MSc in Energy, Architecture and Sustainability **RADIANCE Course**

Axel Jacobs (a.jacobs@unl.ac.uk),
John Solomon (j.solomon@unl.ac.uk)*

26th August 2002

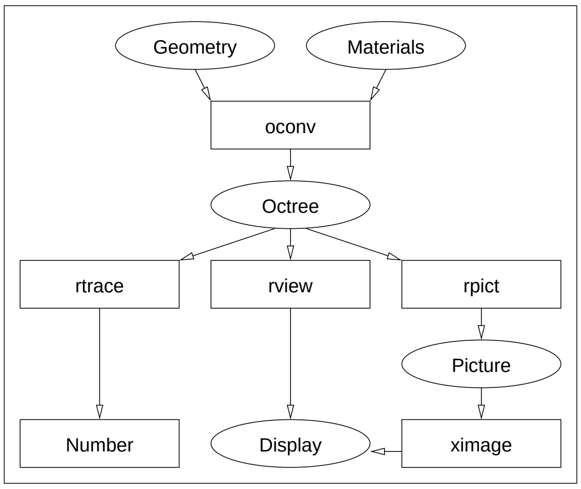
*LEARN, Low Energy Architecture Research uNit, University of North London, http://www.unl.ac.uk/LEARN

Contents

1 Introduction

1.1 What is RADIANCE?

RADIANCE is a highly sophisticated lighting visualisation system. Originally started off as a research project at the Lawrence Berkeley Laboratories, it has evolved into an extremely powerful package that is capable of producing physically correct results and images that are indistinguishable from real photographs.



Its versatility make RADIANCE the ideal choice not only for 'serious' researchers but also for architects, lighting designers and other professionals. Although a challenge to learn, RADIANCE, especially in a UNIX environment, is capable of producing results that no other visualisation package can achieve.[7]

1.2 Raytracing vs. Radiosity

RADIANCE is employs backward ray-tracing algorithms. This means that the light 'rays' are traced back from the point of measurement or view to the light source. There are a number of other ray-tracers on the market because the basic principle is relatively simple to implement on computers. However, where RADIANCE stands out is its ability to handle diffuse interreflections between objects. Very efficient algorithms together with caching are applied for this. Other packages usually try to equate for indirect contributions by defining the 'ambient' light that has no real light source and is instead everywhere. Examples for other ray-tracers include POV or 3DStudio.

Because the calculations are started from the view point, an entirely new calculation has to be done for each individual view. Walk-throughs and videos are therefore extremely resource-hungry requiring fast computers and a lot of time.

There is another conceptually different approach to compute light distributions. This method is called radiosity. Radiosity-based algorithms start off with the energy that is radiated from the light source. Assuming diffuse reflectance properties of the objects, the incoming energy is then modified by the material's reflective properties and bounced back into the room. This is done until the contribution of the reflected light towards the average illuminance in the scene becomes insignificant.

The energy distribution of the entire scene is calculated and stored. This means that once all the calculations are done, new view points can be created in no time at all. This makes the radiosity solutions ideal for the creation of virtual worlds such as VRML. Scenes created this way can usually be told because of their lack of reflective and transparent surfaces, although newer software implements get-arounds to these problems. A typical example of a simulation package that uses radiosity is Lightscape, now owned by Autodesk.

2 RADIANCE and UNIX

2.1 Do it the UNIX Way

The RADIANCE source code is freely available for download from the Internet. RADIANCE was developed to run on UNIX machines. Part of the reason is that in the early 1990s when Greg Ward started writing RADIANCE, computers were very slow compared to today's machines. It did not make sense to run processing intensive applications such as raytracers on desktop PCs. Most 'serious' workstations, however, operated under UNIX which provides a multitasking environment.

It is the UNIX philosopy to have very modular software. This is in stark contrast to the concept that MS Windows and the Mac OS follow. They aim to provide GUI based software packages that do everything the avearge user could possibly ask for and a lot more. The drawback here is that the software can only do what its designers had in mind when they wrote it.

RADIANCE in contrast consists of more than 100 individual programs. This makes it extremely flexible. By defining options and combining two or more programs a maximum flexibility can be achieved. Unfortunately, it also means that a steep learning curve is the price that has to be paid.[2]

2.2 Introduction to UNIX

2.2.1 Shells and Processes

We are going to use RADIANCE the hard way in our course. We are going to completely abandon the world of windows with all the nice buttons and GUIs that would only allow us to do what somebody decided we should be doing.

In order to type in a command, we need a text interface to the OS known as shell. This is something old-time users of DOS will be familiar with, although there it is called a DOS prompt.

A shell is basically a command interpreter. When we type a command, it executes it for us and returns the result. Shells also provide a convenient environment that allows us to work more efficiently. For instance, the BASH shell, which is what we are using, allows us to browse through the command history and re-run a command by simply hitting the up-arrow. Shells can also be programmed. This is called shell scripting.

On a mulituser, multitask platform, many processes run at the same time. The `ps` command shows us the processes that we are running.

```
[student]$ ps
  PID TTY          TIME CMD
  710 pts/0    00:00:00 bash
  779 pts/0    00:00:09 gedit
  787 pts/0    00:00:00 ps
```

However, this is only the so-called foreground processes. Buy appending an apostroph ('&') to the command, we can also run programs in the background. This is what we do when the process takes a long time to complete. No longer can we easily control such processes (for instance, we can't hit ^C to terminate it), but it doesn't block our command line either.

```
[student]$ ps x
  PID TTY      STAT   TIME COMMAND
  667 tty1     S      0:00 -bash
  689 tty1     S      0:00 sh /usr/X11R6/bin/startx
  696 tty1     S      0:00 xinit /home/axel/.xinitrc -- :0
  701 tty1     S      0:01 icewm
  703 ?        S      0:01 /usr/bin/gnome-terminal --use-factory
  705 ?        S      0:00 esd -terminate -nobeeps -as 2 -spawnpid 703
  707 ?        SW     0:00 [gnome-name-serv]
  709 ?        S      0:00 gnome-pty-helper
  710 pts/0    S      0:00 bash
  779 pts/0    S      0:09 gedit
  788 pts/0    R      0:00 ps x
```

The x switch of the ps command shows all processes that are owned by us, including background ones. Note that ps under Linux is one of the few programs that don't use hyphens ('-') with its options. Every process on a UNIX system has a unique process id. To terminate the process, type kill, followed by the pid:

```
[student]$ kill <pid>
```

2.2.2 man pages

Most programs on our system can be called with a whole bunch of different options. This is also true for the RADIANCE commands. Because no-one can remember all the programs with their options, all programs come with the so-called man pages. Man pages are stored on the system and can be used through the man command.

```
[student]$ man kill
KILL(1)            Linux Programmer's Manual             KILL(1)

NAME
       kill - terminate a process

SYNOPSIS
       kill [ -s signal | -p ] [ -a ] pid ...
       kill -l [ signal ]

DESCRIPTION
       kill  sends the specified signal to the specified process.
       If no signal is specified, the TERM signal is  sent.   The
       TERM  signal  will  kill processes which do not catch this
       signal.  For other processes, if may be necessary  to  use
       the KILL (9) signal, since this signal cannot be caught.

OPTIONS
       pid ...
              Specify the list of processes that kill should sigŋ
              nal. Each pid can be one of four things...
```

You should always bring up the man page if you are unclear about what exactly the command does and to learn about options and syntax.[1]

2.2.3 Switches, Pipes, STDIN and STDOUT

The ls command gives a directory listing. It is very similar to the DOS dir command.

```
[student]$ ls
box.rad  chair.rad  msc.mat  msc.oct  msc.rif  nice.vf
```

To find out more about the files in the current directory, the -l switch to ls will give additional information, such as the permissions, ownership, file size and the date and time of its last modification.

```
[student]$ ls -l
-rw-r--r--    1 student     student          1624 Oct 24  1999 box.rad
-rw-r--r--    1 student     student           566 Oct 24  1999 chair.rad
-rw-r--r--    1 student     student           321 Nov  3  1999 msc.mat
-rw-rw-r--    1 student     student         13091 Nov  3  1999 msc.oct
-rw-r--r--    1 student     student           543 Oct 24  1999 msc.rif
-rw-r--r--    1 student     student            82 Oct 24  1999 nice.vf
```

Command line options are almost always preceded by a hyphen. Most commands operate on a file or as in this case a directory. This is given as an argument on the command line. Sometimes this can be an input that is required or optional.

