Radiance Source Tree Roadmap

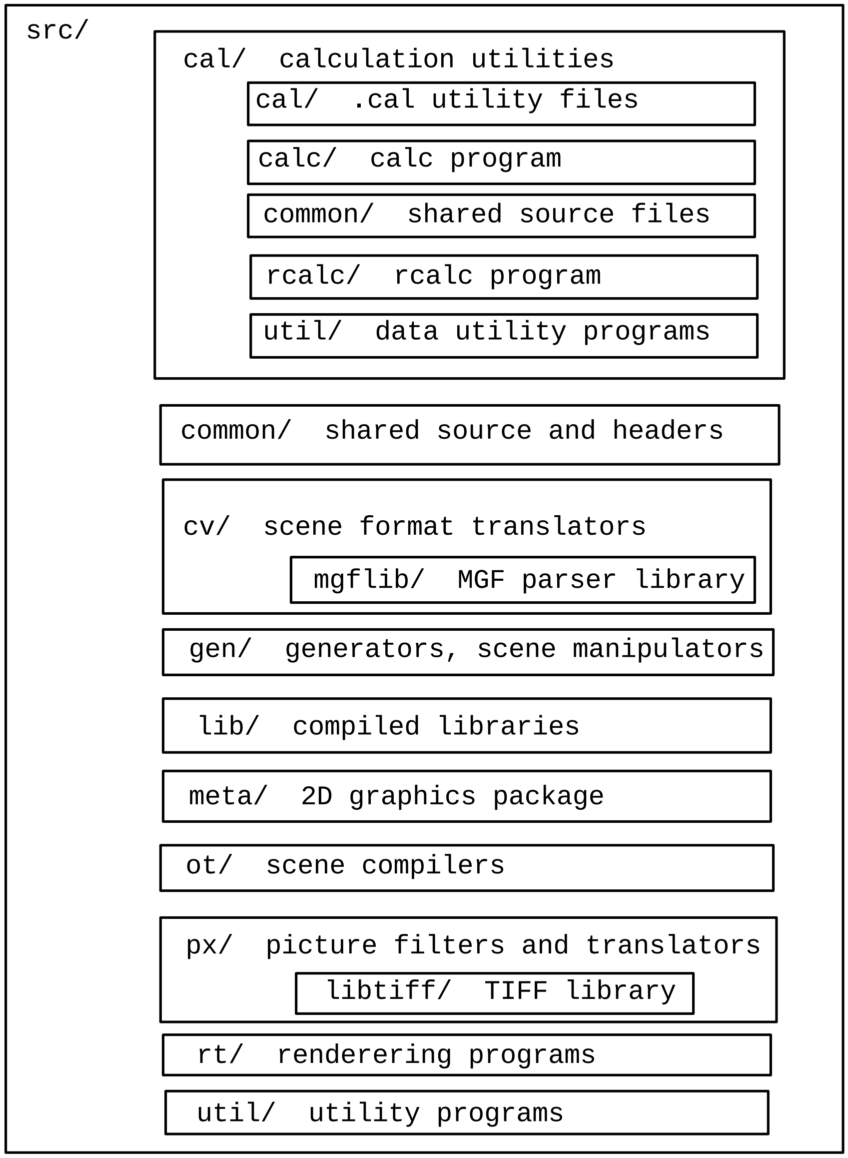

Figure 1. *Radiance* source directory tree.

The *Radiance* source tree is divided into six main subdirectories corresponding to the principal program categories, plus a `src/common/` subdirectory for shared header and library modules. Two other subdirectories, `src/cal/` and `src/meta/`, build programs that were not initially part of the standard distribution. Additional subdirectories of the main subdirectories contain auxiliary libraries that we will describe later.

Let us start first with a list of the programs built in each of the six main source subdirectories, shown in Table 1.

