UNIX for Radiance Tutorial

Axel Jacobs

<a DOT jacobs AT londonmet DOT ac DOT uk>

Revision: 12 September 2010

Contents

Revision History

12 Sep 2010

- Intro to BASH scripting.

25 Oct 2008

- Added `tee` command

26 May 2008

- Initial version created from UNIX chapters in Basic and Advanced tutorials.

About the Use of Fonts

Several different fonts are used throughout this document to improve its readability:

`typewriter`: commands, file listings, command lines, console output

italics: paths and file names

sans serif: Radiance primitives, modifiers, identifiers

Suggested Reading

There are a total of four different Radiance study guides and tutorials available with LEARNIX [3]. Figure 1 illustrates the optimal work flow that will give you the best understanding in the shortest amount of time.

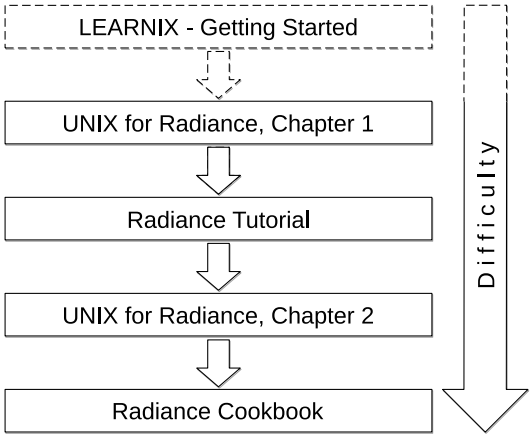

Figure 1: You are strongly encouraged to follow this path while studying the LEARNIX documentation.

You are encouraged to follow this suggestion. If you do feel proficient enough to skip certain sections of a particular document, or even an entire document, you might miss out on some important information that later sections rely upon. Simply skipping over parts that you might find boring or otherwise un-interesting will leave serious gaps in your understanding of Radiance. It is important that you try to understand all exercises, and you absolutely MUST do them yourself. Don't just flick through the pages and look at the pictures.

The LEARNIX Getting Started Guide is, naturally, optional, and only necessary if you use the LEARNIX live CD-ROM, in which case the guide is essential reading.

There are other sources of information, namely:

- The man pages [5],
- The official Radiance web site [6],
- The Radiance mailing list and its archives, available through the Radiance community web site [9],
- The book *Rendering with Radiance* [7].

You may consult them at any time, either while studying with the help of the resources on LEARNIX, or afterwards. Good luck with your efforts.

1 Introduction to UNIX

The successful use of Radiance means to almost completely abandon the world Graphical User Interfaces (GUIs) with all its nice buttons, tick boxes and helpful wizards. This is because any GUIs will only allow us to do what somebody, i.e. the programmer that that software, decided we should be doing, or what he thought we might want to do. While this is fine for software of limited flexibility, it is simply impossible to squeeze Radiance into a window, add some buttons to it and expect to be able to control it this way. Radiance is just too flexible and powerful for this to work.

The Radiance synthetic imaging software is deeply rooted in the UNIX philosophy of software. This means that there are many smaller programs that typically only do a small job, but this they do well. Through building up larger projects by combining those small pieces of software in clever and unforseen ways, we are given ultimate power and control over the end result. Think of assembling a house or automobile from LEGO bricks.[8]

1.1 Shells and Processes

In order to interact with the computer and tell it what to do, we use a text interface to the operating system 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 multiuser, 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. By appending an apostrophe ('&') 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 Ctrl-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. Every process on a UNIX system has a unique process id. To terminate the process, type `kill`, followed by the PID:

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

1.2 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 and switches, all programs come with the so-called man pages. Man pages are stored on the system and can be used through the `man` command. Hit 'q' to quit.

```
[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 sign
              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.[5]

1.3 Command-line Options

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/bin* is where the Radiance executables are stored on our system, amongst many others. To list them all, type:

```
[student]$ ls -l /usr/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. This is explained in the next section. The vertical bar ('|') is called the pipe symbol.

