POV-Ray Reference

POV-Team

for POV-Ray Version 3.6 BETA

Contents

1 Scene Description Language **15**
1.1 Language Basics 16
1.1.1 Identifiers and Keywords 16
1.1.2 Comments 21
1.1.3 Float Expressions 22
1.1.4 Vector Expressions 32
1.1.5 Specifying Colors 37
1.1.6 User-Defined Functions 43
1.1.7 Strings 48
1.1.8 Array Identifiers 51
1.1.9 Spline Identifiers 53
1.2 Language Directives 55
1.2.1 Include Files and the #include Directive 56
1.2.2 The #declare and #local Directives 56
1.2.3 File I/O Directives 60
1.2.4 The #default Directive 63
1.2.5 The #version Directive 64
1.2.6 Conditional Directives 65
1.2.7 User Message Directives 68
1.2.8 User Defined Macros 70

2 Scene Settings **75**
2.1 Command-line Options 75
2.1.1 Animation Options 76
2.1.2 General Output Options 79
2.1.3 Display Output Options 82
2.1.4 File Output Options 86
2.1.5 Scene Parsing Options 90
2.1.6 Shell-out to Operating System 93
2.1.7 Text Output 98
2.1.8 Tracing Options 102
2.2 Camera 108
2.2.1 Placing the Camera 109
2.2.2 Types of Projection 114
2.2.3 Focal Blur 116
2.2.4 Camera Ray Perturbation 117
2.2.5 Camera Identifiers 117

2.3 Atmospheric Effects . . . 118
2.3.1 Atmospheric Media . . . 118
2.3.2 Background . . . 119
2.3.3 Fog . . . 119
2.3.4 Sky Sphere . . . 121
2.3.5 Rainbow . . . 122
2.4 Global Settings . . . 123
2.4.1 ADC_Bailout . . . 124
2.4.2 Ambient_Light . . . 125
2.4.3 Assumed_Gamma . . . 125
2.4.4 HF_Gray_16 . . . 128
2.4.5 Irid_Wavelength . . . 128
2.4.6 Charset . . . 129
2.4.7 Max_Trace_Level . . . 129
2.4.8 Max_Intersections . . . 130
2.4.9 Number_Of_Waves . . . 130
2.4.10 Noise_generator . . . 130
2.4.11 Radiosity Basics . . . 131
2.5 Radiosity . . . 131
2.5.1 How Radiosity Works . . . 131
2.5.2 Adjusting Radiosity . . . 132
2.5.3 Tips on Radiosity . . . 136

3 Objects 137
3.1 Finite Solid Primitives . . . 138
3.1.1 Blob . . . 138
3.1.2 Box . . . 141
3.1.3 Cone . . . 141
3.1.4 Cylinder . . . 142
3.1.5 Height Field . . . 143
3.1.6 Julia Fractal . . . 146
3.1.7 Lathe . . . 149
3.1.8 Prism . . . 151
3.1.9 Sphere . . . 153
3.1.10 Spheresweep . . . 154
3.1.11 Superquadric Ellipsoid . . . 155
3.1.12 Surface of Revolution . . . 156
3.1.13 Text . . . 159
3.1.14 Torus . . . 159
3.2 Finite Patch Primitives . . . 160
3.2.1 Bicubic Patch . . . 161
3.2.2 Disc . . . 162
3.2.3 Mesh . . . 163
3.2.4 Mesh2 . . . 164
3.2.5 Polygon . . . 166
3.2.6 Triangle and Smooth Triangle . . . 167
3.3 Infinite Solid Primitives . . . 168
3.3.1 Plane . . . 168
3.3.2 Poly, Cubic and Quartic . . . 169
3.3.3 Quadric . . . 172

3.4 Isosurface Object . . . 173
3.5 Parametric Object . . . 175
3.6 Constructive Solid Geometry . . . 176
3.6.1 Inside and Outside . . . 176
3.6.2 Union . . . 177
3.6.3 Intersection . . . 179
3.6.4 Difference . . . 179
3.6.5 Merge . . . 180
3.7 Light Sources . . . 181
3.7.1 Point Lights . . . 182
3.7.2 Spotlights . . . 182
3.7.3 Cylindrical Lights . . . 186
3.7.4 Parallel Lights . . . 186
3.7.5 Area Lights . . . 187
3.7.6 Shadowless Lights . . . 191
3.7.7 Looks_like . . . 191
3.7.8 Projected_Through . . . 191
3.7.9 Light Fading . . . 192
3.7.10 Atmospheric Media Interaction . . . 193
3.7.11 Atmospheric Attenuation . . . 193
3.8 Light Groups . . . 193
3.9 Object Modifiers . . . 195
3.9.1 Clipped_By . . . 195
3.9.2 Bounded_By . . . 196
3.9.3 Material . . . 197
3.9.4 Inverse . . . 198
3.9.5 Hollow . . . 199
3.9.6 No_Shadow . . . 199
3.9.7 No_Image, No_Reflection . . . 200
3.9.8 Double_Illuminate . . . 200
3.9.9 Sturm . . . 200

4 Textures **203**
4.1 Pigment . . . 205
4.1.1 Solid Color Pigments . . . 206
4.1.2 Color List Pigments . . . 207
4.1.3 Color Maps . . . 207
4.1.4 Pigment Maps and Pigment Lists . . . 209
4.1.5 Image Maps . . . 210
4.1.6 Quick Color . . . 213
4.2 Normal . . . 213
4.2.1 Slope Maps . . . 215
4.2.2 Normal Maps and Normal Lists . . . 218
4.2.3 Bump Maps . . . 219
4.2.4 Scaling normals . . . 221
4.3 Finish . . . 221
4.3.1 Ambient . . . 222
4.3.2 Diffuse Reflection Items . . . 223
4.3.3 Highlights . . . 224
4.3.4 Specular Reflection . . . 226

4.3.5 Conserve Energy for Reflection . . . 228
4.3.6 Iridescence . . . 228
4.4 Halo . . . 229
4.5 Patterned Textures . . . 229
4.5.1 Texture Maps . . . 230
4.5.2 Tiles . . . 231
4.5.3 Material Maps . . . 232
4.6 Layered Textures . . . 234
4.7 UV Mapping . . . 236
4.7.1 Supported Objects . . . 236
4.7.2 UV Vectors . . . 237
4.8 Triangle Texture Interpolation . . . 238
4.9 Interior Texture . . . 238
4.10 Cutaway Textures . . . 239
4.11 Patterns . . . 239
4.11.1 Agate . . . 240
4.11.2 Average . . . 241
4.11.3 Boxed . . . 242
4.11.4 Bozo . . . 242
4.11.5 Brick . . . 243
4.11.6 Bumps . . . 243
4.11.7 Cells . . . 244
4.11.8 Checker . . . 244
4.11.9 Crackle Patterns . . . 245
4.11.10 Cylindrical . . . 247
4.11.11 Density_File . . . 247
4.11.12 Dents . . . 248
4.11.13 Facets . . . 248
4.11.14 Fractal Patterns . . . 249
4.11.15 Function as pattern . . . 251
4.11.16 Function Image . . . 252
4.11.17 Gradient . . . 253
4.11.18 Granite . . . 254
4.11.19 Hexagon . . . 254
4.11.20 Image Pattern . . . 255
4.11.21 Leopard . . . 256
4.11.22 Marble . . . 257
4.11.23 Object Pattern . . . 257
4.11.24 Onion . . . 258
4.11.25 Pigment Pattern . . . 258
4.11.26 Planar . . . 259
4.11.27 Quilted . . . 259
4.11.28 Radial . . . 262
4.11.29 Ripples . . . 262
4.11.30 Slope . . . 262
4.11.31 Spherical . . . 264
4.11.32 Spiral1 . . . 264
4.11.33 Spiral2 . . . 265
4.11.34 Spotted . . . 265
4.11.35 Waves . . . 265

4.11.36 Wood 266
4.11.37 Wrinkles 266
4.12 Pattern Modifiers 267
4.12.1 Transforming Patterns 268
4.12.2 Frequency and Phase 269
4.12.3 Waveforms 270
4.12.4 Noise Generators 270
4.12.5 Turbulence 271
4.12.6 Warps 271
4.12.7 Bitmap Modifiers 280

5 Interior & Media & Photons 283
5.1 Interior 283
5.1.1 Why are Interior and Media Necessary? 284
5.1.2 Empty and Solid Objects 285
5.1.3 Scaling objects with an interior 285
5.1.4 Refraction 287
5.1.5 Dispersion 288
5.1.6 Attenuation 289
5.1.7 Simulated Caustics 289
5.1.8 Object-Media 290
5.2 Media 290
5.2.1 Media Types 292
5.2.2 Sampling Parameters & Methods 294
5.2.3 Density 296
5.3 Photons 299
5.3.1 Overview 299
5.3.2 Using Photon Mapping in Your Scene 300
5.3.3 Photons FAQ 306
5.3.4 Photon Tips 307
5.3.5 Advanced Techniques 308

6 Include Files 311
6.1 arrays.inc 311
6.2 chars.inc 312
6.3 colors.inc 313
6.3.1 Predefined colors 313
6.3.2 Color macros 314
6.4 consts.inc 316
6.4.1 Vector constants 316
6.4.2 Map type constants 316
6.4.3 Interpolation type constants 316
6.4.4 Fog type constants 316
6.4.5 Focal blur hexgrid constants 316
6.4.6 IORs 317
6.4.7 Dispersion amounts 317
6.4.8 Scattering media type constants 318
6.5 debug.inc 318
6.6 finish.inc 318
6.7 functions.inc 319

6.7.1 Common Parameters . . . 320
6.7.2 Internal Functions . . . 321
6.7.3 Pre defined functions . . . 335
6.8 glass.inc, glass_old.inc . . . 336
6.8.1 Glass colors (with transparency) . . . 336
6.8.2 Glass colors (without transparency, for fade_color) . . . 338
6.8.3 Glass finishes . . . 338
6.8.4 Glass interiors . . . 338
6.8.5 Glass interior macros . . . 338
6.8.6 glass_old.inc . . . 338
6.9 math.inc . . . 339
6.9.1 Float functions and macros . . . 339
6.9.2 Vector functions and macros . . . 342
6.9.3 Vector Analysis . . . 344
6.10 metals.inc, golds.inc . . . 345
6.10.1 metals.inc . . . 345
6.10.2 golds.inc . . . 347
6.11 rand.inc . . . 347
6.11.1 Flat Distributions . . . 347
6.11.2 Other Distributions . . . 348
6.12 shapes.inc, shapes_old.inc, shapes2.inc, shapesq.inc . . . 351
6.12.1 shapes.inc . . . 351
6.12.2 shapes_old.inc . . . 358
6.12.3 shapes2.inc . . . 359
6.12.4 shapesq.inc . . . 360
6.13 skies.inc, stars.inc . . . 362
6.13.1 skies.inc . . . 363
6.13.2 stars.inc . . . 364
6.14 stones.inc, stones1.inc, stones2.inc, stoneold.inc . . . 364
6.14.1 stones1.inc . . . 364
6.14.2 stones2.inc . . . 369
6.15 stdinc.inc . . . 370
6.16 strings.inc . . . 370
6.17 textures.inc . . . 371
6.17.1 Stones . . . 371
6.17.2 Skies . . . 372
6.17.3 Woods . . . 373
6.17.4 Glass . . . 374
6.17.5 Metals . . . 374
6.17.6 Special textures . . . 375
6.17.7 Texture and pattern macros . . . 376
6.18 transforms.inc . . . 377
6.19 woodmaps.inc, woods.inc . . . 379
6.19.1 woodmaps.inc . . . 380
6.19.2 woods.inc . . . 380
6.20 Other files . . . 381
6.20.1 logo.inc . . . 382
6.20.2 rad_def.inc . . . 382
6.20.3 screen.inc . . . 383
6.20.4 stdcam.inc . . . 384

6.20.5 stage1.inc . . . 384
6.20.6 sunpos.inc . . . 384
6.20.7 font files (*.ttf) . . . 385
6.20.8 color_map files (*.map) . . . 385
6.20.9 image files (*.png, *.pot, *.df3) . . . 386

7 Quick Reference 387

7.1 Quick Reference Contents . . . 388
7.2 The Scene . . . 388
7.3 Language Basics . . . 388
7.3.1 Floats . . . 388
7.3.2 Vectors . . . 391
7.3.3 Colors . . . 392
7.3.4 User defined Functions . . . 392
7.3.5 Strings . . . 394
7.3.6 Arrays . . . 395
7.3.7 Splines . . . 396
7.4 Language Directives . . . 396
7.4.1 File Inclusion . . . 397
7.4.2 Identifier Declaration . . . 397
7.4.3 File Input/Output . . . 397
7.4.4 Default Texture . . . 398
7.4.5 Version Identfier . . . 398
7.4.6 Control Flow Directives . . . 399
7.4.7 Message Streams . . . 399
7.4.8 Macro . . . 399
7.4.9 Embedded Directives . . . 400
7.5 Transformations . . . 400
7.6 Camera . . . 401
7.7 Lights . . . 401
7.7.1 Lightgroup . . . 402
7.8 Objects . . . 402
7.8.1 Finite Solid Objects . . . 403
7.8.2 Finite Patch Objects . . . 406
7.8.3 Infinite Solid Objects . . . 409
7.8.4 Isosurface . . . 410
7.8.5 Parametric . . . 410
7.8.6 CSG . . . 411
7.9 Object Modifiers . . . 412
7.9.1 UV Mapping . . . 412
7.9.2 Material . . . 413
7.9.3 Interior . . . 413
7.9.4 Interior Texture . . . 413
7.10 Texture . . . 413
7.10.1 Plain Texture . . . 414
7.10.2 Layered Texture . . . 414
7.10.3 Patterned Texture . . . 414
7.10.4 Pigment . . . 415
7.10.5 Normal . . . 417
7.10.6 Finish . . . 418

7.10.7 Pattern 419
7.10.8 Pattern Modifiers 420
7.11 Media 422
7.12 Atmospheric Effects 423
7.12.1 Background 423
7.12.2 Fog 424
7.12.3 Sky Sphere 424
7.12.4 Rainbow 424
7.13 Global Settings 425
7.13.1 Radiosity 425
7.13.2 Photons 425

Figures

2.1 Display gamma test image. 84
2.2 Example of how the recursive super-sampling works. 107
2.3 The perspective camera. 110

3.1 The geometry of a box. 142
3.2 The geometry of a cone. 142
3.3 The geometry of a cylinder. 143
3.4 The size and orientation of an un-scaled height field. 144
3.5 Relationship of pixels and triangles in a height field. 144
3.6 The geometry of a sphere. 154
3.7 Points on a surface of revolution. 158
3.8 Major and minor radius of a torus. 160
3.9 Two overlapping objects. 177
3.10 The union of two objects. 178
3.11 The intersection of two objects. 179
3.12 The difference between two objects. 180
3.13 Merge removes inner surfaces. 181
3.14 The geometry of a spotlight. 183
3.15 Intensity multiplier curve with a fixed falloff angle of 45 degrees. . . . 184
3.16 Intensity multiplier curve with a fixed radius angle of 45 degrees. . . . 184
3.17 Intensity multiplier curve with fixed angle and falloff angles of 30 and 60 degrees respectively and different tightness values. 185
3.18 Intensity multiplier curve with a negative radius angle and different tightness values. 185
3.19 4x4 Area light, location and vectors. 188
3.20 Area light adaptive samples. 189
3.21 Area light facing object 190
3.22 Area light not facing object 190
3.23 Light fading functions for different fading powers. 192
3.24 An object clipped by another object. 196

4.1 UV Boxmap 237
4.2 The hexagon pattern. 254
4.3 Quilted pattern with c0=0 and different values for c1. 260
4.4 Quilted pattern with c0=0.33 and different values for c1. 260
4.5 Quilted pattern with c0=0.67 and different values for c1. 261
4.6 Quilted pattern with c0=1 and different values for c1. 261
4.7 Turbulence random walk. 278

5.1 The Mie 293
5.2 The Mie 294
5.3 The Rayleigh scattering function. 294
5.4 The Henyey-Greenstein scattering function for different eccentricity values. 295
5.5 Reflective caustics 300
5.6 Photons used for lenses and caustics 300
5.7 Example of the photon autostop option 308

6.1 Primary Colors 313
6.2 Shades of Gray 313
6.3 Shades of Gray 314
6.4 Misc. Colors Part 1 314
6.5 Misc. Colors Part 2 315

Tables

1.1 Arithmetic expressions 24
1.2 Relational expressions 25
1.3 Logical expressions 25
1.4 Conditional expressions 25
1.5 All language directives 55
1.6 All character escape sequences 69

2.1 75
2.2 76
2.3 76
2.4 78
2.5 78
2.6 79
2.7 79
2.8 80
2.9 81
2.10 81
2.11 82
2.12 84
2.13 85
2.14 86
2.15 86
2.16 86
2.17 88
2.18 89
2.19 89
2.20 89
2.21 90
2.22 90
2.23 91
2.24 91
2.25 92
2.26 92
2.27 93
2.28 94
2.29 95
2.30 95
2.31 96

2.32 96
2.33 97
2.34 97
2.35 98
2.36 100
2.37 101
2.38 102
2.39 102
2.40 103
2.41 103
2.42 103
2.43 104
2.44 105
2.45 107

3.1 Quaternion basis vector multiplication rules 148
3.2 Hypercomplex basis vector multiplication rules 148
3.3 Function Keyword Maps 4-D value of h 149
3.4 Cubic and quartic polynomial terms 171
3.5 Some quartic shapes 172

6.1 glass.inc glass colors with transparency 336
6.2 glass.inc glass colors without transparency for fade_color 337

7.1 Quick Reference Overview 389

Chapter 1

Scene Description Language

The reference section describes the POV-Ray *scene description language*. It is supposed to be used as a reference for looking up things. It does not contain detailed explanations on how scenes are written or how POV-Ray is used. It just explains all features, their syntax, applications, limits, drawbacks, etc.

The scene description language allows you to describe the world in a readable and convenient way. Files are created in plain ASCII text using an editor of your choice. The input file name is specified using the `Input_File_Name=`*file* option or `+I`*file* switch. By default the files have the extension `.pov`. POV-Ray reads the file, processes it by creating an internal model of the scene and then renders the scene.

The overall syntax of a scene is shown below. See "Notation and Basic Assumptions" for more information on syntax notation.

```
SCENE:
    SCENE_ITEM...
SCENE_ITEM:
    LANGUAGE_DIRECTIVES        |
    camera { CAMERA_ITEMS... } |
    OBJECTS                    |
    ATMOSPHERIC_EFFECTS        |
    global_settings { GLOBAL_ITEMS }
```

In plain English, this means that a scene contains one or more scene items and that a scene item may be any of the five items listed below it. The items may appear in any order. None is a required item. In addition to the syntax depicted above, a *LANGUAGE_DIRECTIVE* may also appear anywhere embedded in other statements between any two tokens. There are some restrictions on nesting directives also.

For details on those five items see section "Language Directives", section "Objects", section "Camera", section "Atmospheric Effects" and section "Global Settings" for details.

1.1 Language Basics

The POV-Ray language consists of identifiers, reserved keywords, floating point expressions, strings, special symbols and comments. The text of a POV-Ray scene file is free format. You may put statements on separate lines or on the same line as you desire. You may add blank lines, spaces or indentations as long as you do not split any keywords or identifiers.

1.1.1 Identifiers and Keywords

POV-Ray allows you to define identifiers for later use in the scene file. An identifier may be 1 to 40 characters long. It may consist of upper and lower case letters, the digits 0 through 9 or an underscore character ("_"). the first character must be an alphabetic character. The declaration of identifiers is covered later.

POV-Ray has a number of reserved keywords which are listed below.

a

aa_level
aa_threshold
abs
absorption
accuracy
acos
acosh
adaptive
adc_bailout
agate
agate_turb
all
all_intersections
alpha
altitude
always_sample
ambient
ambient_light
angle
aperture
append
arc_angle
area_light
array
asc
ascii
asin
asinh
assumed_gamma
atan
atan2
atanh
autostop
average

b

b_spline
background
bezier_spline
bicubic_patch
black_hole
blob
blue
blur_samples
bounded_by
box
boxed
bozo
break
brick
brick_size
brightness
brilliance
bump_map
bump_size
bumps

c

camera
case
caustics
ceil
cells
charset
checker
chr
circular
clipped_by
clock
clock_delta
clock_on
collect
color

color_map
colour
colour_map
component
composite
concat
cone
confidence
conic_sweep
conserve_energy
contained_by
control0
control1
coords
cos
cosh
count
crackle
crand
cube
cubic
cubic_spline
cubic_wave
cutaway_textures
cylinder
cylindrical

d

debug
declare
default
defined
degrees
density
density_file
density_map
dents
df3
difference
diffuse
dimension_size
dimensions
direction
disc
dispersion
dispersion_samples
dist_exp
distance
div
double_illuminate

e

eccentricity
else
emission
end
error
error_bound
evaluate
exp
expand_thresholds
exponent
exterior
extinction

f

face_indices
facets
fade_color
fade_colour
fade_distance
fade_power
falloff
falloff_angle
false
fclose
file_exists
filter
final_clock
final_frame
finish
fisheye
flatness
flip
floor
focal_point
fog
fog_alt
fog_offset
fog_type
fopen
form
frame_number
frequency
fresnel
function

g

gather
gif
global_lights
global_settings
gradient
granite
gray
gray_threshold
green

h

height_field	hierarchy	
hexagon	hypercomplex	
hf_gray_16	hollow	

i

if	initial_clock	intersection
ifdef	initial_frame	intervals
iff	inside	inverse
ifndef	inside_vector	ior
image_height	int	irid
image_map	interior	irid_wavelength
image_pattern	interior_texture	isosurface
image_width	internal	
include	interpolate	

j

jitter	julia	
jpeg	julia_fractal	

l

lambda	linear_sweep	look_at
lathe	ln	looks_like
leopard	load_file	low_error_factor
light_group	local	
light_source	location	
linear_spline	log	

m

macro	max_gradient	mesh2
magnet	max_intersections	metallic
major_radius	max_iteration	method
mandel	max_sample	metric
map_type	max_trace	min
marble	max_trace_level	min_extent
material	media	minimum_reuse
material_map	media_attenuation	mod
matrix	media_interaction	mortar
max	merge	
max_extent	mesh	

n

natural_spline
nearest_count
no
no_bump_scale
no_image
no_reflection
no_shadow
noise_generator
normal
normal_indices
normal_map
normal_vectors
number_of_waves

o

object
octaves
off
offset
omega
omnimax
on
once
onion
open
orient
orientation
orthographic

p

panoramic
parallel
parametric
pass_through
pattern
perspective
pgm
phase
phong
phong_size
photons
pi
pigment
pigment_map
pigment_pattern
planar
plane
png
point_at
poly
poly_wave
polygon
pot
pow
ppm
precision
precompute
pretrace_end
pretrace_start
prism
prod
projected_through
pwr

q

quadratic_spline
quadric
quartic
quaternion
quick_color
quick_colour
quilted

r

radial
radians
radiosity
radius
rainbow
ramp_wave
rand
range
ratio
read
reciprocal
recursion_limit
red
reflection
reflection_exponent
refraction
render
repeat
rgb
rgbf
rgbft
rgbt
right
ripples
rotate
roughness

s

samples
save_file
scale
scallop_wave
scattering
seed
select
shadowless
sin
sine_wave
sinh
size
sky
sky_sphere
slice
slope
slope_map
smooth
smooth_triangle
solid
sor
spacing
specular
sphere
sphere_sweep
spherical
spiral1
spiral2
spline
split_union
spotlight
spotted
sqr
sqrt
statistics
str
strcmp
strength
strlen
strlwr
strupr
sturm
substr
sum
superellipsoid
switch
sys

t

t
tan
tanh
target
text
texture
texture_list
texture_map
tga
thickness
threshold
tiff
tightness
tile2
tiles
tolerance
toroidal
torus
trace
transform
translate
transmit
triangle
triangle_wave
true
ttf
turb_depth
turbulence
type

u

u
u_steps
ultra_wide_angle
undef
union
up
use_alpha
use_color
use_colour
use_index
utf8
uv_indices
uv_mapping
uv_vectors

v

v
v_steps
val
variance
vaxis_rotate
vcross
vdot
version
vertex_vectors
vlength
vnormalize
vrotate
vstr
vturbulence

w

warning	while	write
warp	width	
water_level	wood	
waves	wrinkles	

x

x

y

y	yes

z

z

All reserved words are fully lower case. Therefore it is recommended that your identifiers contain at least one upper case character so it is sure to avoid conflict with reserved words.

1.1.2 Comments

Comments are text in the scene file included to make the scene file easier to read or understand. They are ignored by the ray-tracer and are there for your information. There are two types of comments in POV-Ray.

Two slashes are used for single line comments. Anything on a line after a double slash (`//`) is ignored by the ray-tracer. For example:

```
// This line is ignored
```

You can have scene file information on the line in front of the comment as in:

```
object { FooBar } // this is an object
```

The other type of comment is used for multiple lines. It starts with "`/*`" and ends with "`*/`". Everything in-between is ignored. For example:

```
/* These lines
   are ignored
   by the
   ray-tracer */
```

This can be useful if you want to temporarily remove elements from a scene file. `/* ... */` comments can *comment out* lines containing other `//` comments and thus can be used to temporarily or permanently comment out parts of a scene. `/* ... */` comments can be nested, the following is legal:

```
/* This is a comment
// This too
/* This also */
*/
```

Use comments liberally and generously. Well used, they really improve the readability of scene files.

1.1.3 Float Expressions

Many parts of the POV-Ray language require you to specify one or more floating point numbers. A floating point number is a number with a decimal point. Floats may be specified using literals, identifiers or functions which return float values. You may also create very complex float expressions from combinations of any of these using various familiar operators.

Where POV-Ray needs an integer value it allows you to specify a float value and it truncates it to an integer. When POV-Ray needs a logical or boolean value it interprets any non-zero float as true and zero as false. Because float comparisons are subject to rounding errors POV-Ray accepts values extremely close to zero as being false when doing boolean functions. Typically values whose absolute values are less than a preset value *epsilon* are considered false for logical expressions. The value of *epsilon* is system dependent but is generally about 1.0e-10. Two floats *a* and *b* are considered to be equal if *abs(a-b) < epsilon.*

The full syntax for float expressions is given below. Detailed explanations are given in the following sub-sections.

```
FLOAT:
    NUMERIC_TERM [SIGN NUMERIC_TERM]...
SIGN:
    + | -
NUMERIC_TERM:
    NUMERIC_FACTOR [MULT NUMERIC_FACTOR]...
MULT:
    * | /
NUMERIC_FACTOR:
    FLOAT_LITERAL        |
    FLOAT_IDENTIFIER     |
    SIGN NUMERIC_FACTOR  |
    FLOAT_FUNCTION       |
    FLOAT_BUILT-IN_IDENT |
    ( FULL_EXPRESSION )  |
    ! NUMERIC_FACTOR
VECTOR DECIMAL_POINT DOT_ITEM FLOAT_LITERAL:
    [DIGIT...] [DECIMAL_POINT] DIGIT... [EXP [SIGN] DIGIT...]
DIGIT:
    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
DECIMAL_POINT:
    .
EXP:
    e | E
DOT_ITEM:
    x | y | z | t | u | v | red | blue | green | filter |
    transmit | gray
FLOAT_FUNCTION:
    abs( FLOAT ) | acos( FLOAT ) | acosh( FLOAT ) | asc( STRING ) |
    asin( FLOAT ) | asinh( FLOAT ) | atan( FLOAT) | atanh( FLOAT) |
```

```
    atan2( FLOAT , FLOAT ) | ceil( FLOAT ) | cos( FLOAT ) |
    cosh( FLOAT ) | defined(IDENTIFIER ) | degrees( FLOAT ) |
    dimensions( ARRAY_IDENTIFIER ) |
    dimension_size( ARRAY_IDENTIFIER , FLOAT ) |
    div( FLOAT , FLOAT ) | exp( FLOAT ) | file_exists( STRING ) |
    floor( FLOAT ) | int( FLOAT ) | ln(Float | log( FLOAT ) |
    max( FLOAT , FLOAT, ... ) | min( FLOAT , FLOAT, ... ) |
    mod( FLOAT , FLOAT ) | pow( FLOAT , FLOAT ) |
    radians( FLOAT ) | rand( FLOAT ) | seed( FLOAT ) |
    select( FLOAT, FLOAT, FLOAT [,FLOAT]) | sin( FLOAT ) |
    sinh( FLOAT ) | sqrt( FLOAT ) | strcmp( STRING , STRING ) |
    strlen( STRING ) | tan( FLOAT ) | tanh( FLOAT ) |
    val( STRING ) | vdot( VECTOR , VECTOR ) | vlength( VECTOR ) |
    FLOAT_BUILT-IN_IDENT:
    clock | clock_delta | clock_on | false | final_clock |
    final_frame | frame_number | initial_clock | initial_frame |
    image_width | image_height | no | off | on | pi | true |
    version | yes |
FULL_EXPRESSION:
    LOGICAL_EXPRESSION [? FULL_EXPRESSION : FULL_EXPRESSION]
LOGICAL_EXPRESSION:
    REL_TERM [LOGICAL_OPERATOR REL_TERM]...
LOGICAL_OPERATOR:
    & | | <em>(note: this means an ampersand or a vertical bar is a logical operator)</em>
REL_TERM:
    FLOAT [REL_OPERATOR FLOAT]...
REL_OPERATOR:
    < | <= | = | >= | > | !=
INT:
    FLOAT <em>(note any syntax which requires a integer INT will accept a FLOAT and it will be truncated
```

Note: *FLOAT_IDENTIFIERS* are identifiers previously declared to have float values. The *DOT_ITEM* syntax is actually a vector or color operator but it returns a float value. See “Vector Operators” or “Color Operators” for details. An *ARRAY_IDENTIFIER* is just the identifier name of a previously declared array, it does not include the `[]` braces nor the index. The syntax for *STRING* is in the section “Strings”.

Literals

Float literals are represented by an optional sign (“+” or “-”) digits, an optional decimal point and more digits. If the number is an integer you may omit the decimal point and trailing zero. If it is all fractional you may omit the leading zero. POV-Ray supports scientific notation for very large or very small numbers. The following are all valid float literals:

```
 -2.0 -4 34 3.4e6 2e-5 .3 0.6
```

Identifiers

Float identifiers may be declared to make scene files more readable and to parameterize scenes so that changing a single declaration changes many values. An identifier is declared as follows.

```
FLOAT_DECLARATION:
    #declare IDENTIFIER = EXPRESSION;    |
    #local IDENTIFIER = EXPRESSION;
```

Where *IDENTIFIER* is the name of the identifier up to 40 characters long and *EXPRESSION* is any valid expression which evaluates to a float value.

Note: there should be a semi-colon after the expression in a float declaration. If omitted, it generates a warning and some macros may not work properly. See " #declare vs. #local" for information on identifier scope.

Here are some examples.

```
#declare Count = 0;
#declare Rows = 5.3;
#declare Cols = 6.15;
#declare Number = Rows*Cols;
#declare Count = Count+1;
```

As the last example shows, you can re-declare a float identifier and may use previously declared values in that re-declaration. There are several built-in identifiers which POV-Ray declares for you. See "Built-in Float Identifiers" for details.

Operators

Arithmetic expressions: Basic math expressions can be created from float literals, identifiers or functions using the following operators in this order of precedence...

`( )`	expressions in parentheses first
`+A -A !A`	unary minus, unary plus and logical "not"
`A*B A/B`	multiplication and division
`A+B A-B`	addition and subtraction

Table 1.1: Arithmetic expressions

Relational, logical and conditional expressions may also be created. However there is a restriction that these types of expressions must be enclosed in parentheses first. This restriction, which is not imposed by most computer languages, is necessary because POV-Ray allows mixing of float and vector expressions. Without the parentheses there is an ambiguity problem. Parentheses are not required for the unary logical not operator "!" as shown above. The operators and their precedence are shown here.

Relational expressions: The operands are arithmetic expressions and the result is always boolean with 1 for true and 0 for false. All relational operators have the same precedence.

Logical expressions: The operands are converted to boolean values of 0 for false and 1 for true. The result is always boolean. All logical operators have the same precedence.

Note: these are not bit-wise operations, they are logical.

Conditional expressions: The operand C is boolean while operands A and B are any expressions. The result is of the same type as A and B.

`(A < B)`	A is less than B
`(A <= B)`	A is less than or equal to B
`(A = B)`	A is equal to B (actually abs(A-B)<EPSILON)
`(A != B)`	A is not equal to B (actually abs(A-B)>=EPSILON)
`(A >= B)`	A is greater than or equal to B
`(A > B)`	A is greater than B

Table 1.2: Relational expressions

`(A & B)`	true only if both A and B are true, false otherwise
`(A | B)`	true if either A or B or both are true

Table 1.3: Logical expressions

Assuming the various identifiers have been declared, the following are examples of valid expressions...

```
1+2+3    2*5     1/3     Row*3    Col*5
(Offset-5)/2      This/That+Other*Thing
((This<That) & (Other>=Thing)?Foo:Bar)
```

Expressions are evaluated left to right with innermost parentheses evaluated first, then unary +, - or !, then multiply or divide, then add or subtract, then relational, then logical, then conditional.

Functions

POV-Ray defines a variety of built-in functions for manipulating floats, vectors and strings. Function calls consist of a keyword which specifies the name of the function followed by a parameter list enclosed in parentheses. Parameters are separated by commas. For example:

```
keyword(param1,param2)
```

The following are the functions which return float values. They take one or more float, integer, vector, or string parameters. Assume that `A` and `B` are any valid expression that evaluates to a float; `I` is a float which is truncated to integer internally, `S`, `S1`, `S2` etc. are strings, and `V`, `V1`, `V2` etc. are any vector expressions.`O` is an object identifier to a pre-declared object.

`abs(A)` Absolute value of `A`. If `A` is negative, returns `-A` otherwise returns `A`.

`acos(A)` Arc-cosine of `A`. Returns the angle, measured in radians, whose cosine is `A`.

`acosh(A)` inverse hyperbolic cosine of `A`.

`asc(S)` Returns an integer value in the range 0 to 255 that is the ASCII value of the first character of the string `S`. For example `asc(''ABC'')` is 65 because that is the value of the character “A”.

`(C ? A : B)`	if C then A else B

Table 1.4: Conditional expressions

`asin(A)` Arc-sine of `A`. Returns the angle, measured in radians, whose sine is `A`.

`asinh(A)` invers hyperbolic sine of `A`

`atan2(A,B)` Arc-tangent of `(A/B)`. Returns the angle, measured in radians, whose tangent is `(A/B)`. Returns appropriate value even if `B` is zero. Use `atan2(A,1)` to compute usual atan(A) function.

`atanh(A)` invers hyperbolic tangent of `A`

`ceil(A)` Ceiling of `A`. Returns the smallest integer greater than `A`. Rounds up to the next higher integer.

`cos(A)` Cosine of `A`. Returns the cosine of the angle `A`, where `A` is measured in radians.

`cosh(A)` The hyperbolic cosine of `A`.

`defined(`*IDENTIFIER*`)` Returns `true` if the identifier is currently defined, `false` otherwise. This is especially useful for detecting end-of-file after a `#read` directive because the file identifier is automatically undefined when end-of-file is reached. See “The #read Directive” for details.

`degrees(A)` Convert radians to degrees. Returns the angle measured in degrees whose value in radians is `A`. Formula is *degrees=A/pi*180.0*.

`dimensions(` *ARRAY_IDENTIFIER* `)` Returns the number of dimensions of a previously declared array identifier. For example if you do `#declare MyArray=array[6][10]` then `dimensions(MyArray)` returns the value `2`.

`dimension_size(` *ARRAY_IDENTIFIER, FLOAT* `)` Returns the size of a given dimension of a previously declared array identifier. Dimensions are numbered left-to-right starting with 1. For example if you do `#declare MyArray=array[6][10]` then `dimension_size(MyArray,2)` returns the value `10`.

`div(A,B)` Integer division. The integer part of `(A/B)`.

`exp(A)` Exponential of `A`. Returns the value of *e* raised to the power `A` where *e* is the base of the natural logarithm, i.e. the non-repeating value approximately equal to 2.71828182846.

`file_exists(S)` Attempts to open the file specified by the string `S`. The current directory and all library directories specified by the `Library_Path` or `+L` options are also searched. See “Library Paths” for details. Returns `1` if successful and `0` if unsuccessful.

`floor(A)` Floor of `A`. Returns the largest integer less than `A`. Rounds down to the next lower integer.

`inside(O,V)` It returns either 0.0, when the vector `V` is outside the object, specified by the object-identifier `O`, or 1.0 if it is inside.

Note: `inside` does not accept object-identifiers to non-solid objects.

`int(A)` Integer part of `A`. Returns the truncated integer part of `A`. Rounds towards zero.

`log(A)` Logarithm of `A`. Returns the logarithm base *10* of the value `A`.

`ln(A)` Natural logarithm of `A`. Returns the natural logarithm base *e* of the value `A`.

`max(A,B,...)` Maximum of two or more float values. Returns `A` if `A` larger than `B`. Otherwise returns `B`.

`min(A,B,...)` Minimum of two or more float values. Returns `A` if `A` smaller than `B`. Otherwise returns `B`.

`mod(A,B)` Value of `A` modulo `B`. Returns the remainder after the integer division of `A/B`. Formula is *mod=((A/B)-int(A/B))*B*.

`pow(A,B)` Exponentiation. Returns the value of `A` raised to the power `B`.

Note: For a negative A and a non-integer B the function has no defined return value. The result then may depend on the platform POV-Ray is compiled on.

`radians(A)` Convert degrees to radians. Returns the angle measured in radians whose value in degrees is `A`. Formula is *radians=A*pi/180.0*.

`rand(I)` Returns the next pseudo-random number from the stream specified by the positive integer `I`. You must call `seed()` to initialize a random stream before calling `rand()`. The numbers are uniformly distributed, and have values between `0.0` and `1.0`, inclusively. The numbers generated by separate streams are independent random variables.

`seed(I)` Initializes a new pseudo-random stream with the initial seed value `A`. The number corresponding to this random stream is returned. Any number of pseudo-random streams may be used as shown in the example below:

```
#declare R1 = seed(0);
#declare R2 = seed(12345);
sphere { <rand(R1), rand(R1), rand(R1)>, rand(R2) }
```

Multiple random generators are very useful in situations where you use `rand()` to place a group of objects, and then decide to use `rand()` in another location earlier in the file to set some colors or place another group of objects. Without separate `rand()` streams, all of your objects would move when you added more calls to `rand()`. This is very annoying.

`select(A, B, C [,D])`. It can be used with three or four parameters. `Select` compares the first argument with zero, depending on the outcome it will return `B`, `C` or `D`. `A,B,C,D` can be floats or funtions.
When used with three parameters, if `A < 0` it will return `B`, else `C` (`A >= 0`).
When used with four parameters, if `A < 0` it will return B. If `A = 0` it will return `C`. Else it will return `D` (`A > 0`).

Example:
If `A` has the consecutive values -2, -1, 0, 1, and 2 :

```
                          //    A =  -2 -1 0 1 2
select (A, -1, 0, 1)      //returns  -1 -1 0 1 1
select (A, -1, 1)         //returns  -1 -1 1 1 1
```

`sin(A)AA A`

`sinh(A)` The hyperbolic sine of `A`.

`strcmp(S1,S2)` Compare string `S1` to `S2`. Returns a float value zero if the strings are equal, a positive number if `S1` comes after `S2` in the ASCII collating sequence, else a negative number.