Directory	Category	Programs
`src/cv/`	Scene Format Translators	**arch2rad ies2rad lampcolor mgf2inv mgf2meta mgf2rad mgfilt nff2rad obj2rad rad2mgf thf2rad tmesh2rad**
`src/gen/`	Generators, Scene Manipulators	**genblinds genbox genclock genprism genrev gensky gensurf genworm mkillum replmarks xform**
`src/ot/`	Scene Compilers	**getbbox oconv**
`src/rt/`	Renderers	**lookamb rpict rtrace rview**
`src/px/`	Picture Filters and Translators	**falsecolor macbethcal normpat oki20 oki20c paintjet pcomb pcompos pcond pdfblur pextrem pfilt pflip pinterp pmblur protate psign pvalue ra_avs ra_bn ra_gif ra_pict ra_ppm ra_pr ra_pr24 ra_ps ra_rgbe ra_t16 ra_t8 ra_tiff ra_xyze ttyimage ximage xshowtrace**
`src/util/`	Utility Programs	**dayfact debugcal findglare getinfo glare glarendx objline objpict objview rad raddepend ranimate rlux rpiece trad vwright xglaresrc**

Table 1. The six main *Radiance* source subdirectories and the programs built there.

The easiest way to explain how *Radiance* programs are built is to select two example programs from each subdirectory, one typical and one atypical, and describe their compilations. We will begin with the `src/cv/` subdirectory (scene format translators), then move through the others in the order given in Table 1. Be sure to read the beginning of the next section, as it gives information that is valuable but not repeated later. Also, the `src/rt/` subdirectory (rendering programs) will be treated specially, with some hints on adding new device drivers to **rview** and creating new scene primitive types.

Scene Format Translators

The two example programs we will describe from the `src/cv/` subdirectory are **obj2rad** and **mgf2rad**, which correspond to a typical and an atypical scene converter, respectively.

In any *Radiance* source directory, you may use **rmake** to build individual programs or install all the programs for that directory using the special target "install". The **rmake** command is itself a short shell script that calls **make** with the appropriate options and variable settings as determined by **makeall** for this particular system. Let us look at a typical **rmake** script:

```
#!/bin/sh
exec make "SPECIAL=tiff" \
        "OPT=-O2" \
        "MACH=-DALIGN=double -cckr" \
        ARCH=sgi "COMPAT=malloc.o strcmp.o" \
        INSTDIR=/usr/local/bin \
        LIBDIR=/usr/local/lib/ray \
        "$@" -f Rmakefile
```

The variables used by **rmake** to control compilations in the various Rmakefile's are:

`SPECIAL` Specific modules to compile on this machine, which would normally be skipped.

`OPT` Compiler optimization options, which may affect performance but should not affect correctness.

`MACH` Compiler options needed to get *Radiance* to work on this machine.

`ARCH` The name of this machine architecture, which is used by some modules for more specific compilations.

`COMPAT` C library modules that should be replaced by *Radiance*-specific versions either for performance or correctness reasons.

`INSTDIR` The destination directory for executable binaries.

`LIBDIR` The central library directory for auxiliary *Radiance* files.

`MLIB` Alternative C math libraries.

`CC` The C compiler command name.

These settings may be altered manually if for some reason **makeall** misses something by editing the `rmake` file in the destination directory (`INSTDIR`).

The **obj2rad** program translates Wavefront .OBJ format files into *Radiance*. It uses the routines in `trans.c` to read in a user's rule file for mapping materials onto surfaces. (See the **obj2rad** man page or `doc/notes/translators` for details.) Additional routines in `tmesh.c` are used for smoothing triangulated meshes. Running "`rmake obj2rad`" results in the following compilations:

```
% rmake obj2rad
        cc -O2 -DALIGN=double -cckr \
-I../common -L../lib -c obj2rad.c
        cc -O2 -DALIGN=double -cckr \
-I../common -L../lib -c trans.c
        cc -O2 -DALIGN=double -cckr \
-I../common -L../lib -c tmesh.c
        cc -O2 -DALIGN=double -cckr \
-I../common -L../lib -o obj2rad obj2rad.o \
trans.o tmesh.o -lrt -lm
```

Note the appearance of `-I../common` and `-L../lib` on each compile line. These are necessary to find the common header files in `src/common/` and the common libraries in `src/lib/`. Specifically, `obj2rad.c` refers to `standard.h`, which is in `src/common/`, and **obj2rad** loads the following modules from the `src/lib/librt.a` library:

```
header.c       - reads and writes info. headers
fgetline.c     - gets backslash-escaped lines
fvect.c        - handles 3D vector math
savestr.c      - saves shared, read-only strings
badarg.c       - checks argument types
words.c        - checks word formats
eputs.c        - puts message to stderr
quit.c         - calls exit(1)
strcmp.c       - replacement for strcmp(3)
```

These descriptions (or something like them) may be found in the `src/common/README` file, and each source directory should contain an up-to-date list of source files and one line descriptions of each. Additionally, each source directory contains a `tags` file, which may be used by **vi** to quickly go between function and macro definitions in that directory (and `src/common/`). This is the easiest way to understand a program, by locating all of its constituent parts. Be careful, though, since some function names appear more than once, and the tag command may not always take you to the correct definition. When in doubt, check the SCCSid at the top of the file and compare it with the executable with the **what** command.

