Radiance File Formats

This chapter discusses the standard file formats specific to *Radiance*, and gives their internal structure, with pointers to routines for reading and writing them. The following file formats (listed with their conventional suffixes) are covered:

Scene Description (.rad suffix)

This is the main input file type, describing materials and geometry for the rendering programs, and must be compiled into an octree by **oconv** prior to ray-tracing. It is an ASCII text format, and is often translated from a CAD description, but may be created or edited by a text editor as well.

Function File (.cal suffix)

Also a text format, these files describe mathematical patterns, textures, and surface shapes. In the case of patterns and textures, the functions serve as input directly to the rendering programs. In the case of surfaces, the functions serve as input to one of the generator programs, **gensurf**, **genrev** or **genworm**. Additionally, **pcomb** may be used to perform math on *Radiance* pictures and the **rcalc** utility may be used in creative ways to manipulate data for scene generation and data analysis.

Data File (.dat suffix)

Another ASCII format, data files are used directly by the rendering programs to interpolate values for luminaire photometry, among other things.

Font File (.fnt suffix)

A simple, polygonal font representation for rendering text patterns. This ASCII format describes each character "glyph" as a sequence of vertices in rectangular, integer coordinates ranging from 0 to 255.

Octree (.oct suffix)

A binary data structure computed from one or more scene description files by the **oconv** program. It may contain frozen scene data in binary form, or merely references to the original scene files.

Picture (.pic suffix)

A binary image file containing calibrated, real radiance values at each pixel. *Radiance* pictures may be displayed, analyzed, and converted to other image formats.

Z-buffer (.zbf suffix)

A binary file with the distances to each pixel in a corresponding picture.

Ambient File (.amb suffix)

A binary file used to store diffuse interreflection values, which are shared between cooperating rendering processes running sequentially or in parallel. Since these values are view-independent, sharing this information across multiple runs is highly economical.

We will discuss each of these formats in turn, giving examples and pointers to routines in the source code for reading and writing them, and the programs that use them. In general, the ASCII text formats have no standard routines for writing them, since they generally originate outside of *Radiance* or are created with simple *printf(3)* statements. Most binary file formats are machine and system independent, meaning they can be moved safely from one place to another and *Radiance* will still understand them (provided no unintentional character translation takes place along the way)[*]. Most binary files also include a standard ASCII header at their beginning that may be read by the **getinfo** program. This offers a convenient method for identifying the file contents when the file name is ambiguous.

Scene Description Format (.rad suffix)

The semantics of the *Radiance* scene description format are covered in the Reference Manual. We will therefore focus on the file syntax and structure, which are simple and straightforward. In fact, some would say that the *Radiance* scene description format is brain-dead, in the sense that it offers few language amenities and requires the awkward counting of string and real arguments (not to mention those non-existent integer arguments). We have little to offer in its defense.

[*] The single exception to this rule is the Z-buffer file, whose contents are dictated by the floating point representation and byte order of the originating machine. This choice was made for economic reasons, and is rarely a problem.

The truth is, the scene format was designed to grow with *Radiance*, and we wanted to keep it as simple as possible so as to encourage others to write translators to and from it. Specifically, we wanted to be able to read files using the *scanf(3)* library function and write files using *printf(3)*. Furthermore, we wanted everyone's parsers to be stable over time, which meant no primitive-specific syntax. We also decided that a flat file structure was most practical, since hierarchies are typically lost on the first translation, and sufficient structure could be provided by the file system itself. Since we did not intend text editing to be the primary input method, we felt the effects of these programming decisions on the human readability and writability of the format were less important.

Even so, the format is relatively easy to read and write once you get used to it, and with the *Radiance* generator programs and in-line command expansion, the text editor becomes a powerful modeling tool in the hands of an experienced user. Together with the need for editing material descriptions, our assumption that users would rarely edit these files turned out to be mistaken. Consequently, it is a good idea for all users to familiarize themselves with the scene description format, awkward or not.

Basic File Structure

There are four statement types in a scene description file: comments, commands, primitives and aliases. These may be interspersed in the file, and the only structural requirement is that modifiers precede the primitives they modify.

Comments

The simplest statement type is a comment statement begins with a pound sign ('#') and continues to the end of line:

```
# This is a comment.
```

Commands

An in-line command, which begins with an exclamation mark ('!') and continues to the end of line:

```
!xform -n chair1 -t 10 5 8 chair.rad
```

The command is executed during file parsing, and its output is read as more input. Long commands may be continued on multiple lines by escaping the newline character with a backslash ('\'):

```
!gensurf marble sink '15.5+x(theta(s),phi(t))' \
     '10.5+y(theta(s),phi(t))' \
     '30.75+z(theta(s),phi(t))' \
     8 29 -f basin.cal -s
```

Since the command is executed by the shell, pipes and other facilities are available as well. The following command creates a counter with a precisely cut hole for the sink basin just given:

```
!( echo marble polygon sink_top 0 0 108 31 \
     10.5 30.75 31 22 30.75 0 22 30.75 0 0 \
     30.75 31 0 30.75 31 10.5 30.75 ; \
     cnt 30 | rcalc \
          -e '$1=15.5+x(theta(0),phi(1-$1/29))' \
          -e '$2=10.5+y(theta(0),phi(1-$1/29))' \
          -e '$3=30.75' -f basin.cal )
```

Note in the above example that two commands are executed in sequence. The first creates the counter perimeter, and the second cuts the hole. The two commands are enclosed in parentheses, so if a final transformation is added by **xform** with the **-c** option, it will be applied to both commands, not just the last one.

Primitives

A primitive can be thought of as an indivisible unit of scene information. It can be a surface, material, pattern, texture or mixture. The basic structure of a primitive is as follows:

```
modifier type identifier
n  S1  S2  S3  ..Sn
0
m  R1  R2  R3  ..Rm
```

The *modifier* is the *identifier* of a previously defined primitive, or "void" if no modifier is appropriate. The *type* is one of the supported *Radiance* primitive keywords, such as *polygon* or *plastic*. Following the modifier, type and identifier are the string arguments, preceded by the number of string arguments and separated by white space. If there are no string arguments, then 0 should be given for *n*. The string arguments are followed

by the integer arguments in the same fashion. (Since there are no *Radiance* primitives currently using integer arguments, the count is always 0.) Finally, the number of real arguments is given, followed by the real arguments.