`strlen(S)` Length of `S`. Returns an integer value that is the number of characters in the string `S`.

`sqrt(A)` Square root of `A`. Returns the value whose square is `A`.

`tan(A)` Tangent of `A`. Returns the tangent of the angle `A`, where `A` is measured in radians.

`tanh(A)` The hyperbolic tangent of `A`.

`val(S)` Convert string `S` to float. Returns a float value that is represented by the text in string `S`. For example `val(''123.45'')` is 123.45 as a float.

`vdot(V1,V2)` Dot product of `V1` and `V2`. Returns a float value that is the dot product (sometimes called scalar product) of `V1` with `V2`. It is directly proportional to the length of the two vectors and the cosine of the angle between them. Formula is *vdot=V1.x*V2.x + V1.y*V2.y + V1.z*V2.z.* See the animated demo scene `VECT2.POV` for an illustration.

`vlength(V)` Length of `V`. Returns a float value that is the length of vector `V`. Formula is *vlength=sqrt(vdot(A,A))*. Can be used to compute the distance between two points. `Dist=vlength(V2-V1)`.

See section "Vector Functions" and section "String Functions" for other functions which are somewhat float-related but which return vectors and strings. In addition to the above built-in functions, you may also define your own functions using the `#macro` directive. See the section "User Defined Macros" for more details.

Built-in Constants

Constants are:

```
FLOAT_BUILT-IN_IDENT:
    false | no | off | on | pi | true | yes
```

The built-in constants never change value. They are defined as though the following lines were at the start of every scene.

```
#declare pi = 3.1415926535897932384626;
#declare true = 1;
#declare yes = 1;
#declare on = 1;
#declare false = 0;
#declare no = 0;
#declare off = 0;
```

The built-in float identifier `pi` is obviously useful in math expressions involving circles. The built-in float identifiers `on`, `off`, `yes`, `no`, `true`, and `false` are designed for use as boolean constants.

The built-in float constats `on`, `off`, `yes`, `no`, `true`, and `false` are most often used as boolean values with object modifiers or parameters such as `sturm`, `hollow`, `hierarchy`, `smooth`, `media_attenuation`, and `media_interaction`. Whenever you see syntax of the form `keyword` *[Bool]*, if you simply specify the keyword without the optional boolean then it assumes `keyword on`. You need not use the boolean but for readability it is a good idea. You must use one of the false booleans or an expression which evaluates to zero to turn it off.

Note: some of these keywords are `on` by default, if no keyword is specified.

For example:

```
object { MyBlob }               // sturm defaults off, but
                                // hierarchy defaults on
object { MyBlob sturm }         // turn sturm on
object { MyBlob sturm on }      // turn sturm on
object { MyBlob sturm off }     // turn sturm off
object { MyBlob hierarchy }     // does nothing, hierarchy was
                                // already on
object { MyBlob hierarchy off } // turn hierarchy off
```

Built-in Variables

There are several built-in float variables. You can use them to specify values or to create expressions but you cannot re-declare them to change their values.

Clock-related are:

```
FLOAT_BUILT-IN_IDENT:
   clock | clock_delta | clock_on | final_clock | final_frame
    frame_number | initial_clock | initial_frame
```

These keywords allow to use the values of the clock which have been set in the command line switch options (or INI-file). They represent float or integer values, read from the animation options. You cannot re-declare these identifiers.

`clock`
The built-in float identifier `clock` is used to control animations in POV-Ray. Unlike some animation packages, the action in POV-Ray animated scenes does not depend upon the integer frame numbers. Rather you should design your scenes based upon the float identifier `clock`. For non-animated scenes its default value is 0 but you can set it to any float value using the INI file option `Clock=`*n.n* or the command-line switch `+K`*n.n* to pass a single float value your scene file.

Other INI options and switches may be used to animate scenes by automatically looping through the rendering of frames using various values for `clock`. By default, the clock value is 0 for the initial frame and 1 for the final frame. All other frames are interpolated between these values.
For example if your object is supposed to rotate one full turn over the course of the animation you could specify `rotate 360*clock*y`. Then as clock runs from 0 to 1, the object rotates about the y-axis from 0 to 360 degrees.

Although the value of `clock` will change from frame-to-frame, it will never change throughout the parsing of a scene.

`clock_delta`
The built-in float identifier `clock_delta` returns the amount of time between clock values in animations in POV-Ray. While most animations only need the clock value itself, some animation calculations are easier if you know how long since the last frame. Caution must be used when designing such scenes. If you render a scene with too few frames, the results may be different than if you render with more frames in a given time

period. On non-animated scenes, `clock_delta` defaults to 1.0. See section "Animation Options" for more details.

clock_on
With this identifier the status of the clock can be checked: 1 is on, 0 is off.

```
#if(clock_on=0)
  //stuff for still image
#else
  //some animation
#end
```

frame_number
If you rather want to define the action in POV-Ray animated scenes depending upon the integer frame numbers, this identifier can be used.
It reads the number of the frame currently being rendered.

```
#if(frame_number=1)
  //stuff for first image or frame
#end
#if(frame_number=2)
  //stuff for second image or frame
#end
#if(frame_number=n)
  //stuff for n th image or frame
#end
```

initial_clock
This identifier reads the value set through the INI file option `Initial_Clock=`*n.n* or the command-line switch `+KI`*n.n*.

final_clock
This identifier reads the value set through the INI file option `Final_Clock=`*n.n* or the command-line switch `+KF`*n.n*.

initial_frame
This identifier reads the value set through the INI file option `Initial_Frame=`*n* or the command-line switch `+KFI`*n*.

final_frame
This identifier reads the value set through the INI file option `Final_Frame=`*n* or the command-line switch `+KFF`*n*.

Note: that these values are the ones actually used. When the option 'cyclic animation' is set, they could be different from the ones originally set in the options.

Image-size are:

```
FLOAT_BUILT-IN_IDENT:
     image_width | image_height
```

image_width
This identifier reads the value set through the INI file option `Width=`*n* or the command-line switch `+W`*n*.

image_height

This identifier reads the value set through the INI file option `Height=`*n* or the command-line switch `+H`*n*.

You could use these keywords to set the camera ratio (up and right vectors) correctly. The viewing angle of the camera covers the full width of the rendered image. The camera ratio will always follow the ratio of the image width to height, regardless of the set image size. Use it like this:

```
up y*image_height
right x*image_width
```

You could also make some items of the scene dependent on the image size:

```
#if (image_width < 300) crand 0.1 #else crand 0.5 #end
```

or:

```
image_map {
  pattern image_width, image_width {  //make pattern resolution
    gradient x                        //dependent of render width
    color_map { [ 0.0 ... ] [ 1.0 ... ] }
  }
}
```

Version is:

```
FLOAT_BUILT-IN_IDENT:
     version
```

The built-in float variable `version` contains the current setting of the version compatibility option. Although this value defaults to the current POV-Ray version number, the initial value of `version` may be set by the INI file option `Version=`*n.n* or by the `+MV`*n.n* command-line switch. This tells POV-Ray to parse the scene file using syntax from an earlier version of POV-Ray.

The INI option or switch only affects the initial setting. Unlike other built-in identifiers, you may change the value of `version` throughout a scene file. You do not use `#declare` to change it though. The `#version` language directive is used to change modes. Such changes may occur several times within scene files.

Together with the built-in `version` identifier the `#version` directive allows you to save and restore the previous values of this compatibility setting. The new `#local` identifier option is especially useful here. For example suppose `mystuff.inc` is in version 1 format. At the top of the file you could put:

```
#local Temp_Vers = version;  // Save previous value
#version 1.0;                // Change to 1.0 mode
...                          // Version 1.0 stuff goes here...
#version Temp_Vers;          // Restore previous version
```

Note: there should be a semi-colon after the float expression in a `#version` directive. If omitted, it generates a warning and some macros may not work properly.

1.1.4 Vector Expressions

POV-Ray often requires you to specify a *vector*. A vector is a set of related float values. Vectors may be specified using literals, identifiers or functions which return vector values. You may also create very complex vector expressions from combinations of any of these using various familiar operators.

POV-Ray vectors may have from two to five components but the vast majority of vectors have three components. Unless specified otherwise, you should assume that the word "vector" means a three component vector. POV-Ray operates in a 3D x, y, z coordinate system and you will use three component vectors to specify x, y and z values. In some places POV-Ray needs only two coordinates. These are often specified by a 2D vector called an *UV vector*. Fractal objects use 4D vectors. Color expressions use 5D vectors but allow you to specify 3, 4 or 5 components and use default values for the unspecified components. Unless otherwise noted, all 2, 4 or 5 component vectors work just like 3D vectors but they have a different number of components.

The syntax for combining vector literals into vector expressions is almost identical to the rules for float expressions. In the syntax for vector expressions below, some of the syntax items are defined in the section for float expressions. See "Float Expressions" for those definitions. Detailed explanations of vector-specific issues are given in the following sub-sections.

```
VECTOR:
    NUMERIC_TERM [SIGN NUMERIC_TERM]
NUMERIC_TERM:
    NUMERIC_FACTOR [MULT NUMERIC_FACTOR]
NUMERIC_FACTOR:
    VECTOR_LITERAL        |
    VECTOR_IDENTIFIER     |
    SIGN NUMERIC_FACTOR   |
    VECTOR_FUNCTION       |
    VECTOR_BUILT-IN_IDENT |
    ( FULL_EXPRESSION )   |
    ! NUMERIC_FACTOR      |
    FLOAT
VECTOR_LITERAL:
    < FLOAT , FLOAT , FLOAT >
VECTOR_FUNCTION:
    min_extent ( OBJECT_IDENTIFIER )         |
    max_extent ( OBJECT_IDENTIFIER )         |
    trace(OBJECT_IDENTIFIER, VECTOR, VECTOR, [VECTOR_IDENTIFIER] )|
    vaxis_rotate( VECTOR , VECTOR , FLOAT )  |
    vcross( VECTOR , VECTOR )                |
    vrotate( VECTOR , VECTOR )               |
    vnormalize( VECTOR )                     |
    vturbulence(FLOAT, FLOAT, FLOAT, VECTOR)
VECTOR_BUILT-IN_IDENT:
     x | y | z | t | u | v
```

Note: *VECTOR_IDENTIFIERS* are identifiers previously declared to have vector values.

Literals

Vector literals consist of two to five float expressions that are bracketed by angle brackets < and >. The terms are separated by commas. For example here is a typical three component vector:

```
< 1.0, 3.2, -5.4578 >
```

The commas between components are necessary to keep the program from thinking that the 2nd term is the single float expression `3.2-5.4578` and that there is no 3rd term. If you see an error message such as "Float expected but '>' found instead" then you probably have missed a comma.

Sometimes POV-Ray requires you to specify floats and vectors side-by-side. The rules for vector expressions allow for mixing of vectors with vectors or vectors with floats so commas are required separators whenever an ambiguity might arise. For example `<1,2,3>-4` evaluates as a mixed float and vector expression where 4 is subtracted from each component resulting in `<-3,-2,-1>`. However the comma in `<1,2,3>,-4` means this is a vector followed by a float.

Each component may be a full float expression. For example `<This+3,That/3,5*Other_Thing>` is a valid vector.

Identifiers

Vector identifiers may be declared to make scene files more readable and to parameterize scenes so that changing a single declaration changes many values. An identifier is declared as follows.

```
VECTOR_DECLARATION:
    #declare IDENTIFIER = EXPRESSION; |
    #local IDENTIFIER = EXPRESSION;
```

Where *IDENTIFIER* is the name of the identifier up to 40 characters long and *EXPRESSION* is any valid expression which evaluates to a vector value.

Note: there should be a semi-colon after the expression in a vector declaration. If omitted, it generates a warning and some macros may not work properly. See " #declare vs. #local" for information on identifier scope.

Here are some examples....

```
#declare Here = <1,2,3>;
#declare There = <3,4,5>;
#declare Jump = <Foo*2,Bar-1,Bob/3>;
#declare Route = There-Here;
#declare Jump = Jump+<1,2,3>;
```

Note: you invoke a vector identifier by using its name without any angle brackets. As the last example shows, you can re-declare a vector identifier and may use previously declared values in that re-declaration. There are several built-in identifiers which POV-Ray declares for you. See section "Built-in Vector Identifiers" for details.

Operators

Vector literals, identifiers and functions may also be combined in expressions the same as float values. Operations are performed on a component-by-component basis. For example `<1,2,3> + <4,5,6>` evaluates the same as `<1+4,2+5,3+6>` or `<5,7,9>`. Other operations are done on a similar component-by-component basis. For example `(<1,2,3> = <3,2,1>)` evaluates to `<0,1,0>` because the middle components are equal but the others are not. Admittedly this isn't very useful but it's consistent with other vector operations.

Conditional expressions such as `(C ? A : B)` require that `C` is a float expression but `A` and `B` may be vector expressions. The result is that the entire conditional evaluates as a valid vector. For example if `Foo` and `Bar` are floats then `(Foo < Bar ? <1,2,3> : <5,6,7>)` evaluates as the vector `<1,2,3>` if `Foo` is less than `Bar` and evaluates as `<5,6,7>` otherwise.

You may use the dot operator to extract a single float component from a vector. Suppose the identifier `Spot` was previously defined as a vector. Then `Spot.x` is a float value that is the first component of this x, y, z vector. Similarly `Spot.y` and `Spot.z` reference the 2nd and 3rd components. If `Spot` was a two component UV vector you could use `Spot.u` and `Spot.v` to extract the first and second component. For a 4D vector use `.x`, `.y`, `.z`, and `.t` to extract each float component. The dot operator is also used in color expressions which are covered later.

Operator Promotion

You may use a lone float expression to define a vector whose components are all the same. POV-Ray knows when it needs a vector of a particular type and will promote a float into a vector if need be. For example the POV-Ray `scale` statement requires a three component vector. If you specify `scale 5` then POV-Ray interprets this as `scale <5,5,5>` which means you want to scale by 5 in every direction.

Versions of POV-Ray prior to 3.0 only allowed such use of a float as a vector in various limited places such as `scale` and `turbulence`. However you may now use this trick anywhere. For example...

```
box{0,1}  // Same as box{<0,0,0>,<1,1,1>}
sphere{0,1} // Same as sphere{<0,0,0>,1}
```

When promoting a float into a vector of 2, 3, 4 or 5 components, all components are set to the float value, however when promoting a vector of a lower number of components into a higher order vector, all remaining components are set to zero. For example if POV-Ray expects a 4D vector and you specify `9` the result is `<9,9,9,9>` but if you specify `<7,6>` the result is `<7,6,0,0>`.

Functions

POV-Ray defines a variety of built-in functions for manipulating floats, vectors and strings. Function calls consist of a keyword which specifies the name of the function followed by a parameter list enclosed in parentheses. Parameters are separated by commas. For example:

```
keyword(param1,param2)
```

The following are the functions which return vector values. They take one or more float, integer, vector, or string parameters. Assume that `A` and `B` are any valid expression that evaluates to a vector; and `F` is any float expression.

`min_extent ( OBJECT_IDENTIFIER ), max_extent ( OBJECT_IDENTIFIER )`. The `min_extent` and `max_extent` return the minimum and maximum coordinates of a #declared object's bounding box (Corner1 and Corner2), in effect allowing you to find the dimensions and location of the object.

Note: this is not perfect, in some cases (such as CSG intersections and differences or isosurfaces) the bounding box does not represent the actual dimensions of the object.

Example:

```
#declare Sphere =
sphere {
  <0,0,0>, 1
  pigment { rgb <1,0,0> }
}
#declare Min = min_extent ( Sphere );
#declare Max = max_extent ( Sphere );
object { Sphere }
box {
    Min, Max
    pigment { rgbf <1,1,1,0.5> }
}
```

`trace(OBJECT_IDENTIFIER, A, B, [VECTOR_IDENTIFIER])`. `trace` helps you finding the exact location of a ray intersecting with an object's surface. It traces a ray beginning at the point `A` in the direction specified by the vector `B`. If the ray hits the specified object, this function returns the coordinate where the ray intersected the object. If not, it returns `<0,0,0>`. If a fourth parameter in the form of a vector identifier is provided, the normal of the object at the intersection point (not including any normal perturbations due to textures) is stored into that vector. If no intersection was found, the normal vector is reset to `<0,0,0>`.

Note: Checking the normal vector for `<0,0,0>` is the only reliable way to determine whether an intersection has actually occurred, intersections can and do occur anywhere, including at `<0,0,0>`.

Example:

```
#declare MySphere = sphere { <0, 0, 0>, 1 }
#declare Norm = <0, 0, 0>;
#declare Start = <1, 1, 1>;
#declare Inter=
  trace ( MySphere, Start, <0, 0, 0>-Start, Norm );
  object {
    MySphere
    texture {
      pigment { rgb 1}
    }
  }
  #if (vlength(Norm)!=0)
```

```
  cylinder {
    Inter, Inter+Norm, .1
    texture {
      pigment {color red 1}
    }
  }
#end
```

`vaxis_rotate(A,B,F)` Rotate `A` about `B` by `F`. Given the x,y,z coordinates of a point in space designated by the vector `A`, rotate that point about an arbitrary axis defined by the vector `B`. Rotate it through an angle specified in degrees by the float value `F`. The result is a vector containing the new x,y,z coordinates of the point.

`vcross(A,B)` Cross product of `A` and `B`. Returns a vector that is the vector cross product of the two vectors. The resulting vector is perpendicular to the two original vectors and its length is equal to the area of the parallelogram defined by them. Or to put in an other way, the cross product can also be formulated as: *`AxB = |A| * |B| * sin(angle(A,B)) * perpendicular_unit_vector(A,B)`* So the length of the resulting vector is proportional to the sine of the angle between `A` and `B`. See the animated demo scene `VECT2.POV` for an illustration.

`vnormalize(A)` Normalize vector `A`. Returns a unit length vector that is the same direction as `A`. Formula is *vnormalize(A)=A/vlength(A)*.

Note:`vnormalize(<0,0,0>)` will result in an error.

`vrotate(A,B)` Rotate `A` about origin by `B`. Given the x,y,z coordinates of a point in space designated by the vector `A`, rotate that point about the origin by an amount specified by the vector `B`. Rotate it about the x-axis by an angle specified in degrees by the float value `B.x`. Similarly `B.y` and `B.z` specify the amount to rotate in degrees about the y-axis and z-axis. The result is a vector containing the new x,y,z coordinates of the point.

`vturbulence(Lambda, Omega, Octaves, A)` Turbulence vector at A. Given the x,y,z coordinates of a point in space designated by the vector A, return the turbulence vector for that point based on the numbers given for Lambda, Omega and Octaves. For the meaning of the parameters, check out the Lambda, Omega and Octaves sections.
The amount of turbulence can be controlled by multiplying the turbulence vector by a multiple. The frequency at which the turbulence vector changes can be controlled by multiplying A with a multiple. The turbulence vector returned by the function can be added to the original point A to obtain a turbulated version of the point A. Example :

```
#declare MyVector = MyVector + Amount*vturbulence( 2, 0.5, 6,MyVector*Frequency );
```

See section “Float Functions” for other functions which are somewhat vector-related but which return floats. In addition to the above built-in functions, you may also define your own functions using the `#macro` directive. See the section “User Defined Macros” for more details.

Built-in Constants

There are several built-in vector identifiers. You can use them to specify values or to create expressions but you cannot re-declare them to change their values. They are:

```
VECTOR_BUILT-IN_IDENT:
    x | y | z | t | u | v
```

All built-in vector identifiers never change value. They are defined as though the following lines were at the start of every scene.

```
#declare x = <1, 0, 0>;
#declare y = <0, 1, 0>;
#declare z = <0, 0, 1>;
#declare t = <0, 0, 0, 1>;
#declare u = <1, 0>;
#declare v = <0, 1>;
```

The built-in vector identifiers `x`, `y`, and `z` provide much greater readability for your scene files when used in vector expressions. For example....

```
plane { y, 1}    // The normal vector is obviously "y".
plane { <0,1,0>, 1} // This is harder to read.
translate 5*x    // Move 5 units in the "x" direction.
translate <5,0,0>  // This is less obvious.
```

An expression like `5*x` evaluates to `5*<1,0,0>` or `<5,0,0>`.

Similarly `u` and `v` may be used in 2D vectors. When using 4D vectors you should use `x`, `y`, `z`, and `t` and POV-Ray will promote `x`, `y`, and `z` to 4D when used where 4D is required.

1.1.5 Specifying Colors

```
COLOR:
    COLOR_BODY           |
    color COLOR_BODY     | <em>(this means the keyword color or</em>
    colour COLOR_BODY      <em> colour may optionally precede</em>
                           <em> any color specification)</em>
COLOR_BODY:
    COLOR_VECTOR         |
    COLOR_KEYWORD_GROUP  |
    COLOR_IDENTIFIER
COLOR_VECTOR:
    rgb <3_Term_Vector>  |
    rgbf <4_Term_Vector> |
    rgbt <4_Term_Vector> |
    [ rgbft ] <5_Term_Vector>
COLOR_KEYWORD_GROUP:
    [ COLOR_KEYWORD_ITEM ]...
COLOR_KEYWORD_ITEM:
    COLOR_IDENTIFIER       |
    red Red_Amount         |
    blue Blue_Amount       |
    green Green_Amount     |
    filter Filter_Amount   |
    transmit Transmit_Amount
```

Note: *COLOR_IDENTIFIERS* are identifiers previously declared to have color values. The 3, 4, and 5 term vectors are usually vector literals but may be vector expressions

or floats promoted to vectors. See “Operator Promotion” and the sections below.

POV-Ray often requires you to specify a color. Colors consist of five values or color components. The first three are called `red`, `green`, and `blue`. They specify the intensity of the primary colors red, green and blue using an additive color system like the one used by the red, green and blue color phosphors on a color monitor.

The 4th component, called `filter`, specifies the amount of filtered transparency of a substance. Some real-world examples of filtered transparency are stained glass windows or tinted cellophane. The light passing through such objects is tinted by the appropriate color as the material selectively absorbs some frequencies of light while allowing others to pass through. The color of the object is subtracted from the light passing through so this is called subtractive transparency.

The 5th component, called `transmit`, specifies the amount of non-filtered light that is transmitted through a surface. Some real-world examples of non-filtered transparency are thin see-through cloth, fine mesh netting and dust on a surface. In these examples, all frequencies of light are allowed to pass through tiny holes in the surface. Although the amount of light passing through is diminished, the color of the light passing through is unchanged.

The color of the object and the color transmitted through the object together contribute 100% of the final color. So if `transmit` is set to 0.9, the transmitted color contributes 90% and the color of the object contributes only 10%. This is also true outside of the 0-1 range, so for example if `transmit` is set to 1.7, the transmitted color contributes with 170% and the color of the object contributes with minus 70%. Using `transmit` values outside of the 0-1 range can be used to create interesting special effects, but does not correspond to any phenomena seen in the real world. An example:

```
#version 3.5;
global_settings {assumed_gamma 1.0}
camera {location -2.5*z look_at 0 orthographic}
box {
   0,1
   texture {
      pigment {
         gradient y
         colour_map {
            [0, red 1]
            [1, blue 1]
         }
      }
      finish{ambient 1}
   }
   texture {
      pigment {
         gradient x
         colour_map {
            [0, rgb 0.5 transmit -3]
            [1, rgb 0.5 transmit  3]
         }
      }
      finish{ambient 1}
   }
```

```
   translate <-0.5,-0.5,0>
   scale <3,2,1>
}
```

When using the `transmit` value for special effects, you can visualize it this way: The `transmit` value means “contrast”. 1.0 is no change in contrast, 0.5 is half contrast, 2.0 is double contrast and so on. You could say that `transmit` “scales” the colors. The color of the object is the “center value”. All colors will get closer to the “center value” if `transmit` is between 0 and 1, and all colors will spread away from the “center value” if `transmit` is greater than 1. If `transmit` is negative the colors will be inverted around the “center value”. Rgb 0.5 is common to use as “center value”, but other values can be used for other effects. The “center value” really is a color, and non-gray colors can be used for interesting effects. The red, green and blue components are handled separately.

Note: early versions of POV-Ray used the keyword `alpha` to specify filtered transparency. However that word is often used to describe non-filtered transparency. For this reason `alpha` is no longer used.

Each of the five components of a color are float values which are normally in the range between 0.0 and 1.0. However any values, even negatives may be used.

Under most circumstances the keyword `color` is optional and may be omitted. We also support the British or Canadian spelling `colour`. Colors may be specified using vectors, keywords with floats or identifiers. You may also create very complex color expressions from combinations of any of these using various familiar operators. The syntax for specifying a color has evolved since POV-Ray was first released. We have maintained the original keyword-based syntax and added a short-cut vector notation. Either the old or new syntax is acceptable however the vector syntax is easier to use when creating color expressions.

The syntax for combining color literals into color expressions is almost identical to the rules for vector and float expressions. In the syntax for vector expressions, some of the syntax items are defined in the section for float expressions. See “Float Expressions” for those definitions. Detailed explanations of color-specific issues are given in the following sub-sections.

Color Vectors

The syntax for a color vector is...

```
COLOR_VECTOR:
    rgb <3_Term_Vector>   |
    rgbf <4_Term_Vector>  |
    rgbt <4_Term_Vector>  |
    [ rgbft ] <5_Term_Vector>
```

...where the vectors are any valid vector expressions of 3, 4 or 5 components. For example

```
 color rgb <1.0, 0.5, 0.2>
```

This specifies a color whose red component is 1.0 or 100% of full intensity. The green component is 0.5 or 50% of full intensity and the blue component is 0.2 or 20% of full

intensity. Although the filter and transmit components are not explicitly specified, they exist and are set to their default values of 0 or no transparency.

The `rgbf` keyword requires a four component vector. The 4th component is the filter component and the transmit component defaults to zero. Similarly the `rgbt` keyword requires four components where the 4th value is moved to the 5th component which is transmit and then the filter component is set to zero.

The `rgbft` keyword allows you to specify all five components. Internally in expressions all five are always used.

Under some circumstances, if the vector expression is a 5 component expression or there is a color identifier in the expression then the `rgbtf` keyword is optional.

Color Keywords

The older keyword method of specifying a color is still useful and many users prefer it.

```
COLOR_KEYWORD_GROUP:
    [ COLOR_KEYWORD_ITEM ]...
COLOR_KEYWORD_ITEM:
    COLOR_IDENTIFIER |
    red Red_Amount | blue Blue_Amount | green Green_Amount |
    filter Filter_Amount | transmit Transmit_Amount
```

Although the `color` keyword at the beginning is optional, it is more common to see it in this usage. This is followed by any of five additional keywords `red`, `green`, `blue`, `filter`, or `transmit`. Each of these component keywords is followed by a float expression. For example

```
color red 1.0 green 0.5
```

This specifies a color whose red component is 1.0 or 100% of full intensity and the green component is 0.5 or 50% of full intensity. Although the blue, filter and transmit components are not explicitly specified, they exist and are set to their default values of 0. The component keywords may be given in any order and if any component is unspecified its value defaults to zero. A *COLOR_IDENTIFIER* can also be specified but it should always be first in the group. See “Common Color Pitfalls” for details.

Color Identifiers

Color identifiers may be declared to make scene files more readable and to parameterize scenes so that changing a single declaration changes many values. An identifier is declared as follows.

```
COLOR_DECLARATION:
    #declare IDENTIFIER = COLOR; |
    #local IDENTIFIER = COLOR;
```

Where *IDENTIFIER* is the name of the identifier up to 40 characters long and *COLOR* is any valid specification.

Note: there should be a semi-colon at the end of the declaration. If omitted, it generates a warning and some macros may not work properly. See " #declare vs. #local" for information on identifier scope.

Here are some examples....

```
#declare White = rgb <1,1,1>;
#declare Cyan = color blue 1.0 green 1.0;
#declare Weird = rgb <Foo*2,Bar-1,Bob/3>;
#declare LightGray = White*0.8;
#declare LightCyan = Cyan red 0.6;
```

As the `LightGray` example shows you do not need any color keywords when creating color expressions based on previously declared colors. The last example shows you may use a color identifier with the keyword style syntax. Make sure that the identifier comes first before any other component keywords.

Like floats and vectors, you may re-define colors throughout a scene but the need to do so is rare.

Color Operators

Color vectors may be combined in expressions the same as float or vector values. Operations are performed on a component-by-component basis. For example `rgb <1.0,0.5,0.2>*0.9` evaluates the same as `rgb<1.0,0.5,0.2>*<0.9,0.9,0.9>` or `rgb<0.9,0.45,0.18>`. Other operations are done on a similar component-by-component basis.

You may use the dot operator to extract a single component from a color. Suppose the identifier `Shade` was previously defined as a color. Then `Shade.red` is the float value of the red component of `Shade`. Similarly `Shade.green`, `Shade.blue`, `Shade.filter` and `Shade.transmit` extract the float value of the other color components. `shade.gray` returns the gray value of the color vector.

Common Color Pitfalls

The variety and complexity of color specification methods can lead to some common mistakes. Here are some things to consider when specifying a color.

When using filter transparency, the colors which come through are multiplied by the primary color components. For example if gray light such as `rgb<0.9,0.9,0.9>` passes through a filter such as `rgbf<1.0,0.5,0.0,1.0>` the result is `rgb<0.9,0.45,0.0>` with the red let through 100%, the green cut in half from 0.9 to 0.45 and the blue totally blocked. Often users mistakenly specify a clear object by

```
color filter 1.0
```

but this has implied red, green and blue values of zero. You've just specified a totally black filter so no light passes through. The correct way is either

```
color red 1.0  green 1.0  blue 1.0  filter 1.0
```

or

```
color transmit 1.0
```

In the 2nd example it doesn't matter what the rgb values are. All of the light passes through untouched. Another pitfall is the use of color identifiers and expressions with color keywords. For example...

```
color My_Color red 0.5
```

this substitutes whatever was the red component of `My_Color` with a red component of 0.5 however...

```
color My_Color + red 0.5
```

adds 0.5 to the red component of `My_Color` and even less obvious...

```
color My_Color * red 0.5
```

that cuts the red component in half as you would expect but it also multiplies the green, blue, filter and transmit components by zero! The part of the expression after the multiply operator evaluates to `rgbft<0.5,0,0,0,0>` as a full 5 component color.

The following example results in no change to `My_Color`.

```
color red 0.5 My_Color
```

This is because the identifier fully overwrites the previous value. When using identifiers with color keywords, the identifier should be first. Another issue to consider: some POV-Ray syntax allows full color specifications but only uses the rgb part. In these cases it is legal to use a float where a color is needed. For example:

```
finish { ambient 1 }
```

The ambient keyword expects a color so the value `1` is promoted to `<1,1,1,1,1>` which is no problem. However

```
pigment { color 0.4 }
```

is legal but it may or may not be what you intended. The `0.4` is promoted to `<0.4,0.4,0.4,0.4,0.4>` with the filter and transmit set to 0.4 as well. It is more likely you wanted...

```
pigment { color rgb 0.4 }
```

in which case a 3 component vector is expected. Therefore the `0.4` is promoted to `<0.4,0.4,0.4,0.0,0.0>` with default zero for filter and transmit. Finally there is another problem which arises when using color dot operators in `#declare` or `#local` directives. Consider the directive:

```
#declare MyColor = rgb <0.75, 0.5, 0.75>;
#declare RedAmt = MyColor.red;
```

Now `RedAmt` should be a float but unfortunately it is a color. POV-Ray looks at the first keyword after the equals to try to guess what type of identifier you want. It sees the color identifier `MyColor` and assumes you want to declare a color. It then computes the float value as 0.75 then promotes that into `rgbft<0.75,0.75,0.75,0.75,0.75>`. It would take a major rewrite to fix this problem so we're just warning you about it. Any of the following work-arounds will work properly.

```
#declare RedAmt = 0.0+MyColor.red;
#declare RedAmt = 1.0*MyColor.red;
#declare RedAmt = (MyColor.red);
```

1.1.6 User-Defined Functions

Some objects allow you to specify functions that will be evaluated while rendering to determine the surface of these objects. In this respect functions are quite different to macros, which are evaluated at parse time but do not otherwise affect rendering. Additionally you may call these functions anywhere a Float Function is allowed, even during parsing. The syntax is identical to Float Expressions, however, only float functions that apply to float values may be used. Excluded are for example `strlen` or `vlength`. You find a full list of supported float functions in the syntax definition below.

```
FLOAT:
    LOGIC_AND [OR LOGIC_AND]
OR:
    |
LOGIC_AND:
    REL_TERM [AND REL_TERM]
AND:
    &
REL_TERM:
    TERM [REL_OPERATOR TERM]
REL_OPERATOR:
    < | <= | >= | > | = | !=
TERM:
    FACTOR [SIGN FACTOR]
SIGN:
    + | -
FACTOR:
    MOD_EXPRESSION [MULT MOD_EXPRESSION]
MULT:
    * | /
EXPRESSION:
    FLOAT_LITERAL          |
    FLOAT_IDENTIFIER       |
    FLOAT_FUNCTION         |
    FLOAT_BUILT-IN_IDENT   |
    FUNCTION_IDENTIFIER    |
    ( FLOAT )              |
    IDENTIFIER             |
    SIGN EXPRESSION
FLOAT_FUNCTION:
    abs( FLOAT ) | acos( FLOAT ) | acosh( FLOAT ) | asin( FLOAT ) |
    asinh( FLOAT ) | atan( FLOAT) | atanh( FLOAT) |
    atan2( FLOAT , FLOAT ) | ceil( FLOAT ) | cos( FLOAT ) |
    cosh( FLOAT ) | degrees( FLOAT ) | exp( FLOAT ) |
    floor( FLOAT ) | int( FLOAT ) | ln (Float) | log( FLOAT ) |
    max( FLOAT , FLOAT, ... ) | min( FLOAT , FLOAT, ... ) |
    mod( FLOAT , FLOAT ) | pow( FLOAT , FLOAT ) |
    radians( FLOAT ) | sin( FLOAT ) | sinh( FLOAT ) |
    sqrt( FLOAT ) | tan( FLOAT ) | tanh( FLOAT ) |
    select( FLOAT , FLOAT , FLOAT [, FLOAT] )
FUNCTION_IDENTIFIER:
    #local FUNCTION_IDENTIFIER = function { FLOAT }                |
    #declare FUNCTION_IDENTIFIER = function { FLOAT }              |
    #local FUNCTION_IDENTIFIER = function(IDENT_LIST) { FLOAT }    |
```

```
    #declare FUNCTION_IDENTIFIER = function(IDENT_LIST) { FLOAT } |
    #local FUNCTION_IDENTIFIER = function{SPECIAL_FLOAT_FUNCTION} |
    #local VECTOR_IDENTIFIER = function{SPECIAL_VECTOR_FUNCTION}  |
    #local COLOR_IDENTIFIER = function { SPECIAL_COLOR_FUNCTION } |
IDENT_LIST:
    IDENT_ITEM [, IDENT_LIST]
IDENT_ITEM:
    x | y | z | u | v | IDENTIFIER
    <em>(Note: x = u and y = v)</em>
SPECIAL_FLOAT_FUNCTION:
    pattern { PATTERN_BLOCK }
SPECIAL_VECTOR_FUNCTION:
    TRANSFORMATION_BLOCK | SPLINE
SPECIAL_COLOR_FUNCTION:
    PIGMENT
PATTERN_BLOCK:
    PATTERN
```

Note: Only the above mentioned items can be used in user-defined functions. For example the rand() function is not available.

All of the above mentioned float functions are described in the section Float Functions.

Sum and Product functions

`prod(i, b, n, a)` The product function.

$$\prod_{i=b}^{n} a$$

Equation 1.1: product function

`sum(i, b, n, a)` The sum function.

$$\sum_{i=b}^{n} a$$

Equation 1.2: sum function

For both `prod` and `sum`: `i` is any variable name and `a` is any expression, usually depending on `i`. `b` and `n` are also any expression.
Example:

```
  #declare factorial = function(C) { prod(i, 1, C, i) }
  #declare A = factorial(5);
```

The first parameter is the name of the iteration variable. The second is the initial value expression and the third is the final value expression. Those may not depend on the iteration variable but the iteration variable may still be used inside those two

expressions (because it happens to already have been defined) but its value is undefined. The last expression is the actual expression which will be iterated through. It may use any variable in scope.

The scope of an iteration variable is the sequence operation function. That is, a iteration variable is only defined when used inside the `sum/prod` function. Of course `sum/prod` functions may be nested. However, there is one limit of a maximum of 56 local variable defined simultaneously, which essentially means that in any combination `sum/prod` functions cannot be nested deeper than 56 levels.

The iteration variable is incremented by one for each step, but its initial and final value may be any value. The iteration will be continued as long as the iteration value is less or equal to the final value.

Note: because the iteration value is a floating-point variable, adding one will add a certain bias in a long iterations and thus the floating-point precision will be an issue in such a case and needs to be considered by allowing a reasonable error for the final value!

If the expression to be added has a negative sign it will of course in effect be substracted. Thus changing the sign will allow to generate negative values in the sum function. Equally multiplying by `1/expression` effectively creates a division when used in the prod function.

Obviously to work in the first place the initial value of the result is the neutral element of the operation. That is, a sum calculation starts with `0` and a product calculation starts with `1` just like it is assumed in the sum and product functions in 'regular' math.

It should be noted that mathematically either sum or product are redundant because:

```
log10(prod(i, b, n, a)) = sum(i, b, n, log10(a))
```

Functions and Macros

You can use macros in functions, but the macros will be called only once when the function is defined, not every time the function is called. You cannot pass function variables to the macros.

You can pass functions to macros, how to do this is best explained by an example:

```
#macro Foo( Bar, X )
  #declare Y = Bar(X);
  #declare Z = Bar(Y);
#end

#declare FUNC=function(n){n+2}

Foo(FUNC, 1)

#debug str(Y,5,5)
#debug "\n"
#debug str(Z,5,5)
#debug "\n"
```

Declaring User-Defined Float Functions

You declare a user defined function using the `#declare` or `#local` directives. By default a function takes three parameters and you do not have to explicitly specify the parameter names. The default three parameters are `x`, `y` and `z`. For example:

```
#declare foo = function { x + y * z }
```

If you need fewer or more parameters you have to explicitly specify the parameter list.

Note: `x` and `u` as well as `y` and `v` are equivalent so you may not specify both parameter names. You may not specify two or more parameters with the same name either. Doing so may result in a parse error or undefined function results.

The following are valid functions with parameters:

```
#declare foo2 = function(x, y, z) { x + y * z }
#declare foo3 = function(k1, k2, z, y) { x + y * z + k1 * y + k2 }
#declare foo4 = function(h) { h * h + h }
#declare foo4 = function(u, v) { x + y * v } //=u + v*v
#declare foo4 = function(x, v, z) { u + y * v + z } //=x + v*v + z
```

Limits:

- The minimum number of parameters per function is 1.
- The maximum number of allowed parameters per function is 56.
- The maximum number of `function` blocks per scene is 1048575.
- The maximum number of operators per function is about 200000. Individual limits will be different depending on the types of operators used in the function.
- The maximum depth for nesting functions is 1024.
- The maximum number of constants in all functions 1048575.

Note: Redeclaring functions, directly, is not allowed. The way to do this is to `undef` it first.

There is one special float function type. You may declare a `pattern` function.

Note: the syntax is identical to that of patterns, however, you may not specify colors. Its result is always a float and not a color vector, as returned by a function containing a pigment.

```
#declare foo = function {
  pattern {
    checker
  }
}
```

Note: the number of parameters of special function types is determined automatically, so you do not need to specify parameter names.

