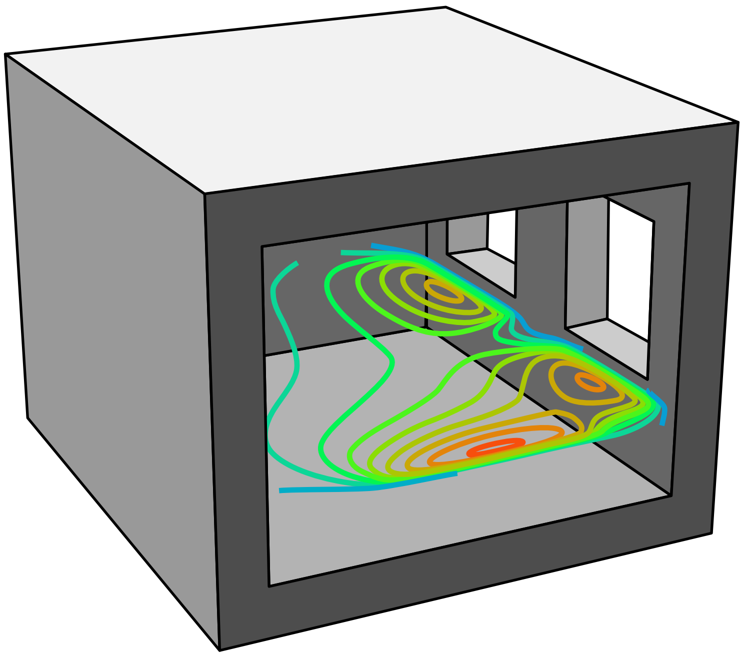
RADIANCE
Cookbook

Axel Jacobs

<axel AT jaloxa DOT eu>



Revision: 10 October 2014

Contents

Revision History

10 Oct 2014

- Fix typo in formula in Section 3.3.2
- Remove DaySim

24 Jan 2010

- Removed Radzilla, `rshow` and `brad` (no longer maintained)
- Updated URLs
- New Radiance 4.1 `falsecolor` palettes in Sec 2.7

2 Apr 2010

- Added a note on the `glass` material, together with a simplified formula;
- Moved "mkillum and Secondary Sources" from the Radiance Tutorial to this document (3.1). I haven't found the time to cover it in about five years, so it might as well be a simple recipe.
- Spell checking and tidying up

7 Oct 2009

- Changed all *.pic* file extensions to *.hdr*
- Removed sections on sky models. It needs rewriting - I found too many mistakes in it.
- Changed all tables to formal ones (only horizontal lines)

26 May 2008

- Moved "UNIX Tips and Tricks" to a separate document: UNIX for Radiance.
- Renamed this document to Cookbook.
- Added chapter on fisheye projections to celebrate the new stereographic projection in Radiance 3.9

17 Feb 2007

- New "Prototyping for `gensurf` with `gnuplot`" (2.4)

26 Dec 2006

- Added index
- Added margin icons for files in the ZIP archive

10 Nov 2006

- Nicer cover page, fancy headings, double-sided layout
- Section on custom cylindrical views (Update: Now called "Panoramic Projection" 4.2)
- Stuck all images into floats and gave them captions.
- Some changes to 2.7 to reflect the new `falsecolor` legend in Radiance 3.8;
- New section 5.4 on plotting brightness along a line.

- Split section "Visualisation and Analysis" into "Visualisation" (4) and "Lighting Analysis" (5).
- Started with a list of some UNIX text processing commands (Update: This is now in a separate document "UNIX for Radiance"). Needs more work.

14 Feb 2006

- added stuff on making a simple plant with `xform` (2.2)

20 April 2005

- added some OpenGL commands: `glrad` (4.5), `rholo` (4.6), `rshow`
- restructured 6: "Add-ons to Radiance"
- re-wrote RADZILLA (well, the section about it)
- finally drew the diagrams and wrote the text explaining the `trans` material (2.5.2)

7 Mar 2005

- Hyperlinked this revision history, so you can quickly jump to the new sections.
- New material on transparent textures in section 2.6.2
- "Vector illuminance" (5.7)
- Re-structured the document. They mayor sections are now "Modelling" (2), "Daylighting" (3), "Visualisation and Analysis" (4) and "Add-ons" (6).
- Added 4 on non-standard views
- Added "Illuminance values on virtual planes" (5.6)

1 Mar 2005

- Added section 3.2 on ground planes
- Increased font size to 11pt.
- New section 2.7 on `pcomb`

19 Dec 2004

- Added revision history
- Added chapter 2: "Advanced Modelling", containing 2.3: "Using of `genworm`" and 2.6: "Image Mapping"

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 number of different Radiance study guides and tutorials available with LEARNIX [9]. Figure 1 illustrates the optimal work flow that will give you the best understanding in the shortest amount of time.

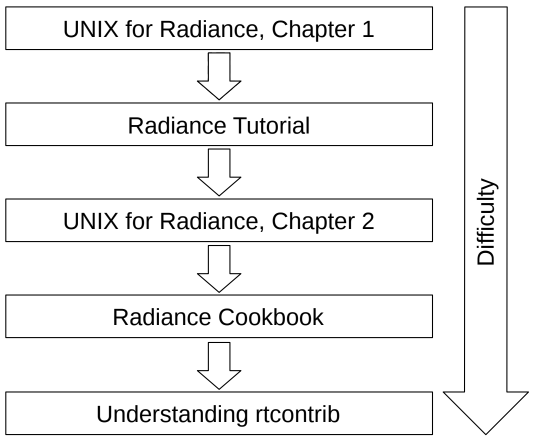


Figure 1: You are strongly encouraged to follow this path while studying the LEARNIX documentation.

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.

There are other sources of information, namely:

- The man pages [17],
- The official Radiance web site [19],
- The Radiance mailing list and its archives, available through the Radiance community web site [28],
- The book *Rendering with Radiance* [22].

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

This document is intended to guide students wishing to master the Radiance lighting simulation software through some more advanced topics. It supplements the Basic Radiance Tutorial [10], and it is assumed that you have read that document and followed all the exercises presented in there. The document is available on the LEARNIX web site [9].

LEARNIX is a modified version of the KNOPPIX live CD-ROM. After booting from the CD-ROM, you will be dropped into a fully functioning and configured GNU/LINUX system with Radiance installed. So if you are having problems with the software, do give LEARNIX a try.

This document is very much work in progress, and it will probably never be finished. I intend to add more material as new add-ons for Radiance come out, and as I explore more of the programs that are already there. The open-source license under which Radiance was released in 2002 has created a lot of activity, and many individuals and institutions are busy writing additional packages which plug into Radiance or enhance its functionality in other ways.

One last remark before we get our hands dirty: Radiance is a complex and powerful beast. It can do so many things that nearly every time I use it in a project, I do something I haven't done before. The price to pay for this is quite a bit of time figuring out how things can be done, and so far I haven't found anything yet that couldn't be done in Radiance. The possibilities are endless. This equally means that there is and never will be an ultimate Guide to Radiance. This document is no exception. What I am mostly trying to do is encourage you to explore for yourself. Chances are that your projects are completely different to what you are going to find in here.

Oh, and please don't be shy of the command-line. Although the way most modern computer operating systems are used these days is that choices and options for a particular task are presented as neat little buttons and tick-boxes, the major disadvantage to this is that you are limited to what the designer of the interface thought you might want to be doing. Radiance, on the other hand, gives you a toolbox with more than 100 tools in it. By using those tools to get parts of the job done, and gluing them together with a few simple shell scripts, you are only limited to what you can do by your imagination and skills, but not by the software itself. It is absolutely **essential** that you study the UNIX for Radiance Tutorial [11] before proceeding any further.

Finally, this tutorial includes material from a number of contributors. Most of them deserve a big thank you not only for their contributions to this tutorial, but also for writing the software those images illustrate, in the first place:

- Greg Ward for the holodeck interactive ray cache (4.6) and of course for Radiance of which he is the original author;
- Francesco Anselmo for `brad`, a Radiance plug-in to the popular Blender 3D modeller;
- Peter Apian-Bennewitz for the `rshow` OpenGL geometry preview;
- Carsten Bauer for `Radzilla`, a re-write of Radiance with much added functionality;
- Roland Schregle for the photon map (6.2.1) forward ray tracing extension;
- Many other that contribute with software, documentation or their advice on the Radiance mailing list.

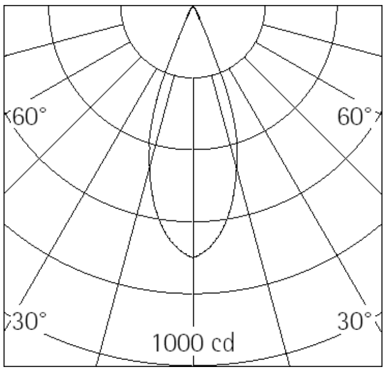


Figure 2: ERCO's Gimbal adjustable spot light and its luminance distribution curve

2 Advanced Modelling

2.1 Using IES Luminaire Data Files

2.1.1 Importing the Luminance Distribution Data

Simulating scenes that are artificially lit will only result in accurate and realistic results if the distribution of the fittings are modelled accurately. There are a number of formats for electronic data files describing the luminance distribution curve of luminaires. One of the more popular formats is defined by the Illuminating Engineering Society of North America, IESNA in their LM-63 file format which was introduced in 1986 under the title "IES Recommended Standard File Format for Electronic Transfer of Photometric Data." It was revised in 1991 and in 1995.

Most mayor luminaire manufacturers provide distribution curves for their products which may be used with Radiance. Fig 2 shows the ERCO Gimbal directional spotlight and a polar plot of the curve [1]. The spotlight may be tilted, but for our example we just assume it points down.

For the lighting designer, additional tables or graphs are usually presented indicating how the illuminance that can be achieved with the fitting drops off with distance. The table is included below for allowing us to check whether the IES import worked correctly.

h [m]	E [lx]	d [m]
1	700	0.69
2	175	1.38
3	78	2.07
4	44	2.75
5	28	3.44

The IES file *ERCO_88100000_1xQR-CBC35_20W_38deg.ies* is included below.

```
IESNA:LM-63-1995
[TEST] 100062_0 BY: ERCO / LUM650
[DATE]04.12.2003
[MANUFAC] ERCO Leuchten GmbH
[LUMCAT] 88100000
```

```
[LUMINAIRE] Gimbal Directional spotlight
[LAMPCAT] QR-CBC35 20W 38
TILT=NONE
1 320 1
37 1
1 2
-.035 0 0
1.00 1.00 20
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100 105 110
115 120 125 130 135 140 145 150 155 160 165 170 175 180 0
700.0 661.1 582.7 467.5 323.2 65.9 28.2 6.7 2.6 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0
```

The first line of the file defines the data format. Header lines starting with tags in square brackets '`[...]`' give additional information about the product and the manufacturer, but are not used by Radiance. There is no upper limit for the number of label lines, and they may also be omitted completely.

The `TILT=NONE` line indicates the end of the label lines. If set to none, the candela values do not vary as a function of the mounting position, which is usually the case. The 13 values which follow have the following meaning:

```
<# lamps> <lumens per lamp> <candela multiplier>
<# vertical angles> <# horizontal angles>
<photometric type> <units>
<width> <length> <height>
<ballast factor> <ballast lamp factor> <input watts>
```

The values for `<# lamps>` and `<lumens per lamp>` are not used by `ies2rad`. Values in the candela table may be normalised to cd/1000 lm, in which case a multiplier needs to be applied to define the lumen output of the fitting. This multiplier is a function of the luminous efficacy of the lamps, their wattage and the number of lamps in the fitting. It may be defined directly in the IES file as `<candela multiplier>`, or at the command line with the `-m` option.

The units may be given in feet (1) or metres (2).

`<width>`, `<length>` and `<height>` do not only specify the actual dimensions of the fixture, but also the shape of the luminous opening:

Shape	`<width>`	`<height>`
rectangular	width	length
circular	- diameter	0
elliptical	- width	length
point	0	0

In addition to the shapes that can be defined within the IES file, spherical sources may be created using the `-i radius` command line option of `ies2rad`.

The rest of the file is occupied by the candela table which holds `<# vertical angles>` times `<# horizontal angles>` entries.

```
$ ies2rad -o erco -dm \
    -t default ERCO_88100000_1xQR-CBC35_20W_38deg.ies
```

The `-dm` switch tells `ies2rad` that the output dimensions should be in metres, while `-t` sets the lamp type. All lamps types known to Radiance are stored in */usr/local/lib/ray/lamp.tab*, together with the chroma coordinates and their lumen depreciation factor.

The `-o` option set the output file name. The program will create the files *erco.rad* and *erco.dat* which are both included below.

```
# ies2rad -o erco -dm -t default
# Dimensions in meters
#<IESNA:LM-63-1995
#<[TEST] 100062_0 BY: ERCO / LUM650
#<[DATE]04.12.2003
#<[MANUFAC] ERCO Leuchten GmbH
#<[LUMCAT] 88100000
#<[LUMINAIRE] Gimbal Directional spotlight
#<[LAMPCAT] QR-CBC35 20W 38
# 20 watt luminaire, lamp*ballast factor = 1

void brightdata erco_dist
4 flatcorr erco.dat source.cal src_theta
0
1 1039.38

erco_dist light erco_light
0
0
3 1 1 1

erco_light ring erco.d
0
0
8 0 0 -0.00025 0 0 -1 0 0.0175

erco_light ring erco.u
0
0
8 0 0 0.00025 0 0 1 0 0.0175
```

Listing of *erco.dat*:

```
1
0 180 37
3.91061 3.6933 3.25531 2.61173
1.80559 0.368156 0.157542 0.0374302
0.0145251 0 0 0
0 0 0 0
```

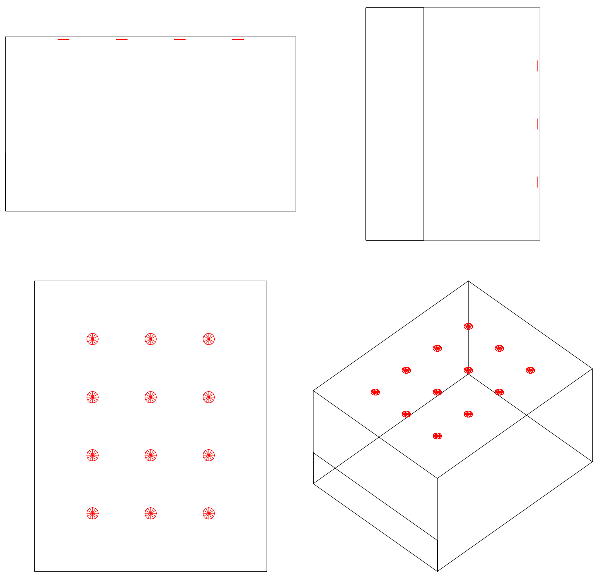


Figure 3: Inspecting the `xform` result with `objline`

```
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
0
```

The check that the conversion worked all right, we will try to match one of the values in ERCO's illuminance table to the Radiance output, e.g. the illuminance at a distance of 3.0 m should be about 78 lx.

```
$ oconv erco.rad > luminaire.oct
$ echo "0 0 -3 0 0 1" |rtrace -I -h -ov luminaire.oct \
    |rcalc -e '$1=179*$1'
77.795906
```

2.1.2 Array of Luminaires with `xform`

The luminaire that we have generated from the IES file will now be put into the test room. A 20 W lamp wattage is not a lot, so we'll need more than one lamp. The `xform` command allows us to easily build arrays of things, so 3 times 4 fittings will be used, spaced 1.0 m between them and also 1.0 m to the wall.

```
$ cat lights.rad
!xform -t 1 1 2.95 -a 3 -t 1 0 0 -a 4 -t 0 1 0 erco.rad
```

To see if it all worked, display a line drawing of the lights and the room (The image below has larger-than-normal light sources to make them easier to spot). See Fig 3.

Now compile a new octree with the room, the lights, some furniture and no sky.

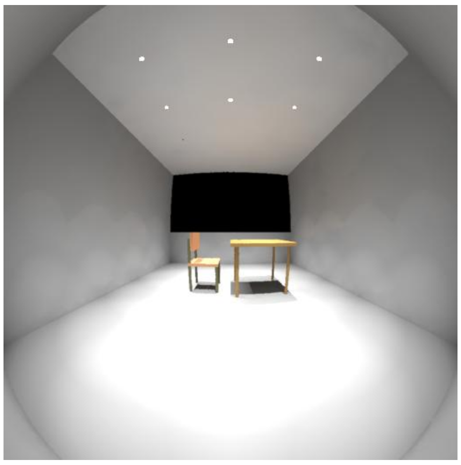


Figure 4: Fish-eye view of the scene with array of luminaires

```
$ oconv course.mat room.rad furniture.rad \
    lights.rad > erco.oct
```

View the scene interactively with `rvu`, and set a fish-eye view for an observer that is seated in the back of the room, looking towards the window, like demonstrated in Fig 4.

Before quitting `rvu`, save the view as *fisheye.vf*. We will required them for our glare assessment in section 5.1.

2.2 Growing a Plant with `xform`

If you take the time to look at a plant closely, you will notice that it basically consists of more or less identical leaves, although in different sizes, growing around a branch. While your first impression will be that the arrangement around the branch is rather chaotic, the growth pattern of a large number of plants can be described very simply mathematically by means of the Fibonacci Series [27]. It describes the members of that series as the sum of the two previous ones:

$$F_n = F_{n-2} + F_{n-1}$$

Starting with zero and one, the first few numbers of the Fibonacci Series are:

$$0, 1, 1, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, ...$$

The ratio of successive Fibonacci numbers F_n/F_{n-1} approaches the Golden Ratio, Φ, as n approaches infinity. The Golden Ratio, also known as Golden Section or just Phi, is frequently used in Arts due to the aesthetic balance it creates. Many natural examples exist in the world of plants and animals which are governed by the Fibonacci Series and the Golden Ratio. The science dealing with growth pattern in plants is called phyllotaxis. In a large number of plants, a new leave will grow offset by a certain angle to the last one. This is to minimise the overlap for any given leaf. This angle may be derived from Φ

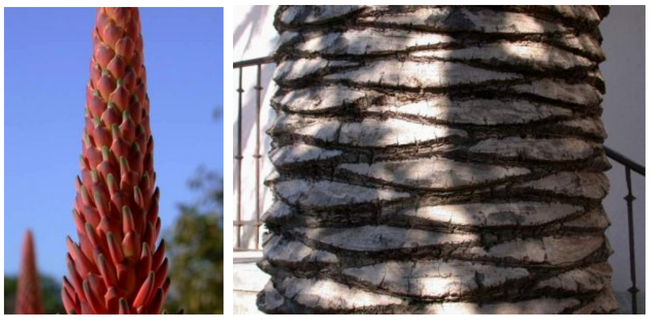


Figure 5: Flower of an Aloe, and stem of a palm tree

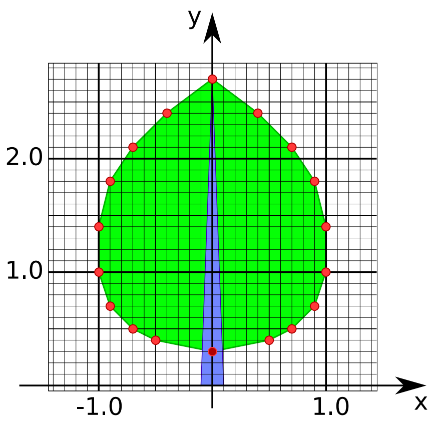


Figure 6: Tracing out a leaf

$$\begin{aligned}\frac{1}{\Phi} \times 360^\circ &= 222.5^\circ \\ \frac{1}{\Phi^2} \times 360^\circ &= 137.5^\circ\end{aligned}$$

Interestingly, those two angles are complementary and result in a full circle when added. The angular offset between successive leafs results in two distinguished spirals along the branch or stem, one clockwise, the other counter-clockwise. The two spirals have different numbers of arms, which are always successive members of the Fibonacci Series. This seems to be the optimum natural packaging for the highest density of leafs. If, for instance, we count five arms in the clockwise direction, there will be either three or eight in the counter-clockwise direction. Next time you look at a pine cone or a pineapple, count the number of spirals in either direction. You'll be surprised [7]. Fig 5 shows those spirals for the flowers of an Aloe and for the stem of a palm tree.

Enough said for the introduction, let us now create a simple plant following the principle mentioned above. To start with, draw the shape of a leave on a bit of graph paper. You might want to get a real leaf and trace it out, or just let your imagination roam. Fig 6 may give you some inspiration.

The red dots were added for clarity. The blue triangle will be conical and form the stem of the leaf. It's a little tedious, but the coordinates have to be translated into a Radiance `polygon` now. Let's call the file *leaf.rad*. You can start the polygon on any of the red dots. The example below starts with the bottom one. Remember that the starting point is not repeated at the end, Radiance will automatically close the polygon for you. It is important that all points within the polygon are in one plane. If you would like to give some depth to your leaf, you must triangulate, making sure that all polygons are in themselves perfectly flat.

```
leaf_mat polygon leaf
0
0
54
   0 .3 0     .5 .4 0    .7 .5 0     .9 .7 0
   1 1 0      1 1.4 0    .9 1.8 0    .7 2.1 0
   .4 2.4 0   0 2.7 0    -.4 2.4 0   -.7 2.1 0
   -.9 1.8 0  -1 1.4 0   -1 1 0      -.9 .7 0
   -.7 .5 0   -.5 .4 0
leaf_mat cone stem
0
0
8
   0 0 0
   0 2.7 0
   .1 0
```

Make the corresponding material a little bit shiny. Our creation should look like a healthy plant.

```
void plastic leaf_mat
0
0
5  0 .5 0  0.02 0.1
```

This leaf is now copied multiple times. With each step, it is moved a little bit along the branch, and rotated by the 'Magic Angle'. Call this file *plant.rad*. It's possible to start with the largest leaf and scale down as more leafs are added. However, we'll take the opposite approach and start with the smallest leaf, growing the plant downwards. This way, we avoid having too many little leaves that are so small we'll never be able to see them in our simulation, and only add to the size of the model.

```
!xform -rx 10 -a 50 -rz 137.5 -s 1.05 -t 0 0 -.3 leaf.rad
```

Let's dissect the `xform` command above. The single leaf is first rotated by 10° to make it point up a bit. With the -a (as in 'array') option, the following transformations are repeated 50 times: The leaf is rotated by 137.5°, scaled up up 5% and moved down by 0.3 units.