The location of the primitive in the scene description has no importance, except that its modifier refers to the most recently defined primitive with that identifier. If no such modifier was defined, an error results. In fact, "undefined modifier" is the most frequently reported error when parsing an invalid scene description, since any random bit of junk encountered where a statement is expected will be interpreted as a modifier. One final note about modifiers -- since surfaces never modify anything, their identifiers are neither stored nor referenced in the parser's modifier list, and serve only for debugging purposes during editing and rendering.

Within a primitive, white space serves only to separate words, and multiple spaces, tabs, form feeds, returns, and newlines are all considered as one separator. Consequently, it is not possible for a string argument to contain any white space, which is OK because no *Radiance* primitive needs this.

Aliases

An alias simply associates a new modifier and identifier with a previously defined primitive. The syntax is as follows:

```
modifier alias new_identifier old_identifier
```

The *old_identifier* should be associated with some modifier primitive (i.e., non-surface) given earlier. The *modifier*, if different from the original, will be used instead in later applications of *new_identifier*.

Aliases are most often used to give new names to previously defined materials. They may also be used to associate different patterns or textures with the same material.

Scene Hierarchy

Hierarchical scene descriptions are achieved through expansion of in-line **xform** commands. The **xform** command is used to read and place other *Radiance* scene description files in the calling file, and these other descriptions may in turn read others, and so on down the tree. No check is

made to assure that none of the calling files is called again, even by itself. If this happens, commands open commands until the system runs out of processes, which is a very nasty business and to be avoided.

Radiance Programs

The following table shows programs in the main *Radiance* distribution that read and write scene description files. Additionally, there are other translators that write scene files, which are available separately as free contributions or as part of other (CAD) programs.

Program	Read	Write	Function
arch2rad		X	Convert Architrion text file to *Radiance*
genblinds		X	Generate curved venetian blinds
genbox		X	Generate parallelepiped
genclock		X	Generate analog clock
genprism		X	Generate extruded polygon
genrev		X	Generate surface of revolution
gensky		X	Generate CIE sky distribution
gensurf		X	Generate arbitrary surface patch
genworm		X	Generate varying diameter curved path
ies2rad		X	Convert IES luminaire file to *Radiance*
mgf2rad		X	Convert MGF file to *Radiance*
mkillum	X	X	Compute *illum* secondary sources
nff2rad		X	Convert NFF file to *Radiance*
objline	X		Generate line drawing of *Radiance* file
objview	X		Quick view of *Radiance* object
oconv	X		Compile *Radiance* scene description
obj2rad		X	Convert Wavefront .OBJ file to *Radiance*
rad	X		Render *Radiance* scene
rad2mgf	X		Convert *Radiance* file to MGF
raddepend	X		Determine scene file dependencies
replmarks	X	X	Replace triangular markers with objects
rpict	X		Batch rendering program
rtrace	X		Customizable ray-tracer
rview	X		Interactive renderer
thf2rad		X	Convert GDS things file to *Radiance*
tmesh2rad		X	Convert triangle mesh file to *Radiance*

xform	X	X	Transform *Radiance* objects

Table 1. *Radiance* programs that read and write scene descriptions.

Radiance C Library

The principal library function for reading scene description files is `readobj(inpspec)`, defined in `src/common/readobj.c`. This routine takes the name of a file, or command beginning with '!', or `NULL` if standard input is to be read, and loads the *Radiance* data structures defined in `src/common/object.h`. If loading *Radiance* data structures is not the action desired, then a more custom approach is necessary, such as that used in `src/gen/xform.c`. If using *Radiance* data structures is acceptable, but the data need not remain resident in memory, then follow the lead in `src/ot/getbbox.c` and use `src/ot/readobj2.c` instead. In any case, the list of defined primitive types in `src/common/otypes.h` is crucial.

Function File Format (.cal suffix)

Function files are used throughout *Radiance* to specify mathematical formulas and relations for procedural textures, patterns and surfaces. They are also used by filter programs such as **rcalc** to manipulate data, and **pcomb** to manipulate pictures.

Function file syntax is simple and should be familiar to most programmers, as it is based on fairly standard algebraic expressions. Here is an example, which corresponds to the in-line commands given in the previous section:

```
{
      basin.cal - calculate coordinates for basin sink.
}

theta(s) = PI*(0.5+0.5*s);
phi(t) = 2*PI*t;

R(th,p) = 5 + ( 3.25*cos(p)^2 +
                  1.75*sin(p)^2 ) * sin(th)^2;

x(th,p) = R(th,p)*sin(th)*cos(p);
y(th,p) = R(th,p)*sin(th)*sin(p);
z(th,p) = R(th,p)*cos(th);
```

In contrast to the usual semantics in programs where each statement corresponds to an evaluation, statements in function files correspond to

definitions. Once a function or variable has been defined, it may be used in later definitions, along with predefined functions such as `sin(x)` and `cos(x)` and constants such as `PI` (1). (All math functions use standard C conventions, hence trigonometry is done in radians rather than degrees.)

Evaluation order (operator precedence) follows standard rules of algebra. Exponentiation is performed first (`x^y`), followed by multiplication and division (`x*y`, `x/y`), then addition and subtraction (`x+y`, `x-y`). Unary minus is most tightly associated (-x), and parentheses override operator precedence in the usual way. Semicolons separate statements, and white space is generally ignored. Comments are enclosed by curly braces, which may be nested.

The above file does not actually *do* anything, it merely defines functions that are useful by a program that *does*. Taking our **gensurf** example from the previous section:

```
!gensurf marble sink '15.5+x(theta(s),phi(t))' \
     '10.5+y(theta(s),phi(t))' \
     '30.75+z(theta(s),phi(t))' \
     8 29 -f basin.cal -s
```

The **-f** option loads in our file, which is then used to evaluate expressions such as `'15.5+x(theta(s),phi(t))'` for specific values of `s` and `t`. These variables range from 0 to 1 over the surface patch in increments of 1/8 and 1/29, respectively. (See the **gensurf** manual page for details.) The entire expression for each evaluation could have been written in the command line, but it is much more convenient to create a function file.