Declaring User-Defined Vector Functions

Right now you may only declare vector functions using one of the special function types. Supported types are `transform` and `spline` functions. For example:

```
#declare foo = function {
  transform {
    rotate <90, 0, 0>
    scale 4
  }
}

#declare myvector = foo(4, 3, 7);

#declare foo2 = function {
  spline {
    linear_spline
    0.0, <0,0,0>
    0.5, <1,0,0>
    1.0, <0,0,0>
  }
}

#declare myvector2 = foo2(0.7);
```

Function splines take the vector size into account. That is, a function containing a spline with five components will also return a five component vector (aka a color), a function containing a spline with two components will only return a two component vector and so on.

Note: the number of parameters of special function types is determined automatically, so you do not need to specify parameter names.

Declaring User-Defined Color Functions

Right now you may only declare color functions using one of the special function types. The only supported type is the `pigment` function. You may use every valid `pigment`. This is a very simple example:

```
#declare foo = function {
  pigment {
    color red 1
  }
}

#declare Vec = foo(1,2,3)
```

An example using a pattern:

```
#declare foo = function {
  pigment {
    crackle
    color_map {
      [0.3, color Red]
```

```
      [1.0, color Blue]
    }
  }
}

#declare Val = foo(2,3,4).gray
```

Note: the number of parameters of special function types is determined automatically, so you do not need to specify parameter names.

Internal Pre-Defined Functions

Several functions are pre-defined. These internal functions can be accessed through the “functions.inc”, so it should be included in your scene.
The number of required parameters and what they control are also given in the include file, but the “functions.inc” chapter in the “Standard Include File” section gives more information.

1.1.7 Strings

The POV-Ray language requires you to specify a string of characters to be used as a file name, text for messages or text for a text object. Strings may be specified using literals, identifiers or functions which return string values. See “String Functions” for details on string functions. Although you cannot build string expressions from symbolic operators such as are used with floats, vectors or colors, you may perform various string operations using string functions. Some applications of strings in POV-Ray allow for non-printing formatting characters such as newline or form-feed.

```
STRING:
    STRING_FUNCTION   |
    STRING_IDENTIFIER |
STRING_LITERAL STRING_LITERAL:
    "up to 256 ASCII characters"
STRING_FUNCTION:
    str( FLOAT , INT , INT ) |
    concat( STRING , STRING , [STRING ,...]) | chr( INT ) |
    substr( STRING , INT , INT ) | strupr( STRING ) |
    strlwr( STRING ) | vstr( INT, VECTOR, STRING, INT, INT )
```

String Literals

String literals begin with a double quote mark ’“’ which is followed by up to 256 characters and are terminated by another double quote mark. You can change the character set of strings using the `global_settings charset` option. The following are all valid string literals:

”Here“ ”There“ ”myfile.gif“ ”textures.inc“

Note: if you need to specify a quote mark in a string literal you must precede it with a backslash.

Example

```
"Joe said \"Hello\" as he walked in."
```

is converted to

```
Joe said "Hello" as he walked in.
```

If you need to specify a backslash, you will have to specify two. For example:

```
"This is a backslash \\ and this is two \\\\"
```

Is converted to:

```
This is a backslash \ and this is two \\
```

Windows users need to be especially wary about this as the backslash is also the windows path separator. For example, the following code does not produce the intended result:

```
#declare DisplayFont = "c:\windows\fonts\lucon.ttf"
text { ttf DisplayFont "Hello", 2,0 translate y*1.50 }
```

New users might expect this to create a text object using the font ”c:\windows\fonts\lucon.ttf”. Instead, it will give an error message saying that it cannot find the font file ”c:windowsontslucon.ttf”.

The correct form of the above code is as follows:

```
#declare DisplayFont = "c:\\windows\\fonts\\lucon.ttf"
text { ttf DisplayFont "Hello", 2,0 translate y*1.50 }
```

The escaping of backslashes occurs in all POV-Ray string literals. There are also other formatting codes such as `\n` for new line. See ”Text Formatting” for details.

String Identifiers

String identifiers may be declared to make scene files more readable and to parameterize scenes so that changing a single declaration changes many values. An identifier is declared as follows.

```
STRING_DECLARATION:
    #declare IDENTIFIER = STRING |
    #local IDENTIFIER = STRING
```

Where *IDENTIFIER* is the name of the identifier up to 40 characters long and *STRING* is any valid string specification.

Note: unlike floats, vectors, or colors, there need not be a semi-colon at the end of the declaration. See ”#declare vs. #local” for information on identifier scope.

Here are some examples...

```
#declare Font_Name = "ariel.ttf"
#declare Inc_File = "myfile.inc"
#declare Name = "John"
#declare Name = concat(Name," Doe")
```

As the last example shows, you can re-declare a string identifier and may use previously declared values in that re-declaration.

String Functions

POV-Ray defines a variety of built-in functions for manipulating floats, vectors and strings. Function calls consist of a keyword which specifies the name of the function followed by a parameter list enclosed in parentheses. Parameters are separated by commas. For example:

```
keyword(param1,param2)
```

The following are the functions which return string values. They take one or more float, integer, vector, or string parameters. Assume that `A` is any valid expression that evaluates to a float; `B`, `L`, and `P` are floats which are truncated to integers internally, `S`, `S1`, `S2` etc are strings.

`chr(B)` Character whose character value is `B`. Returns a single character string. The character value of the character is specified by an integer `B` which must be in the range 0 to 65535 if you specified `charset utf8` in the `global_settings` and 0 to 127 if you specified `charset ascii`. Refer to your platform specific documentation if you specified `charset sys`. For example `chr(70)` is the string "F". When rendering text objects you should be aware that the characters rendered are dependent on the (TTF) font being used.

`concat(S1,S2,...)` Concatenate strings `S1` and `S2`. Returns a string that is the concatenation of all parameter strings. Must have at least 2 parameters but may have more. For example:

```
concat("Value is ", str(A,3,1), " inches")
```

If the float value `A` was `12.34321` the result is `"Value is 12.3 inches"` which is a string.

`str(A,L,P)`: Convert float A to a formatted string. Returns a formatted string representation of float value `A`. The integer parameter `L` specifies the minimum length of the string and the type of left padding used if the string's representation is shorter than the minimum. If `L` is positive then the padding is with blanks. If `L` is negative then the padding is with zeros. The overall minimum length of the formatted string is *abs(L)*. If the string needs to be longer, it will be made as long as necessary to represent the value.

The integer parameter `P` specifies the number of digits after the decimal point. If `P` is negative then a compiler-specific default precision is use. Here are some examples:

```
str(123.456, 0, 3) "123.456"
str(123.456, 4, 3) "123.456"
str(123.456, 9, 3) "  123.456"
str(123.456,-9, 3) "00123.456"

str(123.456, 0, 2) "123.46"
str(123.456, 0, 0) "123"
str(123.456, 5, 0) "  123"
str(123.000, 7, 2) " 123.00"

str(123.456, 0,-1) "123.456000" (platform specific)
```

`strlwr(S)` Lower case of `S`. Returns a new string in which all upper case letters in the string S1 are converted to lower case. The original string is not affected. For example

`strlwr("Hello There!")` results in ”hello there!”.

`substr(S,P,L)` Sub-string from `S`. Returns a string that is a subset of the characters in parameter `S` starting at the position specified by the integer value `P` for a length specified by the integer value `L`. For example `substr("ABCDEFGHI",4,2)` evaluates to the string ”DE”. If *P+L-1>strlen(S)* an error occurs.

`strupr(S)` Upper case of `S`. Returns a new string in which all lower case letters in the string `S` are converted to upper case. The original string is not affected. For example `strupr("Hello There!")` results in ”HELLO THERE!”.

`vstr(N,A,S,L,P)` Convert vector A to a formatted string. Returns a formatted string representation of vector `A` where the elements of the vector are separated by the string parameter `S`. The integer parameter `N` specifies the amount of dimensions in vector `A`. `N` is autoclipped to the range of 2 to 5, without warning. Specifying a vector `A` with more dimensions than given by `N` will result in an error.
The integer parameter `L` specifies the minimum length of the string and the type of left padding used if the string's representation is shorter than the minimum. The integer parameter `P` specifies the number of digits after the decimal point. If `P` is negative then a compiler-specific default precision is use. The function of `L` and `P` is the same as in `str`. Here are some examples:

```
vstr(2, <1,2>, ", ", 0,1)          "1.0, 2.0"
vstr(5, <1,2,3,4,5>, ", ", 0,1)    "1.0, 2.0, 3.0, 4.0, 5.0"
vstr(1, 1, ", ", 0,1)              "1.0, 1.0"
vstr(2, 1, ", ", 0,1)              "1.0, 1.0"
vstr(5, 1, ", ", 0,1)              "1.0, 1.0, 1.0, 1.0, 1.0"
vstr(7, 1, ", ", 0,1)              "1.0, 1.0, 1.0, 1.0, 1.0"
vstr(3, <1,2>, ", ", 0,1)          "1.0, 2.0, 0.0"
vstr(5, <1,2,3>, ", ", 0,1)        "1.0, 2.0, 3.0, 0.0, 0.0"
vstr(3, <1,2,3,4>, ", ", 0,1)      error
```

See section ”Float Functions” for other functions which are somewhat string-related but which return floats. In addition to the above built-in functions, you may also define your own functions using the `#macro` directive. See the section ”User Defined Macros” for more details.

1.1.8 Array Identifiers

You may declare arrays of identifiers of up to five dimensions. Any item that can be declared as an identifier can be declared in an array.

Declaring Arrays

The syntax for declaring an array is as follows:

```
ARRAY_DECLARATION:
  #declare IDENTIFIER = array[ INT ][[ INT ]]..[ARRAY_INITIALIZER] |
  #local IDENTIFIER = array[ INT ][[ INT ]]..[ARRAY_INITIALIZER]
ARRAY_INITIALIZER:
  {ARRAY_ITEM, [ARRAY_ITEM, ]... }
ARRAY_ITEM:
```

```
RVALUE | ARRAY_INITIALIZER
```

Where IDENTIFIER is the name of the identifier up to 40 characters long and INT is a valid float expression which is internally truncated to an integer which specifies the size of the array. The optional *ARRAY_INITIALIZER* is discussed in the next section ”Array Initializers”. Here is an example of a one-dimensional, uninitialized array.

```
#declare MyArray = array[10]
```

This declares an uninitialized array of ten elements. The elements are referenced as `MyArray[0]` through `MyArray[9]`. As yet, the type of the elements are undetermined. Once you have initialized any element of the array, all other elements can only be defined as that type. An attempt to reference an uninitialized element results in an error. For example:

```
#declare MyArray = array[10]
#declare MyArray[5] = pigment{White}     //all other elements must
                                         //be pigments too.
#declare MyArray[2] = normal{bumps 0.2}  //generates an error
#declare Thing = MyArray[4]   //error: uninitialized array element
```

Multi-dimensional arrays up to five dimensions may be declared. For example:

```
#declare MyGrid = array[4][5]
```

declares a 20 element array of 4 rows and 5 columns. Elements are referenced from `MyGrid[0][0]` to `MyGrid[3][4]`. Although it is permissible to reference an entire array as a whole, you may not reference just one dimension of a multi-dimensional array. For example:

```
#declare MyArray = array[10]
#declare MyGrid = array[4][5]
#declare YourArray = MyArray  //this is ok
#declare YourGrid = MyGrid   //so is this
#declare OneRow  = MyGrid[2] //this is illegal
```

The `#ifdef` and `#ifndef` directives can be used to check whether a specific element of an array has been declared. For methods to determine the size of an array look in the float section for `dimensions` and `dimension_size`

Large uninitialized arrays do not take much memory. Internally they are arrays of pointers so they probably use just 4 bytes per element. Once initialized with values, they consume memory depending on what you put in them.

The rules for local vs. global arrays are the same as any other identifier.

Note: this applies to the entire array. You cannot mix local and global elements in the same array. See ”#declare vs. #local” for information on identifier scope.

Array Initializers

Because it is cumbersome to individually initialize the elements of an array, you may initialize it as it is created using array initializer syntax. For example:

```
#include "colors.inc"
#declare FlagColors = array[3] {Red,White,Blue}
```

Multi-dimensional arrays may also be initialized this way. For example:

```
#declare Digits =
array[4][10]
{
 {7,6,7,0,2,1,6,5,5,0},
 {1,2,3,4,5,6,7,8,9,0},
 {0,9,8,7,6,5,4,3,2,1},
 {1,1,2,2,3,3,4,4,5,5}
}
```

The commas are required between elements and between dimensions as shown in the example.

1.1.9 Spline Identifiers

Splines give you a way to define 'pathways' through your scenes. You specify a series of points, and POV-Ray interpolates to make a curve connecting them. Every point along the spline has a numerical value. A good example of a spline is the path of a moving object: the spline itself would be the path traced out by the object and the 'parameter' would be time; as time changes the object's position moves along the spline. Therefore, given a time reference you could use this spline to find the position of the object. In fact, splines are very well suited to animation.

The syntax is:

```
SPLINE_DECLARATION:
  #declare IDENTIFIER =
    spline {
      [SPLINE_IDENTIFIER] |
      [SPLINE_TYPE] |
      [Val_1, <Point_1>[,]
       Val_2, <Point_2>[,]
       ...
       Val_n, <Point_n>]
    }

SPLINE_TYPE:
  linear_spline | quadratic_spline | cubic_spline | natural_spline

SPLINE_USAGE:
  MySpline(Val) | MySpline(Val, SPLINE_TYPE)
```

The first item gives the type of interpolation.
In a `linear_spline`, straight lines connect each point.
In a `quadratic_spline`, a smooth curve defined by a second-order polynomial connects each point.
In `cubic_spline` and `natural_spline`, a smooth curve defined by a third-order polynomial connects each point.
The default is `linear_spline`.

Following this are a number of float values each followed by a position vector, all separated by commas. `Val_1`, `Val_2`, etc, are the value of the spline parameter at each specific point. The points need not be in order of their parameter values. If two points

have the same parameter value, the second point will replace the first. Beyond the range of the lowest and highest parameter values, the spline position is fixed at the endpoints.

Note: Because of the way cubic_splines are defined: the first and last points are tangents rather than points on the spline, cubic_spline interpolation is only valid between the second and next-to-last points. For all other spline types, interpolation is valid from the first point to the last point. For t-values outside the valid range, POV-Ray returns the value of the nearest valid point.

To use a spline, you place the spline identifier followed by the parameter (in parentheses) wherever you would normally put a vector, similar to a macro. Splines behave mostly like three-dimensional vectors.
Here is an example:

```
camera { location <0,2,-2> look_at 0 }
light_source { <-5,30,-10> 1 }
#declare MySpline =
  spline {
    cubic_spline
    -.25, <0,0,-1>
    0.00, <1,0,0>
    0.25, <0,0,1>
    0.50, <-1,0,0>
    0.75, <0,0,-1>
    1.00, <1,0,0>
    1.25, <0,0,1>
  }

#declare ctr = 0;
#while (ctr < 1)
  sphere {
    MySpline(ctr),.25
    pigment { rgb <1-ctr,ctr,0> }
  }
  #declare ctr = ctr + 0.01;
#end
```

You can also have POV-Ray evaluate a spline as if it were a different type of spline by specifying the type of spline after the value to interpolate at, for example:

```
sphere{ <2,0,2>, .25 pigment{rgb MySpline(clock, linear_spline)}}
```

Splines are 'intelligent' when it comes to returning vectors. The vector with the most components in the spline determines the size of the returned vector. This allows vectors from two to five components to be returned by splines.

Also, function splines take the vector size into account. That is, a function containing a spline with five components will also return a five component vector (aka a color), a function containing a spline with two components will only return a two component vector and so on.

Splines and Macros

You can pass functions to macros, how to do this is best explained by an example

```
#macro Foo( Bar, Val )
  #declare Y = Bar(Val).y;
#end

#declare myspline = spline {
    1, <4,5>
    3, <5,5>
    5, <6,5>
}

Foo(myspline, 2)

#debug str(Y,5,5)
#debug "\n"
```

1.2 Language Directives

The POV Scene Language contains several statements called *language directives* which tell the file parser how to do its job. These directives can appear in almost any place in the scene file - even in the middle of some other statements. They are used to include other text files in the stream of commands, to declare identifiers, to define macros, conditional, or looped parsing and to control other important aspects of scene file processing.

Each directive begins with the hash character # (often called a number sign or pound sign). It is followed by a keyword and optionally other parameters.

In versions of POV-Ray prior to 3.0, the use of this # character was optional. Language directives could only be used between objects, camera or light_source statements and could not appear within those statements. The exception was the #include which could appear anywhere. Now that all language directives can be used almost anywhere, the # character is mandatory. The following keywords introduce language directives.

#break	#fopen	#render
#case	#if	#statistics
#debug	#ifdef	#switch
#declare	#ifndef	#undef
#default	#include	#version
#else	#local	#warning
#end	#macro	#while
#error	#range	#write
#fclose	#read	

Table 1.5: All language directives

Earlier versions of POV-Ray considered the keyword `#max_intersections` and the keyword `#max_trace_level` to be language directives but they have been moved to the `global_settings` statement and should be placed there without the # sign. Their use as a directive still works but it generates a warning and may be discontinued in the future.

1.2.1 Include Files and the #include Directive

The language allows include files to be specified by placing the line

```
#include "filename.inc"
```

at any point in the input file. The filename may be specified by any valid string expression but it usually is a literal string enclosed in double quotes. It may be up to 40 characters long (or your computer's limit), including the two double-quote characters.

The include file is read in as if it were inserted at that point in the file. Using include is almost the same as cutting and pasting the entire contents of this file into your scene.

Include files may be nested. You may have at most 10 nested include files. There is no limit on un-nested include files.

Generally, include files have data for scenes but are not scenes in themselves. By convention scene files end in `.pov` and include files end with `.inc`.

It is legal to specify drive and directory information in the file specification however it is discouraged because it makes scene files less portable between various platforms. Use of full lower case is also recommended but not required.

Note: if you ever intend to distribute any source files you make for POV-Ray, remember that some operating systems have case-sensitive file names).

It is typical to put standard include files in a special sub-directory. POV-Ray can only read files in the current directory or one referenced by the `Library_Path` option or `+L` switch. See section ”Library Paths”.

You may use the `#local` directive to declare identifiers which are temporary in duration and local to the include file in scope. For details see ”#declare vs. #local”.

1.2.2 The #declare and #local Directives

Identifiers may be declared and later referenced to make scene files more readable and to parameterize scenes so that changing a single declaration changes many values. There are several built-in identifiers which POV-Ray declares for you. See section ”Built-in Float Identifiers” and ”Built-in Vector Identifiers” for details.

Declaring identifiers

An identifier is declared as follows.

```
DECLARATION:
    #declare IDENTIFIER = RVALUE |
    #local IDENTIFIER = RVALUE
```

```
RVALUE:
    FLOAT; | VECTOR; | COLOR; | STRING | OBJECT | TEXTURE |
    PIGMENT | NORMAL | FINISH | INTERIOR | MEDIA | DENSITY |
    COLOR_MAP | PIGMENT_MAP | SLOPE_MAP | NORMAL_MAP |
    DENSITY_MAP | CAMERA | LIGHT_SOURCE | FOG | RAINBOW |
    SKY_SPHERE | TRANSFORM
```

Where *IDENTIFIER* is the name of the identifier up to 40 characters long and *RVALUE* is any of the listed items. They are called that because they are values that can appear to the *right* of the equals sign. The syntax for each is in the corresponding section of this language reference. Here are some examples.

```
#declare Rows = 5;
#declare Count = Count+1;
#local  Here = <1,2,3>;
#declare White = rgb <1,1,1>;
#declare Cyan = color blue 1.0 green 1.0;
#declare Font_Name = "ariel.ttf"
#declare Rod = cylinder {-5*x,5*x,1}
#declare Ring = torus {5,1}
#local  Checks = pigment { checker White, Cyan }
object{ Rod scale y*5 }     // not "cylinder { Rod }"
object {
 Ring
 pigment { Checks scale 0.5 }
 transform Skew
}
```

Note: that there should be a semi-colon after the expression in all float, vector and color identifier declarations. This semi-colon is introduced in POV-Ray version 3.1. If omitted, it generates a warning and some macros may not work properly. Semicolons after other declarations are optional.

Declarations, like most language directives, can appear almost anywhere in the file - even within other statements. For example:

```
#declare Here=<1,2,3>;
#declare Count=0;         // initialize Count
union {
 object { Rod translate Here*Count }
 #declare Count=Count+1;     // re-declare inside union
 object { Rod translate Here*Count }
 #declare Count=Count+1;     // re-declare inside union
 object { Rod translate Here*Count }
}
```

As this example shows, you can re-declare an identifier and may use previously declared values in that re-declaration.

Note: object identifiers use the generic wrapper statement `object{ ... }`. You do not need to know what kind of object it is.

Declarations may be nested inside each other within limits. In the example in the previous section you could declare the entire union as a object. However for technical reasons there are instances where you may not use any language directive inside the

declaration of floats, vectors or color expressions. Although these limits have been loosened somewhat since POV-Ray 3.1, they still exist.

Identifiers declared within `#macro` ... `#end` blocks are not created at the time the macro is defined. They are only created at the time the macro is actually invoked. Like all other items inside such a #macro definition, they are ignored when the macro is defined.

#declare vs. #local

Identifiers may be declared either global using `#declare` or local using the `#local` directive.

Those created by the `#declare` directive are permanent in duration and global in scope. Once created, they are available throughout the scene and they are not released until all parsing is complete or until they are specifically released using `#undef`. See "Destroying Identifiers".

Those created by the `#local` directive are temporary in duration and local in scope. They temporarily override any identifiers with the same name. See "Identifier Name Collisions".

If `#local` is used inside a `#macro` then the identifier is local to that macro. When the macro is invoked and the `#local` directive is parsed, the identifier is created. It persists until the `#end` directive of the macro is reached. At the `#end` directive, the identifier is destroyed. Subsequent invocations of the macro create totally new identifiers.

Use of `#local` within an include file but not in a macro, also creates a temporary identifier that is local to that include file. When the include file is included and the `#local` directive is parsed, the identifier is created. It persists until the end of the include file is reached. At the end of file the identifier is destroyed. Subsequent inclusions of the file create totally new identifiers.

Use of `#local` in the main scene file (not in an include file and not in a macro) is identical to `#declare`. For clarity sake you should not use `#local` in a main file except in a macro.

There is currently no way to create permanent, yet local identifiers in POV-Ray.

Local identifiers may be specifically released early using `#undef` but in general there is no need to do so. See "Destroying Identifiers".

Identifier Name Collisions

Local identifiers may have the same names as previously declared identifiers. In this instance, the most recent, most local identifier takes precedence. Upon entering an include file or invoking a macro, a new symbol table is created. When referencing identifiers, the most recently created symbol table is searched first, then the next most recent and so on back to the global table of the main scene file. As each macro or include file is exited, its table and identifiers are destroyed. Parameters passed by value reside in the same symbol table as the one used for identifiers local to the macro.

The rules for duplicate identifiers may seem complicated when multiple-nested includes and macros are involved, but in actual practice the results are generally what you intended.

Consider this example: You have a main scene file called myscene.pov and it contains

```
#declare A = 123;
#declare B = rgb<1,2,3>;
#declare C = 0;
#include "myinc.inc"
```

Inside the include file you invoke a macro called MyMacro(J,K,L). It isn't important where MyMacro is defined as long as it is defined before it is invoked. In this example, it is important that the macro is invoked from within myinc.inc.

The identifiers A, B, and C are generally available at all levels. If either myinc.inc or MyMacro contain a line such as #declare C=C+1; then the value C is changed everywhere as you might expect.

Now suppose inside myinc.inc you do...

```
#local A = 546;
```

The main version of A is hidden and a new A is created. This new A is also available inside MyMacro because MyMacro is nested inside myinc.inc. Once you exit myinc.inc, the local A is destroyed and the original A with its value of 123 is now in effect. Once you have created the local A inside myinc.inc, there is no way to reference the original global A unless you #undef A or exit the include file. Using #undef always undefines the most local version of an identifier.

Similarly if MyMacro contained...

```
#local B = box{0,1}
```

then a new identifier B is created local to the macro only. The original value of B remains hidden but is restored when the macro is finished. The local B need not have the same type as the original.

The complication comes when trying to assign a new value to an identifier at one level that was declared local at an earlier level. Suppose inside myinc.inc you do...

```
#local D = 789;
```

If you are inside myinc.inc and you want to increment D by one, you might try to do...

```
#local D = D + 1;
```

but if you try to do that inside MyMacro you'll create a new D which is local to MyMacro and not the D which is external to MyMacro but local to myinc.inc. Therefore you've said "create a MyMacro D from the value of myinc.inc's D plus one". That's probably not what you wanted. Instead you should do...

```
#declare D = D + 1;
```

You might think this creates a new D that is global but it actually increments the myinc.inc version of D. Confusing isn't it? Here are the rules:

1. When referencing an identifier, you always get the most recent, most local version. By ”referencing” we mean using the value of the identifier in a POV-Ray statement or using it on the right of an equals sign in either a `#declare` or `#local`.
2. When declaring an identifier using the `#local` keyword, the identifier which is created or has a new value assigned, is ALWAYS created at the current nesting level of macros or include files.
3. When declaring a NEW, NON-EXISTANT identifier using `#declare`, it is created as fully global. It is put in the symbol table of the main scene file.
4. When ASSIGNING A VALUE TO AN EXISTING identifier using `#declare`, it assigns it to the most recent, most local version at the time.

In summary, `#local` always means ”the current level”, and `#declare` means ”global” for new identifiers and ”most recent” for existing identifiers.

Destroying Identifiers with #undef

Identifiers created with `#declare` will generally persist until parsing is complete. Identifiers created with `#local` will persist until the end of the macro or include file in which they were created. You may however un-define an identifier using the `#undef` directive. For example:

```
#undef MyValue
```

If multiple local nested versions of the identifier exist, the most local most recent version is deleted and any identically named identifiers which were created at higher levels will still exist.

See also ”The #ifdef and #ifndef Directives”.

1.2.3 File I/O Directives

You may open, read, write, append, and close plain ASCII text files while parsing POV-Ray scenes. This feature is primarily intended to help pass information between frames of an animation. Values such as an object’s position can be written while parsing the current frame and read back during the next frame. Clever use of this feature could allow a POV-Ray scene to generate its own include files or write self-modifying scripts. We trust that users will come up with other interesting uses for this feature.

Note: some platform versions of POV-Ray (e.g. Windows) provide means to restrict the ability of scene files to read & write files.

The #fopen Directive

Users may open a text file using the `#fopen` directive. The syntax is as follows:

```
FOPEN_DIRECTIVE:
    #fopen IDENTIFIER "filename" OPEN_TYPE
OPEN_TYPE:
```

```
    read | write | append
```

Where *IDENTIFIER* is an undefined identifier used to reference this file as a file handle, *"filename"* is any string literal or string expression which specifies the file name. Files opened with the `read` are open for read only. Those opened with `write` create a new file with the specified name and it overwrites any existing file with that name. Those opened with `append` opens a file for writing but appends the text to the end of any existing file.

The file handle identifier created by `#fopen` is always global and remains in effect (and the file remains open) until the scene parsing is complete or until you `#fclose` the file. You may use `#ifdef` *FILE_HANDLE_IDENTIFIER* to see if a file is open.

The #fclose Directive

Files opened with the `#fopen` directive are automatically closed when scene parsing completes however you may close a file using the `#fclose` directive. The syntax is as follows:

```
FCLOSE_DIRECTIVE:
    #fclose FILE_HANDLE_IDENTIFIER
```

Where *FILE_HANDLE_IDENTIFIER* is previously opened file opened with the `#fopen` directive. See "The #fopen Directive".

The #read Directive

You may read string, float or vector values from a plain ASCII text file directly into POV-Ray variables using the #read directive. The file must first be opened in "read" mode using the #fopen directive. The syntax for #read is as follows:

```
READ_DIRECTIVE:
  #read (FILE_HANDLE_IDENTIFIER, DATA_IDENTIFIER[,DATA_IDENTIFIER]..)
DATA_IDENTIFIER:
  UNDECLARED_IDENTIFIER | FLOAT_IDENTIFIER | VECTOR_IDENTIFIER |
  STRING_IDENTIFIER
```

Where *FILE_HANDLE_IDENTIFIER* is the previously opened file. It is followed by one or more *DATA_IDENTIFIER*s separated by commas. The parentheses around the identifier list are required. A *DATA_IDENTIFIER* is any undeclared identifier or any previously declared string identifier, float identifier, or vector identifier. Undefined identifiers will be turned into global identifiers of the type determined by the data which is read. Previously defined identifiers remain at whatever globa/õcal status they had when originally created. Type checking is performed to insure that the proper type data is read into these identifiers.

The format of the data to be read must be a series of valid string literals, float literals, or vector literals separated by commas. Expressions or identifiers are not permitted in the data file however unary minus signs and exponential notation are permitted on float values.

If you attempt to read past end-of-file, the file is automatically closed and the *FILE_HANDLE_IDENTIFIER* is deleted from the symbol table. This means that the boolean function `defined(`*IDENTIFIER*`)` can be used to detect end-of-file. For example:

```
#fopen MyFile "mydata.txt" read
#while (defined(MyFile))
  #read (MyFile,Var1,Var2,Var3)
  ...
#end
```

The #write Directive

You may write string, float or vector values to a plain ASCII text file from POV-Ray variables using the `#write` directive. The file must first be opened in either `write` or `append` mode using the `#fopen` directive. The syntax for `#write` is as follows:

```
WRITE_DIRECTIVE:
    #write( FILE_HANDLE_IDENTIFIER, DATA_ITEM[,DATA_ITEM]...)
DATA_ITEM:
    FLOAT | VECTOR | STRING
```

Where *FILE_HANDLE_IDENTIFIER* is the previously opened file. It is followed by one or more *DATA_ITEM*s separated by commas. The parentheses around the identifier list are required. A *DATA_ITEM* is any valid string expression, float expression, or vector expression. Float expressions are evaluated and written as signed float literals. If you require format control, you should use the `str(VALUE,L,P)` function to convert it to a formatted string. See "String Functions" for details on the `str` function. Vector expressions are evaluated into three signed float constants and are written with angle brackets and commas in standard POV-Ray vector notation. String expressions are evaluated and written as specified.

Note: data read by the `#read` directive must have comma delimiters between values and quotes around string data but the `#write` directive does not automatically output commas or quotes.

For example the following `#read` directive reads a string, float and vector.

```
#read (MyFile,MyString,MyFloat,MyVect)
```

It expects to read something like:

```
"A quote delimited string", -123.45, <1,2,-3>
```

The POV-Ray code to write this might be:

```
#declare Val1 = -123.45;
#declare Vect1 = <1,2,-3>;
#write(MyFile,"\"A quote delimited string\",",Val1,",",Vect1,"\n")
```

See "String Literals" and "Text Formatting" for details on writing special characters such as quotes, newline, etc.

1.2.4 The #default Directive

POV-Ray creates a default texture when it begins processing. You may change those defaults as described below. Every time you specify a `texture` statement, POV-Ray creates a copy of the default texture. Anything you put in the texture statement overrides the default settings. If you attach a `pigment`, `normal`, or `finish` to an object without any texture statement then POV-Ray checks to see if a texture has already been attached. If it has a texture then the pigment, normal or finish will modify the existing texture. If no texture has yet been attached to the object then the default texture is copied and the pigment, normal or finish will modify that texture.

You may change the default texture, pigment, normal or finish using the language directive `#default` as follows:

```
DEFAULT_DIRECTIVE:
    #default {DEFAULT_ITEM }
DEFAULT_ITEM:
    TEXTURE | PIGMENT | NORMAL | FINISH
```

For example:

```
#default {
  texture {
    pigment { rgb <1,0,0> }
    normal { bumps 0.3 }
    finish { ambient 0.4 }
  }
}
```

This means objects will default to red bumps and slightly high ambient finish. Also you may change just part of it like this:

```
#default {
 pigment {rgb <1,0,0>}
}
```

This still changes the pigment of the default texture. At any time there is only one default texture made from the default pigment, normal and finish. The example above does not make a separate default for pigments alone.

Note: the special textures `tiles` and `material_map` or a texture with a `texture_map` may not be used as defaults.

You may change the defaults several times throughout a scene as you wish. Subsequent `#default` statements begin with the defaults that were in effect at the time. If you wish to reset to the original POV-Ray defaults then you should first save them as follows:

```
//At top of file
#declare Original_Default = texture {}
```

later after changing defaults you may restore it with...

```
#default {texture {Original_Default}}
```

If you do not specify a texture for an object then the default texture is attached when the object appears in the scene. It is not attached when an object is declared. For example:

```
#declare My_Object =
 sphere{ <0,0,0>, 1 } // Default texture not applied
 object{ My_Object }  // Default texture added here
```

You may force a default texture to be added by using an empty texture statement as follows:

```
#declare My_Thing =
 sphere { <0,0,0>, 1 texture {} } // Default texture applied
```

The original POV-Ray defaults for all items are given throughout the documentation under each appropriate section.

1.2.5 The #version Directive

As POV-Ray has evolved from version 1.0 through 3.5 we have made every effort to maintain some amount of backwards compatibility with earlier versions. Some old or obsolete features can be handled directly without any special consideration by the user. Some old or obsolete features can no longer be handled at all. However *some* old features can still be used if you warn POV-Ray that this is an older scene. The `#version` directive can be used to switch version compatibility to different setting several times throughout a scene file. The syntax is:

```
VERSION_DIRECTIVE:
    #version FLOAT;
```

Note: there should be a semi-colon after the float expression in a `#version` directive. This semi-colon is introduced in POV-Ray version 3.1. If omitted, it generates a warning and some macros may not work properly.

Additionally you may use the `Version=`*n.n* option or the `+MV`*n.n* switch to establish the *initial* setting. See "Language Version" for details. For example one feature introduced in 2.0 that was incompatible with any 1.0 scene files is the parsing of float expressions. Using `#version 1.0` turns off expression parsing as well as many warning messages so that nearly all 1.0 files will still work. Naturally the default setting for this option is `#version 3.5`.

Note: Some obsolete or re-designed features *are totally unavailable in POV-Ray 3.5 REGARDLES OF THE VERSION SETTING.* Details on these features are noted throughout this documentation.

The built-in float identifier `version` contains the current setting of the version compatibility option. See "Built-in Float Identifiers". Together with the built-in `version` identifier the `#version` directive allows you to save and restore the previous values of this compatibility setting. The new `#local` identifier option is especially useful here. For example suppose `mystuff.inc` is in version 1 format. At the top of the file you could put:

```
#local Temp_Vers = version;  // Save previous value
#version 1.0;          // Change to 1.0 mode
... // Version 1.0 stuff goes here...
#version Temp_Vers;      // Restore previous version
```

Future versions of POV-Ray may not continue to maintain full backward compatibility even with the `#version` directive. We strongly encourage you to phase in 3.5 syntax as much as possible.

1.2.6 Conditional Directives

POV-Ray allows a variety of language directives to implement conditional parsing of various sections of your scene file. This is especially useful in describing the motion for animations but it has other uses as well. Also available is a `#while` loop directive. You may nest conditional directives 200 levels deep.

The #if...#else...#end Directives

The simplest conditional directive is a traditional `#if` directive. It is of the form...

```
IF_DIRECTIVE:
    #if ( Cond ) TOKENS... [#else TOKENS...] #end
```

The *TOKENS* are any number of POV-Ray keyword, identifiers, or punctuation and `(` *Cond* `)` is a float expression that is interpreted as a boolean value. The parentheses are required. The `#end` directive is required. A value of 0.0 is false and any non-zero value is true.

Note: extremely small values of about 1e-10 are considered zero in case of round off errors.

If *Cond* is true, the first group of tokens is parsed normally and the second set is skipped. If false, the first set is skipped and the second set is parsed. For example:

```
#declare Which=1;
#if (Which)
  box { 0, 1 }
#else
  sphere { 0, 1 }
#end
```

The box is parsed and the sphere is skipped. Changing the value of `Which` to `0` means the box is skipped and the sphere is used. The `#else` directive and second token group is optional. For example:

```
#declare Which=1;
#if (Which)
  box { 0, 1 }
#end
```

Changing the value of `Which` to `0` means the box is removed.

At the beginning of the chapter ”Language Directives” it was stated that ”These directives can appear in almost any place in the scene file....”. The following is an example where it will not work, it will confuse the parser:

```
#if( #if(yes) yes #end ) #end
```

The #ifdef and #ifndef Directives

The `#ifdef` and `#ifndef` directive are similar to the `#if` directive however they are used to determine if an identifier has been previously declared.

```
IFDEF_DIRECTIVE:
    #ifdef ( IDENTIFIER ) TOKENS... [#else TOKENS...] #end
IFNDEF_DIRECTIVE:
    #ifndef ( IDENTIFIER ) TOKENS... [#else TOKENS...] #end
```

If the *IDENTIFIER* exists then the first group of tokens is parsed normally and the second set is skipped. If false, the first set is skipped and the second set is parsed. This is especially useful for replacing an undefined item with a default. For example:

```
  #ifdef (User_Thing)
  // This section is parsed if the
  // identifier "User_Thing" was
  // previously declared
  object{User_Thing} // invoke identifier
 #else
  // This section is parsed if the
  // identifier "User_Thing" was not
  // previously declared
  box{<0,0,0>,<1,1,1>} // use a default
 #end
  // End of conditional part
```

The `#ifndef` directive works the opposite. The first group is parsed if the identifier is *not* defined. As with the `#if` directive, the `#else` clause is optional and the `#end` directive is required.

The `#ifdef` and `#ifndef` directives can be used to determine whether a specific element of an array has been assigned.

```
 #declare MyArray=array[10]
 //#declare MyArray[0]=7;
 #ifdef(MyArray[0])
    #debug "first element is assigned\n"
 #else
    #debug "first element is not assigned\n"
 #end
```

The #switch, #case, #range and #break Directives

A more powerful conditional is the `#switch` directive. The syntax is as follows...

```
SWITCH_DIRECTIVE:
   #switch ( Switch_Value ) SWITCH_CLAUSE... [#else TOKENS...] #end
SWITCH_CLAUSE:
   #case( Case_Value ) TOKENS... [#break] |
   #range( Low_Value , High_Value ) TOKENS... [#break]
```

The *TOKENS* are any number of POV-Ray keyword, identifiers, or punctuation and `(` *Switch_Value* `)` is a float expression. The parentheses are required. The `#end` directive is required. The *SWITCH_CLAUSE* comes in two varieties. In the `#case` variety, the

float *Switch_Value* is compared to the float *Case_Value*. If they are equal, the condition is true.

Note: that values whose difference is less than 1e-10 are considered equal in case of round off errors.

In the `#range` variety, *Low_Value* and *High_Value* are floats separated by a comma and enclosed in parentheses. If *Low_Value* <= *Switch_Value* and *Switch_Value*<=*High_Value* then the condition is true.

In either variety, if the clause's condition is true, that clause's tokens are parsed normally and parsing continues until a `#break`, `#else` or `#end` directive is reached. If the condition is false, POV-Ray skips until another `#case` or `#range` is found.

There may be any number of `#case` or `#range` clauses in any order you want. If a clause evaluates true but no `#break` is specified, the parsing will fall through to the next `#case` or `#range` and that clause conditional is evaluated. Hitting `#break` while parsing a successful section causes an immediate jump to the `#end` without processing subsequent sections, even if a subsequent condition would also have been satisfied.

An optional `#else` clause may be the last clause. It is only executed if the clause before it was a false clause.