Now use `getbbox` to find out the height of your creation and model a stem.

```
leaf_mat cone stem
0
0
8   0 0 -65
    0 0 1
    1.15 0
```

Fig7 shows the result, with a sky and ground plane added.

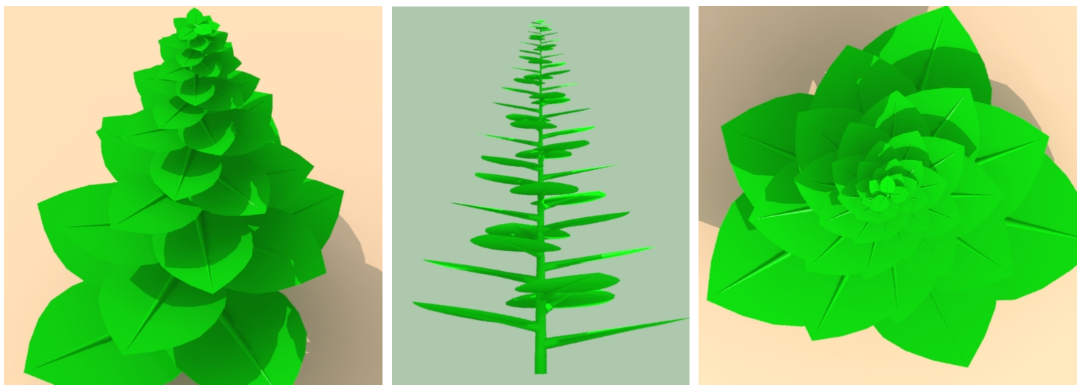


Figure 7: Our home-grown plant

2.3 Using `genworm`

Radiance comes with a number of powerful generators for geometry. We have already used the `genbox` command in the first part of this tutorial to create a simple box for our room. `genbox` is a very basic example of the Radiance generators. Some more advanced ones allow us to create complex shapes based on parametric equations. There are several programs in this group (the definitions are taken from the man pages):

`genworm`: produces a Radiance scene description of a worm defined by the parametric equations x(t), y(t), z(t), and r(t) (the radius). T will vary from 0 to 1 in steps of 1/`nseg`. The surface will be composed of `nseg` cones or cylinders and `nseg`+1 spheres.

`genrev`: produces a Radiance scene description of a surface of revolution. The object will be composed of nseg cones, cups, cylinders, tubes or rings following the parametric curve defined by z(t) (height) and r(t) (radius).

`gensurf`: produces ... a Radiance scene description ... of a functional surface defined by the parametric equations x(s,t), y(s,t), and z(s,t). ... S will vary from 0 to 1 in steps of 1/m, and t will vary from 0 to 1 in steps of 1/n. The surface will be composed of 2 · m · n or fewer triangles and quadrilaterals.

We will examine the first of the three in the following. The other two work in a very similar fashion and should be easy to pick up with the knowledge about `genworm` under your belt.

The general use of `genworm` is:

```
genworm material name 'x(t)' 'y(t)' 'z(t)' 'r(t)' nseg
```

Let's start by creating a quarter of a circle with a round cross-section. If the parameter t in equations 2 and 1 below is in the range from 0 to $\pi/2$, then x and y will describe an arc of 90°.

$$x(t) = cos(t) \quad (1)$$
$$y(t) = sin(t) \quad (2)$$

The equivalent `genworm` command defines the z-coordinate as 0 and gives the thing a radius of 0.1 units, as shown in Fig 8.

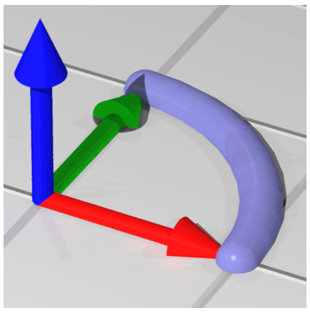


Figure 8: A wormy quarter of a circle.

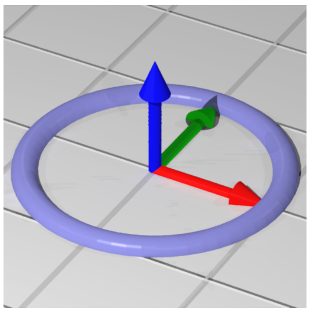


Figure 9: Extending the worm to a full circle.

```
!genworm worm_mat worm 'cos(.5*PI*t)' 'sin(.5*PI*t)' '0' '.1' 10
```

Changing the range of t from $\pi/2$ to 2π results in a circle. This is shown in Fig 9. The last argument is responsible for the number of segments which are created. It should be increased for longer curves to ensure a smooth look

```
!genworm worm_mat worm 'cos(2*PI*t)' 'sin(2*PI*t)' '0' '.1' 60
```

More interesting shapes can be achieved by modifying z at the same time. Please note that shapes other than round ones can be defines just as easily, e.g. elongated worms. Compare Fig 10.

```
!genworm worm_mat worm 'cos(4*PI*t)' 'sin(4*PI*t)' 't' '.1' 200
!genworm worm_mat worm '.5*t*cos(9*PI*t)' '.5*t*sin(9*PI*t)' \
    '2-2*t' '.1' 120
!genworm worm_mat worm 'cos(2*PI*t)' 'sin(2*PI*t)' \
    '.5+.5*cos(8*PI*t)' '.1' 120
```

The diameter of the worms can be modified just as easily as the shape of the axis. See Fig 11 for our final creations.

```
!genworm worm_mat worm '.5*t*cos(9*PI*t)' '.5*t*sin(9*PI*t)' \
    '2-2*t*t' '.001+.5*t' 120
!genworm worm_mat worm 'cos(2*PI*t)' 'sin(2*PI*t)' \
   '.5+.25*cos(8*PI*t)' '(1.3+sin(8*PI*t-PI/2))/10' 180
```

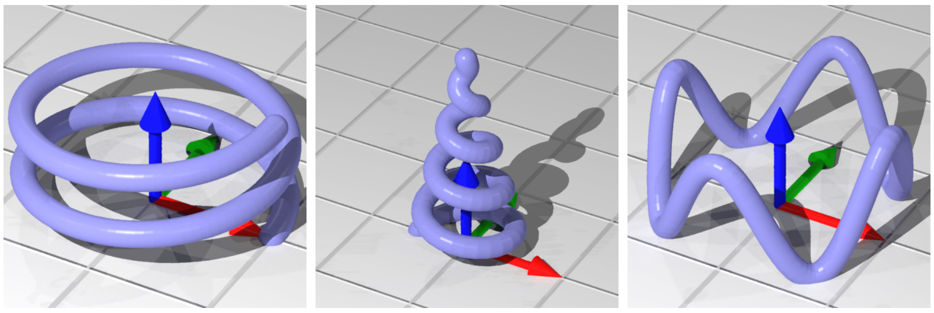


Figure 10: Spirally and wiggly worms

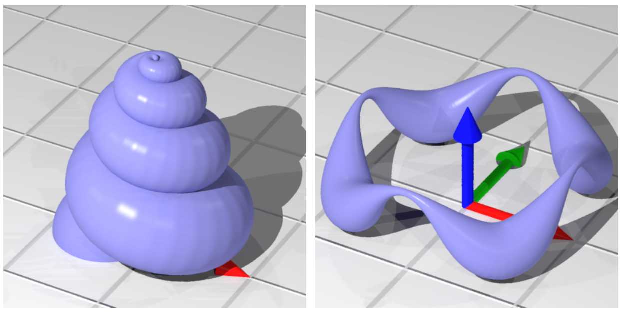


Figure 11: Martian worms. Yes, this is what they look like.

Other than for sophisticated objects of art, `genworm` can be very handy for modelling bends in pipes, towel rails, door handles etc. `genworm` builds up the objects it creates from cones and spheres, closing the ends of the shape with sphere. In contrast, `genrev` creates objects of revolution from cones with the ends open. `gensurf`, finally, produces polygon descriptions. It is very powerful and is used for anything from random landscapes to window panes.

2.4 Prototyping for `gensurf` with `gnuplot`

`gnuplot` is a command-line driven interactive data and function plotting utility. It has a long history, is still under active development [2]. There are many good-quality tutorials on the Internet demonstrating its usage.

The software is not only useful for plotting 2d and 3d set of data, but is also capable of generating complex 3d surfaces. Since the mathematics behind those shapes is not everybody's cup of tea, we can benefit from the many examples. With some minor modifications, the `gnuplot` syntax may be transformed into a description for Radiance's `gensurf`. We'll take a nice knot from the Some Fun with GNUPLOT page as an example [30].

The following lines tell `gnuplot` to output a 3d knot, as in Fig 12:

```
set parametric
set hidden3d
set isosamples 100,20
set view 45,315
splot [-3*pi:3*pi][-pi:pi] \
   cos(u)*cos(v)+3*cos(u)*(1.5+sin(u*5/3)/2), \
```

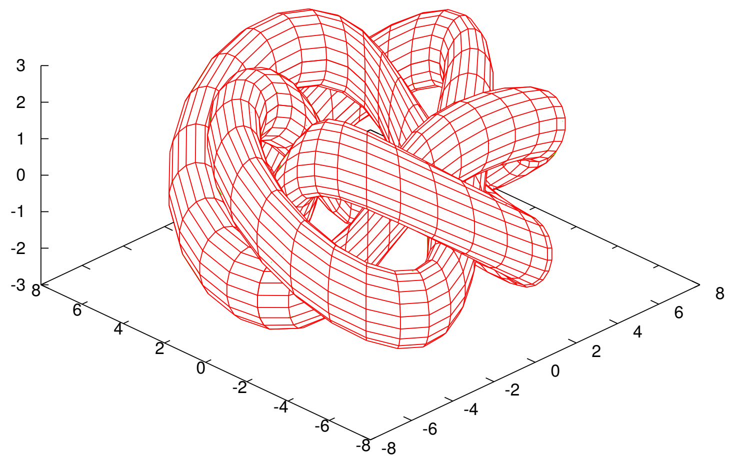


Figure 12: Knot plotted with `gnuplot`

```
sin(u)*cos(v)+3*sin(u)*(1.5+sin(u*5/3)/2), \
sin(v)+2*cos(u*5/3)
```

The general syntax of `gensurf` is:

```
gensurf mat name 'x(s,t)' 'y(s,t)' 'z(s,t)' m n
```

Before we make our alterations, here is a list of the mayor differences in syntax between `gnuplot` and `gensurf`:

Syntax	`gnuplot`	`gensurf`
Parameters	u,v	s,t
Range of parameters	adjustable [from:to]	always [0 to 1]
Representation of π	pi	PI
Separation of equations	', ' (comma and space)	' ' (space)
Equations enclosed in	nothing	' (high comma)
Number of steps	set isosamples u,v	m and n parameter

Using this table, this is what need doing for transforming gnuplot `syntax` into `gensurf` syntax:

1. Replace splot with gensurf,
2. Enclose the equations for x, y and z with high commas and remove the separating commas,
3. Add two more parameters (m and n), separated by space. Use what is given as isosamples in the gnuplot description as a starting point,
4. Optionally add the -s parameter to smoothen the surface,
5. Replace all occurrences of u and v with the modified expression for s and t. Don't forget to put them in brackets as required. Some examples are shown in the following table:

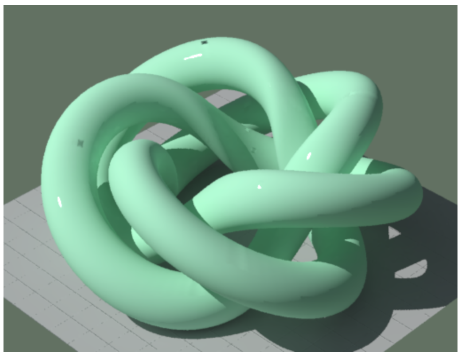


Figure 13: The same knot, rendered with Radiance

Range	`gnuplot`	`gensurf`
0 to 2π	[0:2*pi]	s*2*PI
0 to 5	[0:5]	s*5
$[-3\pi$ to $3\pi]$	[-3*pi:3*pi]	s*6*PI-3*PI

Following these step-by-step instructions, you should end up with the following `gensurf` command:

```
gensurf knot_mat knot \
  'cos(s*6*PI-3*PI)*cos(t*2*PI-PI)+ \
  3*cos(s*6*PI-3*PI)*(1.5+sin((s*6*PI-3*PI)*5/3)/2)' \
  'sin((s*6*PI-3*PI))*cos(t*2*PI-PI)+ \
  3*sin((s*6*PI-3*PI))*(1.5+sin((s*6*PI-3*PI)*5/3)/2)' \
  'sin(t*2*PI-PI)+2*cos((s*6*PI-3*PI)*5/3)' \
  300 30 -s
```

Fig 13 shows the same knot as Fig 12, but this time rendered with Radiance.

There are many weird and wonderful examples of `gnuplot` 3d surfaces out there, feel free to borrow what others have created, and try it out with Radiance.

2.5 Exotic Materials

2.5.1 glass

The Radiance Reference Manual [18] supplies the following formula to convert from transmittance (T_n), which is easily measured, to transmissivity (t_n):

$$t_n = \frac{\sqrt{0.8402528435 + 0.0072522239T_n^2} - 0.9166530661}{0.0036261119T_n} \qquad (3)$$

Standard 88% transmittance glass has a transmissivity of 0.96. Formula 3 looks a bit daunting can be simplified to:

$$t_n = 1.09 \times T_n \qquad (4)$$

As demonstrated in Fig 14, the result are virtually identical. A factor of 1.0895 may be used if a higher accuracy is required.

This simplified formula also has the advantage of being much less sensitive to rounding. Using the example of a 0.88 transmittance, we get:

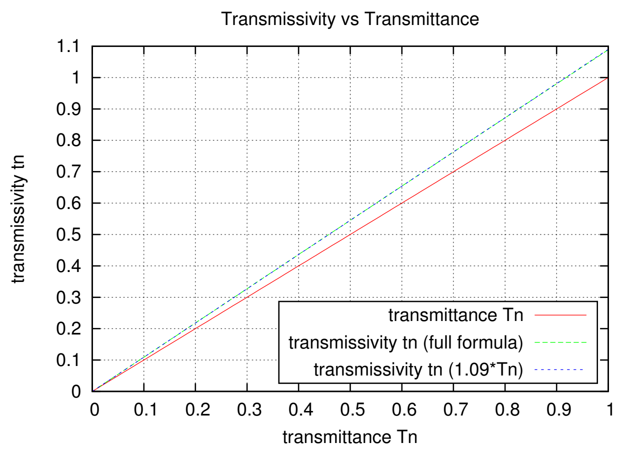


Figure 14: Transmissivity vs transmittance. The red line is there for comparison.

```
$ echo 0.88 |rcalc -e \
    '$1=(sqrt(.8402528435+.0072522239*$1*$1)\
    -.9166530661)/.0036261119/$1'
0.958415433
```

for the transmissivity. The square root of the numerator alone is in this case

```
$ echo 0.88 |rcalc -e '$1=sqrt(.8402528435+.0072522239*$1*$1)'
0.919711349
```

which is rather similar to the 0.9166530661 subtrahend. The result of the subtraction is then

```
$ echo 0.88 |rcalc -e \
    '$1=(sqrt(.8402528435+.0072522239*$1*$1)\
    -.9166530661)'
0.00305828301
```

which again is close to the divisor of 0.0036261119. It is therefore important to carry through all decimal places from the 'full' formula, or else the equation is likely to 'flip', producing the wrong result. The simplified formula is much more robust while, for all practical purposes, producing the same results.

2.5.2 trans

Syntax:

```
modifier trans identifier
0
0
7 red green blue  spec rough  trans tspec
```

Although the trans material seems to be relatively simple, getting the modifiers right to accurately describe the material you are trying to model can be somewhat confusing. To understand why, let's see what Greg Ward has to say about it:

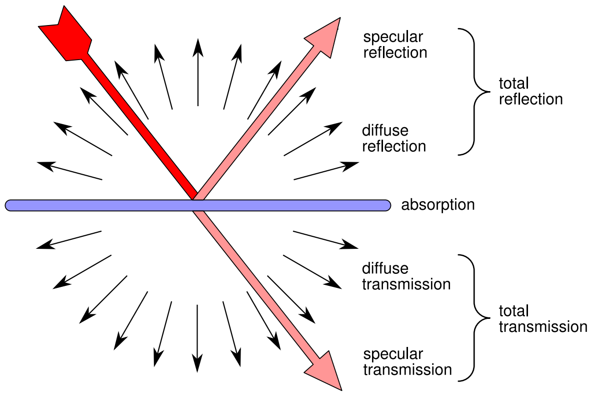


Figure 15: Illustrating the components defined for a trans material

> “By way of excuse, the reason the trans type is so baffling is my stubborn adherence to the principle that Radiance primitives have well-defined legal ranges. All the material parameters (except roughness) have legal physical ranges of [0,1). Since Radiance does not enforce these limits, you can specify values outside this range, but you should know that you are on shaky ground at that point.
>
> “That said, I freely admit that the obtuse derivation of the trans parameters in particular has caused a good deal more consternation than if I had used more usual values of Rd, Td, Rs, and Ts and simply noted that these coefficients must sum up to something less than 1. Oh, well.[1]”

There you have it. If you imagine the light being reflected off the material and passing through it as a process that happens in stages, then the modifiers describe the light’s intensity relative to the previous stage, not relatively to the input (which is considered as 1.0). Let’s list the individual stages (compare Fig15):

1. Specular reflection: Some of the light bounces off the surface without even having the time to take on the surface’s colour
2. Absorption: A fraction of what’s left from 1. is absorbed and ends up as heat
3. Diffuse reflection: Everything that is not transmitted is diffusely reflected in the colour of the material
4. From the transmitted light, a part is diffuse, ...
5. ..., while what’s left is specular.

This concept is illustrated in Fig16.

The roughness value, if greater then zero, will modulate the specular components (reflected and transmitted) and cause them to diverge. Here is an example of the use of the material from the Radiance digest[2].

```
void trans opale
0
0
7 .3 .3 .3  0 0  .6 .1
```

[1]Radiance mailing list, March 2005, thread ’trans mat’
[2]v2n7

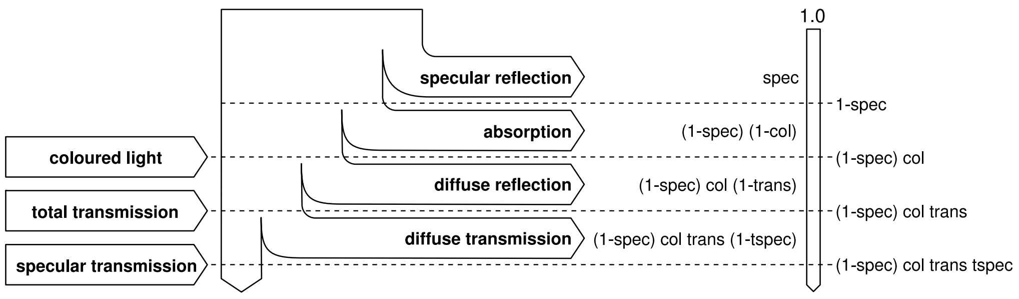


Figure 16: Energy split-up as light passes through a trans material

Using the formulae from the illustration above, we find:

$$\begin{aligned}
\text{diffuse reflectance} &= (1 - spec) \times colour \times (1 - trans) \\
&= (1 - 0) \times 0.3 \times (1 - 0.6) \\
&= 0.12 \\
\text{diffuse transmittance} &= (1 - spec) \times colour \times trans \times (1 - tspec) \\
&= (1 - 0) \times 0.3 \times 0.6 \times (1 - 0.1) \\
&= 0.162 \\
\text{specular transmittance} &= (1 - spec) \times colour \times trans \times tspec \\
&= (1 - 0) \times 0.3 \times 0.6 \times 0.1 \\
&= 0.018
\end{aligned}$$

Please refer to [16] for another very nice diagram, as well as the ranges of usable values for the parameters. One last thing: It should be apparent that a trans parameter of zero will cause the material to behave exactly light plastic. Or is it?

2.6 Image Mapping

2.6.1 Creating a Picture Frame

Radiance allows us to define object surface patterns with powerful mathematical generator, thus creating wood, water, carpets or other building materials. However, it is sometimes necessary to map images taken with a digital camera or scanned in onto surfaces. As an example, we are going to create a picture frame with the image of Tux in it. Tux the penguin is the mascot of LINUX. A particularly well-known image is created by Larry Ewing and is available from his web site [21] and shown in Fig 17.

For the picture, we first create a frame which is 0.8 m wide, 0.9 m high and 2.0 cm thick. We then put a canvas in front of it leaving a 10 cm margin on all sides. The files *picture_tux.rad* and *picture_tux.mat* can be found in the appendix.

We are going to use the `colorpict` primitive which is designed for exactly what we intend to do. The modifiers are a little complex. Here is a snippet from the file *picture_tux.mat*:

```
void colorpict tux_image
7 clip_r clip_g clip_b tux.hdr picture.cal pic_u pic_v
```

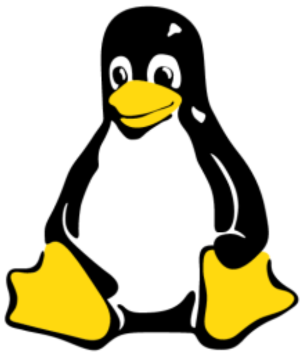


Figure 17: Tux the penguin is the mascot of LINUX

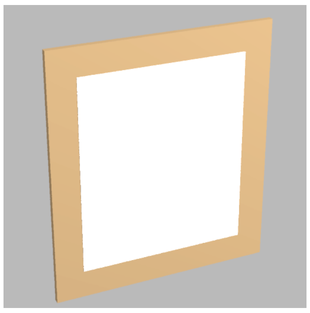


Figure 18: An empty picture frame

```
0
0

tux_image plastic painting_mat
0
0
5  1 1 1  0 0
```

Please note that the `colorpict` pattern which we called `tux_image` acts as a modifier to the material `painting_mat`. The results looks like Fig 19.