Language Features

Subtle features of the functional language provide much greater power than first meets the eye. One of these is the ability to define recursive functions. The following example defines the factorial function (*n!*):

```
fact(n) : if(n-1.5, n*fact(n-1), 1);
```

This uses the library function `if(cond,e1,e0)`, which returns `e1` if `cond` is greater than zero, and `e0` otherwise. Since only one of these expressions is evaluated, `fact(n)` will call itself until `n` is less than 2, when

the `if` expression returns 1*. The colon (':') is used in place of the usual equals assignment ('=') because we want this function to have the *constant attribute*, which means any later appearance in an expression of `fact(`*`ce`*`)` where *ce* is also a constant expression will be replaced by its value. This can be an important savings in cases where an expression or subexpression is expensive to evaluate, and only needs to be computed once. All of the standard library functions have the constant attribute. (See the following section for a complete list.)

Another handy language feature is the ability to pass function names as arguments. A simple example of this is the following function, which computes the numerical derivative of a function given as its first argument:

```
FTINY : 1e-7;
d1(f,x) = (f(x+FTINY)-f(x-FTINY))/FTINY/2;
```

Evaluating `d1(sin,1.1)` using this formula yields 0.4536, which is fairly close to the true derivative, which is `cos(1.1)`.

A third language feature, which is normally transparent to the user, is the notion of *contexts*. Identifiers may be composed of parts, starting with a name and continuing with one or more context names. Each name is delimited by a back-quote ('`'). Names themselves begin with a letter and continue with any sequence of letters, digits, underscores and decimal points. The following are examples of valid identifiers:

```
v1, V.W, x_rand`local, `A_, Qa_5`
```

If a context mark appears at the beginning of the identifier, then its reference must be local. If it appears at the end, then its reference must be global. A local reference must be resolved in the local context, i.e., no contexts above this one will be searched. A global reference must correspond to the original context, ignoring any local redefinitions.

The reason that contexts are normally transparent is that they are controlled by the calling program -- there are no explicit language features for establishing contexts. A new context is established automatically for each function file loaded by the rendering programs. That way, it is safe to reuse

* Note that we compare `n` to 1.5, so as to avoid any round-off problems caused by floating point math. Caution is advised because all expressions are evaluated as double-precision real, and comparisons to zero are unreliable.

variable names that have been used in other files, and even in the main initialization file, `rayinit.cal`.

Although not strictly necessary, there are two good reasons to define variables and functions before referencing them in a function file. One is related to contexts. If a previous definition of a variable name is given in an enclosing context (e.g., `rayinit.cal`), then that reference will be used rather than a later one in the current context, unless the reference is made explicitly local by starting the identifier with a context mark. The second reason for defining before referencing is constant expressions. If a variable or function has the constant attribute (i.e., defined with ':' instead of '='), then a later subexpression referencing it can be replaced by its evaluated result during compilation. If the constant is not defined until after it is referenced, it remains as a reference, which takes time to evaluate each time.

Other features of the language are truly transparent, but knowledge of them can help one to write more efficient function files:

- Once a variable has been evaluated, the result is cached and it is not reevaluated unless the client program changes an internal counter (`eclock`), which indicates that something has changed. This means that using variables to hold frequently used values will not only simplify the function file, it will save time during evaluation.
- An argument passed in a function call is not evaluated until the function calls for it specifically, and the result is also cached to avoid redundant calculation. The conditional evaluation feature is actually a requirement for recursive functions to work, but caching is not. Argument value caching means it is more efficient to pass an expensive-to-compute expression than to have the function compute it internally if it appears more than once in the function definition. This is especially true for recursive functions with deep call trees.

Standard Definitions (Library)

The following are always defined:

`if(a, b, c)`
: Conditional expression. If `a` is positive, return `b`, else return `c`.

select(N, a1, a2, ..)

Return Nth argument. If N is 0, then return the count of arguments excluding the first. This provides basic array functionality.

sqrt(x)

Return square root of x, where x >= 0.

sin(x), cos(x), tan(x), asin(x), acos(x), atan(x), atan2(y,x)

Standard trigonometry functions.

floor(x), ceil(x)

Greatest lower bound and least upper bound (integer).

exp(x), log(x), log10(x)

Exponent and logarithm functions.

rand(x)

Return pseudo-random number in the range [0,1) for any argument x. The same return value is guaranteed for the same argument.

The following are sometimes defined, depending on the program:

PI

The ratio of a circle's circumference to its diameter.

erf(z), erfc(z)

Error function and complimentary error function.

j0(x), j1(x), jn(n,x), y0(x), y1(x), yn(n,x)

Bessel functions.

hermite(p0,p1,r0,r1,t)

One-dimensional Hermite polynomial.

The rendering programs also define the following noise functions:

noise3(x,y,z), noise3x(x,y,z), noise3y(x,y,z), noise3z(x,y,z)

Perlin noise function and its gradient [Perlin85][Arvo91,p.396].

fnoise3(x,y,z)

Fractal noise function, ranging from -1 to 1.

Interaction with the renderer is achieved via special purpose variables and functions whose values correspond to the current ray intersection and the

calling primitive. Unlike the above functions, none of these have the constant attribute since their values change from one ray to the next:

Dx, Dy, Dz
: ray direction

Nx, Ny, Nz
: surface normal

Px, Py, Pz
: intersection point

T
: distance from start

Ts
: single ray (shadow) distance

Rdot
: ray dot product

S
: world scale

Tx, Ty, Tz
: world origin

Ix, Iy, Iz
: world i unit vector

Jx, Jy, Jz
: world j unit vector

Kx, Ky, Kz
: world k unit vector

arg(n)
: real arguments, `arg(0)` is count

For BRDF primitives, the following variables are also available:

NxP, NyP, NzP
: perturbed surface normal

RdotP
: perturbed ray dot product

CrP, CgP, CbP
: perturbed material color

For prism1 and prism2 primitives, the following are available:

DxA, DyA, DzA
direction to target light source

Other functions, variables and constants are defined as well in the file `src/rt/rayinit.cal`, which gets installed in the standard *Radiance* library directory and can be modified or appended as desired[*].

Radiance Programs

Table 2 shows *Radiance* programs that read and write function files.

Program	Read	Write	Function
calc	X	X	Interactive calculator
genrev	X		Generate surface of revolution
gensurf	X		Generate arbitrary surface patch
genworm	X		Generate varying diameter curved path
macbethcal		X	Compute image color & contrast correction
pcomb	X		Perform arbitrary math on picture(s)
rcalc	X		Record stream calculator
rpict	X		Batch rendering program
rtrace	X		Customizable ray-tracer
rview	X		Interactive renderer
tabfunc		X	Create function file from tabulated data

Table 2. Programs in the *Radiance* distribution that read and write function files.