The directory */usr/local/bin* is where the RADIANCE executables are stored on our system, amongst many others. To list them all, type:

```
[student]$ ls -l /usr/local/bin
```

There are quite a lot of programs in this directory, far too many to display on the screen. To display them one screen a time pipe the output of the ls command into more. Piping means that the output of the first command becomes the input for the second one. The vertical bar ('|') is called the pipe symbol. Unless they are redirected, `STDIN` (the standard input) is taken from the keyboard, whereas `STDOUT` (the standard output) is output onto the screen.

More takes whatever is passed to its `STDIN` and pages through it.

To bring up the next page hit the space bar, to scroll down one line hit the ENTER key.

```
[student]$ ls -l /usr/bin | more
...
-rwxr-xr-x    1 root     root       207405 Jan 31  2000 rpict
-rwxr-xr-x    1 root     root        36190 Jan 31  2000 rpiece
-rwxr-xr-x    1 root     root       202125 Jan 31  2000 rtrace
-rwxr-xr-x    1 root     root       229305 Jan 31  2000 rview
-rwxr-xr-x    1 root     root        17347 Jan 31  2000 t4014
-rwxr-xr-x    1 root     root         8927 Jan 31  2000 tabfunc
-rwxr-xr-x    1 root     root         7444 Jan 31  2000 thf2rad
-rwxr-xr-x    1 root     root        13463 Jan 31  2000 tmesh2rad
-rwxr-xr-x    1 root     root         7122 Jan 31  2000 total
-rwxr-xr-x    1 root     root         4677 Jan 31  2000 trad
-rwxr-xr-x    1 root     root        15089 Jan 31  2000 ttyimage
...
```

So we can now look at all the files. How many of them is there, we wonder? A handy little program, `wc` (as in 'word count') will tell us. It displays the number of lines, words, and characters of a given input. The `-l` switch to `wc` makes it produce only the number of lines.

```
[student]$ ls -l /usr/local/bin | wc -l
    116
```

To preserve the directory listing, we can redirect the output of `ls` from the `STDOUT` to a file. We'll call it *ls.txt*. If *ls.txt* doesn't exist, it will be created for us. But be aware–if it does exists and contains data, it will be overwritten and the data will be lost.

```
[student]$ ls -l /usr/local/bin > ls.txt
```

To display the file one screenful at a time, `more` is used once more. This time, we take its input from a file rather than from `STDIN`.

```
[student]$ more < ls.txt
```

Let's look for all commands that have 'gen' in their name. The `grep` command can do this for us and display the result on the screen. To preserve it we might type:

```
[student]$ more < ls.txt | grep gen > gen.txt
[student]$ cat gen.txt
-rwxr-xr-x    1 root     root         7320 Jan 31  2000 genblinds
-rwxr-xr-x    1 root     root         6852 Jan 31  2000 genbox
-rwxr-xr-x    1 root     root         7664 Jan 31  2000 genclock
-rwxr-xr-x    1 root     root        10984 Jan 31  2000 genprism
-rwxr-xr-x    1 root     root        25322 Jan 31  2000 genrev
-rwxr-xr-x    1 root     root        12521 Jan 31  2000 gensky
-rwxr-xr-x    1 root     root        32910 Jan 31  2000 gensurf
-rwxr-xr-x    1 root     root        26181 Jan 31  2000 genworm
```

The `echo` command takes whatever it finds on its command line and displays it. Not very useful unless we redirect the output and do something with it. How about appending it to our *gen.txt* file?

```
[student]$ echo "Hello World" >> gen.txt
```

The old listing of the search results in */usr/local/bin* is still there, we added a new line with 'Hello World' to it. To check whether this is correct, use `cat` to display it. Like `more`, it can display files but it is more useful to join files together.

```
[student]$ cat gen.txt
-rwxr-xr-x    1 root     root         7320 Jan 31  2000 genblinds
-rwxr-xr-x    1 root     root         6852 Jan 31  2000 genbox
-rwxr-xr-x    1 root     root         7664 Jan 31  2000 genclock
-rwxr-xr-x    1 root     root        10984 Jan 31  2000 genprism
-rwxr-xr-x    1 root     root        25322 Jan 31  2000 genrev
-rwxr-xr-x    1 root     root        12521 Jan 31  2000 gensky
-rwxr-xr-x    1 root     root        32910 Jan 31  2000 gensurf
-rwxr-xr-x    1 root     root        26181 Jan 31  2000 genworm
Hello World
```

2.2.4 File Structure and Paths

UNIX systems, like most other operating systems, use a tree shaped file structure also known as 'directory tree'. The base is called root and is indicated by a forward slash ('/'). From here, all other directories and subdirectories branch out. The one where the user data is stored is called */home*. Under this directory every user that has an account has their own home directory. If we are logged on as student, our home directory is */home/student*. Only here are we allowed to write and modify and files. At the same time, our files can not be seen or altered by other users.

To find out where you are, type `pwd` (present working directory). To change into another directory, use the `cd` command, followed by the path. Paths can be absolute or relative. Typing

```
[student]$ cd /home
```

will bring you to */home*, no matter where you are now. Now type

```
[student]$ cd student
```

This will bring you back to your home directory. You should realise that the last command only works if you are in */home*. Typing if from / or from anywhere else will tell you there is no directory called *student*.

There is also a quicker way to go home. The tilda ('~') is a short cut to the user's home directory. So typing

```
[student]$ cd ~
```

works just as well, from anywhere in the filestructure. There are another two special files in every directory. One of them is the parent directory, the level above. This is expressed with two dots. So to go up one level, run `cd` with two dots:

```
[student]$ cd ..
```

The other important file is just called '.' and refers to the present working directory. Both, '.' and '..' are not listed by `ls` unless it is called with the `-a` switch. To display the present working directory, type `pwd`.

```
[student]$ pwd
/home/student/msc
```

3 Describing a Scene in RADIANCE

3.1 General Information and Syntax

RADIANCE uses a cartesian (rectiliear) co-ordinate system. All information is stored in ASCII text format, so it can be edited with any text editor. Please refer to appendix A.1 for some commonly used file name extensions.

There are 4 basic types of primitives:

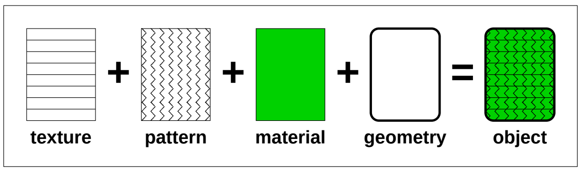
1. Surface primitives
2. Materials
3. Textures
4. Patterns

The syntax for any primitive follows this scheme:

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

The `type` has to be one of the predefined surface, material, pattern or texture types that are known to RADIANCE. The identifier can be freely chosen but should be unique within your project.

Before a surface primitive can be used, a material primitive must exists whose identifier is the same as the modifier of the surface primitive. By chaining several material and texture/pattern descriptions together, very detailed and realistic materials can be defined.