Hmmm. This is not at all what we expected. To ure out what went wrong here, it is helpful to indicate the origin and orientation of the coordinate system (Fig 20). The file is attached in the appendix as *arrows.rad*, containing the geometry and the material

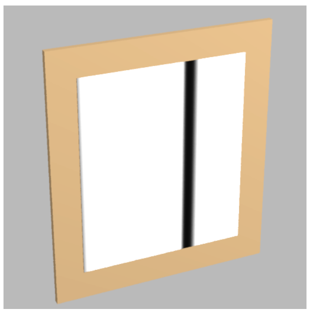


Figure 19: A bit of Tux, but which part?

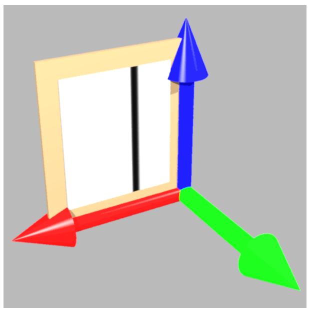


Figure 20: Working out the Tux problem, step 1

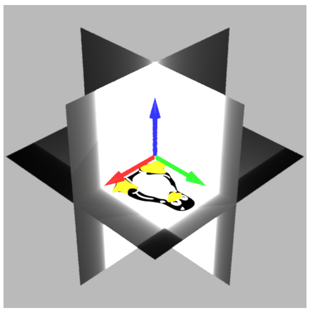


Figure 21: Working out the Tux problem, step 2

of the arrows.

We might suspect that Tux got mapped to a plane other than our canvas. To test this theory, we use the arrows again and additionally create the three cardinal planes extending from -1.5 to +1.5 units. A big white box would do the same trick, but then we wouldn't be able to see the arrows indicating the coordinate axises.

From the new view in Fig 21 it is clear that Radiance maps images to the xy-plane with one of the corners at (0, 0, 0). The black vertical line on the canvas seems to be part of Tux' right foot. We really need the image to be vertical though, not horizontal. The solution is to use the `xform` command. The second line of every Radiance primitive will take character arguments, and `colorpict` is happy to have `xform` instructions included here. So we modify the second line of the `colorpict` primitive in *picture_tux.mat* to

```
9 clip_r clip_g clip_b tux.hdr picture.cal pic_u pic_v -rx 90
```

Please make sure to increment the ure at the beginning of the line. This is the number of arguments that follow.

Fig 22 is a lot better know, but Tux doesn't really fit the frame. He needs to be scaled down. The Radiance reference manual explains that the smaller dimension of the image is taken to be 1, the other is the ratio between the larger and the smaller. The command below outputs the dimensions of *tux.hdr*:

```
$ getinfo -d tux.hdr
tux.hdr: -Y 256 +X 219
```

$256/219 = 1.1689$, so the image is 1 unit wide and 1.1689 units wide. Our canvas is 0.6 wide and 0.7 high. The scaling factor needs to be $0.6/1 = 0.6$ or $0.7/1.1689 = 0.5989$, whichever is smaller. This now becomes another argument to `colorpict`:

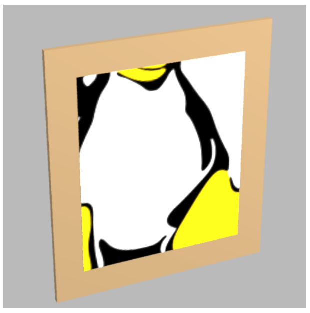


Figure 22: At last! A penguin belly!

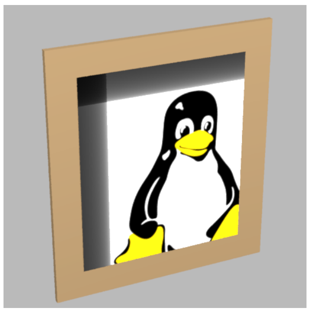


Figure 23: Almost done

```
11 clip_r clip_g clip_b tux.hdr picture.cal pic_u pic_v \
    -rx 90 -s .5989
```

We're almost there. The last thing to do is to move Tux into the centre of the canvas. This is because the bottom corner of the canvas does not touch the co-ordinate origin but is off-set by the picture frame which is 10 cm in x-direction and 10 cm in z-direction.

```
15 clip_r clip_g clip_b tux.hdr picture.cal pic_u pic_v \
    -rx 90 -s .5989 -t .1 0 .1
```

This is it. The final Tux in Fig 24 doesn't seem to be too upset about all the squeezing we've done to him.

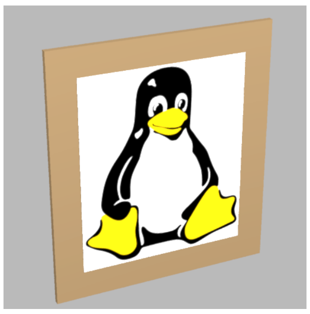


Figure 24: A happy Tux

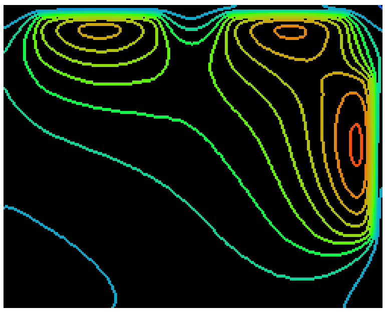


Figure 25: `falsecolor` contour lines on a black background.

2.6.2 Transparent Textures

This exercise will guide you through some more advanced image mapping techniques. To start with, we need the model of the room that has three windows in it and rather thick walls. You will find it in the ZIP archive. We also need the result from section 5.6, namely the illuminance image for the working plane in this room. That section finished with a false colour image of the working plane illuminance, but what we need here is just the contour lines against a black background. Fig 25 is given as a reminder of the contour plots against a black background.

```
$ falsecolor -n 10 -s 1000 -lw 0 -cl -i wp_illu.hdr > sp.hdr
```

The next prerequisite is a polygon which is to act as an impostor for the working plane that doesn't exist in reality. The working plane is usually taken as 0.7 or 0.85 m above the floor. Make sure your polygon stretches right into the four corners of the room without penetrating the walls.

To ensure that nobody mistakes the false colours on the working plane as part of the room geometry, we'll use a glow material. Depending on the level of ambient light in the scene, you might have to adjust the intensity. glow, unlike plastic, can have values above one. Set the glow radius to 0, so no extra light is added to the scene. The file in the listing below implements what we learnt in section 2.6.1. *picture.cal* comes with the Radiance distribution.

```
$ cat wp.rad
void colorpict fc_val
15 red green blue sp.hdr picture.cal pic_u pic_v \
    -s 4 -rz -90 -t .5 5.5 0
0
0

fc_val glow fc_glow
0
0
4  10 10 10 0

fc_glow polygon working_plane
0
0
```

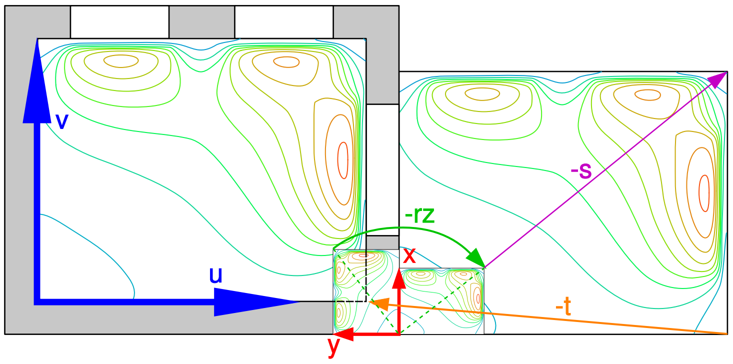


Figure 26: Transforming the contour lines to fit in the room

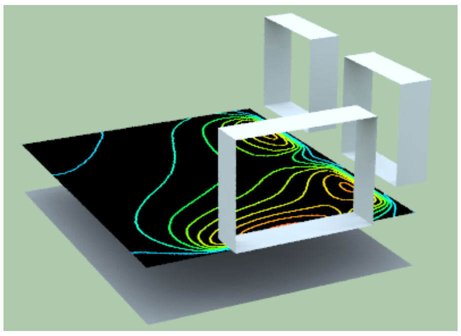


Figure 27: The contour lines in the right position

```
12    4.5     0.5     0.8
      4.5     5.5     0.8
      0.5     5.5     0.8
      0.5     0.5     0.8
```

The sketch in Fig 26 will make the transformations more obvious. It's always a good idea to sit down with pen and paper before attempting anything that you can't easily ure out in your head. (x, y) are the normal Radiance world co-ordinates. (u, v) denote the co-ordiantes within the image that is applied. The smaller of the two runs from zero to one.

Expectedly, the working plane is black and not transparent, but at least it shows the image as desired. All we have achieved so far is to put the contour lines onto the working plane and rotate and scale the image correctly so it lines up with the windows. For better visibility, the walls have been removed in the image below. Only the reveals, the floor and the working plane are shown in Fig 27.

Before we tackle the issue of transparency, let's take a slightly different approach with the mapping procedure. Instead of using `xform` to rotate, scale and move the mapped image, it's advantageous in our case to implement those transformations in a *.cal* file. As an interim stage, let's not use Radiance's *picture.cal* but define our own one, *fc.cal*, instead. Please take a look at *rayinit.cal* (you do know how to find it on your system, don't you?) to learn what those Px and Py things are all about.

```
$ cat fc.cal
fc_u = Px;
```

```
fc_v = Py;
$ cat wp.rad
void colorpict fc_val
15 red green blue sp.hdr fc.cal fc_u fc_v -s 4 \
    -rz -90 -t .5 5.5 0
0
0

fc_val glow fc_glow
0
0
4 10 10 10 0

fc_glow polygon working_plane
0 ...
```

This produces exactly the same output as the version above. What we are trying to achieve is to get rid of the transform commands at the end of the second line of the colorpict primitive and instead have all transformations in *fc.cal*.

```
$ cat fc.cal
fc_u = -(Py-5.5)/4;
fc_v =  (Px-0.5)/4;
$ cat wp.rad
void colorpict fc_val
7 red green blue sp.hdr fc.cal fc_u fc_v
0 ...
```

fc.cal makes a co-ordiante transformation now instead of just a normal 1:1 mapping. -s 4 -rz -90 -t .5 5.5 0 from the original colorpict primitive has the same effect as

$$\begin{aligned} u &= \frac{-y-5.5}{4} \\ v &= \frac{x-0.5}{4} \end{aligned}$$

in *fc.cal*. Now that we have all the transformations in the *.cal* file, it's time to address the transparency issue. The trick is to use the mixpict primitive. It allows you to have two different materials, based on a Radiance picture. What we want is:

- a void, i.e. no material at all for all areas that are black in our false colour contour line map;
- a glow material for the actual false colour lines.

It is important to apply the same transformations to the mixpict and to the colorpict primitives. This is the reason why we put all the transformations in the *.cal* file. It reduces the possibility of introducing errors along the line. To be able to use the same *.cal* file for the image transformation, as well as the transparency handling, we need one more line in *fc.cal* and the new mixpict primitive in *wp.rad*:

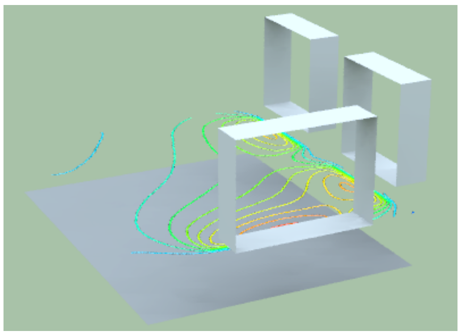


Figure 28: Contour lines on a transparent plane

```
$ cat fc.cal
nonzero(r,g,b) = if(max(r,max(g,b))-FTINY, 1, 0);
fc_u = -(Py-5.5)/4;
fc_v = (Px-.5)/4;
$ cat wp.rad
...
void mixpict mp
7 fc_glow void nonzero sp.hdr fc.cal fc_u fc_v
0
0

mp polygon working_plane
0 ...
```

The line in *fc.cal* beginning with 'nonzero' means:

> If either red, or green, or blue are non-black (which in our case is true only for the actual contour lines), then return one. Otherwise return zero.

In computing, 'zero' is usually equivalent to 'false', while 'one' is associated to 'true'. In other words, for every pixel that evaluates to 'true' (the colour of the contour line map is not black), we use glow as the material, for all that are 'false', we use void. FTINY is explained in *rayinit.cal*. It's a constant evaluating to 1×10^{-7} and is there to guard against rounding errors. You might get away without it, but are advised to leave it as it is. Fig 28 show that our efforts have paid off.

When putting the room's envelope back in, please adjust the parameters of the glow primitive. They depend on the ambient light level, as well as on the exposure. Although a glow radius of zero means that no light is introduced into the scene, an intensity that is too high will not show the graduation of the line colours. For this particular scenario, try something like 2 2 2 or 3 3 3. The final result is given in Fig 29.

2.7 Using `pcomb` for Creating and Combining Images

`pcomb` is a versatile tool which can combine two or more Radiance images but may also be used to actually create pictures. When using `pcomb`, it is often a good idea to make sure the RGB of the pixels is between 0 and 1.

Let's just create a simple red ramp along the width of an image. We want it to be black at the left edge (which has pixel-coordinates of x=0), and red on the other

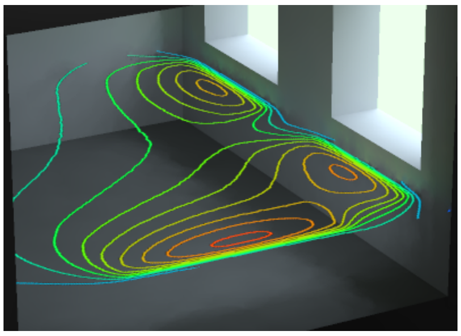


Figure 29: The contour lines, now defined as glow, in the room.

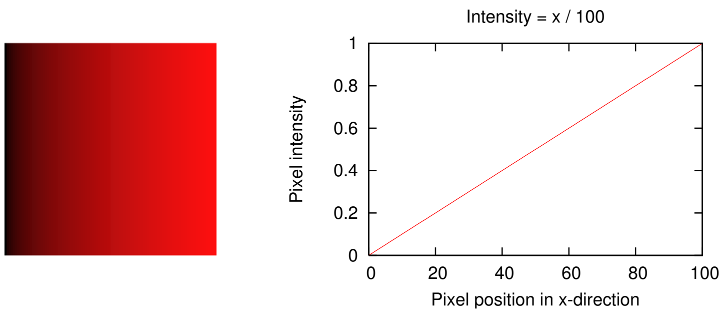


Figure 30: Red ramp: Image and intensity graph.

side (x=max). In in our case the image is 100x100 pixels. The resulting image and the intensity graph are shown in Fig 30.

```
$ pcomb -x 100 -y 100 -e 'ro=(x/100);go=0;bo=0' | ximage
```

The division by 100, which is the maximum dimension in x, ensures that the maximum red value is 1. It is also possible to use if statements with `pcomb`. They take the form of:

if(cond,then,else) if cond is greater than zero, then is evaluated, otherwise else is evaluated

There are other functions as well which are available to many Radiance commands. Please see an overview in the `ev` man page. Using this approach, it is easy to produce an image that is only red and black, like the one in Fig 31:

```
$ pcomb -x 100 -y 100 \
    -e 'ro=if((x-50),1,0);go=0;bo=0' | ximage
```

The output may, of course, depend on both, x and y pixel positions, like the red dot in Fig 32.

```
$ pcomb -x 100 -y 100 -e \
    'ro=(1-((x-50)^2+(y-50)^2)/2500);go=0;bo=0' | ximage
```

Now let's build up for something else: A representation of colour mixing. First, we create three images with different coloured dots on them, like the ones in Fig 33. This time, however, the canvas dimensions are 50% larger.

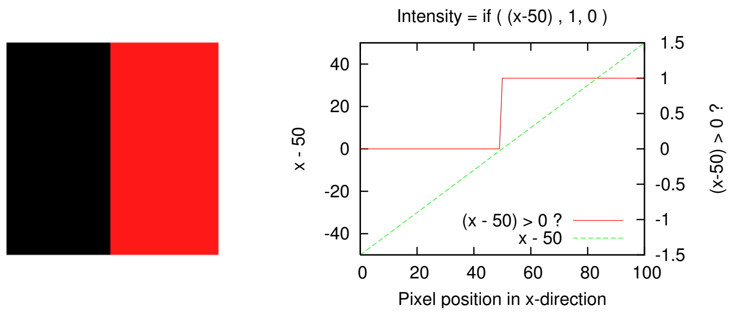


Figure 31: Red-and-black square

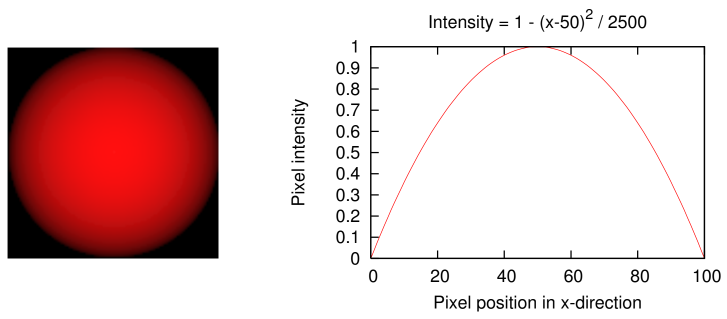


Figure 32: Red dot

```
$ pcomb -x 150 -y 150 -e \
'ro=0;go=b;bo=0;b=if(1-(((x-50)^2)+(y-50)^2)/2500,1,0)' \
> green_dot.hdr
$ pcomb -x 150 -y 150 -e \
'ro=b;go=0;bo=0;b=if(1-(((x-75)^2)+(y-100)^2)/2500,1,0)' \
> red_dot.hdr
$ pcomb -x 150 -y 150 -e \
'ro=0;go=0;bo=b;b=if(1-(((x-100)^2)+(y-50)^2)/2500,1,0)' \
> blue_dot.hdr
```

Adding up those dots of the three primary colours will produce the secondary colours, demonstrating the concept additive of colour mixing. This is used for computer displays and other imaging devices that produce the colours from light. For technologies such as printing, e.g. where colours are created through reflection, the primaries are cyan, magenta and yellow, i.e. the secondary colours from above [12].

```
$ pcomb red_dot.hdr blue_dot.hdr green_dot.hdr > rgb_dots.hdr
```

Isn't the result in Fig 34 nice and colourful? In case you were wondering if there are

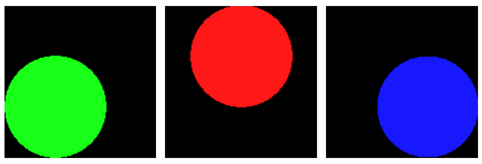


Figure 33: Dots of three different colours

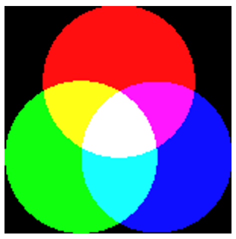


Figure 34: Additive colour mixing

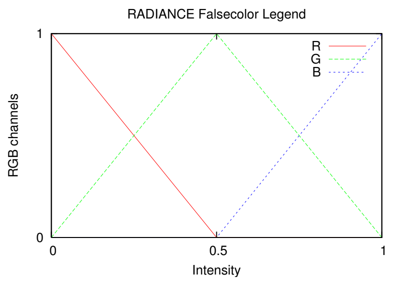


Figure 35: RGB mapping for a false colour scale

any serious applications to this, just imagine the following scenario: A new development is proposed which will reduce the daylight availability to your client's property. To assess the situation, you create simulations with and without the new obstruction. The reduction of light is the difference of the two images processed with `pcomb`. There is an ancient right in the UK called Rights of Light which is exactly about this.

Let's keep it nice and colourful. We're all familiar with the legend which the `falsecolor` command creates. If you need to refresh your memory, take a look at section 3.3.1. Daylight factors (or luminance or illuminance values) are mapped from red (high) via green (medium) down to blue (low). To create something similar, we need to define a mapping function that looks like the one in Fig 35 for the RGB channels:

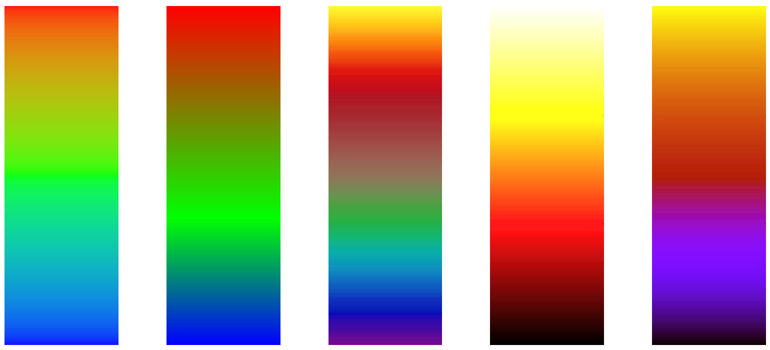
The `pcomb` command line options for this are shown below. To be able to fully appreciate the colours, we make the image a bit wider (300 pixels). Conveniently, negative values which are not allowed in images are cut off at zero.

```
$ pcomb -x 300 -y 100 -e \
    'ro=(x-150)/150;go=if(x-150,1-((x-150)/150),x/150); \
    bo=(150-x)/150' > scale.hdr
```

If this is all a bit too sudden, try breaking this apart into the RGB channels and plot intensity and truth graphs like we did for the dots above.

Our own false colour palette is depicted in Fig 36 on the left hand-side. It has been rotated into an upright position to make a comparison easier. The one to the right of it is the actual palette that `falsecolor` would have created prior to Radiance version 3.8. Our graph is a pretty good match. The only difference to the 'real' one is that the point at which we assumed the green channel to peak is not really at 0.5, but at 0.375 instead. Here are the relevant lines from the actual `falsecolor` Perl script:

```
spec_red(x) = 1.6*x - .6;
spec_grn(x) = if(x-.375, 1.6-1.6*x, 8/3*x);
spec_blu(x) = 1 - 8/3*x;
```



(a) Our own palette

(b) `def` (default before Radiance 3.8, now available with `-pal def`)

(c) `spec` (default since Radiance 3.8, Oct 2006)

(d) `-pal hot` (new in Radiance 4.1, Nov 2011)

(e) `-pal pm3d` (new in Radiance 4.1, Nov 2011)

Figure 36: The different Radiance colour palettes through the ages

In case you're looking for more `falsecolor` exercises, you may take a look at section 5.4.