In addition, the program **ev** evaluates expressions given as command line arguments, though it does not handle function files or definitions. There are also a number of 2-d plotting routines that use a slightly modified statement syntax, called **bgraph**, **dgraph**, **gcomp**, and **igraph**. Additional utility programs are useful in combination with **rcalc** for data analysis and scene generation. The program **cnt** generates simple records to drive **rcalc**, and the **total** program is handy for adding up results. The **histo** program computes histograms needed for certain types of statistical analysis. The **lam** program concatenates columns from multiple input files, and **neat** neatens up columns for better display.

[*] It is usually a good idea to store any such customized files in a personal library location and set the RAYPATH environment variable to search there first. This way, it does not affect other users or get overwritten during the next system installation.

Radiance C Library

The standard routines for loading and evaluating function files are divided into three modules, `src/common/calexpr.c` for expression parsing and evaluation, `src/common/caldefn.c` for variable and function storage and lookup, and `src/common/calfunc.c` for library function storage and function evaluation. There is a fourth module for writing out expressions called `src/common/calprnt.c`, which we will not discuss. They all use the header file `src/common/calcomp.h`, which defines common data structures and evaluation macros. Of these, the three most often used declarations for external routines are:

`typedef struct epnode EPNODE;`

Expression parse tree node. Some routines return pointers to this structure type, and the main evaluation macro, `evalue(ep)`, takes an argument of this type.

`(double) evalue(ep);`

Evaluate an expression parse tree. Uses node type table to access appropriate function depending on root node type. (E.g., an addition node calls `eadd(ep)`.)

`extern unsigned long eclock;`

This is a counter used to determine when variable values need updating. The initial value is 0, which tells the routines always to reevaluate variables. Once incremented to 1, variable evaluations are cached and not recomputed until `eclock` is incremented again. Usually, client programs increment `eclock` each time definitions or internal states affecting returned values change. This assures the quickest evaluation of correct results.

The usual approach to handling definitions is to compile them into the central lookup table; variable and function references are later evaluated by traversing the stored parse trees. Syntax errors and undefined symbol errors during evaluation result in calls to the user-definable routine `eputs(s)` to report the error and `quit(status)` to exit the program. Domain and range errors during evaluation set `errno`, then call the user-definable routine `wputs(s)` to report the error and return zero as the expression result.

Following are standard routines provided for parsing from a file and parsing from a string:

`EPNODE *eparse(char *expr);`

Compile the string `expr` into a parse tree for later evaluation with `evalue(ep)`.

`epfree(EPNODE *ep);`

Free memory associated with `ep`, including any variables referenced if they are no longer defined.

`double eval(char *expr);`

Immediately parse, evaluate and free the given string expression.

`fcompile(char *fname);`

Compile definitions from the given file, or standard input if `fname` is `NULL`.

`scompile(char *str, char *fn, int ln);`

Compile definitions from the string `str`, taken from file `fn` at line number `ln`. If no file is associated, `fn` can be `NULL`, and `ln` can be 0.

The following routines allow one to control the current context for definition lookup and storage:

`char *setcontext(char *ctx);`

Set the current context to `ctx`. If `ctx` is `NULL`, then simply return the current context. An empty string sets the global context.

`char *pushcontext(char *name);`

Push another context onto the stack. Return the new (full) context.

`char *popcontext();`

Pop the top context name off the stack. Return the new (full) context.

The following functions permit the explicit setting of variable and function values:

`varset(char *vname, int assign, double val);`

Set the specified variable to the given value, using a constant assignment if `assign` is ':' or a regular one if it is '='. This is always faster than compiling a string to do the same thing.

`funset(char *fname, int nargs, int assign, double (*fptr)(char *fn));`

Assign the specified library function, which takes a minimum of `nargs` arguments. The function will have the constant attribute if `assign` is ':', or not if it is '='. The only argument to the referenced function pointer is the function name, which will equal `fname`. (This string must therefore be declared **static**.) This offers a convenient method to identify calls to an identical function assigned multiple tasks. Argument values are obtained with calls back to the `argument(n)` library function.

The following functions are provided for evaluating a function or variable in the current definition set:

`double varvalue(char *vname);`

Evaluate the given variable and return the result. Since a hash lookup is necessary to resolve the reference, this is slightly less efficient than evaluating a compiled expression via `evalue(ep)`, which uses soft links generated and maintained during compilation.

`double funvalue(char *fn, int n, double a);`

Evaluate the function `fn`, passing `n` real arguments in the array `a`. There is currently no mechanism for passing functions or function name arguments from client programs.

These routines can be used to check the existence of a specific function or variable:

`int vardefined(char *vname);`

Return non-zero if the named variable is defined. (If the symbol is defined as a function, zero is returned.)

`int fundefined(char *fname);`

Return the number of required arguments for the named function if it is defined, or zero if it is not defined. (If the symbol is defined as a variable, zero is returned.)

These routines allow definitions to be cleared:

`dclear(char *dname);`

Clear the given variable or function, unless it is a constant expression.

dremove(char *dname);

Clear the given variable or function, even if it is a constant expression. Library definitions cannot be removed, except by calling `funset` with a `NULL` pointer for the function argument.

dcleanup(int level);

Clear definitions. If level is 0, then just clear variable definitions. If level is 2, then clear constants as well. If the current context is local, then only local definitions will be affected. If global, all definitions in all contexts will be affected.

These routines may be used during library function evaluation:

int nargum();

Determine the number of arguments available in the current function evaluation context.

double argument(int n);

Evaluate and return the `n`th argument.

char *argfun(n);

Get the name of the function passed as argument `n`. (Generates an error if the `n`th argument is not a function.)

Other, even more specialized routines are provided for controlling the parsing process, printing out expressions and sifting through stored definitions, but these are not accessed by most client programs. Worth noting are the various compile flags that affect which features of the expression language are included. The standard library sets the flags `-DVARIABLE -DFUNCTION -DRCONST` and `-DBIGLIB`. Here is a list of compile flags and their meanings:

-DVARIABLE

Allow user-defined variables and (if `-DFUNCTION`) user-defined functions.

-DFUNCTION

Compile in library functions and (if `-DVARIABLE`) allow user-supplied function definitions.