The translator **mgf2rad** is slightly more complicated, since it is based on the Materials and Geometry Format*, which has its own parser library. This library is built in the `src/cv/mgflib/` subdirectory then moved to `src/lib/libmgf.a` prior to linking. The compilation looks like this:

```
% rmake mgf2rad
        cd mgflib ; rm -f libmgf.a ;
        make libmgf.a CC=cc \
                CFLAGS="-O2 -DALIGN=double -cckr \
'-DMEM_PTR=char *' -DNOPROTO" ; \
        cp libmgf.a ../../lib
        cc -O2 -DALIGN=double -cckr \
'-DMEM_PTR=char *' -DNOPROTO -c parser.c
        cc -O2 -DALIGN=double -cckr \
'-DMEM_PTR=char *' -DNOPROTO -c context.c
        cc -O2 -DALIGN=double -cckr \
'-DMEM_PTR=char *' -DNOPROTO -c xf.c
        cc -O2 -DALIGN=double -cckr \
'-DMEM_PTR=char *' -DNOPROTO -c object.c
        cc -O2 -DALIGN=double -cckr \
'-DMEM_PTR=char *' -DNOPROTO -c lookup.c
        cc -O2 -DALIGN=double -cckr \
'-DMEM_PTR=char *' -DNOPROTO -c badarg.c
        cc -O2 -DALIGN=double -cckr \
'-DMEM_PTR=char *' -DNOPROTO -c words.c
        cc -O2 -DALIGN=double -cckr \
'-DMEM_PTR=char *' -DNOPROTO -c fvect.c
        ar rc libmgf.a parser.o context.o \
xf.o object.o lookup.o badarg.o  words.o fvect.o
        ranlib libmgf.a
        cc -O2 -DALIGN=double -cckr \
-I../common -L../lib '-DMEM_PTR=char *' \
-DNOPROTO -c mgf2rad.c
        cc -O2 -DALIGN=double -cckr \
-I../common -L../lib -o mgf2rad mgf2rad.o \
tmesh.o -lmgf -lrt -lm
```

* The Materials and Geometry Format was developed by the authors as a neutral exchange format for lighting simulation and rendering. For further information, see the MGF web site at "http://radsite.lbl.gov/mgf/HOME.html".

The first part calls **make** in the `mgflib/` subdirectory and moves `libmgf.a` to `src/lib/`, and the second part compiles and links `mgf2rad.c` and `tmesh.c` to the necessary libraries.

Generators and Scene Manipulators

Most generator programs rely on little else besides the basic C module containing the main function, since generating *Radiance* scene files can be accomplished easily with simple calls to `printf(3)`. In some cases, as with **gensurf**, the generator uses the functional language, and therefore needs some of the `cal*` modules from the `src/common/` directory (compiled into the `src/lib/librt.a` library). Unlike generators, scene manipulation programs, such as **replmarks** and **xform**, read as well as write scene descriptions, and may require additional library support. The most exotic program built in this directory by far is **mkillum**, which not only reads in scene descriptions, but calls **rtrace** as a subprocess to compute radiance distributions for surfaces. We will use **mkillum** as our example of an unusual compilation for the `src/gen/` directory, and **genrev** will serve as our more typical example.

The compilation of **genrev** looks something like this:

```
% rmake genrev
        cc -DALIGN=double -cckr -O2 \
-I../common -L../lib -c genrev.c
        cc -DALIGN=double -cckr -O2 \
-I../common -L../lib -o genrev genrev.o -lrt -lm
```

The functional language modules used from `src/common/` are `caldefn.c`, `calfunc.c`, and `calexpr.c`. These modules allow **genrev** to parse and evaluate variable and function definitions that describe the parametric shape of a surface of revolution. Many programs utilize these same library modules and use the functional language in different capacities.