The first primitive in this chain has no modifier, so 'void' is used instead. The identifier of the first primitive becomes the modfier of the second one and so on, e.g.

```
void brightfunc dusty > dusty texfunc woody > woody plastic chairmat > chairmat cylinder leg1
```

The second line in every primitive contains all string arguments that are needed to describe the primitive. The very first character ('n') is an indicator for the number of string arguments to follow.

The third line must always read `0`. It was intended to use this line for integer arguments when new primitives are introduced into RADIANCE but until now no primitive uses integer arguments.

The last line holds real arguments or floating point numbers. Again, the integer in front ('m') indicates the total number of arguments to follow.

Comments are preceeded by a hash sign ('#') and proceed to the end of the line. If the first character in a line is an exclamation mark ('!'), then this is taken as a shell command. The line is executed and the result of this command returned.

3.2 Describing the Geometry

3.2.1 Approaches to Modelling

There are three different approaches to modelling in RADIANCE:

1. The import of CAD models
   - Through the DXF format
   - DesignWorkshop exports native RADIANCE format in its professional version
   - The TORAD interface in AutoCAD 12
   - Translators for ArchiCAD etc.
2. The use of proprietary software that uses RADIANCE as its rendering engine
   - AutoCAD 14 with SiVIEW
   - AutoCAD 14 or 2000 with DesktopRADIANCE
   - Adeline, the DOS version of RADIANCE
   - IES <Virtual Environment>
   - Candle, University College London
3. The construction of the scene with RADIANCE tools
   - Manual input of the co-ordinates, using 10 internal object primitives
   - Powerful generators

```
[student]$ ls /usr/local/bin |grep ^gen
genblinds
genbox
genclock
genprism
genrev
gensky
gensurf
genworm
```

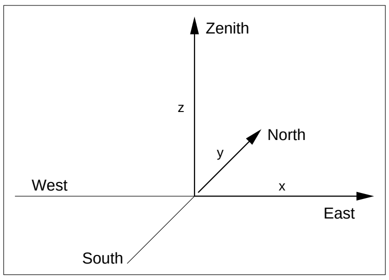
If you decide to use a CAD package such as AutoCAD, you should be aware that you can describe objects such as boxes in three different ways:

- By describing the 12 edges of the box. This is called a wire frame model.
- By modelling the 6 surfaces if the box. This is a surface model.
- By defining the volume of the box. For this, AutoCAD offers extensions such as ACE.

Only the second category models are suitable for use in RADIANCE. If you find a clever translator that can convert volumes into surface, volume models could potentially be used. This is technically possible because the information is there. Wire frame model, however, do not contain any information about the faces and are therefore useless to us.

3.2.2 Modelling Geometry

A RADIANCE scene should always be aligned so that the x-axis points North, the y-axis points East, and the z-axis points upwards to the zenith. This is in contrast to some 3D modelling packages which use x and y for the horizontal and vertical dimensions and describe the depth. i.e. the distance behind or in front of the computer screen with the z co-ordinate.



Sizes and distances can be given in any unit of length, as long as they are used consistantly.

When flicking through the RADIANCE user manual, you will find that many surface primitives come in two flavors. An example is `sphere` and `bubble`. Both describe a ball shaped object. The difference between the two is that a `sphere` has a surface normal that point outwards, whereas the normal of a `bubble` points inwards. As long as the `-bv+` switch to `rpict` is set to turn on back face visibility, this doesn't really matter for most materials. It does matter, however, for light sources and mirrors.

The surface normal follows the right-hand rule. Form a loose fist but have your thumb stick up. Hold your hand in such a way that the axis that is defined by your thumb is perpendicular to the plane of the polygon. Now turn your hand around the thumb-axis following the direction given by the other four fingers. If the indices follow the same direction, the surface normal of the polygon is pointing into the direction of your thumb. Otherwise, it's the opposite.

To get a rough idea about the dimensions and position of an object, use the `getbbox` command. It returns the minimum and maximum of an enclosing box along the x,y, and z axis.

```
[student]$ getbbox chair.rad
     xmin      xmax      ymin      ymax      zmin      zmax
        0       0.5         0       0.5         0         1
```

Now create a new file using `touch` and call it *things.rad*.

```
[student]$ touch things.rad
```

Use your favorite text editor to create the description of a sphere in this new file. The general syntax is:

```
modifier sphere identifier
0
0
4  xcent ycent zcent radius
```

Look at you first RADIANCE object with the `objline` command. It creates what is known as a meta file which can not be view directly. To display it on the screen, simply pipe it into `x11meta`.

```
[student]$ objline things.rad | x11meta
```

Click anywhere on the picture to quit. Also, see what `getbbox` returns when called with *things.rad* now.

Next, use the `genbox` command to create a box. Append it to *things.rad*. The syntax for `genbox` is:

```
genbox material name xsize ysize zsize
```

Open *things.rad* in a text editor and remove two or three adjacent faces. Look at the result with `objline`.

Create another box of different dimensions. This time, don't call it from the command line. Instead, put an extra line in *things.rad* that calls the generator. Remember to begin the line with '!'.

We have just explored two different ways of calling generators:

The first one creates quite large and cumbersome files but allows us to make modifications to individual parts of the object. Unfortunately, this is what most converters from CAD packages will produce. A perfect cylinder, for instance, which is very simple to model using a native RADIANCE primitive, will be split up into a number of polygons.The result will not look not very nice, unless a very large number of polygons is created. However, this will result in large file sizes.

The second way is nice because it is only one line of text that we can easily understand. To change the size of the box only requires the alteration of one argument, compared to 12 co-ordinates done the other way. The drawback of this method is that only the whole object can be modified, not just parts of it.

For what we've done so far, no material definitions were required. This is going to change.

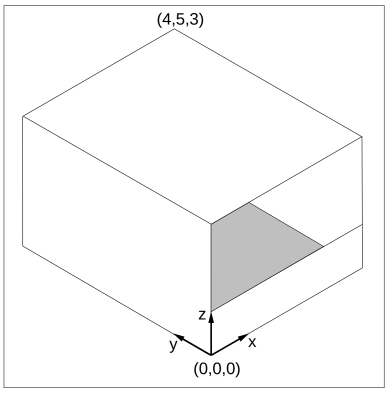
3.2.3 More Complex Scenes

The next exercise will produce something a bit more useful. We are going to build a simple room with a window opening in it. The room will then be used in combination with daylighting.

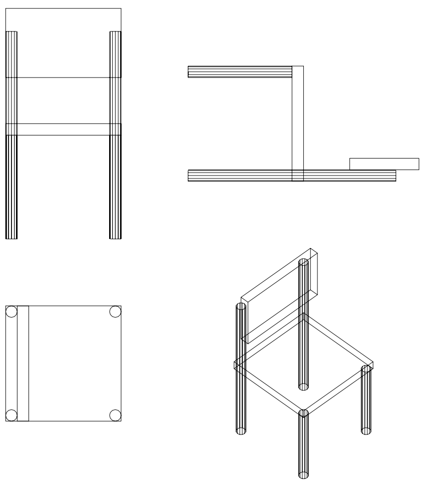
1. Create a room that is 4.0m wide (x-dimension), 5.0m deep and 3.0m high. Give it the material wall_mat. Call the generator from the command line and direct the output of the command into a file *room.rad*.

2. Change the material of the polygons that form the floor and the ceiling to floor_mat and ceiling_mat respectively. The materials wall_mat, floor_mat, and ceiling_mat are already defined in *scene.mat*. You'll find the listings of most files used in our exercises in the appendix of this document.

3. Lower the South facing wall so the height of the window sill is 1.0m.



4. Create a file called *furniture.rad* from which you call `xform` to place a table in the scene and a couple of chairs around it. You'll find them in *table.rad* and *chair.rad*, respectively. Use `getbbox` and `objline` to see whether you get the desired result.



