Understanding `rtcontrib`

Axel Jacobs
<jacobs dot axel at gmail dot com>

September 17, 2010

Contents

Revision History

2 April 2010

- Typeset in LyX
- Nicer code listings
- More coherent document structure
- Diagrams and graphs as vector images
- Spellchecked

1 Feb 2010

- Document is mostly finished
- HTML style sheets applied
- Updates announced on radiance-general

26 Jan 2010

- More on DDS and weather data

23 Jan 2010

- Added section on *tregsamp.dat* and `genskyvec`

22 Jan 2010

- Many corrections based on feedback from Greg
- Added Contributions from Different Parts of the Sky

17 Jan 2010

- First draft announced on radiance-general

1 Introduction

`rtcontrib` is a new addition to the Radiance toolkit. It is based on `rtrace`, but extends its functionality far beyond of what even the most vivid imagination would have thought possible Radiance can do. `rtcontrib` is like the Swiss army knife of Radiance commands, and we are not talking the ones that come with just a dozen blades and tools. Imagine a pocket knife that is so multi-purpose it won't even fit in your pocket. Now double this, and you get an idea of what `rtcontrib` can do.

Well, COULD DO would be a better term. The problem with `rtcontrib` is that so far, there is not much documentation on its use. So which blade do you flip out to get a particular job done, and without doing damage to your fingers?

This document attempts to provide some of the answers. My main motivation is to figure out how `rtcontrib` can be used for dynamic daylight simulations, DDS. This document was primarily written as a log documenting how I went about in my attempt to wrap my head around this `rtcontrib` thing. It is therefore not a step-by-step guide, at least not a concise one. Many intermediate steps are included which might or might not be necessary for DDS. I felt, however, that all those little things are important enough to be part of this document. This also means that all the mistakes I made along the way are documented, so you can learn from them. Some of those mistakes are actually made deliberately for educational purposes, while others are not.

Being a log, this document has a linear structure. Read it top to bottom. I strongly encourage you to follow the exercises rather than skipping over them. There are a million stumbling blocks that you need to be able to spot and circumvent in order to get dynamic daylight simulations working accurately and reliably.

Through writing this document, I have found that my understanding of Radiance, which I would consider to be pretty good, improved quite a lot. I hope that you can benefit as much. Even if DDS is not on your agenda, and you have used Radiance for many years, you will most likely still find a thing here and there where you think "Oh, really!?". You are just as likely to spot errors, inconsistencies, and typos. Please do let me know.

2 Preparations

2.1 Making Sure Radiance Works

Some of the exercises require Radiance from 30 Jan 2010 or newer, so you might need to compile Radiance from source.

When you install Radiance from source, you normally download *radiance-HEAD.tgz* and *rad4R0supp.tar.gz*. You then extract those archives and run `./makeall install` in the *ray* directory. This install script also makes sure that all *cal* files from *rad4R0supp.tar.gz* (where they are stored in the *ray/lib* directory) are copied out. The default destination is */usr/local/lib/ray*, and this is why setting the RAYPATH environmental variable is important: It ensures that Radiance actually finds all those *cal* files. Debian and Ubuntu users who use Bernd's DEB packages will find the library files under */usr/share/radiance/*, so this is where their RAYPATH should point to.

You are probably aware that man pages are not installed by the makeall script, so you need to copy them manually. But this is old news. There are lots more *cal* files in *radiance-HEAD.tgz* where, confusingly enough, they are under *ray/src/cal/cal*. We need some of those *cal* files, so make sure you copy them over to the directory that

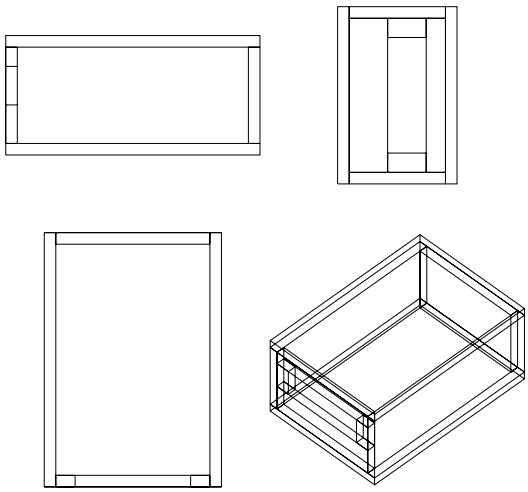


Figure 1: `objline` wire frame model of the simple test room

RAYPATH points to. Radiance does not search RAYPATH recursively, so don't put them in a sub-directory. Stick them straight under RAYPATH, where all the other *cal* files are already.

Debian and Ubuntu users who use the available packages and would like to follow this tutorial need to make sure that `genbox` is working. It's called `genrbox` on Debian-based distros.

```
$ sudo ln -s /usr/bin/genrbox /usr/local/bin/genbox
```

2.2 Base Case: `rpict` Rendering

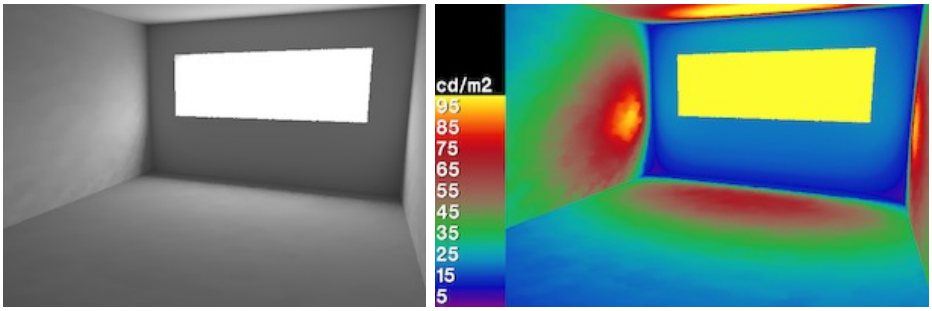
This is a simple test room with a rather boring 10,000 lx overcast sky that is generated with `gensky` (for London):

```
$ cat skies/sky_overcast.mat
!gensky 3 21 10 -c -a 51 -o 0 -m 0 -b 22.86
$ cat skies/sky.rad
skyfunc glow sky_glow 0 0 4 1 1 1 0
sky_glow source sky 0 0 4 0 0 1 180
skyfunc glow ground_glow 0 0 4 1 1 1 0
ground_glow source ground 0 0 4 0 0 -1 180
```

A wire frame representation of our test room is shown in Figure 1:

```
$ objline objects/testroom.rad |psmeta \
    > images/testroom.eps
$ convert -flatten images/testroom.eps images/testroom.png
```

Note that there is no ground plane. This is done deliberately for this exercise. If you were to model a real scenario and expect to get accurate values (luminance, illuminance), then you would need to have a ground plane. Please see the relevant section in the Advanced Radiance Tutorial.



(a) Normal luminance image (b) `falsecolor` image

Figure 2: `rpict` image of the test room. This is our reference case.

```
$ getinfo testroom_overcast.oct
testroom_overcast.oct:
    #?RADIANCE
    oconv materials/testroom.mat objects/testroom.rad
    skies/sky_overcast.mat skies/sky.rad
    FORMAT=Radiance_octree
$ cat image.opt
-ab 3 -ad 1024 -ar 256 -aa .1
```

All files can be downloaded here: rtcontrib_lesson.tgz

The `rpict` image will be our reference case. It is shown in Figure 2, together with a false colour representation.

```
$ DIMS=600
$ vw="-x $DIMS -y $DIMS -vf views/back.vf"
$ rpict @image.opt $vw testroom_overcast.oct \
    > images/rpict.hdr
```

2.3 Replacing `rpict` with `rtrace`

Traditional Radiance used to come with the three Rs: `rvu`, `rpict` and `rtrace`. `rvu` is not covered here, since it's only purpose (at least as far as this exercise is concerned) is to the get the views just right.

Then, there is `rpict`. It renders pretty images.

Finally, we have `rtrace`. What it returns is configurable with the `-o*` option, but usually we want it to return one number (well, actually a triple of R,G,B) representing the radiance at the point where the ray given to it on STDIN hits an object, or the irradiance at the point (`-I` option).

If you imagine calling `rtrace` hundreds or even thousands of times, and getting the direction of those sample rays just right, then you basically have a grid of R,G,B radiance values that might be represented as an image. It doesn't need to be an image– you can do with those values anything you like, but an image is rather, well... visual. For a convincing visual representation, the individual sample rays need to be carefully aligned to one another, and between them, they should ensure a certain mayor view direction, view angle etc. (all the kind of parameters you find in a `.vf` file). This is

where `vwrays` comes in. It takes a view file and target image dimensions and works out all those individual sample rays, so that they may (or may not–the choice is yours) be assembled into an image.

```
$ cat views/back.vf
rvu -vtv -vp .5 5.5 1.5 -vd 0.377493 -0.906892 -0.18721 \
    -vu 0 0 1 -vh 60 -vv 45 -vo 0 -va 0 -vs 0 -vl 0
```

`vwrays` and `rtrace` are designed to work well together. The normal behaviour for `vwrays` is to output those rays (x,y,z of the origin and dx,dy,dz of the direction of the ray) in a human readable form.

For instance, this is the 1000th ray for the given view. With the image being 600 pixels wide, this particular pixel is roughly in the middle of the second scan line:

```
$ DIMS=600
$ vwrays -x $DIMS -y $DIMS -vf views/back.vf \
    |head -1000 |tail -1
5.00000e-01 5.50000e+00 1.50000e+00 \
    2.09723e-01 -9.57613e-01 1.97467e-01
```

Please note that the first triple here is identical to the view point.

When `vwrays` and `rtrace` work together, the data which they exchange (actually it's data passed from `vwrays` to `rtrace`–it's a one-way communication) doesn't actually need to be readable by humans. In fact, it's more efficient if this data is not human readable. This is configured with the `-f*` switch. `vwrays` understands (`a`)scii (this is the human readable format), (`f`)loat and (`d`)ouble. Without getting too technical here, let us just say that anything other than (`a`) can be read only by another Radiance command.

It is important to note that the `-f<output_format>` switch to `vwrays` only refers to its output. `vwrays` does not accept any input. It's purely a generator of ray directions. `rtrace`, on the other hand, does accept in input and also produces something as an output. This something is configurable with the `-o` option. According to `man rtrace`,

> "-o<specification> Produce output fields according to <specification>."

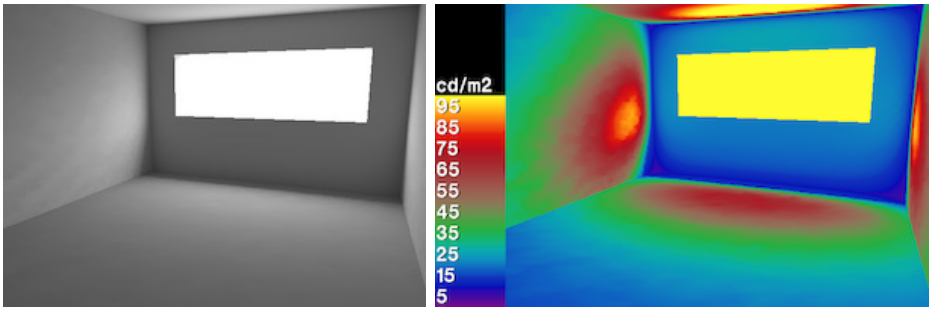
those output fields can be nearly anything we like, but most commonly, we'll want it to be the radiance of the point where the ray intersects an object, which is also the default (`-ov`).

Because `rtrace` takes an input AND produces an output, the option modifying its format takes two characters rather than just one: `-f<input_format><output_format>`.

The `<input_format>` specifier in this case MUST be identical to the one telling `vwrays` what format to produce its output in. In the example below, this is `f`, which is floating-point (remember, this is one of the non-human readable ones).

