RADIANCE Tutorial

Greg Ward
Lawrence Berkeley Laboratory

1. Introduction

RADIANCE is a lighting simulation program that synthesizes images from 3-dimensional geometric models of physical spaces. The input model describes each surface's shape, size, location and composition. A model may contain many thousands of surfaces, and is often produced by a separate CAD program. In addition to arbitrary (planar) polygons, RADIANCE directly models spheres and cones. Generator programs are provided for the creation of more complex shapes from these basic surface primitives. Examples include boxes, prisms and surfaces of revolution. A transformation utility permits the simple duplication of objects and hierarchical construction of a scene.

This tutorial assumes a certain familiarity with the UNIX operating system and its text editing facilities. Ask your system administrator for a basic introduction.

The RADIANCE reference manual will be required to understand the following examples of scene creation and program interaction. Text in *italics* is user input.

2. Input of a Simple Room

In this example, we will use a text editor to create the input for a simple room containing a box, a ball, and a light source. In most applications, a CAD system would be used to describe a scene's geometry, which would then be combined with surface materials, light fixtures, and (optionally) furniture. To get a more intimate understanding of the input to RADIANCE, we will start without the advantages of a CAD program or an object library.

The scene we will be working towards is shown in Figure 1. It is usually helpful to start with a simple drawing showing the coordinate axis and the relative locations of major surfaces.

The minimum input required to get an image is a source of illumination and an object to reflect light to the "camera".* We will begin with two spheres, one emissive and one reflective. First we define the materials, then the spheres themselves. Actually, the order is only important insofar as each modifier definition (i.e. material) must appear before its first reference. (Consult the RADIANCE manual for an explanation of the primitive types and their parameters.) Start your favorite text editor ("vi" in this example) to create the following file called "room.rad":

---

*In fact, a RADIANCE renderer can be thought of an invisible camera in a simulated world.

% *vi room.rad*

```
#
#  My first scene.
#

# this is the material for my light source:

void light bright
0
0
3  100  100  100

# this is the material for my test ball:

void plastic red_plastic
0
0
5  .7  .05  .05  .05  .05

# here is the light source:

bright sphere fixture
0
0
4  2  1  1.5  .125

# here is the ball:

red_plastic sphere ball
0
0
4  .7  1.125  .625  .125
```

Now that we have a simple scene description, we can look at it with the interactive viewing program, **rview.** First, however, we must create the "octree" file that will be used to accelerate the rendering process. To accomplish this, type the following command:

% *oconv room.rad > test.oct*

Note that the suffixes ".rad" and ".oct" are not enforced by the program, but are merely a convenience to aid the user in identifying files later. The command **getinfo** can be used to get information on the origin of binary (unviewable) files created by RADIANCE utilities. Try entering the command:

% *getinfo test.oct*

The usefulness of such a function will be apparent when you find yourself with a dozen files named "test.pic".

To make an image of our scene, we must select a suitable set of view parameters telling RADIANCE where to point its camera. To simplify our example, we will use the same starting position for all our renderings, and change views only once **rview** is started:

% *rview -vp 2.25 .375 1 -vd -.25 .125 -.125 -av .5 .5 .5 test.oct*

The "-vp" option gives the view point, and the "-vd" option gives the view direction vector. The "-av" option specifies the amount of light globally present in the scene, permitting portions of the scene that are not illuminated directly to be visible. **Rview** has many more options, and their default values may be discovered using:

% *rview -defaults*

You should start to see an image of a red ball forming on your screen. Take this opportunity to try each of **rview's** commands, as described in the manual. If you make a mistake in a view specification, use the **last** command to get back to where you were. It is probably a good idea to save your favorite view using the rview command:

: *view default.vp*

You can create any number of "viewfiles" with this command, and retrieve them with:

: *last* viewfile

If you look around enough, you may even be able to see the light source itself. Unlike many rendering programs, the light sources in RADIANCE are visible objects. This illustrates the philosophy behind the program, which is the simulation of physical spaces. Since it is not possible to create an invisible light source in reality, there is no reason to do it in simulation.

Still, there is no guarantee that the user will create physically meaningful descriptions. For example, we have just floated a red ball next to a light source somewhere in intergalactic space. In the interest of making this scene more realistic, let's enclose the light and ball in a room by adding the following text to "room.rad":

% *vi room.rad*

```
# the wall material:

void plastic gray_paint
0
0
5  .5  .5  .5  0  0

# a box shaped room:

!genbox gray_paint room  3  2  1.75  -i
```

The generator program **genbox** is just a command that produces a RADIANCE description, and it is executed when the file is read. It is more convenient than specifying the coordinates of four vertices for each of six polygons, and can be changed later quite easily. (See the manual genbox page for further details.)