5. Create a file *bulb.rad* which describes the geometry and the material for the light bulb. Place the sphere at the origin of the co-ordinate system and give it a radius of 0.03m. For now, we assume a white light source. Use the material `light` and give it equal values for the red, green, and blue radiance. Please refer to section 3.3.3 for more information on coloured lamps.
6. In a file called *lights.rad*, place two of your bulbs at a height between 2.5 and 3.0m.

Before we catch a first glimps of the image, the scene needs to be compiled into an octree. The purpose of an octree is to speed up the calculation by only considering the objects that lay within the path of a ray. The command to use is `oconv`. It takes as arguments all material and scene files that we want to include. `oconv` will not compile a scene file unless all materials that are used are defined. To make things easier, the file *scene.mat* contains everything that is needed in this exercise. The materials have to be given first, or `oconv` will drop out with an error. The octree is produced at `STDOUT`, so you will need to redirect it into a file.

```
[student]$ oconv course.mat room.rad furniture.rad lights.rad > scene.oct
```

There are three commands that accept an octree as an input and trace rays within the scene. They all start with 'r': `rview` for in interactive preview, `rpict` for producing an image, and `rtrace` to trace a single ray. Use `rview` for an interactive view of the scene pulling the parameters from the view file *nice.vf*.

```
[student]$ rview -vf nice.vf scene.oct
```

Once the `rview` window is up, there are a number of interactive commands to adjust parameters and view points. You will probably have to adjust the exposure of the image before you can see all the details correctly. The most commonly used commands are listed below:

Command	Explanation
`aim`	Zoom in.
`exposure`	Set the exposure.
`frame`	Set frame for refinement.
`last`	Restore the previous view.
`L`	Load parameters from file.
`pivot angle`	Pivot view about selected point.
`quit`	Quit.
`rotate angle`	Rotate the camera.
`set`	Change program variable.
`trace`	Trace a ray.
`view`	Change view parameters.
`V`	Append current view to file.
`write`	Write picture file.

Please look up the exact syntax and more detailed explanation in the `rview` man page.

3.3 Describing the Materials

3.3.1 Standard Materials

A total number of 25 different materials are available to describe the characteristics of the surfaces. They range from the simple to use plastic or metal to the more complex ones like dielectric and BRTDfunc which allows for the most accurate (and difficult) use, but has settings for all directional aspects of reflectance and transmittance. Please refer to [?] for a complete list as well as detailed descriptions.

The material used for the majority of cases is plastic which defines a surface that does not alter the colour of the highlights, i.e. highlights appear in the colour of the light source rather than the colour of the material. This is true for most materials around us, be it wood, paper, concrete, plastic or fabric.

This is all the arguments the plastic primitive expects:

```
modifier plastic identifier
0
0
5 redrefl greenrefl bluerefl spec rough
```

All values must be within the range of [0...1] while most materials in reality have a roughness and specularity below 0.2. The contribution of the individual RGB components towards the average reflectance is weighted and equates to:

$$\rho = 0.265R + 0.670G + 0.065B \tag{1}$$

Don't get confused when reading through old documentation. You might find differing values there. As of version 2.5 RADIANCE uses the multipliers found in equation 1.

Use your favorite image manipulation package to pick a nice colour and apply it to the seat and back of our office chair. The range of colours in most graphics packages is between 0 and 255. So you'll need to scale this down to a range between 0 and 1. What is the reflectance of the fabric?

3.3.2 Materials Modified by Patterns and Textures

While defining the materials for plain colours is a straightforward process, the scene will look more interesting in real when we apply patterns and textures to the object. Patterns describe changes in colour while textures refer to perturbations of the surface normal. A picture mapped onto a frame hanging on the wall is an example for a pattern. Ripples on a body of water, on the other hand, can be created through textures.

We can string as many patterns and textures after one another as we like, the sky is the limit. How about a dusty wooden table with stains and a chess board laid in and some scribble on it? A combination of brightfunc, colorpic and texfunc will do the trick.

Course.mat has already defined two brightfunc and one colorfunc primitive for you. Create a nice blue stripe across the walls by applying blue_band to wall_mat. Additionally, use floorpat to put some random tiles on the floor and make them look dirty with dirt. Try to understand what each function does.

3.3.3 Light Sources

The available materials for light sources in RADIANCE are: light, illum, glow and spotlight.

Light is the basic material for self-luminous surfaces. It is used for most light sources. Illum is used for secondary light sources with broad distribution, i.e. windows. Illum sources are treated like ordinary light sources except when looked at directly. They then act as if they were made of a different material. Glow is for self-luminous surfaces that have a limited effect. Spotlight is used for light sources with a directed output.

For physically correct results, it is important to determine the correct radiance values for the red, green, and blue channel. The `lampcolor` program does this for us. We use the type incandescent here. A list of available lamp types can be found in *lamp.tab*. On our system, this file resides under */usr/local/lib/ray*. A normal tungsten lamp has a luminous efficacy of around 15lm/W. We assume a 100W lamp.

```
[student]$ lampcolor
Program to compute lamp radiance. Enter '?' for help.
Enter lamp type [WHITE]: incandescent
Enter length unit [meter]: meter
Enter lamp geometry [polygon]: sphere
Sphere radius [1]: .03
```

```
Enter total lamp lumens [0]: 1500
Lamp color (RGB) = 350.197270 190.536992 54.738765
^C
```

These values need to be given as real arguments to the material primitive defining the material of the bulb. Use the material light.

Although it is possible to model light sources with light of different colours, this is only necessary if the scene contains light sources of different types. If it is lit by only one kind of lamp a thing called colour adaptation that is common to human vision will result. In such a case, even if we deal with a coloured light source, white objects in the scene appear perfectly white to the eye. Moreover, if an image that was modelled with, let's say tungsten light, and it is looked at under daylight, is will appear much more reddish than it would were we in the actual room.

Care should therefore be taken not to overdo the effects of coloured light sources.

3.3.4 Daylight

Descriptions of daylight are generated with the `gensky` command. It produces two objects of type source, one for the sky hemisphere and one for the ground. Source objects are infinitely far away from any observer.

To start with, let's create a sunny sky for London at today's date. You should redirect the output into a file called *sky.mat*.

```
[student]$ gensky 12 09 14 -a 51 -o 0 -m 0
# gensky 12 09 14 -a 51 -o 0 -m 0
# Local solar time: 14.14
# Solar altitude and azimuth: 11.0 29.9
# Ground ambient level: 8.7

void light solar
0
0
3 2.72e+06 2.72e+06 2.72e+06

solar source sun
0
0
4 -0.489041 -0.851114 0.190903 0.5

void brightfunc skyfunc
2 skybr skybright.cal
0
7 1 3.76e+00 3.72e+00 2.98e-01 -0.489041 -0.851114 0.190903
```

`Gensky` will only create the distribution of sky and ground as well as the material definition and the actual object for the sun. Materials for sky and ground and the two hemispheres are left to the user. When defining the material properties, care must be taken to use skyfunc as modifier to the material for both, the sky and the ground. This is already done in the file *sky.rad*. The photometric average of the radiances according to equation 2 must be equal to 1.0, otherwise the light levels will not be correct.

$$1 = 0.265R + 0.670G + 0.065B \tag{2}$$

Now bring up the result in `rview`. Set up a nice fisheye view and save the parameters into *fish.vf*.

```
[student]$ oconv sky.mat sky.rad > sky.oct
[student]$ rview sky.oct
```

The standard CIE overcast sky produces a horizontal illumiance that can be derived from the irradiance at the zenith with the following formula:

$$R_{zenith} = \frac{9}{7}\frac{E_{horiz}}{179\pi} \tag{3}$$

Working with this formula, create a 10,000lx overcast sky. This is done with the `-b` option to `gensky`. Don't forget to make the sky cloudy with `-c`. Overwrite *sky.mat* with the new output from `gensky`. Re-create the octree again and look at it interactively using the view file from the last exercise.