-DBIGLIB

Include larger library of standard functions, i.e., standard C math library. Otherwise, only minimal library is compiled in, and other functions may be added using `funset`.

-DRCONST

Reduce constant subexpressions during compilation. This can result in substantial savings during later evaluation, but the original user-supplied expressions are lost.

-DREDEFW

Issue a warning via `wputs(s)` if a new definition hides a constant definition or library function, or replaces an existing, distinct definition for the same symbol. (The `varset` routine never generates warnings, however.)

-DINCHAN

Provide for "$*N*" syntax for input channels, which result in call-backs to client-supplied `chanvalue(n)` routine on each evaluation.

-DOUTCHAN

Provide for "$*N*" lvalue syntax for output channels, which are evaluated via the `chanout(cs)` library function, which calls `(*cs)(n, value)` for each assigned channel definition.

Data File Format (.dat suffix)

Although it is possible to store tabular data in a function file using the `select` library function, it is more convenient and efficient to devote a special file format to this purpose. *Radiance* data files store scalar values on an N-dimensional rectangular grid. Grid (independent) axes may be regularly or irregularly divided, as shown in Figure 1. This data is interpolated during rendering (using N-dimensional linear interpolation) to compute the desired values.

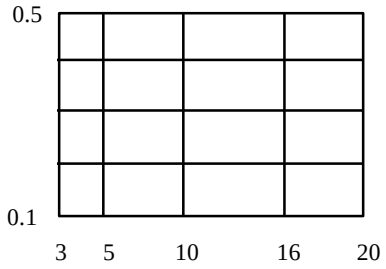

Figure 1. A 2-dimensional grid with one regularly divided axis and one irregularly divided axis. Each intersection corresponds to a data value that appears in the file.

Data files are broken into two sections, the header and the body. The header specifies the grid, and the body contains the data values in a standard order. The first value in the file is a positive integer indicating the number of dimensions. Next comes that number of axis specifications, in one of two formats. For a regularly divided axis, the starting and ending value is given, followed by the number of divisions. For an irregularly divided axis, two zeros are followed by the number of divisions then that number of division values. The two zeros are merely there to indicate an irregular spacing is being specified. Once all the axes have been given, the header is done and the body of the file begins, which consists of one data value after another. The ordering of the data is such that the last axis given is the one being traversed most rapidly, similar to a static array assignment in C.

A file corresponding to the topology shown in Figure 1 is:

```
######### Header ########
2                   # Two-dimensional data array
0.5 0.1 5           # The regularly spaced axis
0 0 5 3 5 10 16 20  # The irregularly spaced axis
########## Body #########
# The data values, starting with the
# upper left, moving right then down:
 19.089   7.001  14.647   6.3671  8.0003
  3.8388 11.873  19.294  16.605   2.7435
 16.699   6.387   2.8123 16.195  17.615
 14.36   14.413  16.184  15.635   4.5403
  3.6740 14.550  10.332  15.932   1.2678
```

Comments begin with a pound sign ('#') and continue to the end of the line. White space is ignored except as a data separator, thus the position of header and data values on each line is irrelevant except to improve readability.

Radiance Programs

Table 3 shows *Radiance* programs that read and write data files.

Program	Read	Write	Function
ies2rad		X	Convert IES luminaire file to *Radiance*
mgf2rad		X	Convert MGF file to *Radiance*
rpict	X		Batch rendering program
rtrace	X		Customizable ray-tracer
rview	X		Interactive renderer

Table 3. Programs in the *Radiance* distribution that read and write data files.

Radiance C Library

The header file `src/rt/data.h` gives the standard data structures used by the routines in `src/rt/data.c` for reading and interpolating data files. The main data type is `DATARRAY`, which is a structure containing the grid specification and a pointer to the data array, which is of the type `DATATYPE` (normally **float** to save space).

The main routine for reading data files is `getdata(dname)`, which searches the standard *Radiance* library locations set by the `RAYPATH` environment variable. The return value is a pointer to the loaded `DATARRAY`, which may have been loaded by a previous call. (The routine keeps a hash table of loaded files to minimize time and memory requirements.) The `freedata(dname)` routine frees memory associated with the named data file, or all data arrays if `dname` is `NULL`.

The routine that interpolates data values is `datavalue(dp,pt)`, which takes a `DATARRAY` pointer and an array of **double**s of the appropriate length (the number of dimensions in `dp`). The **double** returned is the interpolated value at that point in the scalar field. If the requested point lies outside the data's grid, it is extrapolated from the perimeter values up to a distance of one division in each dimension, and falls off harmonically to zero outside of that. This was selected as the most robust compromise, but to be safe it is generally best to avoid giving out-of-domain points to `datavalue`.

Font File Format (.fnt suffix)

Font files are used for text patterns and mixtures, and by the **psign** program to generate simple text labels. Each character glyph is set up on a simple rectangular coordinate system running from [0,255] in x and y, and the glyph itself is a polygon. Figure 2 shows an example of the letter "A".

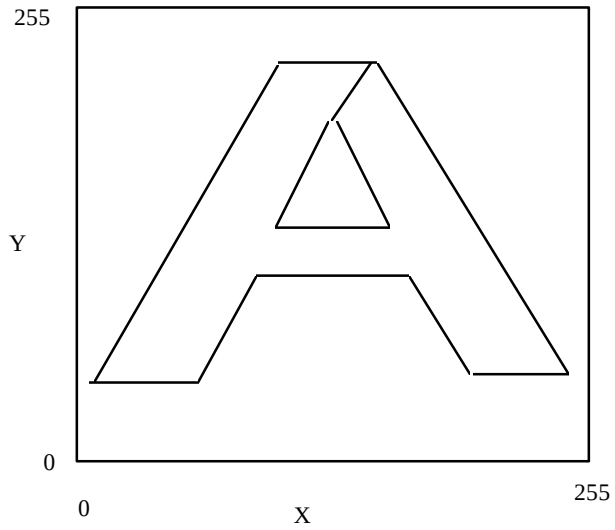

Figure 2. A glyph for an "A" character in standard font coordinates. Note that the hole is made via a seam, just as with *Radiance* scene polygons. The actual aspect and spacing of the character will be determined by the client program.