You can now look at the modified scene, but remember first to regenerate the octree:

```
% oconv room.rad > test.oct
% rview -vf default.vp -av  .5  .5  .5  test.oct
```

This is better, but our ball and light source are still floating, which is an unrealistic condition for most rooms. Let's put a box under the table, and a rod to suspend the light from the ceiling:

```
# a shiny blue box:

void plastic blue_plastic
0
0
5 .1 .1 .6 .05 .1

!genbox blue_plastic box  .5  .5  .5  | xform -rz  15  -t  .5  .75  0

# a chrome rod to suspend the light from the ceiling:

void metal chrome
0
0
5  .8  .8  .8  .9  0

chrome cylinder fixture_support
0
0
7
        2       1       1.5
        2       1       1.75
        .05
```

Note that this time the output of genbox was "piped" into another program, **xform.** **Xform** is used to move, scale and rotate RADIANCE descriptions. **Genbox** always creates a box in the positive octant of 3-space with one corner at the origin. This was what we wanted for the room, but we wanted the box moved away from the wall and rotated slightly. First we rotated the box 15 degrees about the z-axis (pivoting on the origin), then translated the corner from the origin to (.5,.75,0). By no small coincidence, this position is directly under our original ball.

After viewing this new arrangement, you can try changing some of the materials -- here are a few examples:

solid crystal:

void dielectric crystal
0
0
5 .5 .5 .5 1.5 0

dark brown:

void plastic brown
0
0
5 .2 .1 .1 0 0

light gray:

void plastic white
0
0
5 .7 .7 .7 0 0

To change the ball from red_plastic to the crystal defined above, simply replace *red_plastic sphere ball* with *crystal sphere ball*. Note once again that the definition of the new materials must precede any references to them. Changing the materials for the floor and ceiling of the room is a little more difficult. Since **genbox** creates six rectangles, all using the same material, it is necessary to replace the command with its output before we can make the required changes. To do this, enter the command directly:

% *genbox gray_paint room 3 2 1.75 -i >> room.rad*

The double arrow ">>" causes the output to be "appended" to the end of the file, rather than overwriting its contents. Now edit the file and change the ceiling material to "white", and the floor material to "brown". (Hint: the ceiling is the polygon whose z coordinates are all high. And don't forget to remove the original **genbox** command from the file!)

Once you have chosen a nice view, you can generate a high-resolution image in batch mode using the **rpict** command:

% *rpict -vf* myview *-av .5 .5 .5 test.oct* > *test.pic &*
[PID]

The ampersand "&" causes the program to run in the background, so you can log out and go home while the computer continues to work on your picture. The bracketed number [PID] printed by the C-shell command interpreter is the process id that can be used later to check the progress or kill the program. This number can also be determined by the **ps** command:

% *ps*

The number preceding the **rpict** command is the process id. If you want to kill the process later, use the command:

% *kill* PID

If you only want a progress report, use the form:

% *kill -CONT* PID

This sends a "continue" signal to **rpict,** which then prints out the percentage completion. Note that this is a special feature of rpict, and will not work with most programs. Also note that this works only for the current login session. If you log on later on a different terminal (or window), **rpict** will not send the report to the correct place. It is usually a good idea, therefore, to give **rpict** an error file argument if it is running a long job:

% *rpict -e* errfile ...

Now sending an alarm signal will result in **rpict** reporting to the end of the specified error file. Alternatively, you can use the *-t* option to generate reports automatically at regular intervals. You can check the reports at any time by printing the file:

% *cat* errfile

This file will also contain a header, and any errors that occurred.

3. Addition of a Window

Adding a window to the room requires two basic steps. The first step is to cut a hole in the wall an put in a piece of glass. The second step is to put something outside to make the view worth having. Since there are no explicit holes allowed in RADIANCE polygons, we use the trick of coincident edges (making a seam) to give the appearance of a hole. The new polygon for the window wall is shown in Figure 2.

To create the window wall, change the appropriate polygon in the scene file (modified part in italics). If you haven't done so already, follow the instructions in the previous section to change the **genbox** command in the file to its corresponding polygons.

Figure 2. The window wall with a hole cut in it.