Now look in the file *sky.mat*. Check the ground ambient level. It should read 17.7. Now take this value and multiply it with π and 179, the luminous efficacy of daylight as used by RADIANCE. Surprised?

The sky and ground must both be made of the material glow. However, glow in contrast to light, spotlight and illum, does not get sampled in the first instance. It will only make indirect contributions. It is therefore sometimes desirable to map the sky distribution onto a window or turn the window into a secondary light source. See section 4.1 for more details.

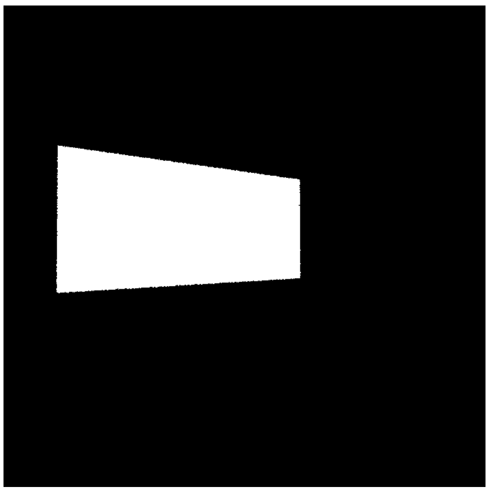
4 How RADIANCE Works

4.1 Ambient Calculations

Compile a new octree and include the following files: *course.mat sky.mat sky.rad* and *room.rad.* Give it the name *scene.oct.* Make sure there is no furniture in the room and that you have an overcast sky in *sky.mat.*

We now view the octree s*cene.oct* with `rview`:

```
[student]$ oconv course.mat sky.mat sky.rad room.rad > scene.oct
[student]$ rview -vf nice.vf scene.oct
rview: warning - no light sources found
```



You will notice that everything inside the room appears black. Using the `trace` command from withing `rview`, check whether this is just a question of poor exposure or if the room is really black.

Check the default settings for `-ab` and `-av` for `rview`.

```
[student]$ rview -defaults |grep -e -a
-av 0.000000 0.000000 0.000000  # ambient value
-aw 0                           # ambient value weight
-ab 0                           # ambient bounces
-aa 0.200000                    # ambient accuracy
-ar 8                           # ambient resolution
-ad 32                          # ambient divisions
-as 0                           # ambient super-samples
```

Both parametres have 0 as default setting. Zero ambient bounces turns the ambient calculation off. So only light sources of type `light`, `spotlight` or `illum` will be sampled. Since the sky is made of `glow`, it does not take part in the direct calculations, resulting in the black interior.

To be able to view the scene, it is sufficient to set an ambient value that is greater than zero, the default. In RADIANCE ambient light is light that is not emitted from a source but instead is assumed to be constant over the whole scene. Remember that in reality, the intensity of the illuminance decreases with the squared distance from the light source.

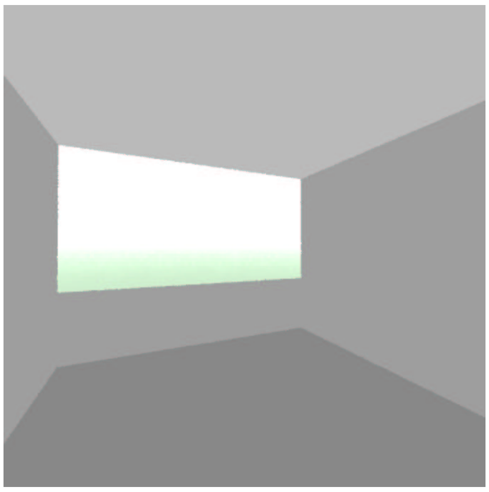
To determine the value of the ambient irrandiance the following formula can be applied. The `-av` option enables us to supply different values for the red, green, and blue channel, however, the three of them will usually be the same.

$$R_{amb} = \frac{E_{amb}}{179\pi} \tag{4}$$

Set `-av` to a value that is equivalent to 500lx and call `rview` again. The `-w` switch will turn off the warning '`no light sources found`'.

```
[student]$ rview -vf nice.vf -w -av .89 .89 .89 scene.oct
```

Different faces of the room can now be distinguished, but the image looks very artificial because all objects are uniformly lit without any shadows.



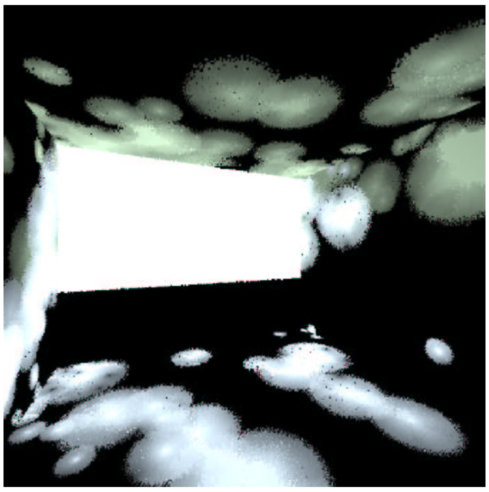
This approach does have advantage, though. For every pixel in the image, only one ray needs to be traced which makes this a 'quick and dirty' solution.

Before you quit `rview`, create a plan view of the entire floor and save it as *floor.vf*. The appropriate view type is 'l' for a parallel view.

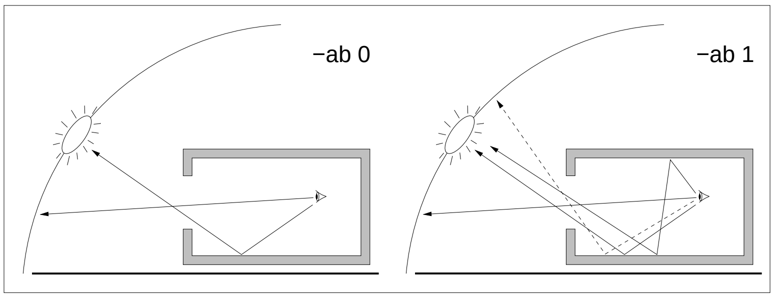
In order to find out how the indirect calculation affects the quality of the rendering, set `-av` back to zero and run `rview` with one ambient bounce. Additionally, set the number of ambient divisions to one with the `-ad 1` option.

```
[student]$ rview -vf floor.vf -av 0 0 0 -w -ab 1 -ad 1 scene.oct
```

This is now quite a strange looking result. The majority of the floor is still black, but there are a number of circular splotches that have a bright centre and fade towards the periphery.



The `-ab 1` option that we used here turns the ambient calculations on. However, only one ambient sample ray is sent off for each position where ambient sampling occurs (`-ad 1` option). So the chances of this ray eventually going through the window and hitting the sky are rather small and decrease even more with the distance from the window.

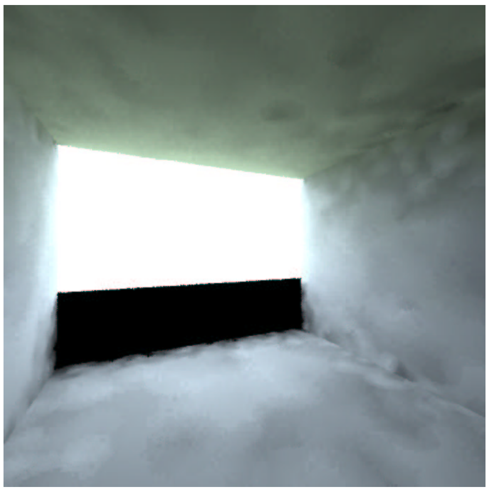


But we also see a cheat that RADIANCE does. In order to reduce its workload, ambient sample rays are not sent out for every pixel. It is assumed that the ambient light does not change a lot throughout the scene, which is usually correct. Every point for which the ambient light did get sampled carries a 'sphere of influence'. As long as a new pixel lies within a radius R of the sphere, a new ambient sampling is not carried out. Instead, the value of adjacent sampling points are interpolated. The radius of the 'sphere of influence' is:

$$R_{min} = \frac{maxSize \cdot aa}{ar} \tag{5}$$

MaxSize is the maximum scene dimension as returned by getbbox, aa and ar refer the settings for `-aa` and `-ar` which control the ambient accuracy and the ambient resolution, respectively.