1.4 `STDIN` and `STDOUT`

Generally speaking, all commands take an input and produce some form of output. The input is normally taken from the command-line, while the output is produced on the screen. The normal input and output are also known as `STDIN` and `STDOUT`, resp. A second output, `STDERR`, also exsists but is only responsible for printing error messages. The three command interfaces are shown in fig. 2.

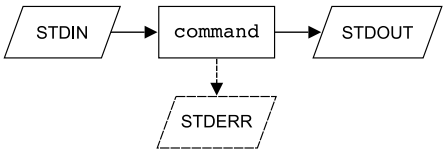

Figure 2: All command have three interfaces: `STDIN`, `STDOUT` and `STDERR`

It is possible to re-direct all three of these interfaces, for instance to write the output to a file, or pass it on to a second command. This concept is illustrated in fig. 3. The `more` command, for example, takes whatever is passed to its `STDIN` and pages through it[1]. This allows us to look at long text files without most of the content scrolling off the screen.

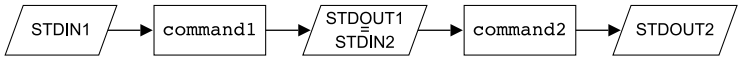

Figure 3: The output of one command may become the input to a second one. `STDERR` omitted for clarity.

Hit the SPACE bar to bring up the next page, hit the ENTER key to scroll down one line, and 'q' to quit.

```
[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 rvu
-rwxr-xr-x    1 root     root        17347 Jan 31  2000 t4014
```

[1] Actually, `less` is an advanced version of `more`, for instance it allows you to scroll up as well as down, unlike `more`.

```
-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/bin | wc -l
    3565
```

To preserve the directory listing, we can redirect the output of `ls` from `STDOUT` to a file. We'll call it *ls.txt*. If *ls.txt* doesn't exist yet, 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/bin > ls.txt
```

To display the file one screen 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 > generators.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 *generators.txt* file?

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

The old listing of the search results in */usr/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 also useful for joining files together.

```
[student]$ cat generators.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
```

1.5 Wild cards

It might be useful at times to list a bunch of files that have something common in their file names. Two so-called wild cards exist for this:

Questionmark ('?') Stands for exactly one character;
Asterisk ('*') Stands for zero or more characters.

Two examples will demonstrate their use: We have already seen that Radiance comes with a number of generators. All of them start with the letters 'gen'. To list all generators under */usr/bin*, we can simply type

```
[student]$ ls /usr/bin/gen*
/usr/bin/genambpos  /usr/bin/genclock   /usr/bin/genrhgrid
/usr/bin/genworm    /usr/bin/genblinds  /usr/bin/genprism
/usr/bin/gensky     /usr/bin/genbox     /usr/bin/genrev
/usr/bin/gensurf
```

If we created backup files of our Radiance scene, *scene.rad*, by adding a number before the dot, e.g. *scene1.rad*, *scene2.rad*, we could list them all with the following command:

```
[student]$ ls scene?.rad
```

Please note that '?' stands for exactly one character, so this will not show the file we're currently working on (*scene.rad*), nor will it show backup files with two-digit number, e.g. *scene12.rad*. For this, the asterisk would be more appropriate.

1.6 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 to 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 file structure. Calling `cd` with no options at all will also take you back to your home directory.

There are 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
```

With this much basic UNIX knowledge under our belts, we are well prepared to dive into the wonderful world of Radiance. One last comment regarding the command-line: You may use a semicolon ';' as a separator for multiple commands on the same line, e.g.

```
[student]$ cd /usr/bin
[student]$ ls
```

is the same as

```
[student]$ cd /usr/bin ; ls
```

which is the same as

```
[student]$ ls /usr/bin
```

(but of course you knew that...)