So let's produce an image identical to the one above, but without the `rpict` 'convenience layer' and closer to bare-metal. The result is shown in Figure 3. It is almost indistinguishable from from the `rpict` image in Figure 2.

```
$ vwrays -ff $vw |rtrace @image.opt -ffc \
    $(vwrays -d $vw) testroom_overcast.oct \
    > images/rtrace.hdr
```



(a) Normal image (b) False colour image

Figure 3: The same view, but the image was produced with `vwrays` and `rtrace` instead of `rpict`.

What is important here is the `c` output format in the `rtrace` command line. On top of `a`, `f` and `d` which `rtrace` accepts as an input format (and `vwrays` produces as an output), it accepts the `c` output format. From `man rtrace`:

> "`-fio` Format input according to the character `i` and output according to the character `o`. `rtrace` understands the following input and output formats: '`a`' for ascii, '`f`' for single-precision floating point, and '`d`' for double-precision floating point. In addition to these three choices, the character '`c`' may be used to denote 4-byte floating point (Radiance) color format for the output of values only (`-ov` option, below). If the output character is missing, the input format is used.
>
> "Note that there is no space between this option and its argument."

While we need the `c` output format to generate an image from `rtrace`'s output, the communication between `vwrays` and `rtrace` could have been defined in ASCII or double-precision float. If this is all double-dutch to you, just stick to the `f` notation.

2.4 A Word on Image Compression

Radiance images typically use the HDR extension. This stands for 'High Dynamic Range'. The HDR image format stores four bytes per pixel: A factor for the Red, Green, and Blue colour channel, and an exponent which is common to all three. This is why the HDR image format is also referred to as 'RGBE': Red, Green, Blue, Exponent. There is also a XYZE HDR format that encodes the three primaries in the CIE XYZ format rather than RGB (which is ambiguous–it depends on the primary colours that you choose, although those are set in stone for Radiance images), but it is not used a lot, and all Radiance commands default to RGBE.

The RGBE image file format comes in two flavours: compressed (resulting in potentially smaller file sizes), and un-compressed. An un-compressed image occupies 4 bytes for every pixel (R,G,B,E). It always does, so you can work out exactly how much hard disk space it occupies.

If neighbouring pixels (we are talking scan lines here: pixels within one row of the image) are identical, the image size may be reduced by what is known as Run-Length Encoding, RLE. So if there are many black pixels in the image, the file size might be reduced by just storing something like:

"This pixel has a value of (0, 0, 0, 0), and it is repeated n times."

Actually, Radiance is even smarter than this and can apply RLE to the individual channels. So if, for instance, a number of successive pixels have different R,G,B values but share the same exponent, it is still possible to reduce the file size with RLE.

The more pixels there are with the same radiance, the more effective the RLE compression. So why is this important?

`rtrace` does NOT compress the images with run-length encoding. If disk space is an issue and you absolutely MUST reduce the size of the HDR files (don't forget there will be hundreds if not thousands of them, and many of them will be partially or even fully black), do NOT use `pfilt`. We are going to use the `pcomb` command later on, and `pcomb` does not honour the image exposure with its default settings. Rather, each image would have to be called with the -o option. This is a per-file option and needs to be repeated for each input file. Better do a reverse `ra_rgbe` (with the `-r` option) to achieve RLE and thus smaller files. This does not change the image's exposure. `rtcontrib` copies this behaviours from its little brother, `rtrace`. Not having RLE also allows partial runs to be recovered if the process dies (`-r` option to `rtcontrib`).

```
$ vwrays -ff $vw |rtrace @image.opt -ffc \
    $(vwrays -d $vw) testroom_overcast.oct \
    > images/rtrace.hdr
$ vwrays -ff $vw |rtrace @image.opt -ffc \
    $(vwrays -d $vw) testroom_overcast.oct \
    |ra_rgbe -r > images/rtrace_rle.hdr
$ ls -sh images/*.hdr
 420K images/rpict.hdr
1016K images/rtrace.hdr
 556K images/rtrace_rle.hdr
```

3 Getting Serious

After the warming-up exercises, the time has come to invoke `rtcontrib` and to find out what it does and how it works.

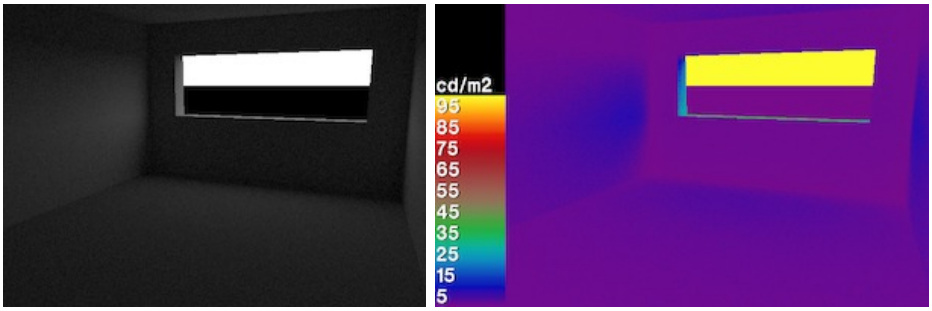
3.1 Replacing `rpict` with `rtcontrib`

Although `rtcontrib` is essentially a modified version of `rtrace`, simply replacing `rtrace` with `rtcontrib` on the command line produces an error:

```
$ vwrays -ff $vw |rtcontrib @image.opt -ffc \
    $(vwrays -d $vw) testroom_overcast.oct \
    > images/rtcontrib.hdr
rtcontrib: fatal - No modifiers specified
```

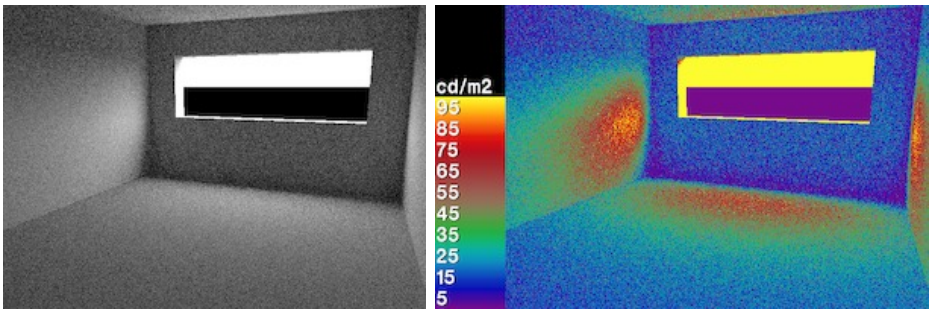
For `rtcontrib` to work we must always specify a modifier. Let us add `-m sky_glow` for now.

```
$ vwrays -ff $vw |rtcontrib @image.opt -ffc \
    $(vwrays -d $vw) -m sky_glow \
    testroom_overcast.oct > images/rtcontrib.hdr
```



(a) Normal image (b) False colour image

Figure 4: First attempt to create an image with `rtcontrib`.



(a) Normal image (b) False colour image

Figure 5: `rtcontrib` image, generated with the `-V+` switch.

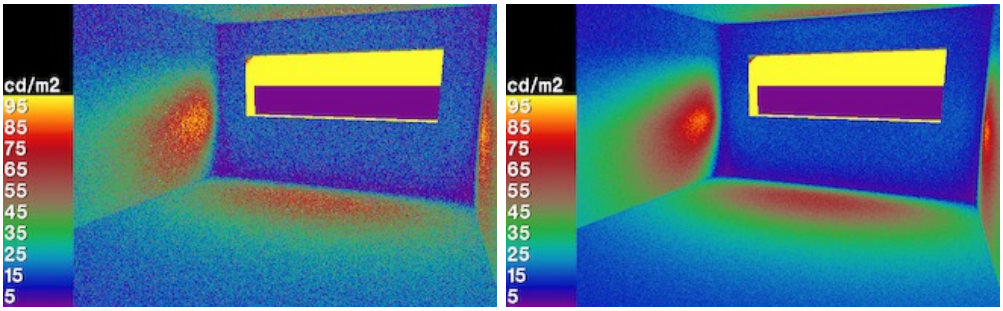
This works but the image is way too dark, as shown in Figure 4. Reading through the man page for `rtcontrib`, we notice this phrase:

> "By setting the boolean `-V` option, you may instruct `rtcontrib` to report the contribution from each material rather than the ray coefficient. This is particularly useful for light sources with directional output distributions, whose value would otherwise be lost in the shuffle. With the default `-V-` setting, the output of `rtcontrib` is a coefficient that must be multiplied by the radiance of each material to arrive at a final contribution. This is more convenient for computing daylight coefficients, or cases where the actual radiance is not desired. Use the `-V+` setting when you wish to simply sum together contributions (with possible adjustment factors) to obtain a final radiance value."

In this exercise, it is the contribution that we're after, not some obscure coefficient. That comes later. Let's see what the `-V+` does to our image.

```
$ vwrays -ff $vw |rtcontrib @image.opt -ffc \
    $(vwrays -d $vw) -m sky_glow -V+ \
    testroom_overcast.oct > images/rtcontrib-V+.hdr
```

This works now, as you can see in Figure 5. The result is roughly identical to those from `rpict` and `rtrace`. Two details are important to notice here:



(a) *img.opt*: `-ad 1024` (b) *rtc.opt*: `-ad 4096 -lw 0.0003`

Figure 6: Because the ambient cache is disabled, `rtcontrib` images should be rendered with a much higher ambient density.

Firstly, the image is very grainy. For `rtcontrib` to be able to do its magic (we haven't actually defined what exactly this magic is, but we'll come to that eventually...), it needs to disable the ambient caching. Irrespective of the rendering options that you tell it to use (ours are stored in *image.opt*), it always sets `-aa` to zero.

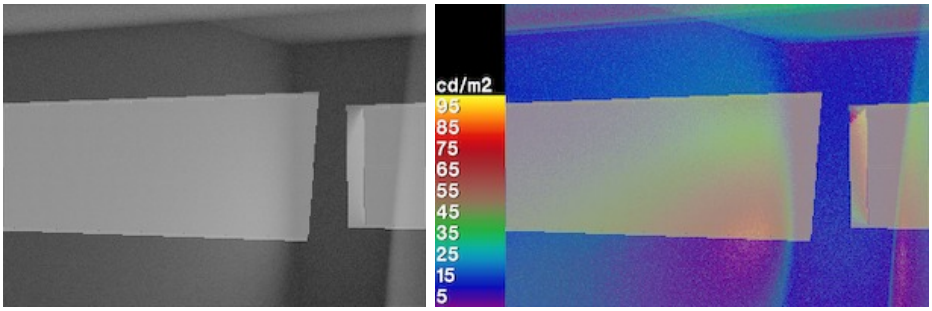
```
$ for cmd in rvu rpict rtrace rtcontrib; do \
    $cmd -defaults |grep ^-aa; done
-aa 0.300000 # ambient accuracy
-aa 0.200000 # ambient accuracy
-aa 0.100000 # ambient accuracy
-aa 0.000000 # ambient accuracy
```

In order to compensate for the disabling of the ambient cache, it is advisable to increase the number of ambient division (`-ad`) option. Try doubling or quadrupling them. You should also adjust the `-lw` option to slightly more than the reciprocal of `-ad`. In our examples so far, we have used `-ad 1024` and the default for `-lw`, which is 0.002. Quadrupling `-ad` bring us to 4096, so $1/4096 = 0.00024$. We'll use `-lw 0.0003` for now, and store the new ambient settings in *rtc.opt*. With the higher ambient settings, the rendering will take much longer, so from now on, we'll ensure that `rtcontrib` uses all available processor cores. In my case, this is two, hence the `-n 2` option, which I simply included in *rtc.opt*:

```
$ cat rtc.opt
-ab 3 -ad 4096 -lw 0.0003 -n 2
$ vwrays -ff $vw |rtcontrib @rtc.opt -ffc \
    $(vwrays -d $vw) -m sky_glow -V+ \
    testroom_overcast.oct > images/rtcontrib-V+_highopt.hdr
```

Figure 6 shows a comparison of the `falsecolor` image from the previous run with *image.opt*, and the new one that uses the optimised `-ad` and `-lw` settings stored in *rtc.opt*.

