which is shown in Figure 4. The origin is always at the lower left corner, with the X coordinate increasing to the right, and the Y coordinate increasing in the upward direction. The actual ordering of pixels in the picture file, however, is addressed by the resolution string.

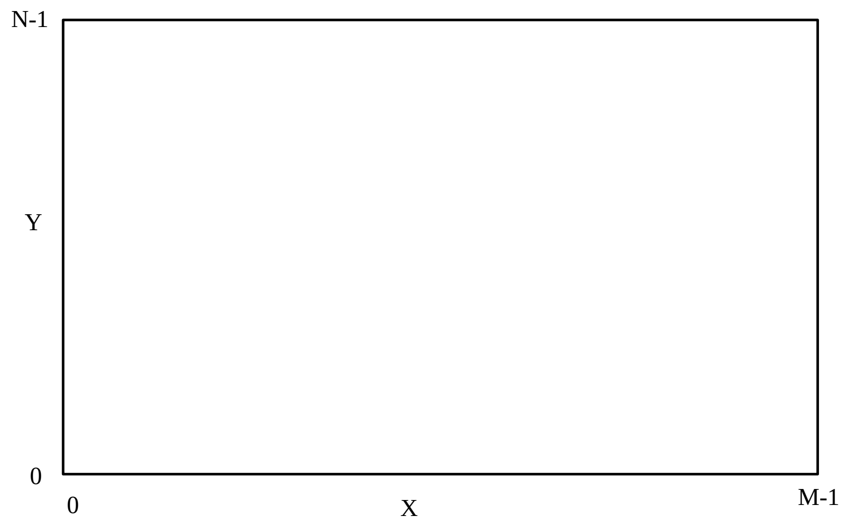

Figure 4. The standard coordinate system for an MxN picture.

The resolution string is given as one of the following:

`-Y N +X M`
: The standard orientation produced by the renderers, indicating that Y is decreasing in the file, and X is increasing. X positions are increasing in each scanline, starting with the upper left position in the picture and moving to the upper right initially, then on down the picture. Some programs will only handle pictures with this ordering.

`-Y N -X M`
: The X ordering has been reversed, effectively flipping the image left to right from the standard ordering.

`+Y N -X M`
: The image has been flipped left to right and also top to bottom, which is the same as rotating it by 180 degrees.

`+Y N +X M`

The image has been flipped top to bottom from the standard ordering.

`+X M +Y N`

The image has been rotated 90 degrees clockwise.

`-X M +Y N`

The image has been rotated 90 degrees clockwise, then flipped top to bottom.

`-X M -Y N`

The image has been rotated 90 degrees counter-clockwise.

`+X M -Y N`

The image has been rotate 90 degrees counter-clockwise, then flipped top to bottom.

The reason for tracking all these changes in picture orientation is so programs that compute ray origin and direction from the `VIEW` variable in the information header will work despite such changes. Also, it can reduce memory requirements on converting from other image formats that have a different scanline ordering, such as Targa.

Scanline Records

Radiance scanlines come in one of three flavors, described below:

Uncompressed

Each scanline is represented by M pixels with 4 bytes per pixel, for a total length of 4xM bytes. This is the simplest format to read and write, since it has a one-to-one correspondence to an array of `COLR` values.

Old run-length encoded

Repeated pixel values are indicated by an illegal (i.e., unnormalized) pixel that has 1's for all three mantissas, and an exponent that corresponds to the number of times the previous pixel is repeated. Consecutive repeat indicators contain higher-order bytes of the count.

New run-length encoded

In this format, the four scanline components (three primaries and exponent) are separated for better compression using adaptive run-length encoding (described by Glassner in Chapter II.8 of *Graphics Gems II* [Arvo91,p.89]). The record begins with an unnormalized pixel having two bytes equal to 2, followed by the upper byte and the lower byte of the scanline length (which must be less than 32768). A run is indicated by a byte with its high-order bit set, corresponding to a count with excess 128. A non-run is indicated with a byte less than 128. The maximum compression ratio using this scheme is better than 100:1, but typical performance for *Radiance* pictures is more like 2:1.

The physical values these scanlines correspond to depend on the format and other information contained in the information header. If the `FORMAT` string indicates RGB data, then the units for each primary are spectral radiances over the corresponding waveband, such that a pixel value of (1,1,1) corresponds to a total energy of 1 watt/steradian/sq.meter over the visible spectrum. The actual luminance value (in lumens/steradian/sq.meter) can be computed from the following formula for the standard *Radiance* RGB primaries:

```
luminance = 179 * (0.265*R + 0.670*G + 0.065*B)
```

The value of 179 lumens/watt is the standard *luminous efficacy* of equal-energy white light that is defined and used by *Radiance* specifically for this conversion. This and the other values above are defined in `src/common/color.h`, and the above formula is given as a macro, `luminance(col)`.

If the `FORMAT` string indicates XYZ data, then the units of the Y primary are already lumens/steradian/sq.meter, so the above conversion is unnecessary.