2 UNIX Textprocessing Commands

Radiance isn't always about pretty pictures. Especially engineers find it very useful to use Radiance for simply churning out numbers, which they can then analyse in a multitude of different ways. The software which immediately springs to mind for fiddling about with numbers would be spread sheet. On a UNIX system, however, we have quite a number of little commands that are designed to deal with text files. Having a good understanding of what those commands are and what they can do will not only save you an enormous amount of time, but will also enable you to combine the different Radiance commands in creative and unexpected ways.

Here is a list of UNIX text processing commands I use most frequently. Don't forget that the full documentation is always available in the relevant `man` page.

2.1 Simple Commands

Here is a very simple test file which we are going to use to see the effect that some of the command have. We'll call it *source.txt*. The columns are tab-separated.

```
0     .629   .988   .142
1     .826   .731   .265
2     .931   .666   .393
```

The commands below are in no particular order. Most of the short descriptions are directly copied from the `man` pages.

tr Translate or delete characters. Very handy for tidying up multiple spaces or tabs in a file that is meant to have it's columns separated by just one tab. A space is denoted as `' '`, and a tab as `'\t'`.

```
[student]$ cat source.txt | tr 9 1
0       .621    .188    .142
1       .826    .731    .265
2       .131    .666    .313
[student]$ cat source.txt |tr -s 6
0       .629    .988    .142
1       .826    .731    .265
2       .931    .6      .393
```

cat Concatenate files and print to standard output. You can always use this as link number one when piping different text processing commands together.

cut Strip columns from files. Assumes that the columns are separated by exactly one tab. If this is a different character, use `-d`, if it is multiples of the same character, use `tr -s` first.

```
[student]$ cat source.txt |cut -f2,4
.629    .142
.826    .265
.931    .393
```

The `cut` command is a bit choosy about the fields separators used in the text file. It expects the columns to be separated by tab characters. If this is not the case, you must tell it which character to use as the separator. The example below shows you how to do this for a space-separated file:

```
[student]$ cat another_source.txt |cut -d' ' -f3
```

head Output the first few lines of files. Default is 10 lines.

```
[student]$ head -2 source.txt
0       .629    .988    .142
1       .826    .731    .265
```

tail Output the last few lines of files. Default is 10 lines.

```
[student]$ tail -2 source.txt
1       .826    .731    .265
2       .931    .666    .393
```

To display the end of the file as it is being created, use the `-f` option (as in 'follow'). This is very useful for log file that just keep getting bigger and bigger. You can interrupt the output at any time by pressing CTRL-C.

echo Display a line of text on `STDOUT`. Often used as the beginning of a piped sequence.

```
[student]$ echo "Hello World"
Hello World
```

wc Print the number of lines, words, and bytes in a file.

```
[student]$ wc source.txt
3 12 51 source.txt
[student]$ cat source.txt |wc
      3      12      51
[student]$ cat source.txt |wc -l
3
```

basename Strip directory and suffix from filenames. This comes in handy for batch-processing or renaming of a large number of files.

```
[student]$ echo `basename source.txt txt`dat
source.dat
```

In the BASH shell, this may also be written as

```
[student]$ echo $(basename source.txt txt)dat
source.dat
```

which is easier to read because the back-ticks can be very small in some fonts. It means that the shell gets the `echo` command to output whatever the command in back-ticks or parenthesis (in this case `basename`) returns. To convert all Radiance **.hdr* files in a given directory into TIFFs, type:

```
[student]$ for f in *.hdr; do ra_tiff $f $(basename $f hdr)tif; done
```

sort Sort lines of text files.

uniq Report or omit repeated lines.

tee Handy little program. If you are redirecting `STDOUT` of a script or program to a file, but wish to also see the output on the screen, you simple pipe it into `tee`.

```
[student]$ sh script.sh | tee results.dat
...
```