To remove the graininess further and get an even smoother image, you need to employ oversampling: Create an image many times the size that you actually need, then filter it down with `pfilt`. It might help to give it a slight blur (`-r` option to `pfilt`). This oversampling is done automatically if you use the `rad` command. With



(a) Normal view (b) `falsecolor` image

Figure 7: Something went wrong here: two images in one

QUAL=Med, the intermediate image is twice as large as the target dimensions, with QUAL=High three times. `rad` also does the filtering for you, so all you need to do is hit the big button. How convenient.

`rtcontrib` does not handle this job for you, and you need to do this oversampling and down-sizing yourself. You will find that oversampling by a factor of two or even three might not be good enough and sill leaves you with a grainy image. You might want to go higher than this. We will not do any oversampling in the images in this document.

Secondly, the ground visible outside the testroom is black. It doesn't just look black–it doesn't actually emit or reflect any light. This is not good.

3.2 Contributions from Different Materials

`rtcontrib` silently ignores all light sources (materials) that are not passed with the `-m` or `-M` options. To get the ground to behave like it does in `rpict` and `rtrace`, we must explicitly pass its modifier to `rtcontrib`. The `-m` options can be repeated as many times as necessary. If there are many modifiers, they can be listed in a file instead. The name of the file is then passed to `rtcontrib` with the `-M` option.

```
$ vwrays -ff $vw |rtcontrib @rtc.opt -ffc \
    $(vwrays -d $vw) -m sky_glow -m ground_glow -V+ \
    testroom_overcast.oct \
    > images/rtcontrib-V+_sky+ground.hdr
```

Oops! Something is not quite right here (Figure 7). What happened? Let us examine the size of the file that we just generated, and compare it to the file size from the sky-only run.

```
$ ls -sh images/rtcontrib-V+_sky+ground.hdr \
    images/rtcontrib-V+.hdr
1016K images/rtcontrib-V+.hdr
 2.0M images/rtcontrib-V+_sky+ground.hdr
```

So the sky-only file is 1 MB, and the sky+ground image is 2 MB in size. Remember what we said about the size of image files when we talked about RLE earlier on? The images are not run-length encoded, which allows us to work out how many pixels they hold. Doing this reversely, we can also calculate how big the file should be:

```
$ vwrays -d $vw -x 600 -y 430 -ld-
```

So the file size we would expect is:

```
$ vwrays -d $vw |rcalc -e '$1=$2*$4*4'
1032000
```

in bytes, or

```
$ vwrays -d $vw |rcalc -e '$1=$2*$4*4/1024/1024'
0.984191895
```

in Megabytes.

This is for the pixel data, the actual image is slightly larger because of the image header. In other words: the image from the last run, the sky+ground run, actually contains enough pixels for two images–one image for the contribution from the sky (more correctly, the contribution from the sky_glow material), and one image for the ground_glow modifier. So the image is really two images in one that are interwoven, although there is only one header:

```
$ strings images/rtcontrib-V+_sky+ground.hdr
#?RADIANCE
oconv materials/testroom.mat objects/testroom.rad
    skies/sky_overcast.mat skies/sky.rad
rtcontrib -ab 3 -ad 4096 -lw 0.0003 -ar 256 -aa .1 -ffc
    -x 600 -y 430 -ld- -m sky_glow -m ground_glow -V+
SOFTWARE= RADIANCE 4.0a
lastmod Wed Jan 13 14:38:49 GMT 2010 by root on dove.localdomain
CAPDATE= 2010:01:22 14:22:05
GMT= 2010:01:22 14:22:05
FORMAT=32-bit_rle_rgbe
-Y 430 +X 600
```

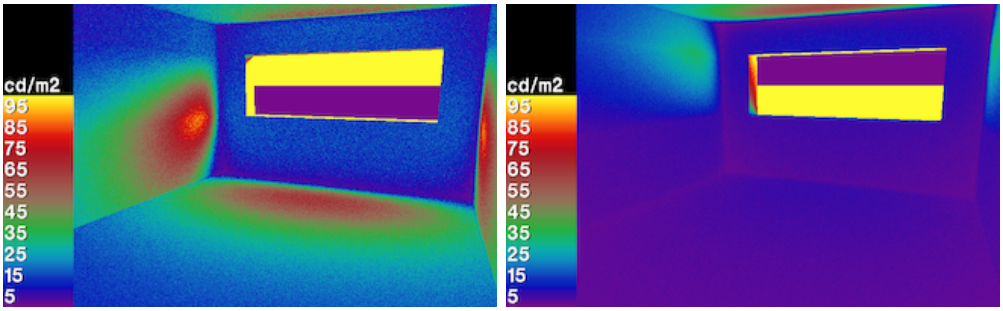
What we need to do is store the two images separately. This is what `rtcontrib`'s `-o` option is for. It is fairly flexible, and is capable of generating file names at run time if the special strings `%s` and/or `%d` are given. We will use the `%d` string in a later exercise, an will focus on `%s` for now. From `man rtcontrib`:

> "If an output specification contains a "`%s`" format, this will be replaced by the modifier name."

This seems to be exactly what we need. Let's see what happens:

```
$ vwrays -ff $vw |rtcontrib @rtc.opt -ffc \
    $(vwrays -d $vw) -m sky_glow -m ground_glow -V+ \
   -o images/rtcontrib-V+_%s.hdr testroom_overcast.oct
```

This is still not what we want. There is lots of output, but nothing ends up in the files which we expect to be created (*images/rtcontrib-V+_sky_glow.hdr* and *images/rtcontrib-V+_ground_glow.hdr*). Instead, our terminal is filled up with rubbish. This is what `rtcontrib` does if something goes wrong and it can't figure out what to do with the data–it dumps it to STDOUT. This is essentially what `rtrace` would do. To prevent this from happening, you could redirect STDOUT to a temporary file. This doesn't sort the problem, but at least it stops `rtcontrib` from clogging up your terminal window.



(a) Sky (b) Ground

Figure 8: Separating illumination from the sky and ground

```
$ vwrays -ff $vw |rtcontrib @rtc.opt -ffc \
    $(vwrays -d $vw) -m sky_glow -m ground_glow -V+ \
    -o images/rtcontrib-V+_%s.hdr testroom_overcast.oct \
    > tmp/catch_all.hdr
```

Digging deeper into `man rtcontrib`, we encounter this phrase:

> "The most recent `-b`, `-bn` and `-o` options to the left of each `-m` setting are the ones used for that modifier."

So the `-o` option must precede the `-m` options. We've just done this the wrong way 'round.

```
$ vwrays -ff $vw |rtcontrib @rtc.opt -ffc \
    $(vwrays -d $vw) -V+ -o images/rtcontrib-V+_%s.hdr \
    -m sky_glow -m ground_glow testroom_overcast.oct
```

This works. Great! The two images, shown in Figure 8a and 8b, can be combined with `pcomb` now. The correct use of `pcomb`, more specifically, its `-s` (or `-c`) options which are applicable to the individual images, is going to be critical.

> "By default, the result is just a linear combination of the input pictures multiplied by `-s` and `-c` coefficients, ..."

For now, we simply add the two images. This is equivalent to `-s 1`, which is the default behaviour of `pcomb`.

```
$ pcomb images/rtcontrib-V+_sky_glow.hdr \
    images/rtcontrib-V+_ground_glow.hdr \
    > images/rtcontrib-V+_skyglow+groundglow.hdr
```

Well done. We've reached an important milestone. We have generated individual images for different light source materials, and managed to combine them in such a way that the absolute values found in the combined image is identical to the original `rpict` and `rtrace` images, except for the graininess.

3.3 Contributions from Different Parts of the Sky

The technical term for 'contributions from different parts of the sky' used in the `rtcontrib` man page is called 'binning' where each 'bin' represents one sky patch. We have briefly mentioned the special `%d` string to `rtcontrib`'s `-o` option, and are now going to make use of it.

But before we can go ahead, a slight change to our model is required. The use of a 'normal' sky, i.e. a sky whose distribution is taken care of by `gensky`, has been convenient. For DDS, more footwork is required. We'll need to define the sky distribution through a clever use of the `-s` multipliers with `pcomb` during the post-processing stage of the results. During the simulation, the sky should be just white, without any distribution attached to it. This is also true for the ground.

```
$ cat skies/sky_white.rad
void glow sky_glow  0  0  4 1 1 1 0
sky_glow source sky  0  0  4 0 0 1 180
void glow ground_glow  0  0  4 1 1 1 0
ground_glow source ground  0  0  4 0 0 -1 180
```

To avoid confusion, we also make a new octree and call it *testroom_whitesky.oct*:

```
$ oconv materials/testroom.mat objects/testroom.rad \
    skies/sky_white.rad > testroom_whitesky.oct
```

To define the bins based on the part of the sky they emanate from, we need some clever algorithm. This already done and comes in the form of a *cal* file. The *cal* file that we need for binning the sky hemisphere is called *tregenza.cal*. Let's see what it does. The key to success is to change the output format (`-o` option) on the `rtcontrib` command line, so that file names have the bin number in them. This is done with the `%d` string. If you want those numbers to be all of the same length, you can left-fill them with zeros to three places, in which case `%d` becomes `%03d`. We will ignore the ground material this time, and only use the sky.

Before we look at the image from the view points that was used in the images above, we render a stereographic view that only shows the sky hemisphere.

```
$ cat views/sky.vf
rvu -vts -vp 0 0 5 -vd 0 0 1 -vu 0 1 0 -vh 180 -vv 180 \
    -vo 0 -va 0 -vs 0 -vl 0
$ vw="-x $DIMS -y $DIMS -vf views/sky.vf"
$ vwrays -ff $vw |rtcontrib @rtc.opt -ffc \
    $(vwrays -d $vw) -V+ -f tregenza.cal -b tbin \
    -o images/patches/p%03d.hdr \
    -m sky_glow testroom_whitesky.oct
```

To show the Tregenza patches (or bins), we can combine the 145 individual images with random colours. The result can be seen in Figure 9. Please see the note on random colours and `rcalc`'s rand() operator in Appendix A.1 to understand what the next command line does. It looks a bit intimidating.

```
$ pcomb $(for i in {000..145}; do \
```

Figure 9: Visualising the Tregenza sky patches trough random colours

```
    echo "-c $(rcalc -e \
    '$1=rand(recno);$2=rand(recno+.3);$3=rand(recno+.7)') \
    images/patches/p$i.hdr"; done) \
    > images/sky_colourcomb.hdr
```

The patch images from the fisheye view of the sky were only used to show the Tregenza patches and are not needed any more. We'll use the same file names for a different view. The `-V+` switch is no longer required but makes no difference. We will have to derive the required per-image `-s` factors for `pcomb` ourselves from now on.

```
$ vw="-x $DIMS -y $DIMS -vf views/back.vf"
$ vwrays -ff $vw |rtcontrib @rtc.opt -ffc \
    $(vwrays -d $vw) -f tregenza.cal -b tbin \
    -o images/patches/p%03d.hdr -m sky_glow \
    testroom_whitesky.oct
rtrace: warning - no light sources found
rtcontrib: system - cannot open \
    'images/patches/p000.hdr' for writing: File exists
```

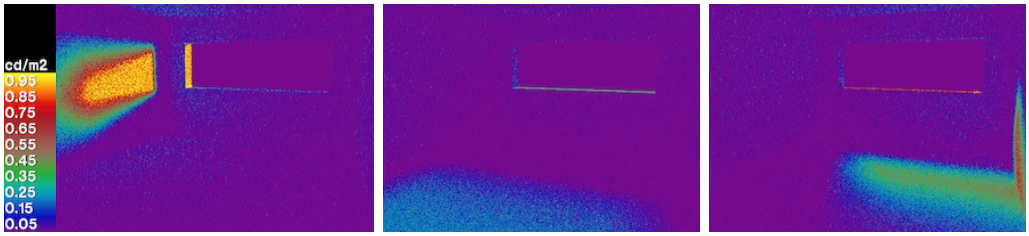
We have encountered the 'warning - no light sources' message many times before are not at all irritated by it and simply turn it off with the `-w` option. This is an option of `rtrace`, not of `rtcontrib` and needs to be placed at the end of the `rtcontrib` command line, just before the octree.