Radiance programs

Table 6 shows the many programs that read and write *Radiance* pictures.

Program	**Read**	**Write**	**Function**
falsecolor	X+	X	Create false color image
findglare	X+		Find sources of discomfort glare
getinfo	X		Print information header from binary file
macbethcal	X	X	Compute image color & contrast correction
normpat	X	X	Normalize picture for use as pattern tile
objpict		X	Generate composite picture of object
pcomb	X+	X	Perform arbitrary math on picture(s)
pcond	X+	X	Condition a picture for display
pcompos	X	X	Composite pictures
pextrem	X+		Find minimum and maximum pixels
pfilt	X+	X+	Filter and anti-alias picture
pflip	X+	X+	Flip picture left-right and/or top-bottom
pinterp	X+	X+	Interpolate/extrapolate picture views
protate	X+	X+	Rotate picture 90 degrees clockwise
psign		X	Create text picture
pvalue	X+	X+	Convert picture to/from simpler formats
ra_avs	X	X	Convert to/from AVS image format
ra_pict	X	X	Convert to/from Macintosh PICT2 format
ra_ppm	X	X	Convert to/from Poskanzer Port. Pixmap
ra_pr	X	X	Convert to/from Sun 8-bit rasterfile
ra_pr24	X	X	Convert to/from Sun 24-bit rasterfile
ra_ps	X		Convert to B&W or color PostScript
ra_rgbe	X	X	Convert to/from uncompressed picture
ra_t8	X	X	Convert to/from Targa 8-bit format
ra_t16	X	X	Convert to/from Targa 16-bit and 24-bit
ra_tiff	X	X	Convert to/from TIFF 8-bit and 24-bit
ra_xyze	X	X	Convert to/from CIE primary picture
rad		X+	Render *Radiance* scene
ranimate		X+	Animate *Radiance* scene
rpict	X	X+	Batch rendering program
rpiece	X	X+	Parallel batch rendering program
rtrace	X	X+	Customizable ray-tracer

rview	X	X+	Interactive renderer
vwright	X		Get view parameters and shift them
xglaresrc	X		Display glare sources from **findglare**
ximage	X+		Display *Radiance* picture under *X11*
xshowtrace	X		Show ray traces on *X11* display

Table 6. *Radiance* programs that read and write picture files. Pluses indicate when a program makes use of or preserves physical pixel values.

Radiance C Library

There are a fair number of routines for reading, writing and manipulating *Radiance* pictures. If you want to write a converter to or from a 24-bit image format, you can follow the skeletal example in `src/px/ra_skel.c`. This has all of the basic functionality of the other **ra_*** image conversion programs, with the actual code for the destination type removed (or simplified). The method in `ra_skel` uses the routines in `src/common/colrops.c` to avoid conversion to machine floating point, which can slow things down and is not necessary in this case.

Below we describe routines for reading and writing pictures, which rely heavily on definitions in `src/common/color.h`. We start with the calls for manipulating information headers, followed by the calls for resolution strings, then the calls for scanline records.

Information headers are manipulated with the routines in `src/common/header.c` and the macros in `color.h`. Features for handing views are defined in `src/common/view.h` with routines in `src/common/image.c`. Here are the relevant calls for reading and copying information headers:

`int checkheader(FILE *fin, char *fmt, FILE *fout);`

Read the header information from `fin`, copying to `fout` (unless `fout` is `NULL`), checking any `FORMAT` line against the string `fmt`. The `FORMAT` line (if it exists) will not be copied to `fout`. The function returns 1 if the header was OK and the format matched, 0 if the header was OK but there was no format line, and -1 if the format line did not match or there was some problem reading the header. Wildcard characters ('*' and '?') may appear in `fmt`, in which case a globbing match is applied, and the matching format value will be copied to `fmt` upon success. The normal `fmt` values for pictures are `COLRFMT` for *Radiance* RGB, `CIEFMT` for 4-byte XYZ pixels, or a copy of `PICFMT` for glob matches to either. (Do not pass `PICFMT` directly, as this will cause an illegal memory access on systems that store static strings in read-only memory.)

`int getheader(FILE *fp, int (*f)(), char *p);`

For those who need more control, `getheader` reads the header from `fp`, calling the function `f` (if not `NULL`) with each input line and the client data pointer `p`. A simple call to skip the header is `getheader(fp,NULL,NULL)`. To copy the header unconditionally to `stdout`, call `getheader(fp,fputs,stdout)`. More often, `getheader` is called with a client function, which checks each line for specific variable settings.

`int isformat(char *s);`

Returns non-zero if the line `s` is a `FORMAT` assignment.

`int formatval(char *r, char *s);`

Returns the `FORMAT` value from line `s` in the string `r`. Returns non-zero if `s` is a valid format line.

`fputformat(char *s, FILE *fp);`

Put format assignment `s` to the stream `fp`.

`isexpos(s)`

Macro returns non-zero if the line `s` is an `EXPOSURE` setting.