As promised, the compilation of **mkillum** is more complicated, and is broken into three main modules, `mkillum.c`, `mkillum2.c` and `mkillum3.c`, which share the common header file `mkillum.h`. The compilation looks like this:

```
% rmake mkillum
        cc -DALIGN=double -cckr -O2 \
I../common -L../lib -c  mkillum.c
        cc -DALIGN=double -cckr \
-O2 -I../common -L../lib -c mkillum2.c
        cc -DALIGN=double -cckr -DSPEED=60 \
-O2 -I../common -L../lib -c mkillum3.c
        cc -DALIGN=double -cckr \
-O2 -I../common -L../lib -o mkillum \
mkillum.o mkillum2.o mkillum3.o -lrt -lm
```

This program relies quite heavily on the modules in `src/common/`, including the following:

```
error.c         - error reporting function
getpath.c       - search for full path to file
process.c       - administrate subprocess
urand.c         - low-discrepancy sequence
generator
otypes.c        - determine primitive type
readfargs.c     - read primitive argument list
face.c          - initialize polygon primitive
multisamp.c     - multi-dimensional LDS
cone.c          - initialize cone/cylinder/ring
mat4.c          - 4x4 matrix computations
```

Using these routines, **mkillum** is able to read in *Radiance* scene files and generate ray samples for **rtrace** to compute outgoing radiance values. These are then collected into data files, which are referenced in a modified scene description and sent to the standard output.

Scene Compilers

There are currently two programs compiled in the `src/ot/` directory, **genbox** and **oconv**. The **oconv** program generates an *octree* for the given scene file(s), which is then used to accelerate the ray tracing process. The **genbox** program is a gutted version of **oconv** whose sole purpose is to compute the bounding box of one or more scene files. The compilation of **oconv** looks like this:

```
% rmake oconv
        cc -DSTRICT -DALIGN=double -cckr \
-O2 -I../common -L../lib -c oconv.c
        cc -DSTRICT -DALIGN=double -cckr \
-O2 -I../common -L../lib -c sphere.c
        cc -DSTRICT -DALIGN=double -cckr \
-O2 -I../common -L../lib -c writeoct.c
        cc -DSTRICT -DALIGN=double -cckr \
-O2 -I../common -L../lib -c o_face.c
        cc -DSTRICT -DALIGN=double -cckr \
-O2 -I../common -L../lib -c o_cone.c
        cc -DSTRICT -DALIGN=double -cckr \
-O2 -I../common -L../lib -c o_instance.c
        cc -DSTRICT -DALIGN=double -cckr \
-O2 -I../common -L../lib -c bbox.c
        cc -DSTRICT -DALIGN=double -cckr \
-O2 -I../common -L../lib -c initotypes.c
        cc -DSTRICT -DALIGN=double -cckr \
-O2 -I../common -L../lib -DMEMHOG -c readfargs.c
        cc -DSTRICT -DALIGN=double -cckr \
-O2 -I../common -L../lib -c malloc.c
        cc -DSTRICT -DALIGN=double -cckr \
-O2 -I../common -L../lib -o oconv oconv.o
writeoct.o sphere.o o_face.o \
o_cone.o o_instance.o bbox.o readfargs.o \
initotypes.o malloc.o -lrt -lm
```

The `-DSTRICT` option makes sure that **oconv** generates tight bounds around its surfaces. Without this, **oconv** would produce octrees slightly faster (especially if there are many cones), but rendering times might be significantly longer in certain cases. Even though the `readfargs.c` module is compiled in the `src/lib/librt.a` library, it is recompiled here with the `-DMEMHOG` flag to avoid the memory overhead associated with `malloc()` by substituting `bmalloc()` instead.

Once **oconv** has been built, **getbbox** requires only a few special-purpose modules, `init2otypes.c` and `readobj2.c`. These substitute certain function assignments and scene parsing code to avoid storing the model in memory like the standard routines. Also, the library-compiled `readfargs.c` is used because memory is being freed shortly after it is read. The compilation looks like this:

```
% rmake getbbox
        cc -DSTRICT -DALIGN=double -cckr \
-O2 -I../common -L../lib -c getbbox.c
        cc -DSTRICT -DALIGN=double -cckr \
-O2 -I../common -L../lib
        cc -DSTRICT -DALIGN=double -cckr \
-O2 -I../common -L../lib -c init2otypes.c
        cc -DSTRICT -DALIGN=double -cckr \
-O2 -I../common -L../lib -o getbbox getbbox.o
readobj2.o bbox.o init2otypes.o -lrt -lm
```

Rendering Programs

As one might expect, the compilation of the rendering programs is the most complicated. There are many object files local to the `src/rt/` subdirectory and many library modules linked in as well. Other modules have symbolic links to the sources in `src/common`, but are recompiled locally to enable or disable specific features and optimize performance.