However, there is a new error message now. From `man rtcontrib`:

> "An existing file of the same name will not be clobbered, unless the `-fo` option is given."

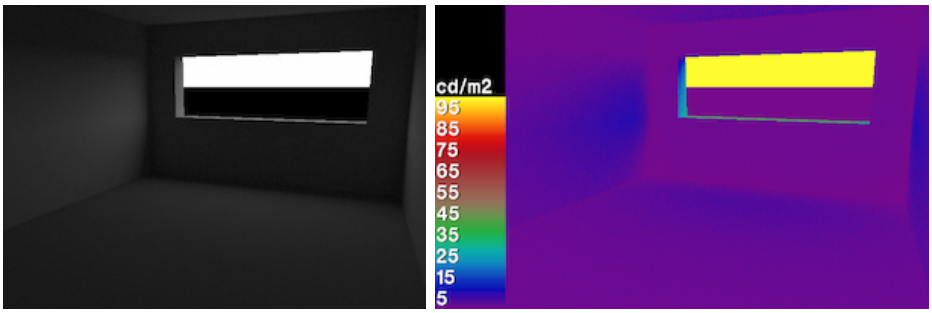
'to clobber' is computer-speech for 'to overwrite'. This means that `rtcontrib` will refuse to touch the file if it already exists. The way around this is to use the `-fo` option, forcing `rtcontrib` to overwrite any existing file, or to delete those files first. This is a built-in safety mechanism. You might have half-finished output files here as the result of a crash. In such a case, you will want to continue from when the crash happened (`-r` option), rather than re-run all calculations again.

If you think that you're having a deja-vu here, you might well be right. We are already using two other `rtcontrib` options that start with `-f`: One for the *cal* file (this



(a) Patch 020 (b) patch 046 (c) patch 071

Figure 10: Examples from illumination from different parts of the sky



(a) Normal image (b) False colour representation

Figure 11: Combined image with illumination from sky and ground

is like the `-f` in `rcalc`), the other to define the data representation (here: `-ffc`). To put it in other words: the 'f' in 'minus-f' can mean:

- force;
- format;
- (cal) file.

Guess it could be worse: There could be twenty different flavours of `-f`, not just three... The reason for having all the f-options is to have a certain consistency with the option names in the other Radiance rendering commands.

Anyhow, we re-run `rtcontrib` with the *views/back.vf* view and combine the patch images. Some of the patch images are shown in Figure 10:

```
$ pcomb images/patches/*.hdr > images/combined_whitesky.hdr
```

Compared to all of the previous images, this combined one, as shown in 11, is much darker, but expectedly so. After all, we removed the skyfunc modifier from the glow sources of the sky. This results in a uniform sky distribution where each part of the sky has a radiance of $1\,\mathrm{W}\cdot\mathrm{m}^{-2}\cdot\mathrm{sr}^{-1}$.

What we need to do next is work out the `-s` multipliers for `pcomb`, so that the combined patch images accurately represent the desired sky distribution.

3.4 Combining the Patch Images

There are a number of different ways of working out the `-s` factor. Because our aim is to fully understand what is going on, we will determine them manually first, before moving on to Radiance tools that can help us automate this process.

3.4.1 Working Out the Multipliers with Pen and Paper

Our simulations so far have left us with a set of 145 images that, when combined, result in a uniform sky of radiance $1\,\mathrm{W}\cdot\mathrm{m}^{-2}\cdot\mathrm{sr}^{-1}$. To do this for any other sky distribution, we need to determine the actual radiance of the sky element. Because its brightness only changes with altitude and not with azimuth, we'll use an overcast sky here, so once we're done, the resulting super-imposed image should look exactly as the ones above.

Here are two code snippets from *gensky.c* that tell us how Radiance calculates the sky brightness from the sun's altitude. We are using the term "brightness" somewhat loosely here, actually referring to the "radiance". The last line in the code snippet below shows the parameters for the brightfunc modifier. An overcast sky has type 2.

```
if (overcast)
    zenithbr = 8.6*sundir[2] + .123;
...
if (overcast)
    printf("3 %d %.2e %.2e\n", \
            skytype, zenithbr, groundbr);
```

Below is a `gensky` command line that generates a 10,000 lx overcast sky. The zenith brightness can be found in the very last line that holds all float parameters of the sky's brightfunc primitive.

```
$ gensky 3 21 10 -c -a 51 -o 0 -m 0 -b 22.86
# gensky 3 21 10 -c -a 51 -o 0 -m 0 -b 22.86
# Local solar time: 9.88
# Solar altitude and azimuth: 31.9 -38.5
# Ground ambient level: 17.8
void brightfunc skyfunc
2 skybr skybright.cal
0
3 2 2.29e+01 3.56e+00
```

So the second parameter, 22.9 in this case, is the zenith brightness. Just what we need. The brightness for any part of the overcast sky is a function of the altitude:

$$L = L_z \frac{1 + 2\sin(altitude)}{3}$$

We can use `rcalc` to extract the zenith brightness:

```
$ gensky 3 21 10 -c -a 51 -o 0 -m 0 -b 22.86 \
    |rcalc -i '3 2 ${zenithbr} ${groundbr}' \
    -e '$1=zenithbr'
```

Next, we have to determine the patch elevation from its bin number. Radiance's *tregsrc.cal* does that for us:

```
$ cnt 146 |rcalc -f /usr/local/lib/ray/tregsrc.cal \
    -e 'Tbin=$1;$1=Talt'
0.104719755
0.104719755
0.104719755
...
1.36135682
1.36135682
1.57079633
```

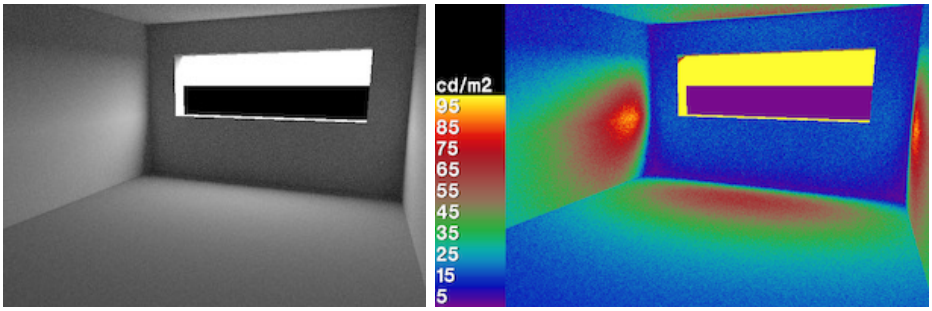
The elevation angles are in radians, so the last one, patch 145, has an elevation of $\pi/2$ or 90°. Looks about right. Note that `rcalc` does not search in RAYPATH to find the `-f` files because it is not technically part of the rendering toolkit, so we need to give it the full path to the *cal* file.

To make things a little more convenient, we get `rcalc` to scan the bin number from the file names. This was put there with the `%03d` string which we used with the `-o` option in `rtcontrib`.

```
$ ls images/patches/*.hdr |rcalc -i \
    'images/patches/p${pnum}.hdr' -e '$1=pnum'
0
1
2
...
144
145
```

So let's wrap this all up in a little BASH script. This script is included in the download that comes with this tutorial.

```
#!/bin/bash

# apply_overcast.bash
#
# Work out the -s multipliers for pcomb to combine a set of 145
# DC images so they define a 10klx overcast sky
#
# (c) Jan 2010, Axel Jacobs

# Determine zenith brightness for 10klx overcast sky
zenithbr=$(gensky 3 21 10 -c -a 51 -o 0 -m 0 -b 22.86 \
        |rcalc -i '3 2 ${zenithbr} ${groundbr}' -e '$1=zenithbr')

# Path to the patch images
imgpath="images/patches"

for i in $imgpath/*.hdr; do
    # Compute per-image multiplier, based on patch altitude
    multi=$(ls $i \
            |rcalc -i "$imgpath/p\${pnum}.hdr" \
            -f /usr/local/lib/ray/tregsrc.cal \
            -e "Tbin=pnum;\$1=(1+2*sin(Talt))/3*$zenithbr")
    echo -n " -s $multi $i"
done

#EOF
```



(a) Normal image (b) False colour image

Figure 12: Combining the patch images to represent an overcast sky

You run it like so:

```
$ pcomb $(bash apply_overcast.bash) \
    > images/combined_overcast.hdr
```

... and voila! The new image, *combined_overcast.hdr*, is shown in Figure 12. It looks just like we want it to.

It's probably a good idea to put the ground back into our scene. However, if we simply add the `-m ground_glow` to the `rtcontrib` command line in addition to the `-m sky_glow`, we are back at the two-images-in-one problem, just this time with each of the 145 patch images. There are two ways to fix this:

Add a `%s` string to the output format:

```
$ vwrays -ff $vw |rtcontrib @image.opt -ffc \
    $(vwrays -d $vw) -f tregenza.cal -b tbin \
    -o images/patches/p%03d%s.hdr -m sky_glow \
    -m ground_glow -w testroom_whitesky.oct
```

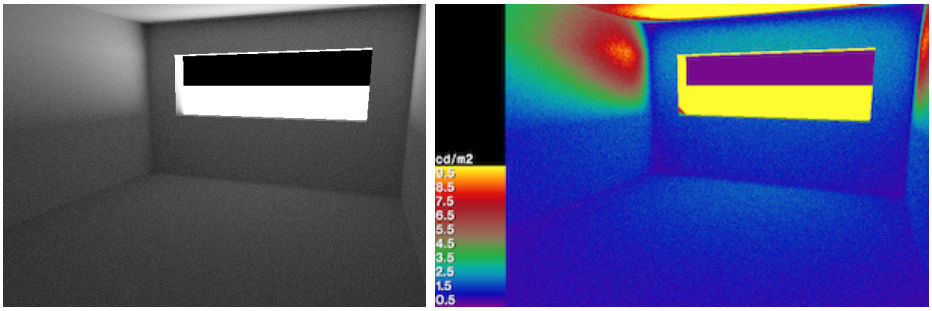
This would then create $2 \cdot 146$ images:

- *images/patches/p000sky_glow.hdr* to *images/patches/p145sky_glow.hdr*, and
- *images/patches/p000ground_glow.hdr* to *images/patches/p145ground_glow.hdr*

The binning that we are doing with *tregenza.cal* doesn't actually care about the materials involved here. Moreover, our way of working out the `-s` factors for `pcomb` is independent of them, too, and only ray directions are important.