`exposval(s)`

Macro returns **double** exposure value from line `s`.

`fputexpos(ex,fp)`

Macro puts real exposure value `ex` to stream fp.

`iscolcor(s)`

Macro returns non-zero if the line `s` is a `COLORCORR` setting.

`colcorval(cc,s)`

Macro assign color correction value from line `s` in the `COLOR` variable `cc`.

`fputcolcor(cc,fp)`

Macro puts correction `COLOR` `cc` to stream fp.

`isaspect(s)`

Macro returns non-zero if the line `s` is a `PIXASPECT` setting.

`aspectval(s)`

Macro returns **double** pixel aspect value from line `s`.

`fputaspect(pa,fp)`

Macro puts real pixel aspect value `pa` to stream fp.

`int isview(char *s);`

Returns non-zero if header line `s` contains view parameters. Note that `s` could be either a VIEW assignment or a rendering command.

`int sscanview(VIEW *vp, char *s);`

Scan view options from the string `s` into the `VIEW` structure pointed to by `vp`.

`fprintview(VIEW *vp, FILE *fp);`

Print view options in `vp` to the stream `fp`. Note that this does not print out "`VIEW=`" first, or end the line. Therefore, one usually calls `fputs(VIEWSTR,fp)` followed by `fprintview(vp,fp)`, then `putc('\n',fp)`.

`isprims(s)`

Macro returns non-zero if the line `s` is a `PRIMARIES` setting.

`primsval(p,s)`

Macro assign color primitives from line `s` in the `RGBPRIMS` variable `p`.

`fputprims(p,fp)`

Macro puts color primitives `p` to stream fp.

The header file `src/common/resolu.h` has macros for resolution strings, which are handled by routines in `src/common/resolu.c`. Here are the relevant calls:

`fgetsresolu(rs,fp)`

Macro to get a resolution string from the stream `fp` and put it in the `RESOLU` structure pointed to by `rs`. The return value is non-zero if the resolution was successfully loaded.

`fputsresolu(rs,fp)`

Macro to write the `RESOLU` structure pointed to by `rs` to the stream `fp`.

`scanlen(rs)`

Macro to get the scanline length from the `RESOLU` structure pointed to by `rs`.

`numscans(rs)`

Macro to get the number of scanlines from the `RESOLU` structure pointed to by `rs`.

`fscnresolu(slp,nsp,fp)`

Macro to read in a resolution string from `fp` and assign the scanline length and number of scanlines to the integers pointed to by `slp` and `nsp`, respectively. This call expects standard English-text ordered scanlines, and returns non-zero only if the resolution string agrees.

`fprtresolu(sl,ns,fp)`

Macro to print out a resolution string for `ns` scanlines of length `sl` in standard English-text ordering to `fp`.

The file `src/common/color.c` contains the essential routines for reading and writing scanline records. Here are the relevant calls and macros:

`int freadcolrs(COLR *scn, int sl, FILE *fp);`

Read a scanline record of length `sl` from stream `fp` into the `COLR` array `scn`. Interprets uncompressed, old, and new run-length encoded records. Returns 0 on success, -1 on failure.

`int fwritecolrs(COLR *scn, int sl, FILE *fp);`

Write the scanline record stored in the `COLR` array `scn`, length `sl`, to the stream `fp`. Uses the new run-length encoding unless `sl` is less than 8 or greater than 32767, when an uncompressed record is written. Returns 0 on success, -1 if there was an error.

`int freadscan(COLOR *fscn, int sl, FILE *fp);`

Reads a scanline of length `sl` from the stream `fp` and converts to machine floating-point format in the array `fscn`. Recognizes all scanline record encodings. Returns 0 on success, or -1 on failure.

`int fwritescan(COLOR *fscn, int sl, FILE *fp);`

Write the floating-point scanline of length `sl` stored in the array `fscn` to the stream `fp`. Converts to 4-byte/pixel scanline format and calls `fwritecolrs` to do the actual write. Returns 0 on success, or -1 if there was an error.

`colval(col,p)`

Macro to get primary `p` from the floating-point `COLOR` `col`. The primary index may be one of `RED`, `GRN` or `BLU` for RGB colors, or `CIEX`, `CIEY` or `CIEZ` for XYZ colors. This macro is a valid lvalue, so can be used on the left of assignment statements as well.

`colrval(clr,p)`

Macro to get primary `p` from the 4-byte `COLR` pixel `clr`. The primary index may be one of `RED`, `GRN` or `BLU` for RGB colors, or `CIEX`, `CIEY` or `CIEZ` for XYZ colors. Unless just one primary is needed, it is more efficient to call `colr_color` and use the `colval` macro on the result.

`colr_color(COLOR col, COLR clr);`

Convert the 4-byte pixel `clr` to a machine floating-point representation and store the result in `col`.

`setcolr(COLR clr, double p1, p2, p3);`

Assign the 4-byte pixel `clr` according to the three primary values `p1`, `p2` and `p3`. These can be either *Radiance* RGB values or CIE XYZ values.