Rather than reproducing the long and tedious compile here, let us look in some detail at `src/rt/Rmakefile` instead. This will give us better insight into what is going on and how we might make modifications to the code or the compilation process. `Rmakefile` begins with the following variable settings, which will often be overridden by the `rmake` script:

```
OPT = -O
MACH = -DBSD
CFLAGS = $(MACH) $(OPT) -I../common -L../lib
SPECIAL = aed
CC = cc
MLIB = -lm
```

The `CFLAGS` variable is usually left alone, and affected indirectly instead by the settings of the `MACH` and `OPT` variables. An exception to this might occur if a particular C compiler wants a space between its `-I` and `-L` options and their arguments. These options are essential to the compiler finding the *Radiance*-specific header and library files it needs.

Rview Device Drivers

Skipping down a bit, we reach the variables corresponding to device drivers needed for the **rview** program:

```
#
# Device drivers for rview (see also devtable.c):
#
DOBJS = devtable.o devcomm.o editline.o \
     x11.o x11twind.o colortab.o
DSRC = devtable.c devcomm.c editline.c \
     x11.c x11twind.c colortab.c
DLIBS = -lX11
```

These are the objects that will be linked directly into **rview**, and which ones are needed is determined by the contents of `src/rt/devtable.c`:

```
/*
 *  devtable.c - device table for rview.
 */

#include  "driver.h"

char  dev_default[] = "x11";

extern struct driver  *x11_init();

struct device  devtable[] = {            /* supported
devices */
      {"slave", "Slave driver", slave_init},
      {"x11", "X11 color or greyscale display", x11_init},
      {"x11d", "X11 display using stdin/stdout", x11_init},
      {0}                           /* terminator */
};
```

Originally, **rview** supported more devices than X11, and through the routines in `src/rt/devcomm.c`, it still can. One of these routines is `slave_init()`, which sets up `stdin` and `stdout` to act as communication channels between **rview** and its parent process, through which control commands are taken in and display commands are sent by **rview**. Another routine, `comm_init()`, permits unlinked device drivers to be used through UNIX interprocess communication channels (i.e., pipes). The simple driver protocol is defined and described in `src/rt/driver.h`. Creating a new device driver means following the templates for the requisite device driver routines, and either linking it into rview directly via `devtable` or as a separate process by linking to the routines in `src/rt/devmain.c`.

As an example of this, let us look at the original X Version 10 driver, which may be linked in a separate program executable for ancient systems that still support it. In `src/rt/Rmakefile`, we find the following lines:

```
$(DEVDIR)/x10:  x10.o xtwind.o colortab.o \
devmain.o editline.o
        $(CC) $(CFLAGS) -s -o $(DEVDIR)/x10 \
x10.o xtwind.o devmain.o colortab.o \
editline.o -lX $(LIBS)

x10.o:  x10.c
        $(CC) $(CFLAGS) -Dx_init=dinit -c x10.c
```

Since the X10 library cannot be linked to the same program that links to the X11 library (due to name collisions and numerous other problems), this driver *must* be a separate executable. Other drivers could share the same name space as the rest of **rview**, but putting them in a separate executable might still make sense if they are rarely used and/or take up a lot of program memory whether they are used or not, as is the case with the SunView driver. The special compilation of a separate driver executable requires redefining the initialization routine because the `main()` function in `src/rt/devmain.c` always calls `dinit()` as its driver initialization routine. This shows up in the define used for compiling `x10.o` above.

The `src/rt/Rmakefile` variable `DEVDIR` determines where driver executables are stored. This can be a standard location, but is usually a subdirectory of the *Radiance* `DESTDIR` executable directory so that they are not accidentally invoked by a user, and regular programs are not mistaken by **rview** for drivers.

If desired, a driver compiled separately in this way may be entered into `devtable` with `comm_init()` as its initializing routine so that it shows up when **rview -devices** is run, but this is not necessary. Any driver given to **rview** with the **-o** option that is not found in `devtable` will be handed to `comm_init()` as a possible external driver program name.