This allows us to modify *sky_white.rad* and change the modifier of the ground source to sky_glow.

```
$ cat skies/sky_white1.rad
void glow sky_glow  0  0  4 1 1 1 0
sky_glow source sky  0  0  4 0 0 1 180
sky_glow source ground  0  0  4 0 0 -1 180
$ oconv materials/testroom.mat objects/testroom.rad \
    skies/sky_white1.rad > testroom_whitesky1.oct
```



(a) Normal image (b) False colour representation

Figure 13: Illuminance from patch zero, which is the ground

We could alternatively remove the ground source altogether and instead extend the angle of the sky source so that it covers the full 360°:

```
$ cat skies/sky_white2.rad
void glow sky_glow  0  0  4 1 1 1 0
sky_glow source sky  0  0  4 0 0 1 360
```

This lets us get away with only the `-m sky_glow`, yet produces an image for patch zero, the ground, that is not entirely black as before. This is shown in Figure 13.

```
$ vwrays -ff $vw |rtcontrib @rtc.opt -ffc \
    $(vwrays -d $vw) -f tregenza.cal -b tbin \
    -o images/patches/p%03d.hdr -m sky_glow \
    -w testroom_whitesky1.oct
```

3.4.2 Sampling the Sky with *tregsamp.dat*

Our first method of determining the `-s` factors for `pcomb` has been a little inflexible. Without significantly altering the `apply_overcast.bash` script, we are stuck with an overcast sky distribution. It's also somewhat crude in the sense that the patch brightness is determined from just the elevation. This makes it difficult to adapt for distributions that do depend on the azimuth, too.

There is actually no need for us to figure out the sky brightness distribution ourselves. Radiance can do this much faster with `gensky` and `gendaylit`. All we need to do is send out sample rays into different directions and 'measure' the sky brightness.

A set of sample rays is included with Radiance in *tregsamp.dat*

```
$ cat /usr/local/lib/ray/tregsamp.dat |tail -4
0 0 0 0.0484951248 0.0135990803 0.998730839
0 0 0 0.0515372887 0.062926967 0.996686563
0 0 0 -0.0338406476 -0.0284913061 0.999021049
0 0 0 -0.0471448632 0.0750632905 0.996063685
```

The total number of rays in this file is

```
$ cat /usr/local/lib/ray/tregsamp.dat |wc -l
9344
```

which makes it 64 samples for each of the 145 Tregenza patches and the ground: 9344/146 = 64. This enables us to derive an average across that part of the sky.

```
$ gensky 3 21 10 -c -a 51 -o 0 -m 0 -b 22.86 \
    |oconv - skies/sky.rad > tmp/overcast.oct
$ rtrace -w- -h- < /usr/local/lib/ray/tregsamp.dat \
    tmp/overcast.oct |total -64 -m
3.66846014 3.66846014 3.66846014
8.41266244 8.41266244 8.41266244
8.57949702 8.57949702 8.57949702
...
22.478328  22.478328  22.478328
22.4648664 22.4648664 22.4648664
22.8553616 22.8553616 22.8553616
```

We are using Radiance's `total` command here to output the mean (average) over every 64 lines of input. Because our sky is not coloured, we only need one of the columns. We output the result to a temporary file, *tmp/skydist.dat*

```
$ rtrace -w- -h- < /usr/local/lib/ray/tregsamp.dat \
    tmp/overcast.oct |total -64 -m |cut -f1 \
    > tmp/skydist.dat
```

To get the relevant multiplier, we look up the record (line number) in this file corresponding to the patch number. This is done in the BASH script `apply_tregsamp.bash`

```
1  #!/bin/bash
2
3  # apply_tregsamp.bash
4  #
5  # Work out the -s multipliers for pcomb from rtrace sky brightness
6  # measurements using tregsamp.dat
7  #
8  # (c) Jan 2010, Axel Jacobs
9
10 # Path to the patch images
11 imgpath="images/patches_illu"
12
13 for i in $imgpath/*.hdr; do
14     # Scan patch number from file name
15     patch=$(ls $i \
16             |rcalc -i "$imgpath/p\${pnum}.hdr" \
17             -e "\$1=pnum")
18     # Get multiplier from dat file, based on record number
19     multi=$(cat tmp/skydist.dat \
20             |rcalc -e "diff=$patch-recno+1;absdiff=if(diff,diff,-1*diff); \
21             $1=if(absdiff,0,\$1);" |total)
22     echo " -s $multi $i"
23 done
24
25 #EOF
```

It's used like so:

```
$ pcomb $(bash apply_tregsamp.bash) \
    > tmp/combined_tregsamp.hdr
```

When combining many images with `pcomb` like we are doing here, all the headers from the individual images are kept and included in the header of the combined image, which can grow quite large. This needn't worry you too much, but you might wish to reduce the header size with `pcomb`'s `-h` option.

```
$ pcomb -h $(bash apply_tregsamp.bash) \
    > images/combined_tregsamp.hdr
```

Using the `grep` command, we see that the original head is indeed over one thousand lines long, and that `pcomb`'s `-h` option has reduced it to just a few.

```
$ strings tmp/combined_tregsamp.hdr |grep -m 1 -n ^-Y
1028:-Y 430 +X 600
$ strings images/combined_tregsamp.hdr |grep -m 1 -n ^-Y
6:-Y 430 +X 600
```

This is working rather nicely now, but we need to address another problem. The technique based on *tregsamp.dat* appears to be flexible enough for determining the patch multipliers for any sky. But what about the sun? We need to find a way to somehow include the sun's contribution, but this needs to be based on the same patches that are used for the sky.

3.4.3 Letting `genskyvec` Do the Job

A quick-n-dirty approach to this could be to simply work out which one of the 145 sky patches is closest to the sun, and to make this patch much brighter. Obviously, the overall radiance needs to remain the same, so we need to account for the different solid angles of the sun (it's 0.5° across) and the much larger one of the sky patch.

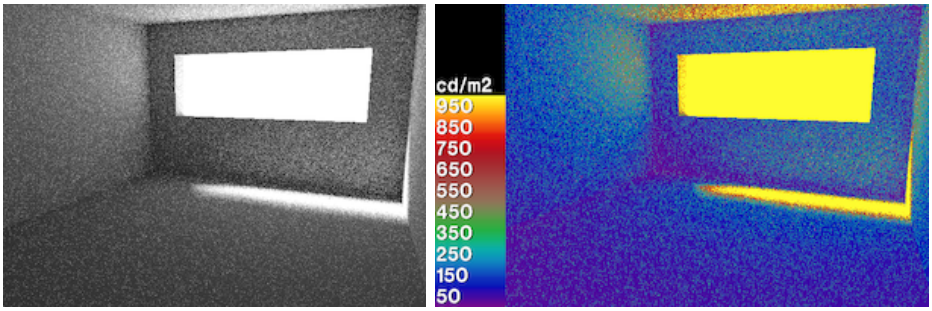
While this will work, it's actually better to split the sun's radiation across the three or four nearest sky patches. The relative weight of each patch is then calculated from how far its centre is away from the position of the sun. The sum of the radiance of those three patches multiplied by their solid angles must be equal to the sun's radiance multiplied by its solid angle.

$$R_{sun}\Omega_{sun} = \sum_{n=1}^{3} \Omega_i R_i$$

This is all a bit complicated, but luckily Radiance has a tool for this. It's called `genskyvec`. `genskyvec` takes a `gensky` or `gendaylit` generated sky on STDIN and works out the patch multipliers. If there is a sun, `genskyvec` makes sure that its radiant energy is distributed amongst the three nearest sky patches.

The output format of `genskyvec` is identical to what we created in the earlier exercise with `rcalc` and *tregsamp.dat*, so we can use `apply_tregsamp.bash` again. Figure 14 shows an image for a sunny sky that demonstrates how `genskyvec` takes care of the direct solar component.

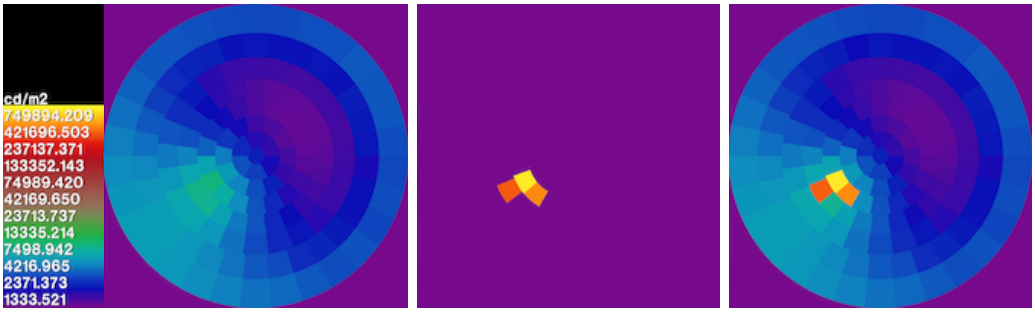
```
$ gensky 4 21 +10 +s | genskyvec -m 1 \
    -c 1 1 1 > tmp/skydist.dat
$ pcomb -h $(bash apply_tregsamp.bash) \
    > images/combined_skyvec.hdr
```



(a) Normal image (b) False colour representation

Figure 14: Combining the patch images to represent a clear sky

Examples of such patch intensities are shown in Figure 15. We normally only need the one for global (sky + sun), but I have separated them here to show what is happening.



(a) Diffuse (b) Direct (c) Global

Figure 15: The intensity of the individual sky patches is a combination of the diffuse sky, as well as the sun

The output from `genskyvec` is one R,G,B triple per line. `gensky` actually adds some colour, so the sky will be blueish, unless you tell it otherwise with the `-c R G B` option. If you do, then make sure the weighted average of the three channels is 1.0. If you would like to play with different colours, feel free to use the Radiance Colour Picker over on LUXAL. It does the normalisation for you, just copy and past the result.

The only other option that `genskyvec` understands deserves to be mentioned, too. This is the `-m` option. It takes an integer argument defaulting to four. To achieve a finer sky subdivision, each of the Tregenza patches is further sub-divided into m by m patches. The width of the Tregenza patch (it's extend in azimuth) and its height (its extend in altitude) are divided by the same number. This naturally results in m^2 sub-patches. With the default `-m 4`, 2305 patches are created for the sky plus one for the ground. Like the ground, the zenith patch is not subdivided: $2306 = 144 \cdot 16 + 1 + 1$

```
$ for i in {1..4}; do gensky 3 21 10 |genskyvec -m $i \
    |wc -l; done
146
578
1298
```

```
2306
```

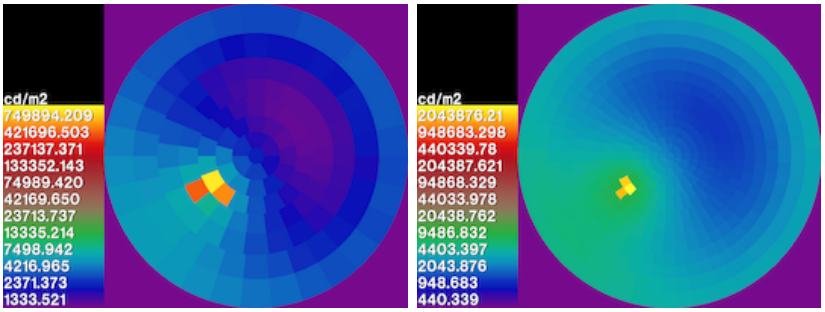
The more finely-grained the sky subdivision, the more accurately the sun can be represented, so this `-m` options is quite nice to have. Remember that this will not result in much extra render time, so other than the additional storage space required for the images, there is no drawback. Just ensure that the images are run-length encoded.

There is one show stopper to very fine Reinhart sub-divisions, however. As of March 2010, `pcomb` can't handle more than 1024 input files. See this thread on the radiance-dev mailing list from August 2009. So either stick to `-m 2`, or combine the images in stages. 578 patches isn't too bad if you think about it. This equally means that genskyvec CANNOT be run with the default `-m`, and that any invocation of *reinhart.cal* MUST set the `MF` variable (see below).