Increase the ambient divisions to 64 and see what this results in.



In order to make the scene look less patchy, two approaches can be taken:

- Greatly increase the setting for `-ad`
- Get RADIANCE to sample our window as if it was a 'real' light source.

[5] features a table that shows the minimum number of sample rays that are needed to certainly hit a glow source that sustains a certain solid angle.

Here are examples taken from there:-

Angular Resoution (degrees)	Required `-ad`	Required `-ds`
1	33863247	0.02
5	54446	0.09
10	3455	0.17
20	230	0.35
30	50	0.54

It can clearly be seen that as the light source gets smaller, the number of ambient sample rays that is required to hit the glow source explodes. For very small sources, this stochastic sampling becomes too unreliable and computing intensive.

The following section explains the second way–the use of the `mkillum` command.

4.2 Secondary Light Sources

The `mkillum` command takes an object and creates a distribution for it. When the object is looked at directly, it shows up with its real material properties. Since it also carries the characteristics of a light source, test rays are sent out to it every time the illuminance of a point are calculated, without the need to perform ambient calculations.

We are going to use a polygon in the window plane for this purpose. Illum objects don't necessarily have to be real objects that exists in our scene. Virtual planes work just as well, in which case the 'void' modifier should be applied. In our case, we might as well use a glass material for the window.

Fit a window into the opening. Make sure that it covers the entire opening and that the surface normal points into the room. Use the file *window.rad* which already contains the necessary `mkillum` options. The material windowglass is already defined in *course.mat*. Check that the octree *scene.oct* only contains the descriptions of the sky and the room.

To create the distribution for the window, run the following command:

```
[student]$ mkillum -ab 0 < window.rad scene.oct > iwindow.rad
```

A successful run will create two new files: *iwindow.rad* which is a modified version of *window.rad*, modifying the material windowglass with the calculated distribution found in *illum.dat*.

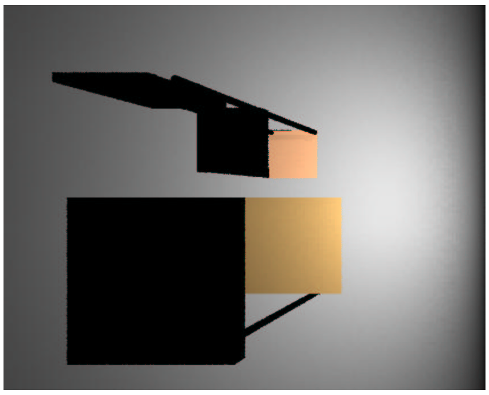
Now compile a new octree that includes the old one and adds furniture and the new illum window to it:

```
[student]$ oconv -i scene.oct furniture.rad iwindow.rad > iscene.oct
```

Look at the result again:

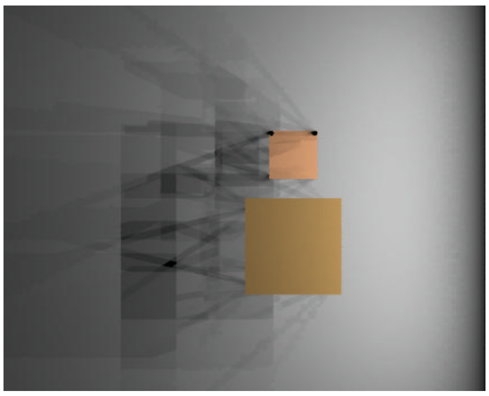
```
[student]$ rview -vf floor.vf -ab 0 -av 0 0 0 iscene.oct
```

This time, the warning message '`no light sources found`' doesn't come up any more. We find that, although the ambient calculation is turned off, the result looks pretty nice. Something is still wrong, though. All the light seems to be emitted from one point in the centre of the window.



A quick check in the default options for `rview` reveals that the `-ds` option is set to zero unless otherwise specified. This variable controls the 'source substructuring'. If set to a value between 0 and 1, large light sources get split up, so there is more than one point that get sampled for shadows.

While still in `rview`, use the `:set` command to give `ds` a value of .3. Instead of one shadow, there are now many, resulting in 'penumbras' (soft shadows). The smaller the value for `-ds`, the more points on large light sources get sampled, resulting in more realistic shadows. But don't overdo it–the time it takes to render an image is directly proportional to the number of light sources.



Render a view of your coice with `rpict`. Set `-ab` to 1 and give `-ad`, `-ds` and `-av` a reasonably value. Call the image *scene.pic*. With exactly the same options, also do an illuminance picture. This requires the `-i` option. Give it the name *scene_ i.pic*. We will need both images in our next exercise.

5 Analysing Scenes

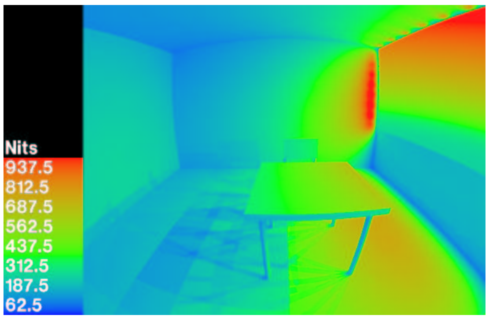
5.1 Analysing RADIANCE pictures

5.1.1 Creating False Colour Images

The `falsecolor` command is a shell script that create impressively looking false colour images. Brightness values are mapped to colours to make it easier for us the determine areas of equal luminance or illuminance. It is possible to use values from one image, let's say an illuminance picture, and overlay them onto another one, such as the corresponding luminance picture.

The following command line will do just this:

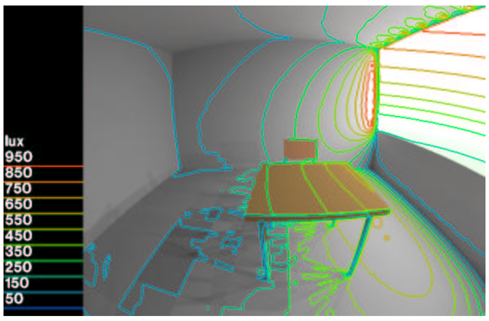
```
[student]$ falsecolor -i scene_i.pic -p scene.pic > false.pic
```



`Falsecolor` also creates legend, explaining the used colours and their corresponding values. The default label is nits, which is the same as cd/m^2.

To make the result just a tiny bit more appealing, we can create it with contour lines instead of a full false colour representation, increase the number of divisions and change the maximum scale of it.

```
[student]$ falsecolor -i scene_i.pic -p scene.pic -cl -n 15 -s 3000 > false.pic
```



5.1.2 Analysis Through `ximage`

If only a handful of values are required from an image or if other value such as pixel position or the ray direction are of interest, the ximage command can be of help. The L key will display the luminance/illuminance value at that point on the screen. If ximage was started from a command line, typing the T key or pressing the middle mouse buttion will print out one or more of the following: Ray origin, ray direction, radiance value, luminance value or pixel position. This output can be controlled with the `-o` option when calling `ximage`. It can then be processed in a spread sheet or dealt with directly with, for instance, the `rcalc` command.