Rendering Modules And Version String

Getting back to `src/rt/Rmakefile`, we see the definition of several variables to hold the many source and object files of the rendering programs. The commonalty between **rtrace**, **rpict** and **rview** is evident in the three variables that define their differences:

```
RTOBJS = rtmain.o rtrace.o duphead.o persist.o \
preload.o $(ROBJS) Version.o

RPOBJS = rpmain.o rpict.o srcdraw.o duphead.o \
persist.o preload.o $(ROBJS) Version.o

RVOBJS = rvmain.o rview.o rv2.o rv3.o \
freeobjmem.o $(DOBJS) $(ROBJS) Version.o
```

Here we see that each program has its own special module, called respectively, `rtrace.o`, `rpict.o` and `rview.o`. In addition, **rtrace** and **rpict** link to `duphead.o`, `persist.o` and `preload.o`, which are needed for persistent and parallel execution (the **-P** and **-PP** options). The **rview** link includes `rv2.o`, `rv3.o`, `freeobjmem.o` and the device driver objects (`DOBJS`) mentioned earlier. The module `Version.o` is special and deserves some mention here.

The source file `src/rt/Version.c` indicates the current renderer version, and usually looks something like this:

```
/*
 * This file was created automatically during
make.
 */

char VersionID[]="RADIANCE 3.1a lastmod Sat Jul 27
09:01:38 PDT 1996 by greg on hobbes";
```

This file is created automatically and is used to identify the particular version of the *Radiance* renderer. In `src/rt/Rmakefile`, we see this module is dependent on all the common rendering source and header files. Thus, any change to any of the constituent source code will cause `src/rt/Version.c` to be rebuilt with the name of the user compiling the new version and when it was compiled.

Adding a New Primitive Type

One of the most common source modifications is adding a new scene primitive type. This involves changes first to `src/common/otypes.h`, where all the primitives are defined and named. This header file is used by a number of *Radiance* programs, including **oconv**, **xform** and the renderers **rtrace**, **rpict** and **rview**. If the new primitive being added is a material, pattern or texture, it may not be necessary to modify the code to **oconv** or **xform** unless changes in coordinates somehow affect the parameters, in which case an appropriate routine must be added to `src/gen/xform.c`. If the new primitive is a surface type, then it will be necessary to write an octree intersection function in the `src/ot/` directory as well as a transformation routine in `src/gen/xform.c`.

A material, pattern or texture primitive means that new code must be added to `src/rt/` to implement whatever it is the new primitive does. A new link is then added to `src/rt/Rmakefile` and `src/rt/initotypes.c`.

Picture Filters and Translators

More programs are built in the `src/px/` subdirectory than any other. This is due partly to the many interesting and orthogonal operations that may be performed on picture files, and partly to the plethora of other image formats available for translation. In fact, the current release of *Radiance* supports only a small subset of the existing image formats, and anyone who wants to volunteer their services in creating new ones will get nothing but encouragement.

Returning to our earlier expository style, we pick two programs from the `src/px/` directory and explain their compilations. The first program, **pfilt**, is the main picture filter for anti-aliasing, exposure setting and resizing. The command **rmake pfilt** yields the following:

```
% rmake pfilt
        cc -O2 -DALIGN=double -cckr -I../common \
-L../lib -c pfilt.c
        cc -O2 -DALIGN=double -cckr -I../common \
-L../lib -c pf2.c
        cc -O2 -DALIGN=double -cckr -I../common \
-L../lib -c pf3.c
        cc -O2 -DALIGN=double -cckr -I../common \
-L../lib -o pfilt pfilt.o pf2.o pf3.o -lrt -lm
```

From the `src/lib/librt.a` library, the following modules are also loaded into **pfilt**:

```
badarg.c        - check arg list against format
color.c         - routines for scanline i/o
fropen.c        - find and open a library file
fvect.c         - routines for float vectors
getlibpath.c    - return standard library path from
image.c         - routines for image generation
lamps.c         - load lamp data
resolu.c        - read and write image resolutions
rexpr.c         - regular expression parser
spec_rgb.c      - convert colors and spectral
ranges
words.c         - routines for recognizing words
```

Many of these modules are needed for reading lamp data for color balancing. Specifically, `fropen.c`, `getlibpath.c`, `lamps.c`, `rexpr.c` and `spec_rgb.c` are not needed except to support the **pfilt -t** option. The main module needed are `color.c` for reading and writing picture scanlines.