Each glyph begins with the decimal value of that character's index, which is 65 for "A" according to the ASCII standard. This is followed by the number of vertices, then the vertices themselves in x1 y1 x2 y2 order. White space again serves as a separator, and comments may begin with a pound sign ('#') and continue to the end of line. Here is the glyph entry for the letter "A" corresponding to Figure 2:

```
65   15          # Helvetica "A"
     155  222  242  48   185  48   168  86
     83   86   65   48   12   48   101  222
     155  222  128  179  126  179  97   116
     155  116  128  179  155  222
```

If the number of vertices given is zero, then the character is a space. This is not the same as no entry, which means there is no valid glyph for that

character. Glyphs may appear in any order, with indices ranging from 0 to 255. The maximum number of vertices for a single glyph is 32767.

Two standard font files are provided, "helvet.fnt" and "hexbit4x1.fnt". The former is a Helvetica font from the public domain Hershey font set. The second is a simple bit pattern font for hexadecimal encodings of bitmaps.

Radiance Programs

Table 4 shows *Radiance* programs that read and write font files.

Program	Read	Write	Function
pcompos	X		Compose *Radiance* pictures
psign	X		Generate *Radiance* picture label
rpict	X		Batch rendering program
rtrace	X		Customizable ray-tracer
rview	X		Interactive renderer

Table 4. Programs in the *Radiance* distribution that read and write font files.

Radiance C Library

Similar to data files, font files are usually read and stored in a lookup table. The data structures for fonts are in `src/common/font.h`, and the routines for reading and spacing them are in `src/common/font.c`. The main structure type is `FONT`. The routine `getfont(fname)` loads a font file from the *Radiance* library (set by the `RAYPATH` environment variable), and returns a pointer to the resulting `FONT` structure. If the file has been previously loaded, a pointer to the stored structure is returned. The `freefont(fname)` routine frees memory associated with the named font file and deletes it from the table, or frees all font data if `fname` is `NULL`.

Three different routines are available for text spacing. The `uniftext(sp,tp,f)` function takes the nul-terminated string `tp` and computes uniform per-character spacing for the font `f`, returned in the short integer array `sp`. (This is a fairly simple process, and all spacing values will be 255 unless a character has no corresponding glyph.) The `squeeztext(sp,tp,f,cis)` performs a similar function, but puts only `cis` units between adjacent characters, based on the actual width of each font glyph. The most sophisticated spacing function is `proptext(sp,tp,f,cis,nsi)`, which produces a total line length

equal to what it would be with uniform spacing, while maintaining equal inter-character spacing throughout (i.e., proportional spacing). The `nsi` argument is the number of spaces (zero-vertex glyphs) considered as an indent. That is, if this many or more adjacent spaces occur in `tp`, the indented text following will appear at the same point as it would have had the spacing been uniform. This maintains columns in tabulated text despite the proportional spacing. Tabs are not understood or interpreted by any of these routines, and must be expanded to the appropriate number of spaces via **expand**.

Octree Format (.oct suffix)

In *Radiance*, octrees are used to accelerate ray intersection calculations as described by Glassner [Glassner84]. This data structure is computed by the **oconv** program, which produces a binary file as its output. An octree file contains a list of *Radiance* scene description files (which may be empty), some information to guarantee portability between systems and different versions of the code, followed by the octree data itself. If the octree file is "frozen," then it will also contain the scene data, compiled into a binary format for quick loading. This is most convenient for octrees that are used in *instance* primitives, which may be moved to a different (library) location from the originating scene files.

An octree recursively subdivides 3-dimensional space into 8 subtrees, hence its name. Each "leaf" node contains between zero and `MAXSET` surface primitives, indicating that section of space contains part or all of those surfaces. (Surface primitives may appear more than once in the octree.) The goal of **oconv** is to build an octree that contains no more than N surfaces in each leaf node, where N is set by the **-n** option (5 by default). It may allow more surfaces in places where the octree has reached its maximum resolution (depth), set by the **-r** option (1024 -- depth 10 by default). Figure 3 shows a quadtree dividing 2-dimensional space, analogous to our 3-dimensional octree.

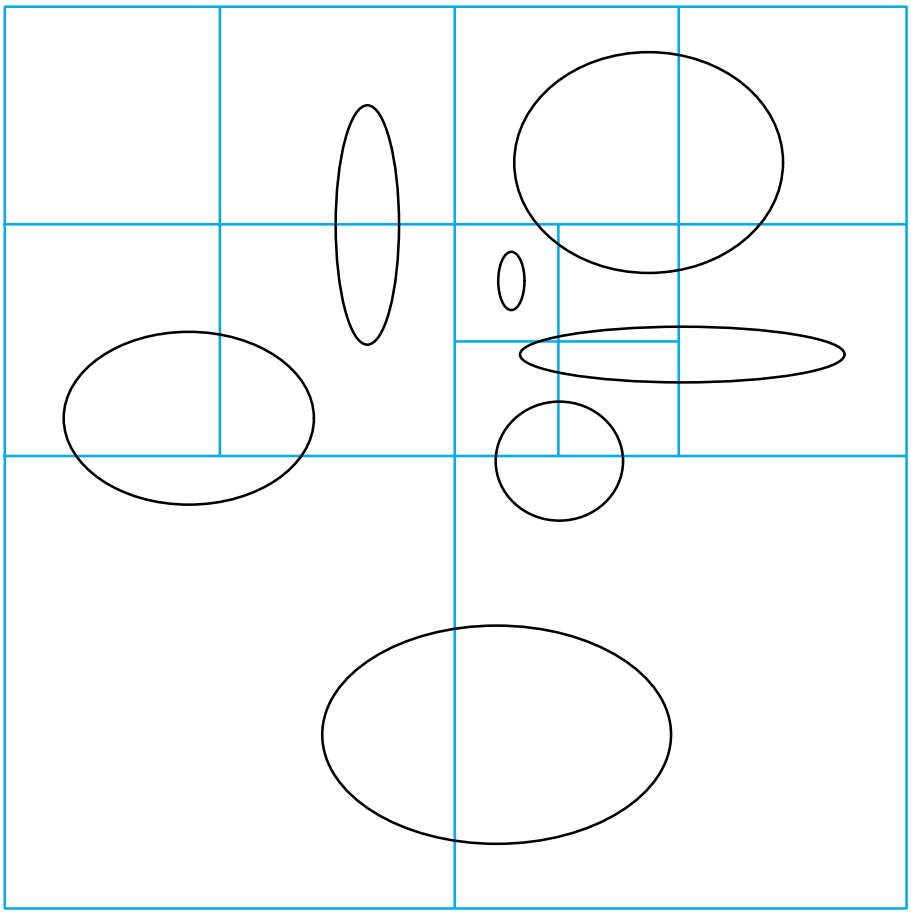

Figure 3. An example quadtree divided so that no leaf node contains more than 2 objects. A three-dimensional octree works the same way. Each leaf node is either empty, or contains a list of intersecting surfaces.