To be able to use the `genskyvec` functionality with small sky patches, we need to generate our patch images so that they match the finer sky subdivision. For this, simply replace *tregenza.cal* with *reinhart.cal* on the `rtcontrib` command line. The bin variable in *reinhart.cal* is `rbin`, not `tbin` as it is in *tregenza.cal*. Due to limitations with `pcomb` (and most like with your system's resources, too), you need to set the `MF` variable to either 1 or 2 before loading the *cal* file.

```
$ vwrays -ff $vw |rtcontrib @rtc.opt -ffc \
    $(vwrays -d $vw) -e MF:2 -f reinhart.cal \
    -b rbin -o images/patches/p%03d.hdr \
    -m sky_glow -w testroom_whitesky1.oct
```

Figure 16a is the same fisheye image that we have already seen. It shows the 'classic' Tregenza sky divisions. In Figure 16b, we have the same sky distribution, but each Tregenza patch is further sub-divided into four smaller patches. The `falsecolor` scales are different to show that the three 'sun patches' are now much brighter because their solid angle is smaller.



(a) 145 Tregenza patches

(b) Each Tregenza patch is sub-divided into four smaller patches

Figure 16: The finer the sky subdivision, the better the sun can be represented

Just so you don't get confused: There are two files with very similar names in your RAYPATH directory, *reinhard.cal* and *reinhart.cal*. Erik Reinhard is an HDR researcher who developed a global tone-mapping operator which is implemented in *reinhard.cal*. Christoph Reinhart, on the other hand, is a daylighting researcher, and it was him who came up with the idea of sub-dividing the Tregenza patches so that a finer representation of the sky can be achieved. So make sure you invite the right person over for a cup of tea.

3.5 Identifying Bottlenecks

With the very simple geometry of the model and the original render options (`-ad 1024`, `-lw` default), one run takes about 80 seconds if the sky patches follow the Tregenza layout (`-e MF:1`) on my dual core computer. If the sky is subdivided further, the run time increases a lot: 215 seconds for `-e MF:2`, and 499 seconds for `-e MF:3`. This is somewhat unexpected because the same number of rays are traced, and the render options are identical, too.

We calculated earlier on the size of each file. It's about 1 MB. 146 files therefore occupy 146 MB. A finer sky sub-division bring us to 578 MB or even 1.3 GB of file storage requirements. Assuming that there is no additional overhead in `rtcontrib` if the number of output files is increased (which is not quite the case but almost), 1.3 GB need to be written to disk in 80 seconds. The sustained transfer rate would thus have to be 1298 MB/80 s = 16 MB/s. Modern hard disk are rated at well over 100 MB/sec, but this is only true for small files that fit in the drive's buffer of typically 8 or 16 MB. As the amount of data increases, this buffer becomes less useful. Moreover, if many different files are being written to at the same time, the drive's seek time becomes a bottleneck. Mechanical hard disks prefer one large file over many smaller ones because the read/write heads need to frantically move back and forth for each of the files.

All this has the effect that the files can't actually be written as fast as the pixel information is generated, and that `rtcontrib` takes much longer than what is necessary for the ray tracing.

With this many files, it is far more efficient if we store them in RAM instead of writing them to hard disk immediately. This can be done with a RAM disk[1]. Creating such a RAM disk is as easy as making a directory, which can be anywhere, and issuing one simple command as root:

```
$ mkdir ramdisk0
$ sudo mount -t tmpfs none ramdisk0 -o size=2g
```

Make sure the RAM disk is large enough to hold all your files. You can be generous, though, because the RAM disk will only occupy as much of your system's main memory as is required. Here is a quick test without and with RAM disk:

```
$ for mf in 1 2 3; do time vwrays -ff $vw \
    |rtcontrib -fo @image.opt -ffc $(vwrays -d $vw) \
    -e MF:$mf -f reinhart.cal -b rbin \
    -o images/patches/p%04d.hdr \
    -m sky_glow -w testroom_whitesky1.oct; done
real 1m20.403s
user 1m5.564s
sys  0m3.532s

real 3m34.780s
user 1m46.552s
sys  0m5.986s

real 8m19.158s
```

[1] This in on LINUX, I don't know about OS X and Windows.

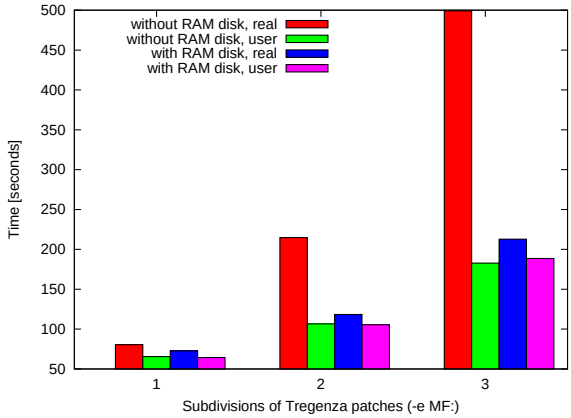


Figure 17: A RAM disk could speed up `rtcontrib` simulations by eliminating the hard disk bottleneck.

```
user 3m2.851s
sys 0m10.316s
$ for mf in 1 2 3; do time vwrays -ff $vw \
    |rtcontrib -fo @image.opt -ffc $(vwrays -d $vw) \
    -e MF:$mf -f reinhart.cal -b rbin \
    -o ramdisk0/p%04d.hdr \
    -m sky_glow -w testroom_whitesky1.oct; done
real 1m12.624s
user 1m4.412s
sys  0m3.042s

real 1m58.391s
user 1m45.518s
sys  0m3.909s

real 3m32.798s
user 3m8.595s
sys  0m5.457s
```

This is visualised in the bar chart in Figure 17. The bars labelled "user" indicate how long the rendering takes. Those labelled "real" show how much time actually goes by. Note the large discrepancy between "real" time and "render" time that we get with many open files if we don't use the trick with the RAM disk.

It is obviously important to move the data out of the RAM disk as soon as possible once the calculations are done. It would not survive a system crash or restart. If your scene is more complex than our little test room, and if your render settings are higher (which they should be for daylighting), it is less likely for the hard disk transfer to be a problem. On the other hand, computers are getting fast in clock speed, and the number of processing cores increases, too. The transfer rates of (mechanical) hard disks goes up by a much smaller rate, and it will be a while before large solid state disks, SSDs, are available in large sizes and at affordable prices. So if your system becomes very unresponsive during a large `rtcontrib` job, try using a RAM disk.

4 Dynamic Daylight Simulations

Most countries have building regulations stating the requirements for illuminance levels that need to be achieved at the working plane. This typically refers to the level of the desk. All such guidelines have been developed and honed over the last one hundred years, and refer to illuminance levels achieved by artificial lighting.

If there are any guidelines for daylighting in your country, they will probably just include a phrase such as

> "For the actual illuminance levels, please refer to the guidelines for artificial lighting."

On the other hand, there are recommendations for good daylight design. Virtually all of them are based on the daylight factor, DF. The DF might be a good-enough metric for daylight design of building, such as sizing of the windows. Trouble is that the DF is a RELATIVE measure, and only tells us about the internal illuminance with regards to the outside one. It does not tell us how, for any particular sky distribution, how this can be related to an ABSOLUTE indoor illuminance. This is because the DF is based on an overcast sky which, even in the 'cloudy' UK, does not occur very often. We will refer to daylighting simulations that are based on the DF approach as "static simulations".

With rising energy prices, it is increasingly important to design a building for the lowest possible energy consumption. From the lighting perspective, we need to answer questions such as

> "Over a typical year, how much of the required working plane illuminance can be achieved through daylight alone, and how much electric energy is needed for supplementary artificial lighting?"

A 'typical year' is site-dependent. This includes not just the geographical location, but also the local micro-climate. Luckily, historical weather data for many hundreds to locations all over the world is freely available from the EnergyPlus web site [10]. This data is based on measurements taken for each hour of the year, so there are 8760 data sets in a weather file. Such a meteorological data set includes irradiation data.

Using such weather data, we can answer the question posed above. This is called Dynamic Daylight Simulation, DDS. A DDS is carried out in several steps:

1. Derive a sky model from the irradiation data for each hour;

2. Decide whether the time step lies within the working hours, e.g. between 9:00 and 18:00 on a Monday to Friday;

3. If it does, run a Radiance simulation and derive illuminance readings for each of the sensors points that we are interested in;

4. Count how many of those readings satisfy (at least partially) the required illuminance levels;

5. Work out how much artificial light is required to supplement the daylight, and how much it is going to cost.

Let's do a quick back-of-an-envelope calculation. The year has 8760 hrs. Half of those hours are at night, the other half at daytime. If our working hours are from nine to six (9 hrs) on Monday to Friday, this is $9 \cdot 5/(7 \times 24) = 27\%$ of the week. Assuming that all working hours are during the day (which they are not, especially in winter), this leaves us with $0.27 \times 8760 = 2365$ hrs for which we need to run a Radiance simulation.

This is, of course, assuming that each sky is unique. If we somehow manage to group the skies, this number can be reduced. The problem is that any non-overcast sky is defined not only by its 'cloudyness', but also by the position of the sun which changes throughout the day and with the season. So the chances of any grouping are slim, and we are likely to still end up with a large number of simulations.

This is, you will have guessed by now, exactly where Radiance's Swiss army knife, `rtcontrib`, comes in. Here is what we do:

Rather than looking at the illuminance from the whole sky, we break up the sky into small segments. For each segment, we work out its contribution towards the internal illuminance. These contributions are known as Daylight Coefficients, DC [9].

There are many different ways to split a hemisphere into smaller parts, but a commonly used procedure that was developed by Peter Tregenza [8] uses 145 patches that are arranged in eight rings of equal altitude. If we calculate the RELATIVE contribution from each part of the sky, the DC, then all we need to do is multiply this value by the average brightness of that part of the sky for any sky distribution we choose to come up with its ABSOLUTE contribution. The 145 partial illuminance values are then simply added up.

The beauty here is that the DCs only need to be calculated once. Moreover, we don't even need to run 145 Radiance simulations–with the help of `rtcontrib`, this can be done with just one simulation! So the mind-boggling number of 2000+ is reduced to 145, and then to just one.

4.1 Processing Weather Data

As mentioned above, good quality weather data is available from the EnergyPlus web site courtesy of the US Department of Energy [10]. Go and grab the ZIP file that is closest to your location. I am going to use the London Gatwick one. The file that we need has a *.epw* extension. Have quick look at it in a text editor. There are eight header lines which are of no interest to us, except maybe for the site's latitude and longitude. After the header, we have 8760 lines of data. The columns are comma-separated. There is lots of climatic data in here, but we only need the radiation data. The table below shows what radiation data is available in an EPW file.

To decide which variables we need, we need to have a look at the `gendaylit` command. `gendaylit` used to be available from the website of Fraunhofer ISE (Institute of Solar Energy), but was merged into Radiance proper in 2009. It's syntax is mostly compatible with `gensky` in that it needs either month, day, hour as the first three arguments, or can be invoked with the `-ang` option, in which case the solar altitude and azimuth (Radiance convention, in degrees) are required.

For practical applications, `gensky` is limited by the sky distributions that it can generate: CIE overcast and clear, uniform, and intermediate. The uniform sky is of hardly any relevance these days except in the UK, where it forms the basis of the Rights-of-Light. The overcast and clear distributions may be seen as the two extreme cases of naturally occurring skies. Trouble is almost any 'real' sky is neither overcast nor clear–not even most 'cloudy' or 'sunny' skies where one could be forgiven for thinking

Column	Data	Unit
11	Extraterrestrial Horizontal Radiation	Wh/m^2
12	Extraterrestrial Direct Normal Radiation	Wh/m^2
13	Horizontal Infrared Radiation from Sky	Wh/m^2
14	Global Horizontal Radiation	Wh/m^2
15	Direct Normal Radiation	Wh/m^2
16	Diffuse Horizontal Radiation	Wh/m^2
17	Global Horizontal Illuminance	lux
18	Direct Normal Illuminance	lux
19	Diffuse Horizontal Illuminance	lux
20	Zenith Luminance	cd/m^2

Table 1: Radiometric and photometric data available in a EPW weather file [3]

that they are.