3 Daylighting

3.1 `mkillum` and 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 it covers the entire opening and that the surface normal points into the room. Use the file *objects/window.rad* which already contains the necessary `mkillum` options. The material windowglass is already defined in *materials/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:

```
$ mkillum -ab 0 < objects/window.rad scene.oct \
    > tmp/iwindow.rad
```

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

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

```
$ oconv -i scene.oct objects/furniture.rad \
    tmp/iwindow.rad > tmp/iscene.oct
```

Look at the result again:

```
$ rvu -vf views/floor.vf -ab 0 -av 0 0 0 tmp/iscene.oct
```

The warning message '`no light sources found`' no longer appears. 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 `rvu` reveals that the `-ds` option is set to zero unless otherwise specified. This variable controls the 'source sub-structuring'. If set to a value between 0 and 1, large light sources are split up into smaller parts, so there is more than one point that gets sampled for shadows.

While still in `rvu`, use the `:set` command to give `ds` a value of 0.3. 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.

To render a view of your choice with `rpict`, set `-ab` to 1 and give `-ad`, `-ds` and `-av` a reasonably value. Call the image *images/scene.hdr*.

```
$ rpict -vf views/myview.vf -ab 1 -ad 128 -ds .2 \
    -av .02 .02 .02 iscene.oct > images/scene.hdr
```

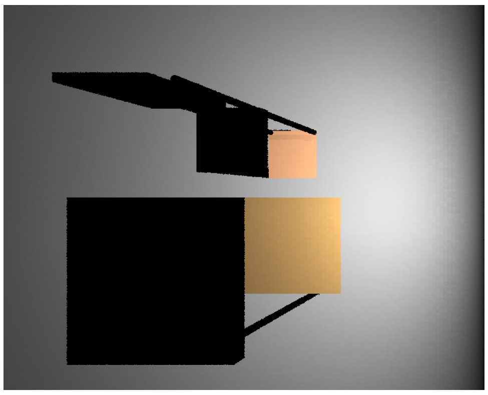


Figure 37: Strange light radiating from the centre of the window pane

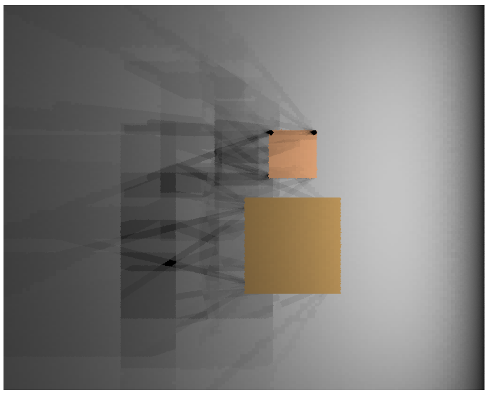


Figure 38: More light sources create more realistic shadows

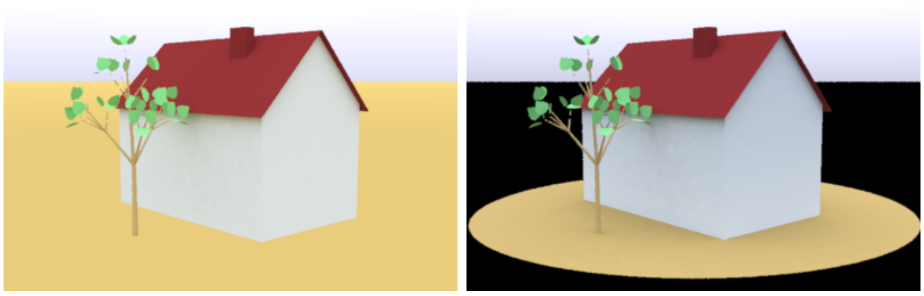


Figure 39: Ground hemisphere vs ground plane

3.2 Using a Ground Plane

We have two choices for modelling the ground our building stands on:

1. Extend the sky so it covers the lower hemisphere;
2. Introduce a ground plane.

The first approach is usually more convenient because `gensky` takes care of the material definitions. If ground-reflected light is important, however (and it may contribute a significant fraction of the available daylight in a space), it is important to accurately model the ground plane reflection. This is only possible by introducing a ground plane. See Fig 39.

Although `gensky` models the lower hemisphere as a Lambertian diffuser which is usually not too bad, problems arise when the internal illumination is highly dependent on light which is reflected off the ground very close to the building. A bottom glow hemisphere is, since it's infinitely far away from us just like the sky, never shaded. In effect the ground brightness is constant, even very close to an obstruction.

While you might think now 'the bigger the better', it is not true in this case. A very large ground plane might well improve the distribution of the ground reflected light. The downside is, however, that the size of the splotches that are sometimes the result of the ambient calculation is directly related to the largest model dimension. This is explained in more detail in the Radiance Tutorial [10].

So how then do we match up the colour of the ground plane and the ground glow while making sure the reflectance is right as well[3]?

Let's assume our ground plane has a colour of (r,g,b) = (0.4,0.3,0.1). The weighted average reflectance is

$$\begin{aligned}\rho &= 0.265r + 0.67g + 0.065b \\ \rho &= 0.265 \times 0.4 + 0.67 \times 0.3 + 0.065 \times 0.1 \\ \rho &= 0.3135\end{aligned}$$

This should be the `-g` option (ground reflectance) to `gensky`. The colour of the glow material of the bottom half of the sky should be the colour of the ground plane divided

[3]See Radiance digest V2n10.

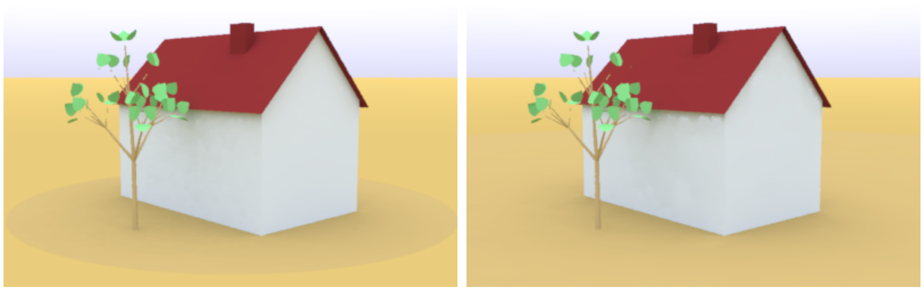


Figure 40: The ground plane initially, and after some fine-tuning.

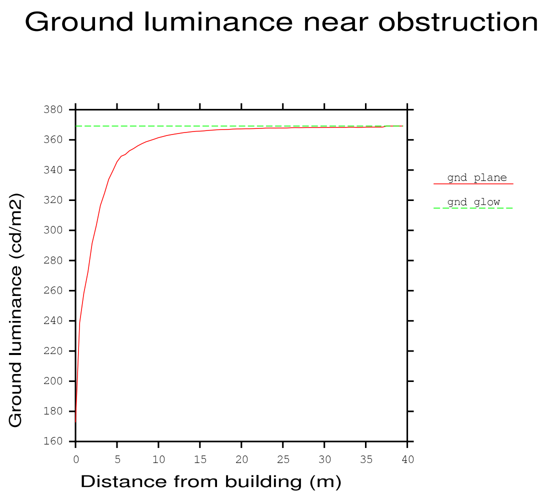


Figure 41: Plotting the ground plane luminance

by its grey value (which is the `-g` option of `gensky`):

$$\begin{aligned}(r,g,b) &= \left(\frac{r}{\rho},\frac{g}{\rho},\frac{b}{\rho}\right)\\ (r,g,b) &= \left(\frac{0.4}{0.3135},\frac{0.3}{0.3135},\frac{0.1}{0.3135}\right)\\ (r,g,b) &= (1.28, 0.957, 0.319)\end{aligned}$$

It is important to note that the weighted average of those glow values is 1.0. This must always be the case for the ground and sky to assure correct luminances.

$$\begin{aligned}\rho &= 0.265r + 0.67g + 0.065b\\ \rho &= 0.265\times 1.28 + 0.67\times 0.957 + 0.065\times 0.319\\ \rho &= 1.00\end{aligned}$$

While the image on the left hand-side of Fig 40 was rendered with `-g 0.3131` as suggested in the Radiance digest, the one on the right has been hand-tweaked to make the transition from ground plane to `glow` as invisible as possible. This is, however, rather time consuming and usually not justified. If the ground plane is large enough, the formulae above should give a ground plane colour that makes it indistinguishable from the ground glow of the lower sky.

A graph like the one in Fig 41 will allow you to estimate the required size of ground plane. It largely depends on the dimensions of your model. In the example above, a disc with an outer radius of 40 m was used while the images above utilise a 10 m disc. Our model has the following dimensions:

```
$ getbbox house.rad
     xmin     xmax     ymin     ymax     zmin     zmax
     -0.3      6.3     -0.3     10.3        0      8.3
```

40 m is probably too large a dimension and should be reduced. But remember that reducing the size of the ground plane is always a trade-off between the size of ambient splotches and the visibility of the plane.

3.3 Daylight Factors

3.3.1 Daylight Factors with `falsecolor`

False colour images with DF lines are quite straight-forward to produce. It's much easier if the global horizontal illuminance as produced by the sky (remember: no sun!) is known. The best way to ensure is to get `gensky` to create a sky that produces, say, 10,000 lx. Since there is no option to `gensky` which would take this value directly, we need to define the zenith brightness instead. Using formula 4 in the Radiance Tutorial [10], we get

$$R_{zenith} = \frac{9}{7} \times \frac{10000 lx}{179 ^{lm}/_{W} \pi} = 22.86 \frac{W}{m^2}$$

This value is given to `gensky` with the `-b` option.

To check, find the line

```
# Ground ambient level: 17.8
```

in the output of `gensky`. Take this radiance and multiply by 179 $^{lm}/_{W}$ (luminous efficacy of daylight as used by Radiance) and by π to get the illuminance. This should give you 10,000 lx. Save your sky file as *oc_sky.mat*.

Now create your luminance and illuminance images with `rpict`. Name them *oc.hdr* and *oc_i.hdr*, respectively.

We are now ready to call `falsecolor`. The trick is to use the `-m` option for defining a multiplier. The default of 179 (luminous efficacy) converts from radiance or irradiance to luminance or illuminance, respectively. The illuminance picture *oc_i.hdr* shows lux values. Since the DF is defined as the internal horizontal illuminance (on the working plane), divided by the external unobstructed horizontal illuminance (which we defined as 10,000 lx), the multiplier needs to be 179 × 100%.

```
$ falsecolor -p oc.hdr -i oc_i.hdr -l 'DF(%)' \
   -e -m 1.79 -n 10 -s 20 > oc_fc.hdr
```

The resulting image is shown in Fig 42.

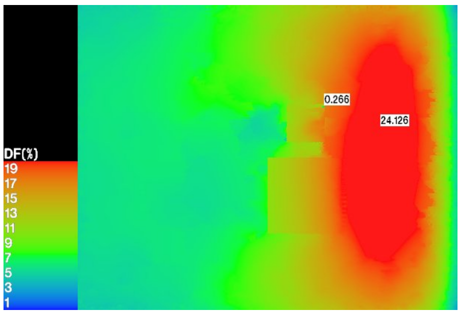


Figure 42: False colour image with daylight factors

3.3.2 Daylight Factors with `rtrace`

This is very similar to the false colour daylight factor images from above, and the illuminance readings from 5.2.1 in Part One.

To take, for instance, a DF reading for the centre of the test room at working plane height, run the following command line:

```
$ echo "2 2.5 0.85 0 0 1" \
   | rtrace -h -w -ab 2 -I -ad 512 -as 128 oc.oct \
   | rcalc -e '$1=179*(.265*$1+.67*$2+.065*$3)/10000*100'
14.1982568
```

This feeds the location and orientation of the lux meter to `rtrace` which then takes a reading of RGB irradiance. Within the parenthesis, the weighted average is taken. Multiplied by 179 lm/W gives the illuminance. Knowing that our sky produces 10,000 lx horizontally, we divide by this value and multiply by 100 to get a percentage. We may now use `bgraph` or any spread sheet to plot illuminance graphs.

3.3.3 Using the `dayfact` Script

`dayfact` is a shell script which walks you interactively through a number of questions. Based on an octree which needs to exist, it renders a Radiance image for the working plane and produces false colour images for the daylight factor, illuminance and potential daylight savings.

An example session is listed below:

```
$ dayfact
                         DAYLIGHT FACTOR CALCULATION

This script calculates daylight factors, illuminance levels, and/or
energy savings due to daylight on a rectangular workplane and
produces a contour plot from the result.  The input is a Radiance
scene description (and octree) and the output is one or more color
Radiance pictures.

Have you already calculated an illuminance picture using dayfact?
Enter illumpic [none]:
Enter octree [none]: oc.oct
Is the z-axis your zenith direction? y
What is the origin (smallest x y z coordinates) of the workplane?
```

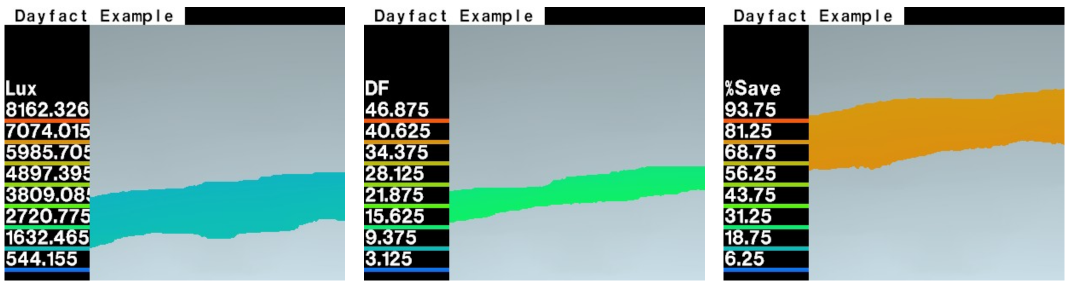


Figure 43: Results from the `dayfact` script

```
Enter wporig [0 0 0]: 1 1.5 0.85
What is the x and y size (width and length) of the workplane?
Enter wpsize [1 1]:
What calculation options do you want to give to rtrace? (It is
very important to set the -a* options correctly.)
Enter rtargs [-ab 1 -ad 256 -as 128 -aa .15 -av .3 .3 .3 -ar 40]:
 -ab 3 -ad 256 -as 128 -aa .15 -av 0 0 0 -ar 40
Do you want to save the illuminance picture for later runs?
Enter illumpic [none]: dayfact_i.hdr
In what scene file is the gensky command located?
Enter genskyf [none]: oc_sky.mat
# gensky 03 23 12 -b 22.86 -c
# gensky 03 23 12 -b 22.86 -c
Illuminance contour picture if you want one Enter ilpict [none]:
 dayfact_i_fc.hdr
Daylight factor contour picture if you want one Enter dfpict [none]:
 dayfact_df_fc.hdr
Energy savings contour picture if you want one Enter dspict [none]:
 dayfact_savings_fc.hdr
Workplane design level (lux)
Enter designlvl [500]:
Title for output picture
Enter title [oc 03 23 12]: Dayfact Example
[1] 25138 25142 Your job is started in the background.
You will be notified by mail when it is done.
```

The output is shown in the images in Fig 43:

3.4 Time of Day Image Sequence

The command-line interface to the operating system is called a shell. In a X Windows session, the shell is accessed through a terminal window such as `xterm` or `konsole`. Most if not all Linux distributions default to BASH (Bourne Again SHell).

BASH has many advanced features to help us get a job done quickly. Here are two that you will probably use all the time. Both are related to the command history which keeps track of what we type at the shell prompt:

- Hit the up and down arrows to move back and forward in your command history;
- Type CTRL-r to search the command history.

Another powerful feature is BASH's scriptability. Many built-in commands allow us to automate jobs. This is somewhat similar to batch jobs under MS Windows, but has a lot more features.

We are going to create a number of images 1 hr time-steps generating one `rpict` image per step. The idea is to investigate into sun patches and how they move throughout the day.

1. Compile *course.mat*, *room.rad* and *furniture.rad* into an octree. Name it *script.oct*.
2. Create the directories *temp* and *images*.
3. If your view file is not called *nice.vf*, please change this in *time_of_day.sh* which can be found in appendix A.2
4. Check the settings for the ambient parameters

With these preparations, we are now ready to run the shell script. If the file has executable permissions for yourself, you may run it with:

```
$ ./time_of_day.bash
```

Otherwise, please type

```
$ bash time_of_day.bash
```

The script will now create an image for every daylight hour of the day. It will also take care of converting the Radiance image to a JPEG which can be opened by any image editor and put up on the web.

Although it is not part of Radiance, ImageMagick has many interesting option to manipulate images. It is installed on LEARNIX, and can be downloaded for most UNIXes. In true UNIX style, ImageMagick is not a monolithic program, rather a collection of smaller ones:

`animate`	animate a sequence of images
`composite`	composite images together
`conjure`	process a Magick Scripting Language (MSL) script
`convert`	convert an image or sequence of images
`display`	display an image on any workstation running X
`identify`	describe an image or image sequence
`import`	capture some or all of an X server screen and save the image to a file.
`mogrify`	transform an image or sequence of images
`montage`	create a composite image by combining several separate images

The *time_of_day.bash* has already taken care of converting all Radiance files into JPEGs. What is left to do is to show an animation of all images over a day. `animate` is just the one we're looking for:

```
$ animate -delay 30 *.jpg
```

Since animations can not be reproduced in a printed document such as this one, we need to find another solution. Although what we are doing now can also be achieved with ImageMagick commands, we use Radiance's `pcompos` which provides similar flexibility.

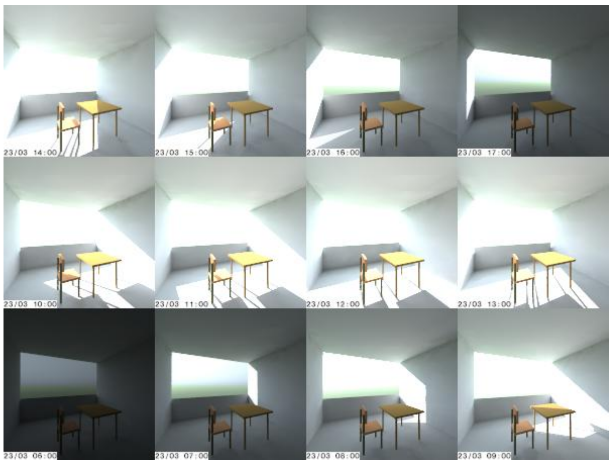


Figure 44: Time-of-day image sequence

```
$ pcompos -a 4 ??.hdr |pfilt -x /2 -y /2 -1 -e 1 \
   | ra_tiff - sequence_all.tif
$ convert -quality 85 sequence_all.tif sequence_all.jpg
```

The image sequence is depicted in Fig 44.

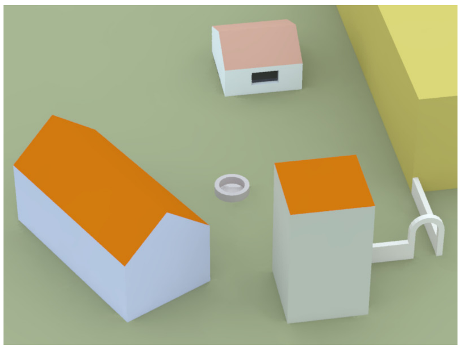


Figure 45: Simple scene for testing different fisheye projections. The view is from the sill of the window of the smaller building.

4 Visualisation

The Radiance `rpict` and `rvu` rendering command offers quite a selection of different views or projections. They are selected with the `-vt` (as in 'view type') option:

v Perspective view. This is what you would get from a camera.

l Parallel view. What's used in architectural drawings.

c Cylindrical panorama. This view is like a standard perspective vertically, but projected on a cylinder horizontally (like a soupcan's-eye view).

h Hemispherical fisheye view. A hemispherical fisheye is a projection of the hemisphere onto a circle. The maximum view angle for this type is 180°.

a Angular fisheye distortion. An angular fisheye view is defined such that distance from the centre of the image is proportional to the angle from the central view direction. An angular fisheye can display a full 360°.

s Planispheric fisheye. This projection is also known a 'stereographic' projection. When using this alternative term, be careful not to confuse it with a stereoscopic pair of images, which, when presented appropriately, may give to the impression of a looking at a real 3D scene. Stereographic projections are often used in sun path diagrams, to test the overshadowing caused by obstructions. This projection was added to Radiance in version 3.9, in Spring 2008.

4.1 Fisheye Views

To see what the difference between the three different fisheye projections is, let's create a simple scene, like the one shown in Fig 45. We will put our virtual camera on the window sill of the small building at the top centre of the image. Additionally, the altitude and azimuth lines are mapped onto the sky hemisphere. You may find the files and instructions for this in the accompanying ZIP archive.

In general terms, all Radiance fisheye views try to map the hemisphere onto a plane. Since this is not possible without distorting the image, it is advantageous to know what the result will be, and where the highest distortion will be in the image. The three fisheye projections have in common that they are azimuthal projections. This means that the azimuth lines are preserved, and only the altitude is distorted. In other words,

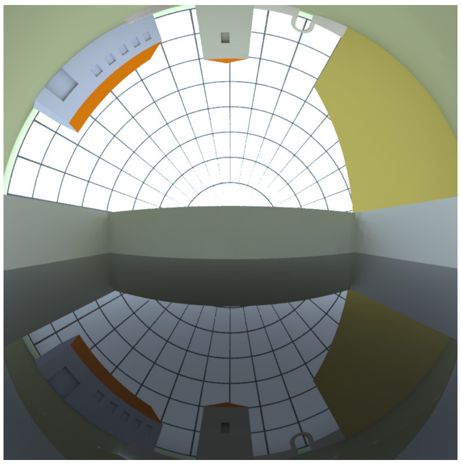


Figure 46: Equiangular fisheye projection

the direction of the reference point at the zenith to other points is preserved, but not necessarily the distance [15].