% *vi room.rad*

```
gray_paint polygon room.5137
0
0
30
        3       2       1.75
        3       2       0
        3       0       0
        3       0       1.75
        3       .625    1.75
        3       .625    .625
        3       1.375 .625
        3       1.375 1.375
        3       .625    1.375
        3       .625    1.75
```

Next, create a separate file for the window. (The use of separate files is desirable for parts of the scene that will be manipulated independently, as we will see in a moment.)

% *vi window.rad*

```
# an 88% transmittance glass window has a transmission of 96%:

void glass window_glass
0
0
```

```
3 .96 .96 .96

window_glass polygon window
0
0
12
        3       .625   1.375
        3       1.375 1.375
        3       1.375 .625
        3       .625   .625
```

The vertex order is very important, especially for polygons with holes. Normally, vertices are listed in counterclockwise order as seen from the front. However, the hole of a polygon has its vertices listed in the opposite order. This insures that the seam does not cross itself.

The next step is the description of the scene outside the window. A special purpose generator, **gensky,** will create a description of the sun and sky which will be stored in a separate file. The arguments to **gensky** are the month, day and hour (local standard time). The following command produces a description for 10:00 solar time on March 20th at latitude of 40 degrees:

```
% gensky 3 20 10 -o 0 -m 0 -a 40 > sky.rad
```

The file "sky.rad" contains only a description of the sun and the sky distribution. The actual sky and ground are still undefined, so we will create another short file containing a generic background:

```
% vi outside.rad

#
# A standard sky and ground to follow a gensky sun and sky distribution.
#

skyfunc glow sky_glow
0
0
4 .9 .9 1.15 0

sky_glow source sky
0
0
4 0 0 1 180

skyfunc glow ground_glow
0
0
4 1.4 .9 .6 0

ground_glow source ground
```

0
0
4 0 0 -1 180

We can now put these elements together in one octree file using **oconv:**

% *oconv outside.rad sky.rad window.rad room.rad* > *test.oct*

Note that the above command results in the following error message:

oconv: fatal - (outside.rad): undefined modifier "skyfunc"

The modifier is undefined because we put "outside.rad", which uses *skyfunc* before "sky.rad" where *skyfunc* is defined. It is therefore necessary to change the order of the files so that *skyfunc* is defined before it is used:

% *oconv sky.rad outside.rad window.rad room.rad* > *test.oct*

Now let's look at our modified scene, using the same rview command as before:

% *rview -vf default.vp -av .5 .5 .5 test.oct*

As you look around the scene, you will need to adjust the exposure repeatedly to be able to see detail over the wide dynamic range now present. To do this, wait a few seconds after choosing each new view and enter the command:

: *exposure 1*

or simply:

: *e 1*

All commands in **rview** can be abbreviated by one or two letters. Additional control over the exposure is possible by changing the multiplier factor to a value greater than one to lighten or less than one to darken. It is also possible to use absolute settings and spot normalization. See the **rview** manual entry for details.

You may notice that other than a patch of sun on the floor, the window does not seem to illuminate the room. In RADIANCE, certain surfaces act as light sources and others do not. Whether or not a surface is a light source is determined by its material type. Surfaces made from the material types *light, illum, spotlight* and (sometimes) *glow* will act as light sources, whereas surfaces made from *plastic, metal, glass* and other material types will not. In order for the window to directly illuminate the room, it is therefore necessary to change its material type. We will use the type *illum* because it is specially designed for "secondary" light sources such as windows and other bright objects that are not merely emitters but have other important visual properties. An *illum* will act as a light source for parts of the calculation, but when viewed directly will appear as if made from a different material (or disappear altogether).

Rather than modify the contents of "window.rad" which is a perfectly valid description of a non-source window, let's create a new file, which we can substitute during octree creation, called "srcwindow.rad":

```
% vi srcwindow.rad

#
# An emissive window
#

# visible glass type for illum:

void glass window_glass
0
0
3 .96 .96 .96

# window distribution function, including angular transmittance:

skyfunc brightfunc window_dist
2 winxmit winxmit.cal
0
0

# illum for window, using 88% transmittance at normal incidence:

window_dist illum window_illum
1 window_glass
0
3 .88 .88 .88

# the source polygon:

window_illum polygon window
0
0
12
        3       .625    1.375
        3       1.375   1.375
        3       1.375   .625
        3       .625    .625
```

There are a couple of things you should notice in this file The first definition is the normal glass type, window_glass, which is used for the alternate material for the *illum* window_illum. Next is the window distribution function, which is the sky distribution modified by angular transmittance of glass defined in winxmit.cal. Finally comes the *illum* itself, which is the secondary source material for the window.

