RayLab

COLLABORATORS			
	TITLE : RayLab		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		July 31, 2024	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 RayLab **1**

1.1 main . . . 1
1.2 sct1.0 . . . 4
1.3 sct1.1 . . . 4
1.4 sct1.2 . . . 5
1.5 sct2.0 . . . 5
1.6 sct2.1 . . . 6
1.7 sct2.2 . . . 6
1.8 sct2.3 . . . 6
1.9 sct2.4 . . . 7
1.10 sct2.5 . . . 7
1.11 sct2.6 . . . 7
1.12 sct3.0 . . . 8
1.13 sct3.1 . . . 8
1.14 sct3.2 . . . 9
1.15 sct4.0 . . . 10
1.16 sct4.1 . . . 13
1.17 sct4.2 . . . 13
1.18 sct4.2.1 . . . 14
1.19 sct4.2.2 . . . 14
1.20 sct4.2.3 . . . 14
1.21 sct4.2.4 . . . 15
1.22 sct4.2.5 . . . 15
1.23 sct4.2.6 . . . 15
1.24 sct4.2.7 . . . 16
1.25 sct4.2.8 . . . 16
1.26 sct4.2.9 . . . 17
1.27 sct4.3 . . . 17
1.28 sct4.3.1 . . . 18
1.29 sct4.3.1.1 . . . 18

1.30 sct4.3.1.2 . . . 18
1.31 sct4.3.1.3 . . . 19
1.32 sct4.3.1.4 . . . 19
1.33 sct4.3.2 . . . 19
1.34 sct4.4 . . . 20
1.35 sct4.4.1 . . . 21
1.36 sct4.4.2 . . . 21
1.37 sct4.4.3 . . . 21
1.38 sct4.4.3.1 . . . 22
1.39 sct4.4.3.2 . . . 22
1.40 sct4.4.3.3 . . . 23
1.41 sct4.4.3.4 . . . 23
1.42 sct4.4.3.5 . . . 23
1.43 sct4.4.3.6 . . . 23
1.44 sct4.4.3.7 . . . 23
1.45 sct4.4.3.8 . . . 24
1.46 sct4.4.3.9 . . . 24
1.47 sct4.4.3.10 . . . 24
1.48 sct4.4.3.11 . . . 24
1.49 sct4.4.3.12 . . . 25
1.50 sct4.4.3.13 . . . 25
1.51 sct4.4.4 . . . 25
1.52 sct4.4.5 . . . 25
1.53 sct4.4.6 . . . 26
1.54 sct4.4.7 . . . 26
1.55 sct4.4.8 . . . 26
1.56 sct4.4.9 . . . 26
1.57 sct4.4.10 . . . 27
1.58 sct4.4.11 . . . 27
1.59 sct4.4.11.1 . . . 28
1.60 sct4.4.11.2 . . . 28
1.61 sct4.4.11.3 . . . 28
1.62 sct4.4.11.4 . . . 28
1.63 sct4.4.11.4.1 . . . 29
1.64 sct4.4.11.4.2 . . . 29
1.65 sct4.4.11.4.3 . . . 29
1.66 sct4.4.11.5 . . . 29
1.67 sct4.4.11.6 . . . 30
1.68 sct4.4.12 . . . 30

1.69 sct4.4.13 . . . 30
1.70 sct4.5 . . . 30
1.71 sct4.6 . . . 31
1.72 sct4.6.1 . . . 32
1.73 sct4.6.2 . . . 32
1.74 sct4.6.3 . . . 33
1.75 sct4.6.4 . . . 33
1.76 sct4.6.5 . . . 33
1.77 sct4.6.6 . . . 33
1.78 sct4.7 . . . 34
1.79 sct4.8 . . . 34
1.80 sct4.8.1 . . . 35
1.81 sct4.8.2 . . . 35
1.82 sct4.9 . . . 35
1.83 sct4.9.1 . . . 36
1.84 sct4.10 . . . 36
1.85 sct4.10.1 . . . 36
1.86 sct4.10.2 . . . 37
1.87 sct4.10.3 . . . 37
1.88 sct4.10.3.1 . . . 37
1.89 sct4.10.3.2 . . . 37
1.90 sct4.10.3.3 . . . 37
1.91 sct4.11 . . . 38
1.92 sct4.11.1 . . . 38
1.93 sct4.11.2 . . . 38
1.94 sct4.11.3 . . . 38
1.95 sct4.12 . . . 39
1.96 sct4.12.1 . . . 39
1.97 sct4.12.2 . . . 40
1.98 sct4.12.3 . . . 40
1.99 sct4.12.4 . . . 40
1.100 sct4.12.5 . . . 40
1.101 sct4.12.6 . . . 41
1.102 sct4.12.7 . . . 41
1.103 sct4.12.8 . . . 42
1.104 sct4.12.9 . . . 42
1.105 sct4.12.10 . . . 42
1.106 sct4.12.11 . . . 43
1.107 sct4.12.12 . . . 43

1.108 sct4.12.13 43
1.109 sct4.13 44
1.110 sct5.0 44
1.111 sct5.1 44
1.112 sct5.2 45
1.113 sct5.3 46
1.114 sct5.4 47
1.115 sct5.5 47
1.116 sct5.6 48

Chapter 1

RayLab

1.1 main

R a y L a b

v1.1

Users Documentation

TABLE OF CONTENTS
==========

1.0 INTRODUCTION

1.1 Background

1.2 What does RayLab do?

2.0 FEATURES

2.1 Primitives

2.2 Textures

2.3 Transforms

2.4 Constructive Solid Geometry (CSG)

2.5 Anti-aliasing

2.6 Display

3.0 HOW TO CREATE 3D IMAGES WITH RAYLAB

3.1 An example scene

3.2 Invoking RayLab

4.0 LANGUAGE REFERENCE

4.1 General

4.2 Primitives
- 4.2.1 Sphere
- 4.2.2 Ellipsoid
- 4.2.3 Plane
- 4.2.4 Box
- 4.2.5 Triangle
- 4.2.6 Trianglelist
- 4.2.7 Disc
- 4.2.8 Cylinder
- 4.2.9 Cone

4.3 Constructive Solid Geometry (CSG)
- 4.3.1 CSG Operators
  - 4.3.1.1 And
  - 4.3.1.2 Or
  - 4.3.1.3 Minus
  - 4.3.1.4 Plus
- 4.3.2 Inverting an object

4.4 Textures
- 4.4.1 Color
- 4.4.2 Colormap
- 4.4.3 Pattern
  - 4.4.3.1 checker
  - 4.4.3.2 circles
  - 4.4.3.3 rings
  - 4.4.3.4 spots
  - 4.4.3.5 gradient
  - 4.4.3.6 squares
  - 4.4.3.7 marble
  - 4.4.3.8 softmarble
  - 4.4.3.9 blurb
  - 4.4.3.10 mandel
  - 4.4.3.11 wood
  - 4.4.3.12 angular
  - 4.4.3.13 none
- 4.4.4 Reflect
- 4.4.5 Diffuse
- 4.4.6 Ambient
- 4.4.7 Phong
- 4.4.8 Phongsize
- 4.4.9 Filter
- 4.4.10 Ior
- 4.4.11 Image
  - 4.4.11.1 iff
  - 4.4.11.2 tga
  - 4.4.11.3 ppm
  - 4.4.11.4 maptype
    - 4.4.11.4.1 planar
    - 4.4.11.4.2 spherical

4.4.11.4.3 cylindrical
4.4.11.5 tile and notile
4.4.11.6 interpolate and nointerpolate
4.4.12 Turbulence
4.4.13 Default

4.5 Deftexture

4.6 Transform
4.6.1 Scale
4.6.2 Move
4.6.3 Rotate
4.6.4 Whirl
4.6.5 Twist
4.6.6 None

4.7 Deftransform

4.8 Bounding volumes
4.8.1 How to use bounding volumes in RayLab
4.8.2 When to use bounding volumes

4.9 Miscellaneous object modifers
4.9.1 Noshadow and shadow

4.10 Lights
4.10.1 Location
4.10.2 Color
4.10.3 Soft light-sources
4.10.3.1 size
4.10.3.2 softness
4.10.3.3 jitter

4.11 Camera
4.11.1 Location
4.11.2 Viewpoint
4.11.3 Aspect

4.12 Globals
4.12.1 Picwidth
4.12.2 Picheight
4.12.3 Format
4.12.4 Recdepth
4.12.5 Display
4.12.6 Antialiasrec
4.12.7 Antialiasthreshold
4.12.8 Antialiastjitter
4.12.9 Quickscan
4.12.10 Trace
4.12.11 Backgroundcolor
4.12.12 Fogdistance
4.12.13 Fogcolor

4.13 Comments

5.0 MISCELLANEOUS

5.1 How to contact the author

5.2 Legal stuff

5.3 Past, present and the future

5.4 Acknowledgements

5.5 Compability

5.6 Comments by the author

1.2 sct1.0

1. INTRODUCTION
==========

Background

What does RayLab do?

1.3 sct1.1

1.1 Background

My intention with RayLab was, first of all, to create a decent raytracer based on my newfound (rather limited) knowledge in linear algebra, as I took a course in that particular subject at the university. I also wanted to improve my skills in C-programming (this is my third 'real' C program). As it turned out it was not hard at all to create a working program, and after about a week of programming I could already produce good looking pictures with shades, reflections, highlights, texture-patterns and several different shape primitives. A working (but primitive) description language had also been introduced successfully at that time.