Here is an example:

```
#switch (VALUE)
 #case (TEST_1)
  // This section is parsed if VALUE=TEST_1
 #break //First case ends
 #case (TEST_2)
  // This section is parsed if VALUE=TEST_2
 #break //Second case ends
 #range (LOW_1,HIGH_1)
  // This section is parsed if (VALUE>=LOW_1)&(VALUE<=HIGH_1)
 #break //Third case ends
 #range (LOW_2,HIGH_2)
  // This section is parsed if (VALUE>=LOW_2)&(VALUE<=HIGH_2)
 #break //Fourth case ends
 #else
  // This section is parsed if no other case or
  // range is true.
#end // End of conditional part
```

The #while...#end Directive

The `#while` directive is a looping feature that makes it easy to place multiple objects in a pattern or other uses.

```
WHILE_DIRECTIVE:
    #while ( Cond ) TOKENS... #end
```

The *TOKENS* are any number of POV-Ray keyword, identifiers, or punctuation marks which are the *body* of the loop. The `#while` directive is followed by a float expression that evaluates to a boolean value. A value of 0.0 is false and any non-zero value is true.

Note: extremely small values of about 1e-10 are considered zero in case of round off errors.

The parentheses around the expression are required. If the condition is true parsing continues normally until an `#end` directive is reached. At the end, POV-Ray loops back to the `#while` directive and the condition is re-evaluated. Looping continues until the condition fails. When it fails, parsing continues after the `#end` directive.

Note: it is possible for the condition to fail the first time and the loop is totally skipped. It is up to the user to insure that something inside the loop changes so that it eventually terminates.

Here is a properly constructed loop example:

```
#declare Count=0;
#while (Count < 5)
 object { MyObject translate x*3*Count }
 #declare Count=Count+1;
#end
```

This example places five copies of `MyObject` in a row spaced three units apart in the x-direction.

1.2.7 User Message Directives

With the addition of conditional and loop directives, the POV-Ray language has the potential to be more like an actual programming language. This means that it will be necessary to have some way to see what is going on when trying to debug loops and conditionals. To fulfill this need we have added the ability to print text messages to the screen. You have a choice of five different text streams to use including the ability to generate a fatal error if you find it necessary. Limited formatting is available for strings output by this method.

Text Message Streams

The syntax for a text message is any of the following:

```
TEXT_STREAM_DIRECTIVE:
    #debug STRING | #error STRING | #warning STRING
```

Where *STRING* is any valid string of text including string identifiers or functions which return strings. For example:

```
#switch (clock*360)
 #range (0,180)
  #debug "Clock in 0 to 180 range\n"
 #break
 #range (180,360)
  #debug "Clock in 180 to 360 range\n"
 #break
 #else
  #warning "Clock outside expected range\n"
  #warning concat("Value is:",str(clock*360,5,0),"\n")
```

```
#end
```

There are seven distinct text streams that POV-Ray uses for output. You may output only to three of them. On some versions of POV-Ray, each stream is designated by a particular color. Text from these streams are displayed whenever it is appropriate so there is often an intermixing of the text. The distinction is only important if you choose to turn some of the streams off or to direct some of the streams to text files. On some systems you may be able to review the streams separately in their own scroll-back buffer. See "Directing Text Streams to Files" for details on re-directing the streams to a text file.

Here is a description of how POV-Ray uses each stream. You may use them for whatever purpose you want except note that use of the `#error` stream causes a fatal error after the text is displayed.

Debug: This stream displays debugging messages. It was primarily designed for developers but this and other streams may also be used by the user to display messages from within their scene files.

Error: This stream displays fatal error messages. After displaying this text, POV-Ray will terminate. When the error is a scene parsing error, you may be shown several lines of scene text that leads up to the error.

Warning: This stream displays warning messages during the parsing of scene files and other warnings. Despite the warning, POV-Ray can continue to render the scene.

The `#render` and `#statistsics` could be accessed in previous versions. Their output is now redirected to the `#debug` stream. The `#banner` and `#status` streams can not be accessed by the user.

Text Formatting

Some escape sequences are available to include non-printing control characters in your text. These sequences are similar to those used in string literals in the C programming language. The sequences are:

`"\a"`	Bell or alarm,	0x07
`"\b"`	Backspace,	0x08
`"\f"`	Form feed,	0x0C
`"\n"`	New line (line feed)	0x0A
`"\r"`	Carriage return	0x0D
`"\t"`	Horizontal tab	0x09
`"\uNNNN"`	Unicode character code NNNN	0xNNNN
`"\v"`	Vertical tab	0x0B
`"\0"`	Null	0x00
`"\\"`	Backslash	0x5C
`"\'"`	Single quote	0x27
`"\""`	Double quote	0x22

Table 1.6: All character escape sequences

For example:

```
#debug "This is one line.\nBut this is another"
```

Depending on what platform you are using, they may not be fully supported for console output. However they will appear in any text file if you re-direct a stream to a file.

1.2.8 User Defined Macros

POV-Ray 3.1 introduced user defined macros with parameters. This feature, along with the ability to declare `#local` variables, turned the POV-Ray Language into a fully functional programming language. Consequently, it is now possible to write scene generation tools in POV-Ray's own language that previously required external utilities.

The #macro Directive

The syntax for declaring a macro is:

```
MACRO_DEFINITION:
#macro IDENTIFIER ([PARAM_IDENT] [, PARAM_IDENT]... ) TOKENS... #end
```

Where *IDENTIFIER* is the name of the macro and *PARAM_IDENT*s are a list of zero or more formal parameter identifiers separated by commas and enclosed by parentheses. The parentheses are required even if no parameters are specified.

The *TOKENS* are any number of POV-Ray keyword, identifiers, or punctuation marks which are the *body* of the macro. The body of the macro may contain almost any POV-Ray syntax items you desire. It is terminated by the `#end` directive.

Note: any conditional directives such as `#if`...`#end`, `#while`...`#end`, etc. must be fully nested inside or outside the macro so that the corresponding `#end` directives pair-up properly.

A macro must be declared before it is invoked. All macro names are global in scope and permanent in duration. You may redefine a macro by another `#macro` directive with the same name. The previous definition is lost. Macro names respond to `#ifdef`, `#ifndef`, and `#undef` directives. See "The #ifdef and #ifndef Directives" and "Destroying Identifiers with #undef".

Invoking Macros

You invoke the macro by specifying the macro name followed by a list of zero or more actual parameters enclosed in parentheses and separated by commas. The number of actual parameters must match the number of formal parameters in the definition. The parentheses are required even if no parameters are specified. The syntax is:

```
MACRO_INVOCATION:
    MACRO_IDENTIFIER ( [ACTUAL_PARAM] [, ACTUAL_PARAM]... )
ACTUAL_PARAM:
    IDENTIFIER | RVALUE
```

An *RVALUE* is any value that can legally appear to the right of an equals sign in a `#declare` or `#local` declaration. See "Declaring identifiers" for information on *RVALUE*s. When the macro is invoked, a new local symbol table is created. The actual

parameters are assigned to formal parameter identifiers as local, temporary variables. POV-Ray jumps to the body of the macro and continues parsing until the matching #end directive is reached. There, the local variables created by the parameters are destroyed as well as any local identifiers expressly created in the body of the macro. It then resumes parsing at the point where the macro was invoked. It is as though the body of the macro was cut and pasted into the scene at the point where the macro was invoked.

Note: it is possible to invoke a macro that was declared in another file. This is quite normal and in fact is how many "plug-ins" work (such as the popular Lens Flare macro). However, be aware that calling a macro that was declared in a file different from the one that it is being called from involves more overhead than calling one in the same file.

This is because POV-Ray does not tokenize and store its language. Calling a macro in another file therefore requires that the other file be opened and closed for each call. Normally, this overhead is inconsequential; however, if you are calling the macro many thousands of times, it can cause significant delays. A future version of the POV-Ray language will remove this problem.

Here is a simple macro that creates a window frame object when you specify the inner and outer dimensions.

```
#macro Make_Frame(OuterWidth,OuterHeight,InnerWidth,
                  InnerHeight,Depth)
  #local Horz = (OuterHeight-InnerHeight)/2;
  #local Vert = (OuterWidth-InnerWidth)/2;
  difference {
    box{
  <0,0,0>,<OuterWidth,OuterHeight,Depth>
}
    box{
  <Vert,Horz,-0.1>,
      <OuterWidth-Vert,OuterHeight-Horz,Depth+0.1>
}
  }
#end
Make_Frame(8,10,7,9,1) //invoke the macro
```

In this example, the macro has five float parameters. The actual parameters (the values 8, 10, 7, 9, and 1) are assigned to the five identifiers in the #macro formal parameter list. It is as though you had used the following five lines of code.

```
 #local OuterWidth = 8;
 #local OuterHeight = 10;
 #local InnerWidth, = 7;
 #local InnerHeight = 9;
 #local Depth = 1;
```

These five identifiers are stored in the same symbol table as any other local identifier such as Horz or Vert in this example. The parameters and local variables are all destroyed when the #end statement is reached. See "Identifier Name Collisions" for a detailed discussion of how local identifiers, parameters, and global identifiers work when a local identifier has the same name as a previously declared identifier.

Are POV-Ray Macros a Function or a Macro?

POV-Ray macros are a strange mix of macros and functions. In traditional computer programming languages, a macro works entirely by token substitution. The body of the routine is inserted into the invocation point by simply copying the tokens and parsing them as if they had been cut and pasted in place. Such cut-and-paste substitution is often called *macro substitution* because it is what macros are all about. In this respect, POV-Ray macros are exactly like traditional macros in that they use macro substitution for the body of the macro. However traditional macros also use this cut-and-paste substitution strategy for parameters but POV-Ray does not.

Suppose you have a macro in the C programming language `Typical_Cmac(Param)` and you invoke it as `Typical_Cmac(else A=B)`. Anywhere that `Param` appears in the macro body, the four tokens `else`, `A`, `=`, and `B` are substituted into the program code using a cut-and-paste operation. No type checking is performed because anything is legal. The ability to pass an arbitrary group of tokens via a macro parameter is a powerful (and sadly often abused) feature of traditional macros.

After careful deliberation, we have decided against this type of parameters for our macros. The reason is that POV-Ray uses commas more frequently in its syntax than do most programming languages. Suppose you create a macro that is designed to operate on one vector and two floats. It might be defined `OurMac(V,F1,F2)`. If you allow arbitrary strings of tokens and invoke a macro such as `OurMac(<1,2,3>,4,5)` then it is impossible to tell if this is a vector and two floats or if its 5 parameters with the two tokens `<` and `1` as the first parameter. If we design the macro to accept 5 parameters then we cannot invoke it like this... `OurMac(MyVector,4,5)`.

Function parameters in traditional programming languages do not use token substitution to pass values. They create temporary, local variables to store parameters that are either constant values or identifier references which are in effect a pointer to a variable. POV-Ray macros use this function-like system for passing parameters to its macros. In our example `OurMac(<1,2,3>,4,5)`, POV-Ray sees the `<` and knows it must be the start of a vector. It parses the whole vector expression and assigns it to the first parameter exactly as though you had used the statement `#local V=<1,2,3>;`.

Although we say that POV-Ray parameters are more like traditional function parameters than macro parameters, there still is one difference. Most languages require you to declare the type of each parameter in the definition before you use it but POV-Ray does not. This should be no surprise because most languages require you to declare the type of any identifier before you use it but POV-Ray does not. This means that if you pass the wrong type value in a POV-Ray macro parameter, it may not generate an error until you reference the identifier in the macro body. No type checking is performed as the parameter is passed. So in this very limited respect, POV-Ray parameters are somewhat macro-like but are mostly function-like.

Returning a Value Like a Function

POV-Ray macros have a variety of uses. Like most macros, they provide a parameterized way to insert arbitrary code into a scene file. However most POV-Ray macros will be used like functions or procedures in a traditional programming language. Macros are designed to fill all of these roles.

When the body of a macro consists of statements that create an entire item such as an object, texture, etc. then the macro acts like a function which returns a single value. The Make_Frame macro example in the section "Invoking Macros" above is such a macro which returns a value that is an object. Here are some examples of how you might invoke it.

```
union {  //make a union of two objects
  object{ Make_Frame(8,10,7,9,1) translate  20*x}
  object{ Make_Frame(8,10,7,9,1) translate -20*x}
}
#declare BigFrame = object{ Make_Frame(8,10,7,9,1)}
#declare SmallFrame = object{ Make_Frame(5,4,4,3,0.5)}
```

Because no type checking is performed on parameters and because the expression syntax for floats, vectors, and colors is identical, you can create clever macros which work on all three. See the sample scene MACRO3.POV which includes this macro to interpolate values.

```
// Define the macro.  Parameters are:
//   T:  Middle value of time
//   T1: Initial time
//   T2: Final time
//   P1: Initial position (may be float, vector or color)
//   P2: Final position (may be float, vector or color)
//   Result is a value between P1 and P2 in the same proportion
//    as T is between T1 and T2.
#macro Interpolate(T,T1,T2,P1,P2)
   (P1+(T1+T/(T2-T1))*(P2-P1))
#end
```

You might invoke it with P1 and P2 as floats, vectors, or colors as follows.

```
sphere{
  Interpolate(I,0,15,<2,3,4>,<9,8,7>), //center location is vector
  Interpolate(I,0,15,3.0,5.5)          //radius is float
  pigment {
    color Interpolate(I,0,15,rgb<1,1,0>,rgb<0,1,1>)
  }
}
```

As the float value I varies from 0 to 15, the location, radius, and color of the sphere vary accordingly.

There is a danger in using macros as functions. In a traditional programming language function, the result to be returned is actually assigned to a temporary variable and the invoking code treats it as a variable of a given type. However macro substitution may result in invalid or undesired syntax. The definition of the macro Interpolate above has an outermost set of parentheses. If those parentheses are omitted, it will not matter in the examples above, but what if you do this...

```
#declare Value = Interpolate(I,0,15,3.0,5.5)*15;
```

The end result is as if you had done...

```
#declare Value = P1+(T1+T/(T2-T1))*(P2-P1) * 15;
```

which is syntactically legal but not mathematically correct because the P1 term is not

multiplied. The parentheses in the original example solves this problem. The end result is as if you had done...

```
#declare Value = (P1+(T1+T/(T2-T1))*(P2-P1)) * 15;
```

which is correct.

Returning Values Via Parameters

Sometimes it is necessary to have a macro return more than one value or you may simply prefer to return a value via a parameter as is typical in traditional programming language procedures. POV-Ray macros are capable of returning values this way. The syntax for POV-Ray macro parameters says that the actual parameter may be an *IDENTIFIER* or an *RVALUE*. Values may only be returned via a parameter if the parameter is an *IDENTIFIER*. Parameters that are *RVALUES* are constant values that cannot return information. An *RVALUE* is anything that legally may appear to the right of an equals sign in a `#declare` or `#local` directive. For example consider the following trivial macro which rotates an object about the x-axis.

```
#macro Turn_Me(Stuff,Degrees)
  #declare Stuff = object{Stuff rotate x*Degrees}
#end
```

This attempts to re-declare the identifier `Stuff` as the rotated version of the object. However the macro might be invoked with `Turn_Me(box{0,1},30)` which uses a box object as an *RVALUE* parameter. This won't work because the box is not an identifier. You can however do this

```
#declare MyObject=box{0,1}
Turn_Me(MyObject,30)
```

The identifier `MyObject` now contains the rotated box.

See "Identifier Name Collisions" for a detailed discussion of how local identifiers, parameters, and global identifiers work when a local identifier has the same name as a previously declared identifier.

While it is obvious that `MyObject` is an identifier and `box{0,1}` is not, it should be noted that `Turn_Me(object{MyObject},30)` will not work because `object{MyObject}` is considered an object statement and is not a *pure* identifier. This mistake is more likely to be made with float identifiers versus float expressions. Consider these examples.

```
#declare Value=5.0;
MyMacro(Value)     //MyMacro can change the value of Value but...
MyMacro(+Value)    //This version and the rest are not lone
MyMacro(Value+0.0) // identifiers. They are float expressions
MyMacro(Value*1.0) // which cannot be changed.
```

Although all four invocations of `MyMacro` are passed the value 5.0, only the first may modify the value of the identifier.

Chapter 2

Scene Settings

2.1 Command-line Options

The reference section describes all command line switches and INI file keywords that are used to set the options of POV-Ray. It is supposed to be used as a reference for looking up things. It does not contain detailed explanations on how scenes are written or how POV-Ray is used. It just explains all features, their syntax, applications, limits, drawbacks, etc.

Options may be specified by switches or INI-style options. Almost all INI-style options have equivalent +/ - switches and most switches have equivalent INI-style option. The following sections give a detailed description of each POV-Ray option. It includes both the INI-style settings and the +/ - switches.

The notation and terminology used is described in the tables below.

`Keyword=bool`	Turn `Keyword` on if bool equals `true`, `yes`, `on` or `1` and Turn it off if it is any other value.
`Keyword=true`	Do this option if `true`, `yes`, `on` or `1` is specified.
`Keyword=false`	Do this option if `false`, `no`, `off` or `0` is specified.
`Keyword=filename`	Set `Keyword` to filename where filename is any valid file name. **Note:** some options prohibit the use of any of the above `true` or `false` values as a file name. They are noted in later sections.
n	Any integer such as in `+W320`
n.n	Any float such as in `Clock=3.45`
0.n	Any float < 1.0 even if it has no leading 0
s	Any string of text
x or y	Any single character
path	Any directory name, drive optional, no final path separator ("\" or "/", depending on the operating system)

Table 2.1:

Unless otherwise specifically noted, you may assume that either a plus or minus sign before a switch will produce the same results.

2.1.1 Animation Options

Internal animation loop, automatic output file name numbering and the ability to shell out to the operating system to external utilities which can assemble individual frames into an animation, greatly improved the animation capability. The internal animation loop is simple yet flexible. You may still use external programs or batch files to create animations without the internal loop.

External Animation Loop

Clock=n.n	Sets clock float identifier to n.n
+Kn.n	Same as Clock=n.n

Table 2.2:

The Clock=*n.n* option or the +K*n.n* switch may be used to pass a single float value to the program for basic animation. The value is stored in the float identifier clock. If an object had a `rotate <0,clock,0>` attached then you could rotate the object by different amounts over different frames by setting +K10.0,+K20.0... etc. on successive renderings. It is up to the user to repeatedly invoke POV-Ray with a different Clock value and a different Output_File_Name for each frame.

Internal Animation Loop

Initial_Frame=n	Sets initial frame number to n
Final_Frame=n	Sets final frame number to n
Initial_Clock=n.n	Sets initial clock value to n.n
Final_Clock=n.n	Sets final clock value to n.n
+KFIn	Same as Initial_Frame=n
+KFFn	Same as Final_Frame=n
+KIn.n	Same as Initial_Clock=n.n
+KFn.n	Same as Final_Clock=n.n

Table 2.3:

The internal animation loop relieves the user of the task of generating complicated sets of batch files to invoke POV-Ray multiple times with different settings. While the multitude of options may look intimidating, the clever set of default values means that you will probably only need to specify the Final_Frame=*n* or the +KFF*n* option to specify the number of frames. All other values may remain at their defaults.

Any Final_Frame setting other than -1 will trigger POV-Ray's internal animation loop. For example Final_Frame=10 or +KFF10 causes POV-Ray to render your scene 10 times. If you specified Output_File_Name=file.tga then each frame would be output as file01.tga, file02.tga, file03.tga etc. The number of zero-padded digits

in the file name depends upon the final frame number. For example `+KFF100` would generate `file001.tga` through `file100.tga`. The frame number may encroach upon the file name. On MS-DOS with an eight character limit, `myscene.pov` would render to `mysce001.tga` through `mysce100.tga`.

The default `Initial_Frame=1` will probably never have to be changed. You would only change it if you were assembling a long animation sequence in pieces. One scene might run from frame 1 to 50 and the next from 51 to 100. The `Initial_Frame=`*n* or `+KFI`*n* option is for this purpose.

Note: if you wish to render a subset of frames such as 30 through 40 out of a 1 to 100 animation, you should not change `Initial_Frame` or `Final_Frame`. Instead you should use the subset commands described in section "Subsets of Animation Frames".

Unlike some animation packages, the action in POV-Ray animated scenes does not depend upon the integer frame numbers. Rather you should design your scenes based upon the float identifier `clock`. By default, the clock value is 0.0 for the initial frame and 1.0 for the final frame. All other frames are interpolated between these values. For example if your object is supposed to rotate one full turn over the course of the animation, you could specify `rotate 360*clock*y`. Then as clock runs from 0.0 to 1.0, the object rotates about the y-axis from 0 to 360 degrees.

The major advantage of this system is that you can render a 10 frame animation or a 100 frame or 500 frame or 329 frame animation yet you still get one full 360 degree rotation. Test renders of a few frames work exactly like final renders of many frames.

In effect you define the motion over a continuous float valued parameter (the clock) and you take discrete samples at some fixed intervals (the frames). If you take a movie or video tape of a real scene it works the same way. An object's actual motion depends only on time. It does not depend on the frame rate of your camera.

Many users have already created scenes for POV-Ray 2 that expect clock values over a range other than the default 0.0 to 1.0. For this reason we provide the `Initial_Clock=`*n.n* or `+KI`*n.n* and `Final_Clock=`*n.n* or `+KF`*n.n* options. For example to run the clock from 25.0 to 75.0 you would specify `Initial_Clock=25.0` and `Final_Clock=75.0`. Then the clock would be set to 25.0 for the initial frame and 75.0 for the final frame. In-between frames would have clock values interpolated from 25.0 through 75.0 proportionally.

Users who are accustomed to using frame numbers rather than clock values could specify `Initial_Clock=1.0` and `Final_Clock=10.0` and `Frame_Final=10` for a 10 frame animation.

For new scenes, we recommend you do not change the `Initial_Clock` or `Final_Clock` from their default 0.0 to 1.0 values. If you want the clock to vary over a different range than the default 0.0 to 1.0, we recommend you handle this inside your scene file as follows...

```
#declare Start    = 25.0;
#declare End      = 75.0;
#declare My_Clock = Start+(End-Start)*clock;
```

Then use `My_Clock` in the scene description. This keeps the critical values 25.0 and 75.0 in your .pov file.

Note: more details concerning the inner workings of the animation loop are in the

section on shell-out operating system commands in section "Shell-out to Operating System".

Subsets of Animation Frames

`Subset_Start_Frame=n`	Set subset starting frame to n
`Subset_Start_Frame=0.n`	Set subset starting frame to n percent
`Subset_End_Frame=n`	Set subset ending frame to n
`Subset_End_Frame=0.n`	Set subset ending frame to n percent
`+SFn` or `+SF0.n`	Same as `Subset_Start_Frame`
`+EFn` or `+EF0.n`	Same as `Subset_End_Frame`

Table 2.4:

When creating a long animation, it may be handy to render only a portion of the animation to see what it looks like. Suppose you have 100 frames but only want to render frames 30 through 40. If you set `Initial_Frame=30` and `Final_Frame=40` then the clock would vary from 0.0 to 1.0 from frames 30 through 40 rather than 0.30 through 0.40 as it should. Therefore you should leave `Initial_Frame=1` and `Final_Frame=100` and use `Subset_Start_Frame=30` and `Subset_End_Frame=40` to selectively render part of the scene. POV-Ray will then properly compute the clock values.

Usually you will specify the subset using the actual integer frame numbers however an alternate form of the subset commands takes a float value in the range *0.0 <=n.nnn <=1.0* which is interpreted as a fraction of the whole animation. For example, `Subset_Start_Frame=0.333` and `Subset_End_Frame=0.667` would render the middle 1/3rd of a sequence regardless of the number of frames.

Cyclic Animation

`Cyclic_Animation=bool`	Turn cyclic animation on/off
`+KC`	Turn cyclic animation on
`-KC`	Turn cyclic animation off

Table 2.5:

Many computer animation sequences are designed to be run in a continuous loop. Suppose you have an object that rotates exactly 360 degrees over the course of your animation and you did `rotate 360*clock*y` to do so. Both the first and last frames would be identical. Upon playback there would be a brief one frame jerkiness. To eliminate this problem you need to adjust the clock so that the last frame does not match the first. For example a ten frame cyclic animation should not use clock 0.0 to 1.0. It should run from 0.0 to 0.9 in 0.1 increments. However if you change to 20 frames it should run from 0.0 to 0.95 in 0.05 increments. This complicates things because you would have to change the final clock value every time you changed `Final_Frame`. Setting `Cyclic_Animation=on` or using `+KC` will cause POV-Ray to automatically adjust the final clock value for cyclic animation regardless of how many total frames. The default value for this setting is off.

Field Rendering

`Field_Render=bool`	Turn field rendering on/off
`Odd_Field=bool`	Set odd field flag
`+UF`	Turn field rendering on
`-UF`	Turn field rendering off
`+UO`	Set odd field flag on
`-UO`	Set odd field flag off

Table 2.6:

Field rendering is sometimes used for animations when the animation is being output for television. TVs only display alternate scan lines on each vertical refresh. When each frame is being displayed the fields are interlaced to give the impression of a higher resolution image. The even scan lines make up the even field, and are drawn first (i.e. scan lines 0, 2, 4, etc.), followed by the odd field, made up of the odd numbered scan lines are drawn afterwards. If objects in an animation are moving quickly, their position can change noticeably from one field to the next. As a result, it may be desirable in these cases to have POV-Ray render alternate fields at the actual field rate (which is twice the frame rate), rather than rendering full frames at the normal frame rate. This would save a great deal of time compared to rendering the entire animation at twice the frame rate, and then only using half of each frame.

By default, field rendering is not used. Setting `Field_Render=on` or using `+UF` will cause alternate frames in an animation to be only the even or odd fields of an animation. By default, the first frame is the even field, followed by the odd field. You can have POV-Ray render the odd field first by specifying `Odd_Field=on`, or by using the `+UO` switch.

2.1.2 General Output Options

Height and Width of Output

`Height=n`	Sets screen height to n pixels
`Width=n`	Sets screen width to n pixels
`+Hn`	Same as `Height=n`
`+Wn`	Same as `Width=n`

Table 2.7:

These switches set the height and width of the image in pixels. This specifies the image size for file output. The preview display, if on, will generally attempt to pick a video mode to accommodate this size but the display settings do not in any way affect the resulting file output.

Partial Output Options

When doing test rendering it is often convenient to define a small, rectangular subsection of the whole screen so you can quickly check out one area of the image. The

Start_Column=n	Set first column to n pixels
Start_Column=0.n	Set first column to n percent of width
+SCn or +SC0.n	Same as Start_Column
Start_Row=n	Set first row to n pixels
Start_Row=0.n	Set first row to n percent of height
+SRn or +Sn	Same as Start_Row=n
+SR0.n or +S0.n	Same as Start_Row=0.n
End_Column=n	Set last column to n pixels
End_Column=0.n	Set last column to n percent of width
+ECn or +EC0.n	Same as End_Column
End_Row=n	Set last row to n pixels
End_Row=0.n	Set last row to n percent of height
+ERn or +En	Same as End_Row=n
+ER0.n or +E0.n	Same as End_Row=0.n

Table 2.8:

Start_Row, End_Row, Start_Column and End_Column options allow you to define the subset area to be rendered. The default values are the full size of the image from (1,1) which is the upper left to (w,h) on the lower right where w and h are the Width=*n* and Height=*n* values you have set.

Note: if the number specified is greater than 1 then it is interpreted as an absolute row or column number in pixels. If it is a decimal value between 0.0 and 1.0 then it is interpreted as a percent of the total width or height of the image.

For example: Start_Row=0.75 and Start_Column=0.75 starts on a row 75% down from the top at a column 75% from the left. Thus it renders only the lower-right 25% of the image regardless of the specified width and height.

The +SR, +ER, +SC and +EC switches work in the same way as the corresponding INI-style settings for both absolute settings or percentages. Early versions of POV-Ray allowed only start and end rows to be specified with +S*n* and +E*n* so they are still supported in addition to +SR and +ER.

When rendering a subset of *columns* (+sc/+ec) POV-Ray generates a full width image and fills the not rendered columns with black pixels. This should not be a problem for any image reading program no matter what file format is used.

when rendering a subset of *rows* (+sr/+er) POV-Ray writes the full height into the image file header and only writed those lines into the image that are rendered. This can cause problems with image reading programs that are not checking the file while reading and just read over the end.

if POV-Ray wrote the actual height of the partial image into the image header there would be no way to continue the trace in a later run.

Interrupting Options

On some operating systems once you start a rendering you must let it finish. The Test_Abort=on option or +X switch causes POV-Ray to test the keyboard for keypress. If you have pressed a key, it will generate a controlled user abort. Files will be flushed

Test_Abort=bool	Turn test for user abort on/off
+X	Turn test abort on
-X	Turn test abort off
Test_Abort_Count=n	Set to test for abort every n pixels
+Xn	Set to test for abort every n pixels on
-Xn	Set to test for abort off (in future test every n pixels)

Table 2.9:

and closed but only data through the last full row of pixels is saved. POV-Ray exits with an error code 2 (normally POV-Ray returns 0 for a successful run or 1 for a fatal error).

When this option is on, the keyboard is polled on every line while parsing the scene file and on every pixel while rendering. Because polling the keyboard can slow down a rendering, the Test_Abort_Count=*n* option or +X*n* switch causes the test to be performed only every *n* pixels rendered or scene lines parsed.

Resuming Options

Continue_Trace=bool	Sets continued trace on/off
+C	Sets continued trace on
-C	Sets continued trace off
Create_Ini=file	Generate an INI file to file
Create_Ini=true	Generate file.ini where file is scene name.
Create_Ini=false	Turn off generation of previously set file.ini
+GIfile	Same as Create_Ini=file

Table 2.10:

If you abort a render while it's in progress or if you used the End_Row option to end the render prematurely, you can use Continue_Trace=on or +C option to continue the render later at the point where you left off. This option reads in the previously generated output file, displays the partial image rendered so far, then proceeds with the ray-tracing. This option cannot be used if file output is disabled with Output_to_file=off or -F.

The Continue_Trace option may not work if the Start_Row option has been set to anything but the top of the file, depending on the output format being used. Also POV-Ray cannot continue the file once it has been opened and saved again by any program

POV-Ray tries to figure out where to resume an interrupted trace by reading any previously generated data in the specified output file. All file formats contain the image size, so this will override any image size settings specified. Some file formats (namely TGA and PNG) also store information about where the file started (i. e. +SC*n* and +SR*n* options), alpha output +UA, and bit-depth +FN*n*, which will override these settings. It is up to the user to make sure that all other options are set the same as the original render.

The Create_Ini option or +GI switch provides an easy way to create an INI file with all of the rendering options, so you can re-run files with the same options, or ensure you have all the same options when resuming. This option creates an INI file with every

option set at the value used for that rendering. This includes default values which you have not specified. For example if you run POV-Ray with...

```
POVRAY +Isimple.pov MYOPTS +GIrerun.ini MOREOPTS
```

POV-Ray will create a file called `rerun.ini` with all of the options used to generate this scene. The file is not written until all options have been processed. This means that in the above example, the file will include options from both `myopts.ini` and `moreopts.ini` despite the fact that the `+GI` switch is specified between them. You may now re-run the scene with...

```
POVRAY RERUN
```

or resume an interrupted trace with

```
POVRAY RERUN +C
```

If you add other switches with the `rerun.ini` reference, they will be included in future re-runs because the file is re-written every time you use it.

The `Create_Ini` option is also useful for documenting how a scene was rendered. If you render `waycool.pov` with `Create_Ini=on` then it will create a file `waycool.ini` that you could distribute along with your scene file so other users can exactly re-create your image.

2.1.3 Display Output Options

Display Hardware Settings

`Display=bool`	Turns graphic display on/off
`+D`	Turns graphic display on
`-D`	Turns graphic display off
`Video_Mode=x`	Set video mode to x; does not affect on/off
`+Dx`	Set display on; Set mode to x
`-Dx`	Set display off; but for future use mode x
`Palette=y`	Set display palette to y; does not affect on/off
`+Dxy`	Set display on; Set mode x; Set palette y
`-Dxy`	Set display off; use mode x, palette y in future
`Display_Gamma=n.n`	Sets the display gamma to n.n

Table 2.11:

The `Display=on` or `+D` switch will turn on the graphics display of the image while it is being rendered. Even on some non-graphics systems, POV-Ray may display an 80 by 24 character *"ASCII-Art"* version of your image. Where available, the display may be full, 24-bit true color. Setting `Display=off` or using the `-D` switch will turn off the graphics display which is the default.

On the Windows platform, the default is `Display=on`. Turning display off does not, of course, turn off the actual video display. Instead, POV-Ray will not open the output window that it normally shows a render in.

The `Video_Mode=`*x* option sets the display mode or hardware type chosen where *x* is a single digit or letter that is machine dependent. Generally `Video_Mode=0` means the

default or an auto-detected setting should be used. When using switches, this character immediately follows the switch. For example the +D0 switch will turn on the graphics display in the default mode.

The Palette=*y* option selects the palette to be used. Typically the single character parameter *y* is a digit which selects one of several fixed palettes or a letter such G for gray scale, H for 15-bit or 16-bit high color or T for 24-bit true color. When using switches, this character is the 2nd character after the switch. For example the +D0T switch will turn on the graphics display in the default mode with a true color palette. The Display_Gamma=*n.n* setting is not available as a command-line switch.

The Display_Gamma setting overcomes the problem of images (whether ray-traced or not) having different brightness when being displayed on different monitors, different video cards, and under different operating systems.

Note: the Display_Gamma is a setting based on your computer's display hardware, and should be set correctly once and not changed.

The Display_Gamma INI setting works in conjunction with the new assumed_gamma global setting to ensure that POV scenes and the images they create look the same on all systems. See section "Assumed_Gamma" which describes the assumed_gamma global setting and describes gamma more thoroughly.

While the Display_Gamma can be different for each system, there are a few general rules that can be used for setting Display_Gamma if you don't know it exactly. If the Display_Gamma keyword does not appear in the INI file, POV-Ray assumes that the display gamma is 2.2. This is because most PC monitors have a gamma value in the range 1.6 to 2.6 (newer models seem to have a lower gamma value). Mac has the ability to do gamma correction inside the system software (based on a user setting in the gamma control panel). If the gamma control panel is turned off, or is not available, the default Macintosh system gamma is 1.8. Many newer PC graphics cards can do hardware gamma correction and should use the current Display_Gamma setting, usually 1.0.

Setting your Display Gamma

The following gamma test image can be used to help you set your Display_Gamma accurately.

Before viewing the gamma image darken the room and set the monitor brightness and contrast to maximum. While viewing a black screen, lower the brightness gradually until the "background" is no longer noticeable (ie when it just fades from view). This may be difficult on monitors that use overscanning, unless you change the viewable area settings.

Now, lower the contrast until the alternating white and black bars on the left edge of each column are equal in width. This is trying to get a 50% gray by using half white and half black. If this is not possible, choose a contrast setting which is about in the middle. While viewing the image from a distance, or with squinted eyes, one of the numbered "swatches" will best match the gray value approximated by the white and black bars. The number in this "swatch" is your display's actual gamma value.

Normal display gamma values are in the range 2.0 to 2.6. If your monitor is usually used in a dim environment, we often use a gamma value that is 15% - 25% lower than

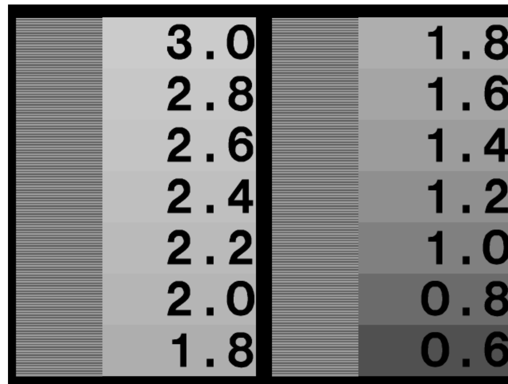

Figure 2.1: Display gamma test image.

the actual display gamma to give the images more contrast. Some systems, such as Macs and SGIs, already do gamma correction, so they may have display gammas of 1.0 or 1.8.

For scene files that do not contain an `assumed_gamma` global setting the INI file option `Display_Gamma` will not have any affect on the preview output of POV-Ray or for most output file formats. However, the `Display_Gamma` value is used when creating PNG format output files, and also when rendering the POV-Ray example files (because they have an `assumed_gamma`), so it should still be correctly set for your system to ensure proper results.

Display Related Settings

`Pause_When_Done=`bool	Sets pause when done on/off
`+P`	Sets pause when done on
`-P`	Sets pause when done off
`Verbose=`bool	Set verbose messages on/off
`+V`	Set verbose messages on
`-V`	Set verbose messages off
`Draw_Vistas=`bool	Turn draw vistas on/off
`+UD`	Turn draw vistas on
`-UD`	Turn draw vistas off

Table 2.12:

On some systems, when the image is complete, the graphics display is cleared and POV-Ray switches back into text mode to print the final statistics and to exit. Normally when the graphics display is on, you want to look at the image awhile before continuing. Using `Pause_When_Done=on` or `+P` causes POV-Ray to pause in graphics mode until you press a key to continue. The default is not to pause (`-P`).

When the graphics display is not used, it is often desirable to monitor progress of the rendering. Using `Verbose=on` or `+V` turns on verbose reporting of your rendering progress. This reports the number of the line currently being rendered, the elapsed time for the current frame and other information. On some systems, this textual information

can conflict with the graphics display. You may need to turn this off when the display is on. The default setting is off (-V).

The option Draw_Vistas=on or +UD was originally a debugging help for POV-Ray's vista buffer feature but it was such fun we decided to keep it. Vista buffering is a spatial sub-division method that projects the 2-D extents of bounding boxes onto the viewing window. POV-Ray tests the 2-D x, y pixel location against these rectangular areas to determine quickly which objects, if any, the viewing ray will hit. This option shows you the 2-D rectangles used. The default setting is off (-UD) because the drawing of the rectangles can take considerable time on complex scenes and it serves no critical purpose. See section "Automatic Bounding Control" for more details.

Mosaic Preview

Preview_Start_Size=n	Set mosaic preview start size to n
+SPn	Same as Preview_Start_Size=n
Preview_End_Size=n	Set mosaic preview end size to n
+EPn	Same as Preview_End_Size=n

Table 2.13:

Typically, while you are developing a scene, you will do many low resolution test renders to see if objects are placed properly. Often this low resolution version doesn't give you sufficient detail and you have to render the scene again at a higher resolution. A feature called *"mosaic preview"* solves this problem by automatically rendering your image in several passes.

The early passes paint a rough overview of the entire image using large blocks of pixels that look like mosaic tiles. The image is then refined using higher resolutions on subsequent passes. This display method very quickly displays the entire image at a low resolution, letting you look for any major problems with the scene. As it refines the image, you can concentrate on more details, like shadows and textures. You don't have to wait for a full resolution render to find problems, since you can interrupt the rendering early and fix the scene, or if things look good, you can let it continue and render the scene at high quality and resolution.

To use this feature you should first select a Width and Height value that is the highest resolution you will need. Mosaic preview is enabled by specifying how big the mosaic blocks will be on the first pass using Preview_Start_Size=*n* or +SP*n*. The value n should be a number greater than zero that is a power of two (1, 2, 4, 8, 16, 32, etc.) If it is not a power of two, the nearest power of two less than n is substituted. This sets the size of the squares, measured in pixels. A value of 16 will draw every 16th pixel as a 16*16 pixel square on the first pass. Subsequent passes will use half the previous value (such as 8*8, 4*4 and so on.)