4.1.1 Equiangular Projection

The equiangular projection (`-vta`) is the simplest of the three. The altitude lines in the image are evenly spaced. The distortion is the same at the centre of the projected view as it is around the edge.

Although optical systems can be constructed for a number of different projections, most common camera lenses, such as the popular Nikon FC-E8 fisheye lens, produce a roughly equiangular image.

4.1.2 Hemispherical Projection

This projection (`-vth`), which is also known as azimuthal orthographic projection, shows the hemisphere as it would appear from a view point infinitely far away. This results in a very high distortion around the edges of the image: Objects far away from the view direction appear very short, and may not even show up in the image.

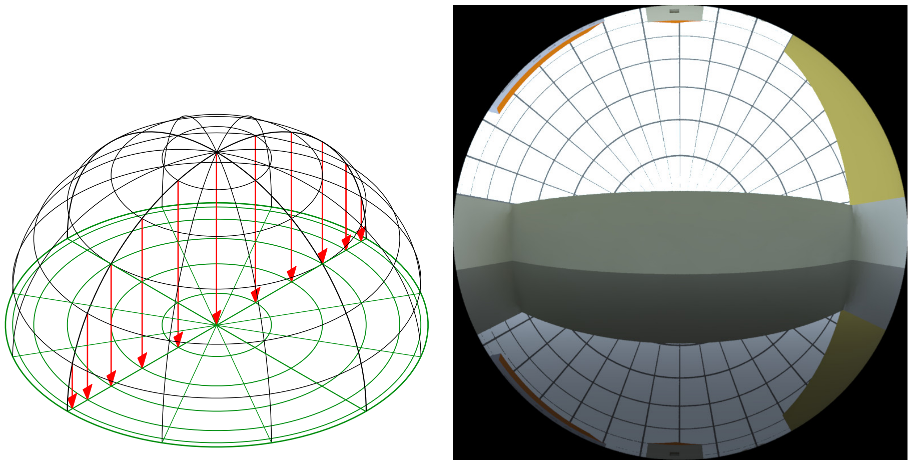
The hemispherical projection is not defined for angles greater than 90°, which is why the corners of the image are always black.

4.1.3 Stereographic Projection

The planispheric or stereographic projection (`-vts`) is constructed by projecting the altitude onto a plane tangential to the zenith. The vantage point of the construction lines is at the nadir, i.e. the point on the sphere that is opposite to the zenith overhead.

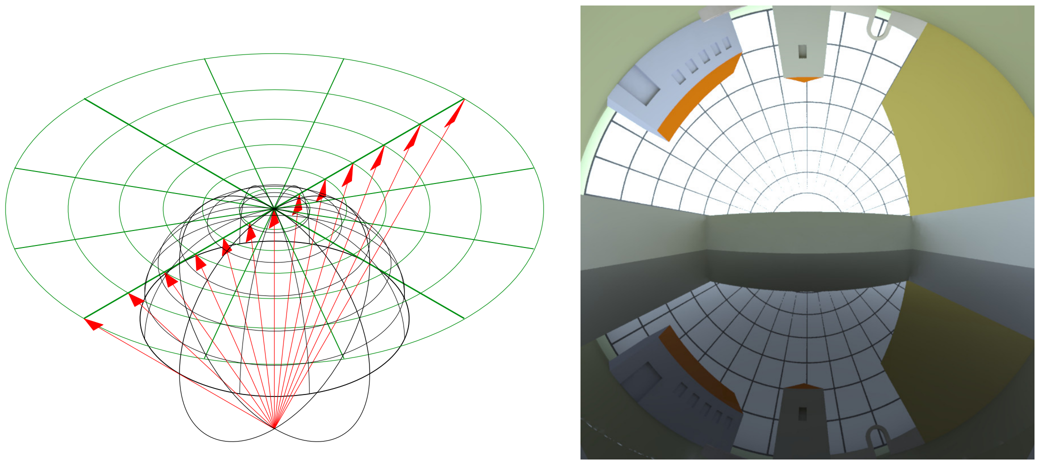
The stereographic projection is often used in combination with sun path diagrams to assess sun light availability and overshadowing from external obstructions with regards to a particular window. It has the advantage of amplifying the height of objects located near the edge of the image. If the view is rendered so that the zenith is in view direction, as is the case with sun path diagrams, a higher angular resolution around the horizon, i.e. where the neighbouring buildings are located, is the result.

Daylight researchers like to use the stereographic projection for sun path and shading analysis because of its special properties. This is why we created a web application for



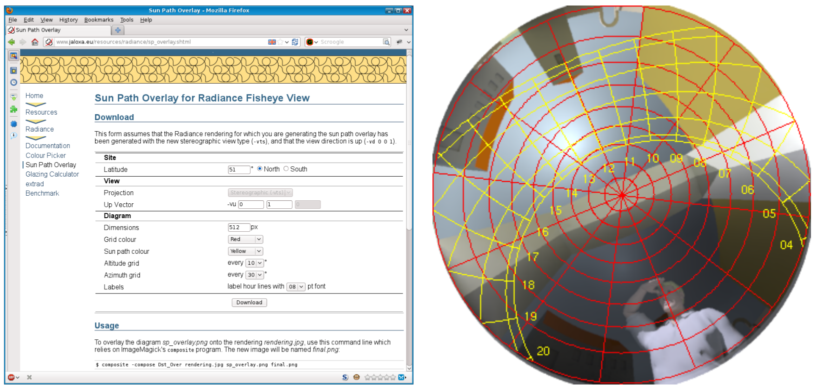
(a) Construction (b) Result

Figure 47: Hemispherical fisheye projection



(a) Construction (b) Result

Figure 48: Planispheric (stereographic) fisheye projection



Sun Path Overlay for Radiance Fisheye View

Download

This form assumes that the Radiance rendering for which you are generating the sun path overlay has been generated with the new stereographic view type (`-vts`), and that the view direction is up (`-vd 0 0 1`).

Site	
Latitude	51 ° [✓ North] South
View	
Projection	Stereographic (-vts)
Up Vector	-vu 0 1 0
Diagram	
Dimensions	512 px
Grid colour	Red
Sun path colour	Yellow
Altitude grid	every 10 °
Azimuth grid	every 30 °
Labels	label hour lines with 08 pt font

Download

Usage

To overlay the diagram *sp_overlay.png* onto the rendering *rendering.jpg*, use this command line which relies on ImageMagick's `composite` program. The new image will be named *final.png*:

```
$ composite -compose Dst_Over rendering.jpg sp_overlay.png final.png
```

Figure 49: Screenshot of the JALOXA web form for creating sun path overlays, and a Radiance rendering with the overlay applied.

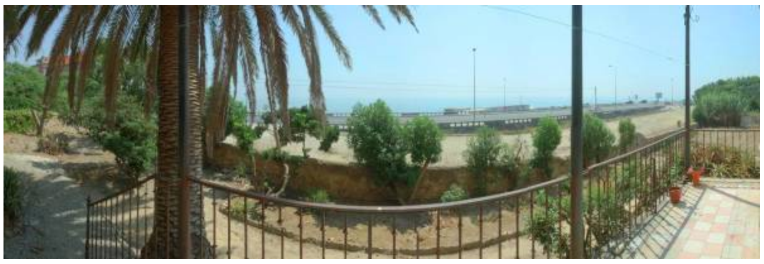


Figure 50: HDR panorama image

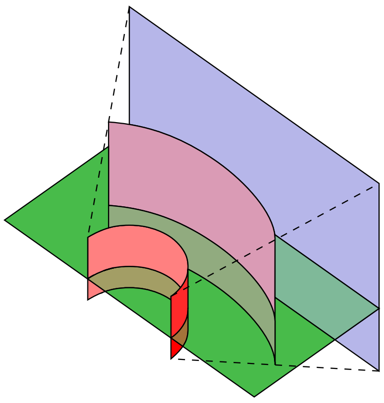


Figure 51: Projecting a cylinder (red) onto a plane (blue). The green plane is the horizontal.

creating them. Instructions for use are on the JALOXA web site Axel Jacobs [8]. Fig 49 shows the form, and the final result.

4.2 Panoramic Projection

A number of image processing applications not only allows us to retouch digital camera images, such as remove red eyes or adjust brightness and contrast, but can be used to create and correct distortions, or even transform a normal camera view into other projections. An example would be panorama images, such as the ones produced by Hugin [5]. The image in Fig 50 is a HDR panorama taken with an ordinary digital camera, and post-processed in Hugin.

The image is a superposition of two rows of six images each times three different exposures. It is a cylindrical projection. We'll try to re-create this projection with a Radiance image. The model we're using was not created to look as realistic as possible, so don't be disappointed if we only get the view right, but not the objects and materials, too.

This particular exercise could be accomplished with the normal cylindrical view in `rpict/rvu` using the `-vtc` option, but where would be the fun in that?

The cylindrical projection maps a cylinder or part of a cylinder (red object) onto a plane (blue rectangle). The purple object in Fig 51 is shown as an intermediate step to make it a bit less confusing, but has no relevance for any of our calculations.

We use Raphael Compagnon's 1997 LEARN course notes as a basis [29]. They are available from the official Radiance site Lawrence Berkeley National Laboratory [24]. In those notes, Raphael explains how to take a cylindrical picture of the whole sky. For this

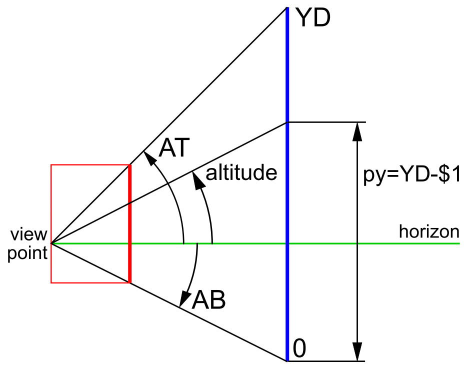


Figure 52: Mapping the altitude to image pixels

to work, he had to use `rtrace` instead of `rpict`. This means that the command itself needs to take care of the rays for each single pixel. This isn't too difficult. However, there are two things in particular that we need to modify in order to match this shot:

1. Raphael's intention was to map the sky hemisphere onto a cylinder surface. The upper altitude was set to 90°. This is fine for a full sky, but not for our photograph.
2. The azimuth was fixed to a full 360°. We only want 180°.

Because the x and y-dimension of the resulting image needs to be specified three times, we export those as shell variables. This means that they are available for all subsequent commands and scripts that are run in the same terminal. If you would like to over-sample the image and then filter it down to improve upon the image quality, you may want to double those figures and then use `pfilt` with `-x /2` and `-y /2`.

```
$ export X=600
$ export Y=200
```

We are already familiar with the Radiance `cnt` command, as well as with `rcalc`. Its `-f` option allows us to pass the name of a file with algorithms for `rcalc` to use. The full listing of *pcyl.cal* is given in appendix A.7. Fig 52 will help you understand how the formula for the altitude (Eqn 5) is derived.

$$altitude = AB + py\frac{AT - AB}{YD} \tag{5}$$

with $py = YD - \$1$. The last term in line 27 of the listing, $\pi/180$ is simply there to convert from degrees to radians.

The azimuth is actually rather similar. Fig 53 should explain enough for you to understand what's going on in eqn. 6

$$azimuth = \left(px - \frac{XD}{2}\right)\frac{180}{XD} \tag{6}$$

With the altitude and azimuth sorted out, what is required next is to convert them into a direction vector. You will remember that `rtrace` requires the input of the ray's origin and direction.

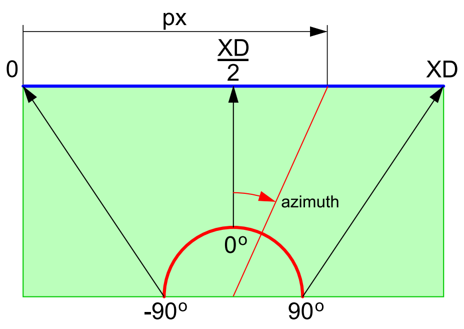


Figure 53: Mapping the azimuth to image pixels

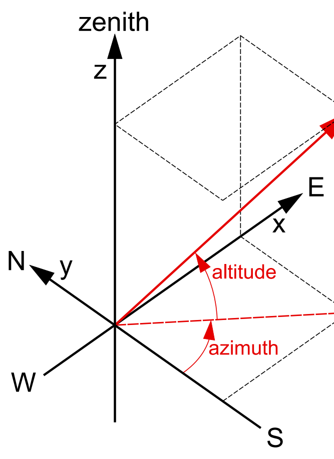


Figure 54: Converting (altitude, azimuth) pairs to a direction vector

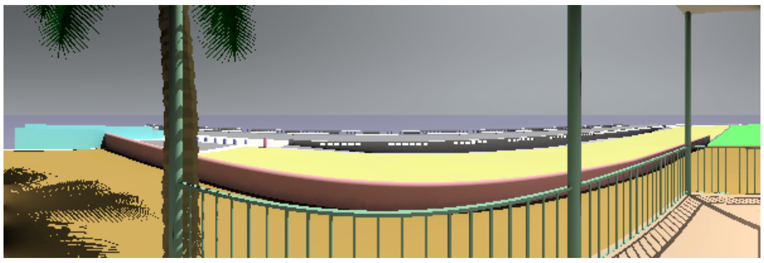


Figure 55: Cylindrical Radiance view matching the photograph from Fig 50

$$\begin{aligned}dx &= -\sin azimuth \times \cos altitude\\ dy &= -\cos azimuth \times \cos altitude\\ dz &= \sin altitude\end{aligned}$$

We should now understand what the strange looking *cal* file does. Line 43 defines the view point, and line 44 the view direction that is computed in lines 24 to 37.

```
$ cnt $Y $X | rcalc -f pcyl.cal -e \
    "XD=$X;YD=$Y;AB=-34;AT=26" | rtrace -ab 1 \
    -af p.amp -x $X -y $Y -fac parede.oct > pcyl.hdr
```

To get the view point and the bottom and top altitude right required a bit of fiddling and some patience, but it worked out all right, as can be seen in Fig 55.

We were fortunate with our model in the sense that the veranda faced exactly South, which is zero azimuth in Radiance. Had this not been the case, we would have needed an extra term in the equation at line 28 like this: $azimuth = ((px - XD/2 - OF) \times 180/XD)$. The parameter OF would then have to be defined on the `rcalc` command line, e.g. OF=90 for a view East.

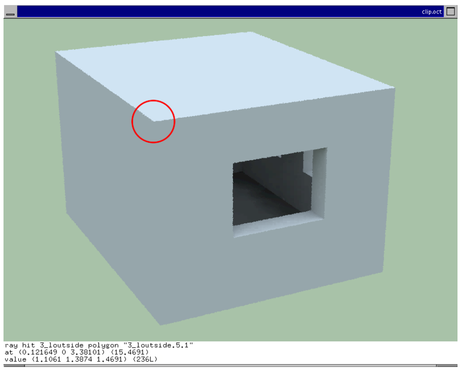
4.3 Using Clipping Planes

The `rvu` and `rpict` commands allow for a great deal of flexibility when it comes to defining views. Several view types are available, each with the usual large number of options which we are used to in Radiance. The man page to `rpict` has all the gory details. The image below is a simple room looked at from the outside.

The relatively small window doesn't allow us to see much of what goes on in the inside. Luckily enough, the `-vo` and `-va` options to `rpict` allow us to specify fore and aft clipping planes. They basically cut off everything that is in front (`-vo`) or behind (`-va`) a certain plane which is defined by its distance from the view point (camera point). To get a first starting point, simply use the `:trace` command interactively at the `rvu` prompt and get the distance to the nearest corner of the box indicated Fig 56 by the red ring.

Since the output might be difficult to read in the screen shot above, it is repeated below:

```
:trace
Pick ray
```



```
ray hit 3_loutside polygon "3_loutside.5.1"
at (0.121649 0 3.38101) (15.4691)
value (1.1061 1.3874 1.4691) (236L)
```

Figure 56: Finding the object nearest to the view point

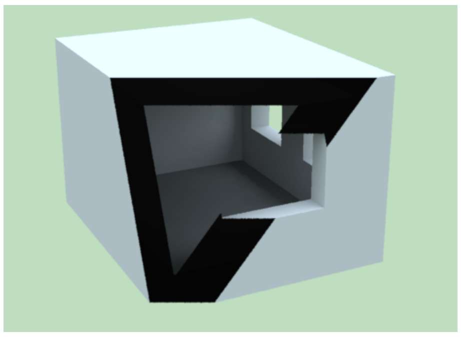


Figure 57: Using a fore clipping plane

```
ray hit 3_loutside polygon "3_loutside.5.1"
at (0.0416403 0 3.46688) (15.4153)
value (1.1067 1.3882 1.4695) (236L)
```

The line starting with 'at' returns the intersection with the nearest object in (x, y, z) coordinates and, more importantly for us now, the distance to this point. Knowing that the nearest corner is 15.41 m away from us, let's try a fore clipping plane of 17.20 m:

```
:view
view type (v):
view point (-5.33101 -13.7018 8.05233):
view direction (0.414116 0.836599 -0.358624):
view up (0 0 1):
view horiz and vert size (28.29 20.4975):
fore and aft clipping plane (0 0): 17.2 0
view shift and lift (0 0)
```

This view, shown in Fig 57, allows us to look inside the scene without letting in any additional light. The opening only exists for the observer, but not for the light. Although this might sound a little strange, it's perfectly normal mathematically. Don't worry about it.

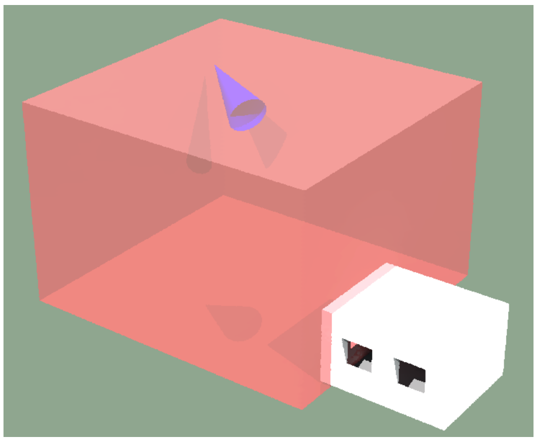


Figure 58: Clipping box

4.4 Creating Arbitrary Clipping Planes

It might sometimes be desired to have a clipping plane that is not perpendicular to the view direction[4]. This can be achieved with a little bit more footwork [20].

One of the most overlooked commands in the Radiance toolbox is `vwrays`. According to the man page, it

> "takes a picture or view specification and computes the ray origin and direction corresponding to each pixel in the image"

This means that it can produce a list of (ray origin, ray direction) values that allows us to not only do the usual ray tracing, but at the same time carry out something else for each pixel individually. If this is all a bit too much now, just keep reading. It will become clearer as we go along.

Let's re-visit the last exercise. The fore clipping plane allowed us to peek through the wall, but didn't look particularly nice. The result would be much nicer if the clipping plane was just inside the room but parallel to the wall, not perpendicular to the view. The blue cone in Fig 58 symbolises the camera. The semi-transparent red box is our clipping box. It's somewhat hard to see, but the red box penetrates the room and has one side just inside the room.

For this process to work, we need to generate two octrees:

First octree: Only the red clipping box. I called mine *1st.oct*;

Second octree: The 'normal' geometry like in the last exercise. Named *2nd.oct*

What happens next is a two-step process with a different octree for each of the stages (compare Fig 59).

Stage one: The ray tracing process is started normally. Instead of relying on `rpict` to generate the rays, we call `vwrays`. Remember that the only object in octree 1 is the clipping box. There is no other object in this scene. Rays originate from the camera (green dot) and travel until they hit the red plane. This is the green line in the drawing above.

[4]Please note that for fish-eye views, the clipping plane becomes a clipping sphere which is perpendicular to each of the view rays, not just to the view direction.

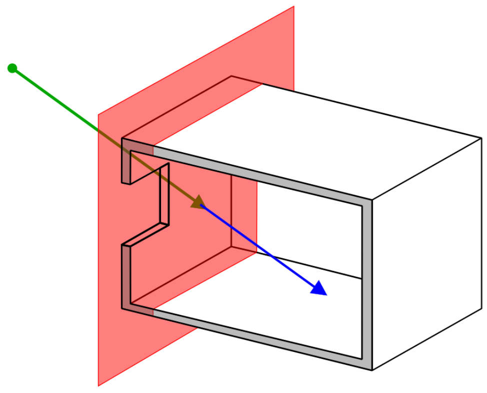


Figure 59: The two-stage process for creating arbitrary clipping planes.

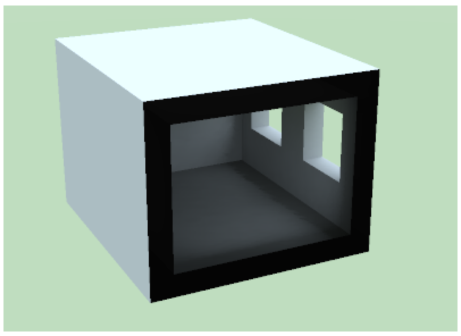


Figure 60: Clipping off the front wall.

Stage two: At the intersection of the first ray with the clipping box, a new ray is spawned (the blue one) travelling in the same direction as the first ray. It is important to note that this second ray is not obstructed by the wall nearest to us since the clipping box lies just inside the room. This second ray therefore has an unobstructed view of the interior of the room.

This is the command line:

```
$ vwrays -fd -vf outside.vf -x 400 -y 400 \
    | rtrace -w -h -fd -opd 1st.oct | rtrace -w -ab 2 \
    -fdc ‘vwrays -d -vf outside.vf -x 400 -y 400‘ 2nd.oct \
    > clipped.hdr; ximage clipped.hdr
```

The result as shown in Fig 60 is just what we expected to get: The wall nearest to us is surgically cut off, laying open the inside of the room to our prying eyes. But let’s examine that long command line in more detail.

```
$ vwrays -fa -vf outside.vf -x 400 -y 400 |wc -l
114800
$ getinfo -d clipped.hdr
clipped.hdr: -Y 287 +X 400
```

`vwrays` takes a view file and produces the rays for each individual pixel for a given image dimension on its STDOUT. In our case, a total of $400 \times 287 = 114800$ rays are computed.