A similar result could be achieved by using the normal redirector:

```
[student]$ sh script.sh > results.txt
```

Then, in a different shell, you can use `tail` with its `-f` options:

```
[student]$ tail -f results.txt
```

2.2 More Powerful Commands

(needs more explanation...)

grep Print lines matching a pattern

sed stream editor for filtering and transforming text

3 Shell Scripting

If you do a large number of repetive tasks, such as rendering Radiance images for different views, you might wish there was a way of automating this. Imagine the following situation: You want to run `rpict` for multiple view files, then filter the images and convert them all to PNG format. Here is roughly what you'd have to do:

```
[student]$ oconv materials/boring.mat objects/object1.rad \
    objects/object2.rad skies/overcast.mat skies/sky.rad \
    > scene.oct
[student]$ rpict -ab 3 -ad 1024 -aa 0.13 -vf views/1st.vf \
    scene.oct > tmp/scene_1st.hdr
[student]$ pfilt -e 1 -x /2 -y /2 tmp/scene_1st.hdr \
    > images/scene_1st.hdr
[student]$ convert -gamma 2.2 images/scene_1st.hdr \
    images/scene_1st.png
```

You then need to re-type those commands, substituting all occurences of '*1st*' with '*2nd*', then '*3rd*' and so on. This is where a little BASH script can help you.

3.1 For loops

Copy-and-past the above command lines into a text file and name it *rendering.bash*. Remove all the prompts (this is '`[student]$ `' in our example.) The result should look like this:

```
[student]$ cat rendering.bash
oconv materials/boring.mat objects/object1.rad \
    objects/object2.rad skies/overcast.mat skies/sky.rad \
    > scene.oct
rpict -ab 3 -ad 1024 -aa 0.13 -vf views/1st.vf scene.oct \
    > tmp/tmp.hdr
pfilt -e 1 -x /2 -y /2 tmp/tmp.hdr \
    > images/scene_1st.hdr
rm tmp/tmp.hdr
convert -gamma 2.2 images/scene_1st.hdr images/scene_1st.png
```

You can now run this file like so:

```
[student]$ bash rendering.bash
```

This does exactly the same as the initial four command lines and so far, hasn't really improved anything. To run additional views, we still need to edit this file by copying and changing the lines within. There is, however, a more elegant way, which is shown in the example below:

```
[student]$ cat rendering1.bash
#!/bin/bash
#
# Script to render Radiance images for different skies and views.

# Compile the octree. This is only done once.
oconv materials/boring.mat objects/object1.rad objects/object2.rad \
    skies/overcast.mat skies/sky.rad > scene.oct

# Run the simulation with different view files.
for view in 1st 2nd 3rd 4th; do
    rpict -ab 3 -ad 1024 -aa 0.13 -vf views/$view.vf \
            scene.oct > tmp/tmp.hdr
    pfilt -e 1 -x /2 -y /2 tmp/tmp.hdr > images/scene_$view.hdr
```

```
    rm tmp/tmp.hdr
    convert -gamma 2.2 images/scene_$view.hdr \
            images/scene_$view.png
done
```

The first line is called 'shebang'. It ensures that your system knows it's a BASH script. Given the right file permissions, you can now run the command like so:

```
[student]$ ./rendering1.bash
```

The interesting line is the one starting with '`for view in ...`'. This construct is called a loop. It iterates over all items between the '`in`' and the '`;`'. We defined the variable as '`view`'. Later one, when whenever we type '`$view`', the current value of `view` is inserted instead.