In other words: almost all 'real' skies are in-between ones, for which `gen`sky only has one distribution–the intermediate one. We need a more flexible sky model, one that can describe the subtleties of 'cloudyness' between 'overcast' and 'clear'.

This is exactly what `gendaylit` provides. It is based on the Perez All Weather model which is defined by three variables:

1. The solar zenith angle, Z
2. the sky clearness, ϵ
3. the sky brightness, Δ

`gendaylit` can derive the sky distribution from three different sets of input (note that the zenith angle must either be supplied directly, or can be derived from time, date and location):

-P the epsilon and delta parameters

-W direct normal irradiance and diffuse horizontal irradiance (in W/m^2)

-L direct normal illuminance and diffuse horizontal illuminance (in $lm/m^2 = \text{lux}$)

Deriving the epsilon and delta parameters for use with the `-P` invocation is quite complicated, and you are unlikely to need this. What we need for the other two options, `-W` and `-L`, is all available in the EPW file. However, not all weather stations actually measure the photometric data, in which case those quantities are derived from the radiometric data. We will leave this task to `gendaylit` and always use the `-W`.

Extracting the month, day, hour, direct normal irradiance and diffuse horizontal irradiance is rather simple. Below is just an exercises; we'll let a script handle this for us.

```
$ tail -n +9 weather/GBR_London.Gatwick_IWEC.epw \
    |rcalc -t, -e '$1=$2;$2=$3;$3=$4-1+0.5;$4=$15;$5=$16' \
    |tr ',' '\t'
...
1   1   7.5  0   0
1   1   8.5  0   19
1   1   9.5  7   51
1   1   10.5 17  88
1   1   11.5 22  109
1   1   12.5 22  110
...
```

4.2 Annual Light Exposure

All our `rtcontrib` exercises so far have produced images (luminances) in one way or another. This was to allow us to easily spot differences to our reference case, the `rpict` image, in the false colour representation.

About the only thing that DDS simulations are useful for with images is annual exposure values. lux-hour values are of interest in the museum sector where strict guidelines exist for delicate artwork to ensure the pieces don't age too quickly through high exposure to light.

Annually available irradiation levels, on the other hand (in $\mathrm{kWhrs \cdot m^{-2}}$ or $\mathrm{MJ \cdot m^{-2}}$) are of importance for the sizing, placement and orientation of solar panels. This is implemented in Francesco Anselmo's `radmap` program for Radiance [4]. `radmap` does not produce lux-hour images, so our next `falsecolor` image doesn't have a reference case, although we can do some quick calculations to make sure we're not too far off target.

Just like with `rpict`, illuminance images are generated by setting the `-i` option.

```
$ vwrays -ff $vw |rtcontrib -i -fo @rtc.opt -ffc \
    $(vwrays -d $vw) -e MF:1 -f reinhart.cal -b rbin \
    -o images/patches_illu/p%03d.hdr -m sky_glow \
    -w testroom_whitesky1.oct
```

To get at the annual cumulative exposure, we need to call `gendaylit` and `genskyvec` for each of the time steps.

There is one additional complication that we need to take care of. The Perez All-Weather model is derived from sky luminance measurements taken in Berkeley, CA, USA between June 1995 and December 1986 [6]. This means that although a large number of data points form the basis of the model (some three million), it doesn't necessarily cover all possible sky conditions in any part of the world. Shortly after publication of his paper in 1993, Perez issued an erratum [7] based on feedback from Paul Littlefair of the BRE in the UK. Littlefair validated the model against BRE sky scan data and found that it failed for a number of 'real' UK skies. This erratum is implemented in `gendaylit`.

Roughly at the same time (1991), the CIE set out on an ambitious project named International Daylight Measurement Programme, IDMP. The goal was to gather a large data set of international daylight and solar radiation data. It is not quite clear what the status of the IDMP is right now. The last update to the web site [2] happened in 1998. A few dozen organisations from many different countries are (or were?) joined

in the project. Fifteen of the weather stations were equipped with sky scanners and recorded sky luminance data in addition of all the other photometric and radiometric data.

Perez was aware of this development and of the shortcomings of his model. He stated in his 1993 paper [6]:

> "Of course, the validation performed here is dependent and will have to be repeated on independent data. It is possible that the model may require adjustment to account for this data base site specificity. The International Daylight Measurement Program initiated by the CIE and WMO will provide such as climatically diverse data base."

In a sense, this is what happened with the 2003 CIE General Standard Sky [1] which, like Perez' model, is based on a generalisation of the CIE clear sky formula. With the CIE General Standard Sky being derived from IDMP data, it can be expected to be more generally applicable than the Perez sky. However, the CIE standard does not explain how a sky model can be derived from measured irradiance or illuminance data. A large number of papers have been and still are being written addressing this problem. As of January 2010, no Radiance sky generator exists that implements the General Standard Sky that can provide the same convenience that `gendaylit` does. The actual distribution (Radiance *.cal* file) was implemented by Philip Greenup [!link].

Another aspect that needs some contemplation is the available of weather data and their quality. At present, the EnergyPlus site has 1890 weather files available for download. There is other freely available data, namely from SatelLight, but I haven't looked at it.

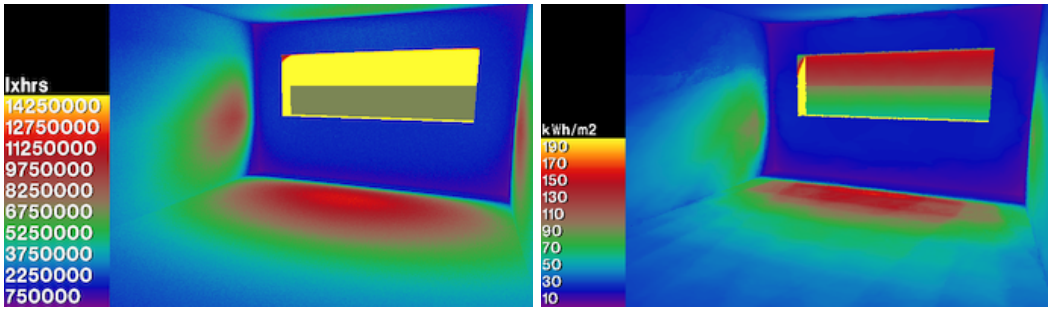
The quality of the radiation data varies. To help you get an idea of the data, I've run `gendaylit` with all of them. The results are listed in on an HTML page (requires JavaScript).

What I found somewhat disturbing is the percentage of time steps for which gendaylit fails to produce a valid Perez sky description. This is a serious problem which puts a big question mark behind the whole DDS-with-Radiance thing. What we need is a new sky generator based on the CIE General standard sky or an update to the Perez model. This generator should ideally be based on the vast data set from the IDMP. Sadly, however, this seems to be locked away.

Our script which generates a sky vector for each time step takes care of any errors produced that gendaylit might produce. It is listed below. It outputs one non-coloured sky vector per time step to *weather/gatwick.vecs*.

```
1  #!/bin/bash
2
3  # gen_yearvec.bash
4  #
5  # Call gendaylit and genskyvec for each hour in a weather file.
6  # gendaylit errors are handled and logged in an error file.
7  #
8  # (c) Jan 2010, Axel Jacobs
9
10 # Input: the EPW weather file
11 wea="weather"
12 epw="$wea/GBR_London.Gatwick.037760_IWEC.epw"
13 # We keep all gendaylit errors to ponder about them...
14 err="$wea/gatwick.err"
15 # Output: time series of sky vectors
```

```
16 res="$wea/gatwick_year.vecs"
17 # Output: month, day, decimal hour
18 dtm="$wea/timedate.csv"
19 rm -f $err $res $dtm
20
21 nullvec='0'
22 # Adjust of you use gendaylit with anything other than -m 1
23 for i in {1..145}; do
24     nullvec="$nullvec 0"
25 done
26 # Sky vector from the previous time step
27 prevvec=""
28
29 # Extract some information from the EPW header:
30 # Location is the first line the EPW header
31 loc=$(head -1 $epw)
32 lat=$(echo $loc |cut -d',' -f7)
33 longitude=$(echo $loc |cut -d',' -f8)
34 # EPW lists longitude + to the east, Radiance + to the west
35 lon=$(echo $longitude |rcalc -e '$1=-1*$1')
36 meridian=$(echo $loc |cut -d',' -f9)
37 # EPW lists time zone in hours, but we need degrees
38 mer=$(echo $meridian |rcalc -e '$1=$1*-15')
39 #echo "lat/lon: £lat/£lon, mer: £mer"
40
41 tail -n +9 $epw |rcalc -t, \
42         -e '$1=$2;$2=$3;$3=$4-1+($5/2)/60;$4=$15;$5=$16' \
43         |tr ',' '\t' |while read line; do
44
45     datetime=$(echo $line |cut -d' ' -f1-3)
46     dirnorm="$(echo $line |cut -d' ' -f4)"
47     diffhor="$(echo $line |cut -d' ' -f5)"
48     radiation="$dirnorm $diffhor"
49
50     # Out put date and time to separate file
51     echo "$datetime" |tr ' ' '\t' >> $dtm
52     # Output null vector at night
53     if [ "$radiation" == '0 0' ]; then
54         skyvec=$nullvec
55     else
56         # Assemble gendaylit command line
57         gdlcmd="$datetime -a $lat -o $lon -m $mer -W $radiation"
58         # Test run. See if gendaylit succeeds
59         gendaylit $gdlcmd >/dev/null 2>&1
60         # Did something go wrong?
61         if [ $? -eq 0 ]; then
62             # All fine. Generate sky vector, extract first column and transpose.
63             skyvec=$(gendaylit $gdlcmd \
64                     |genskyvec -c 1 1 1 -m 1 |cut -f1 |tr '\n' ' ')
65             prevvec=$skyvec
66         elif [ $dirnorm -eq 0 ]; then
67             # Error: gendaylit dropped out.
68             # Most errors in the Gatwick file occur at dawn or nightfall
69             # when the direct normal radiance is zero. Assume it's night.
70             # For Gatwick, this happens 190 times.
71             echo "$datetime: $radiation (assuming night time)" >> $err
72             skyvec=$nullvec
73         else
74             # This is a serious gendaylit error. Re-use previous sky vector.
75             # For Gatwick, this doesn't happen at all.
```



(a) Computed with `rtcontrib` (b) `radmap` result for comparison

Figure 18: Annual light exposure

```
76              skyvec=$prevvec
77              echo "$datetime: $radiation (re-using previous sky vector)" >> $err
78          fi
79      fi
80      echo $skyvec >> $res
81  done
82
83  #EOF
```

To get the annual cumulative values, we simply add up all columns with Radiance's `total` command, then transpose the vector back into its original columnar form so that we can use the `apply_tregsamp.bash` script again.

```
$ bash gen_yearvec.bash
$ cat weather/gatwick_year.vecs |total \
    |tr '\t' '\n' > tmp/skydist.dat
$ pcomb -h $(bash apply_tregsamp.bash) \
    > images/gatwick_yearly_i.hdr
```

Before we look at the result, let us make a `radmap` image of the same scene. A side-by-side comparison of our `rtcontrib` annual exposure image and the `radmap` result is shown in Figure 18.

```
$ mkdir radmap
$ cat materials/testroom.mat objects/testroom.rad \
    > radmap/all_in_one.rad
$ cp views/back.vf radmap/
$ cp weather/GBR_London.Gatwick.037760_IWEC.epw radmap/
$ cd radmap
$ radmap -s 8 -a 51 -o 0 -m 0 -n 1 -w \
    GBR_London.Gatwick.037760_IWEC.epw --albedo 0.2 \
    --viewfile back.vf --radfile all_in_one.rad \
    --prefix out -x 800 -y 800 --sky-resolution 128 \
    --max-kWh 200 --max-MJ 50 --keep-temporary-maps
$cd ..
```

`radmap` uses a different method for producing the annual yearly irradiance, so we can't really expect to get a perfect match. Let's do a quick comparison of the images.