After being convinced that it was possible to create a usable raytracer, I wanted to use it as a base for introducing features that I miss in other raytracers, not having to think about rendering times; with fast computers at the university I can sacrifice time to get a good looking image. In short, RayLab was, for me, a personal 'hack'.

As time went on, more and more people contacted me from all over the world, telling me about how they liked RayLab and feeding me with ideas about how to improve RayLab. Jean-Baptiste Novoit in France helped me out with documentation and compilation. Brian Jones in Canada chose RayLab as his favourite raytracer and helped me a lot with beta-testing. Amith Yamsani at Cornell University used RayLab as a base for a project in parallell computing. Just to name a few. Also, the amount of visitors to the RayLab home page on the WWW has increased drastically since the release of RayLab 1.0 rev 1. All of this moral support has triggered me to really try to make something great out of RayLab, which means, among other things, that

it will be faster and easier to use (things that came in second hand in my old plan for RayLab).

One thing still remains though, and that is that RayLab is totally free of cost! I have always felt that free software is good software, and I hope that I will never have to get paid for programming RayLab; it is an act of creation, not greed.

1.4 sct1.2

1.2 What does RayLab do?

RayLab is capable of producing realistic three-dimensional images using a technique called 'ray-tracing'. What this basically means is that you place 3D objects in a space (room, world, scene or whatever you like to call it). Then you add some light-sources and a camera. What the raytracer does is to mathematically calculate what the picture would look like, if it was to be taken by a real camera.

To get technical, this is done by dividing the picture into small elements (pixels) and for each element, a ray (or beam) is cast, from the camera out into the three-dimensional space. If this ray 'hits' an object, some steps are taken:

1) The color of the object is checked.

2) The light from different light-sources are checked, and also another ray (from the point on the object to the light-source(s)) is cast to see if any other objects block the light. This way shadows are produced.

3) If the object is reflective, a new ray will be cast to see if any other objects are visible thorugh reflection of the first object.

4) If the object is transparent, a new ray will be produced which continues on the other side of the surface of the object.

The information from all these calculations is used to set the color of the pixel in the picture. This procedure is repeated for every ray, including all 'camera-rays' and those produced by reflection and transparency. As you would probably guess, this requires a great deal of computation. On a normal personal computer calcualtion times of several hours are not too uncommon (a processor with good floating point performance is strongly recommended).

1.5 sct2.0

2. FEATURES
==========

RayLab is a program that will probably be updated with new features every now and then, as I like experimenting with new effects. Some of the major

features of the current version of RayLab are mentioned in this section.

Primitives

Textures

Transforms

Constructive Solid Geometry (CSG)

Anti-aliasing

Display

1.6 sct2.1

2.1 Primitives

Primitives are the basic shapes that make the building blocks of objects. Some raytracers are specialized at only handling objects built up from triangles (just as a 2D object can be built up from lines, a 3D object can be described by triangles, but only roughly), but RayLab handles several different shapes. Those are (at the moment): sphere, ellipsoid, plane (infinitly long and wide), box, triangle, disc, cylinder and cone.

1.7 sct2.2

2.2 Textures

The shape of an object is not enough to describe it. The surface of an object is very significant for its appearance. For instance, a polished wooden table looks completely different than a transparent glass-table. In RayLab a surface texture means a set of properties that describe the looks and feels of a surface. The different properties are: color (or set of color-transitions), pattern, diffuse reflection (color intensity proportional to surrounding light-sources), highlight intensity and concentration, ambient light, reflection, transparency with refraction and finally turbulence (for distorting patterns). It is also possible to map an image onto an object, which further extends the possibilities to customize the appearance of an object.

1.8 sct2.3

2.3 Transforms

In general, primitives and textures in RayLab have a basic orientaion in space. Therefore you may need to transform a shape and/or texture into what you need for your specific needs. For instance, the box primitive is always aligned along the x-y-z-axes, which is not always convenient when

you want to construct complex scenes. This is where the transformation facilities of RayLab come in handy. All primitives can be transformed in any of the following ways (or a complex combination of these): scaling, movement (displacement) and rotation. This is also valid for all textures. Be warned though: transformation can consume much computation power, especially rotation of primitives, so do not use it unless you need it (or have a monster computer). There are some special transformations that can be done to textures, but not to objects, namely whirl and twist.

1.9 sct2.4

2.4 Constructive Solid Geometry (CSG)

When neither the basic primitives nor transformation of these are enough to produce the shapes that you want, constructive solid geometry (CSG) can do magic. The idea is to combine two or more primitives with logical operations to produce a new shape. The operations that are needed (and supported by RayLab) are: AND, OR, MINUS and PLUS (the latter is not really necessary though). With AND, the intersection of two shapes is produced. With OR, you get a union of two objects. With MINUS, you can cut out a piece of an object with another object. The PLUS operator acts like the OR operator, but it is faster and not suitable for transparent objects nor objects that are used in other "CSG's". The exact workins of each operator is described in section 4.3.1. Note that throughout this documentation, 'CSG' written by itself often refers to a combination of objects with the constructive solid geometry facilities of RayLab (i.e. a CSG section).

1.10 sct2.5

2.5 Anti-aliasing

When you render a picture with a computer program such as RayLab, chances are that the picture will look very 'jaggy'. This is because the pixels on the computer screen are square, and often visible to the eye. With anti-aliasing this rough look can be eliminated. The idea is to use several rays per pixel, and calculate the average color found by these rays. This method gives a very smooth and realistic appearance, but it also consumes more computation power. Anti-aliasing should be used for final output pictures, as it is quite unnecessary during the process of creating a scene.

1.11 sct2.6

2.6 Display

One useful feature of RayLab is that it can display the output picture to your screen while rendering it. This is good for displaying previews. Currently only the Amiga and the OS/2 versions of RayLab have support for graphical displays, but I hope for more persons to develop display routines

for different platforms, or to improve avalable routines.

1.12 sct3.0

```
                3.  HOW TO CREATE 3D IMAGES WITH RAYLAB
                ======================================
```

To have RayLab create a picture for you, you first have to describe the picture to it. This is done by setting up a 'scene', which can be created in any text-editor or word-processor that can save raw ascii files (Amiga: ed, bed, ced, PC: edit, Unix: vi, emacs, nedit... etc.). The description is made in a special language, which is described in detail in chapter 4. In the scene you will need to have at least one object and one light-source. You will probably want to set up a camera as well.

An example scene

Invoking RayLab

1.13 sct3.1

3.1 An example scene

To get to know a program or programming language, it is always wise to look at demonstration examples. So to get things started, here is an example of a complete scene description (detailed explanations will follow). This scene can also be found in the scenes directory of the RayLab distribution, named 'demo1.rl':

```
Globals:
        picwidth 320            # Image dimensions: 320x240
        picheight 240
        backgroundcolor 0 0 0   # Red=0, green=0, blue=0 => black
:end

Sphere:
        centre  0 0 2           # Place the sphere in x=0, y=0, z=2
        radius  2               # ...and it will have a radius of 2 units
:end

Light:
        location 3 -8 4         # A light-source at x=3, y=-8, z=4
        color    1 1 1          # Red=1.0, green=1.0, blue=1.0 => white
:end

Camera:
        location -1 -9 2        # Place the camera in x=-1, y=-9, z=2
        viewpoint 0 0 2         # Look at x=0, y=0, z=2
:end
```

On the first line, the keyword 'Globals:' is found. This means that the next section will contain some information that is global to the whole scene. This section is ended by the keyword ':end' on line five. On the second line the picture width is declared by 'picwidth 320'. Next follows a comment, which is started by a '#' and ended by the end of the line. On the next line the picture height is declared. The fourth line says 'backgroundcolor 0 0 0', which means that the background color should be black. All colors in RayLab are described by their red, green and blue components, in that order, each component ranging from 0.0 (min) to 1.0 (max).

The next section describes a primitive. This primitive is a sphere with its centre in (0,0,2), and the radius 2. All points and vectors in RayLab are described by their x, y and z components, and are aligned to a right-hand system. This means that the x-axis points right, the y-axis away and the z-axis points upwards. The sphere is not given a texture, so it will use the default texture. The default texture is, if none else given, a non-reflective dull red surface.

The third section describes a light-source, which is located at (3,-8,4) and has the color white (all three components are set to their maximum value 1.0). The light-source casts its light all the way to the infinity in all directions.

Last, but not least, a camera is set up. The parameters location and viewpoint sort of speak for themselves.

To finally render (compute) the picture which is described by this scene, you just have to call RayLab from your shell (cli, dos, or whatever). To render the example above, type:

```
raylab  -iscenes/demo1.rl -opics/demo1.iff
```

That will produce an iff picture named 'demo1.iff' in the directory 'pics'.

If you have installed RayLab properly, i.e. if this guide is located in the raylab/docs directory and you have renamed the version of RayLab that you want to use to 'raylab' (e.g. rename raylab881 to raylab), then you can see RayLab in action. Just click here to see the scene 'demo1.rl' rendered to your screen!

1.14 sct3.2

3.2 Invoking RayLab

RayLab is started from shell. When invoking RayLab you may pass several control parameters. The general syntax is:

```
raylab <parameters>
```

Parameters are, in general, given as a flag with an argument attached to

it. For instance, you specify the width of the output image to 320 pixels with -w320. Note that no space is allowed between the flag (in this case -w) and the argument (320).