Basic File Structure

An octree file is divided into five sections: the information header, the scene boundaries, the scene file names, the octree data structure, and the compiled scene data. If the octree is frozen, then the compiled scene data is included and the scene file names are not. Otherwise, the scene data is left off.

Information Header

As with other binary *Radiance* formats, the beginning of an octree file is the information header. The first line is "`#?RADIANCE`" to aid in identification by the UNIX **file** program. Following this is the **oconv** command (or commands) used to produce the octree, then a line indicating

the format, "FORMAT=Radiance_octree". The end of the information header is always an empty line. Here is an example of an octree information header, as reported by **getinfo**:

```
#?RADIANCE
oconv model.b90 desk misc
oconv -f -i modelb.oct window blinds lights lamp
FORMAT=Radiance_octree
```

The actual content of this header is ignored when an octree is read except for the FORMAT line, which if it appears must match the one shown above.

Scene Boundaries

After the information header, there is a magic number indicating the format version and the size of object indices (in bytes per index). This is a two-byte quantity, which must be one of the following in the current release:

285 Two-byte object indices.

287 Four-byte object indices.

291 Eight-byte object indices. (Only supported on architectures with 64-bit **long**s.)

Technically, the code will also support odd-sized integers, but they are not found on any existing machine architectures so we can forget about them.

Following the octree magic number, we have the enclosing cube for the scene, which defines the dimensions of the octree's root node. The cube is aligned along the world coordinate axes, so may be defined by one corner (the 3-dimensional minimum) and the side length. For historical reasons, these four values are stored as ASCII-encoded real values in nul-terminated strings. (The octree boundaries may also be read using **getinfo** with the **-d** option.)

Scene File Names

Following the octree dimensions, the names of the scene description files are given, each stored a nul-terminated string. The end of this file list is indicated by an empty string. If the octree is "frozen," meaning it contains the compiled scene information as well, then no file names will be present (i.e., the first string will be empty).

Octree Data Structure

After the scene description files, an N-byte integer indicates the total number of primitives given to **oconv**, where N is the size derived from the magic number as we described. This object count will be used to verify that the files have not changed significantly since the octree was written[*].

After the primitive count, the actual octree is stored, using the following recursive procedure:

```
puttree(ot) begin
     if ot is a tree then
          write the character '\002'
          call puttree on each child node (0-7)
     else if ot is empty then
          write the character '\000'
     else
          write the character '\001'
          write out the number of surfaces
          write out each surface's index
     end
end puttree
```

The number of surfaces and the surface indices are each N-byte integers, and the tree node types are single bytes. Reading the octree is accomplished with a complementary procedure.

Compiled Scene Data

If the octree is frozen, then this data structure is followed by a compiled version of the scene. This avoids the problems of changes to scene files, and allows an octree to be moved conveniently from one location and one system to another without worrying about the associated scene files.

The scene data begins with a listing of the defined primitive types. This list consists of the name of each type as a nul-terminated string, followed by an empty string once the list has been exhausted. This permits the indexing of primitive types with a single byte later on, without concern about changes to *Radiance* involving `src/common/otypes.h`.

[*] Small changes that do not affect geometry will not cause problems, but if the primitive count changes, so does the indexing of surfaces, and with that the octree data structure becomes invalid. A second check is made to insure that no non-surface primitives appear in any leaf nodes, and this at least guarantees that the renderer will not dump core from an outdated octree, even if the results are wrong.

The scene primitives are written one at a time. First is a single byte with the primitive type index, as determined from the list given above. Second is the N-byte modifier index, followed by the primitive's identifier as a nul-terminated string. String arguments start with a 2-byte integer indicating the argument count, followed by the strings themselves, which are nul-terminated. Real arguments next have a 2-byte count followed by the real values, each stored as a 4-byte mantissa followed by a 1-byte (signed) exponent. (The mantissa is the numerator of a fraction of 2^31-1.) The end of data is indicated with a -1 value for the object type (byte=255).

Radiance Programs

Table 5 shows *Radiance* programs that read and write octree files.

Program	Read	Write	Function
getinfo	X		Print information header from binary file
oconv	X	X	Compile *Radiance* scene description
rad	X	X	Render *Radiance* scene
rpict	X		Batch rendering program
rpiece	X		Parallel batch rendering program
rtrace	X		Customizable ray-tracer
rview	X		Interactive renderer

Table 5. Programs in the *Radiance* distribution that read and write octree files.

Radiance C Library

Since reading an octree file also may involve reading a *Radiance* scene description, some of the same library routines are called indirectly. The header file `src/common/octree.h` is needed in addition to the `src/common/object.h` file. The module `src/ot/writeoct.c` contains the main routines for writing an octree to `stdout`, while `src/common/readoct.c` contains the corresponding routines for reading an octree from a file. Both modules access routines in `src/common/portio.c` for reading and writing portable binary data.

Here is the main call for writing out an octree:

`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.

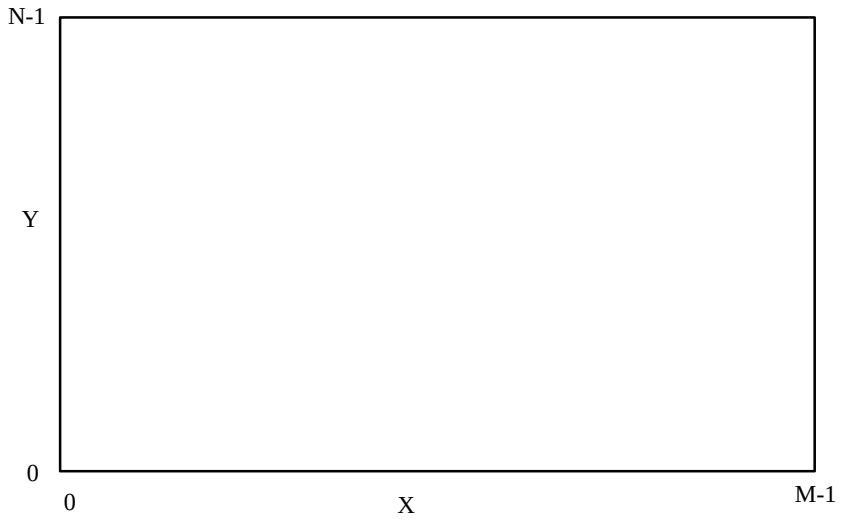

Figure 4. The standard coordinate system for an MxN picture.