In the original command line, `vwrays` was called with the `-fd` option to create double precision floating point numbers. They are are a more efficient and accurate number format than the default ASCII, but can not be displayed in our terminal. For this discussion, plain text format, ASCII, is piped between programs. The -f options are therefore different to the original ones.

```
$ cat outside.vf
rvu -vtv -vp -5.33101 -13.7018 8.05233 \
    -vd 0.414116 0.836599 -0.358624 \
    -vu 0 0 1 -vh 28.29 -vv 20.4975 -vo 0 -va 0 -vs 0 -vl 0
$ vwrays -fa -vf outside.vf -x 400 -y 400 |head -2
-5.331e+00 -1.370e+01 8.052e+00 2.077e-01 9.611e-01 -1.819e-01
-5.331e+00 -1.370e+01 8.052e+00 2.089e-01 9.608e-01 -1.819e-01
$ vwrays -fa -vf outside.vf -x 400 -y 400 |tail -2
-5.331e+00 -1.370e+01 8.052e+00 5.825e-01 6.380e-01 -5.034e-01
-5.331e+00 -1.370e+01 8.052e+00 5.834e-01 6.373e-01 -5.032e-01
```

The output from `vwrays` are two triplets, one for the (x,y,z) view point, the other for the (x,y,z) view direction. The view point remains the same for all pixels since a perspective view is requested[5].

```
$ vwrays -fd -vf outside.vf -x 400 -y 400 \
    | rtrace -w -h -fda -opd 1st.oct
-2.258e+00 5.100e-01 5.362e+00 2.077e-01 9.611e-01 -1.819e-01
-2.241e+00 5.100e-01 5.360e+00 2.089e-01 9.608e-01 -1.819e-01
...
5.144e+00 -2.228e+00 -1.000e+00 5.825e-01 6.380e-01 -5.034e-01
5.163e+00 -2.237e+00 -1.000e+00 5.834e-01 6.373e-01 -5.032e-01
```

The first invocation of `rtrace` computes the point of intersection (`-op` option) and passes on the ray direction which it has received from `vwrays` to its output (`-od` option). This is the intersection with the red clipping box, not with the room geometry. Remember that *1st.oct* has only the clipping box in it and nothing else.

```
$ vwrays -fd -vf outside.vf -x 400 -y 400 \
    | rtrace -w -h -fd -opd 1st.oct \
    | rtrace -ab 1 -fda -x 400 -y 287 2nd.oct
#?Radiance
oconv room.mat room.rad overcast_sky.mat coloured_sky.rad
rtrace -w -ab 1 -fda -x 400 -y 287 -ld-
SOFTWARE= Radiance 3.6.1 patch release 27 Oct 2004 by G Ward
CAPDATE= 2005:03:02 15:40:54
FORMAT=ascii
```

[5]If we were to use a parallel view, it would be the view point which varies, while the view direction remains constant.

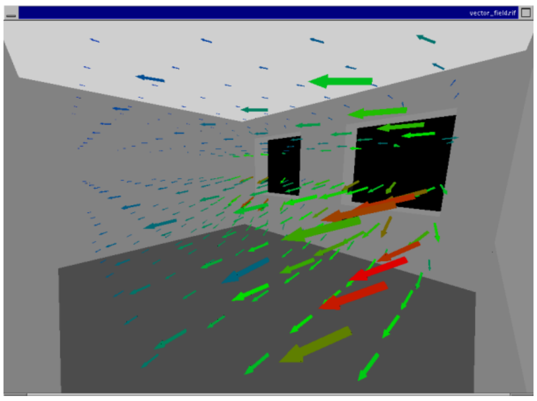


Figure 61: Screen shot of `glrad` in action

```
-Y 287 +X 400
 1.427633e+00   1.962996e+00   1.427633e+00
 1.427612e+00   1.962966e+00   1.427612e+00
 ...
 1.415998e+00   1.946998e+00   1.415998e+00
 1.415998e+00   1.946998e+00   1.415998e+00
```

Called with the second octree, `rtrace` creates a view with rays that emerge just inside the room, completely ignoring the wall. With the `-x` and `-y` options, `rtrace` can create a Radiance image file for us. If the exact image dimensions are not know, which is usually the case if the picture is not square, `vwrays` may be called with the `-d` switch to figure them out for us.

```
 $ vwrays -d -vf outside.vf -x 400 -y 400
-x 400 -y 287 -ld-
 $ vwrays -fd -vf outside.vf -x 400 -y 400 |rtrace -w -h \
     -fd -opd 1st.oct | rtrace -ab 2 -fdc \
     `vwrays -d -vf outside.vf -x 400 -y 400` 2nd.oct |ximage
```

4.5 glrad

`glrad` is part of the standard distribution. It renders a Radiance scene description in OpenGL. This makes it useful for interactively moving around within a scene. Note that the lighting is provided by the OpenGL system and is not correct for simulations, so `glrad` is only useful for looking at the geometry and choosing views. An example screen shot is given in Fig 61.

4.6 The Radiance Holodeck

From the man page of `rholo`:

> `rholo` is a program for generating and viewing holodeck files. Similar to `rvu`, `rholo` can compute views interactively, but unlike `rvu`, it reuses any and all information that was previously computed in this or earlier runs using the given holodeck file, hdkfile.

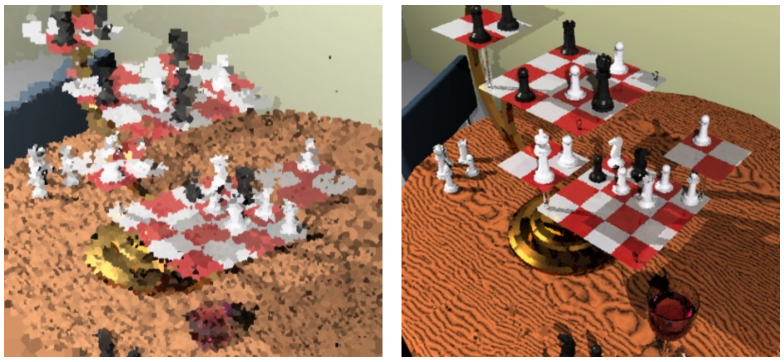


Figure 62: Initial view with `rholo` and image refinement after some time.

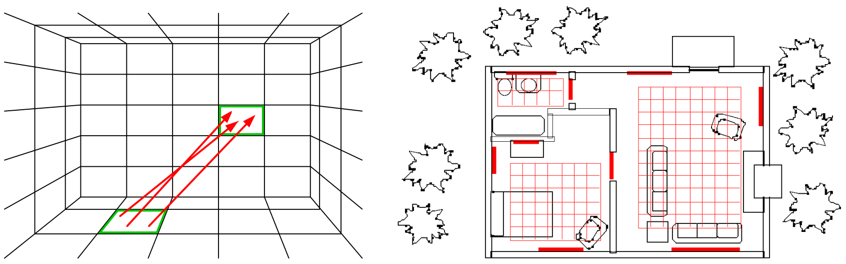


Figure 63: Explaining sections and portals in `rholo`

If this confused you, see if this makes more sense: `rholo` is basically a souped-up version of `rvu` that tries to remedy `rvu`'s biggest shortcoming: For every new view, the entire image has to be regenerated. This takes a while, even on fast computers. `rholo` tries to re-use as many of the rays from the previous view for the new one. Like `rvu`, the image starts rather coarse and may have artifacts, but is refined after a while. This is shown in Fig 62.

All rays that were traced are stored in a cache file, called the 'holodeck file'. Over time and with every new view, rays are added to this file which can become rather large. To speed up the interactive sessions even more, `rholo` may be run without generating a visual output. This makes it possible to populate the holodeck file upfront, e.g. over night.

The holodeck process works best if used in combination with `rad` to control the input parameters. An additional control file for `rholo` needs to be created. This is similar to the rad input file (*.rif*), but instead sets the variables controlling the holodeck generation and is therefore given a *.hif* extension. The only required variable is section. Sections are parallelepipeds[6] and are described by their origin and three axis vectors. They are responsible for the efficient storage and retrieval of rays within the holodeck cache file: All rays passing through the same two cells are stored as a so-called beam. The size and shape of the section, as well as the grid size affect the speed of the the rendering process.

The images in Fig 63 show three rays of the same beam on the left and an example scene with sections and portals in it. Portals are another optimisation allowing for good interactivity of the `rholo` process.

[6] A parallelepiped is a prism whose faces are all parallelograms.

Please read [23] by Greg Ward, available from [3] for more technical details and images regarding the Radiance holodeck, and refer to the rholo(1) man page for details of the hif file format and the use of `rholo`. Greg was also kind enough to contribute the images in this section.

Figure 64: People will get inventive when glare is a problem at their place of work

5 Lighting Analysis

5.1 Glare Calculations

Glare (both, discomfort and disability glare) can cause enormous problems and is probably the number one complaint people will make about a lighting installation. This exercise is based on the model from section 2.1.2 in which an array of spot lights was created.

It's important to address the glare issue before the lighting gets installed, replacing it later may be a very expensive exercise. Radiance allows us to predict glare in a scene with four different commands:

`findglare`	locates sources of glare for a given scene and view
`glarendx`	computes a glare index
`xglaresrc`	displays glare sources in an image
`glare`	interactive script calling `findglare` and `glarendx` (Experiment with this one yourself!)

`findglare` is the actual work horse responsible for the analysis. The default behaviour is to consider objects that are seven times brighter than the average field luminance as sources of glare. This can be overwritten with the `-m` option. If the sources are very small, like in our scene, the resolution may be increased from the default of 150 samples.

```
$ findglare -r 600 -p erco_fisheye.hdr -vf fisheye.vf \
    erco.oct > erco.glr
$ cat erco.glr
#?Radiance
findglare -r 600 -p erco_fisheye.hdr -vf fisheye.vf erco.oct
VIEW= -vth -vp 2 4.5 1 -vd 0 -1 0 -vu 0 0 1 -vh 180 -vv 180 \
    -vo 0 -va 0 -vs 0 -vl 0
FORMAT=ascii
BEGIN glare source
        0.000000 -0.613107 0.790000    0.000364   2039.002649
        0.443333 -0.228789 0.866667    0.000547   11420.742257
       -0.443333 -0.228789 0.866667    0.000499   12390.143008
        0.000000 -0.256038 0.966667    0.000802   129051.375000
```

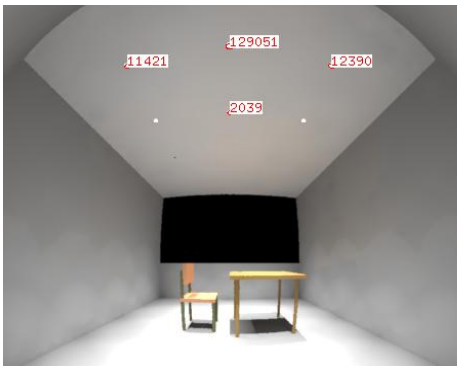


Figure 65: Drawing glare sources onto an image

```
END glare source
BEGIN indirect illuminance
       0       41.565254
END indirect illuminance
```

To compute a glare index such as DGI or UGR from those values, call `glarendx` with the glare file:

```
$ glarendx -h -t ugr erco.glr
0.000000        17.283998
```

Glare sources can be displayed by the `xglaresrc` program. When called with an image file and a glare file, it will display the image with `ximage` and draw circles around glare sources as listed in the glare file and also display the average brightness of the source. The radius of the circles is proportional to the projected area of the source. It might take a few seconds before the lines are drawn onto the image as shown in Fig 5.1.

```
$ xglaresrc erco_fisheye.hdr erco.glr
```

To save this image, you need to take a screen shot.

5.2 Dynamic grids with `cnt`

This is the same exercise as in the Radiance Tutorial in section 'Getting an Illuminance Reading' [Axel Jacobs [10]]. The difference is that this time, we will not read the measurement points from a file, but use `cnt` to generate them on-the fly.

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

```
$ cnt 9
    0
    1
...
    7
    8
```

OK, this is nine numbers now from 0 to 8. `rcalc` will convert them into co-ordinates for us. 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. If this is a bit confusing, just think of the normal notation a formula is written in:

$$A_{out} = A_{in} + B_{in} \tag{7}$$

A_{out} is the value that is being computed, while A_{in} and B_{in} are the input values which must exist for A_{out} to be defined. `rcalc` takes the input and writes the output from/to TAB-separated files, where each column represents one variable and each row represents one set of data. For example, if the input file *input.dat* looks like this:

```
$ cat input.dat
2      5
3      8
1      3
```

Eqn 8 in `rcalc` notation

$$\$1 = \$1 + \$2 \tag{8}$$

is equivalent to Eqn 7, producing the following output:

```
 $ cat input.dat | rcalc -e '$1=$1+$2'
7
11
4
```

The only difference is that `rcalc` applies the given formula to each line of input, producing the same number of output lines. It is also capable of generating more than one output value, in which case they are again separated by TABs. But back to our exercise...

```
$ cnt 9 | rcalc -e '$1=$1/2+.5'
0.5
1
...
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 little bit above the table rather than in exactly the same plane.

```
$ 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       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`. Please proceed from here with section 6.2.1 in the Radiance Tutorial, which also shows you how the result can be conveniently plotted with `bgraph`.

5.3 Illuminance Uniformity

The two most important criteria of a lighting system are the average illuminance and the uniformity. We have dealt with illuminance measurements already and are now looking into determining the uniformity of illuminance.

The CIBSE Code for Interior lighting defines uniformity as the ratio of the minimum illuminance to the average illuminance and suggests that, over the task area it should not be less than 0.8. If a task area can be identified, the illuminance calculations should be on a 0.25 m^2 grid over the task area, otherwise the task area is assumed to span the whole room excluding a 0.5 m band along the walls.

For this exercise, please create a new octree from the array of spot lights and the room. Do not include the sky. Call the file *uniformity.oct*.

Our old friends `cnt` and `rcalc` will help us in setting up the grid input for `rtrace`. We need the position (x, y, z) of the measurement and the direction vector (dx, dy, dz). 221 points are required.

```
$ cnt 13 17 | rcalc -e \
    '$1=$1/4+0.5;$2=$2/4+.5;$3=0.8;$4=0;$5=0;$6=1'
0.5     0.5     0.8     0       0       1
0.5     0.75    0.8     0       0       1
0.5     1       0.8     0       0       1
0.5     1.25    0.8     0       0       1
...
```

In the next step, `rtrace` will take the readings which are converted from irradiance to illuminance with `rcalc`. Since the spot lights emit a pure white light, there is no need to figure out the weighted average across the red, green and blue channel; we just pick any of them.

```
$ cnt 13 17 | rcalc -e \
    '$1=$1/4+0.5;$2=$2/4+.5;$3=0.8;$4=0;$5=0;$6=1' \
    |rtrace -I -h -ov uniformity.oct |rcalc -e '$1=$1*179'
70.5079389
95.9135879
110.181857
125.38379
...
```

We will let `awk` handle the averaging. It will also figure out the smallest value for us. `awk` is a powerful tool for reformatting columnar data and doing calculations on the input stream. Please refer to [13] for reference and tutorials.

```
$ cnt 13 17 | rcalc -e \
    '$1=$1/4+0.5;$2=$2/4+.5;$3=0.8;$4=0;$5=0;$6=1' \
    |rtrace -I -h -ov uniformity.oct |rcalc -e '$1=$1*179' \
    |awk 'BEGIN {min=100000} {if ($1<min) min=$1; \
```

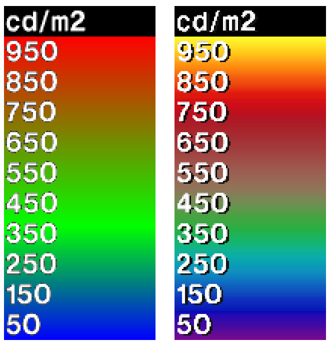


Figure 66: Radiance `falsecolor` mapping before version 3.8 and since.

```
    total+=$1; lines++} END {print min/total*lines}'
0.369748
```

Voila! Measurement, processing and calculations all done in one line. You should also make yourself familiar with the Radiance `total` command which sums up columns, and the UNIX `sort` command for sorting lines of text files. Please refer to the relevant man pages.

5.4 Luminance Along a Line

While the `falsecolor` command maps the luminance distribution of an entire picture to a false colour scale, it is often desirable to only look at the luminance along an arbitrary line in the picture. This can be much more informative, especially when plotted in a graph. To pick up on an earlier section 2.7, we will investigate how the new `falsecolor` scale as introduced in Radiance 3.8 in October 2006 compares to the old one[7][8]. Both scales are shown in Fig 66.

To start with, take any false colour image and extract a one-pixel wide image that is as high as the legend. Unless you changed this with the `-lh` option of `falsecolor`, the legend is 200 px high.

```
$ pcompos -x 1 -y 200 fc_new.hdr 0 0 > scale_new.hdr
```

The next step is to extract the RGB values from the image into a file. Make sure you don't print the header.

```
$ pvalue -h scale_new.hdr | head -3
```

[7]Greg Ward on the Radiance-general mailing list, thread "New falsecolor scale in 3.8", 8 November 2006:

> It occurred to me that the original scale in false color didn't actually traverse that many "named colors," making it more difficult than it needed to be to differentiate values. Having seen a number of color scales, including spectral scales like the original one I used and thermal scales, I decided that some mix of the two might work better. So, I essentially chose colors from a palette moving through the cooler spectral colors, then shifting to a thermal scale in the green-to-red (which became a green-to-brown) transition. This then continues on through red and orange to yellow, avoiding white at the end as I found it to be confusing.

[8]Also see Sec 2.7 for additional colour palettes that were introduced in Radiance 4.1 (Nov 2011).

```
-Y 200 +X 1
    0    199      9.863e-01      9.512e-01      2.539e-02
    0    198      9.902e-01      9.160e-01      2.148e-02
```

Also remove the resolution string from the output.

```
$ pvalue -h -H scale_new.hdr | head -3
    0    199      9.863e-01      9.512e-01      2.539e-02
    0    198      9.902e-01      9.160e-01      2.148e-02
    0    197      9.902e-01      8.809e-01      2.148e-02
```

We need to ensure that the columns are separated by one single tab or space. For some strange reason however, `pvalue` outputs columns which are separated by multiple spaces. To remove those and only leave one space, we make use of the `tr` command. `tr` stands for 'translate'. Running it with the `-s` option will squeeze multiple successive occurrences of the character in question so that only one remains.

```
$ pvalue -h -H scale_new.hdr | tr -s ' ' | head -3
0 199 9.863e-01 9.512e-01 2.539e-02
0 198 9.902e-01 9.160e-01 2.148e-02
0 197 9.902e-01 8.809e-01 2.148e-02
```

From here, there are two options. We will use `cut` to extract individual columns. Trouble is that `cut` assumes those columns are separated by exactly one tab. So we either translate the space to a tab,

```
$ pvalue -h -H scale_new.hdr | tr -s ' ' | tr ' ' '\t' \
   | cut -f 3-6 > rgb_new.dat
```

or we tell `cut` to look out for a space character instead:

```
$ pvalue -h -H scale_new.hdr | tr -s ' ' \
   | cut -d ' ' -f 3-6 > rgb_new.dat
```

Either way is fine with `cut`, and will produce the same result:

```
$ head -3 rgb_new.dat
199     9.863e-01       9.512e-01       2.539e-02
198     9.902e-01       9.160e-01       2.148e-02
197     9.902e-01       8.809e-01       2.148e-02
```

Now let's separate the channels, making sure the first column is in every one of the new files. This first column is the y-position on the false colour legend, or in other words 200 times the normalised image brightness.

```
$ cat rgb_new.dat |cut -f 1,2 > red_new.dat
$ cat rgb_new.dat |cut -f 1,3 > green_new.dat
$ cat rgb_new.dat |cut -f 1,4 > blue_new.dat
```

Just for the sake of comparison, we'll plot the brightness in the same diagram. This makes it a total of four plots which is fortunate for us since this is the maximum `bgraph` can do.

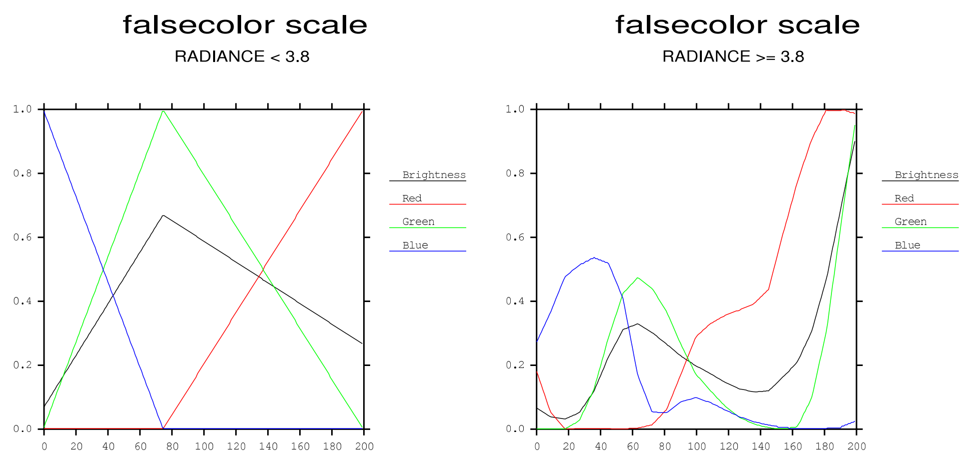


Figure 67: RGB channels for the old and new Radiance `falsecolor` scales

```
$ pvalue -h -H -b scale_new.hdr | tr -s ’ ’ | tr ’ ’ ’\t’ \
    | cut -f 3,4 > brightness_new.dat
```

Create a file named *values_new.braph* with instruction for `bgraph` what to plot and how. We are lucky again: The only four available colours are black, red, green, and blue which are mapped to the numbers from one to four. So we'll pick black for the brightness.

```
title = "falsecolor scale"
subtitle = "Radiance >= 3.8"
Acolor = 1
Adata = brightness_new.dat
Alabel = "Brightness"
Bcolor = 2
Bdata = red_new.dat
Blabel = "Red"
Ccolor = 3
Cdata = green_new.dat
Clabel = "Green"
Dcolor = 4
Ddata = blue_new.dat
Dlabel = "Blue"
```

To do the plotting to the screen or to a file, run:

```
$ bgraph values_new.bgraph | x11meta
$ bgraph values_new.bgraph | psmeta \
    > falsecolor_scale_new.eps
```

All done. The results for Radiance prior version 3.8, and for version 3.8 and higher are shown in Fig 67.