The process continues until it reaches 1*1 pixels or until it reaches the size you set with Preview_End_Size=*n* or +EP*n*. Again the value n should be a number greater than zero that is a power of two and less than or equal to Preview_Start_Size. If it is not a power of two, the nearest power of two less than n is substituted. The default ending value is 1. If you set Preview_End_Size to a value greater than 1 the mosaic passes will end before reaching 1*1, but POV-Ray will always finish with a 1*1. For

example, if you want a single 8*8 mosaic pass before rendering the final image, set `Preview_Start_Size=8` and `Preview_End_Size=8`.

No file output is performed until the final 1*1 pass is reached. Although the preliminary passes render only as many pixels as needed, the 1*1 pass re-renders every pixel so that anti-aliasing and file output streams work properly. This makes the scene take up to 25% longer than the regular 1*1 pass to render, so it is suggested that mosaic preview not be used for final rendering. Also, the lack of file output until the final pass means that renderings which are interrupted before the 1*1 pass can not be resumed without starting over from the beginning.

2.1.4 File Output Options

`Output_to_File`=bool	Sets file output on/off
`+F`	Sets file output on (use default type)
`-F`	Sets file output off

Table 2.14:

By default, POV-Ray writes an image file to disk. When you are developing a scene and doing test renders, the graphic preview may be sufficient. To save time and disk activity you may turn file output off with `Output_to_File=off` or `-F`.

Output File Type

`Output_File_Type=x`	Sets file output format to x
`+Fxn`	Sets file output on; sets format x, depth n
`-Fxn`	Sets file output off; but in future use format x, depth n
`Output_Alpha`=bool	Sets alpha output on/off
`+UA`	Sets alpha output on
`-UA`	Sets alpha output off
`Bits_Per_Color`=n	Sets file output bits/color to n

Table 2.15:

The default type of image file depends on which platform you are using. MS-DOS and most others default to 24-bit uncompressed Targa. Windows defaults to 'sys', which is 24-bit BMP. See your platform-specific documentation to see what your default file type is. You may select one of several different file types using `Output_File_Type=`*x* or `+F`*x* where *x* is one of the following...

..	`C`	**C**ompressed Targa-24 format (RLE, run length encoded)
..	`N`	**PNG** (portable network graphics) format
..	`P`	Unix **PPM** format
..	`S`	**S**ystem-specific such as Mac Pict or Windows BMP
..	`T`	Uncompressed **T**arga-24 format

Table 2.16:

Note: the obsolete +FD dump format and +FR raw format have been dropped because they were rarely used and no longer necessary. PPM, PNG, and system specific formats have been added. PPM format images are uncompressed, and have a simple text header, which makes it a widely portable image format. PNG is an image format designed not only to replace GIF, but to improve on its shortcomings. PNG offers the highest compression available without loss for high quality applications, such as raytracing. The system specific format depends on the platform used and is covered in the appropriate system specific documentation.

Most of these formats output 24 bits per pixel with 8 bits for each of red, green and blue data. PNG and PPM allow you to optionally specify the output bit depth from 5 to 16 bits for each of the red, green, and blue colors, giving from 15 to 48 bits of color information per pixel. The default output depth for all formats is 8 bits/color (16 million possible colors), but this may be changed for PNG and PPM format files by setting Bits_Per_Color=*n* or by specifying +FN*n* or +FP*n*, where n is the desired bit depth.

Specifying a smaller color depth like 5 bits/color (32768 colors) may be enough for people with 8- or 16-bit (256 or 65536 color) displays, and will improve compression of the PNG file. Higher bit depths like 10 or 12 may be useful for video or publishing applications, and 16 bits/color is good for grayscale height field output (See section ”Height Field” for details on height fields).

Targa format also allows 8 bits of alpha transparency data to be output, while PNG format allows 5 to 16 bits of alpha transparency data, depending on the color bit depth as specified above. You may turn this option on with Output_Alpha=on or +UA. The default is off or -UA.

The alpha channel stores a transparency value for each pixel, just like there is also stored a value for red green and blue light for each pixel. In POV-Ray, when the alpha channel is turned on, all areas of the image where the background is partly or fully visible will be partly or fully transparent. Refractions of the background will also be transparent, but not reflections. Also anti-aliasing is taken into account

The philosophy of the alpha channel feature in POV-Ray is that the background color should not be present in the color of the image when the alpha channel is used. Instead, the amount of visible background is kept in the alpha and *only* in the alpha channel. That ensures that images look correct when viewed with the alpha channel.

See section ”Using the Alpha Channel” for further details on using transparency in imagemaps in your scene.

In addition to support for variable bit-depths, alpha channel, and grayscale formats, PNG files also store the Display_Gamma value so the image displays properly on all systems (see section ”Display Hardware Settings”). The hf_gray_16 global setting, as described in section ”HF_Gray_16” will also affect the type of data written to the output file.

Output File Name

The default output filename is created from the scene name and need not be specified. The scene name is the input name with all drive, path, and extension information

Output_File_Name=file	Sets output file to file
+Ofile	Same as Output_File_Name=file

Table 2.17:

stripped. For example if the input file name is c:\povray3\mystuff\myfile.pov the scene name is myfile. The proper extension is appended to the scene name based on the file type. For example myfile.tga or myfile.png might be used.

You may override the default output name using Output_File_Name=*file* or +O*file*. For example:

```
Input_File_Name=myinput.pov
Output_File_Name=myoutput.tga
```

If an output file name of "-" is specified (a single minus sign), then the image will be written to standard output, usually the screen. The output can then be piped into another program or to a GUI if desired.

If the file specified is actually a path or directory or folder name and not a file name, then the default output name is used but it is written to the specified directory. For example:

```
Input_File_Name=myscene.pov
Output_File_Name=c:\povray3\myimages\
```

This will create c:\povray3\myimages\myscene.tga as the output file.

Output File Buffer

The output-file buffer options Buffer_Output and Buffer_Size are removed per POV-Ray 3.51

Note: the options are still accepted, but ignored, in order to be backward compatible with old INI files.

CPU Utilization Histogram

The CPU utilization histogram is a way of finding out where POV-Ray is spending its rendering time, as well as an interesting way of generating heightfields. The histogram splits up the screen into a rectangular grid of blocks. As POV-Ray renders the image, it calculates the amount of time it spends rendering each pixel and then adds this time to the total rendering time for each grid block. When the rendering is complete, the histogram is a file which represents how much time was spent computing the pixels in each grid block.

Not all versions of POV-Ray allow the creation of histograms. The histogram output is dependent on the file type and the system that POV-Ray is being run on.

Histogram_Type=y	Set histogram type to y (Turn off if type is 'X')
+HTy	Same as Histogram_Type=y

Table 2.18:

File Type

The histogram output file type is nearly the same as that used for the image output file types in "Output File Type". The available histogram file types are as follows.

+HTC	Comma separated values (CSV) often used in spreadsheets
+HTN	PNG (portable network graphics) format grayscale
+HTP	Unix PPM format
+HTS	System-specific such as Mac Pict or Windows BMP
+HTT	Uncompressed Targa-24 format (TGA)
+HTX	No histogram file output is generated

Table 2.19:

Note: +HTC does not generate a compressed Targa-24 format output file but rather a text file with a comma-separated list of the time spent in each grid block, in left-to-right and top-to bottom order. The units of time output to the CSV file are system dependent. See the system specific documentation for further details on the time units in CSV files.

The Targa and PPM format files are in the POV heightfield format (see "Height Field"), so the histogram information is stored in both the red and green parts of the image, which makes it unsuitable for viewing. When used as a height field, lower values indicate less time spent calculating the pixels in that block, while higher indicate more time spent in that block.

PNG format images are stored as grayscale images and are useful for both viewing the histogram data as well as for use as a heightfield. In PNG files, the darker (lower) areas indicate less time spent in that grid block, while the brighter (higher) areas indicate more time spent in that grid block.

File Name

Histogram_Name=file	Set histogram name to file
+HNfile	Same as Histogram_Name=file

Table 2.20:

The histogram file name is the name of the file in which to write the histogram data. If the file name is not specified it will default to histogram.ext, where ext is based on the file type specified previously.

Note: that if the histogram name is specified the file name extension should match the file type.

Histogram_Grid_Size=nn.mm	Set histogram grid to nn by mm
+HSnn.mm	Same as Histogram_Grid_Size=nn.mm

Table 2.21:

Grid Size

The histogram grid size gives the number of times the image is split up in both the horizontal and vertical directions. For example

```
povray +Isample +W640 +H480 +HTN +HS160.120 +HNhistogram.png
```

will split the image into 160*120 grid blocks, each of size 4*4 pixels, and output a PNG file, suitable for viewing or for use as a heightfield. Smaller numbers for the grid size mean more pixels are put into the same grid block. With CSV output, the number of values output is the same as the number of grid blocks specified. For the other formats the image size is identical to the rendered image rather than the specified grid size, to allow easy comparison between the histogram and the rendered image. If the histogram grid size is not specified, it will default to the same size as the image, so there will be one grid block per pixel.

Note: on systems that do task-switching or multi-tasking the histogram may not exactly represent the amount of time POV-Ray spent in a given grid block since the histogram is based on real time rather than CPU time. As a result, time may be spent for operating system overhead or on other tasks running at the same time. This will cause the histogram to have speckling, noise or large spikes. This can be reduced by decreasing the grid size so that more pixels are averaged into a given grid block.

2.1.5 Scene Parsing Options

POV-Ray reads in your scene file and processes it to create an internal model of your scene. The process is called `parsing`. As your file is parsed other files may be read along the way. This section covers options concerning what to parse, where to find it and what version specific assumptions it should make while parsing it.

Constant

Declare=IDENTIFIER=FLOAT	Declares an identifier with a float value

Table 2.22:

You can now declare a constant in an INI file, and that constant will be available to the scene. Since INI file statements may also be laced on the command-line, you can therefore also declare on the command-line (though there is no switch for it).

```
Declare=MyValue=24
```

This would be the same as a `#declare MyValue=24;` in a scene file. The value on the right-hand side must be a constant float value.

A possible use could be switching off radiosity or photons from commandline:

```
--in INI-file / on command-line

Declare=RAD=0

--in scenefile

global_settings {
   #if (RAD)
      radiosity {
         ...
      }
   #end
}
```

Input File Name

Input_File_Name=file	Sets input file name to file
+Ifile	Same as Input_File_Name=file

Table 2.23:

Note: there may be no space between +I and file.

You will probably always set this option but if you do not the default input filename is object.pov. If you do not have an extension then .pov is assumed. On case-sensitive operating systems both .pov and .POV are tried. A full path specification may be used (on MS-DOS systems +Ic:\povray3\mystuff\myfile.pov is allowed for example). In addition to specifying the input file name this also establishes the *scene name*.

The scene name is the input name with drive, path and extension stripped. In the above example the scene name is myfile. This name is used to create a default output file name and it is referenced other places.

Note: as per version 3.5 you can now specify a POV file on the command-line without the use of the +i switch (i.e. it works the same way as specifying an INI file without a switch), the POV file then should be the last on the commandline.

If you use "-" as the input file name the input will be read from standard input. Thus you can pipe a scene created by a program to POV-Ray and render it without having a scene file.

Under MS-DOS you can try this feature by typing.

```
type ANYSCENE.POV | povray +I-
```

Include File Name

Include_Header=file	Sets primary include file name to file
+HIfile	Same as Include_Header=file

Table 2.24:

This option allows you to include a file as the first include file of a scene file. You can for example use this option to always include a specific set of default include files used by all your scenes.

Library Paths

`Library_Path=path`	Add path to list of library paths
`+Lpath`	Same as `Library_Path=path`

Table 2.25:

POV-Ray looks for files in the current directory. If it does not find a file it needs it looks in various other library directories which you specify. POV-Ray does not search your operating system path. It only searches the current directory and directories which you specify with this option. For example the standard include files are usually kept in one special directory. You tell POV-Ray to look there with...

```
Library_Path=c:\povray3\include
```

You must not specify any final path separators ("\" or "/") at the end.

Multiple uses of this option switch do not override previous settings. Up to twenty unique paths may be specified. If you specify the exact same path twice it is only counted once. The current directory will be searched first followed by the indicated library directories in the order in which you specified them.

Language Version

`Version=n.n`	Set initial language compatibility to version n.n
`+MVn.n`	Same as `Version=n.n`

Table 2.26:

As POV-Ray has evolved from version 1.0 through to today we have made every effort to maintain some amount of backwards compatibility with earlier versions. Some old or obsolete features can be handled directly without any special consideration by the user. Some old or obsolete features can no longer be handled at all. However *some* old features can still be used if you warn POV-Ray that this is an older scene. In the POV-Ray scene language you can use the `#version` directive to switch version compatibility to different settings. See section "The #version Directive" for more details about the language version directive. Additionally you may use the `Version=`*n.n* option or the `+MV`*n.n* switch to establish the *initial* setting. For example one feature introduced in 2.0 that was incompatible with any 1.0 scene files is the parsing of float expressions. Setting `Version=1.0` or using `+MV1.0` turns off expression parsing as well as many warning messages so that nearly all 1.0 files will still work. Naturally the default setting for this option is `Version=3.5`.

Note: some obsolete or re-designed features *are totally unavailable in the current version of POV-Ray REGARDLES OF THE VERSION SETTING.* Details on these features are noted throughout this documentation.

2.1.6 Shell-out to Operating System

Pre_Scene_Command=s	Set command before entire scene
Pre_Frame_Command=s	Set command before each frame
Post_Scene_Command=s	Set command after entire scene
Post_Frame_Command=s	Set command after each frame
User_Abort_Command=s	Set command when user aborts POV-Ray
Fatal_Error_Command=s	Set command when POV-Ray has fatal error

Table 2.27:

Note: no + or - switches are available for these options. They cannot be used from the command line. They may only be used from INI files.

POV-Ray offers you the opportunity to shell-out to the operating system at several key points to execute another program or batch file. Usually this is used to manage files created by the internal animation loop however the shell commands are available for any scene. The string *s* is a single line of text which is passed to the operating system to execute a program. For example

```
Post_Scene_Command=tga2gif -d -m myfile
```

would use the utility tga2gif with the -D and -M parameters to convert myfile.tga to myfile.gif after the scene had finished rendering.

Note: individual platforms may provide means of preventing shell-outs from occurring. For example, the Windows version provides a menu command to turn shell-outs off (which is the default setting for that platform). The reason for this (along with file I/O restrictions) is to attempt to prevent untrusted INI files from doing harm to your system.

String Substitution in Shell Commands

It could get cumbersome to change the Post_Scene_Command every time you changed scene names. POV-Ray can substitute various values into a command string for you. For example:

```
Post_Scene_Command=tga2gif -d -m \%s
```

POV-Ray will substitute the %s with the scene name in the command. The *scene name* is the Input_File_Name or +I setting with any drive, directory and extension removed. For example:

```
Input_File_Name=c:\povray3\scenes\waycool.pov
```

is stripped down to the scene name waycool which results in...

```
Post_Scene_Command=tga2gif -d -m waycool
```

In an animation it may be necessary to have the exact output file name with the frame number included. The string %o will substitute the output file name. Suppose you want to save your output files in a zip archive using the utility program pkzip. You could do...

```
Post_Frame_Command=pkzip -m \%s \%o
```

After rendering frame 12 of myscene.pov POV-Ray would shell to the operating system with

```
pkzip -m myscene mysce012.tga
```

The -M switch in pkzip moves mysce012.tga to myscene.zip and removes it from the directory. Note that %o includes frame numbers only when in an animation loop. During the Pre_Scene_Command and Post_Scene_Command there is no frame number so the original, unnumbered Output_File_Name is used. Any User_Abort_Command or Fatal_Error_Command not inside the loop will similarly give an unnumbered %o substitution.

Here is the complete list of substitutions available for a command string.

%o	Output file name with extension and embedded frame number if any
%s	Scene name derived by stripping path and ext from input name
%n	Frame number of this frame
%k	Clock value of this frame
%h	Height of image in pixels
%w	Width of image in pixels
%%	A single % sign.

Table 2.28:

Shell Command Sequencing

Here is the sequence of events in an animation loop. Non-animated scenes work the exact same way except there is no loop.

1. Process all INI file keywords and command line switches just once.
2. Open any text output streams and do Create_INI if any.
3. Execute Pre_Scene_Command if any.
4. Loop through frames (or just do once on non-animation).
   (a) Execute Pre_Frame_Command if any.
   (b) Parse entire scene file, open output file and read settings, turn on display, render the frame, destroy all objects, textures etc., close output file, close display.
   (c) Execute Post_Frame_Command if any.
   (d) Repeat above steps until all frames are done.
5. Execute Post_Scene_Command if any.
6. Finish

If the user interrupts processing the User_Abort_Command, if any, is executed. User aborts can only occur during the parsing and rendering parts of step (4b) above. If a fatal error occurs that POV-Ray notices the Fatal_Error_Command, if any, is executed. Sometimes an unforeseen bug or memory error could cause a total crash of the program in which case there is no chance to shell out. Fatal errors can occur just about anywhere

including during the processing of switches or INI files. If a fatal error occurs before POV-Ray has read the `Fatal_Error_Command` string then obviously no shell can occur.

Note: the entire scene is re-parsed for every frame. Future versions of POV-Ray may allow you to hold over parts of a scene from one frame to the next but for now it starts from scratch every time.

Note: that the `Pre_Frame_Command` occurs before the scene is parsed. You might use this to call some custom scene generation utility before each frame. This utility could rewrite your `.pov` or `.inc` files if needed. Perhaps you will want to generate new `.gif` or `.tga` files for image maps or height fields on each frame.

Shell Command Return Actions

`Pre_Scene_Return=s`	Set pre scene return actions
`Pre_Frame_Return=s`	Set pre frame return actions
`Post_Scene_Return=s`	Set post scene return actions
`Post_Frame_Return=s`	Set post frame return actions
`User_Abort_Return=s`	Set user abort return actions
`Fatal_Error_Return=s`	Set fatal return actions

Table 2.29:

Note: that no + or - switches are available for these options. They cannot be used from the command line. They may only be used from INI files.

Most operating systems allow application programs to return an error code if something goes wrong. When POV-Ray executes a shell command it can make use of this error code returned from the shell process and take some appropriate action if the code is zero or non-zero. POV-Ray itself returns such codes. It returns 0 for success, 1 for fatal error and 2 for user abort.

The actions are designated by a single letter in the different `..._Return=s` options. The possible actions are:

`I`	ignore the code
`S`	skip one step
`A`	all steps skipped
`Q`	quit POV-Ray immediately
`U`	generate a user abort in POV-Ray
`F`	generate a fatal error in POV-Ray

Table 2.30:

For example if your `Pre_Frame_Command` calls a program which generates your height field data and that utility fails then it will return a non-zero code. We would probably want POV-Ray to abort as well. The option `Pre_Frame_Return=F` will cause POV-Ray to do a fatal abort if the `Pre_Frame_Command` returns a non-zero code.

Sometimes a non-zero code from the external process is a good thing. Suppose you want to test if a frame has already been rendered. You could use the `S` action to skip this frame if the file is already rendered. Most utilities report an error if the file is not found. For example the command...

```
pkzip -V myscene mysce012.tga
```

tells pkzip you want to view the catalog of `myscene.zip` for the file `mysce012.tga`. If the file isn't in the archive `pkzip` returns a non-zero code.

However we want to skip if the file is found. Therefore we need to reverse the action so it skips on zero and doesn't skip on non-zero. To reverse the zero vs. non-zero triggering of an action precede it with a "-" sign (note a "!" will also work since it is used in many programming languages as a negate operator).

`Pre_Frame_Return=S` will skip if the code shows error (non-zero) and will proceed normally on no error (zero). `Pre_Frame_Return=-S` will skip if there is no error (zero) and will proceed normally if there is an error (non-zero).

The default for all shells is `I` which means that the return action is ignored no matter what. POV-Ray simply proceeds with whatever it was doing before the shell command. The other actions depend upon the context. You may want to refer back to the animation loop sequence chart in the previous section "Shell Command Sequencing". The action for each shell is as follows.

On return from any User_Abort_Command if there is an action triggered...

...and you have specified...	...then POV-Ray will..
`F`	Then turn this user abort into a fatal error. Do the `Fatal_Error_Command`, if any. Exit POV-Ray with error code 1.
`S`, `A`, `Q`, or `U`	Then proceed with the user abort. Exit POV-Ray with error code 2.

Table 2.31:

On return from any `Fatal_Error_Command` then POV-Ray will proceed with the fatal error no matter what. It will exit POV-Ray with error code 1.

On return from any `Pre_Scene_Command`, `Pre_Frame_Command`, `Post_Frame_Command` or `Post_Scene_Commands` if there is an action triggered...

...and you have specified...	...then POV-Ray will...
`F`	...turn this user abort into a fatal error. Do the `Fatal_Error_Command`, if any. Exit POV-Ray with error code 1.
`U`	...generate a user abort. Do the `User_Abort_Command`, if any. Exit POV-Ray with an error code 2.
`Q`	..quit POV-Ray immediately. Acts as though POV-Ray never really ran. Do no further shells, (not even a `Post_Scene_Command`) and exit POV-Ray with an error code 0.

Table 2.32:

On return from a `Pre_Scene_Command` if there is an action triggered...

...and you have specified...	...then POV-Ray will...
S	...skip rendering all frames. Acts as though the scene completed all frames normally. Do not do any Pre_Frame_Command or Post_Frame_Commands. Do the Post_Scene_Command, if any. Exit POV-Ray with error code 0. On the earlier chart this means skip step #4.
A	...skip all scene activity. Works exactly like Q quit. On the earlier chart this means skip to step #6. Acts as though POV-Ray never really ran. Do no further shells, (not even a Post_Scene_Command) and exit POV-Ray with an error code 0.

Table 2.33:

...and you have specified...	...then POV-Ray will...
S	...skip only this frame. Acts as though this frame never existed. Do not do the Post_Frame_Command. Proceed with the next frame. On the earlier chart this means skip steps (4b) and (4c) but loop back as needed in (4d).
A	...skip rendering this frame and all remaining frames. Acts as though the scene completed all frames normally. Do not do any further Post_Frame_Commands. Do the Post_Scene_Command, if any. Exit POV-Ray with error code 0. On the earlier chart this means skip the rest of step (4) and proceed at step (5).

Table 2.34:

On return from a `Pre_Frame_Command` if there is an action triggered...

On return from a `Post_Frame_Command` if there is an action triggered...

...and you have specified...	...then POV-Ray will...
`S` or `A`	...skip all remaining frames. Acts as though the scene completed all frames normally. Do not do any further `Post_Frame_Commands`. Do the `Post_Scene_Command`, if any. Exit POV-Ray with error code 0. On the earlier chart this means skip the rest of step (4) and proceed at step (5).

Table 2.35:

On return from any `Post_Scene_Command` if there is an action triggered and you have specified `S` or `A` then no special action occurs. This is the same as `I` for this shell command.

2.1.7 Text Output

Text output is an important way that POV-Ray keeps you informed about what it is going to do, what it is doing and what it did. The program splits its text messages into 7 separate streams. Some versions of POV-Ray color-codes the various types of text. Some versions allow you to scroll back several pages of messages. All versions allow you to turn some of these text streams off/on or to direct a copy of the text output to one or several files. This section details the options which give you control over text output.

Text Streams

There are seven distinct text streams that POV-Ray uses for output. On some versions each stream is designated by a particular color. Text from these streams are displayed whenever it is appropriate so there is often an intermixing of the text. The distinction is only important if you choose to turn some of the streams off or to direct some of the streams to text files. On some systems you may be able to review the streams separately in their own scroll-back buffer.

Here is a description of each stream.

Banner: This stream displays the program's sign-on banner, copyright, contributor's list, and some help screens. It cannot be turned off or directed to a file because most of this text is displayed before any options or switches are read. Therefore you cannot use an option or switch to control it. There are switches which display the help screens. They are covered in section ”Help Screen Switches”.

Debug: This stream displays debugging messages. It was primarily designed for developers but this and other streams may also be used by the user to display messages from within their scene files. See section ”Text Message Streams” for details on this feature. This stream may be turned off and/or directed to a text file.

Fatal: This stream displays fatal error messages. After displaying this text, POV-Ray will terminate. When the error is a scene parsing error, you may be shown several lines of scene text that leads up to the error. This stream may be turned off and/or directed to a text file.

Render: This stream displays information about what options you have specified to render the scene. It includes feedback on all of the major options such as scene name, resolution, animation settings, anti-aliasing and others. This stream may be turned off and/or directed to a text file.

Statistics: This stream displays statistics after a frame is rendered. It includes information about the number of rays traced, the length of time of the processing and other information. This stream may be turned off and/or directed to a text file.

Status: This stream displays one-line status messages that explain what POV-Ray is doing at the moment. On some systems this stream is displayed on a status line at the bottom of the screen. This stream cannot be directed to a file because there is generally no need to. The text displayed by the `Verbose` option or `+V` switch is output to this stream so that part of the status stream may be turned off.

Warning: This stream displays warning messages during the parsing of scene files and other warnings. Despite the warning, POV-Ray can continue to render the scene. You will be informed if POV-Ray has made any assumptions about your scene so that it can proceed. In general any time you see a warning, you should also assume that this means that future versions of POV-Ray will not allow the warned action. Therefore you should attempt to eliminate warning messages so your scene will be able to run in future versions of POV-Ray. This stream may be turned off and/or directed to a text file.

Console Text Output

You may suppress the output to the console of the debug, fatal, render, statistic or warning text streams. For example the `Statistic_Console=off` option or the `-GS` switch can turn off the statistic stream. Using `on` or `+GS` you may turn it on again. You may also turn all five of these streams on or off at once using the `All_Console` option or `+GA` switch.

Note: that these options take effect immediately when specified. Obviously any error or warning messages that might occur before the option is read are not be affected.

Directing Text Streams to Files

You may direct a copy of the text streams to a text file for the debug, fatal, render, statistic, or warning text streams. For example the `Statistic_File=`*s* option or the `+GS`*s* switch. If the string *s* is `true` or any of the other valid `true` strings then that stream is redirected to a file with a default name. Valid `true` values are `true`, `yes`, `on` or `1`. If the value is `false` the direction to a text file is turned off. Valid `false` values are `false`, `no`, `off` or `0`. Any other string specified turns on file output and the string is interpreted as the output file name.

Debug_Console=bool	Turn console display of debug info text on/off
+GD	Same as Debug_Console=On
-GD	Same as Debug_Console=Off
Fatal_Console=bool	Turn console display of fatal error text on/off
+GF	Same as Fatal_Console=On
-GF	Same as Fatal_Console=Off
Render_Console=bool	Turn console display of render info text on/off
+GR	Same as Render_Console=On
-GR	Same as Render_Console=Off
Statistic_Console=bool	Turn console display of statistic text on/off
+GS	Same as Statistic_Console=On
-GS	Same as Statistic_Console=Off
Warning_Console=bool	Turn console display of warning text on/off
+GW	Same as Warning_Console=On
-GW	Same as Warning_Console=Off
All_Console=bool	Turn on/off all debug, fatal, render, statistic and warning text to console.
+GA	Same as All_Console=On
-GA	Same as All_Console=Off

Table 2.36:

Debug_File=true	Echo debug info text to DEBUG.OUT
Debug_File=false	Turn off file output of debug info
Debug_File=file	Echo debug info text to file
+GDfile	Both Debug_Console=On, Debug_File=file
-GDfile	Both Debug_Console=Off, Debug_File=file
Fatal_File=true	Echo fatal text to FATAL.OUT
Fatal_File=false	Turn off file output of fatal
Fatal_File=file	Echo fatal info text to file
+GFfile	Both Fatal_Console=On, Fatal_File=file
-GFfile	Both Fatal_Console=Off, Fatal_File=file
Render_File=true	Echo render info text to RENDER.OUT
Render_File=false	Turn off file output of render info
Render_File=file	Echo render info text to file
+GRfile	Both Render_Console=On, Render_File=file
-GRfile	Both Render_Console=Off, Render_File=file
Statistic_File=true	Echo statistic text to STATS.OUT
Statistic_File=false	Turn off file output of statistics
Statistic_File=file	Echo statistic text to file
+GSfile	Both Statistic_Console=On, Statistic_File=file
-GSfile	Both Statistic_Console=Off, Statistic_File=file
Warning_File=true	Echo warning info text to WARNING.OUT
Warning_File=false	Turn off file output of warning info
Warning_File=file	Echo warning info text to file
+GWfile	Both Warning_Console=On, Warning_File=file
-GWfile	Both Warning_Console=Off, Warning_File=file
All_File=true	Echo all debug, fatal, render, statistic, and warning text to ALLTEXT.OUT
All_File=false	Turn off file output of all debug, fatal, render, statistic, and warning text.
All_File=file	Echo all debug, fatal, render, statistic, and warning text to file
+GAfile	Both All_Console=On, All_File=file
-GAfile	Both All_Console=Off, All_File=file

Table 2.37:

Similarly you may specify such a true, false or file name string after a switch such as +GS*file*. You may also direct all five streams to the same file using the All_File option or +GA switch. You may not specify the same file for two or more streams because POV-Ray will fail when it tries to open or close the same file twice.

Note: that these options take effect immediately when specified. Obviously any error or warning messages that might occur before the option is read will not be affected.

Warning Level

Warning_Level=n	Allows you to turn off classes of warnings.
+WLn	Same as Warning_Level=n

Table 2.38:

Level 0 turns off all warnings. Level 5 turns off all language version related warnings. The default is level 10 and it enables all warnings. All other levels are reserved and should not be specified.

Help Screen Switches

+H or +?	Show help screen 0 if this is the only switch

Table 2.39:

Note: there are no INI style equivalents to these options.

After displaying the help screens, POV-Ray terminates. Because some operating systems do not permit a question mark as a command line switch you may also use the +H switch.

Note: this switch is also used to specify the height of the image in pixels. Therefore the +H switch is only interpreted as a help switch if it is the only switch on the command line.

Graphical interface versions of POV-Ray such as Mac or Windows have extensive online help.

2.1.8 Tracing Options

There is more than one way to trace a ray. Sometimes there is a trade-off between quality and speed. Sometimes options designed to make tracing faster can slow things down. This section covers options that tell POV-Ray how to trace rays with the appropriate speed and quality settings.

Quality Settings

The Quality=*n* option or +Q*n* switch allows you to specify the image rendering quality. You may choose to lower the quality for test rendering and raise it for final renders. The

Quality=n	Set quality value to n (0 <= n <= 11)
+Qn	Same as Quality=n

Table 2.40:

quality adjustments are made by eliminating some of the calculations that are normally performed. For example settings below 4 do not render shadows. Settings below 8 do not use reflection or refraction. The duplicate values allow for future expansion. The values correspond to the following quality levels:

0, 1	Just show quick colors. Use full ambient lighting only. Quick colors are used only at 5 or below.
2, 3	Show specified diffuse and ambient light.
4	Render shadows, but no extended lights.
5	Render shadows, including extended lights.
6, 7	Compute texture patterns, compute photons
8	Compute reflected, refracted, and transmitted rays.
9, 10, 11	Compute media and radiosity

Table 2.41:

The default is 9 if not specified.

Automatic Bounding Control

Bounding=bool	Turn bounding on/off
+MB	Turn bounding on; Set threshold to 25 or previous amount
-MB	Turn bounding off
Bounding_Threshold=n	Set bound threshold to n
+MBn	Turn bounding on; bound threshold to n
-MBn	Turn bounding off; set future threshold to n
Light_Buffer=bool	Turn light buffer on/off
+UL	Turn light buffer on
-UL	Turn light buffer off
Vista_Buffer=bool	Turn vista buffer on/off
+UV	Turn vista buffer on
-UV	Turn vista buffer off

Table 2.42:

POV-Ray uses a variety of spatial sub-division systems to speed up ray-object intersection tests. The primary system uses a hierarchy of nested bounding boxes. This system compartmentalizes all finite objects in a scene into invisible rectangular boxes that are arranged in a tree-like hierarchy. Before testing the objects within the bounding boxes the tree is descended and only those objects are tested whose bounds are hit by a ray. This can greatly improve rendering speed. However for scenes with only a few objects the overhead of using a bounding system is not worth the effort. The Bounding=off option or -MB switch allows you to force bounding off. The default value is on.

The Bounding_Threshold=*n* or +MB*n* switch allows you to set the minimum number

of objects necessary before bounding is used. The default is +MB25 which means that if your scene has fewer than 25 objects POV-Ray will automatically turn bounding off because the overhead isn't worth it. Generally it's a good idea to use a much lower threshold like +MB5.

Additionally POV-Ray uses systems known as *vista buffers* and *light buffers* to further speed things up. These systems only work when bounding is on and when there are a sufficient number of objects to meet the bounding threshold. The vista buffer is created by projecting the bounding box hierarchy onto the screen and determining the rectangular areas that are covered by each of the elements in the hierarchy. Only those objects whose rectangles enclose a given pixel are tested by the primary viewing ray. The vista buffer can only be used with perspective and orthographic cameras because they rely on a fixed viewpoint and a reasonable projection (i. e. straight lines have to stay straight lines after the projection).

The light buffer is created by enclosing each light source in an imaginary box and projecting the bounding box hierarchy onto each of its six sides. Since this relies on a fixed light source, light buffers will not be used for area lights.

Reflected and transmitted rays do not take advantage of the light and vista buffer.

The default settings are Vista_Buffer=on or +UV and Light_Buffer=on or +UL. The option to turn these features off is available to demonstrate their usefulness and as protection against unforeseen bugs which might exist in any of these bounding systems.

In general, any finite object and many types of CSG of finite objects will properly respond to this bounding system. In addition blobs and meshes use an additional internal bounding system. These systems are not affected by the above switch. They can be switched off using the appropriate syntax in the scene file (see "Blob" and "Mesh" for details).

Text objects are split into individual letters that are bounded using the bounding box hierarchy. Some CSG combinations of finite and infinite objects are also automatically bound. The end result is that you will rarely need to add manual bounding objects as was necessary in earlier versions of POV-Ray unless you use many infinite objects.

Removing User Bounding

Remove_Bounds=bool	Turn unnecessary bounds removal on/off
+UR	Turn unnecessary bounds removal on
-UR	Turn unnecessary bounds removal off
Split_Unions=bool	Turn split bounded unions on/off
+SU	Turn split bounded unions on
-SU	Turn split bounded unions off

Table 2.43:

Early versions of POV-Ray had no system of automatic bounding or spatial sub-division to speed up ray-object intersection tests. Users had to manually create bounding boxes to speed up the rendering. Since version 3.0, POV-Ray has had more sophisticated automatic bounding than any previous version. In many cases the manual bounding on older scenes is slower than the new automatic systems. Therefore POV-Ray removes

manual bounding when it knows it will help. In rare instances you may want to keep manual bounding. Some older scenes incorrectly used bounding when they should have used clipping. If POV-Ray removes the bounds in these scenes the image will not look right. To turn off the automatic removal of manual bounds you should specify `Remove_Bounds=off` or use `-UR`. The default is `Remove_Bounds=on`.

One area where the jury is still out is the splitting of manually bounded unions. Unbounded unions are always split into their component parts so that automatic bounding works better. Most users do not bound unions because they know that doing so is usually slower. If you do manually bound a union we presume you really want it bound. For safety sake we do not presume to remove such bounds. If you want to remove manual bounds from unions you should specify `Split_Unions=on` or use `+SU`. The default is `Split_Unions=off`.

Anti-Aliasing Options

`Antialias=`bool	Turns anti-aliasing on/off
`+A`	Turns aa on with threshold 0.3 or previous amount
`-A`	Turns anti-aliasing off
`Sampling_Method=n`	Sets aa-sampling method (only `1` or `2` are valid)
`+AMn`	Same as `Sampling_Method=n`
`Antialias_Threshold=n.n`	Sets anti-aliasing threshold
`+An.n`	Sets aa on with aa-threshold at n.n
`-An.n`	Sets aa off (aa-threshold n.n in future)
`Jitter=`bool	Sets aa-jitter on/off
`+J`	Sets aa-jitter on with 1.0 or previous amount
`-J`	Sets aa-jitter off
`Jitter_Amount=n.n`	Sets aa-jitter amount to n.n. If n.n <= 0 aa-jitter is set off
`+Jn.n`	Sets aa-jitter on; jitter amount to n.n. If n.n <= 0 aa-jitter is set off
`-Jn.n`	Sets aa-jitter off (jitter amount n.n in future)
`Antialias_Depth=n`	Sets aa-depth (1 <= n <= 9)
`+Rn`	Same as `Antialias_Depth=n`

Table 2.44:

The ray-tracing process is in effect a discrete, digital sampling of the image with typically one sample per pixel. Such sampling can introduce a variety of errors. This includes a jagged, stair-step appearance in sloping or curved lines, a broken look for thin lines, moiré patterns of interference and lost detail or missing objects, which are so small they reside between adjacent pixels. The effect that is responsible for those errors is called *aliasing*.

Anti-aliasing is any technique used to help eliminate such errors or to reduce the neg-

ative impact they have on the image. In general, anti-aliasing makes the ray-traced image look *smoother*. The `Antialias=on` option or `+A` switch turns on POV-Ray's anti-aliasing system.

When anti-aliasing is turned on, POV-Ray attempts to reduce the errors by shooting more than one viewing ray into each pixel and averaging the results to determine the pixel's apparent color. This technique is called super-sampling and can improve the appearance of the final image but it drastically increases the time required to render a scene since many more calculations have to be done.

POV-Ray gives you the option to use one of two alternate super-sampling methods. The `Sampling_Method=`*n* option or `+AM`*n* switch selects either type `1` or type `2`. Selecting one of those methods does not turn anti-aliasing on. This has to be done by using the `+A` command line switch or `Antialias=on` option.

Type 1 is an adaptive, non-recursive, super-sampling method. It is *adaptive* because not every pixel is super-sampled. Type 2 is an adaptive and recursive super-sampling method. It is *recursive* because the pixel is sub-divided and sub-sub-divided recursively. The *adaptive* nature of type 2 is the variable depth of recursion.

In the default, non-recursive method (`+AM1`), POV-Ray initially traces one ray per pixel. If the color of a pixel differs from its neighbors (to the left or above) by at least the set threshold value then the pixel is super-sampled by shooting a given, fixed number of additional rays. The default threshold is 0.3 but it may be changed using the `Antialias_Threshold=`*n.n* option. When the switches are used, the threshold may optionally follow the `+A`. For example `+A0.1` turns anti-aliasing on and sets the threshold to 0.1.

The threshold comparison is computed as follows. If r1, g1, b1 and r2, g2, b2 are the rgb components of two pixels then the difference between pixels is computed by

```
diff = abs(r1-r2) + abs(g1-g2) + abs(b1-b2)
```

If this difference is greater than the threshold then both pixels are super-sampled. The rgb values are in the range from 0.0 to 1.0 thus the most two pixels can differ is 3.0. If the anti-aliasing threshold is 0.0 then every pixel is super-sampled. If the threshold is 3.0 then no anti-aliasing is done. Lower threshold means more anti-aliasing and less speed. Use anti-aliasing for your final version of a picture, not the rough draft. The lower the contrast, the lower the threshold should be. Higher contrast pictures can get away with higher tolerance values. Good values seem to be around 0.2 to 0.4.

When using the non-recursive method, the default number of super-samples is nine per pixel, located on a 3*3 grid. The `Antialias_Depth=`*n* option or `+R`*n* switch controls the number of rows and columns of samples taken for a super-sampled pixel. For example `+R4` would give 4*4=16 samples per pixel.