The currently supported parameters are (the order in which they are entered on the command line is of no significance):

Parameter:	Specifies:
-i<input scene-description>	The input scene (required)
-o<output image>	The output image file (required *)
-f<image format>	File format, which may be: iff (24 bit compressed IFF (ILBM)) tga (24 bit uncompressed Targa) ppm (24 bit PPM) none (No output file at all)
-w<width>	Width of the output image (columns)
-h<height>	Height of the output image (rows)
-d<display-type>	Display type (platform-specific)
-r<trace recursion-depth>	Maximum recursion depth for trace
-ar<anti-alias recursion-depth>	Anti alias recursion depth
-at<anti-alias threshold>	Anti alias threshold
-aj<anti-alias jitter>	Anti alias jitter
-q	Perform quick-scan (preview)
-t	Perform full-featured trace (default)
-l	List all available display-modes, then exit RayLab (do nothing)

(*) An output file has to be specified, unless the image format is set to 'none'.

For more information about what each parameter means, and what their default values are, please consult section 4.12 (globals).

Here is an example for rendering a scene described in 'myscenes/test.rl' to an iff image called 'pics/test.iff':

```
raylab -imyscenes/test.rl -opics/test.iff -fiff -d1 -w640 -h480 -ar2
```

This means: input file = 'myscenes/test.rl', output file = 'pics/test.iff', format = iff, display type = 1, image width = 640, image-height = 480, and anti-aliasing recursion depth = 2.

To interrupt RayLab while it is generating an image, press ctrl+c. This should work on most systems. If the output format is iff or tga, the output image will be correctly truncated to contain only the lines that were generated before the break. If you for instance start rendering a 320x240 image, and press ctrl+c when RayLab has come to line 100, the output image will be truncated to 320x99. So far, this feature is not implemented for ppm output.

1.15 sct4.0

4. LANGUAGE REFERENCE

- General
- Primitives
  - Sphere
  - Ellipsoid
  - Plane
  - Box
  - Triangle
  - Trianglelist
  - Disc
  - Cylinder
  - Cone
- Constructive Solid Geometry (CSG)
  - CSG Operators
    - And
    - Or
    - Minus
    - Plus
  - Inverting an object
- Textures
  - Color
  - Colormap
  - Pattern
    - checker
    - circles
    - rings
    - spots
    - gradient
    - squares
    - marble
    - softmarble
    - blurb
    - mandel
    - wood
    - angular
    - none
  - Reflect
  - Diffuse
  - Ambient
  - Phong
  - Phongsize
  - Filter
  - Ior
  - Image
    - iff
    - tga
    - ppm
    - maptype
      - planar
      - spherical

```
            cylindrical
        tile and notile
        interpolate and nointerpolate
    Turbulence
    Default

Deftexture

Transform
    Scale
    Move
    Rotate
    Whirl
    Twist
    None

Deftransform

Bounding volumes
    How to use bounding volumes in RayLab
    When to use bounding volumes

Miscellaneous object modifers
    Noshadow and shadow

Lights
    Location
    Color
    Soft light-sources
        size
        softness
        jitter

Camera
    Location
    Viewpoint
    Aspect

Globals
    Picwidth
    Picheight
    Format
    Recdepth
    Display
    Antialiasrec
    Antialiasthreshold
    Antialiastjitter
    Quickscan
    Trace
    Backgroundcolor
    Fogdistance
    Fogcolor

Comments
```

1.16 sct4.1

4.1 General

The description language that RayLab uses is very simple, which may be a benefit or a drawback, depending on your personal needs. I have tried to make the interpreter as flexible as possible though to leave the layout to the user. This has been accomplished by the following means:

- o Keywords can be entered in (almost) any order (*).
- o RayLab is totally case insensitive (SpHeRe: is the same thing as SPHERE:).
- o RayLab does not differ between spaces, tabs and line-feeds (except when you use comments), so indents, line-spaces etc. does not make a difference to RayLab; e.g. you can write several keywords on one line.
- o In many cases you can leave some keywords out. This will result in that RayLab will use default settings for those keywords that are not specified.

(*) The shape specific keywords in object declarations must be entered before general object keywords.

A description is built up from several "sections", each section beginning with a keyword with a terminating colon (e.g. Box:), and ending with :end. Sections may also have sub-sections. For instance a primitive may have a texture specification, which is also a section.

RayLab uses a right-hand system for the space-coordinates, which means: the x-axis points right, the y-axis points away (into the screen) and the z-axis points up. All coordinates and vectors are given by three decimal numbers. For instance, -2 0.5 1.5 means x=-2, y=0.5 and z=1.5. In RayLab, much is based on unit-sizes, which means that you should keep your coordinates and sizes somewhere around the 1.0 - 10.0 range for comfort (but of course, you may scale textures etc to fit a sphere with the radius 1000, if you prefer those dimensions).

Colors are described with three decimal numbers, like coordinates, but the numbers represent the red, green and blue components of the color. Each number must be in the range 0.0 - 1.0, where 0.0 means 0%, and 1.0 means 100%. So, the statement 'color 0.0 1.0 0.5' means 0% red, 100% green and 50% blue, which makes a light green color with a blue touch.

1.17 sct4.2

4.2 Primitives

As mentioned before, the so called primitvies are the actual shapes that you will use to build your scene. Each primitive is declared by creating a section that describes the primitive. The section must begin with the data that is specific to the primitive that is declared (e.g. radius, corners, centre etc.), unless you choose to leave that information out to use the default values. After that, general descriptions may be entered, such as texture, transformation, bounding volume etc.

```
Primitives that are supported by RayLab are: sphere, ellipsoid, plane,
box, triangle, trianglelist, disc, cylinder and cone.
```

1.18 sct4.2.1

```
4.2.1  Sphere

The sphere primitive is a classic round shape, which is defined by its
location in space (centre) and its radius.  Example:

Sphere:
        centre 0 0 3
        radius 2
:end

The default sphere has the radius 1.0 and has its centre at (0,0,0).
```

1.19 sct4.2.2

```
4.2.2  Ellipsoid

The ellipsoid is a more or less redundant shape, as it can be produced from
a scaled sphere, but it was the first primitive introduced in RayLab and as
such is kept due to nostalgic reasons.  Unlike the sphere, the ellipsoid
needs three values to describe the radius, one for each axis.  Example:

Ellipsoid:
        centre 0 0 3
        radius 2 1.5 4
:end

The default ellipsoid has the radius (1,1,1) and is located at (0,0,0).
```

1.20 sct4.2.3

```
4.2.3  Plane

A plane is an infinitely long and wide, totally flat surface.  It is
defined by its surface normal, which is orthogonal to the surface, and its
offset along this normal.  This example shows a plane which could be
described as all points (x,y,z) which has x=-1 (regardless of y and z):

Plane:
        normal 1 0 0
        offset -1
:end

The default plane has the surface normal (0,0,1) and offset 0.0, i.e. it
stretches along the x and y axes.
```

1.21 sct4.2.4

4.2.4 Box

The box is simply a polyhedron with six sides, like a dice. It is described by two points in space (its corners). For instance, a cube with its eight courners:

(0,0,0), (2,0,0), (2,2,0), (0,2,0), (0,0,2), (2,0,2), (2,2,2), (0,2,2)

would look like this described to RayLab:

```
Box:
        corners 0 0 0
                2 2 2
:end
```

Thus each side of the box is always parallel to two of the axes. To create a box with any alignment, you can simply transform it.

The default box is a unit cube with the corners (0,0,0), (1,1,1).

1.22 sct4.2.5

4.2.5 Triangle

A triangle is defined by its three corners in space. Example:

```
Triangle:
        corners 0 0 -1
                3 -2 4
                -5 -1 3
:end
```

The default triangle has the corners (0,0,0), (1,1,1), (-1,1,1).

See also: Trianglelist

1.23 sct4.2.6

4.2.6 Trianglelist

A trianglelist is a list of several triangles. The advantages of a trianglelist over several separate triangles are obvious for large triangle-objects. Trianglelists consume less memory, as it only needs one texture and one transformation for all triangles, as compared to one for each triangle. Trianglelists occupy less space in the description-file, and trianglelists are faster. The trianglelist is used just like the triangle, with the one difference that you may enter more than one corner-description. Example:

```
Trianglelist:
        corners 0 0 0     1 0 0     1 0 1
        corners 0 0 0     0 1 1     1 0 1
        ...
        ...
:end
```

The default trianglelist has one triangle with the corners (0,0,0), (1,1,1), (-1,1,1).

See also: Triangle

1.24 sct4.2.7

4.2.7 Disc

The disc is an infinitely thin surface, just like the plane. The difference between the plane and the disc, is that the disc has a radius, whereas the plane is infinitely long and wide. Thus the disc looks like a disc (he-he). The definition of a disc requires three parameters: centre, normal and radius. Example:

```
Disc:
        centre 2 -3 2
        normal 1 -2 1
        radius 1.5
:end
```

The default disc has the following settings:

```
        centre:  (0,0,0)
        normal:  (0,0,1)
        radius:  1
```

1.25 sct4.2.8

4.2.8 Cylinder

The cylinder is defined by its radius and its two end positions in space (for compability with RayLab 1.0, you can also specify it with its height, and thus leaving the end positions out). The ends are specified with the keywords start and end. The cylinder is "closed" by two discs, one at each end (all in the spirit of solid geometry). This example demonstrates a cylinder that lies along the x-axis, with its ends in (-3,0,0) and (3,0,0):

```
Cylinder:
        radius 1.4
        start -3 0 0
        end 3 0 0
:end
```

The default cylinder has a radius of 1.0, start (0,0,0), and end (0,0,1).