The highest value of the bright patch on the wall close to the window is about ten million lux hours in the `rtcontrib` image, compared to roughly $100,000\,\mathrm{Whrs} \cdot \mathrm{m}^{-2}$. $1\,\mathrm{lux} = 1\,\mathrm{cd} \cdot \mathrm{m}^{-2}$, so both images show the power per unit area. The difference should then simply lie in the luminous efficacy which is fixed to $179\,\mathrm{lm} \cdot \mathrm{W}^{-1}$ in Radiance. Dividing the 10,000,000 lxhrs by $179\,\mathrm{lm} \cdot \mathrm{W}^{-1}$, we get roughly 56,000 Whrs.

4.3 Dynamic Daylight Performance Metrics

Ye good ol' daylight factor is a building performance metric for static daylight simulations under an overcast sky. One of the reasons why it's so popular is because it expresses the 'daylighting quality' of a space with just one number (typically the average DF). People are good with numbers, as long there is only one or two of them. What we get from a DDS is a whole bunch of numbers. Interpreting them is difficult, so how do you compare the daylight performance of a building against a benchmark? You need to boil down the bunch of numbers to just one or two. Maybe three.

Several daylight metrics have been proposed in recent years. They are all based on illuminance readings taken at selected points in the room, and they are all derived from yearly climatic data. This would normally mean that for each point, we get 8760 illuminance readings. However, for practical purposes, only the working hours are of interest. This still leaves us with some 2000+ values. Dynamic daylight metrics apply thresholds to those illuminances. Such binning might be as simple as "too low", "fine" and "too high". The results are expressed as a percentage of the working year, so that the 'quality' of the daylighting is described by just three numbers (technically, it's only two in this example, because between them, they add up to 100%).

Getting `rtcontrib` to produce point illuminance readings is done with the `-I` switch, just as it is with `rtrace`. The input points and directions are now taken from *data/photocells.pts*. I've placed one on the roof, too, so we can make some comparisons against the outside.

```
$ cat data/photocells.pts
0 0  3 0 0 1
2 1 .7 0 0 1
2 2 .7 0 0 1
2 3 .7 0 0 1
2 4 .7 0 0 1
2 5 .7 0 0 1
$ cat data/photocells.pts |rtcontrib -h -I -fo @rtc.opt \
    -e MF:1 -f reinhart.cal -b rbin \
    -o results/sensors/p%03d.dat -m sky_glow \
    -w testroom_whitesky1.oct
```

Each of the output files contains the R,G,B of the daylight coefficient for that sky patch for all sensors, one sensor per line.

```
$ cat results/sensors/p016.dat
4.633618e-03 4.633618e-03 4.633618e-03
3.000000e-04 3.000000e-04 3.000000e-04
2.616809e-03 2.616809e-03 2.616809e-03
4.933618e-03 4.933618e-03 4.933618e-03
6.905887e-03 6.905887e-03 6.905887e-03
```

```
6.733618e-03 6.733618e-03 6.733618e-03
```

The next script combines all those files and generates the sensor illuminance for each time step. The results are stored in a tab-sep CSV which we give a *.tmsr* extension (for time series).

```
#!/bin/bash

# gen_sensorillu.bash
#
# Take (modified) rtcontrib sensor readings and produce
# a time series file for each sensor.
#
# (c) Jan 2010, Axel Jacobs

sensors="data/photocells.pts"
sensor_dc="results/sensors"
sensors_vec="tmp/sensors.vec"
rm -f $sensors_vec
year_vecs="weather/gatwick_year.vecs"

# Work out how many sensor points we have
# Make sure there are no blank lines in that file!
npts=$(cat $sensors |wc -l)

# Convert to illu, combine DCs into a vector and transpose
for i in {000..145}; do
    cat results/sensors/p$i.dat |rcalc \
            -e '$1=179*($1*0.265+$2*0.67+$3*0.065)' \
            |tr '\n' '\t' >> $sensors_vec
    # Append a line break
    echo >> $sensors_vec
done

sensorfiles=""
# Loop over all sensors
for s in $(seq $npts); do
    # Extract and transpose the sensor vector
    cat $sensors_vec |cut -f$s |tr '\n' '\t' > tmp/tmp.vec
    echo >> tmp/tmp.vec

    tmpfile="tmp/sensor${s}_illu.dat"
    rm -f $tmpfile

    while read line; do
        # Multiply the time step vector with the partial sensor readings
        # and add them all up. This gives us the illu for each time step.
        echo $line |cat - tmp/tmp.vec |total -p |tr '\t' '\n' |total \
                >> $tmpfile
    done < $year_vecs

    sensorfiles="$sensorfiles $tmpfile"
    # Put the month, day, time columns in from to get a time series file
    # Dynamic daylight metrics can then be done in any spread sheet.
    cat weather/timedate.csv |rlam - $sensorfiles \
            > results/sensors_illu.tmsr
done

#EOF
```

Run it:

```
$ bash gen_sensorillu.bash
```

I've removed most of the 8760 lines and also some of the decimal places from the file listing:

```
$ cat results/sensors_illu.tmsr
...
1 1 7.5  0       0      0      0     0     0
1 1 8.5  2039.5  267.3  106.4  55.1  33.7  20.1
1 1 9.5  5576.7  765.0  310.9  162.4 99.8  60.5
1 1 10.5 9711.2  1530.3 730.1  391.2 214.1 120.0
1 1 11.5 12151.4 2021.9 1054.9 632.7 364.6 211.1
...
```

With this CSV file, it's straight-forward to filter by occupancy and bin the sensor illuminance readings to get at dynamic daylight metrics. As an example, we try out Michel and Scartezzini's Temporal Fraction of Satisfaction, TFS [5]. I just love that term.

```
1  #!/bin/awk -f
2
3  # gen_tfs.awk
4  #
5  # Derive Temporal Fraction of Satisfaction as an example for a
6  # dynamic daylight metric. This was proposed in:
7  # Laurent Michel, Jean-Louis Scartezzini: Implementing the partial
8  # daylight factor method under a scanning sky simulator.
9  # Solar Energy, 72(6):473-492, 2002.
10 #
11 # (c) Feb 2010, Axel Jacobs
12
13 BEGIN {
14 }
15
16 {
17     # Assume that we work every day from 9 to 5. You need to keep track
18     # of the weekdays if your building is not occupied Sat and Sun.
19     # $1: month, starting with 1
20     # $2: day, starting with 1
21     # $3: decimal hour, from 0.5 to 23.5 in one-hour steps
22     if ($3>9 && $3<17) {
23         # Keep track of the working hours. We need result as a percentage.
24         hours+=1;
25         # Loop over all sensors
26         for (i=4; i<=NF; i++) {
27             # Is illu above the minimum?
28             if ($i>500) min[i]+=1
29             # Is illu below the maximum?
30             if($i<1500) max[i]+=1
31             # Are both criteria satisfied?
32             if($i>500 && $i<1500) minmax[i]+=1
33         }
34     }
35 }
36
37 END {
38     for (i=4; i<=NF; i++) {
```

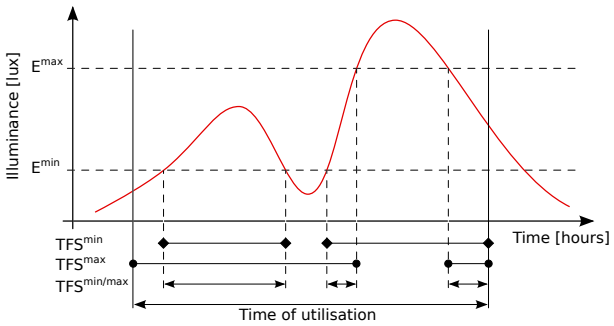


Figure 19: Explaining the concept behind TFS. After [5]

```
39          # Print the results in a nicely formatted list.
40          printf "Sensor%02d - TFS_min: %5.1f%%, TFS_max: %5.1f%%, \
41          TFS_minmax: %5.1f%%\n", i-3, min[i]/hours*100, \
42          max[i]/hours*100, minmax[i]/hours*100
43      }
44  }
45
46  #EOF
```

With the TFS being a fraction, I guess the authors intended it to be presented in the 0..1 range. I personally find percentages a little easier to read.

```
$ awk -f gen_tfs.awk results/sensors_illu.tmsr
S01 - TFS_min: 96.6%, TFS_max:  4.8%, TFS_minmax:  1.4%
S02 - TFS_min: 90.0%, TFS_max: 26.0%, TFS_minmax: 16.0%
S03 - TFS_min: 77.3%, TFS_max: 49.6%, TFS_minmax: 26.8%
S04 - TFS_min: 63.5%, TFS_max: 84.8%, TFS_minmax: 48.3%
S05 - TFS_min: 50.4%, TFS_max: 94.1%, TFS_minmax: 44.5%
S06 - TFS_min: 18.8%, TFS_max: 97.6%, TFS_minmax: 16.5%
```

Currently, there are no generally accepted must-do DDS metrics. A number of different yet similar metrics are being discussed, and so are the threshold values that are to be applied. If you want to get into dynamic daylight simulations, you have a lot of reading to do.

References

[1] CIE. Spatial distribution of daylight – CIE standard general sky. Iso 15469:2004(e)/cie s 011/e:2003, CIE, 2004.

[2] CIE. International Daylight Measurement Programme. Internet: http://idmp.entpe.fr/, Dec 2009.

[3] D. Crawley, J. Hand, and L. Lawrie. Improving the Weather Information Available to Simulation Programs. In *Building Simulation '99*, 1999.

[4] Francesco Anselmo. radmap, 2005.

[5] Laurent Michel and Jean-Louis Scartezzini. Implementing the partial daylight factor method under a scanning sky simulator. *Solar Energy*, 72(6):473–492, 2002.

[6] R. Perez, R. Seals, and J. Michalsky. All-weather model for sky luminance distribution–preliminary configuration and validation. *Solar Energy*, 50(3):235–245, 1993.

[7] Richard Perez, Robert Seals, and Joseph Michalsky. All-weather model for sky luminance distribution–preliminary configuration and validation: Erratum. *Solar Energy*, 51(5):423, 1993.

[8] P R Tregenza. Subdivision of the sky hemisphere of luminance measurements. *Lighting Res. Technol.*, 19:13–14, 1987.

[9] P. R. Tregenza and I. M. Waters. Daylight coefficients. *Lighting Res. Technol.*, 15(2):65–71, 1983.

[10] United States Department of Energy. Weather data. Internet: http://apps1.eere.energy.gov/buildings/energyplus/cfm/Weather_data.cfm, Dec 2009.

A Appendices

A.1 Random Colours

`rcalc`'s rand() operator produces the same 'random' number when called with the same seed.

```
$ for i in {0..2}; do rcalc -n -e '$1=rand(1)'; done
0.35008306
0.35008306
0.35008306
```

To seed the rand() operator with different values, we could use the current nanoseconds:

```
$ for i in {0..2}; do date +%N |rcalc \
    -e '$1=rand($1)'; done
0.193584177
0.326966857
0.238298252
```

However, this is overkill. For multi-line records, we can just us the record number (line number) and add different constants to the three fields:

```
$ rcalc -e \
    '$1=rand(recno);$2=rand(recno+.3);$3=rand(recno+.7)'
0.82532675 0.540973626 0.472099463
```