If you are not reading this document front-to-back (and I'm not saying that you should), you might want turn to section 2.7 for a bit more info on the such false colour scales, and how you can create your own.

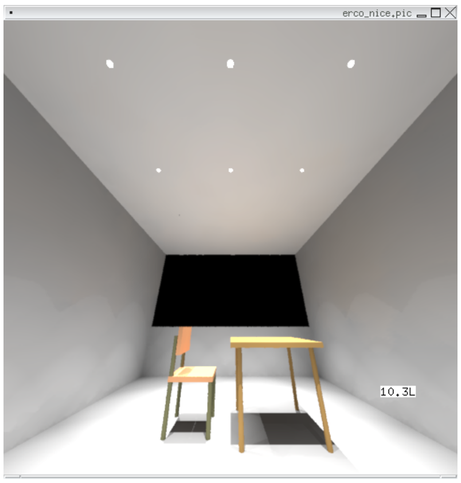


Figure 68: Tracing illuminance with `ximage` and `rlux`

5.5 Computing Illuminance with `rlux`

The `rlux` program is a little wrapper that calls `rtrace` and `rcalc` to compute the illuminance. It requires the ray origin and direction for this. You might remember that this is the default output on `STDOUT` when the t key is pressed in a `ximage` display. We can therefore use `ximage` to generate those values by piping the command line output from `ximage` into `rlux`. Please replace $RTRACEOPTS with appropriate `rtrace` options:

```
$ ximage erco_nice.hdr | rlux $RTRACEOPTS erco.oct
16.7569418
```

This is what `rlux` does internally:

```
rtrace -i+ -dv- -h- -x 1 $RTRACEOPTS \
    | rcalc -e '$1=47.4*$1+120*$2+11.6*$3' -u
```

It might come in handy when a few spot measurements of the illuminance are required, because creating a proper illuminance image (`-i` option to `rpict`, `rvu`) is not necessary.

5.6 Illuminance Values on a Virtual Plane

Towards the end of the BASIC tutorial, we used a combination of `rtrace` and `bgraph` to plot illuminance values at working plane height in a room. Graphs like that serve at least two very important purposes: They allow you analyse the availability of daylight in a space, and they are also very handy for fine-tuning the ambient parameters for a Radiance simulation.

This exercise will take this a step or two further and create a false colour illuminance map for the entire work plane. The `vwrays` command that we already know from section 4.4 will again prove its usefulness for non-standard visualisations. We start by setting up a view file for a parallel view of the floor, from wall to wall. The image in Fig 69 is deliberately done with larger dimensions to show that the view point is over the centre of the room. Make sure that your view covers only the floor, no more and no less, and save it as *horizontal.vf*.

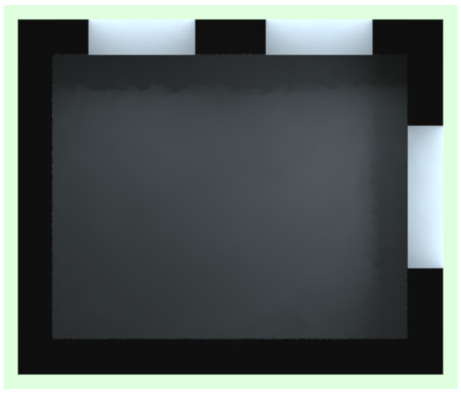


Figure 69: Parallel view of the floor of our little test room

```
$ cat horizontal.vf
rvu -vtl -vp 2.5 3 2 -vd 0 0 -1 -vu 1 0 0 -vh 5 \
    -vv 4 -vo 0 -va 0 -vs 0 -vl 0
```

This view is passed as an argument to `vwrays` which will create a view vector for each pixel of the image. Naturally, it needs to know how large the image should be.

```
$ vwrays -x 400 -y 400 -vf horizontal.vf |head -3
5.1920e+00 6.1920e+00 2.000e+00 0.000e+00 0.000e+00 -1.000e+00
5.1920e+00 6.1760e+00 2.000e+00 0.000e+00 0.000e+00 -1.000e+00
5.1920e+00 6.1600e+00 2.000e+00 0.000e+00 0.000e+00 -1.000e+00
```

You will notice that the direction of those rays is in -z (down towards the floor) which is no good if what we are after is the horizontal illuminance (in +z). Nothing that `rcalc` couldn't fix.

```
$ vwrays -x 400 -y 400 -vf horizontal.vf \
    |rcalc -e '$1=$1;$2=$2;$3=.4;$4=$4;$5=$5;$6=1 |head -3
5.19201  6.192  0.4  0  0  1
5.19201  6.176  0.4  0  0  1
5.19201  6.16   0.4  0  0  1
```

It's now time to get `rtrace` involved to get some readings. Remember to switch into illuminance mode by applying the `-I` option. You will have to set the ambient parameter to more accurate values to get an image quality like below. This is not reflected in the command line. The combination of the `-ov`, `-fac`, `-x` and `-y` options lets `rpict` write out a proper Radiance image file instead of just numbers. Please refer to section 4.4 if you are lost with the last part of the command line.

```
$ vwrays -x 400 -y 400 -vf horizontal.vf \
    | rcalc -e '$1=$1;$2=$2;$3=.4;$4=$4;$5=$5;$6=1 \
    | rtrace -fac -w -I -ov `vwrays -d -vf outside.vf \
    -x 400 -y 400` room.oct > wp_illu.hdr
```

To make the result more visual and understandable, the false colour information can be put back into the image and shown superimposed to the actual scene geometry. This is shown in Fig 70 and dealt with in section 2.6.2.

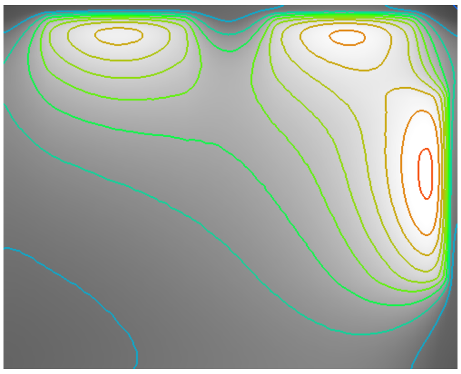


Figure 70: View of the floor with illuminance graph superimposed

5.7 Vector Illuminance

The accurate description of the light in a room is actually more difficult than you might imagine. Most of us will be aware that regulations regarding work place lighting are based on illuminance readings on the working plane. Illuminance is nice and easy to measure, and lux meters are affordable to anybody.

It has been argued, however, that instead of the illuminance, luminance readings should be the basis of such regulations. This makes a lot of sense, because what our eyes perceive is the light reflected off the surfaces of our surrounding, whereas lux readings only tell us how much light falls onto an object. Unless we know the reflectance of this object, an illuminance reading tells us very little.

Another interesting problem is how the light can be described in three-dimensional terms. Defining a working plane is good and well, but how can we know how the objects are rendered, how shadows accentuate their shape and texture, and what the direction of the light is? For those kinds of question, a simple luminance or illuminance measurement in one plane is not sufficient. What is needed is the vector illuminance that describes the intensity of the light, as well as its direction.

The vector or cubic illuminance is somewhat difficult to measure due to the large number of readings required. The only dedicated cubic illuminance meter commercially available seems to come from a company called Megatron in the UK [25]. Luckily for us, Radiance can be used to produce illuminance vectors for us. Here is how.

```
###############################################################
WARNING: This is the most difficult exercise in this document.
A good working knowledge of Radiance on the UNIX command line
is essential for its understanding.
###############################################################
```

!

First of all, we need a 3D grid of all the points of measurement. We'll use a 0.5 m grid for this.

```
$ cnt 7 9 5 | rcalc -e '$1=$1/2+1;$2=$2/2+1;$3=$3/2+.5' \
    > points.dat
$ head -2 points.dat; tail -2 points.dat
1  1  0.5
1  1  1
```

```
4  5  2
4  5  2.5
$ wc -l points.dat
315 points.dat
```

For each point, the illuminance needs to be measured in all 6 major directions. We create a file with just the direction vector in it. The file must have as many lines as *points.dat* above.

```
$ for i in $(cnt 315); do echo -e "1\t0\t0" \
    >> direction+x.dat done
$ head -3 direction+x.dat
1   0   0
1   0   0
1   0   0
```

Please repeat this procedure for the remaining 5 directions (-x, +y, -y, +z, -z). Now it's time to do some number crunching. Make sure the ambient options are set to something sensible. `rtrace` needs the position of the measurement as well as the direction. For this, `rlam` combines the two files and sends them to `rtrace`. Again, repeat this another five times for the remaining directions.

```
$ rlam points.dat direction+x.dat | head -3
1   1   0.5   1   0   0
1   1   1     1   0   0
1   1   1.5   1   0   0
$ rlam points.dat direction+x.dat |rtrace -h -w \
    $AMBOPTS -I -ov room.oct |rcalc -e \
    '$1=179*($1*.265+$2*.67+$3*.065)' > lux+x.dat
```

The next step is to compute the difference of the two illuminance readings along each axis.

```
$ rlam lux+x.dat lux-x.dat |rcalc -e '$1=$1-$2' > x.dat
```

Repeat for y and z. This is the job half-done. But how good are a bunch of numbers when a graphic representation is much easier to understand. We are therefore going to create arrows that reflect the intensity, as well as the direction of the light at each point.

Next, create a little *cal* file:

```
$ cat size.cal
{ Take the squares }
sx=dx*dx;
sy=dy*dy;
sz=dz*dz;
{ Length in 3D }
len=sqrt(sx+sy+sz);
{ Length in xy plane }
len_xy=sqrt(sx+sy);
{ pivot angles up and around z-axis }
roty=atan(dz/len_xy)*180/PI;
rotz=atan(dy/dx)*180/PI+180;
```

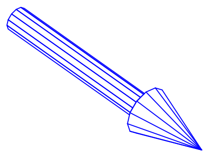


Figure 71: A prototype arrow

The longest arrow should be exactly the spacing of our grid (0.5 m) to ensure none of the arrows will overlap. To find the longest arrow, take the results and process them with `rcalc` using the *size.cal* file above. The `export` command is specific to the BASH shell and stores the MAX variable for later use. Variables are retrieved with the `echo` command and a dollar sign preceding the variable name.

```
$ rlam x.dat y.dat z.dat |rcalc -f size.cal -e \
    'dx=$1;dy=$2;dz=$3;$1=len' > length.dat
$ export MAX=$(cat length.dat \
    |awk '{if ($1>max) max=$1} END {print max}')
$ echo $MAX
1374.09712
$ echo "s(input) = input*$(echo $MAX \
    |rcalc -e '$1=0.5/$1')" > scale.cal
$ cat scale.cal
s(input) = input*0.000363875299
```

The arrows will be scaled-up and rotated copies of this one which is 1 unit long:

```
$ cat arrow.norm
arrow_mat cylinder arrow_base
0
0
7  0 0 0  .7 0 0  .05

arrow_mat cone arrow_head
0
0
8  .7 0 0  1 0 0  .1 0

arrow_mat ring arrow_ring
0
0
8  .7 0 0  -1 0 0  0 .1

arrow_mat ring arrow_end
0
0
8  0 0 0  -1 0 0  0 .05


$ cat arrow_xform.fmt
```

```
!xform -s ${s} -ry ${ry} -rz ${rz} -
t ${x} ${y} ${z} arrow.norm
```

The next, rather long command line scales this norm arrow with the first invocation of `rcalc` and then rotates it in two directions to make it point into the direction of the illuminance vector. The file *arrow_xform.fmt* is a kind of template file where `rcalc` fills in the defined variables for scaling, rotation around the y-axis, rotation around the z-axis and translation. See the man page to `rcalc` for details on such format files.

```
$ rlam points.dat x.dat y.dat z.dat length.dat | head -3
1  1  0.5  221.58811   12.445351   180.961482  286.36207
1  1  1    323.732759  -63.490781  151.584646  363.059063
1  1  1.5  293.528747  -61.114234  60.9687604  305.95958
```

The first part outputs the measurement point (x,y,z), the illuminance along the major axises and the length of the resulting illuminance vector.

```
$ rlam points.dat x.dat y.dat z.dat length.dat |rcalc -f scale.cal \
    -e '$1=$1;$2=$2;$3=$3;$4=s($4);$5=s($5);$6=s($6);$7=s($7)' \
    |head -3
1  1  0.5  0.08063043  0.0045285558  0.065847413  0.10420008
1  1  1    0.11779835  -0.023102726  0.055157908  0.13210822
1  1  1.5  0.10680786  -0.022237960  0.022185025  0.11133113
```

This is normalised to the highest 3D illuminance, before the values are pasted into the format file and appended to *arrows.rad*.

```
$ rlam points.dat x.dat y.dat z.dat length.dat \
    |rcalc -f scale.cal -e \
    '$1=$1;$2=$2;$3=$3;$4=s($4);$5=s($5);$6=s($6);$7=s($7)' \
    |rcalc -f size.cal -o arrow_xform.fmt -e \
    'x=$1;y=$2;z=$3;dx=$4;dy=$5;dz=$6;s=$7;ry=roty;rz=rotz' \
    > arrows.rad
$ head -3 arrows.rad
!xform -s 0.1042 -ry 39.1928 -rz 183.215 -t 1 1 0.5 arrow.norm
!xform -s 0.1321 -ry 24.6782 -rz 168.904 -t 1 1 1   arrow.norm
!xform -s 0.1113 -ry 11.4943 -rz 168.239 -t 1 1 1.5 arrow.norm
```

Well, this looks really nice so far, doesn't it (Fig 72)?

While it was a really good idea to scale the length of the arrows depending on the illuminance, it would be even neater to have them colour-coded like a `falsecolor` image. Each of the arrows will have to have its own material definition, rather than a generic one. For this to work, the file *arrow_xform.fmt* needs a little alteration:

```
$ cat arrow_xform.fmt
!xform -m arrow${m}_mat -s ${s} -ry ${ry} -rz ${rz} \
    -t ${x} ${y} ${z} arrow.norm
```

The `-m` option to `xform` overrides the existing material and assigns a new one to the object that is being transformed. We use a running number starting from 1 to achieve this:

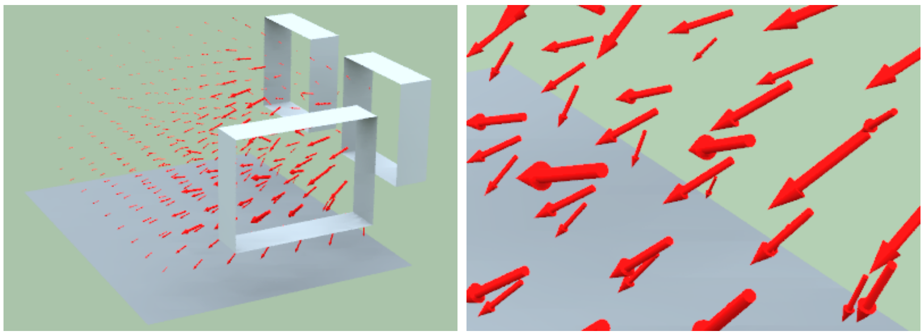


Figure 72: Vector illuminance visualised with arrow heads

```
$ rlam points.dat x.dat y.dat z.dat length.dat |rcalc -f scale.cal \
    -e '$1=$1;$2=$2;$3=$3;$4=s($4);$5=s($5);$6=s($6);$7=s($7)' \
    |rcalc -f size.cal -o arrow_xform.fmt -e \
    'x=$1;y=$2;z=$3;dx=$4;dy=$5;dz=$6;s=$7;ry=roty;rz=rotz;m=outno' \
    > arrows.rad
$ head -3 arrows.rad
!xform -m arrow1_mat -s 0.1042 -ry 39.1928 -rz 183.215 \
    -t 1 1 0.5 arrow.norm
!xform -m arrow2_mat -s 0.1321 -ry 24.6782 -rz 168.904 \
    -t 1 1 1   arrow.norm
!xform -m arrow3_mat -s 0.1113 -ry 11.4943 -rz 168.239 \
    -t 1 1 1.5 arrow.norm
```

Please note the m=outno which we have to append to the `rcalc` line. It outputs the number of records processed so far and does the indexing for us. In addition to the `rcalc` format file for the arrow's geometry, a new one for the material is now required:

```
$ cat arrow_mat.fmt
void plastic arrow${m}_mat
0
0
5 ${r} ${g} ${b} 0 0
```

To do the colour transformations we first need to create an appropriate *.cal* file which we name *colour.cal*:

```
$ cat <<_EndOfColour_ > colour.cal
max=$(echo $MAX);
multi=100/max;
t=in(1)*multi;
red=if(t-50,-1*(50-t)/50,0);
grn=if(t-50,1-((t-50)/50),t/50);
blu=if(50-t,-1*(t-50)/50,0);
_EndOfColour_
```

This structure which is a feature of the BASH shell is called a 'here document'. We use it to dynamically generate the *.cal* file, since we don't know up-front what the

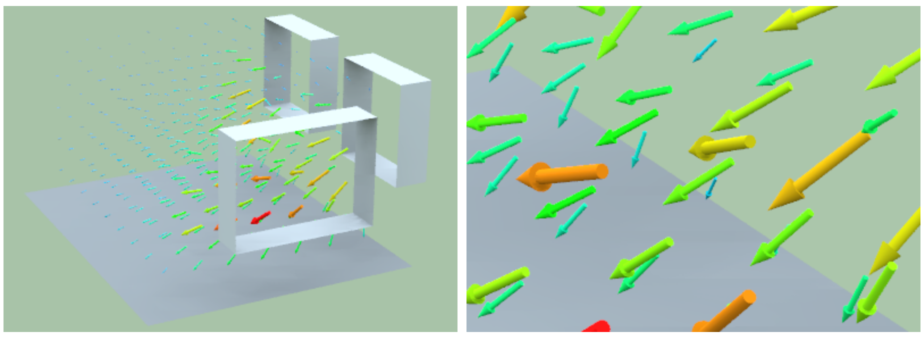


Figure 73: Illuminance arrow-heads with false colour coding applied

value of MAX is. The file could be hand coded, but this approach is better and can be automated in a shell script.

Please take a quick look at the end of section 2.7 to remind yourself how the red, green, and blue channels are computed to use the same coding as a `falsecolor` legend.

The following command creates the material file for the arrow. It writes out one primitive for each of the arrows.

```
$ cat data/len.dat \
   | rcalc -f colour.cal -o arrow_mat.fmt \
   -e 'r=red;g=grn;b=blu;m=outno' > arrows.mat
$ head -2 arrows.mat
void plastic arrow1_mat 0 0 5 0 0.4168 0.5832 0 0
void plastic arrow2_mat 0 0 5 0 0.5284 0.4715 0 0
```

Colour-coded as they are now, the arrows are much easier to interpret. Red indicates a high illuminance (near the window) while blue appears in darker areas, such as the back of the room. Remember that the actual room geometry is not shown in Fig73, only the windows. We are going to put the roof back on a bit.

The arrows are so far made of the material plastic. This has the advantage that the colour coding is straight-forward since the valid values for plastic are between zero and one. To show more convincingly that the arrows are objects which do not belong to the scene, it is actually better to use the glow material. This makes them look somewhat unreal and stick out.

What we need to adjust is the intensity of the glow. This is dependent on the ambient light as well as the image exposure and will need some tweaking from us, as well as modifications to some of the files. Change the bright multiplier to set the glow intensity in *colour.cal*.

```
$ cat <<_EndOfColour_ > colour.cal
max=$MAX;
multi=100/max;
t=in(1)*multi;
bright=1;
red=if(t-50,-1*(50-t)/50,0)*bright;
grn=if(t-50,1-((t-50)/50),t/glow)*bright;
blu=if(50-t,-1*(t-50)/50,0)*bright;
_EndOfColour_
```

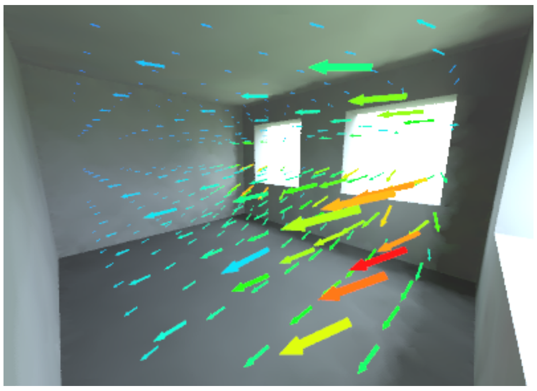


Figure 74: The complete simulation with illuminance vectors

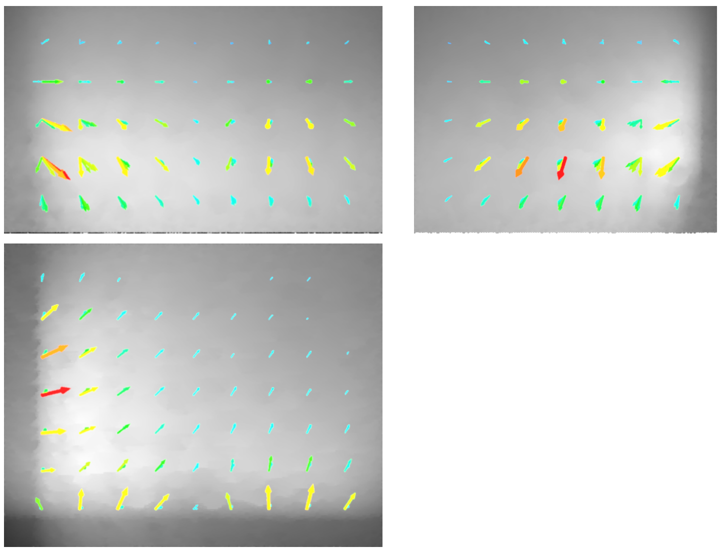


Figure 75: Flat projections of the simulation

```
$ cat arrow_mat.fmt
void glow arrow${m}_mat
0
0
4  ${r} ${g} ${b} 0
```

Before fiddling with the bright multiplier, it's a good idea to re-assemble your model. Stick the walls and roof back into the octree, like you see in Fig 74.