1.26 sct4.2.9

```
4.2.9  Cone

The cone is similar to the cylinder, with the one difference that you
specify one radius for each end of the cone.  Thus the cone needs its two
end positions in space and its two radii.  The ends are specified with the
keywords start and end.  The two radii are specified with the keyword
radius, followed by start <start-radius> end <end-radius>.  The cone, like
the cylinder, is "closed" by two discs, one at each end.  Here is an
example of a cone:

Cone:
        start 0 0 0
        end 3 -2 1
        radius start 1.5 end 0.5
:end

The default cone starts in (0,0,0), ends in (0,0,1), has the start-radius
1.0 and end-radius 0.0.
```

1.27 sct4.3

```
4.3  Constructive Solid Geometry (CSG)

A very powerful feature of RayLab is the ability to combine several objects
into a new shape.  You may perform logical operations with two or more
shapes.  To declare such a combination to RayLab, use the following syntax:

Csg:
        <shape>:                # First shape in CSG section
                ...
                ...
        :end

        <operator>

        <shape>:                # Second shape in CSG section
                ...
                ...
        :end

        <operator>
        ...
        ...
        <operator>

        <shape>:                # N:th shape in CSG section
                ...
                ...
        :end
:end

In a CSG section you must specify at least two shapes, but there is no
```

limit to how many you can have in each section. The operator may be one of the following: AND, OR, MINUS or PLUS. In a single CSG section, all operators must be of the same kind (you may not mix different operators). One exception to this rule, is that AND and MINUS are treated equally. Actually, MINUS is an AND operand, but the second object is inverted, which means that the inside becomes the outside, and vice versa.

The shapes in a CSG section are described with normal object declarations, with or without object modifiers, such as texture and transformation. Texture declarations must be defined within the individual shape declarations (or with a deftexture statement before the CSG section), but transformations can be placed at the end of a CSG section, thus transforming all objects contained in the section.

The shapes that may be part of a CSG section must be such that they have a clear inside and outside. Currently they are: plane, sphere, ellipsoid, box, cylinder, cone, and even another CSG section. The inside and outside of these objects are very easy to determine (the "outside" of a plane is the "upside", which is where the surface normal points, and the "inside" is the opposite side). Shapes that may not be part of a CSG section are the triangle, the trianglelist and the disc (they do not have an inside nor an outside).

See also: Inverting an object

1.28 sct4.3.1

4.3.1 CSG Operators

As mentioned before, the logical operators that RayLab supports are AND, OR, MINUS and PLUS. Here you get an explanation of each operator. Note that although I have chosen to describe the situation where only two objects are used in a CSG-combination, several objects can be combined in the same manner.

1.29 sct4.3.1.1

4.3.1.1 And

If you combine two objects with the and operator, a new body is produced which consists of all points in space that are inside both the first and the second object. Points that are inside one of the objects, but not the other, are not part of the new body.

This figure illustrates how AND works.

1.30 sct4.3.1.2

4.3.1.2 Or

If you combine two objects with the or operator, a new body is produced which consists of all points in space that are inside either the first or the second object, or inside both objects. Surfaces from either of the two objects that are inside the new body are removed, which looks better for transparent objects, and it works better if you want to use the new body in another CSG section.

This figure illustrates how OR works.

1.31 sct4.3.1.3

4.3.1.3 Minus

With the minus operator you can cut out a piece of an object with another object. The first object is the object that is being 'cut' by the second object. If more than two objects are used in a minus-combination, the first object is the'original', and all the following objects (the second, the third etc...) are 'cutting' objects.

This figure illustrates how MINUS works.

1.32 sct4.3.1.4

4.3.1.4 Plus

The plus-operator acts similarly to the or-operator. To combine several objects in a plus-CSG is an easy way to group several objects into a single object, and it is faster than to use the or-operator. Be warned though: if you use the plus-operator, the inner surfaces will not be removed, thus making transparent objects look 'funny', and if you use a CSG with plus-operators with another CSG, you may not get the results that you expect.

This figure illustrates how PLUS works.

1.33 sct4.3.2

4.3.2 Inverting an object

By adding the keyword invert to an object declaration, the inside of the object becomes the outside, and vice versa. If you, for instance, invert an object that is part of a CSG, where the operator is AND, you achieve the same effect as with MINUS, where the inverted object is the removal-object. The invert keyword also affects transparency, if the surface refracts light, as 'going into the object' is replaced by 'going out of the object', and vice versa. If you would like to produce air-bubbles in water, for instance, you could use inverted transparent spheres as bubbles.

1.34 sct4.4

```
4.4  Textures

RayLab provides many facilities to make the surface texture of an object
realistic.  To add or change a texture of a primitive, add a texture-
section to the primitive declaration.  Example:

Plane:
        normal 0 0 1
        offset -2
        texture:
                color 0.5 0.4 0.8
                reflect 0.5 0.4 0.5
                phong 0.6
        :end
:end

In the following sections all supported texture keywords are described:

            Color
            Colormap
            Pattern
                checker
                circles
                rings
                spots
                gradient
                squares
                marble
                softmarble
                blurb
                mandel
                wood
                angular
                none
            Reflect
            Diffuse
            Ambient
            Phong
            Phongsize
            Filter
            Ior
            Image
                iff
                tga
                ppm
                maptype
                    planar
                    spherical
                    cylindrical
                tile and notile
                interpolate and nointerpolate
            Turbulence
            Default
```

1.35 sct4.4.1

4.4.1 Color

All primitives have their own color, which can be specified by the keyword color. Following the keyword there should be three decimal values specifying the red, green and blue components of the color (in that order). The values can range from 0.0 (0%) to 1.0 (100%).

The default color is 1.0 0.3 0.0, which is 100% red + 30% green.

1.36 sct4.4.2

4.4.2 Colormap

When you want to give an object surface a pattern (e.g. the rings of a piece of wood), it is not satisfying to only have one color for the object. Normally you want to have a set of colors, and smooth transitions between those colors. This is acomplished with a colormap. A colormap sort of creates a spectrum with different colors, just like the sky can have nice transitions between red and blue and maybe some other colors when the sun sets. A colormap can consist of a maximum of ten entries. Each entry is given a color, and a place in the map, which is a value between 0.0 and 1.0. When you declare a colormap, you also have to give the amount of entries with an integer value following the keyword colormap. Here is an example of a colormap with the colors of the rainbow:

```
    colormap 7                  # This colormap contains seven entries
        0.0   0.4 0.0 0.7       # The first entry (at 0.0) has the color
        0.3   0.0 0.0 0.9       #  0.4 0 0.7
        0.4   0.0 0.4 1.0
        0.6   0.0 1.0 0.0
        0.75  1.0 1.0 0.0
        0.9   1.0 0.0 0.0
        1.0   0.4 0.0 0.0       # The last entry (at 1.0) has the color
                                #  0.4 0 0
```

Note: All entries must be entered in ascending order with 0.0 first and 1.0 last.

Actually, when a color is specified with the color keyword, a colormap with two entries is created; the first entry is 0.0, with the color which is specified after the color keyword, and the second entry is 1.0, with the color black. This is done so that a pattern will be visible even if only one color is specified. This means that the default colormap has two entries: the first is red and the second is black.

1.37 sct4.4.3

4.4.3 Pattern

Patterns can significantly improve the appearance of an object. RayLab

gives you a variety of patterns to chose from, and more are sure to come.
A pattern will assign a value to each point of the object. This value
ranges from 0.0 to 1.0 and is used as index to the colormap of the object
(see section 4.4.2). What follows is a description of the currently avail-
able patterns in RayLab. Experiment with them on different shapes to get
to know how they work!

To sepcify a patter, use the keyword pattern, followed by one of the
following pattern identifiers:

```
checker
circles
rings
spots
gradient
squares
marble
softmarble
blurb
mandel
wood
angular
none
```

1.38 sct4.4.3.1

4.4.3.1 checker

Now this is a pattern we all want! A raytracer is not worth the name
without this pattern. The checker pattern is simply an infinite amount of
1x1x1 cubes in 3D space. If applied to a plane it will look just like a
chess-board, but if you apply it to a sphere it can look rather strange
(try it, and you will see exactly how it works!). Each cube can have one
of two colors, and the color is chosen from the colormap of the object.
The first color is picked from the colormap with index 0.0, and the second
will be picked from index 1.0. Here is an example of a blue and green
checkered texture:

```
texture:
    pattern checker
    colormap 2
            0  0 0 1        # index = 0.0, color = 0.0 0.0 1.0
            1  0 1 0        # index = 1.0, color = 0.0 1.0 0.0
:end
```

1.39 sct4.4.3.2

4.4.3.2 circles

Circles are like onion-rings. The colormap index is simply proportional to
the distance from (0,0,0). Within one unit-distance from (0,0,0) the
index-value changes from 0.0 to 1.0, then it goes straight back to 0.0, and

increaces to 1.0 at two units, and then it starts over again...

1.40 sct4.4.3.3

4.4.3.3 rings

The rings pattern works like circles, but it only depends on the x-y coordinates. This means that it extends along the z-axis.

1.41 sct4.4.3.4

4.4.3.4 spots

The spots pattern will simply produce spots on the surface of an object. The spots are three-dimensional. The centre of the spot will get the color from the colormap at 0.0. The further away from the centre of the spot, the larger the colormap index. At the edge of the spot, and outside of the spot, the colormap index is 1.0.

1.42 sct4.4.3.5

4.4.3.5 gradient

Gradient is probably the simplest pattern possible. The colormap index changes with the z-axis. It increases from 0.0 to 1.0 within one unit along the z-axis, then it starts over from 0.0 again...