The second, adaptive, recursive super-sampling method starts by tracing four rays at the corners of each pixel. If the resulting colors differ more than the threshold amount additional samples will be taken. This is done recursively, i.e. the pixel is divided into four sub-pixels that are separately traced and tested for further subdivision. The advantage of this method is the reduced number of rays that have to be traced. Samples that are common among adjacent pixels and sub-pixels are stored and reused to avoid re-tracing of rays. The recursive character of this method makes the super-sampling

concentrate on those parts of the pixel that are more likely to need super-sampling (see figure below).

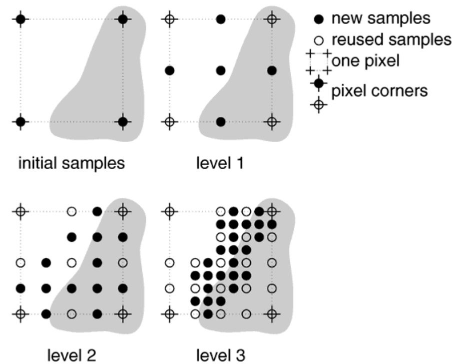

Figure 2.2: Example of how the recursive super-sampling works.

The maximum number of subdivisions is specified by the `Antialias_Depth=`*n* option or `+R`*n* switch. This is different from the adaptive, non-recursive method where the total number of super-samples is specified. A maximum number of *n* subdivisions results in a maximum number of samples per pixel that is given by the following table.

`+Rn`	Number of additional samples per super-sampled pixel for the non-recursive method `+AM1`	Maximum number of samples per super-sampled pixel for the recursive method `+AM2`
1	1	9
2	4	25
3	9	81
4	16	289
5	25	1089
6	36	4225
7	49	16641
8	64	66049
9	81	263169

Table 2.45:

Note: the maximum number of samples in the recursive case is hardly ever reached for a given pixel. If the recursive method is used with no anti-aliasing each pixel will be the average of the rays traced at its corners. In most cases a recursion level of three is sufficient.

Another way to reduce aliasing artefacts is to introduce noise into the sampling process. This is called *jittering* and works because the human visual system is much more forgiving to noise than it is to regular patterns. The location of the super-samples is jittered or wiggled a tiny amount when anti-aliasing is used. Jittering is used by default but it may be turned off with the `Jitter=off` option or `-J` switch. The amount of jittering can be set with the `Jitter_Amount=`*n.n* option. When using switches the jitter scale may be specified after the `+J`*n.n* switch. For example `+J0.5` uses half the

normal jitter. The default amount of 1.0 is the maximum jitter which will insure that all super-samples remain inside the original pixel.

Note: the jittering noise is random and non-repeatable so you should avoid using jitter in animation sequences as the anti-aliased pixels will vary and flicker annoyingly from frame to frame.

If anti-aliasing is not used one sample per pixel is taken regardless of the super-sampling method specified.

2.2 Camera

The camera definition describes the position, projection type and properties of the camera viewing the scene. Its syntax is:

```
CAMERA:
    camera{ [CAMERA_ITEMS...] }
CAMERA_ITEM:
    CAMERA_TYPE | CAMERA_VECTOR | CAMERA_MODIFIER |
    CAMERA_IDENTIFIER
CAMERA_TYPE:
    perspective | orthographic | fisheye | ultra_wide_angle |
    omnimax | panoramic | cylinder CylinderType | spherical
CAMERA_VECTOR:
    location <Location> | right <Right> | up <Up> |
    direction <Direction> | sky <Sky>
CAMERA_MODIFIER:
    angle HORIZONTAL [VERTICAL] | look_at <Look_At> |
    blur_samples Num_of_Samples | aperture Size |
    focal_point <Point> | confidence Blur_Confidence |
    variance Blur_Variance | NORMAL | TRANSFORMATION
```

Camera default values:

```
DEFAULT CAMERA:
  camera {
    perspective
    location <0,0,0>
    direction <0,0,1>
    right 1.33*x
    up y
    sky <0,1,0>
  }

CAMERA TYPE: perspective
angle      : \~{}67.380 ( direction_length=0.5*
                   right_length/tan(angle/2) )
confidence : 0.9 (90\%)
direction  : <0,0,1>
focal_point: <0,0,0>
location   : <0,0,0>
look_at    : z
right      : 1.33*x
```

```
sky         : <0,1,0>
up          : y
variance    : 1/128
```

Depending on the projection type zero or more of the parameters are required:

- If no camera is specified the default camera is used.
- If no projection type is given the perspective camera will be used (pinhole camera).
- The *CAMERA_TYPE* has to be the first item in the camera statement.
- Other *CAMERA_ITEMs* may legally appear in any order.
- For other than the perspective camera, the minimum that has to be specified is the CAMERA_TYPE, the cylindrical camera also requires the *CAMERA_TYPE* to be followed by a float.
- The Orthographic camera has two 'modes'. For the pure orthographic projection up or right have to be specified. For an orthographic camera, with the same area of view as a perspective camera at the plane which goes through the look_at point, the angle keyword has to be use. A value for the angle is optional.
- All other *CAMERA_ITEM*s are taken from the default camera, unless they are specified differently.

2.2.1 Placing the Camera

The POV-Ray camera has ten different models, each of which uses a different projection method to project the scene onto your screen. Regardless of the projection type all cameras use the `location`, `right`, `up`, `direction`, and keywords to determine the location and orientation of the camera. The type keywords and these four vectors fully define the camera. All other camera modifiers adjust how the camera does its job. The meaning of these vectors and other modifiers differ with the projection type used. A more detailed explanation of the camera types follows later. In the sub-sections which follows, we explain how to place and orient the camera by the use of these four vectors and the `sky` and `look_at` modifiers. You may wish to refer to the illustration of the perspective camera below as you read about these vectors.

Location and Look_At

Under many circumstances just two vectors in the camera statement are all you need to position the camera: `location` and `look_at` vectors. For example:

```
camera {
  location <3,5,-10>
  look_at <0,2,1>
}
```

The location is simply the x, y, z coordinates of the camera. The camera can be located anywhere in the ray-tracing universe. The default location is `<0,0,0>`. The `look_at` vector tells POV-Ray to pan and tilt the camera until it is looking at the specified x, y,

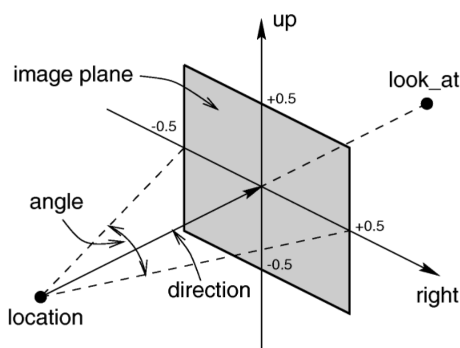

Figure 2.3: The perspective camera.

z coordinates. By default the camera looks at a point one unit in the z-direction from the location.

The look_at modifier should almost always be the last item in the camera statement. If other camera items are placed after the look_at vector then the camera may not continue to look at the specified point.

The Sky Vector

Normally POV-Ray pans left or right by rotating about the y-axis until it lines up with the look_at point and then tilts straight up or down until the point is met exactly. However you may want to slant the camera sideways like an airplane making a banked turn. You may change the tilt of the camera using the sky vector. For example:

```
camera {
 location <3,5,-10>
 sky   <1,1,0>
 look_at <0,2,1>
}
```

This tells POV-Ray to roll the camera until the top of the camera is in line with the sky vector. Imagine that the sky vector is an antenna pointing out of the top of the camera. Then it uses the sky vector as the axis of rotation left or right and then to tilt up or down in line with the sky until pointing at the look_at point. In effect you're telling POV-Ray to assume that the sky isn't straight up.

The sky vector does nothing on its own. It only modifies the way the look_at vector turns the camera. The default value is sky<0,1,0>.

Angles

The angle keyword followed by a float expression specifies the (horizontal) viewing angle in degrees of the camera used. Even though it is possible to use the direction vector to determine the viewing angle for the perspective camera it is much easier to use the angle keyword.

When you specify the `angle`, POV-Ray adjusts the length of the `direction` vector accordingly. The formula used is *direction_length = 0.5 * right_length / tan(angle / 2)* where *right_length* is the length of the `right` vector. You should therefore specify the `direction` and `right` vectors before the `angle` keyword. The `right` vector is explained in the next section.

There is no limitation to the viewing angle except for the perspective projection. If you choose viewing angles larger than 360 degrees you'll see repeated images of the scene (the way the repetition takes place depends on the camera). This might be useful for special effects.

The `spherical` camera has the option to also specify a vertical angle. If not specified it defaults to the horizontal angle/2

For example if you render an image with a 2:1 aspect ratio and map it to a sphere using spherical mapping, it will recreate the scene. Another use is to map it onto an object and if you specify transformations for the object before the texture, say in an animation, it will look like reflections of the environment (sometimes called environment mapping).

The Direction Vector

You will probably not need to explicitly specify or change the camera `direction` vector but it is described here in case you do. It tells POV-Ray the initial direction to point the camera before moving it with the `look_at` or `rotate` vectors (the default value is `direction<0,0,1>`). It may also be used to control the (horizontal) field of view with some types of projection. The length of the vector determines the distance of the viewing plane from the camera's location. A shorter `direction` vector gives a wider view while a longer vector zooms in for close-ups. In early versions of POV-Ray, this was the only way to adjust field of view. However zooming should now be done using the easier to use `angle` keyword.

If you are using the `ultra_wide_angle`, `panoramic`, or `cylindrical` projection you should use a unit length `direction` vector to avoid strange results. The length of the `direction` vector doesn't matter when using the `orthographic`, `fisheye`, or `omnimax` projection types.

Up and Right Vectors

The primary purpose of the `up` and `right` vectors is to tell POV-Ray the relative height and width of the view screen. The default values are:

```
right 4/3*x
up y
```

In the default `perspective` camera, these two vectors also define the initial plane of the view screen before moving it with the `look_at` or `rotate` vectors. The length of the `right` vector (together with the `direction` vector) may also be used to control the (horizontal) field of view with some types of projection. The `look_at` modifier changes both the `up` and `right` vectors. The `angle` calculation depends on the `right` vector.

Most camera types treat the `up` and `right` vectors the same as the `perspective` type. However several make special use of them. In the `orthographic` projection: The

lengths of the up and right vectors set the size of the viewing window regardless of the direction vector length, which is not used by the orthographic camera.

When using cylindrical projection: types 1 and 3, the axis of the cylinder lies along the up vector and the width is determined by the length of right vector or it may be overridden with the angle vector. In type 3 the up vector determines how many units high the image is. For example if you have up 4*y on a camera at the origin. Only points from y=2 to y=-2 are visible. All viewing rays are perpendicular to the y-axis. For type 2 and 4, the cylinder lies along the right vector. Viewing rays for type 4 are perpendicular to the right vector.

Note: that the up, right, and direction vectors should always remain perpendicular to each other or the image will be distorted. If this is not the case a warning message will be printed. The vista buffer will not work for non-perpendicular camera vectors.

Aspect Ratio

Together the up and right vectors define the *aspect ratio* (height to width ratio) of the resulting image. The default values up<0,1,0> and right<1.33,0,0> result in an aspect ratio of 4 to 3. This is the aspect ratio of a typical computer monitor. If you wanted a tall skinny image or a short wide panoramic image or a perfectly square image you should adjust the up and right vectors to the appropriate proportions.

Most computer video modes and graphics printers use perfectly square pixels. For example Macintosh displays and IBM SVGA modes 640x480, 800x600 and 1024x768 all use square pixels. When your intended viewing method uses square pixels then the width and height you set with the Width and Height options or +W or +H switches should also have the same ratio as the up and right vectors.

Note: 640/480 = 4/3 so the ratio is proper for this square pixel mode.

Not all display modes use square pixels however. For example IBM VGA mode 320x200 and Amiga 320x400 modes do not use square pixels. These two modes still produce a 4/3 aspect ratio image. Therefore images intended to be viewed on such hardware should still use 4/3 ratio on their up and right vectors but the pixel settings will not be 4/3.

For example:

```
camera {
 location <3,5,-10>
 up    <0,1,0>
 right  <1,0,0>
 look_at <0,2,1>
}
```

This specifies a perfectly square image. On a square pixel display like SVGA you would use pixel settings such as +W480 +H480 or +W600 +H600. However on the non-square pixel Amiga 320x400 mode you would want to use values of +W240 +H400 to render a square image.

The bottom line issue is this: the up and right vectors should specify the artist's intended aspect ratio for the image and the pixel settings should be adjusted to that

same ratio for square pixels and to an adjusted pixel resolution for non-square pixels. The `up` and `right` vectors should *not* be adjusted based on non-square pixels.

Handedness

The `right` vector also describes the direction to the right of the camera. It tells POV-Ray where the right side of your screen is. The sign of the `right` vector can be used to determine the handedness of the coordinate system in use. The default value is: `right<1.33,0,0>`. This means that the +x-direction is to the right. It is called a *left-handed* system because you can use your left hand to keep track of the axes. Hold out your left hand with your palm facing to your right. Stick your thumb up. Point straight ahead with your index finger. Point your other fingers to the right. Your bent fingers are pointing to the +x-direction. Your thumb now points into +y-direction. Your index finger points into the +z-direction.

To use a right-handed coordinate system, as is popular in some CAD programs and other ray-tracers, make the same shape using your right hand. Your thumb still points up in the +y-direction and your index finger still points forward in the +z-direction but your other fingers now say the +x-direction is to the left. That means that the right side of your screen is now in the -x-direction. To tell POV-Ray to act like this you can use a negative x value in the `right` vector such as: `right<-1.33,0,0>`. Since having x values increasing to the left doesn't make much sense on a 2D screen you now rotate the whole thing 180 degrees around by using a positive z value in your camera's location. You end up with something like this.

```
camera {
 location <0,0,10>
 up    <0,1,0>
 right  <-1.33,0,0>
 look_at <0,0,0>
}
```

Now when you do your ray-tracer's aerobics, as explained in the section "Understanding POV-Ray's Coordinate System", you use your right hand to determine the direction of rotations.

In a two dimensional grid, x is always to the right and y is up. The two versions of handedness arise from the question of whether z points into the screen or out of it and which axis in your computer model relates to up in the real world.

Architectural CAD systems, like AutoCAD, tend to use the *God's Eye* orientation that the z-axis is the elevation and is the model's up direction. This approach makes sense if you're an architect looking at a building blueprint on a computer screen. z means up, and it increases towards you, with x and y still across and up the screen. This is the basic right handed system.

Stand alone rendering systems, like POV-Ray, tend to consider you as a participant. You're looking at the screen as if you were a photographer standing in the scene. The up direction in the model is now y, the same as up in the real world and x is still to the right, so z must be depth, which increases away from you into the screen. This is the basic left handed system.

Transforming the Camera

The various transformations such as `translate` and `rotate` modifiers can re-position the camera once you've defined it. For example:

```
camera {
 location < 0, 0, 0>
 direction < 0, 0, 1>
 up    < 0, 1, 0>
 right   < 1, 0, 0>
 rotate  <30, 60, 30>
 translate < 5, 3, 4>
}
```

In this example, the camera is created, then rotated by 30 degrees about the x-axis, 60 degrees about the y-axis and 30 degrees about the z-axis, then translated to another point in space.

2.2.2 Types of Projection

The following list explains the different projection types that can be used with the camera. The most common types are the perspective and orthographic projections. The *CAMERA_TYPE* should be the *first* item in a `camera` statement. If none is specified, the `perspective` camera is the default.

You should note that the vista buffer can only be used with the perspective and orthographic camera.

Perspective projection

The `perspective` keyword specifies the default perspective camera which simulates the classic pinhole camera. The (horizontal) viewing angle is either determined by the ratio between the length of the `direction` vector and the length of the `right` vector or by the optional keyword `angle`, which is the preferred way. The viewing angle has to be larger than 0 degrees and smaller than 180 degrees. See the figure in "Placing the Camera" for the geometry of the perspective camera.

Orthographic projection

The orthographic camera offers two modes of operation:

The pure `orthographic` projection. This projection uses parallel camera rays to create an image of the scene. The area of view is determined by the lengths of the `right` and `up` vectors. One of these has to be specified, they are not taken from the default camera. If omitted the second method of the camera is used.

If, in a perspective camera, you replace the `perspective` keyword by `orthographic` and leave all other parameters the same, you'll get an orthographic view with the same image area, i.e. the size of the image is the same. The same can be achieved by adding the `angle` keyword to an orthographic camera. A value for the angle is optional. So

this second mode is active if no up and right are within the camera statement, or when the angle keyword is within the camera statement.

You should be aware though that the visible parts of the scene change when switching from perspective to orthographic view. As long as all objects of interest are near the look_at point they'll be still visible if the orthographic camera is used. Objects farther away may get out of view while nearer objects will stay in view.

If objects are too close to the camera location they may disappear. Too close here means, behind the orthographic camera projection plane (the plane that goes through the `look_at` point).

Fisheye projection

This is a spherical projection. The viewing angle is specified by the `angle` keyword. An angle of 180 degrees creates the "standard" fisheye while an angle of 360 degrees creates a super-fisheye ("I-see-everything-view"). If you use this projection you should get a circular image. If this isn't the case, i.e. you get an elliptical image, you should read "Aspect Ratio".

Ultra wide angle projection

This projection is somewhat similar to the fisheye but it projects the image onto a rectangle instead of a circle. The viewing angle can be specified using the `angle` keyword.

Omnimax projection

The omnimax projection is a 180 degrees fisheye that has a reduced viewing angle in the vertical direction. In reality this projection is used to make movies that can be viewed in the dome-like Omnimax theaters. The image will look somewhat elliptical. The `angle` keyword isn't used with this projection.

Panoramic projection

This projection is called "cylindrical equirectangular projection". It overcomes the degeneration problem of the perspective projection if the viewing angle approaches 180 degrees. It uses a type of cylindrical projection to be able to use viewing angles larger than 180 degrees with a tolerable lateral-stretching distortion. The `angle` keyword is used to determine the viewing angle.

Cylindrical projection

Using this projection the scene is projected onto a cylinder. There are four different types of cylindrical projections depending on the orientation of the cylinder and the position of the viewpoint. A float value in the range 1 to 4 must follow the `cylinder` keyword. The viewing angle and the length of the `up` or `right` vector determine the

dimensions of the camera and the visible image. The camera to use is specified by a number. The types are:

1. vertical cylinder, fixed viewpoint
2. horizontal cylinder, fixed viewpoint
3. vertical cylinder, viewpoint moves along the cylinder's axis
4. horizontal cylinder, viewpoint moves along the cylinder's axis

Spherical projection

Using this projection the scene is projected onto a sphere.
Syntax:

```
camera {
  spherical
  [angle HORIZONTAL [VERTICAL]]
  [CAMERA_ITEMS...]
}
```

The first value after `angle` sets the horizontal viewing angle of the camera. With the optional second value, the vertical viewing angle is set: both in degrees. If the vertical angle is not specified, it defaults to half the horizontal angle.

The spherical projection is similar to the fisheye projection, in that the scene is projected on a sphere. But unlike the fisheye camera, it uses rectangular coordinates instead of polar coordinates; in this it works the same way as spherical mapping (map_type 1).

This has a number of uses. Firstly, it allows an image rendered with the spherical camera to be mapped on a sphere without distortion (with the fisheye camera, you first have to convert the image from polar to rectangular coordinates in some image editor). Also, it allows effects such as "environment mapping", often used for simulating reflections in scanline renderers.

2.2.3 Focal Blur

POV-Ray can simulate focal depth-of-field by shooting a number of sample rays from jittered points within each pixel and averaging the results.

To turn on focal blur, you must specify the `aperture` keyword followed by a float value which determines the depth of the sharpness zone. Large apertures give a lot of blurring, while narrow apertures will give a wide zone of sharpness.

Note: while this behaves as a real camera does, the values for aperture are purely arbitrary and are not related to *f*-stops.

You must also specify the `blur_samples` keyword followed by an integer value specifying the maximum number of rays to use for each pixel. More rays give a smoother appearance but is slower. By default no focal blur is used, i. e. the default aperture is 0 and the default number of samples is 0.

The center of the *zone of sharpness* is specified by the `focal_point` vector. The *zone of sharpness* is a plane through the `focal_point` and is parallel to the camera. Objects close to this plane of focus are in focus and those farther from that plane are more blurred. The default value is `focal_point<0,0,0>`.

Although `blur_samples` specifies the maximum number of samples, there is an adaptive mechanism that stops shooting rays when a certain degree of confidence has been reached. At that point, shooting more rays would not result in a significant change.

The `confidence` and `variance` keywords are followed by float values to control the adaptive function. The `confidence` value is used to determine when the samples seem to be *close enough* to the correct color. The `variance` value specifies an acceptable tolerance on the variance of the samples taken so far. In other words, the process of shooting sample rays is terminated when the estimated color value is very likely (as controlled by the confidence probability) near the real color value.

Since the `confidence` is a probability its values can range from 0 to <1 (the default is 0.9, i. e. 90%). The value for the `variance` should be in the range of the smallest displayable color difference (the default is 1/128). If 1 is used POV-Ray will issue a warning and then use the default instead.

Rendering with the default settings can result in quite grainy images. This can be improved by using a lower `variance`. A value of 1/10000 gives a fairly good result (with default confidence and blur_samples set to something like 100) without being unacceptably slow.

Larger `confidence` values will lead to more samples, slower traces and better images. The same holds for smaller `variance` thresholds.

2.2.4 Camera Ray Perturbation

The optional `normal` may be used to assign a normal pattern to the camera. For example:

```
camera{
  location Here
  look_at There
  normal { bumps 0.5 }
}
```

All camera rays will be perturbed using this pattern. The image will be distorted as though you were looking through bumpy glass or seeing a reflection off of a bumpy surface. This lets you create special effects. See the animated scene `camera2.pov` for an example. See ”Normal” for information on normal patterns.

2.2.5 Camera Identifiers

Camera identifiers may be declared to make scene files more readable and to parameterize scenes so that changing a single declaration changes many values. You may declare several camera identifiers if you wish. This makes it easy to quickly change cameras. An identifier is declared as follows.

```
CAMERA_DECLARATION:
    #declare IDENTIFIER = CAMERA |
    #local IDENTIFIER = CAMERA
```

Where *IDENTIFIER* is the name of the identifier up to 40 characters long and *CAMERA* is any valid camera statement. See ”#declare vs. #local” for information on identifier scope. Here is an example...

```
 #declare Long_Lens = camera {
    location -z*100
    look_at <0,0,0>
    angle 3
 }

 #declare Short_Lens = camera {
    location -z*50
    look_at <0,0,0>
    angle 15
 }

 camera {
    Long_Lens  // edit this line to change lenses
    translate <33,2,0>
 }
```

Note: only camera transformations can be added to an already declared camera. Camera behaviour changing keywords are not allowed, as they are needed in an earlier stage for resolving the keyword order dependencies.

2.3 Atmospheric Effects

Atmospheric effects are a loosely-knit group of features that affect the background and/or the atmosphere enclosing the scene. POV-Ray includes the ability to render a number of atmospheric effects, such as fog, haze, mist, rainbows and skies.

2.3.1 Atmospheric Media

Atmospheric effects such as fog, dust, haze, or visible gas may be simulated by a `media` statement specified in the scene but not attached to any object. All areas not inside a non-hollow object in the entire scene. A very simple approach to add fog to a scene is explained in section ”Fog” however this kind of fog does not interact with any light sources like `media` does. It will not show light beams or other effects and is therefore not very realistic.

The atmosphere media effect overcomes some of the fog’s limitations by calculating the interaction between light and the particles in the atmosphere using volume sampling. Thus shafts of light beams will become visible and objects will cast shadows onto smoke or fog.

Note: POV-Ray can’t sample media along an infinitely long ray. The ray must be finite in order to be possible to sample. This means that sampling media is only possible for

rays that hit an object. So no atmospheric media will show up against `background` or `sky_sphere`.
Another way of being able to sample media is using spotlights because also in this case the ray is not infinite (it is sampled only inside the spotlight cone).

With spotlights you'll be able to create the best results because their cone of light will become visible. Pointlights can be used to create effects like street lights in fog. Lights can be made to not interact with the atmosphere by adding `media_interaction off` to the light source. They can be used to increase the overall light level of the scene to make it look more realistic.

Complete details on `media` are given in the section "Media". Earlier versions of POV-Ray used an `atmosphere` statement for atmospheric effects but that system was incompatible with the old object `halo` system. So `atmosphere` has been eliminated and replaced with a simpler and more powerful media feature. The user now only has to learn one `media` system for either atmospheric or object use.

If you only want media effects in a particular area, you should use object media rather than only relying upon the media pattern. In general it will be faster and more accurate because it only calculates inside the constraining object.

Note: the atmosphere feature will not work if the camera is inside a non-hollow object (see section "Empty and Solid Objects" for a detailed explanation).

2.3.2 Background

A background color can be specified if desired. Any ray that doesn't hit an object will be colored with this color. The default background is black. The syntax for `background` is:

```
BACKGROUND:
     background {COLOR}
```

2.3.3 Fog

If it is not necessary for light beams to interact with atmospheric media, then `fog` may be a faster way to simulate haze or fog. This feature artificially adds color to every pixel based on the distance the ray has traveled. The syntax for fog is:

```
FOG:
    fog { [FOG_IDENTIFIER] [FOG_ITEMS...] }
FOG_ITEMS:
    fog_type Fog_Type | distance Distance | COLOR |
    turbulence <Turbulence> | turb_depth Turb_Depth |
    omega Omega | lambda Lambda | octaves Octaves |
    fog_offset Fog_Offset | fog_alt Fog_Alt |
    up <Fog_Up> | TRANSFORMATION
```

Fog default values:

```
lambda     : 2.0
fog_type   : 1
```

```
fog_offset : 0.0
fog_alt    : 0.0
octaves    : 6
omega      : 0.5
turbulence : <0,0,0>
turb_depth : 0.5
up         : <0,1,0>
```

Currently there are two fog types, the default `fog_type 1` is a constant fog and `fog_type 2` is ground fog. The constant fog has a constant density everywhere while the ground fog has a constant density for all heights below a given point on the up axis and thins out along this axis.

The color of a pixel with an intersection depth *d* is calculated by

*PIXEL_COLOR = exp(-d/D) * OBJECT_COLOR + (1-exp(-d/D)) * FOG_COLOR*

where *D* is the specified value of the required fog `distance` keyword. At depth 0 the final color is the object's color. If the intersection depth equals the fog distance the final color consists of 64% of the object's color and 36% of the fog's color.

Note: for this equation, a distance of zero is undefined. In practice, povray will treat this value as "fog is off". To use an extremely thick fog, use a small nonzero number such as 1e-6 or 1e-10.

For ground fog, the height below which the fog has constant density is specified by the `fog_offset` keyword. The `fog_alt` keyword is used to specify the rate by which the fog fades away. The default values for both are 0.0 so be sure to specify them if ground fog is used. At an altitude of `Fog_Offset+Fog_Alt` the fog has a density of 25%. The density of the fog at height less than or equal to *Fog_Offset* is 1.0 and for height larger than than *Fog_Offset* is calculated by:

```
1/(1 + (y - Fog_Offset) / Fog_Alt) ^2
```

The total density along a ray is calculated by integrating from the height of the starting point to the height of the end point.

The optional `up` vector specifies a direction pointing up, generally the same as the camera's up vector. All calculations done during the ground fog evaluation are done relative to this up vector, i. e. the actual heights are calculated along this vector. The up vector can also be modified using any of the known transformations described in "Transformations". Though it may not be a good idea to scale the up vector - the results are hardly predictable - it is quite useful to be able to rotate it. You should also note that translations do not affect the up direction (and thus don't affect the fog).

The required fog color has three purposes. First it defines the color to be used in blending the fog and the background. Second it is used to specify a translucency threshold. By using a transmittance larger than zero one can make sure that at least that amount of light will be seen through the fog. With a transmittance of 0.3 you'll see at least 30% of the background. Third it can be used to make a filtering fog. With a filter value larger than zero the amount of background light given by the filter value will be multiplied with the fog color. A filter value of 0.7 will lead to a fog that filters 70% of the background light and leaves 30% unfiltered.

Fogs may be layered. That is, you can apply as many layers of fog as you like. Generally this is most effective if each layer is a ground fog of different color, altitude and

with different turbulence values. To use multiple layers of fogs, just add all of them to the scene.

You may optionally stir up the fog by adding turbulence. The `turbulence` keyword may be followed by a float or vector to specify an amount of turbulence to be used. The `omega`, `lambda` and `octaves` turbulence parameters may also be specified. See section "Pattern Modifiers" for details on all of these turbulence parameters.

Additionally the fog turbulence may be scaled along the direction of the viewing ray using the `turb_depth` amount. Typical values are from 0.0 to 1.0 or more. The default value is 0.5 but any float value may be used.

Note: the fog feature will not work if the camera is inside a non-hollow object (see section "Empty and Solid Objects" for a detailed explanation).

2.3.4 Sky Sphere

The sky sphere is used create a realistic sky background without the need of an additional sphere to simulate the sky. Its syntax is:

```
SKY_SPHERE:
    sky_sphere { [SKY_SPHERE_IDENTIFIER] [SKY_SPHERE_ITEMS...] }
SKY_SPHERE_ITEM:
    PIGMENT | TRANSFORMATION
```

The sky sphere can contain several pigment layers with the last pigment being at the top, i. e. it is evaluated last, and the first pigment being at the bottom, i. e. it is evaluated first. If the upper layers contain filtering and/or transmitting components lower layers will shine through. If not lower layers will be invisible.

The sky sphere is calculated by using the direction vector as the parameter for evaluating the pigment patterns. This leads to results independent from the view point which pretty good models a real sky where the distance to the sky is much larger than the distances between visible objects.

If you want to add a nice color blend to your background you can easily do this by using the following example.

```
sky_sphere {
  pigment {
    gradient y
    color_map {
      [ 0.5  color CornflowerBlue ]
      [ 1.0  color MidnightBlue ]
    }
    scale 2
    translate -1
  }
}
```

This gives a soft blend from `CornflowerBlue` at the horizon to `MidnightBlue` at the zenith. The scale and translate operations are used to map the direction vector values, which lie in the range from <-1, -1, -1> to <1, 1, 1>, onto the range from <0, 0, 0> to

<1, 1, 1>. Thus a repetition of the color blend is avoided for parts of the sky below the horizon.

In order to easily animate a sky sphere you can transform it using the usual transformations described in ”Transformations”. Though it may not be a good idea to translate or scale a sky sphere - the results are hardly predictable - it is quite useful to be able to rotate it. In an animation the color blendings of the sky can be made to follow the rising sun for example.

Note: only one sky sphere can be used in any scene. It also will not work as you might expect if you use camera types like the orthographic or cylindrical camera. The orthographic camera uses parallel rays and thus you'll only see a very small part of the sky sphere (you'll get one color skies in most cases). Reflections in curved surface will work though, e. g. you will clearly see the sky in a mirrored ball.

2.3.5 Rainbow

Rainbows are implemented using fog-like, circular arcs. Their syntax is:

```
RAINBOW:
    rainbow { [RAINBOW_IDENTIFIER] [RAINBOW_ITEMS...] }
RAINBOW_ITEM:
    direction <Dir> | angle Angle | width Width |
    distance Distance | COLOR_MAP | jitter Jitter | up <Up> |
    arc_angle Arc_Angle | falloff_angle Falloff_Angle
```

Rainbow default values:

```
arc_angle     : 180.0
falloff_angle : 180.0
jitter        : 0.0
up            : y
```

The required `direction` vector determines the direction of the (virtual) light that is responsible for the rainbow. Ideally this is an infinitely far away light source like the sun that emits parallel light rays. The position and size of the rainbow are specified by the required `angle` and `width` keywords. To understand how they work you should first know how the rainbow is calculated.

For each ray the angle between the rainbow's direction vector and the ray's direction vector is calculated. If this angle lies in the interval from *`Angle-Width/2`* to *`Angle+Width/2`* the rainbow is hit by the ray. The color is then determined by using the angle as an index into the rainbow's color_map. After the color has been determined it will be mixed with the background color in the same way like it is done for fogs.

Thus the angle and width parameters determine the angles under which the rainbow will be seen. The optional `jitter` keyword can be used to add random noise to the index. This adds some irregularity to the rainbow that makes it look more realistic.

The required `distance` keyword is the same like the one used with fogs. Since the rainbow is a fog-like effect it's possible that the rainbow is noticeable on objects. If this effect is not wanted it can be avoided by using a large distance value. By default a sufficiently large value is used to make sure that this effect does not occur.

The color_map statement is used to assign a color map that will be mapped onto the rainbow. To be able to create realistic rainbows it is important to know that the index into the color map increases with the angle between the ray's and rainbow's direction vector. The index is zero at the innermost ring and one at the outermost ring. The filter and transmittance values of the colors in the color map have the same meaning as the ones used with fogs (see section "Fog").

The default rainbow is a 360 degree arc that looks like a circle. This is no problem as long as you have a ground plane that hides the lower, non-visible part of the rainbow. If this isn't the case or if you don't want the full arc to be visible you can use the optional keywords up, arc_angle and falloff_angle to specify a smaller arc.

The arc_angle keyword determines the size of the arc in degrees (from 0 to 360 degrees). A value smaller than 360 degrees results in an arc that abruptly vanishes. Since this doesn't look nice you can use the falloff_angle keyword to specify a region in which the rainbow will smoothly blend into the background making it vanish softly. The falloff angle has to be smaller or equal to the arc angle.

The up keyword determines were the zero angle position is. By changing this vector you can rotate the rainbow about its direction. You should note that the arc goes from *-Arc_Angle/2* to *+Arc_Angle/2*. The soft regions go from *-Arc_Angle/2* to *-Falloff_Angle/2* and from *+Falloff_Angle/2* to *+Arc_Angle/2*.

The following example generates a 120 degrees rainbow arc that has a falloff region of 30 degrees at both ends:

```
rainbow {
  direction <0, 0, 1>
  angle 42.5
  width 5
  distance 1000
  jitter 0.01
  color_map { Rainbow_Color_Map }
  up <0, 1, 0>
  arc_angle 120
  falloff_angle 30
}
```

It is possible to use any number of rainbows and to combine them with other atmospheric effects.

2.4 Global Settings

The global_settings statement is a catch-all statement that gathers together a number of global parameters. The statement may appear anywhere in a scene as long as it is not inside any other statement. You may have multiple global_settings statements in a scene. Whatever values were specified in the last global_settings statement override any previous settings.

Note: some items which were language directives in earlier versions of POV-Ray have been moved inside the global_settings statement so that it is more obvious to the user that their effect is global. The old syntax is permitted but generates a warning.

The new syntax is:

```
GLOBAL_SETTINGS:
    global_settings { [GLOBAL_SETTINGS_ITEMS...] }
GLOBAL_SETTINGS_ITEM:
    adc_bailout Value | ambient_light COLOR | assumed_gamma Value |
    hf_gray_16 [Bool] | irid_wavelength COLOR |
    charset GLOBAL_CHARSET | max_intersections Number |
    max_trace_level Number | number_of_waves Number |
    noise_generator Number | radiosity { RADIOSITY_ITEMS... } |
    photon { PHOTON_ITEMS... }
GLOBAL_CHARSET:
    ascii | utf8 | sys
```

Global setting default values:

```
charset           : ascii
adc_bailout       : 1/255
ambient_light     : <1,1,1>
assumed_gamma     : No gamma correction
hf_gray_16        : off
irid_wavelength   : <0.25,0.18,0.14>
max_trace_level   : 5
max_intersections : 64
number_of_waves   : 10
noise_generator   : 2

Radiosity:
  adc_bailout      : 0.01
  always_sample    : on
  brightness       : 1.0
  count            : 35  (max = 1600)
  error_bound      : 1.8
  gray_threshold   : 0.0
  low_error_factor : 0.5
  max_sample       : non-positive value
  minimum_reuse    : 0.015
  nearest_count    : 5   (max = 20)
  normal           : off
  pretrace_start   : 0.08
  pretrace_end     : 0.04
  recursion_limit  : 3
```

Each item is optional and may appear in any order. If an item is specified more than once, the last setting overrides previous values. Details on each item are given in the following sections.

2.4.1 ADC_Bailout

In scenes with many reflective and transparent surfaces, POV-Ray can get bogged down tracing multiple reflections and refractions that contribute very little to the color of a particular pixel. The program uses a system called *Adaptive Depth Control* (ADC) to stop computing additional reflected or refracted rays when their contribution is insignificant.

You may use the global setting adc_bailout keyword followed by float value to specify the point at which a ray's contribution is considered insignificant. For example:

```
global_settings { adc_bailout 0.01 }
```

The default value is 1/255, or approximately 0.0039, since a change smaller than that could not be visible in a 24 bit image. Generally this setting is perfectly adequate and should be left alone. Setting adc_bailout to 0 will disable ADC, relying completely on max_trace_level to set an upper limit on the number of rays spawned.

See section "Max_Trace_Level" for details on how ADC and max_trace_level interact.

2.4.2 Ambient_Light

Ambient light is used to simulate the effect of inter-diffuse reflection that is responsible for lighting areas that partially or completely lie in shadow. POV-Ray provides the ambient_light keyword to let you easily change the brightness of the ambient lighting without changing every ambient value in all finish statements. It also lets you create interesting effects by changing the color of the ambient light source. The syntax is:

```
global_settings { ambient_light COLOR }
```

The default is a white ambient light source set at rgb <1,1,1>. Only the rgb components are used. The actual ambient used is: *Ambient = Finish_Ambient * Global_Ambient*.

See section "Ambient" for more information.

2.4.3 Assumed_Gamma

Many people may have noticed at one time or another that some images are too bright or dim when displayed on their system. As a rule, Macintosh users find that images created on a PC are too bright, while PC users find that images created on a Macintosh are too dim.

The assumed_gamma global setting works in conjunction with the Display_Gamma INI setting (see section "Display Hardware Settings") to ensure that scene files render the same way across the wide variety of hardware platforms that POV-Ray is used on. The assumed gamma setting is used in a scene file by adding

```
global_settings { assumed_gamma Value }
```

where the assumed gamma value is the correction factor to be applied before the pixels are displayed and/or saved to disk. For scenes created in older versions of POV-Ray, the assumed gamma value will be the same as the display gamma value of the system the scene was created on. For PC systems, the most common display gamma is 2.2, while for scenes created on Macintosh systems should use a scene gamma of 1.8. Another gamma value that sometimes occurs in scenes is 1.0.

Scenes that do not have an assumed_gamma global setting will not have any gamma correction performed on them, for compatibility reasons. If you are creating new scenes or rendering old scenes, it is strongly recommended that you put in an appropriate assumed_gamma global setting. For new scenes, you should use an assumed gamma value of 1.0 as this models how light appears in the real world more realistically.

Before we go to the following sections, that explain more thoroughly what gamma is and why it is important, a short overview of how gamma works in POV-Ray:

no `assumed_gamma` in scene :

No gamma correction is applied to output file.

`assumed_gamma 1` :

Gamma `Display_Gamma` is applied to output file.

If `Display_Gamma` is not specified, 2.2 is used.

`assumed_gamma G` :

Gamma `Display_Gamma/G` is applied to output file.

If `Display_Gamma` is not specified, 2.2/G is used.

Recommended value for `assumed_gamma` is 1.

Monitor Gamma

The differences in how images are displayed is a result of how a computer actually takes an image and displays it on the monitor. In the process of rendering an image and displaying it on the screen, several gamma values are important, including the POV scene file or image file gamma and the monitor gamma.

Most image files generated by POV-Ray store numbers in the range from 0 to 255 for each of the red, green and blue components of a pixel. These numbers represent the intensity of each color component, with 0 being black and 255 being the brightest color (either 100% red, 100% green or 100% blue). When an image is displayed, the graphics card converts each color component into a voltage which is sent to the monitor to light up the red, green and blue phosphors on the screen. The voltage is usually proportional to the value of each color component.