To look at the scene, simply substitute "srcwindow.rad" for "window.rad" in the previous **oconv** command, thus:

% *oconv sky.rad outside.rad srcwindow.rad room.rad* > *test.oct*

You can look at the room at different times by changing the **gensky** command used to create "sky.rad" and regenerating the octree. (Although the octree does not strictly need to be recreated for every change to the input files, it is good to get in the habit until the exceptions are well understood.)

4. Outside Geometry

If the exterior of a space is not approximated well by an infinitely distant sky and ground, we can add a better description to calculate a more accurate window output distribution as well as a better view outside the window. Let's add a ground plane and a nearby building to the "outside.rad" file we created earlier:

```
# Terra Firma:

void plastic ground_mat
0
0
5 .28 .18 .12 0 0

ground_mat ring groundplane
0
0
8
	0	0	-.01
	0	0	1
	0	30

# A big, ugly, mirrored glass building:

void mirror reflect20
0
0
3 .15 .2 .2

!genbox reflect20 building 10 10 2 | xform -t 10 5 0
```

Note that the groundplane was given a slightly negative z value. This is very important so that the ground does not peek through the floor we have defined. The material type *mirror,* used to define the neighboring structure, is special in RADIANCE. Surfaces of this type as well as the types *prism1* and *prism2* participate in something called the "virtual light source" calculation. In short, this means that the surfaces of the building we have created will reflect sunlight and any other light source present in our scene. The virtual light source material types should be used with care since they can result in substantial growth in the calculation. It would be a good idea in the example above to remove the bottom surface of the building (which cannot be seen from the outside anyway) and change the roof type to metal or some non-reflecting material. This can be done using the same manual process described earlier for changing the room surface materials.

Now that we have a better description of the outside, what do we do with it? If we simply substitute it in our scene without changing the description of the window *illum,* the distribution of light from the window will be slightly wrong because the "skybright" function only describes light from the sky and the ground, not from other structures. Using this approximation might be acceptable in some cases, but let's assume that accuracy is one of our main goals in life. There are two ways to an accurate calculation of light from a window. The first is to treat the window as an ordinary window and rely on the default interreflection calculation of RADIANCE, and the second is to use the program **mkillum** to calculate the window distribution separately so we can still treat it as an *illum* light source. Let's try them both.

Using the default interreflection calculation is probably easier, but as we shall see it takes a little longer to get a good result in this case. To use the interreflection calculation, we first create an octree using the original glass window and our new description of the outside:

% *oconv sky.rad outside.rad window.rad room.rad* > *test.oct*

Now we just add a few new options to our rview command like so:

% *rview -vf default.vp -dr 1 -av .5 .5 .5 -ab 1 -ad 200 -as 80 -ar 80 test.oct*

The *–dr* parameter tells **rview** how many reflections to allow in the virtual light source calculation. Because of the geometry of our scene, there will never be more than one reflection from our exterior building. The *–ab* parameter tells **rview** how many diffuse interreflections to follow. When this value is 0, there is no interreflection calculation. By setting this parameter to 1, we compute the first diffuse interreflection (eg. from the sun patch to the wall) as well as the contribution from the sky and exterior objects through the window. The *–ad* option tells **rview** how many rays to use in its initial sampling of the hemisphere for the diffuse interreflection calculation. We have set this value high enough so the program can find the relatively small window and its sun patch. The *–as* option tells the program how many rays to send to particularly bright areas of the hemisphere. Since we know that we will have such bright areas, we set this value to something which is around half the initial sample. The *–ar* option tells **rview** at what resolution it can relax the accuracy of the interreflection calculation. This number is relative to the overall size of our scene, which can be determined using *getinfo* on the octree like so:

% *getinfo -d test.oct*

This command reveals that our scene has an overall dimension of 60 (due to the large ring we have used for our ground plane). In order to have good resolution in our interior space, we set the value so that 60 divided by this parameter is somewhat less than one quarter the dimension of our interior space. Unfortunately, many of the calculation parameters used in Radiance are difficult to set correctly the first time, and require some experience as well as knowledge about the calculation.

Probably the first thing you notice after starting **rview** is that nothing happens. It takes the calculation a while to get going, as it must trace may rays at the outset to determine the contribution at each point from the window area. Once **rview** has stored up some values, the progress rate improves, but it never really reaches blistering speed.