1.43 sct4.4.3.6

4.4.3.6 squares

The squares pattern works much like the {"spots" link sct4.4.3.4} pattern (not ↩ like the checker
pattern). At the centre of each square, the colormap index is 0.0, and at the edge the color-map index is 1.0. The index smoothly changes from 0.0 to 1.0 within the square. Just like the spots pattern, the squares are three-dimensional.

1.44 sct4.4.3.7

4.4.3.7 marble

Marble is a very good looking texture to use on stone-like floors, tables etc. If the marble pattern is applied to an object 'as-is', it will only produce an infinite amount of distorted lines (which are parallel to the yz

plane). To make it look good, you would typically have to add some turbulence to the texture, and maybe rotate and/or scale it too.

1.45 sct4.4.3.8

4.4.3.8 softmarble

Softmarble is nearly identical to marble, but it has a smoother look. Again, try experimenting...

1.46 sct4.4.3.9

4.4.3.9 blurb

For a surface texture that is more or less stocastic (such as different kinds of stone), the blurb pattern is very useful. It simply uses a solid noise function to produce a colormap index. Solid noise in RayLab means that two points that are close to each other have values that are close, and two points that are far away from each other have values that may differ greatly. The blurb pattern needs some turbulence to give it life, else it may look a bit 'stiff'.

1.47 sct4.4.3.10

4.4.3.10 mandel

Long before I made RayLab, I made a program that generated three-dimensionally projected fractal-images, such as the mandelbrot set. As it is very easy to implement, I added the mandelbrot fractal as a texture pattern for RayLab too (called mandel). It uses the xy plane as the complex plane (x is the real axis, and y is the imaginary axis). The fractal image looks identical regardless of the position on the z-axis. In the current implementation a maximum of 100 iterations are performed. For you who do not know how a mandelbrot fractal looks (you must have seen it somewhere), try rendering the example scene 'scenes/mandel.rl' (click here to render it to your screen).

1.48 sct4.4.3.11

4.4.3.11 wood

The wood pattern is very much like the rings pattern, but it contains a small variation of the ring-radius, which depends on the position on the circle (the angle from the x-axis) and the position along the z-axis. Try a small rotation around the x or y axes, and perhaps a small turbulence aswell. It is not easy to make wood look realistic, but experiment a bit, and it may turn out lovely.

1.49 sct4.4.3.12

4.4.3.12 angular

The angular pattern uses the angle from the positive x-axis in the xy-plane for the colormap index (it sweeps around the z-axis). Very simple indeed, but useful sometimes.

1.50 sct4.4.3.13

4.4.3.13 none

Specifying a pattern with the identifier none will give the object a monocolored surface. The color will be picked from the colormap with index 0.0.

The default pattern is none.

1.51 sct4.4.4

4.4.4 Reflect

Reflection is one of the most interesting and powerful featurs of the raytracing technique compared to other forms of 3D rendering. In RayLab you can choose how much light an object should reflect (with the keyword reflect). You can also decide what colors will be reflected. Here is an example of a reflection that reflects 70% of all light:

```
    reflect  0.7 0.7 0.7           # red=70%, green=70%, blue=70%
```

The default reflection is no reflection at all (0 0 0).

1.52 sct4.4.5

4.4.5 Diffuse

The diffuse keyword specifies how much of the light from the surrounding light-sources will affect the color intensity of the object. The diffuse intensity is specified by a decimal value ranging from 0.0 to 1.0 (0% to 100%). If a value of zero is given, the object will have the same color intensity on its lightened side as on its shadowed side. Normally, you would want a value between 0.5 and 1.0.

The default diffuse intensity is 0.8 (80%).

1.53 sct4.4.6

4.4.6 Ambient

If you look around in "the real world", you will see that even in dark shadows, objects are visible. This is due to the diffuse reflection from other surrounding objects. In a normal raytracer like RayLab, this is a very difficult and power consuming task to produce. Instead this is simulated by the ambient effect, which will give the object a default light-intensity on every point of the surface. The ambient effect can also be used to create "glowing" objects. However, the ambient of one object will not affect any other objects. The keyword ambient is followed by a decimal value ranging from 0.0 to 1.0 (0% ambient to 100%).

The default ambient is 0.2 (20%).

1.54 sct4.4.7

4.4.7 Phong

The phong keyword specifies how intense highlights from light-sources should be. Phong highlights have the same color as the light-source that it is produced from, and is not proportional to the object color at all (a black object will have the same highlight color and intensity as a green object, if their phong-values are the same).

The default phong value is 0.3 (30%).

1.55 sct4.4.8

4.4.8 Phongsize

With phongsize you can specify the tightness of the phong highlight. The keyword is followed by a positive decimal value. Higher values give tighter spots. Theoretically the phongsize value can range from zero to infinity, but in practice you would typically want a value larger than 1.0 (at least) and smaller than 100.

The default phongsize is 10.0.

1.56 sct4.4.9

4.4.9 Filter

With the keyword filter you tell RayLab how transparent an object should be, and also what colors it should 'let through'. After the keyword you must specify three decimal values, representing the amount of red, green and blue light the surface of the object should filter through. If all three components are set to 1.0, the object would be perfectly transparent.

If all components are set to 0.0, no light will come through the surface of the object.

The default filter is (0.0 0.0 0.0), which means 0% transparency.

1.57 sct4.4.10

4.4.10 Ior

Most transparent objects refract light. This is because the light has different speeds in different media; the thicker the media, the slower the speed of light. To tell RayLab how much an object should refract light, you specify the index of refraction, ior. This index is 1.0 or more, where 1.0 represents air or vaccum. Typical values are: water - 1.3, glass - 1.5, and diamond - 2.5. A phenomenon which may occur when light exits a thick medium, is that it bounces back if the angle between the light-ray and the surface is too small. This is called total internal reflection, and it is handled automatically by RayLab.

The default ior is 1.0, which means no refraction.

1.58 sct4.4.11

4.4.11 Image

If the built in patterns of RayLab are not enough, you may experiment with mapping images onto your objects. You may choose from three different maptypes at the moment: planar, spherical and cylindrical. Three image-formats are supported: IFF (1-8 bit colormapped and 24-bit true color), Targa (24-bit true color) and PPM (24-bit true color, P6-format).

Specifying a texture image is done by adding an image section to your texture section, like this, for example:

```
    texture:
        ...
        image:
                iff textures/myface.iff
                ...
        :end
        ...
    :end
```

The image may be transformed by adding a transform section to the image section. If the texture is transformed, the image too will be transformed.

In the image section, you may enter some additional information about how the image will be treated: maptype, tile and notile, interpolate and nointerpolate.

1.59 sct4.4.11.1

4.4.11.1 iff

If your image is stored as an iff file, you specify its name after the keyword iff. Note that you may not enclose the file-name within brackets, and the file-name may not contain any spaces. File-names are case-sensitive, if your operating system is. RayLab currently handles 24-bit iff files (IFF24), as well as 1-8 bit (2-256 colors) color-mapped images. It does not, however, handle HAM6, HAM8, EHB, 12-, 15- or 16-bit images.

1.60 sct4.4.11.2

4.4.11.2 tga

If your image is stored as a tga file, you specify its name after the keyword tga. The file-name is treated in the same way as for iff files. RayLab currently only handles uncompressed 24-bit tga files (which is the most common format anyway).

1.61 sct4.4.11.3

4.4.11.3 ppm

As I added PPM output support, I figured it might be a good idea to support PPM input as well. As I do not know very much about the PPM format, it may be that RayLab does not handle your pictures. The currently supported format is the P6 format, which holds uncompressed 24-bit binary data.

1.62 sct4.4.11.4

4.4.11.4 maptype

Depending on the shape of the object you want to use your image on, you may want to use different types of mappings. That is, how to place the image in 3D space. As a normal image is two-dimensional, it is not clear how it should be placed in the three-dimensional space that RayLab deals with. With the keyword maptype, you may specify which method you want to use. After the keyword, you enter one of these three keywords: planar, spherical or cylindrical. This part may make you a bit dizzy (at least I got dizzy writing it), so I recommend that you experiment on your own to find out how the different maptypes work.

The default maptype is planar.

1.63 sct4.4.11.4.1

4.4.11.4.1 planar

The most common method is to use planar mapping, which means that you simply 'throw away' one axis. In the case of RayLab, it is the z-axis that is being lost, and the x and y axes are used as the coordinates in the 2D image. Regardless of the original image size, it will always be placed so that the lower left corner lies in (x,y)=(0,0), and the upper right corner lies in (x,y)=(1,1). As the z-axis is not used, the image will be 'stretched' along the z-axis. To make the image fit your object properly, you may need to transform the texture (you can not transform the image individually). By default, the image will be 'tiled' when you use planar mapping, which means that the image is repeated over the entire xy-plane.

1.64 sct4.4.11.4.2

4.4.11.4.2 spherical

The spherical mapping is useful for spheres. It acts radically differently from planar mapping, in that it uses two angles as the x and y coordinates. The simplest way to understand how spherical mapping works, is to think of your image as a flat map of the world and your object (the sphere) being the earth-globe. To get the countries, oceans and polar ices in the right place, you would have to 'wrap' the map around the globe (which is physically impossible, but in theory everything can be done!). That is exactly what RayLab does whith your image and your sphere. As a result of this method, there will only be one copy of your image on your object, and you should also try to use an image which is twice as wide as it is high, if you want to retain the original image proportions. RayLab assumes that the object is located in (0,0,0) in space. Otherwise you will have to transform the texture to make it fit your object.