The second program we will look at is the image translator, **ra_t8**, which converts *Radiance* pictures to and from 8-bit Targa format. The compile looks like this:

```
% rmake ra_t8
        cc -O2 -DALIGN=double -cckr -I../common \
-L../lib -c ra_t8.c
        cc -O2 -DALIGN=double -cckr -I../common \
-L../lib -c clrtab.c
        cc -O2 -DALIGN=double -cckr -I../common \
-L../lib -c neuclrtab.c
        cc -O2 -DALIGN=double -cckr -I../common \
-L../lib -o ra_t8 ra_t8.o clrtab.o neuclrtab.o \
-lrt -lm
```

The module `src/px/clrtab.c` implements Paul Heckbert's median-cut color quantization and `src/px/neuclrtab.c` implements Anthony Dekker's neural-net quantization, respectively [Heckbert82][Dekker94]. (Thanks to both these authors for their help in implementing these routines.) The other routines loaded from the standard library are:

```
color.c         - routines for scanline i/o
colrops.c       - integer operations on COLR data
header.c        - information header i/o
resolu.c        - read and write image resolutions
```

The routines in `src/common/colrops.c` are important for improving the performance of the program by avoiding floating point operations and conversions on the *Radiance* picture scanlines. These routines are also used in the conversion of 24-bit color images, and a convenient starting point for writing new image translators is the skeletal converter `src/px/ra_skel.c`.

Utility Programs

The `src/util/` subdirectory builds programs that do not fit conveniently into any of the previous categories. Some of these programs perform handy little functions like computing new views from old ones (**vwright**), while others are executive interfaces capable of integrating and coordinating other *Radiance* programs (**rad**, **trad** and **ranimate**). Since this is really a miscellaneous collection, there are really no typical or unusual programs. Instead, we give examples of two subclasses of utilities. The first example is **rpiece**, a program that runs **rpict** to do something that it could not do as easily by itself. The second example is an executive program that coordinates rendering tasks, **rad**.

The program **rpiece** is a control program for running **rpict** in parallel on one or more hosts. Its compilation looks like this:

```
% rmake rpiece
        cc -DALIGN=double -cckr -O2 -I../common \
-L../lib -c rpiece.c
        cc -DALIGN=double -cckr -O2 -I../common \
-L../lib -c Version.c
        cc -DALIGN=double -cckr -O2 -I../common \
-L../lib -o rpiece rpiece.o Version.o -lrt -lm
```

The only thing unusual about this compilation is the inclusion of `Version.c`, which is actually a symbolic link to `src/rt/Version.c`, the module automatically built during compilation of the renderers to indicate what version of the software is being created. This is the only example of a link to a source module someplace besides `src/common/`. This module is used by **rpiece** to indicate the software version in the header of its output picture.

As our example of an executive program in `src/util/`, we look at **rad**, whose compilation looks something like this:

```
% rmake rad
        cc -DALIGN=double -cckr -O2 -I../common \
-L../lib -c rad.c
        cc -DALIGN=double -cckr O2 -I../common \
L../lib -c loadvars.c
        cc -DALIGN=double -cckr -O2 -I../common \
-L../lib -o rad rad.o loadvars.o -lrt -lm
```

The `src/util/loadvars.c` module is common to both **rad** and **ranimate**, and contains routines for reading control files with variable assignments, as described in their respective manual pages. Links to the `src/lib/librt.a` library provide a few additional capabilities, but most of the utility of this and the other executive programs is provided by the running of the other *Radiance* programs such as **oconv**, **mkillum**, **rpict** and **pfilt**.

Conclusion

We have given an overview of the principal *Radiance* source directories and examples of some program compilations with the hope that this will start the interested reader in their investigation of the code itself. We hope that the relatively simple organization of the source code into a `src/common/` subdirectory with shared headers and source files and six logically organized program categories will simplify the understanding of the system. We did not discuss the presence of other file types besides C program source, but certain auxiliary files (e.g., `*.cal`), C-shell and Tcl/Tk scripts are included also in the source directories either because they are needed by C programs or they are program source in and of themselves.