A more efficient alternative in this case is to use the program **mkillum** to create a modified window file that uses calculated data values to define its light output distribution. Applying mkillum is relatively straightforward in this case. You simply give it an octree defined with the normal window and the file describing this window and it creates a new window description using calculated data values:

% *oconv sky.rad outside.rad window.rad room.rad > test.oct*
% *mkillum -av 18 18 18 -ab 0 test.oct < window.rad > mkiwindow.rad*

The *−av* value is appropriate for the outside, which is much brighter, as suggested by the output of the gensky command stored in "sky.rad". The *−ab* option is set to 0, because outside the building we do not expect interreflections to play as important a role as in the interior (plus we are trying to save some time). The **mkillum** command may take a few minutes, so be patient. If we look at the output file from *mkillum,* we see that it uses a pattern type called *brightdata,* which refers to a data file called "illum.dat". This file contains the results of an **rtrace** calculation, which traced many rays in each direction from the window in order to determine its brightness distribution. Now we can create an octree using this new file and render it with **rview,** this time without the interreflection calculation:

% *oconv sky.rad outside.rad mkiwindow.rad room.rad > test.oct*
% *rview -vf default.vp -av .5 .5 .5 test.oct*

You will notice that the calculation proceeds much more quickly, and even produces a smoother looking result. Aside from waiting for mkillum to finish, there is an additional price for this speed advantage, however. The contribution from the sun patch on the floor is no longer being considered, since we are not performing an interreflection calculation inside our space. The light from the window is being taken care of by the mkillum output, but the solar patch is not. In most cases, we endeavor to prevent direct sun from entering the space, and in the morning hours this is true for our model, but otherwise it is necessary to use the diffuse interreflection calculation to correctly account for all contributions.

5. Discomfort Glare

Especially in scenes containing daylight, situations arise where the occupant is uncomfortable due to large differences in the brightness of the visual field. For example, trying to look at someone’s face when they are backlit by a bright window is difficult and painful. We would therefore like some way to predict the effects of such bright sources of illumination when designing a space. RADIANCE provides just such a calculation.

Let’s start by generating a fisheye picture of our design space:

% *rpict -vta -vp 1.5 .8 1 -vd 0 1 0 -vh 240 -vv 180 -av .5 .5 .5 test.oct > fish.pic*

This picture will be used to identify sources of glare from a particular view point (1.5, 1, 1) about a particular direction (0, 1, 0). Since RADIANCE pictures contain true floating point radiance values, they can be used to analyze a visual environment for problems such as discomfort glare. Once *rpict* has finished (and this may take the better part of an hour), you may display the image using **ximage** or whatever display program you have available on your system. A fisheye perspective is a type of distortion that allows a wider field of view than a standard perspective image. This larger field is used by the program *findglare* to locate any and all bright spots that might affect visual comfort. **Findglare** can also use direct calculation with **rtrace,** but this takes a long time without providing the nice visual feedback one gets from an image.

To facilitate the use of **findglare** and the associated programs **glarendx** and **xglaresrc,** a script has been written called **glare.** This script asks you some questions and to make it a little easier to get a nice result. To start the script, simply type *glare* on the command line. When it asks you a name for the glare file you want to work with, enter something innocuous like "test.glr". It will tell you that the file does not exist, so we will have to create it. It then prompts you for the name of the picture and the name of the octree, which are "fish.pic" and "test.oct", respectively. When it asks for the view, just hit return because we want to use the default view from the picture file. **Glare** then asks what parameters it should use for the **rtrace** calculation. **(Findglare** will still use **rtrace** to calculate any points that it needs which are off the picture we have given it.) For this, you should give the *–av* setting from before (.5 .5 .5). Then **glare** will ask you if the sources of glare are small, which they are not, and if you would like to set the glare threshold manually, which you would not. Finally, you will be asked what maximum angle you want to calculate away from the view center. Let's use 60 degrees (to the right and left). **Glare** then proceeds to call the program **findglare** to locate the actual glare sources. Once **findglare** has finished, **glare** will call the program **xglaresrc** (if you are running X) to circle the glare sources found on the image.

You then have a choice of a few different glare indices you may calculated from the computed sources. The Guth visual comfort probability (VCP) is perhaps the easiest to understand for those who are not familiar with glare metrics, because it gives a result in terms of a percentage of people who would be satisfied. Calculating the Guth VCP in this case, we see that looking straight ahead (ie. 0 degrees), we see that less than 15% of the people would find this visual environment comfortable. Although the visual comfort probability increases as we look to the left (positive angles), it never gets over 20% for the range of angles we've given. That is primarily because the window is unshielded and very near the horizontal line of sight. Skylights and ceiling fixtures are a little nicer from a glare standpoint for that reason.