1.65 sct4.4.11.4.3

4.4.11.4.3 cylindrical

You could say that the cylindrical mapping is a combination of spherical mapping and planar mapping. When you use cylindrical mapping, RayLab assumes that you have a cylinder that stretches along the z-axis. The image will be wrapped one turn around the z-axis, and be drawn one time from z=0.0 to z=1.0. By default the image will repeat at z=1.0 to z=2.0 and so on.

1.66 sct4.4.11.5

4.4.11.5 tile and notile

With the keywords tile and notile you can control whether or not the image

should be repeated outside of its origin (e.g. (x,y)=(0,0) to (x,y)=(1,1) for planar mapping). If you choose notile, the color/colormap/pattern settings will be used outside of the image. The keywords have no effect on spherical mapping.

The default is to tile the image.

1.67 sct4.4.11.6

4.4.11.6 interpolate and nointerpolate

The keyword interpolate tells RayLab to use a simple linear method for interpolating the color-values 'inbetween' the pixels of your image. The method is quite fast, and can produce a much better impression in most cases. The keyword nointerpolate does the opposite.

The default is no interpolation.

1.68 sct4.4.12

4.4.12 Turbulence

The turbulence keyword can be added to any texture, and it will make the pattern or image (if any) of the texture 'distorted'. After the turbulence keyword you must specify a decimal value specifying the turbulence intensity. The value may range from zero to "infinity", where zero means no turbulence. However, 0.0 to 1.0 is a more reasonable range, where 1.0 makes the texture very turbulent. Often values like 0.1-0.4 are enough. Try experimenting with turbulence on different textures; it is really powerfull (ever seen a turbulent mandelbrot set before?). It should be mentioned that turbulence is quite CPU consuming.

The default turbulence intensity is 0 (no turbulence).

1.69 sct4.4.13

4.4.13 Default

The keyword default can be placed in a texture section to initialize the texture with RayLabs hardcoded default settings. The default setting for each component is given in the corresponding section in this document (e.g. the color in section 4.4.1).

1.70 sct4.5

4.5 Deftexture

With a deftexture section, you can specify a default texture that will be applied to all the following objects in the scene description (it does, however, not affect the 'default' keyword described in section 4.4.13). Example:

```
Deftexture:
        reflect  0.6 0.6 0.6
        pattern  spots
        colormap 4
                0.0  1 0 0
                0.7  1 0 0
                0.8  0.2 0.2 0.2
                1.0  0 0 0
:end
```

At the end of the list of all objects that are to have the texture specified by deftexture, you will probably want to reinitialize the hard-coded default texture. Simply write:

```
Deftexture:
        default
:end
```

1.71 sct4.6

4.6 Transform

When the shape of a primitve is too limited for your demands, chances are that the transformation facilities of RayLab can help you. Transformations can change the shape, location and orientation of any primitive or texture. There are currently three transformations available in RayLab that apply to both objects and textures: scale, move and rotate. For textures, you may also use: whirl and twist. A transform section can contain a maximum of ten transformations, which are applied to the object or texture in the order they are entered in the description. A transform section is started with the transform keyword, and ended with :end.

To transform an object, the transform section should be placed somewhere before the end of the object. If the transformation is placed after the texture-declaration (if any), the texture too is transformed, but if it is placed before the texture section, the texture is not transformed. You can also place a transform section inside a texture section, which will only affect the texture. Image-textures can also be transformed (please refer to the section that describes how to use image-textures for more information).

Here is an example of a transform section:

```
    transform:
        rotate 30 0 0          # rotate 30 degrees around the x-axis
        scale  3 0.6 1         # scale 3*x, 0.6*y and 1*z
        move   -3 0 4          # move the object -3*x +4*z from its current
```

```
    :end                        #  location
```

Note that the following example will NOT do the same thing as the above:

```
    transform:
        move   -3 0 4
        rotate 30 0 0
        scale  3 0.6 1
    :end
```

All transformations that can be done to an object, can also be done to a texture or an image, independently. On the other hand, some special transformations that are valid for textures and images, do not work with objects.

Often you may want to change the size or orientation of a pattern. To transform a texture, add a transform section inside the texture section. This example shows how you can make a pattern twice as large as its original size:

```
    texture:
        ...
        transform:
                scale 2 2 2
        :end
    :end
```

See also: None

1.72 sct4.6.1

4.6.1 Scale

Scaling an object means stretching or shrinking it along the x, y and z axes respectively. A scale value of 1.0 means no change. A value larger than 1.0 means enlarging and a value less than 1.0 means shrinking (the scaling value must be larger than 0.0). One value must be given for each direction: x, y, z (i.e. three decimal values must follow the scale keyword). The scaling is always done relative to (0,0,0) in space, so if you scale a primitive that has an origin different from (0,0,0), it will also be moved.

1.73 sct4.6.2

4.6.2 Move

Moving an object means moving it from its current location to a location given by (current location) + (movement). Also here three decimal values must be given after the keyword.

1.74 sct4.6.3

4.6.3 Rotate

After the rotate keyword you must specify three angles (in degrees). They represent the rotation around each of the x, y and z axes, and the rotation is performed in that order (first x, then y and last z). To find out which way is the positive rotation, try this trick: hold up your right hand in front of you, extend your thumb and curl the other fingers. Now, if you place your hand so that the thumb points in the positive direction of the axis that you want to rotate about, the four other fingers will show the positive direction of rotation (believe me, I do this all the time). Remember that RayLab uses a right hand system, so the orientation of the axes is as follows: the x-axis points right, the y-axis points "away" (into the screen), and the z-axis points up. An object or a texture is always rotated around (0,0,0) in space, so if you rotate a primitive that has an origin different from (0,0,0), it will also be moved.

1.75 sct4.6.4

4.6.4 Whirl

The whirl transformation may only be used for textures or images. It creates a whirl with its centre in (0,0,0), which 'rotates' in the positive direction around the z-axis. With the keyword whirl you must specify the radius of the whirl, and its intensity, in that order. If the intensity is negative, the whirl rotates in the negative direction around the z-axis. Here is an example of a 'whirly' texture:

```
texture:
    ...
    transform:
        whirl   2.5  0.3    # radius = 2.5, intensity = 0.3
    :end
:end
```

1.76 sct4.6.5

4.6.5 Twist

The twist transformation may only be used for textures or images. It will twist the texture around the z-axis. After the twist keyword, you must enter a decimal number which indicates the z-height of 'one twist' (a full turn, 360\textdegree{}).

1.77 sct4.6.6

4.6.6 None

The keyword none will clear a whole transform sequence.

The default transformation is no transformation.

1.78 sct4.7

4.7 Deftransform

With a deftransform section, you can specify a default transformation sequence that will be applied to all the following objects and textures in the scene description. Example:

```
Deftransform:
        rotate 30 -45 17
        move   10 0 -5
:end
```

At the end of the list of all objects that are to be transformed with a deftransform, you need to withdraw the deftransform. Simply write:

```
Deftransform:
    none
:end
```

NOTE: I have found little use for the deftransform function. It is better replaced by combining several objects into a CSG section, and then transforming the entire CSG (which is also faster, as only one transformation is needed, instead of one for each object). The deftransform was, however, necessary in RayLab 1.0, as CSG was not implemented then.

1.79 sct4.8

4.8 Bounding volumes

A ray-tracer spends most of its time on finding out whether a ray hits an object or not. The more objects in a scene, the more testing is needed. To reduce the work that is needed for this task, there are several techniques. One of the oldest and simplest is to 'surround' one shape with another shape that is easier to test against (it is called bounding one shape with another one). Before an intersection-test with the 'real' shape is done, an intersection is tested with the bounding volume. If the bounding volume was hit, the bounded shape is tested, otherwise it is not. The bounding shape will not be visible, but it must surround the visible shape fully.

RayLab supports two very optimized shapes: box and sphere. Two special intersection routines have been written for those shapes to be used as bounding volumes, and they are very fast.

How to use bounding volumes in RayLab
When to use bounding volumes

1.80 sct4.8.1

4.8.1 How to use bounding volumes in RayLab

To bound one object (all objects, including CSG's, may be bounded), add a bounding section before the end of the object. A bounding section is started with the keyword bounding, and followed by the bounding shape (box: or sphere:). Example:

```
Cylinder:
        start -3 2 1
        end   3 2 1
        radius 1
        bounding box:
                corners  -3 1 0
                          3 3 2
        :end
:end
```

The bounding shape may not contain anything but the shape-specific information (e.g. it may not have any texture or transformation specification). Bounding volumes are affected by transformations that come after the bounding declaration, but not by transformations that come before. Although not transformed, the bounding volume is modified to make sure that it corresponds to the transformed shape.

1.81 sct4.8.2

4.8.2 When to use bounding volumes

First of all, bounding volumes are only used for speedup reasons. Do not use bounding volumes to clip away parts of an object (try to use the CSG facilities of RayLab for that instead), as it will most likely not give the results that you expect.

Bounding volumes are most efficient under these two conditions:

- When an object is very complex, such as a CSG object.
- When an object does not occupy huge portions of the rendered image; it does not make much sense to bound an object that is hit by almost every ray.

If a shape is easily bounded (approximated) with a box or a sphere, it is possible that you may find speedups even by bounding simple shapes, especially if they are transformed.