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.

Z-buffer Format (.zbf suffix)

The Z-buffer format used in *Radiance* hardly qualifies as a format at all. It is in fact a straight copy on the 4-byte machine floating point values of each pixel in standard scanline order. There is no information header or resolution string that would make the file independently useful. This is usually OK, because Z-buffer files are almost always created and used in conjunction with a picture file, which has this identifying information.

The major shortcoming of this format is that the machine representation and byte ordering is not always the same from one system to another, which limits the portability of Z-buffer files. Since they are primarily used for interpolating animation frames, and this usually occurs on networks with similar architectures, there is usually no problem. Also, since the adoption of IEEE standard floating-point calculations, different machine representations are becoming quite rare.

Radiance programs

Table 7 shows the programs that read and write *Radiance* Z-buffer files. The **pvalue** program may be used to convert Z-buffer files to *Radiance* pictures for the purpose of visualizing the values using **falsecolor**. For example, the following command converts the Z-buffer file "`frame110.zbf`" associated with the picture "`frame110.pic`" to a viewable image:

```
% pvalue -h `getinfo -d < frame110.pic` -r -b -df
frame110.zbf | falsecolor -m 1 -s 40 -l Meters >
frame110z.pic
```

The **getinfo** program appearing in back-quotes was used to get the dimensions associated with the Z-buffer from its corresponding picture file.

Program	Read	Write	Function
pinterp	X	X	Interpolate/extrapolate picture views
pvalue	X	X	Convert picture to/from simpler formats
rad		X	Render *Radiance* scene
ranimate	X	X	Animate *Radiance* scene
rpict		X	Batch rendering program
rtrace		X	Customizable ray-tracer

Table 7. *Radiance* programs that read and write Z-buffer files.

Radiance C Library

There are no library functions devoted to reading and writing Z-buffer files in particular. The normal method is to read and write Z-buffer scanlines with the standard `fread` and `fwrite` library functions using an appropriate **float** array.

Ambient File Format (.amb suffix)

Radiance can store its diffuse interreflection cache in an *ambient file* for reuse by other processes. This file is in a system-independent binary format, similar to an octree or picture file, with an information header that can be read using **getinfo**. Following the header, there is a magic number specific to this file type, then the ambient value records themselves in encoded form.

Information Header

The information header begins with the usual "`#?RADIANCE`" identifier, followed by the originating program and the ambient calculation parameters (and octree name). After this is the line:

```
FORMAT=Radiance_ambval
```

This identifies the general file type, followed by an empty line ending the header. As with most information headers, this exact sequence need not be followed, so long as there is no inconsistent `FORMAT` setting.

Magic Number

Following the information header is the two-byte magic number, which for the current ambient file format is 557. This number may change later should the file format be altered in incompatible ways.

Ambient Value Records

Ambient values are written to the file in no particular order. Each diffuse interreflection value in *Radiance* has the following members:

Level
: The number of reflections between the primary (eye) ray and this surface. A value with fewer reflections may be used in place of one with more, but not the other way around.

Weight
: The weighting value associated with this ray or ambient value. Similar to the level to avoid using inappropriate values from the cache.

Position
: The origin point of this interreflection calculation.

Direction
: The surface normal indicating the zenith of the sample hemisphere for this value.

Value
: The calculated indirect irradiance at this point, in watts/sq.meter (RGB color).

Radius
: The cosine-weighted, harmonic mean distance to other surfaces visible from this point, used to decide point spacing.

Posgradient
: The position-based gradient vector, indicating how brightness changes as a function of position in the sample plane.

Dirgradient
: The direction-based gradient vector, indicating how brightness changes as a function of surface orientation.

The members are stored one after the other in the above order using system-independent binary representations. The Level member takes 1 byte, Weight takes 5, Position takes 15, Direction another 15, Value is 4 bytes (using the same color format as *Radiance* pictures), Radius takes 5 bytes, and Posgradient and Dirgradient each take 15 bytes, for a total size of 75 bytes per record.

Radiance Programs

Table 8 shows *Radiance* programs that read and write ambient files. The program **lookamb** is especially useful for examining the contents of ambient files and debugging problems in the calculation.

Program	Read	Write	Function
getinfo	X		Print information header from binary file
lookamb	X	X	Convert *Radiance* ambient file
rad	X	X	Render *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

Table 8. Programs in the *Radiance* distribution that read and write ambient files.

Radiance C Library

The `src/rt/ambient.h` file contains definitions of the `AMBVAL` structure and certain details of the ambient file format. The `src/rt/ambio.c` module contains the specialized routines for reading and writing ambient files, and these functions in turn access routines in `src/common/portio.c` for reading and writing portable binary data. The information header is handled by the routines in `src/common/header.c`, similar to the method described for *Radiance* picture files. Here are the main calls from `src/rt/ambio.c`:

`putambmagic(FILE *fp);`

Put out the appropriate two-byte magic number for a *Radiance* ambient file to the stream `fp`.

`int hasambmagic(FILE *fp);`

Read the next two bytes from the stream `fp` and return non-zero if they match an ambient file's magic number.

`int writeambval(AMBVAL *av, FILE *fp);`

Write out the ambient value structure `av` to the stream `fp`, returning -1 if a file error occurred, or 0 normally.

`int readambval(AMBVAL *av, FILE *fp);`

Read in the next ambient value structure from the stream `fp` and put the result in `av`. Return 1 if the read was successful, 0 if the end of file was reached or there was an error. The `ambvalOK` function is used to check the consistency of the value read.

`int ambvalOK(AMBVAL *av);`

Return non-zero if the member values of the `av` structure are not too outlandish. This is handy as insurance against a corrupted ambient file.

Conclusion

We have described the main file formats native to *Radiance* and shown how even the binary formats can be reliably shared in heterogeneous computing environments. This corresponds to one of the basic philosophies of UNIX software, which is system independence. A more detailed understanding of the formats may still require some use of binary dump programs and delving into the *Radiance* source code.