Gamma becomes important when displaying intensities that aren't the maximum or minimum possible values. For example, 127 should represent 50% of the maximum intensity for pixels stored as numbers between 0 and 255. On systems that don't do gamma correction, 127 will be converted to 50% of the maximum voltage, but because of the way the phosphors and the electron guns in a monitor work, this may be only 22% of the maximum color intensity on a monitor with a gamma of 2.2. To display a pixel which is 50% of the maximum intensity on this monitor, we would need a voltage of 73% of the maximum voltage, which translates to storing a pixel value of 186.

The relationship between the input pixel value and the displayed intensity can be approximated by an exponential function *`obright = ibright ^ display_gamma`* where obright is the output intensity and ibright is the input pixel intensity. Both values are in the range from 0 to 1 (0% to 100%). Most monitors have a fixed gamma value in the range from 1.8 to 2.6. Using the above formula with display_gamma values greater than 1 means that the output brightness will be less than the input brightness. In order to have the output and input brightness be equal an overall system gamma of 1 is needed. To do this, we need to gamma correct the input brightness in the same manner as above but with a gamma value of 1/display_gamma before it is sent to the monitor. To correct

for a display gamma of 2.2, this pre-monitor gamma correction uses a gamma value of 1.0/2.2 or approximately 0.45.

How the pre-monitor gamma correction is done depends on what hardware and software is being used. On Macintosh systems, the operating system has taken it upon itself to insulate applications from the differences in display hardware. Through a gamma control panel the user may be able to set the actual monitor gamma and Mac will then convert all pixel intensities so that the monitor will appear to have the specified gamma value. On Silicon Graphics machines, the display adapter has built-in gamma correction calibrated to the monitor which gives the desired overall gamma (the default is 1.7). Unfortunately, on PCs and most UNIX systems, it is up to the application to do any gamma correction needed.

Image File Gamma

Since most PC and UNIX applications and image file formats don't understand display gamma, they don't do anything to correct for it. As a result, users creating images on these systems adjust the image in such a way that it has the correct brightness when displayed. This means that the data values stored in the files are made brighter to compensate for the darkening effect of the monitor. In essence, the 0.45 gamma correction is built in to the image files created and stored on these systems. When these files are displayed on a Macintosh system, the gamma correction built in to the file, in addition to gamma correction built into MacOS, means that the image will be too bright. Similarly, files that look correct on Macintosh or SGI systems because of the built-in gamma correction will be too dark when displayed on a PC.

The PNG format files generated by POV-Ray overcome the problem of too much or not enough gamma correction by storing the image file gamma (which is 1.0/display_gamma) inside the PNG file when it is generated by POV-Ray. When the PNG file is later displayed by a program that has been set up correctly, it uses this gamma value as well as the current display gamma to correct for the potentially different display gamma used when originally creating the image.

Unfortunately, of all the image file formats POV-Ray supports, PNG is the only one that has any gamma correction features and is therefore preferred for images that will be displayed on a wide variety of platforms.

Scene File Gamma

The image file gamma problem itself is just a result of how scenes themselves are generated in POV-Ray. When you start out with a new scene and are placing light sources and adjusting surface textures and colors, you generally make several attempts before the lighting is how you like it. How you choose these settings depends upon the preview image or the image file stored to disk, which in turn is dependent upon the overall gamma of the display hardware being used.

This means that as the artist you are doing gamma correction in the POV-Ray scene file for your particular hardware. This scene file will generate an image file that is also gamma corrected for your hardware and will display correctly on systems similar to your own. However, when this scene is rendered on another platform, it may be

too bright or too dim, regardless of the output file format used. Rather than have you change all the scene files to have a single fixed gamma value (heaven forbid!), POV-Ray allows you to specify in the scene file the display gamma of the system that the scene was created on.

The `assumed_gamma` global setting, in conjunction with the `Display_Gamma` INI setting lets POV-Ray know how to do gamma correction on a given scene so that the preview and output image files will appear the correct brightness on any system. Since the gamma correction is done internally to POV-Ray, it will produce output image files that are the correct brightness for the current display, regardless of what output format is used. As well, since the gamma correction is performed in the high-precision data format that POV-Ray uses internally, it produces better results than gamma correction done after the file is written to disk.

Although you may not notice any difference in the output on your system with and without an `assumed_gamma` setting, the assumed gamma is important if the scene is ever rendered on another platform.

2.4.4 HF_Gray_16

The `hf_gray_16` setting is useful when using POV-Ray to generate heightfields for use in other POV-Ray scenes. The syntax is... global_settings { hf_gray_16 [Bool] }

The boolean value turns the option on or off. If the keyword is specified without the boolean value then the option is turned on. If `hf_gray_16` is not specified in any `global_settings` statement in the entire scene then the default is off.

When `hf_gray_16` is on, the output file will be in the form of a heightfield, with the height at any point being dependent on the brightness of the pixel. The brightness of a pixel is calculated in the same way that color images are converted to grayscale images: *`height = 0.3 * red + 0.59 * green + 0.11 * blue`*.

Setting the `hf_gray_16` option will cause the preview display, if used, to be grayscale rather than color. This is to allow you to see how the heightfield will look because some file formats store heightfields in a way that is difficult to understand afterwards. See section ”Height Field” for a description of how POV-Ray heightfields are stored for each file type.

2.4.5 Irid_Wavelength

Iridescence calculations depend upon the dominant wavelengths of the primary colors of red, green and blue light. You may adjust the values using the global setting `irid_wavelength` as follows...

```
global_settings { irid_wavelength COLOR }
```

The default value is `rgb <0.25,0.18,0.14>` and any filter or transmit values are ignored. These values are proportional to the wavelength of light but they represent no real world units.

In general, the default values should prove adequate but we provide this option as a means to experiment with other values.

2.4.6 Charset

This allows you to specify the assumed character set of all text strings. If you specify `ascii` only standard ASCII character codes in the range from 0 to 127 are valid. You can easily find a table of ASCII characters on the internet. The option `utf8` is a special Unicode text encoding and it allows you to specify characters of nearly all languages in use today. We suggest you use a text editor with the capability to export text to UTF8 to generate input files. You can find more information, including tables with codes of valid characters at http://www.unicode.org/ The last possible option is to use a system specific character set. For details about the `sys` character set option refer to the platform specific documentation.

2.4.7 Max_Trace_Level

In scenes with many reflective and transparent surfaces POV-Ray can get bogged down tracing multiple reflections and refractions that contribute very little to the color of a particular pixel. The global setting `max_trace_level` defines the integer maximum number of recursive levels that POV-Ray will trace a ray.

```
global_settings { max_trace_level Level }
```

This is used when a ray is reflected or is passing through a transparent object and when shadow rays are cast. When a ray hits a reflective surface, it spawns another ray to see what that point reflects. That is trace level one. If it hits another reflective surface another ray is spawned and it goes to trace level two. The maximum level by default is five.

One speed enhancement added to POV-Ray in version 3.0 is *Adaptive Depth Control* (ADC). Each time a new ray is spawned as a result of reflection or refraction its contribution to the overall color of the pixel is reduced by the amount of reflection or the filter value of the refractive surface. At some point this contribution can be considered to be insignificant and there is no point in tracing any more rays. Adaptive depth control is what tracks this contribution and makes the decision of when to bail out. On scenes that use a lot of partially reflective or refractive surfaces this can result in a considerable reduction in the number of rays fired and makes it safer to use much higher `max_trace_level` values.

This reduction in color contribution is a result of scaling by the reflection amount and/or the filter values of each surface, so a perfect mirror or perfectly clear surface will not be optimizable by ADC. You can see the results of ADC by watching the `Rays Saved` and `Highest Trace Level` displays on the statistics screen.

The point at which a ray's contribution is considered insignificant is controlled by the `adc_bailout` value. The default is 1/255 or approximately 0.0039 since a change smaller than that could not be visible in a 24 bit image. Generally this setting is perfectly adequate and should be left alone. Setting `adc_bailout` to 0 will disable ADC, relying completely on `max_trace_level` to set an upper limit on the number of rays spawned.

If `max_trace_level` is reached before a non-reflecting surface is found and if ADC hasn't allowed an early exit from the ray tree the color is returned as black. Raise

max_trace_level if you see black areas in a reflective surface where there should be a color.

The other symptom you could see is with transparent objects. For instance, try making a union of concentric spheres with a clear texture on them. Make ten of them in the union with radius's from 1 to 10 and render the scene. The image will show the first few spheres correctly, then black. This is because a new level is used every time you pass through a transparent surface. Raise max_trace_level to fix this problem.

Note: that raising max_trace_level will use more memory and time and it could cause the program to crash with a stack overflow error, although ADC will alleviate this to a large extent.

Values for max_trace_level can be set up to a maximum of 256. If there is no max_trace_level set and during rendering the default value is reached, a warning is issued.

2.4.8 Max_Intersections

POV-Ray uses a set of internal stacks to collect ray/object intersection points. The usual maximum number of entries in these *I-Stacks* is 64. Complex scenes may cause these stacks to overflow. POV-Ray doesn't stop but it may incorrectly render your scene. When POV-Ray finishes rendering, a number of statistics are displayed. If you see I-Stack Overflows reported in the statistics you should increase the stack size. Add a global setting to your scene as follows:

```
global_settings { max_intersections Integer }
```

If the I-Stack Overflows remain increase this value until they stop.

2.4.9 Number_Of_Waves

The waves and ripples patterns are generated by summing a series of waves, each with a slightly different center and size. By default, ten waves are summed but this amount can be globally controlled by changing the number_of_waves setting.

```
global_settings { number_of_waves Integer }
```

Changing this value affects both waves and ripples alike on all patterns in the scene.

2.4.10 Noise_generator

There are three noise generators implemented.

- noise_generator 1 the noise that was used in POV_Ray 3.1
- noise_generator 2 'range corrected' version of the old noise, it does not show the plateaus seen with noise_generator 1
- noise_generator 3 generates Perlin noise

The default is noise_generator 2

Note: The noise_generators can also be used within the pigment/normal/etc. statement.

2.4.11 Radiosity Basics

Important notice: The radiosity features in POV-Ray are somewhat experimental. There is a high probability that the design and implementation of these features will be changed in future versions. We cannot guarantee that scenes using these features in this version will render identically in future releases or that full backwards compatibility of language syntax can be maintained.

Radiosity is an extra calculation that more realistically computes the diffuse interreflection of light. This diffuse interreflection can be seen if you place a white chair in a room full of blue carpet, blue walls and blue curtains. The chair will pick up a blue tint from light reflecting off of other parts of the room. Also notice that the shadowed areas of your surroundings are not totally dark even if no light source shines directly on the surface. Diffuse light reflecting off of other objects fills in the shadows. Typically ray-tracing uses a trick called *ambient* light to simulate such effects but it is not very accurate.

Radiosity calculations are only made when a `radiosity{}` block is used inside the `global_settings{}` block.

The following sections describes how radiosity works, how to control it with various global settings and tips on trading quality vs. speed.

2.5 Radiosity

2.5.1 How Radiosity Works

The problem of ray-tracing is to figure out what the light level is at each point that you can see in a scene. Traditionally, in ray tracing, this is broken into the sum of these components:

Diffuse

: the effect that makes the side of things facing the light brighter;

Specular

: the effect that makes shiny things have dings or sparkles on them;

Reflection

: the effect that mirrors give; and

Ambient

: the general all-over light level that any scene has, which keeps things in shadow from being pure black.

POV-Ray's radiosity system, based on a method by Greg Ward, provides a way to replace the last term - the constant ambient light value - with a light level which is based on what surfaces are nearby and how bright in turn they are.

The first thing you might notice about this definition is that it is circular: the brightness and color of everything is dependent on everything else and vice versa. This is

true in real life but in the world of ray-tracing, we can make an approximation. The approximation that is used is: the objects you are looking at have their `ambient` values calculated for you by checking the other objects nearby. When those objects are checked during this process, however, their `diffuse` term is used. The brightness of radiosity in POV-Ray is based on two things:

1. the amount of light ”gathered”
2. the ’diffuse’ property of the surface finish

`ambient_lightglobal_settingsambient 0`

How does POV-Ray calculate the ambient term for each point? By sending out more rays, in many different directions, and averaging the results. A typical point might use 200 or more rays to calculate its ambient light level correctly.

Now this sounds like it would make the ray-tracer 200 times slower. This is true, except that the software takes advantage of the fact that ambient light levels change quite slowly (remember, shadows are calculated separately, so sharp shadow edges are not a problem). Therefore, these extra rays are sent out only *once in a while* (about 1 time in 50), then these calculated values are saved and reused for nearby pixels in the image when possible.

This process of saving and reusing values is what causes the need for a variety of tuning parameters, so you can get the scene to look just the way you want.

2.5.2 Adjusting Radiosity

As described earlier, radiosity is turned on by using the `radiosity{}` block in `global_setting`. Radiosity has many parameters that are specified as follows:

```
    global_settings { radiosity { [RADIOSITY_ITEMS...] } }
RADIOSITY_ITEMS:
    adc_bailout Float | always_sample Bool | brightness Float |
    count Integer | error_bound Float | gray_threshold Float |
    load_file Filename | low_error_factor Float | max_sample Float |
    media Bool | minimum_reuse Float | nearest_count Integer |
    normal Bool | pretrace_end Float | pretrace_start Float |
    recursion_limit Integer | save_file Filename
```

Each item is optional and may appear in any order. If an item is specified more than once the last setting overrides previous values. Details on each item is given in the following sections.

Note: Considerable changes have been made to the way radiosity works in POV-Ray 3.5 compared to POV-Ray 3.1. Old scene will not render to the same result, if they render at all. It is not possible to use the `#version` directive to get backward compatibility for radiosity.

radiosity adc_bailout

You can specify an adc_bailout for radiosity rays. Use `adc_bailout = 0.01 / brightest_ambient_object` for good results. Default is 0.01.

always_sample

You can force POV-Ray to only use the data from the pretrace step and not gather any new samples during the final radiosity pass. This may reduce splotchiness. To do this, use `always_sample off`, by default it is `on`. It can also be usefully when reusing previously saved radiosity data.

brightness

The `brightness` keyword specifies a float value that is the degree to which objects are brightened before being returned upwards to the rest of the system. The default value is 1.0. In cases where you would raise the `global_settings{ambient_light value}` to increase the over all brightness in a non-radiosity scene, you can use `brightness` in a radiosity scene.

count

The integer number of rays that are sent out whenever a new radiosity value has to be calculated is given by `count`. A value of 35 is the default, the maximum is 1600. When this value is too low, the light level will tend to look a little bit blotchy, as if the surfaces you're looking at were slightly warped. If this is not important to your scene (as in the case that you have a bump map or if you have a strong texture) then by all means use a lower number.

error_bound

The `error_bound` float value is one of the two main speed/quality tuning values (the other is of course the number of rays shot). In an ideal world, this would be the `only` value needed. It is intended to mean the fraction of error tolerated. For example, if it were set to 1 the algorithm would not calculate a new value until the error on the last one was estimated at as high as 100%. Ignoring the error introduced by rotation for the moment, on flat surfaces this is equal to the fraction of the reuse distance, which in turn is the distance to the closest item hit. If you have an old sample on the floor 10 inches from a wall, an error bound of 0.5 will get you a new sample at a distance of about 5 inches from the wall.

The default value of 1.8 is good for a smooth general lighting effect. Using lower values is more accurate, but it will strongly increase the danger of artifacts and therefore require higher `count`. You can use values even lower than 0.1 but both render time and memory use can become extremely high then.

gray_threshold

Diffusely interreflected light is a function of the objects around the point in question. Since this is recursively defined to millions of levels of recursion, in any real life scene, every point is illuminated at least in part by every other part of the scene. Since we can't afford to compute this, if we only do one bounce, the calculated ambient light

is very strongly affected by the colors of the objects near it. This is known as color bleed and it really happens but not as much as this calculation method would have you believe. The `gray_threshold` float value grays it down a little, to make your scene more believable. A value of .6 means to calculate the ambient value as 60% of the equivalent gray value calculated, plus 40% of the actual value calculated. At 0%, this feature does nothing. At 100%, you always get white/gray ambient light, with no hue.

Note: this does not change the lightness/darkness, only the strength of hue/grayness (in HLS terms, it changes S only). The default value is 0.0

low_error_factor

If you calculate just enough samples, but no more, you will get an image which has slightly blotchy lighting. What you want is just a few extra interspersed, so that the blending will be nice and smooth. The solution to this is the mosaic preview, controlled by `pretrace`: it goes over the image one or more times beforehand, calculating radiosity values. To ensure that you get a few extra, the radiosity algorithm lowers the error bound during the pre-final passes, then sets it back just before the final pass. The `low_error_factor` is a float tuning value which sets the amount that the error bound is dropped during the preliminary image passes. If your low error factor is 0.8 and your error bound is set to 0.4 it will really use an error bound of 0.32 during the first passes and 0.4 on the final pass. The default value is 0.5.

max_sample

Sometimes there can be problems with splotchiness that is caused by objects that are very bright. This can be sometimes avoided by using the `max_sample` keyword. `max_sample` takes a float parameter which specifies the brightest that any gathered sample is allowed to be. Any samples brighter than this will have their brightness decreased (without affecting color). Specifying a non-positive value for `max_sample` will allow any brightness of samples (which is the default).

Media and Radiosity

Radiosity estimation can be affected by media. To enable this feature, add `media on` to the `radiosity{}` block. The default is `off`

minimum_reuse

The minimum effective radius ratio is set by `minimum_reuse` float value. This is the fraction of the screen width which sets the minimum radius of reuse for each sample point (actually, it is the fraction of the distance from the eye but the two are roughly equal). For example, if the value is 0.02, the radius of maximum reuse for every sample is set to whatever ground distance corresponds to 2% of the width of the screen. Imagine you sent a ray off to the horizon and it hits the ground at a distance of 100 miles from your eye point. The reuse distance for that sample will be set to 2 miles. At a resolution of 300*400 this will correspond to (very roughly) 8 pixels. The theory is

that you don't want to calculate values for every pixel into every crevice everywhere in the scene, it will take too long. This sets a minimum bound for the reuse. If this value is too low, (which it should be in theory) rendering gets slow, and inside corners can get a little grainy. If it is set too high, you don't get the natural darkening of illumination near inside edges, since it reuses. At values higher than 2% you start getting more just plain errors, like reusing the illumination of the open table underneath the apple. Remember that this is a unit less ratio. The default value is 0.015.

nearest_count

The `nearest_count` integer value is the minimum number of old ambient values blended together to create a new interpolated value. The total number blended will vary depending on `error_bound`. All previous values that fit within the specified error_bound will be used in the average.

It will always be the n geometrically closest reusable points that get used. If you go lower than 4, things can get pretty patchy. This can be good for debugging, though. Must be no more than 20, since that is the size of the array allocated. The default value is 5.

Normal and Radiosity

Radiosity estimation can be affected by normals. To enable this feature, add `normal on` to the `radiosity{}` block. The default is `off`

Pretrace

To control the radiosity pre-trace gathering step, use the keywords `pretrace_start` and `pretrace_end` within the `radiosity{}` block. Each of these is followed by a decimal value between 0.0 and 1.0 which specifies the size of the blocks in the mosaic preview as a percentage of the image size. The defaults are 0.08 for `pretrace_start` and 0.04 for `pretrace_end`

recursion_limit

The `recursion_limit` is an integer value which determines how many recursion levels are used to calculate the diffuse inter-reflection. The default value is 3, the upper limit is 20.

Save and load radiosity data

You can save the radiosity data using `save_file "file_name"` and load the same data later using `load_file "file_name"`. In general, it is not a good idea to save and load radiosity data if scene objects are moving. Even if data are loaded, more samples may be

taken during rendering (which produces a better approximation). You can disable samples from being taken during the final rendering phase by specifying `always_sample off`.

2.5.3 Tips on Radiosity

Have a look at the "Radiosity Tutorial" in the "Advanced Tutorial" section, to get a feel for what the visual result of changing radiosity parameters is.

If you want to see where your values are being calculated set radiosity `count` down to about 20, set radiosity `nearest_count` to 1 and set `gray_threshold` to 0. This will make everything maximally patchy, so you'll be able to see the borders between patches. There will have been a radiosity calculation at the center of most patches. As a bonus, this is quick to run. You can then change the `error_bound` up and down to see how it changes things. Likewise modify `minimum_reuse`.

One way to get extra smooth results: crank up the sample count (we've gone as high as 1300) and drop the `low_error_factor` to something small like 0.6. Bump up the `nearest_count` to 7 or 8. This will get better values, and more of them, then interpolate among more of them on the last pass. This is not for people with a lack of patience since it is like a squared function. If your blotchiness is only in certain corners or near certain objects try tuning the error bound instead. Never drop it by more than a little at a time, since the run time will get very long.

Sometimes extra samples are taken during the final rendering pass. These newer samples can cause discontinuities in the radiosity in some scenes. To decrease these artefacts, use a pretrace_end of 0.04 (or even 0.02 if you're really patient and picky). This will cause the majority of the samples to be taken during the preview passes, and decrease the artefacts created during the final rendering pass. You can force POV-Ray to only use the data from the pretrace step and not gather any new samples during the final radiosity pass. To do this, use "always_sample no" within the radiosity block inside global_settings.

If your scene uses ambient objects (especially small ambient objects) as light sources, you should probably use a higher count (100-150 and higher). For such scenes, an error_bound of 1.0 is usually good. Higher causes too much error, but lower causes very slow rendering. And it's important to adapt adc_bailout.

Chapter 3

Objects

Objects are the building blocks of your scene. There are a lot of different types of objects supported by POV-Ray. In the sections which follows, we describe ”Finite Solid Primitives”, ”Finite Patch Primitives”, ”Infinite Solid Primitives”, ”Isosurface Object”, ”Parametric Object”, and ”Light Sources”. These primitive shapes may be combined into complex shapes using ”Constructive Solid Geometry” (also known as CSG).

The basic syntax of an object is a keyword describing its type, some floats, vectors or other parameters which further define its location and/or shape and some optional object modifiers such as texture, interior_texture, pigment, normal, finish, interior, bounding, clipping or transformations. Specifically the syntax is:

```
OBJECT:
    FINITE_SOLID_OBJECT | FINITE_PATCH_OBJECT |
    INFINITE_SOLID_OBJECT | ISOSURFACE_OBJECT | PARAMETRIC_OBJECT |
    CSG_OBJECT | LIGHT_SOURCE |
    object { OBJECT_IDENTIFIER [OBJECT_MODIFIERS...] }
FINITE_SOLID_OBJECT:
    BLOB | BOX | CONE | CYLINDER | HEIGHT_FIELD | JULIA_FRACTAL |
    LATHE | PRISM | SPHERE | SPHERESWEEP | SUPERELLIPSOID | SOR |
    TEXT | TORUS
FINITE_PATCH_OBJECT:
    BICUBIC_PATCH | DISC | MESH | MESH2 | POLYGON | TRIANGLE |
    SMOOTH_TRIANGLE
INFINITE_SOLID_OBJECT:
    PLANE | POLY | CUBIC | QUARTIC | QUADRIC
ISOSURFACE_OBJECT:
ISOSURFACE
PARAMETRIC_OBJECT:
PARAMETRIC
CSG_OBJECT:
    UNION | INTERSECTION | DIFFERENCE | MERGE
```

Object identifiers may be declared to make scene files more readable and to parameterize scenes so that changing a single declaration changes many values. An identifier is declared as follows.

```
OBJECT_DECLARATION:
    #declare IDENTIFIER = OBJECT |
    #local IDENTIFIER = OBJECT
```

Where *IDENTIFIER* is the name of the identifier up to 40 characters long and *OBJECT* is any valid object. To invoke an object identifier, you wrap it in an `object{...}` statement. You use the `object` statement regardless of what type of object it originally was. Although early versions of POV-Ray required this `object` wrapper all of the time, now it is only used with *OBJECT_IDENTIFIERS*.

Object modifiers are covered in detail later. However here is a brief overview.

The texture describes the surface properties of the object. Complete details are in "Textures". Textures are combinations of pigments, normals, and finishes. In the section "Pigment" you'll learn how to specify the color or pattern of colors inherent in the material. In "Normal" we describe a method of simulating various patterns of bumps, dents, ripples or waves by modifying the surface normal vector. The section on "Finish" describes the reflective properties of the surface. The "Interior" is a feature introduced in POV-Ray 3.1. It contains information about the interior of the object which was formerly contained in the finish and halo parts of a texture. Interior items are no longer part of the texture. Instead, they attach directly to the objects. The halo feature has been discontinued and replaced with a new feature called "Media" which replaces both halo and atmosphere.

Bounding shapes are finite, invisible shapes which wrap around complex, slow rendering shapes in order to speed up rendering time. Clipping shapes are used to cut away parts of shapes to expose a hollow interior. Transformations tell the ray-tracer how to move, size or rotate the shape and/or the texture in the scene.

3.1 Finite Solid Primitives

There are fourteen different solid finite primitive shapes: blob, box, cone, cylinder, height field, Julia fractal, lathe, prism, sphere, spheresweep, superellipsoid, surface of revolution, text object and torus. These have a well-defined *inside* and can be used in CSG (see section "Constructive Solid Geometry"). They are finite and respond to automatic bounding. You may specify an interior for these objects.

3.1.1 Blob

Blobs are an interesting and flexible object type. Mathematically they are iso-surfaces of scalar fields, i.e. their surface is defined by the strength of the field in each point. If this strength is equal to a threshold value you're on the surface otherwise you're not.

Picture each blob component as an object floating in space. This object is *filled* with a field that has its maximum at the center of the object and drops off to zero at the object's surface. The field strength of all those components are added together to form the field of the blob. Now POV-Ray looks for points where this field has a given value, the threshold value. All these points form the surface of the blob object. Points with a greater field value than the threshold value are considered to be inside while points with a smaller field value are outside.

There's another, simpler way of looking at blobs. They can be seen as a union of flexible components that attract or repel each other to form a blobby organic looking shape. The components' surfaces actually stretch out smoothly and connect as if they were made of honey or something similar.

The syntax for `blob` is defined as follows:

```
BLOB:
    blob { BLOB_ITEM... [BLOB_MODIFIERS...]}
BLOB_ITEM:
    sphere{<Center>, Radius,
           [ strength ] Strength[COMPONENT_MODIFIER...] } |
    cylinder{<End1>, <End2>, Radius,
             [ strength ] Strength [COMPONENT_MODIFIER...] } |
    component Strength, Radius, <Center> |
    threshold Amount
COMPONENT_MODIFIER:
    TEXTURE | PIGMENT | NORMAL | FINISH | TRANSFORMATION
BLOB_MODIFIER:
    hierarchy [Boolean] | sturm [Boolean] | OBJECT_MODIFIER
```

Blob default values:

```
hierarchy : on
sturm     : off
threshold : 1.0
```

The `threshold` keyword is followed by a float value which determines the total field strength value that POV-Ray is looking for. The default value if none is specified is `threshold 1.0`. By following the ray out into space and looking at how each blob component affects the ray, POV-Ray will find the points in space where the field strength is equal to the threshold value. The following list shows some things you should know about the threshold value.

1. The threshold value must be positive.
2. A component disappears if the threshold value is greater than its strength.
3. As the threshold value gets larger, the surface you see gets closer to the centers of the components.
4. As the threshold value gets smaller, the surface you see gets closer to the surface of the components.

Cylindrical components are specified by a `cylinder` statement. The center of the end-caps of the cylinder is defined by the vectors <*End1*> and <*End2*>. Next is the float value of the *Radius* followed by the float *Strength*. These vectors and floats are required and should be separated by commas. The keyword `strength` may optionally precede the strength value. The cylinder has hemispherical caps at each end.

Spherical components are specified by a `sphere` statement. The location is defined by the vector <*Center*>. Next is the float value of the *Radius* followed by the float *Strength*. These vector and float values are required and should be separated by commas. The keyword `strength` may optionally precede the strength value.

You usually will apply a single texture to the entire blob object, and you typically use transformations to change its size, location, and orientation. However both the

`cylinder` and `sphere` statements may have individual texture, pigment, normal, finish, and transformations applied to them. You may not apply separate `interior` statements to the components but you may specify one for the entire blob.

Note: by unevenly scaling a spherical component you can create ellipsoidal components. The tutorial section on "Blob Object" illustrates individually textured blob components and many other blob examples.

The `component` keyword is an obsolete method for specifying a spherical component and is only used for compatibility with earlier POV-Ray versions. It may not have textures or transformations individually applied to it.

The `strength` parameter of either type of blob component is a float value specifying the field strength at the center of the object. The strength may be positive or negative. A positive value will make that component attract other components while a negative value will make it repel other components. Components in different, separate blob shapes do not affect each other.

You should keep the following things in mind.

1. The strength value may be positive or negative. Zero is a bad value, as the net result is that no field was added – you might just as well have not used this component.
2. If strength is positive, then POV-Ray will add the component's field to the space around the center of the component. If this adds enough field strength to be greater than the threshold value you will see a surface.
3. If the strength value is negative, then POV-Ray will subtract the component's field from the space around the center of the component. This will only do something if there happen to be positive components nearby. The surface around any nearby positive components will be dented away from the center of the negative component.

After all components and the optional `threshold` value have been specified you may specify zero or more blob modifiers. A blob modifier is any regular object modifier or the `hierarchy` or `sturm` keywords.

The components of each blob object are internally bounded by a spherical bounding hierarchy to speed up blob intersection tests and other operations. Using the optional keyword `hierarchy` followed by an optional boolean float value will turn it off or on. By default it is on.

The calculations for blobs must be very accurate. If this shape renders improperly you may add the keyword `sturm` followed by an optional boolean float value to turn off or on POV-Ray's slower-yet-more-accurate Sturmian root solver. By default it is off.

An example of a three component blob is:

```
BLOB:
 blob {
  threshold 0.6
  sphere { <.75, 0, 0>, 1, 1 }
  sphere { <-.375, .64952, 0>, 1, 1 }
  sphere { <-.375, -.64952, 0>, 1, 1 }
  scale 2
```

```
}
```

If you have a single blob component then the surface you see will just look like the object used, i.e. a sphere or a cylinder, with the surface being somewhere inside the surface specified for the component. The exact surface location can be determined from the blob equation listed below (you will probably never need to know this, blobs are more for visual appeal than for exact modeling).

For the more mathematically minded, here's the formula used internally by POV-Ray to create blobs. You don't need to understand this to use blobs. The density of the blob field of a single component is:

$$density = strength \cdot \left(1 - \left(\frac{distance}{radius}\right)^2\right)^2$$

Equation 3.1: Density of a blob field.

where *distance* is the distance of a given point from the spherical blob's center or cylinder blob's axis. This formula has the nice property that it is exactly equal to the strength parameter at the center of the component and drops off to exactly 0 at a distance from the center of the component that is equal to the radius value. The density formula for more than one blob component is just the sum of the individual component densities.

3.1.2 Box

A simple box can be defined by listing two corners of the box using the following syntax for a `box` statement:

```
BOX:
    box
    {
        <Corner_1>, <Corner_2>
        [OBJECT_MODIFIERS...]
    }
```

Where <*Corner_1*> and <*Corner_2*> are vectors defining the x, y, z coordinates of the opposite corners of the box.

Note: that all boxes are defined with their faces parallel to the coordinate axes. They may later be rotated to any orientation using the `rotate` keyword.

Boxes are calculated efficiently and make good bounding shapes (if manually bounding seems to be necessary).

3.1.3 Cone

The `cone` statement creates a finite length cone or a *frustum* (a cone with the point cut off). The syntax is:

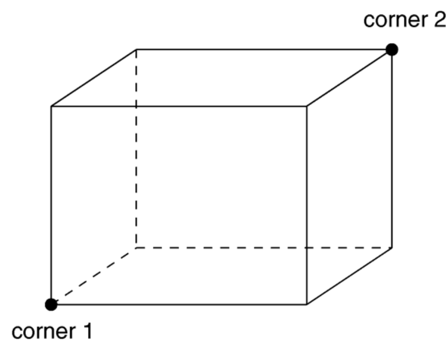

Figure 3.1: The geometry of a box.

```
CONE:
    cone
    {
        <Base_Point>, Base_Radius, <Cap_Point>, Cap_Radius
        [ open ][OBJECT_MODIFIERS...]
    }
```

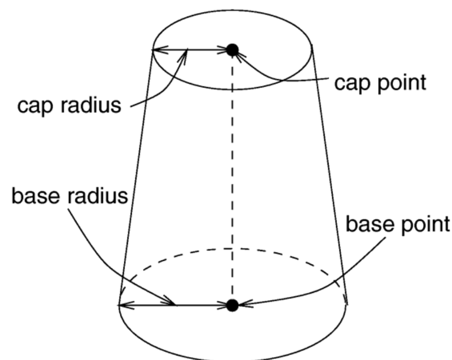

Figure 3.2: The geometry of a cone.

Where <*Base_Point*> and < *Cap_Point*> are vectors defining the x, y, z coordinates of the center of the cone's base and cap and *Base_Radius* and *Cap_Radius* are float values for the corresponding radii.

Normally the ends of a cone are closed by flat discs that are parallel to each other and perpendicular to the length of the cone. Adding the optional keyword `open` after *Cap_Radius* will remove the end caps and results in a tapered hollow tube like a megaphone or funnel.

3.1.4 Cylinder

The `cylinder` statement creates a finite length cylinder with parallel end caps The syntax is:

```
CYLINDER:
    cylinder
    {
        <Base_Point>, <Cap_Point>, Radius
        [ open ][OBJECT_MODIFIERS...]
    }
```

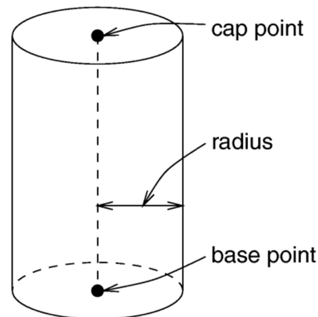

Figure 3.3: The geometry of a cylinder.

Where <*Base_Point*> and <*Cap_Point*> are vectors defining the x, y, z coordinates of the cylinder's base and cap and *Radius* is a float value for the radius.

Normally the ends of a cylinder are closed by flat discs that are parallel to each other and perpendicular to the length of the cylinder. Adding the optional keyword `open` after the radius will remove the end caps and results in a hollow tube.

3.1.5 Height Field

Height fields are fast, efficient objects that are generally used to create mountains or other raised surfaces out of hundreds of triangles in a mesh. The `height_field` statement syntax is:

```
HEIGHT_FIELD:
    height_field{
      [HF_TYPE]
      "filename"
      [HF_MODIFIER...]
      [OBJECT_MODIFIER...]
    }
HF_TYPE:
    gif | tga | pot | png | pgm | ppm | jpeg | tiff | sys | function image
HF_MODIFIER:
    hierarchy [Boolean]  |
    smooth               |
    water_level Level
```

Height_field default values:

```
hierarchy   : on
smooth      : off
```

```
water_level : 0.0
```

A height field is essentially a one unit wide by one unit long square with a mountainous surface on top. The height of the mountain at each point is taken from the color number or palette index of the pixels in a graphic image file. The maximum height is one, which corresponds to the maximum possible color or palette index value in the image file.

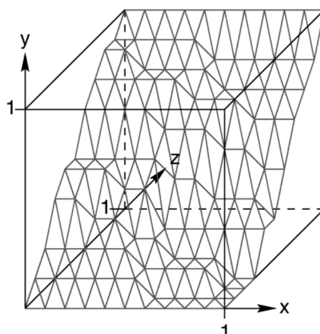

Figure 3.4: The size and orientation of an un-scaled height field.

The mesh of triangles corresponds directly to the pixels in the image file. Each square formed by four neighboring pixels is divided into two triangles. An image with a resolution of `N*M` pixels has `(N-1)*(M-1)` squares that are divided into `2*(N-1)*(M-1)` triangles.

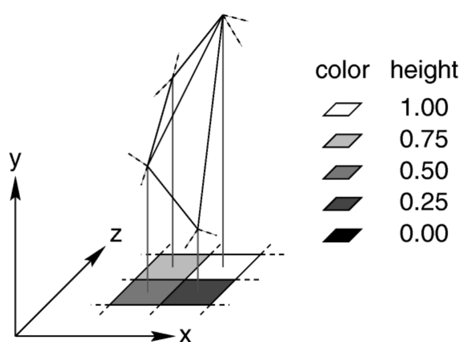

Figure 3.5: Relationship of pixels and triangles in a height field.

The resolution of the height field is influenced by two factors: the resolution of the image and the resolution of the color/index values. The size of the image determines the resolution in the x- and z-direction. A larger image uses more triangles and looks smoother. The resolution of the color/index value determines the resolution along the y-axis. A height field made from an 8-bit image can have 256 different height levels while one made from a 16-bit image can have up to 65536 different height levels. Thus the second height field will look much smoother in the y-direction if the height field is created appropriately.

The size/resolution of the image does not affect the size of the height field. The un-

scaled height field size will always be 1 by 1 by 1. Higher resolution image files will create smaller triangles, not larger height fields.

There are eight or possibly nine types of files which can define a height field. The image file type used to create a height field is specified by one of the keywords `gif`, `tga`, `pot`, `png`, `pgm`, `ppm`, `tiff`, `jpeg` and possibly `sys` which is a system specific (e. g. Windows BMP or Macintosh Pict) format file. Specifying the file type is optional. If it is not defined the same file type will be assumed as the one that is set as the output file type. This is useful when the source for the `height_field` is also generated with POV-Ray.

The GIF, PNG, PGM, TIFF and possibly SYS format files are the only ones that can be created using a standard paint program. Though there are paint programs for creating TGA image files they won't be of much use for creating the special 16 bit TGA files used by POV-Ray (see below and "HF_Gray_16" for more details).

In an image file that uses a color palette, like GIF, the color number is the palette index at a given pixel. Use a paint program to look at the palette of a GIF image. The first color is palette index zero, the second is index one, the third is index two and so on. The last palette entry is index 255. Portions of the image that use low palette entries will result in lower parts of the height field. Portions of the image that use higher palette entries will result in higher parts of the height field.

Height fields created from GIF files can only have 256 different height levels because the maximum number of colors in a GIF file is 256.

The color of the palette entry does not affect the height of the pixel. Color entry 0 could be red, blue, black or orange but the height of any pixel that uses color entry 0 will always be 0. Color entry 255 could be indigo, hot pink, white or sky blue but the height of any pixel that uses color entry 255 will always be 1.

You can create height field GIF images with a paint program or a fractal program like `Fractint`. You can usually get `Fractint` from most of the same sources as POV-Ray.

A POT file is essentially a GIF file with a 16 bit palette. The maximum number of colors in a POT file is 65536. This means a POT height field can have up to 65536 possible height values. This makes it possible to have much smoother height fields.

Note: the maximum height of the field is still 1 even though more intermediate values are possible.

At the time of this writing the only program that created POT files was a freeware MS-Dos/Windows program called `Fractint`. POT files generated with this fractal program create fantastic landscapes.

The TGA and PPM file formats may be used as a storage device for 16 bit numbers rather than an image file. These formats use the red and green bytes of each pixel to store the high and low bytes of a height value. These files are as smooth as POT files but they must be generated with special custom-made programs. Several programs can create TGA heightfields in the format POV uses, such as `Gforge` and `Terrain Maker`.