5.2 Analysing Models with `rtrace`

`Rtrace` traces rays given to it on stdin and prodeces the result on `STDOUT`. Like the two other `r*`-commands (`rview` and `rpict`), it does this with an octree.

The rays need to be specified in the following format:

```
        xorg yorig zorig xdir ydir zdir
```

Technically, its possible to create entire images with `rtrace`. Because it is usually only used for a couple of rays, the rendering parametres default to far more accurate settings. Check `rtrace -defaults` to find out more.

5.2.1 Getting an Illuminance Reading

In a first step, we will use `rtrace` to find out the horizontal illuminance that is created by our overcast sky. Since we're only passing the one ray, we do this from the command line rather than through a file. The UNIX `echo` command will do the job–it sends its arguments to `STDOUT`. Make sure the file *sky.oct* only contains the description and the material of the sky.

```
[student]$ echo '0 0 0  0 0 1' | rtrace -I -ab 1 sky.oct
```

The value that is returned is `5.587436e+01`, or 55.87 W/m^2. To get the illuminance, we simply multiply this irradiance with the luminous efficacy of 179 lm/W. The result is 10,000lx. This is no surprise to us, because the sky was created with the `-b` option to produce a horizontal illuminance of 10,000lx.

If we were using a non-gray sky (a blue one for instance), we would first have to calulate the photometric average of the red, green, and blue channel. This could be done this way:

```
[student]$ echo '0 0 0  0 0 1' | rtrace -I -ab 1 -h -w sky.oct \
           | rcalc -e '$1=179*(.265*$1+.670*$2+.065*$3)'
```

The `-h` option turns off the header information, and the `-w` makes sure we don't get the warning that there's no light sources around. Both would confuse the `rcalc` command.

5.2.2 Plotting Illuminance Values

This exercise is based on the last one but uses the `bgraph` command to plot a graph of lux levels in working plane hight against the distance from the window.

The `cnt` command that comes with the RADIANCE distribution can be used to conveniently create a one or moredimensional array of integer values between zero and the number given as an argument. We need to cover the distance between 0.5m and 4.5m in half-metre intervalls. That makes it nine calls to `rtrace`.

```
[student]$ cnt 9
        0
        1
        2
        3
        4
        5
        6
        7
        8
```

Ok, this is nine numbers now from 0 to 8. `rcalc` will convert them for us into co-ordinates. Variables are referred to using a dollar sign ('$'). Variables in front of the equal sign ('=') are input that is passed to `rcalc`, whereas variables behind the equal sign are the output that is produced.

```
[student]$ cnt 9 | rcalc -e '$1=$1/2+.5'
0.5
1
1.5
2
2.5
3
3.5
4
4.5
```

Fine. But what we actually need is vectors, three indices for the origin and three for the direction. That's easily done because everything other than the y-position is constant. If you have furniture in your scene, change `zorig` so it is a bit above the table rather than in exactly the same plane.

```
[student]$ cnt 9 | rcalc -e '$1=2;$2=$1/2+.5;$3=.85;$4=0;$5=0;$6=1
2       0.5     0.85    0       0       1
2       1       0.85    0       0       1
2       1.5     0.85    0       0       1
2       2       0.85    0       0       1
2       2.5     0.85    0       0       1
2       3       0.85    0       0       1
2       3.5     0.85    0       0       1
2       4       0.85    0       0       1
2       4.5     0.85    0       0       1
```

That's it. These nine lines are now piped into `rtrace`. We need to output the co-ordinates as well as the radiance value, because that is needed to plot the graph.

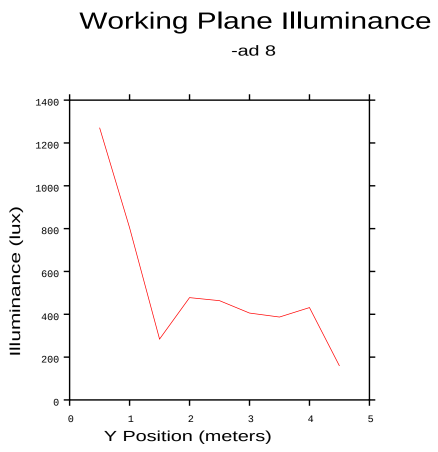
```
2.000000e+00    5.000000e-01    1.850000e+00    6.037447e+00    7.386994e+00    8.523456e+00
2.000000e+00    1.000000e+00    1.850000e+00    3.819175e+00    4.672872e+00    5.391775e+00
2.000000e+00    1.500000e+00    1.850000e+00    1.349180e+00    1.650761e+00    1.904725e+00
2.000000e+00    2.000000e+00    1.850000e+00    2.268448e+00    2.775512e+00    3.202515e+00
2.000000e+00    2.500000e+00    1.850000e+00    2.201090e+00    2.693099e+00    3.107422e+00
2.000000e+00    3.000000e+00    1.850000e+00    1.926298e+00    2.356882e+00    2.719479e+00
2.000000e+00    3.500000e+00    1.850000e+00    1.838468e+00    2.249420e+00    2.595485e+00
2.000000e+00    4.000000e+00    1.850000e+00    2.047739e+00    2.505469e+00    2.890926e+00
2.000000e+00    4.500000e+00    1.850000e+00    7.537872e-01    9.222808e-01    1.064170e+00
```

To get the reading in lux rather than red, green, and blue irradiance values, equation 2 has to help us out once more.

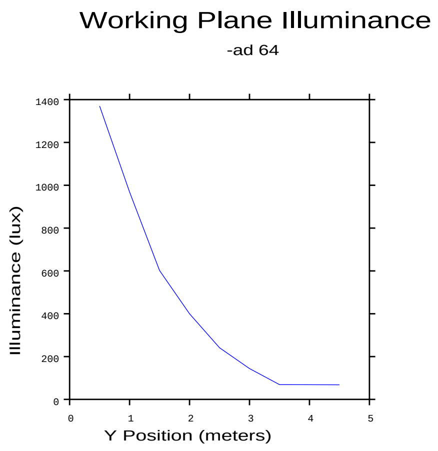
```
[student]$ cnt 9 \
           | rcalc -e '$1=2;$2=$1/2+.5;$3=.85;$4=0;$5=0;$6=1' \
           | rtrace -I -ab 1 -ad 8 -h -w -oov scene.oct \
           | rcalc -e '$1=$2;$2=179*(.265*$4+.670*$5+.065*$6)' > lux.dat
[student]$ cat lux.dat
0.5      1271.4789
1        804.313407
1.5      284.135595
2        477.732247
2.5      463.546922
3        405.675942
3.5      387.179138
4        431.251321
4.5      158.74665
```

Prepared for you is a file named *lux.plt* which gives some instructions to the `bgraph` command. Modify it as you like.

```
[student]$ bgraph lux.plt | x11meta
```



The result will probably not show a perfectly smooth curve. That is due to the undersampling that occurs with the **-ad** set to 8. The default which is 512 produces a much better result. Try again and increase **-ad** so you get a nice curve without pushing it up too much.



Such and similar plots are a good approach to fine-tune the parameters for **rpict** in order to get convincing images and, more importantly, realistic readings. Try, for instance, to plot the working plane illuminance in the back of a room against the number of ambient bounces. You will see that the graph changes dramatically for small numbers but than approaches a final value asymptotically.[7]

6 The Joy of Rendering

6.1 Too Much to Remember

`rpict -defaults` displays a total number of 42 (What's the question again?) options that allow complete control over all aspects of the rendering process. While this allows exerienced RADIANCE users to fine-tune the results, even they sometimes wish to just hit a button and get some result quickly without having to fiddle all those options. This is certainly even more true for the beginner. We already found out that sticking to the defaults hardly ever produces the desired result.