All our efforts paid out in the end. Here are some more pictures (Fig 75): views from +x, -y, +z. Appendix A.6 contains the full listing of the BASH shell script.

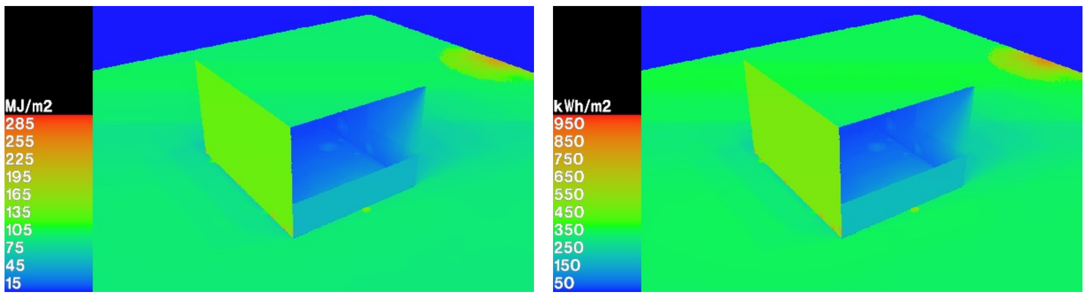


Figure 76: False colour results of `radmap`

6 Add-ons to Radiance

6.1 Annual Daylight Availability

6.1.1 Using `radmap` to Produce Irradiation Maps

`radmap` is a command line tool for producing irradiation and illumination maps. It was written by Francesco Anselmo. A presentation and the software are available from [14]. Since `radmap` is written in Python, there is no need to compile it. Python should be installed on most Linux system as standard.

Before we start, it is necessary to download weather files for our particular location. The web site of EnergyPlus [4], a thermal simulation package, has many international weather files available for free download. Select the one closest to your location.

```
$ radmap -s 8 -a 51 -o 0 -m 0 -n 1 \
   -w GBR_London.Gatwick_IWEC.epw --no-albedo \
   --viewfile birdseye.vf --radfile all_in_one.rad \
   --prefix out -x 500 -y 500 --keep-temporary-maps \
   --sky-resolution 128 --max-kWh 1000 --max-MJ 300
```

`radmap` produces many lines of output on STDOUT. Appendix A.1 lists what is going on under the hood.

Although `radmap` uses a clever approach based on daylight coefficients, it will take a while for the simulation to run. The results are given in Fig 76:

In case you are not happy with the false colour images, there is no need to re-run the entire simulation. The illuminance maps are saved, and new false colour images can be created very quickly. The table below lists the defaults used by `radmap`. The only ones that may be changed are `--max-kWh` and `--max-MJ`, which are equivalent to the `-s` in `falsecolor`. `radmap` also comes with a `-s` option, but this has a different meaning. Please type `radmap -h` for help on all available options. The `falsecolor` options are just for your information and can not be altered without editing the source code.

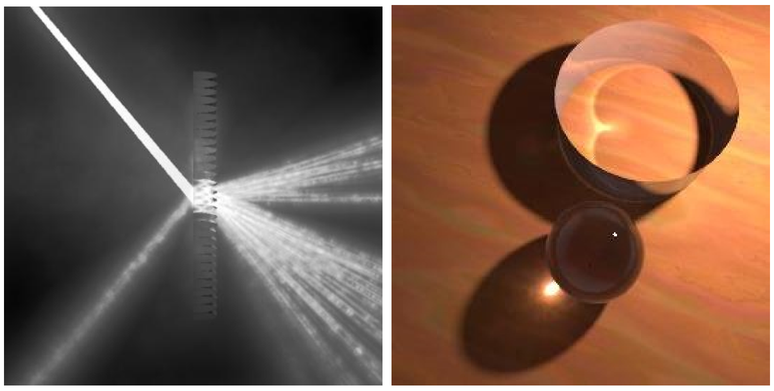


Figure 77: Example photon map simulations

image	`falsecolor` options	`radmap` options
kWh	-l kWh/m2	-
	-s 600	–max-kWh 600
	-m 0.01	-
MJ	-l MJ/m2	-
	-s 2000	–max-MJ 2000
	-m 0.00027777778	-

`radmap` was written for a project in Sicily. If your weather file is for a less sunny climate, you might want to reduce the maximum scale of the false colour image. For London, reduce it to half or one third of the default settings.

6.2 Misc

6.2.1 The photon-map

A forward ray-tracer plug-in originally written by Roland Schregle of the Fraunhofer Institut [6]. The limitations of Radiance 'Classic' become apparent when the scene geometry involves a large number of specular reflections, e.g. in a light pipe, or with the phenomenon of caustics. This is due to the backward-raytracing approach which Radiance takes. The `pmap` plug-in solves this problem. As of Oct 2014, work is underway to include photon-mapping into the official Radiance source tree . Separately downloading and compiling the plug-in will no longer be required once this work has been completed.

`pmap` comes with excellent documentation that should get you started on its (admittedly rather difficult) use. Please also see Peter Apian-Bennewitz' photon map tutorial on light redirection using prisms and mirrors [26].

References

[1] ERCO Lighting. URL http://www.erco.com.

[2] Gnuplot homepage. URL http://gnuplot.info.

[3] Some of Greg Ward's papers. URL http://anyhere.com/gward/papers.html.

[4] Energy+ weather files, 2012. URL http://www.eere.energy.gov/buildings/energyplus/.

[5] Hugin Panorama Tools, 2012. URL http://hugin.sourceforge.net.

[6] Radiance Photon Map, 2012. URL http://www.ise.fraunhofer.de/en/areas-of-business-and-market-areas/applied-optics-and-functional-surfaces/lighting-technology/lighting-simulations/radiance/photon-mapping?set_language=en.

[7] Aidrian O'Connor. Phi in Nature, 2012. URL http://www.natures-word.com/sacred-geometry/phi-the-golden-proportion/phi-the-golden-proportion-in-nature.

[8] Axel Jacobs. Radiance resources on www.jaloxa.eu. URL http://www.jaloxa.eu/resources/radiance.

[9] Axel Jacobs. LEARNIX. Web site, Jan 2012. URL http://www.jaloxa.eu/mirrors/learnix/index.shtml.

[10] Axel Jacobs. *Radiance Tutorial*, 2012. URL http://www.jaloxa.eu/resources/radiance/documentation/index.shtml.

[11] Axel Jacobs. *UNIX for Radiance*, 2012. URL http://www.jaloxa.eu/resources/radiance/documentation/index.shtml.

[12] Axel Jacobs, Marc Fontoynont, Aris Tsangrassoulis, Afroditi Synnefa, Wilfried Pohl, and Andreas Zimmermann. SynthLight Handbook, 2002. URL http://new-learn.info/packages/synthlight/handbook/index.html.

[13] 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.

[14] Francesco Anselmo. radmap, 2005. URL http://www.bozzograo.net/radiance/.

[15] C. A. Furuti. Azimuthal Map Projection, 2012. URL http://www.progonos.com/furuti/MapProj/Normal/ProjAz/projAz.html.

[16] Georg Mischler. *Rayfront Manual*, 2012. URL http://www.schorsch.com/rayfront/manual/transdef.html.

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

[18] Greg Ward. *Radiance 4.1 Reference Manual*, 2011. URL http://radsite.lbl.gov/radiance/refer/refman.pdf.

[19] Greg Ward. Official Radiance web site, 2012. URL http://radsite.lbl.gov/radiance.

[20] John Mardaljevic. Radiance Technical Notes, 2012. URL http://www.iesd.dmu.ac.uk/~jm/doku.php?id=resources.

[21] Larry Ewing. Linux 2.0 Penguins, 2012. URL http://www.isc.tamu.edu/~lewing/linux/.

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

[23] G.W. Larson. The Holodeck: A Parallel Ray-caching Rendering System. In *Proceedings of the Second Eurographics Workshop on Parallel Graphics and Visualisation*, Sep 1998. URL http://anyhere.com/gward/papers/tog99.pdf.

[24] Lawrence Berkeley National Laboratory. Radiance web site. Web site, 2012. URL http://radsite.lbl.gov/radiance.

[25] Megatron. Cubic Illuminance Meter. URL http://www.otc.co.uk/megatron/cim/index.html.

[26] Peter Apian Bennewitz. *Pmap Tutorial*, 2005. URL http://www.pab-opto.de/radiance/pm-pab/.

[27] Praving Chandra and Eric W. Weisstein. Fibonacci Number, 2012. URL http://mathworld.wolfram.com/FibonacciNumber.html.

[28] Radiance Community. Radiance-Online. Web site, 2012. URL http://www.radiance-online.org.

[29] Raphael Compagnon. *Radiance Course on Daylighting*, 1997. URL http://radsite.lbl.gov/radiance/refer/rc97tut.pdf.

[30] William Soukoreff. Some Fun with GNUPLOT, 2006. URL http://dynamicnetservices.com/~will/gnuplot/.

A Appendices

A.1 Output of radmap session

```
$ radmap -s 8 -a 51 -o 0 -m 0 -n 1 -w GBR_London.Gatwick_IWEC.epw --no-albedo \
    --viewfile birdseye.vf --radfile all_in_one.rad --prefix out -x 500 -y 500 \
    --keep-temporary-maps --sky-resolution 128 --max-kWh 1000 --max-MJ 300
[radmap] v. 0.1b / 2003-11-24
--------------------------------------------
Radiance based irradiation map production
--------------------------------------------
by Francesco Anselmo <anselmo@dream.unipa.it>
D.R.E.Am.
Dipartimento di Ricerche
Energetiche ed Ambientali - Palermo - Italy

This work has been financially supported by
the European Social Fund
ITALIAN NATIONAL OPERATIVE PROGRAM 2000/2006
Ricerca Scientifica, Sviluppo Tecnologico,
Alta Formazione
Misura III.4.
Formazione Superiore e Universitaria

Location latitude: 51.0
Location longitude: 0.0
Location meridian 0.0
Number of samples per hour: 1
Loading EPW weather data from file GBR_London.Gatwick_IWEC.epw
No albedo component will be taken into account
View file to be used: birdseye.vf
.rad file to be used: all_in_one.rad
Prefix for output files: out
Output image x resolution: 300
Output image y resolution: 300
Sky image resolution:  128
Using EPW weather data ...
1991/1/1 .......cioooooo..........
1991/1/2 .......ooooooo..........
...
1983/12/31 ......ioooooooo.........
Total number of daylight weather samples :: 4391
Composing the cumulative sky picture ...
Combining skies from 6 to 82 :: 0
Combining skies from 83 to 174 :: 1
...
Combining skies from 8744 to 8760 :: 137
group n. 0
group n. 1
...
group n. 137
Using a previously populated ambient cache ...
rpict: 0 rays, 0.00% after 0.001u 0.001s 0.011r hours on albatros
rpict: 50629 rays, 100.00% after 0.002u 0.001s 0.011r hours on albatros
Rendering picture out_0000_irrmap.pic
rpict: 0 rays, 0.00% after 0.001u 0.001s 0.010r hours on albatros
rpict: 1331964 rays, 34.37% after 0.010u 0.001s 0.031r hours on albatros
...
rpict: 10575322 rays, 100.00% after 0.064u 0.001s 0.166r hours on albatros
Irradiation map saved as /home/axel/projects/radiance/radmap/out_0000_irrmap.pic
...
```

A.2 *time_of_day.bash*

```
#!/bin/bash

# Bash shell script to produce time_of_day image sequences
# Taken from Rendering with RADIANCE,
# Modified by Axel Jacobs, 2 Apr 2010

# The arguments for gensky and the label
DAY=23
MONTH=03
# Geographical location of the site
SITE="-a 52 -o 0 -m 0"

# The ambient parameters
AF="tmp/tmp.amb"
AMB="-ab 2 -ad 512 -as 128 -ar 64 -aa 0.2 -av 0 0 0 -af $AF"

for hour in {04..22}; do
    # An intermediate sky makes the sunpatches less bright
    skypar="$MONTH $DAY +$hour $SITE +i"
    gambv=$(gensky $skypar \
            | rcalc -i '# Ground ambient level: £{ga}' -e '£1=ga')
    psign -h 16 $DAY/$MONTH $hour:00 > tmp/label.hdr
    # If it's night outside, do nothing
    if [ "$gambv" != "0" ] ; then
        # Generate the sky
        gensky $skypar > tmp/sky_tmp.mat
        # Compile the octree
        oconv -i script.oct tmp/sky_tmp.mat sky.rad > tmp/tmp.oct

        # Render the picture
        # Use an ambient file for the calculation
        rm -f $AF
        rpict -vf nice.vf -x 64 -y 64 $AMB tmp/tmp.oct > /dev/null
        rpict -vf nice.vf -x 800 -y 600 tmp/tmp.oct \
                | pfilt -1 -x /2 -y /2 -e .2 \
                | pcompos - 0 0 tmp/label.hdr 0 0 \
                > images/$hour.hdr
        # Convert HDR image to JPEG format
        convert -gamma 2.2 images/$hour.hdr images/$hour.jpg
    fi
done

#EOF
```

A.3 *picture_tux.mat*

```
# This is the material file for picture_tux.rad
void colorpict tux_image
15 clip_r clip_g clip_b tux.hdr picture.cal pic_u pic_v -rx 90 -s .5989 -t .1 0 .1
0
0

tux_image plastic painting_mat
0
0
5  1 1 1  0 0

void plastic frame_mat
0
0
5  0.3 0.2 0.1  0 0
```

A.4 *picture_tux.rad*

```
# A picture of Tux the penguin
# It hangs on the zx wall at y=0, facing +y.
# The origin is at the bottom right-hand corner.
# The frame is 0.9 units high 0.8 units wide and 0.02 units thick.

# The frame
!genbox frame_mat frame .8 .02 .9

# Make sure the canvas is slightly in front of the frame!
painting_mat polygon painting
0
0
12  .1 .021 .8  .7 .021 .8  .7 .021 .1  .1 .021 .1
```

A.5 *arrows.rad*

```
# This is a set of three arrow heads indicating the origin
# of the co-ordinate system.
# Colours are: x ... red
#              y ... green
#              z ... blue

# The materials (a little shiny)
void plastic x_mat 0 0 5  1 0 0  .01 .1
void plastic y_mat 0 0 5  0 1 0  .01 .1
void plastic z_mat 0 0 5  0 0 1  .01 .1
void plastic origin_mat 0 0 5  1 1 1  .01 .1

# The objects
x_mat cylinder x_axis 0 0 7  0 0 0  .8 0 0  .05
x_mat cone x_arrow 0 0 8  .7 0 0  1 0 0  .12 0
x_mat ring x_base 0 0 8  .7 0 0  -1 0 0  0 .12
y_mat cylinder y_axis 0 0 7  0 0 0  0 .8 0  .05
y_mat cone y_arrow 0 0 8  0 .7 0  0 1 0  .12 0
y_mat ring y_base 0 0 8  0 .7 0  0 -1 0  0 .12
z_mat cylinder z_axis 0 0 7  0 0 0  0 0 .8  .05
z_mat cone z_arrow 0 0 8  0 0 .7  0 0 1  .12 0
z_mat ring z_base 0 0 8  0 0 .7  0 0 -1  0 .12
origin_mat sphere origin 0 0 4  0 0 0  .05
```

A.6 Vector Illuminance BASH script

```
#!/bin/bash

# do_vector_illu.bash
#
```

```
# Create a 3D grid of vector illuminances. The numbers are
# converted to 'real' arrows, colour coded for intensity.
#
# (c) Axel Jacobs, 2 Apr 2010

OCTREE="vector.oct"
AF="vector.amb"
AV="0.087"
AMB="-af $AF -ab 3 -ar 1024 -ad 4096 -as 2048 -aa .07 -av $AV $AV $AV"

TMPDIR="tmp"
if ! [ -d $TMPDIR ]; then
    mkdir $TMPDIR
fi
PTSFILE="$TMPDIR/points.dat"
PTSTMPFILE="$TMPDIR/points_tmp.dat"

# To determine the illuminance vector, the illu has to be
# measured in six directions: +x, -x, +y, -y, +z, -z
VECTBASE="$TMPDIR/vector"
DATBASE="$TMPDIR/lux"
EXTN=".dat"
VECT[1]="1\t0\t0"
VECT[2]="-1\t0\t0"
VECT[3]="0\t1\t0"
VECT[4]="0\t-1\t0"
VECT[5]="0\t0\t1"
VECT[6]="0\t0\t-1"
for i in {1..6}; do
    VECTFILE[$i]="${VECTBASE}${i}${EXTN}"
    DATFILE[$i]="${DATBASE}${i}${EXTN}"
done

XFILE="$TMPDIR/xdat$EXTN"
YFILE="$TMPDIR/ydat$EXTN"
ZFILE="$TMPDIR/zdat$EXTN"
LENFILE="$TMPDIR/len$EXTN"

# The creation of the points is a hand-job at the moment.
# It could be automated with some getbbox magic.
total=$(cnt 7 9 5 |wc -l)
GRID="0.5"
cnt 7 9 5 |rcalc -e '$1=$1/2+1;$2=$2/2+1;$3=$3/2+.5' > $PTSFILE

for i in {1..6}; do
    echo -n "" > $PTSTMPFILE
    for j in $(cnt $total); do
        echo -e ${VECT[$i]} >> $PTSTMPFILE
    done
    rlam $PTSFILE $PTSTMPFILE > ${VECTFILE[$i]}
    cat ${VECTFILE[$i]} |rtrace -h -w -ab 1 -I -ov $OCTREE \
            |rcalc -e '$1=179*($1*.265+$2*.67+$3*.065)' \
            > ${DATFILE[$i]}
done

```

```
# Take the differences: illu(-x) - illu(+x) etc.
rlam ${DATFILE[1]} ${DATFILE[2]} |rcalc -e '$1=$1-$2' > $XFILE
rlam ${DATFILE[3]} ${DATFILE[4]} |rcalc -e '$1=$1-$2' > $YFILE
rlam ${DATFILE[5]} ${DATFILE[6]} |rcalc -e '$1=$1-$2' > $ZFILE

# Determine the absolute illuminance as the length of the vector.
# This is a 3D Pythagoras for you.
cat <<_EndOfSize_ > size.cal
{ Take the squares }
sx=dx*dx;
sy=dy*dy;
sz=dz*dz;
{ Length in 3D }
len=sqrt(sx+sy+sz);
{ Length in xy plane }
len_xy=sqrt(sx+sy);
{ pivot angles up and around z-axis }
roty=atan(dz/len_xy)*180/PI;
rotz=atan(dy/dx)*180/PI+180;
_EndOfSize_

rlam $XFILE $YFILE $ZFILE \
        |rcalc -f size.cal -e 'dx=$1;dy=$2;dz=$3;$1=len' \
        > $LENFILE

# Which is the longest illu vector?
#max=£(cat £LENFILE |awk '{if (£1>max) max=£1} END {print max}')
max=$(total -u $LENFILE)

# Write out a RADIANCE cal for the colour mapping
cat <<_EndOfColour_ > colour.cal
max=$max;
av=$AV;
multi=100/max;
t=in(1)*multi;
{ Edit arrow_mat.fmt to select the use of plastic
  or glow material for the arrows.
  Plastic: Make sure that 'bright' is 1, so R, G, B
           run between 0 and 1.
  Glow: This is more difficult. I tried basing it on the
           ambient level, assuming it's R=G=B. 10*the
           ambient value looks about right. YMMV.
}
bright=av*10;
red=if(t-50,-1*(50-t)/50,0)*bright;
grn=if(t-50,1-((t-50)/50),t/glow)*bright;
blu=if(50-t,-1*(t-50)/50,0)*bright;
_EndOfColour_

# The longest arrow should be as long as the grid spacing.
multi=$(echo $max $GRID |rcalc -e '$1=$2/$1')
echo "s(input) = input*$multi" > scale.cal

# RADIANCE geometry file of the vector arrows
rlam $PTSFILE $XFILE $YFILE $ZFILE $LENFILE \
        |rcalc -f scale.cal \
                -e '$1=$1;$2=$2;$3=$3;$4=s($4);\
                $5=s($5);$6=s($6);$7=s($7)' \
        |rcalc -f size.cal -o arrow_xform.fmt \
                -e 'x=$1;y=$2;z=$3;dx=$4;dy=$5;dz=$6; \
                s=$7;ry=roty;rz=rotz;m=outno' \
        > arrows.rad

# Each arrow has its own colour, depending on its length.
# Like with falsecolor, the longest arrow is red,
# while a length of zero is mapped to blue.
cat $LENFILE \
        |rcalc -f colour.cal -o arrow_mat.fmt \
            -e 'r=red;g=grn;b=blu;m=outno' \
        > arrows.mat

#EOF
```

A.7 *pcyl.cal* for Cylindrical View

```
{
  pcyl.cal

  Definitions for cylindrical projection

  R. Compagnon, Martin Centre, Cambridge UK, 04/JUN/97

  Adopted for more general cylindrical projections by
  Axel Jacobs, 8 Nov 2006
}

{ Parameters defined externally:
  XD : horizontal picture dimension (pixels)
  YD : vertical picture dimension (pixels)
  AT : altitude at the top line of the picture (degrees)
  AB : altitude at the bottom line of the picture (degrees) }

{ Direction of the current pixel (angles in radians) }
px=$2;
py=YD-$1;

altitude=(AB+py*(AT-AB)/YD)*PI/180;
azimut=((px-XD/2)*180/XD-OF)*PI/180;

{ Transformation into a direction vector }
n=sqrt(1+sin(altitude)^2);
dx=-sin(azimut)*cos(altitude);
dy=-cos(azimut)*cos(altitude);
dz=sin(altitude);

{ Output line to rtrace }
$1=10.5;$2=-.05;$3=15.5;
$4=dx;$5=dy;$6=dz;

{ EOF }
```

Index