1.82 sct4.9

4.9 Miscellaneous object modifiers

Noshadow and shadow

1.83 sct4.9.1

4.9.1 Noshadow and shadow

Sometimes you do not want an object to cast shadows. For instance, if you want to give a shape to a light-source, you would typically surround the light-source with an object, such as a white sphere, but then that object would cover the light-source totally, letting no light through. Adding the keyowrd noshadow to the object declaration will make it cast no shadows. The opposite keyword, shadow, will make an object cast shadows, which is the default. Here is an example:

```
Sphere:
        centre -3 4 7
        radius 0.5
        texture:
                color 1 1 1
        :end
        noshadow                # This sphere will not cast any shadows
:end
```

If you add a noshadow/shadow keyword to a CSG-section, all objects that are part of the CSG will be given the corresponding noshadow/shadow property. If any noshadow/shadow keyword is entered in an individual object of the CSG-section, it will be over-ridden by the keyword for the whole CSG.

1.84 sct4.10

4.10 Lights

Every scene needs at least one light-source. Light-sources in RayLab are s.c. point lights, which means that they cast their light in all directions, and they are infinitely small (you can not see them if you turn your camera to look at them). Light-sources can have different colors, and thereby also different intensities (a grey light-source does not give as much light as a white light-source does). A normal light-source is simply defined by a location and a color.

To find out how to use soft light-sources, click here.

1.85 sct4.10.1

4.10.1 Location

With the keyword location you specify where the light-source should be in space.

The default location is (10,-10,10).

1.86 sct4.10.2

4.10.2 Color

The keyword color specifies what color and intensity a light-source should have.

The default color is (1.0 1.0 1.0), which is 100% white.

1.87 sct4.10.3

4.10.3 Soft light-sources

One problem with the basic ray-tracing technique is that the edges of shadows become very sharp. This is because the light-sources are infinitely small. To produce softer shadows you need a light model where the light-sources have a size. RayLab 1.1 implements a very un-optimized light-model to produce soft shadows. To control soft light-sources, you use the keywords size, softness and jitter. Be warned though: soft light-sources are extremely slow in RayLab 1.1, but I hope that I can use the same syntax for a more optimized model in the future.

1.88 sct4.10.3.1

4.10.3.1 size

The size keyword indicates the 'diameter' of the light-source.

The default size is 0.5.

1.89 sct4.10.3.2

4.10.3.2 softness

With softness, you specify how many rays will be used to approximate the soft shadow. The amount of rays can be calculated as the square of (softness+1), e.g. softness 1 means four rays, softness 2 means nine rays etc.

The default softness is 0, which means no softness at all.

1.90 sct4.10.3.3

4.10.3.3 jitter

To improve the appearance of the shadows, some jittering can be added to the ray-directions. The jittering can range from 0.0 (no jittering) to 1.0 (full jittering).

The default jittering is 0.1.

1.91 sct4.11

4.11 Camera

The camera is indeed a very important part of the scene. The camera does not have a shape, and will not be seen through a mirror or alike. To specify the properties of the camera, the keywords location, viewpoint and aspect can be used.

1.92 sct4.11.1

4.11.1 Location

The camera can be placed anywhere in space. This is given by the keyword location.

The default location is (0,-10,1).

1.93 sct4.11.2

4.11.2 Viewpoint

When the camera has been placed, it also needs to be directed to look at some point in space. In RayLab you only have to tell where to look at with the keyword viewpoint. The direction of the camera will be calculated automatically.

The default viewpoint is (0,0,0).

1.94 sct4.11.3

4.11.3 Aspect

The camera is also defined by its x:y:z aspect, which enables you to set the pixel-aspect of the output picture and the field of view of the camera. The x:y aspect should be the same as that of the output picture. E.g. a 640x480 picture has the aspect 4:3 if the pixels are to be completely square. In general, you should use the same xy aspect as that of the display (regardless of the amount of pixels in each direction). The aspect

ratio does not have to be integer values in RayLab. The z-aspect is the 'depth' of the lens, and it is also proporional to the x and y aspect. With the z aspect you may change the field of view. This figure may be of some help.

Larger values of the z aspect will give more tele-zoom, and smaller values will give wide-angle views.

Here is an example of a camera with extreme tele-zoom:

```
Camera:
        location   0 -100 30    # We have to back off a bit due
        viewpoint  0 0 0        # to the zooming
        aspect     4 3 20
:end
```

The default aspect is 4:3:5, which gives quite a normal angle of view.

1.95 sct4.12

4.12 Globals

In RayLab some interesting and useful parameters can be setup in a section called globals. Most of these parameters may be setup from the command line, which will override parameters written in a globals section.

```
Picwidth
Picheight
Format
Recdepth
Display
Antialiasrec
Antialiasthreshold
Antialiastjitter
Quickscan
Trace
Backgroundcolor
Fogdistance
Fogcolor
```

1.96 sct4.12.1

4.12.1 Picwidth

The keyword picwidth is followed by an integer value telling RayLab how many columns the output picture will have. For a 640x480 image this would be 640.

The default picture width is 200. The command line equivalent is -w<picwidth>.

1.97 sct4.12.2

4.12.2 Picheight

Picheight specifies the amount of lines that the output image will consist of.

The default picture height is 150. The command line equivalent is -h<picheight>.

1.98 sct4.12.3

4.12.3 Format

With the keyword format, you may specify the output file format. The currently supported formats are: iff (24-bit compressed IFF ILBM), tga (24-bit uncompressed Targa), ppm (24-bit PPM) and none (no output image is produced). Example:

```
format ppm              # Use PPM image output
```

The default format is iff. The command line equivalent is -f<format>.

1.99 sct4.12.4

4.12.4 Recdepth

The keyword recdepth is used to specify how many recursions RayLab is allowed to do for each camera-ray. A recursion-depth of 2 means that a ray will only 'bounce' one time through reflection and/or transparency, and a recursion-depth of 1 results in no reflection/transparency at all. Higher values mean more realistic pictures if there are many reflective or transparent surfaces in the scene, but also longer rendering-times. For a fast preview, you could specify a recdepth of 1:

```
recdepth 1
```

The default recursion depth is 10. The command line equivalent is -r<recdepth>.

1.100 sct4.12.5

4.12.5 Display

If you want to view a picture as it is being rendered, add the keyword display to your globals section. Following the keyword you must specify a display type. The display types are machine specific, so please consult the documentation for your specific platform for more information. The display type is an integer value. Example:

```
        display 1
```

Zero (0) means no display, which is the default setting. The command line equivalent is -d<displaytype>.

1.101 sct4.12.6

4.12.6 Antialiasrec

This keyword turns on anti-aliasing, which can greatly improve the appearance of a scene. You must specify the recursion depth of the anti-aliasing, which is done with an integer value placed after the keyword. Example:

```
        antialiasrec 3
```

Zero (0) means no anti-aliasing, which is the default. The command line equivalent is -ar<antialiasrec>.

The recursion level can range from 0 (no anti-aliasing) to 4 (extremely heavy anti-aliasing). A value of 2 or 3 should be enought in most cases. Never use a recursion level of 1 though, as it is totally useless (it just blurs the picture, even turning anti-aliasing off looks better). For the different recursion levels, this is the maximum amount of rays that are traced for each pixel:

```
        Level:  Rays:
            0      1
            1      4
            2      9
            3     25
            4     81
```

Well, RayLab uses a very intelligent adaptive method (meaning it does not have to trace more rays than really necessary), so in reality only one to six rays have to be traced per pixel if there are not too many edges and contours in the picture.

1.102 sct4.12.7

4.12.7 Antialiasthreshold

As mentioned, RayLab uses an adaptive method for anti-aliasing. This is achived by comparing the colors from the four corners of a square (this square is in reality a pixel from the picture). If the difference between the colors is not too big, it is assumed that this entire square has the same color (which is the average of the four colors). If the difference exceeds a certain value, the first square is divided into four new squares, which are checked in the same manner. With antialiasthreshold you can specify the threshold for how big the difference between the colors may be before another recursion takes place. The value specified after the

keyword must be a decimal value ranging from 0.0 (all rays are cast as specified by the anti-aliasing recursion level) to 3.0 (no squares are subdivided). Example:

```
antialiasthreshold 0.4
```

The color difference is calculated as the sum of the maximum red, green and blue differences. The default value is 0.3, which seems to be a very good value. In some cases I have seen the need for stepping down to 0.2. The command line equivalent is -at<antialiasthreshold>.

1.103 sct4.12.8

4.12.8 Antialiasjitter

When the above mentioned anti-aliasing method is used, you may still be able to distinguish certain 'patterns' at sharp edges of objects and textures. To help prevent this, some jittering of the ray-directions may be added. This is done with the keyword antialiasjitter, which takes a value between 0.0 (no jittering) and 1.0 (very much jittering).

The default value is 0.05. The command line equivalent is -aj<antialiastjitter>.

1.104 sct4.12.9

4.12.9 Quickscan

Sometimes you just want to see how the objects in a scene are placed, and how the camera is set. Then the quickscan alternative is very useful. With quickscan activated, many of RayLabs features are disabled; no reflection or transparency is produced, most texture features are lost (such as patterns, images, highlights, diffuse and ambient) and no shadows are produced. All light-sources are skipped, and the light is considered to come from the point of the observer. As a result of all this, the rendering of a picture is faster than usual, much faster in many cases.

To activate the quickscan rendering method, add the keyword quickscan to a globals section. Anti-aliasing settings have no effect when quickscan is activated.

The default is no quickscan. The command line equivalent is -q.

1.105 sct4.12.10

4.12.10 Trace

Trace is just the opposite to quickscan. With the keyword trace you specify that you want all ray-tracing features of RayLab enabeled (such as transparency, reflection, highlights, shadows and texture patterns), which

is the default.

Trace is activated by default. The command line equivalent is -t.