6.2 The `rad` command

But don't despair, help is there! It comes in form of the `rad` command. Once the control file is set up, which doesn't take more than a couple of minutes, a short command line will either bring up an interactive view of the model with `rview` or create a high quality image calling `rpict`. But it does even more than that: By setting only three variables for the overall quality, the importance of indirect calculations and the level of detail in the scene, `rad` automatically takes care of most of the `rpict` options that are vital for getting a good quality image. Copy *template.rif* to *course.rif* and edit it. To call it with `rview`, you need to give it an output device. Type `rview -devices` to get a list. Use `x11` for now.

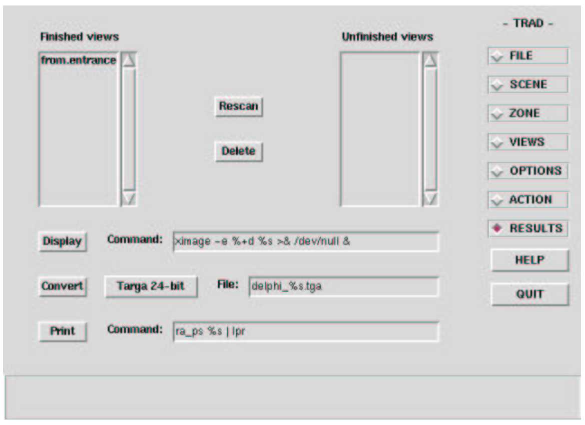
```
[student]$ rad -o x11 -v nice course.rif
        rview -vu 0 0 1 -vf nice.vf -ps 6 -pt .08 -dp 512 -ar 20 -ms 0.1
-ds .3 -dt .1 -dc .5 -dr 1 -sj .7 -st .1 -ab 1 -aa .25 -ad 196 -as 0 -av
0.01 0.01 0.01 -lr 6 -lw .002 -R nice.rif -o x11 nice.oct
```

To run `rpict`, simply drop the `-o x11` option. If you also drop `-v`, one image for each view will be rendered.

6.3 Getting Lazy

As if all this wasn't enough, there is even a graphical user interface (GUI) with RADIANCE. It is called `trad`. `trad` is an X11 (UNIX) front end for the `rad` command using the Tcl/Tk toolkit. It doesn't need any explanation. Just try it out. It comes with a help system that will happily answer all your question.

```
[student]$ trad &
```



Finished views: from.entrance

Unfinished views:

Display Command: xImage -e %+d %s >& /dev/null &

Convert Targa 24-bit File: delphi_%s.tga

Print Command: ra_ps %s | lpr

References

[1] RADIANCE man pages.*

[2] Mike Gancarz: *The UNIX Philosophy.* Digital Press, 19??

[3] Radiance 3.1 Reference Manual*

[4] Radiance Tutorial*

[5] Raphael Compagnon's '97 Daylighting Course on Radiance*

[6] Greg Ward Larson, Rob Shakespeares*: Rendering With Radiance: The Art and Science of Lighting Visualization,* Morgan Kaufmann, 1998

[7] Greg Ward Larson, Rob Shakespeare, John Mardaljevic, Charles Ehrlich: *Rendering with Radiance: A Practical Tool for Global Illuminance*, Siggraph 1998 Course #33, Orlando, Florida*

* On the RADIANCE web site

Websites

- European RADIANCE mirros site:
  http://lesowww.epfl.ch/anglais/radiance/a_radiance.html
- RADIANCE web site:
  http://radsite.lbl.gov/radiance
- LEARN, University of North London
  http://www.unl.ac.uk/LEARN

Appendix

A.2 Suggested File Name Extensions

Under UNIX, it is not necessary to stick to certain file extensions. However, it is highly recommended to always use the same extensions. This will help to stay organised even in directories that contain large numbers of files.

File Type	**Extension**
object description	*.rad*
material definition	*.mat*
octree	*.oct*
view file	*.vf*
project file	*.rif*
image file	*.pic*
rcalc file	*.cal*
ambient file	*.amb*
data table	*.dat*
plot file	*.plt*
bash shell script	*.sh*

A.2 Files Used in the Course

File Name	**Use**
chair.rad	office chair
table.rad	simple table
course.mat	material definitions
sky.rad	definition of sky and ground
lux.plt	plotfile for bgraph
nice.vf	view definitions
template.rif	template for a RADIANCE input file

A.3 File Listings

A.3.1 *chair.rad*

```
# This is a very simple chair.
# Dimensions are in metres.
# It is 0.5m heigh (back 1.0m) and 0.4 x 0.4m wide.
# The corners are at (0,0) and (.5,.5).

void plastic chairmat
0
0
5
        .2 .2 .1  0 0

void plastic fabric
0
0
5
        .8 .4 .2  0 0

!genbox fabric seat .5 .5 .05 |xform -t 0 0 .45

!genbox fabric back .05 .5 .3 |xform -t .05 0 .7

chairmat cylinder leg1
0
0
7
        .025 .025 0  .025 .025 .9 .025

chairmat cylinder leg2
0
0
7
        .475 .025 0  .475 .025 .45  .025


chairmat cylinder leg3
0
0
7
        .025 .475 0  .025 .475 .9  .025

chairmat cylinder leg4
0
0
7
        .475 .475 0  .475 .475 .45  .025
```

A.3.2 *table.rad*

```
# This is a very simple table.
# Dimensions are in metres.
# It is 0.85m heigh and 1x1m wide.
# The corners are at (0,0) and (1,1).

void plastic tablemat
0
0
5
        .5 .3 .1  0 0

!genbox tablemat tabletop 1 1 .05 |xform -t 0 0 .8

tablemat cylinder leg1
0
```

```
0
7
        .1 .1 0  .1 .1 .8 .025

tablemat cylinder leg2
0
0
7
        .9 .1 0  .9 .1 .8 .025


tablemat cylinder leg3
0
0
7
        .1 .9 0  .1 .9 .8 .025

tablemat cylinder leg4
0
0
7
        .9 .9 0  .9 .9 .8 .025
```

A.3.3 *course.mat*

```
void brightfunc dusty
4 dirt dirt.cal -s .1
0
1 .4

void brightfunc floorpat
2 .4*rand(floor(Px/.25)-.25*floor(Py/.25)-.25)+.6 .
0
0

void colorfunc blue_band
4 if(Pz-1.2,1,if(Pz-1,0,1)) if(Pz-1.2,1,if(Pz-1,0,1)) 1 .
0
0

void plastic wall_mat
0
0
5
    .7 .7 .7  0 0

void plastic ceiling_mat
0
0
5
    1 1 1  0 0

void plastic floor_mat
0
0
5
    .5 .5 .5  0 0
```

A.3.4 *sky.rad*

```
skyfunc glow skyglow
0
0
4
        .85 1.04 1.2  0

skyglow source sky
```

```
0
0
4
        0 0 1  180

skyfunc glow groundglow
0
0
4
        .8 1.1 .8  0

groundglow source ground
0
0
4
        0 0 -1  180
```

A.3.5 *lux.plt*

```
include=line.plt
title="Working Plane Illuminance"
subtitle="-ad 8"
xlabel="Y Position (meters)"
ylabel="Illuminance (lux)"
Adata=lux.dat
Acolor="2"
```

A.3.6 *nice.vf*

```
rview -vtv -vp 3 4 1.6 -vd -0.404488 -0.914494 0.00946962 -vu 0 0 1 -vh 90 -vv 90
-vo 0 -va 0 -vs 0 -vl 0
```

A.3.7 *template.rif*

```
DETAIL= Medium
INDIRECT= 1
OCTREE= somefile.oct
PENUMBRAS= False
PICTURE= pickie
QUALITY= Medium
REPORT= 2 course.err
RESOLUTION= 512
UP= Z
VARIABILITY= Low
ZONE= Interior 0 5 0 4 0 3
materials= somefile.mat
scene= somefile.rad
view= nice -vf nice.vf
```