Variables don't need to be used within a loop. You can do anything you like with them. Here is another example. Suppose we want to render the same four views, but for two different skies, overcast and cloudy. We'll also introduce a third variable, `AMB`, which makes the `rpict` line more readable. I tend to use UPPERCASE variable names for constants, and lowercase ones for variables that are changed by the script.

```
[student]$ cat rendering2.bash
#!/bin/bash
#
# Script to render Radiance images for different views.

# Define the ambient parameters for rpict.
AMB="-ab 3 -ad 1024 -aa 0.13"
for sky in overcast clear; do
    # Compile the octree. This is done once for each sky distribution.
    oconv materials/boring.mat objects/object1.rad objects/object2.rad \
        skies/$sky.mat skies/sky.rad > scene_$sky.oct

    # Run the simulation with different view files.
    for view in 1st 2nd 3rd 4th; do
        rpict $AMB -vf views/$view.vf \
                scene_$sky.oct > tmp/tmp.hdr
        pfilt -e 1 -x /2 -y /2 tmp/tmp.hdr images/scene_${sky}_$view.hdr
        rm tmp/tmp.hdr
        convert -gamma 2.2 images/scene_${sky}_$view.hdr \
                images/scene_${sky}_$view.png
    done
done
```

This modified script now contains two loops: one that is responsible for the sky conditions, the other one for different views. Please note that BASH doesn't really care about indentation. It only relies on the correct order of the commands. I would still recommend that you use indentation consistently. This makes it much easier to spot and correct mistakes.

Variable names in BASH are case-sensitive you may use any number of UPPERCASE, lowercase characters, numbers, '-' and '_', but the name must not start with a number. Because the underscore is allowed in variable names, we had to include curly braces in expression like this one: `scene_${sky}_$view.hdr`. Without the braces, BASH would have looked for the variable named '`sky_$view`', not '`sky`', followed by '`_`', followed by '`view`'.

4 Programming Languages

Before you seriously dig into one of those, I suggest you start with BASH shell scripting. It's much easier to pick up, and it will get you a long way before you need to look into a 'proper' programming language. There are tons of references and tutorials on the Internet. The biggest problem with BASH scripts in my opinion is that native BASH algebra can only do integer calculation. This is easy to circumvent with Radiance's `rcalc`, but still somewhat limiting.

Having said that, I quite like `awk` for some simple calculations that Radiance's `rcalc` can't handle, and will be using it every now and then throughout these tutorials. You may notice that the `rcalc` syntac has borrowed quite heavily from `awk`.

There are no references to the other heavy-weights. If you do want to move beyond and want my opinion—I personally think that Python is a more well-conceived language which is easier to read and learn. However, I actually did get stuck with Perl for the web programming I do, simply because the language benefits enormously from the modules that people have contributed to it. Python hasn't quite build up the same community yet, but this is only a matter of time.

Anyhow, everything fancy in this document is done with BASH scripts. So don't get yourself into a Python-or-Perl twist just yet. BASH will get you a long way.

awk Pattern scanning and processing language [4];

perl Practical Extraction and Report Language [1];

python An interpreted, interactive, object-oriented programming language [2].

References

[1] Perl.org. URL http://www.perl.org/.

[2] Python.org. URL http://python.org/.

[3] Axel Jacobs. LEARNIX. Web site, Nov 2009. URL http://luminance.londonmet.ac.uk/learnix.

[4] Diane Barlow Close, Arnold D. Robbins, Paul H. Rubin, Richard Stallman, and Piet van Oostrum. *The awk Manual*, 1995. URL http://www.cs.uu.nl/docs/vakken/st/nawk/nawk_toc.html.

[5] Greg Ward. *Radiance man Pages*, 2008. URL http://radsite.lbl.gov/radiance/man_html/comp.html.

[6] Greg Ward. Official Radiance web site, 2008. URL http://radsite.lbl.gov/radiance.

[7] Greg Ward Larson and Rob Shakespeare. *Rendering with Radiance: The Art and Science of Lighting Visualisation*. Morgan Kaufmann Publishers, San Francisco, 1997.

[8] Mike Gancarz. *The UNIX Philosophy*. Digital Press, 1995.

[9] Radiance Community. Radiance-Online. Web site, Nov 2009. URL http://www.radiance-online.org.

Index