1.106 sct4.12.11

4.12.11 Backgroundcolor

You can specify a color that will be the background color of your scene. For instance this can be very useful if you make an outdoor scene where you want the sky to be blue; simply set the background color to blue. Backgroundcolor is followed by three decimal values ranging from 0.0 (0%) to 1.0 (100%), each value representing the red, green and blue components of the color, respectively, in that order. Example:

```
backgroundcolor 0.3 0.0 0.0     # Dark red sky
```

The default background color is 0 0 0 (black). Backgroundcolor has no command line equivalent.

1.107 sct4.12.12

4.12.12 Fogdistance

An effect which does not require much rendering-time, but yet can produce very interesting results, is the global fog. By enabeling fog, the further away objects are, the less visible they are, and the more intense the fog is. With the keyword fogdistance, you specify the distance at which the fog-intensity is 63%. That number is derived from the formula that is used to calculate the fog-intensity:

```
fogintensity = 1 - exp(-distance/fogdistance)
```

Thus you can see, that at infinity the fog-intensity is 100%, and at zero-distance, the intensity is 0%. Example of fogdistance usage:

```
fogdistance 20
```

By specifying a fog-distance of less than or equal to zero, you disable fog.

The default is no fog. Fogdistance has no command line equivalent.

1.108 sct4.12.13

4.12.13 Fogcolor

To specify the color of the fog, use the keyword fogcolor. Example:

```
fogcolor 0.9 0.8 1.0    # Bright, slightly blue/purple fog
```

Apart from the normal grey/white fogs, you can use black fog for a nice fade-away effect, or blue fog for an under-water effect. Or how about some extreme applications? I used a bright yellow fog in a scene where the camera was placed inside a tunnel, which made it look like a bright light at the end of the tunnel, very cool!

The default fog color is 0.8 0.8 0.8 (light grey). Fogcolor has no command line equivalent.

1.109 sct4.13

4.13 Comments

In RayLab scene descriptions you can put comments to clarify your work, both to yourself and others. A comment can be placed anywhere, except in the middle of or right after (no space, tab or newline between) a keyword or a number. Every comment is started by a comment identifier and it is ended at the end of the line. Valid comment identifiers are # ; and *

Example:

```
Sphere:                 # This is a sphere
        centre  0 0 0   ; and it is placed in (0,0,0)
        radius          ****** ok, here comes the radius:
        2
:end
```

As you see, it is fully possible to put a comment between the keyword and its parameter(s), although it does not look very nice in this example.

1.110 sct5.0

5. MISCELLANEOUS
==========

How to contact the author

Legal stuff

Past, present and the future

Acknowledgements

Compability

Comments by the author

1.111 sct5.1

5.1 How to contact the author

I always like to get response from people who somehow come across my work, be it good or be it bad. So if you have used RayLab, read the documentation or heard about my latest assembler program for the C=64, please drop a mail in my e-box:

internet: e4geeln@etek.chalmers.se

To check out the latest news about RayLab and to get the latest version for any supported platform, you can go to The RayLab Home Page on the world wide web:

WWW: http://www.etek.chalmers.se/~e4geeln/raylab/

If you get any error-messages when compiling RayLab on your system, please let me know as it would be nice to have it compile flawlessly on as many systems as possible. Do not hesitate to report things like "strings.h not found..." (it worked fine on my Amiga and at the university, but not at a friends PC with an old Microsoft compiler. Now I know it should be string.h!!).

Any bug-reports etc. will of course be warmly welcomed. If you do not have access to internet, you can contact me by snail-mail:

s-mail: Marcus Geelnard
Utbynäsgatan 11
S-415 06 Göteborg
Sweden

1.112 sct5.2

5.2 Legal stuff

Here is what you should know about what you can do, and what you can not:

o RayLab is provided as is, and the author can not be held responsible for any system or hardware failure or data loss as a result, direct or indirect, of the use of RayLab. Use RayLab at your own risk.

o RayLab is totally free! You should not pay a penny for the actual software, nor can anybody claim any money for RayLab (except for formal fees for storage media or transfer costs). You may not sell RayLab for money or in any other way charge money for any part(s) of the RayLab package.

o You may NOT change RayLab in part or in whole, and then redistribute it! The files and the contents of the files must remain as is, and they must all be there with their original names and in their original directory structures!

o You may NOT use any part(s) of the RayLab sourcecode for your own productions, neither commercial nor noncommercial!

o If you want to use RayLab or any pictures that are produced with RayLab

for commercial purposes, you will have to get the permission to do so from the author of RayLab.

o For noncommercial use of RayLab, such as a private art-gallery, you need no permission from the author of RayLab.

o You may NOT spread your own compilation of RayLab without permission from the author of RayLab!

If anything is uncertain, or if you feel like breaking any of the above stated rules, e.g. if you want to improve RayLab, try contacting me (the author of RayLab) first. I am not unreasonable, but I do like to have some control of my own software.

1.113 sct5.3

5.3 Past, present and the future

Since the previous release of RayLab (1.0 rev 1), these changes/additions have been made:

o Transparency with refraction was added (at last).
o Constructive solid geometry (CSG) was added.
o Added compressed 24-bit IFF and 24-bit PPM image output.
o Added new patterns: marble (thanks to Tobias Mellqvist), squares, mandel, wood, blurb and angular.
o Texture turbulence was added.
o Added new transformations for textures: whirl and twist.
o Improved 'safe termination', which among other things means that you can view an image correctly that was only partly rendered before a CTRL-C.
o The cone primitive was added.
o The cylinder primitive was improved.
o Speedup with bounding volumes was implemented.
o The 'noshadow' option was added.
o Improved anti-aliasing with 'antialiasjitter' (and removed an old bug that limited the anti-aliasing recursion level to level three, which should have been four).
o The parsing routines have been improved a great deal, which means better handling of bad description files and less redundancy in the code.
o New display-modes were added for the Amiga, HAM8 in particular.
o A primitive model for soft shadows was introduced.
o The quick-scan rendering method was added for faster previews.
o Large portions of the program code was rearranged and improved, mostly due to the introduction of CSG.

Things that I would like to implement in RayLab, but I don't know when, or even if, it will be (it's a question of time and priority):

o Speeeedups! Main ideas that are rushing around in my head are: octree-optimization and light-buffers.
o Support for variable declarations in scene descriptions.
o Animation support, which will consist of a good expression handler and some useful mathematical functions like sin(x), sqrt(x), log(x) etc.
o Better (=faster) soft light-sources, and more light models (like spot-

lights).
o Surface normal modifiers, like bumpiness and waves.
o Even more patterns, and multiple weighted textures.
o Depth of field.
o Motion blur (based on the animation support, don't know how though?).
o Some other fun things that may pop up, like fish-eye camera-projection.
o .....more, more, more, drewl, pant, more!....

Any further suggestions are of course welcome...

1.114 sct5.4

5.4 Acknowledgements

I would like to thank many people, but the following people in particular, for their help and support:

o Brian Jones, for his dedication and his patient beta-testing of RayLab.

o Tobias Mellqvist, for feeding me with information and ideas.

o Andreas Magnusson, for his work on (the not yet finished) numerical methods of root-solving.

o Jean-Baptiste Novoit, for his help with documentation and for compiling RayLab for Linux and the BeBox.

1.115 sct5.5

5.5 Compability

When RayLab was designed, compability between platforms was a high-prioroty issue, and it still is. The major reason at first was that I use an Amiga 3000 at home, where most of the development has been done, but I use DEC-Alpha stations at the university (Chalmers university of technology), which are very fast, and thus very attractive for ray-tracing. So my goal was to be able to compile RayLab on both the Amiga platform and on the Unix platform.

A 'side-effect' of this is that it is possible to compile RayLab on almost any platform whitout any major changes/additions to the code (the system should be at least 32-bit though, like the Amiga 500). And for RayLab to gain users, it is a good thing that it can be used on many platforms, of course. I have compiled RayLab for these systems with success:

- Amiga, compiled with gcc 2.7.0 and Sas/C

- DEC-Alpha (OSF/1), compiled with gcc and cc

- HP-UX, compiled with cc

- 486-PC (no FPU) running OS/2, compiled with gcc 2.7.2

- Pentium-PC running DOS/Win 3.1, compiled with gcc 2.7.2

Worth noting is that the DEC-Alpha is a 64-bit system, which many programs have problems to cope with. In the RayLab code, no special assumptions about the sizes or formats of different variable-types are made. E.g. long int may be 32-bit, 64-bit, 48-bit, 256-bit or whatever, and no strange casting between different types are made (such as turning a (void *) into a (long int)). The only real requirement is that long int is at least 32 bits wide.

1.116 sct5.6

5.6 Comments by the author

I admit that RayLab was influenced by other raytracers that I have used (POV-Ray, Imagine, Rayshade etc.), but that is mostly due to my finding parts of those raytracers very natural. I did, however, start from scratch when I developed RayLab. I have not used or hardly even looked at sourcecode from other raytracers during the development of RayLab, so RayLab is based upon my own assumptions about how things work. One day I came across a book called 'Fundamentals of Three-Dimensional Computer Graphics' by Alan Watt. It was fun to see that what I had done was actually identical to what was described in that book, like how the general Phong shading model works.

There are some undocumented features of RayLab, which you will see if you study the source code, that I have chosen not to mention as they either do not work properly (yet) or are redundant. For instance you can specify the quickscan rendering method by adding the keyword 'quickscan' to a globals section, according to the documentation. You may also write 'rendermethod quickscan' or 'rendermethod quick'.

Pheew! This documentation, AND RayLab as a whole, got a bit larger than expected... This program was intended to be a simple personal hack, not a commercial raytracer (don't worry: RayLab will be freeware as long as I can keep it that way)!