PNG format heightfields are usually stored in the form of a grayscale image with black corresponding to lower and white to higher parts of the height field. Because PNG files can store up to 16 bits in grayscale images they will be as smooth as TGA and PPM images. Since they are grayscale images you will be able to view them with a

regular image viewer. gforge can create 16-bit heightfields in PNG format. Color PNG images will be used in the same way as TGA and PPM images.

SYS format is a platform specific file format. See your platform specific documentation for details.

In addition to all the usual object modifiers, there are three additional height field modifiers available.

The optional water_level parameter may be added after the file name. It consists of the keyword water_level followed by a float value telling the program to ignore parts of the height field below that value. The default value is zero and legal values are between zero and one. For example water_level 0.5 tells POV-Ray to only render the top half of the height field. The other half is *below the water* and couldn't be seen anyway. Using water_level renders faster than cutting off the lower part using CSG or clipping. This term comes from the popular use of height fields to render landscapes. A height field would be used to create islands and another shape would be used to simulate water around the islands. A large portion of the height field would be obscured by the water so the water_level parameter was introduced to allow the ray-tracer to ignore the unseen parts of the height field. water_level is also used to cut away unwanted lower values in a height field. For example if you have an image of a fractal on a solid colored background, where the background color is palette entry 0, you can remove the background in the height field by specifying, water_level 0.001.

Normally height fields have a rough, jagged look because they are made of lots of flat triangles. Adding the keyword smooth causes POV-Ray to modify the surface normal vectors of the triangles in such a way that the lighting and shading of the triangles will give a smooth look. This may allow you to use a lower resolution file for your height field than would otherwise be needed. However, smooth triangles will take longer to render. The default value is off.

In order to speed up the intersection tests a one-level bounding hierarchy is available. By default it is always used but it can be switched off using hierarchy off to improve the rendering speed for small height fields (i.e. low resolution images). You may optionally use a boolean value such as hierarchy on or hierarchy off.

3.1.6 Julia Fractal

A *julia fractal* object is a 3-D *slice* of a 4-D object created by generalizing the process used to create the classic Julia sets. You can make a wide variety of strange objects using the julia_fractal statement including some that look like bizarre blobs of twisted taffy. The julia_fractal syntax is:

```
JULIA_FRACTAL:
    julia_fractal
    {
        <4D_Julia_Parameter>
        [JF_ITEM...] [OBJECT_MODIFIER...]
    }
JF_ITEM:
    ALGEBRA_TYPE | FUNCTION_TYPE | max_iteration Count |
    precision Amt | slice <4D_Normal>, Distance
```

```
ALGEBRA_TYPE:
    quaternion | hypercomplex
FUNCTION_TYPE:
    QUATERNATION:
         sqr | cube
    HYPERCOMPLEX:
         sqr | cube | exp | reciprocal | sin | asin | sinh |
         asinh | cos | acos | cosh | acosh | tan | atan |tanh |
         atanh | ln | pwr( X_Val, Y_Val )
```

Julia Fractal default values:

```
ALGEBRA_TYPE    : quaternion
FUNCTION_TYPE   : sqr
max_iteration   : 20
precision       : 20
slice, DISTANCE : <0,0,0,1>, 0.0
```

The required 4-D vector `<4D_Julia_Parameter>` is the classic Julia parameter `p` in the iterated formula `f(h) + p`. The julia fractal object is calculated by using an algorithm that determines whether an arbitrary point `h(0)` in 4-D space is inside or outside the object. The algorithm requires generating the sequence of vectors `h(0), h(1), ...` by iterating the formula `h(n+1) = f(h(n)) + p (n = 0, 1, ..., max_iteration-1)` where `p` is the fixed 4-D vector parameter of the julia fractal and `f()` is one of the functions `sqr`, `cube`, ... specified by the presence of the corresponding keyword. The point `h(0)` that begins the sequence is considered inside the julia fractal object if none of the vectors in the sequence escapes a hypersphere of radius 4 about the origin before the iteration number reaches the integer `max_iteration` value. As you increase `max_iteration`, some points escape that did not previously escape, forming the julia fractal. Depending on the `<4D_Julia_Parameter>`, the julia fractal object is not necessarily connected; it may be scattered fractal dust. Using a low `max_iteration` can fuse together the dust to make a solid object. A high `max_iteration` is more accurate but slows rendering. Even though it is not accurate, the solid shapes you get with a low `max_iteration` value can be quite interesting. If none is specified, the default is `max_iteration 20`.

Since the mathematical object described by this algorithm is four-dimensional and POV-Ray renders three dimensional objects, there must be a way to reduce the number of dimensions of the object from four dimensions to three. This is accomplished by intersecting the 4-D fractal with a 3-D "plane" defined by the `slice` modifier and then projecting the intersection to 3-D space. The keyword is followed by 4-D vector and a float separated by a comma. The slice plane is the 3-D space that is perpendicular to `<4D_Normal>` and is `Distance` units from the origin. Zero length `<4D_Normal>` vectors or a `<4D_Normal>` vector with a zero fourth component are illegal. If none is specified, the default is `slice <0,0,0,1>,0`.

You can get a good feel for the four dimensional nature of a julia fractal by using POV-Ray's animation feature to vary a slice's `Distance` parameter. You can make the julia fractal appear from nothing, grow, then shrink to nothing as `Distance` changes, much as the cross section of a 3-D object changes as it passes through a plane.

The `precision` parameter is a tolerance used in the determination of whether points are inside or outside the fractal object. Larger values give more accurate results but slower rendering. Use as low a value as you can without visibly degrading the fractal

object's appearance but note values less than 1.0 are clipped at 1.0. The default if none is specified is `precision 20`.

The presence of the keywords `quaternion` or `hypercomplex` determine which 4-D algebra is used to calculate the fractal. The default is `quaternion`. Both are 4-D generalizations of the complex numbers but neither satisfies all the field properties (all the properties of real and complex numbers that many of us slept through in high school). Quaternions have non-commutative multiplication and hypercomplex numbers can fail to have a multiplicative inverse for some non-zero elements (it has been proved that you cannot successfully generalize complex numbers to four dimensions with all the field properties intact, so something has to break). Both of these algebras were discovered in the 19th century. Of the two, the quaternions are much better known, but one can argue that hypercomplex numbers are more useful for our purposes, since complex valued functions such as sin, cos, etc. can be generalized to work for hypercomplex numbers in a uniform way.

For the mathematically curious, the algebraic properties of these two algebras can be derived from the multiplication properties of the unit basis vectors 1 = <1,0,0,0>, i=< 0,1,0,0>, j=<0,0,1,0> and k=< 0,0,0,1>. In both algebras 1 x = x 1 = x for any x (1 is the multiplicative identity). The basis vectors 1 and i behave exactly like the familiar complex numbers 1 and i in both algebras.

```
ij = k              jk = i     ki = j
ji = -k             kj = -i    ik = -j
ii = jj = kk = -1   ijk = -1
```

Table 3.1: Quaternion basis vector multiplication rules

```
ij = k              jk = -i   ki = -j
ji = k              kj = -i   ik = -j
ii = jj = kk = -1   ijk = 1
```

Table 3.2: Hypercomplex basis vector multiplication rules

A distance estimation calculation is used with the quaternion calculations to speed them up. The proof that this distance estimation formula works does not generalize from two to four dimensions but the formula seems to work well anyway, the absence of proof notwithstanding!

The presence of one of the function keywords `sqr`, `cube`, etc. determines which function is used for `f(h)` in the iteration formula `h(n+1) = f(h(n)) + p`. The default is `sqr`. Most of the function keywords work only if the `hypercomplex` keyword is present. Only `sqr` and `cube` work with `quaternion`. The functions are all familiar complex functions generalized to four dimensions. Function Keyword Maps 4-D value h to:

A simple example of a julia fractal object is:

```
julia_fractal {
 <-0.083,0.0,-0.83,-0.025>
 quaternion
 sqr
 max_iteration 8
 precision 15
```

sqr	h*h
cube	h*h*h
exp	e raised to the power h
reciprocal	1/h
sin	sine of h
asin	arcsine of h
sinh	hyperbolic sine of h
asinh	inverse hyperbolic sine of h
cos	cosine of h
acos	arccosine of h
cosh	hyperbolic cos of h
acosh	inverse hyperbolic cosine of h
tan	tangent of h
atan	arctangent of h
tanh	hyperbolic tangent of h
atanh	inverse hyperbolic tangent of h
ln	natural logarithm of h
pwr(x,y)	h raised to the complex power x+iy

Table 3.3: Function Keyword Maps 4-D value of h

```
}
```

The first renderings of julia fractals using quaternions were done by Alan Norton and later by John Hart in the '80's. This POV-Ray implementation follows Fractint in pushing beyond what is known in the literature by using hypercomplex numbers and by generalizing the iterating formula to use a variety of transcendental functions instead of just the classic Mandelbrot $z2 + c$ formula. With an extra two dimensions and eighteen functions to work with, intrepid explorers should be able to locate some new fractal beasts in hyperspace, so have at it!

3.1.7 Lathe

The lathe is an object generated from rotating a two-dimensional curve about an axis. This curve is defined by a set of points which are connected by linear, quadratic, cubic or bezier spline curves. The syntax is:

```
LATHE:
    lathe
    {
        [SPLINE_TYPE] Number_Of_Points, <Point_1>
        <Point_2>... <Point_n>
        [LATHE_MODIFIER...]
    }
SPLINE_TYPE:
    linear_spline | quadratic_spline | cubic_spline | bezier_spline
LATHE_MODIFIER:
    sturm | OBJECT_MODIFIER
```

Lathe default values:

```
SPLINE_TYPE   : linear_spline
sturm         : off
```

The first item is a keyword specifying the type of spline. The default if none is specified is `linear_spline`. The required integer value *`Number_Of_Points`* specifies how many two-dimensional points are used to define the curve. The points follow and are specified by 2-D vectors. The curve is not automatically closed, i.e. the first and last points are not automatically connected. You will have to do this yourself if you want a closed curve. The curve thus defined is rotated about the y-axis to form the lathe object, centered at the origin.

The following examples creates a simple lathe object that looks like a thick cylinder, i.e. a cylinder with a thick wall:

```
lathe {
 linear_spline
 5,
 <2, 0>, <3, 0>, <3, 5>, <2, 5>, <2, 0>
 pigment {Red}
}
```

The cylinder has an inner radius of 2 and an outer radius of 3, giving a wall width of 1. It's height is 5 and it's located at the origin pointing up, i.e. the rotation axis is the y-axis.

Note: the first and last point are equal to get a closed curve.

The splines that are used by the lathe and prism objects are a little bit difficult to understand. The basic concept of splines is to draw a curve through a given set of points in a determined way. The default `linear_spline` is the simplest spline because it's nothing more than connecting consecutive points with a line. This means the curve that is drawn between two points only depends on those two points. No additional information is taken into account. The other splines are different in that they do take other points into account when connecting two points. This creates a smooth curve and, in the case of the cubic spline, produces smoother transitions at each point.

The `quadratic_spline` keyword creates splines that are made of quadratic curves. Each of them connects two consecutive points. Since those two points (call them second and third point) are not sufficient to describe a quadratic curve, the predecessor of the second point is taken into account when the curve is drawn. Mathematically, the relationship (their relative locations on the 2-D plane) between the first and second point determines the slope of the curve at the second point. The slope of the curve at the third point is out of control. Thus quadratic splines look much smoother than linear splines but the transitions at each point are generally not smooth because the slopes on both sides of the point are different.

The `cubic_spline` keyword creates splines which overcome the transition problem of quadratic splines because they also take a fourth point into account when drawing the curve between the second and third point. The slope at the fourth point is under control now and allows a smooth transition at each point. Thus cubic splines produce the most flexible and smooth curves.

The `bezier_spline` is an alternate kind of cubic spline. Points 1 and 4 specify the end points of a segment and points 2 and 3 are control points which specify the slope at the endpoints. Points 2 and 3 do not actually lie on the spline. They adjust the slope

of the spline. If you draw an imaginary line between point 1 and 2, it represents the slope at point 1. It is a line tangent to the curve at point 1. The greater the distance between 1 and 2, the flatter the curve. With a short tangent the spline can bend more. The same holds true for control point 3 and endpoint 4. If you want the spline to be smooth between segments, points 3 and 4 on one segment and points 1 and 2 on the next segment must form a straight line and point 4 of one segment must be the same as point 1 on the next segment.

You should note that the number of spline segments, i. e. curves between two points, depends on the spline type used. For linear splines you get n-1 segments connecting the points P[i], i=1,...,n. A quadratic spline gives you n-2 segments because the last point is only used for determining the slope, as explained above (thus you'll need at least three points to define a quadratic spline). The same holds for cubic splines where you get n-3 segments with the first and last point used only for slope calculations (thus needing at least four points). The bezier spline requires 4 points per segment, creating n/4 segments.

If you want to get a closed quadratic and cubic spline with smooth transitions at the end points you have to make sure that in the cubic case P[n-1] = P[2] (to get a closed curve), P[n] = P[3] and P[n-2] = P[1] (to smooth the transition). In the quadratic case P[n-1] = P[1] (to close the curve) and P[n] = P[2].

The `sturm` keyword can be used to specify that the slower, but more accurate, Sturmian root solver should be used. Use it, if the shape does not render properly. Since a quadratic polynomial has to be solved for the linear spline lathe, the Sturmian root solver is not needed.

3.1.8 Prism

The `prism` is an object generated specifying one or more two-dimensional, closed curves in the x-z plane and sweeping them along y axis. These curves are defined by a set of points which are connected by linear, quadratic, cubic or bezier splines. The syntax for the prism is:

```
PRISM:
    prism
    {
        [PRISM_ITEMS...] Height_1, Height_2, Number_Of_Points,
        <Point_1>, <Point_2>, ... <Point_n>
        [ open ] [PRISM_MODIFIERS...]
    }
PRISM_ITEM:
    linear_spline | quadratic_spline | cubic_spline |
    bezier_spline | linear_sweep | conic_sweep
PRISM_MODIFIER:
    sturm | OBJECT_MODIFIER
```

Prism default values:

```
SPLINE_TYPE   : linear_spline
SWEEP_TYPE    : linear_sweep
sturm         : off
```

The first items specify the spline type and sweep type. The defaults if none is specified is linear_spline and linear_sweep. This is followed by two float values *Height_1* and *Height_2* which are the y coordinates of the top and bottom of the prism. This is followed by a float value specifying the number of 2-D points you will use to define the prism. (This includes all control points needed for quadratic, cubic and bezier splines). This is followed by the specified number of 2-D vectors which define the shape in the x-z plane.

The interpretation of the points depends on the spline type. The prism object allows you to use any number of sub-prisms inside one prism statement (they are of the same spline and sweep type). Wherever an even number of sub-prisms overlaps a hole appears.

Note: you need not have multiple sub-prisms and they need not overlap as these examples do.

In the linear_spline the first point specified is the start of the first sub-prism. The following points are connected by straight lines. If you specify a value identical to the first point, this closes the sub-prism and next point starts a new one. When you specify the value of that sub-prism's start, then it is closed. Each of the sub-prisms has to be closed by repeating the first point of a sub-prism at the end of the sub-prism's point sequence. In this example, there are two rectangular sub-prisms nested inside each other to create a frame.

```
prism {
 linear_spline
 0, 1, 10,
 <0,0>, <6,0>, <6,8>, <0,8>, <0,0>,  //outer rim
 <1,1>, <5,1>, <5,7>, <1,7>, <1,1>   //inner rim
}
```

The last sub-prism of a linear spline prism is automatically closed - just like the last sub-polygon in the polygon statement - if the first and last point of the sub-polygon's point sequence are not the same. This make it very easy to convert between polygons and prisms. Quadratic, cubic and bezier splines are never automatically closed.

In the quadratic_spline, each sub-prism needs an additional control point at the beginning of each sub-prisms' point sequence to determine the slope at the start of the curve. The first point specified is the control point which is not actually part of the spline. The second point is the start of the spline. The sub-prism ends when this second point is duplicated. The next point is the control point of the next sub-prism. The point after that is the first point of the second sub-prism. Here is an example:

```
prism {
  quadratic_spline
  0, 1, 12,
  <1,-1>, <0,0>, <6,0>, //outer rim; <1,-1> is control point and
  <6,8>, <0,8>, <0,0>,  //<0,0> is first & last point

  <2,0>, <1,1>, <5,1>,  //inner rim; <2,0> is control point and
  <5,7>, <1,7>, <1,1>   //<1,1> is first & last point
}
```

In the cubic_spline, each sub-prism needs two additional control points – one at the beginning of each sub-prisms' point sequence to determine the slope at the start of the curve and one at the end. The first point specified is the control point which is not

actually part of the spline. The second point is the start of the spline. The sub-prism ends when this second point is duplicated. The next point is the control point of the end of the first sub-prism. Next is the beginning control point of the next sub-prism. The point after that is the first point of the second sub-prism.

Here is an example:

```
prism {
  cubic_spline
  0, 1, 14,
  <1,-1>, <0,0>, <6,0>, //outer rim; First control is <1,-1> and
  <6,8>, <0,8>, <0,0>,  //<0,0> is first & last point.
  <-1,1>,               //Last control of first spline is <-1,1>

  <2,0>, <1,1>, <5,1>,  //inner rim; First control is <2,0> and
  <5,7>, <1,7>, <1,1>,  //<1,1> is first & last point
  <0,2>                 //Last control of first spline is <0,2>
}
```

The `bezier_spline` is an alternate kind of cubic spline. Points 1 and 4 specify the end points of a segment and points 2 and 3 are control points which specify the slope at the endpoints. Points 2 and 3 do not actually lie on the spline. They adjust the slope of the spline. If you draw an imaginary line between point 1 and 2, it represents the slope at point 1. It is a line tangent to the curve at point 1. The greater the distance between 1 and 2, the flatter the curve. With a short tangent the spline can bend more. The same holds true for control point 3 and endpoint 4. If you want the spline to be smooth between segments, point 3 and 4 on one segment and point 1 and 2 on the next segment must form a straight line and point 4 of one segment must be the same as point one on the next segment.

By default linear sweeping is used to create the prism, i.e. the prism's walls are perpendicular to the x-z-plane (the size of the curve does not change during the sweep). You can also use `conic_sweep` that leads to a prism with cone-like walls by scaling the curve down during the sweep.

Like cylinders the prism is normally closed. You can remove the caps on the prism by using the `open` keyword. If you do so you shouldn't use it with CSG because the results may get wrong.

For an explanation of the spline concept read the description of the "Lathe" object. Also see the tutorials on "Lathe Object" and "Prism Object".

The `sturm` keyword specifies the slower but more accurate Sturmian root solver which may be used with the cubic or bezier spline prisms if the shape does not render properly. The linear and quadratic spline prisms do not need the Sturmian root solver.

3.1.9 Sphere

The syntax of the `sphere` object is:

```
SPHERE:
    sphere
    {
        <Center>, Radius
```

```
    [OBJECT_MODIFIERS...]
}
```

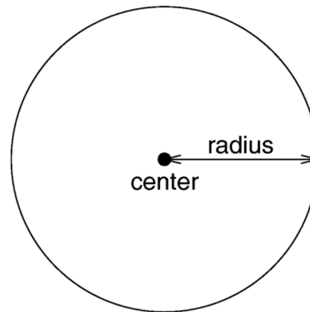

Figure 3.6: The geometry of a sphere.

Where <*Center*> is a vector specifying the x, y, z coordinates of the center of the sphere and *Radius* is a float value specifying the radius. Spheres may be scaled unevenly giving an ellipsoid shape.

Because spheres are highly optimized they make good bounding shapes (if manual bounding seems to be necessary).

3.1.10 Spheresweep

The syntax of the `sphere_sweep` object is:

```
SPHERE_SWEEP:
sphere_sweep {
  linear_spline | b_spline | cubic_spline
  NUM_OF_SPHERES,

  CENTER, RADIUS,
  CENTER, RADIUS,
  ...
  CENTER, RADIUS
  [tolerance DEPTH_TOLERANCE]
  [OBJECT_MODIFIERS]
}
```

Sphere_sweep default values:

```
tolerance : 1.0e-6 (0.000001)
```

A Sphere Sweep is the envelope of a moving sphere with varying radius, or, in other words, the space a sphere occupies during its movement along a spline.
Sphere Sweeps are modeled by specifying a list of single spheres which are then interpolated.
Three kinds of interpolation are supported:

- `linear_spline` : Interpolating the input data with a linear function, which means that the single spheres are connected by straight tubes.
- `b_spline` : Approximating the input data using a cubic b-spline function, which results in a curved object.
- `cubic_spline` : Approximating the input data using a cubic spline, which results in a curved object.

The sphere list (center and radius of each sphere) can take as many spheres as you like to describe the object, but you will need at least two spheres for a `linear_spline`, and four spheres for `b_spline` or `cubic_spline`.

Optional: The depth tolerance that should be used for the intersection calculations. This is done by adding the `tolerance` keyword and the desired value: the default distance is 1.0e-6 (0.000001) and should do for most sphere sweep objects.
You should change this when you see dark spots on the surface of the object. These are probably caused by an effect called "Self-Shading". This means that the object casts shadows onto itself at some points because of calculation errors. A ray tracing program usually defines the minimal distance a ray must travel before it actually hits another (or the same) object to avoid this effect. If this distance is chosen too small, Self-Shading may occur.
If so, specify `tolerance 1.0e-4` or higher.

Note: if these dark spots remain after raising the tolerance, you might get rid of these spots by using Adaptive Supersampling (Method 2) for antialiasing. Images look better with antialiasing anyway.

Note: the merge CSG operation is not recommended with Sphere Sweeps: there could be a small gap between the merged objects!

3.1.11 Superquadric Ellipsoid

The `superellipsoid` object creates a shape known as a *superquadric ellipsoid* object. It is an extension of the quadric ellipsoid. It can be used to create boxes and cylinders with round edges and other interesting shapes. Mathematically it is given by the equation:

$$f(x,y,z) = \left(|x|^{\left(\frac{2}{e}\right)} + |y|^{\left(\frac{2}{e}\right)}\right)^{\left(\frac{e}{n}\right)} + |z|^{\left(\frac{2}{n}\right)} - 1 = 0$$

Equation 3.2:

The values of `e` and `n`, called the *east-west* and *north-south* exponent, determine the shape of the superquadric ellipsoid. Both have to be greater than zero. The sphere is given by *e = 1* and *n = 1*.

The syntax of the superquadric ellipsoid is:

```
SUPERELLIPSOID:
    superellipsoid
    {
```

```
        <Value_E, Value_N>
        [OBJECT_MODIFIERS...]
    }
```

The 2-D vector specifies the `e` and `n` values in the equation above. The object sits at the origin and occupies a space about the size of a `box{<-1,-1,-1>,<1,1,1>}`.

Two useful objects are the rounded box and the rounded cylinder. These are declared in the following way.

```
#declare Rounded_Box = superellipsoid { <Round, Round> }
#declare Rounded_Cylinder = superellipsoid { <1, Round> }
```

The roundedness value `Round` determines the roundedness of the edges and has to be greater than zero and smaller than one. The smaller you choose the values, the smaller and sharper the edges will get.

Very small values of `e` and `n` might cause problems with the root solver (the Sturmian root solver cannot be used).

3.1.12 Surface of Revolution

The `sor` object is a *surface of revolution* generated by rotating the graph of a function about the y-axis. This function describes the dependence of the radius from the position on the rotation axis. The syntax is:

```
SOR:
    sor
    {
        Number_Of_Points, <Point_1>, <Point_2>, ... <Point_n>
        [ open ] [SOR_MODIFIERS...]
    }
SOR_MODIFIER:
    sturm | OBJECT_MODIFIER
```

SOR default values:

```
sturm : off
```

The float value *`Number_Of_Points`* specifies the number of 2-D vectors which follow. The points *`<Point_1>`* through *`<Point_n>`* are two-dimensional vectors consisting of the radius and the corresponding height, i.e. the position on the rotation axis. These points are smoothly connected (the curve is passing through the specified points) and rotated about the y-axis to form the SOR object. The first and last points are only used to determine the slopes of the function at the start and end point. They do not actually lie on the curve. The function used for the SOR object is similar to the splines used for the lathe object. The difference is that the SOR object is less flexible because it underlies the restrictions of any mathematical function, i.e. to any given point y on the rotation axis belongs at most one function value, i.e. one radius value. You can't rotate closed curves with the SOR object. Also, make sure that the curve does not cross zero (y-axis) as this can result in 'less than perfect' bounding cylinders. POV-Ray will very likely fail to render large chunks of the part of the spline contained in such an interval.

The optional keyword `open` allows you to remove the caps on the SOR object. If you do this you shouldn't use it with CSG because the results may be wrong.

The SOR object is useful for creating bottles, vases, and things like that. A simple vase could look like this:

```
#declare Vase = sor {
 7,
 <0.000000, 0.000000>
 <0.118143, 0.000000>
 <0.620253, 0.540084>
 <0.210970, 0.827004>
 <0.194093, 0.962025>
 <0.286920, 1.000000>
 <0.468354, 1.033755>
 open
}
```

One might ask why there is any need for a SOR object if there is already a lathe object which is much more flexible. The reason is quite simple. The intersection test with a SOR object involves solving a cubic polynomial while the test with a lathe object requires to solve a 6th order polynomial (you need a cubic spline for the same smoothness). Since most SOR and lathe objects will have several segments this will make a great difference in speed. The roots of the 3rd order polynomial will also be more accurate and easier to find.

The `sturm` keyword may be added to specify the slower but more accurate Sturmian root solver. It may be used with the surface of revolution object if the shape does not render properly.

The following explanations are for the mathematically interested reader who wants to know how the surface of revolution is calculated. Though it is not necessary to read on it might help in understanding the SOR object.

The function that is rotated about the y-axis to get the final SOR object is given by

$$r^2 = f(h) = A \cdot h^3 + B \cdot h^2 + C \cdot h + D$$

Equation 3.3:

with radius `r` and height `h`. Since this is a cubic function in h it has enough flexibility to allow smooth curves.

The curve itself is defined by a set of n points P(i), i=0...n-1, which are interpolated using one function for every segment of the curve. A segment j, j=1...n-3, goes from point P(j) to point P(j+1) and uses points P(j-1) and P(j+2) to determine the slopes at the endpoints. If there are n points we will have n-3 segments. This means that we need at least four points to get a proper curve. The coefficients A(j), B(j), C(j) and D(j) are calculated for every segment using the equation

where r(j) is the radius and h(j) is the height of point P(j).

The figure below shows the configuration of the points P(i), the location of segment j, and the curve that is defined by this segment.

$$b = M \cdot x, \text{with :}$$

$$b = \begin{bmatrix} r(j)^2 \\ r(j+1)^2 \\ \dfrac{2 \cdot r(j) \cdot (r(j+1) - r(j-1))}{h(j+1) - h(j-1)} \\ \dfrac{2 \cdot r(j+1) \cdot (r(j+2) - r(j))}{h(j+2) - h(j)} \end{bmatrix}$$

$$M = \begin{bmatrix} h(j)^3 & h(j)^2 & h(j) & 1 \\ h(j+1)^3 & h(j+1)^2 & h(j+1) & 1 \\ 3 \cdot h(j)^2 & 2 \cdot h(j) & 1 & 0 \\ 3 \cdot h(j+1)^2 & 2 \cdot h(j+1) & 1 & 0 \end{bmatrix}$$

$$x = \begin{bmatrix} A(j) \\ B(j) \\ C(j) \\ D(j) \end{bmatrix}$$

Equation 3.4:

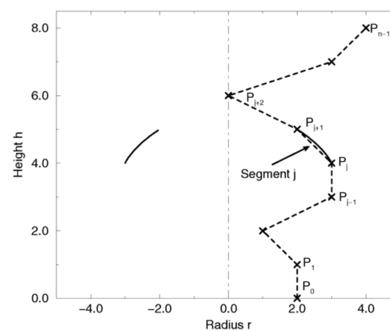

Figure 3.7: Points on a surface of revolution.

3.1.13 Text

A `text` object creates 3-D text as an extruded block letter. Currently only TrueType fonts (ttf) and TrueType Collections (ttc) are supported but the syntax allows for other font types to be added in the future. If TrueType Collections are used, the first font found in the collection will be used. The syntax is:

```
TEXT_OBECT:
  text {
    ttf "fontname.ttf/ttc" "String_of_Text"
Thickness, <Offset>
    [OBJECT_MODIFIERS...]
  }
```

Where `fontname.ttf` or `fontname.ttc` is the name of the TrueType font file. It is a quoted string literal or string expression. The string expression which follows is the actual text of the string object. It too may be a quoted string literal or string expression. See section "Strings" for more on string expressions.

The text will start with the origin at the lower left, front of the first character and will extend in the +x-direction. The baseline of the text follows the x-axis and descender drop into the -y-direction. The front of the character sits in the x-y-plane and the text is extruded in the +z-direction. The front-to-back thickness is specified by the required value *`Thickness`*.

Characters are generally sized so that 1 unit of vertical spacing is correct. The characters are about 0.5 to 0.75 units tall.

The horizontal spacing is handled by POV-Ray internally including any kerning information stored in the font. The required vector *`<Offset>`* defines any extra translation between each character. Normally you should specify a zero for this value. Specifying `0.1*x` would put additional 0.1 units of space between each character. Here is an example:

```
  text {
    ttf "timrom.ttf" "POV-Ray" 1, 0
    pigment { Red }
  }
```

Only printable characters are allowed in text objects. Characters such as return, line feed, tabs, backspace etc. are not supported.

For easy access to your fonts, set the Library_Path to the directory that contains your font collection.

3.1.14 Torus

A `torus` is a 4th order quartic polynomial shape that looks like a donut or inner tube. Because this shape is so useful and quartics are difficult to define, POV-Ray lets you take a short-cut and define a torus by:

```
TORUS:
    torus
    {
```

```
        Major, Minor
        [TORUS_MODIFIER...]
    }
TORUS_MODIFIER:
    sturm | OBJECT_MODIFIER
```

Torus default values:

```
sturm : off
```

where *Major* is a float value giving the major radius and *Minor* is a float specifying the minor radius. The major radius extends from the center of the hole to the mid-line of the rim while the minor radius is the radius of the cross-section of the rim. The torus is centered at the origin and lies in the x-z-plane with the y-axis sticking through the hole.

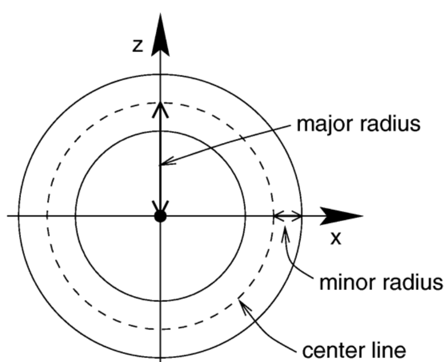

Figure 3.8: Major and minor radius of a torus.

The torus is internally bounded by two cylinders and two rings forming a thick cylinder. With this bounding cylinder the performance of the torus intersection test is vastly increased. The test for a valid torus intersection, i.e. solving a 4th order polynomial, is only performed if the bounding cylinder is hit. Thus a lot of slow root solving calculations are avoided.

Calculations for all higher order polynomials must be very accurate. If the torus renders improperly you may add the keyword `sturm` to use POV-Ray's slower-yet-more-accurate Sturmian root solver.

3.2 Finite Patch Primitives

There are six totally thin, finite objects which have no well-defined inside. They are bicubic patch, disc, smooth triangle, triangle, polygon and mesh / mesh2. They may be combined in CSG union but cannot be used in other types of CSG (or inside a `clipped_by` statement). Because these types are finite POV-Ray can use automatic bounding on them to speed up rendering time. As with all shapes they can be translated, rotated and scaled.

3.2.1 Bicubic Patch

A `bicubic_patch` is a 3D curved surface created from a mesh of triangles. POV-Ray supports a type of bicubic patch called a *Bezier patch*. A bicubic patch is defined as follows:

```
BICUBIC_PATCH:
    bicubic_patch
    {
        PATCH_ITEMS...
        <Point_1>,<Point_2>,<Point_3>,<Point_4>,
        <Point_5>,<Point_6>,<Point_7>,<Point_8>,
        <Point_9>,<Point_10>,<Point_11>,<Point_12>,
        <Point_13>,<Point_14>,<Point_15>,<Point_16>
        [OBJECT_MODIFIERS...]
    }
PATCH_ITEMS:
    type Patch_Type | u_steps Num_U_Steps | v_steps Num_V_Steps |
    flatness Flatness
```

Bicubic patch default values:

```
flatness : 0.0
u_steps  : 0
v_steps  : 0
```

The keyword `type` is followed by a float *`Patch_Type`* which currently must be either 0 or 1. For type 0 only the control points are retained within POV-Ray. This means that a minimal amount of memory is needed but POV-Ray will need to perform many extra calculations when trying to render the patch. Type 1 preprocesses the patch into many subpatches. This results in a significant speedup in rendering at the cost of memory.

The four parameters `type`, `flatness`, `u_steps` and `v_steps` may appear in any order. Only `type` is required. They are followed by 16 vectors (4 rows of 4) that define the x, y, z coordinates of the 16 control points which define the patch. The patch touches the four corner points *`<Point_1>`*, *`<Point_4>`*, *`<Point_13>`* and *`<Point_16>`* while the other 12 points pull and stretch the patch into shape. The Bezier surface is enclosed by the convex hull formed by the 16 control points, this is known as the *convex hull property*.

The keywords `u_steps` and `v_steps` are each followed by integer values which tell how many rows and columns of triangles are the minimum to use to create the surface, both default to 0. The maximum number of individual pieces of the patch that are tested by POV-Ray can be calculated from the following: *pieces = 2^u_steps * 2^v_steps*.

This means that you really should keep `u_steps` and `v_steps` under 4. Most patches look just fine with `u_steps 3` and `v_steps 3`, which translates to 64 subpatches (128 smooth triangles).

As POV-Ray processes the Bezier patch it makes a test of the current piece of the patch to see if it is flat enough to just pretend it is a rectangle. The statement that controls this test is specified with the `flatness` keyword followed by a float. Typical flatness values range from 0 to 1 (the lower the slower). The default if none is specified is 0.0.

If the value for flatness is 0 POV-Ray will always subdivide the patch to the extend

specified by u_steps and v_steps. If flatness is greater than 0 then every time the patch is split, POV-Ray will check to see if there is any need to split further.

There are both advantages and disadvantages to using a non-zero flatness. The advantages include:

- If the patch isn't very curved, then this will be detected and POV-Ray won't waste a lot of time looking at the wrong pieces.

- If the patch is only highly curved in a couple of places, POV-Ray will keep subdividing there and concentrate it's efforts on the hard part.

The biggest disadvantage is that if POV-Ray stops subdividing at a particular level on one part of the patch and at a different level on an adjacent part of the patch there is the potential for cracking. This is typically visible as spots within the patch where you can see through. How bad this appears depends very highly on the angle at which you are viewing the patch.

Like triangles, the bicubic patch is not meant to be generated by hand. These shapes should be created by a special utility. You may be able to acquire utilities to generate these shapes from the same source from which you obtained POV-Ray. Here is an example:

```
bicubic_patch {
  type 0
  flatness 0.01
  u_steps 4
  v_steps 4
  <0, 0, 2>, <1, 0, 0>, <2, 0, 0>, <3, 0,-2>,
  <0, 1  0>, <1, 1, 0>, <2, 1, 0>, <3, 1, 0>,
  <0, 2, 0>, <1, 2, 0>, <2, 2, 0>, <3, 2, 0>,
  <0, 3, 2>, <1, 3, 0>, <2, 3, 0>, <3, 3, -2>
}
```

The triangles in a POV-Ray bicubic_patch are automatically smoothed using normal interpolation but it is up to the user (or the user's utility program) to create control points which smoothly stitch together groups of patches.

3.2.2 Disc

Another flat, finite object available with POV-Ray is the disc. The disc is infinitely thin, it has no thickness. If you want a disc with true thickness you should use a very short cylinder. A disc shape may be defined by:

```
DISC:
    disc
    {
        <Center>, <Normal>, Radius [, Hole_Radius]
        [OBJECT_MODIFIERS...]
    }
```

Disc default values:

```
HOLE RADIUS : 0.0
```

The vector <*Center*> defines the x, y, z coordinates of the center of the disc. The <*Normal*> vector describes its orientation by describing its surface normal vector. This is followed by a float specifying the *Radius*. This may be optionally followed by another float specifying the radius of a hole to be cut from the center of the disc.

Note: The inside of a disc is the inside of the plane that contains the disc. Also note that it is not constrained by the radius of the disc.

3.2.3 Mesh

The mesh object can be used to efficiently store large numbers of triangles. Its syntax is:

```
MESH:
    mesh
    {
        MESH_TRIANGLE...
        [MESH_MODIFIER...]
    }
MESH_TRIANGLE:
    triangle
    {
        <Corner_1>, <Corner_2>, <Corner_3>
        [uv_vectors <uv_Corner_1>, <uv_Corner_2>, <uv_Corner_3>]
        [MESH_TEXTURE]
    } |
    smooth_triangle
    {
        <Corner_1>, <Normal_1>,
        <Corner_2>, <Normal_2>,
        <Corner_3>, <Normal_3>
        [uv_vectors <uv_Corner_1>, <uv_Corner_2>, <uv_Corner_3>]
        [MESH_TEXTURE]
    }
MESH_TEXTURE:
  texture { TEXTURE_IDENTIFIER }
  texture_list {
    TEXTURE_IDENTIFIER TEXTURE_IDENTIFIER TEXTURE_IDENTIFIER
  }

MESH_MODIFIER:
    inside_vector <direction> | hierarchy [ Boolean ] |
    OBJECT_MODIFIER
```

Mesh default values:

```
hierarchy : on
```

Any number of triangle and/or smooth_triangle statements can be used and each of those triangles can be individually textured by assigning a texture identifier to it. The texture has to be declared before the mesh is parsed. It is not possible to use texture definitions inside the triangle or smooth triangle statements. This is a restriction that is necessary for an efficient storage of the assigned textures. See "Triangle and Smooth Triangle" for more information on triangles.

The mesh object can support uv_mapping. For this, per triangle the keyword uv_vectors has to be given, together with three 2D uv-vectors. Each vector specifies a location in the xy-plane from which the texture has to be mapped to the matching points of the triangle. Also see the section uv_mapping.

The mesh's components are internally bounded by a bounding box hierarchy to speed up intersection testing. The bounding hierarchy can be turned off with the hierarchy off keyword. This should only be done if memory is short or the mesh consists of only a few triangles. The default is hierarchy on.

Copies of a mesh object refer to the same triangle data and thus consume very little memory. You can easily trace a hundred copies of a 10000 triangle mesh without running out of memory (assuming the first mesh fits into memory). The mesh object has two advantages over a union of triangles: it needs less memory and it is transformed faster. The memory requirements are reduced by efficiently storing the triangles vertices and normals. The parsing time for transformed meshes is reduced because only the mesh object has to be transformed and not every single triangle as it is necessary for unions.

The mesh object can currently only include triangle and smooth triangle components. That restriction may change, allowing polygonal components, at some point in the future.

Solid Mesh

Triangle mesh objects (mesh and mesh2) can now be used in CSG objects such as difference and intersect, because, after adding inside_vector, they do have a defined 'inside'. This will only work for well-behaved meshes, which are completely closed volumes. If meshes have any holes in them, this might work, but the results are not guaranteed.

To determine if a point is inside a triangle mesh, POV-Ray shoots a ray from the point in some arbitrary direction. If this vector intersects an odd number of triangles, the point is inside the mesh. If it intersects an even number of triangles, the point is outside of the mesh. You can specify the direction of this vector. For example, to use +z as the direction, you would add the following line to the triangle mesh description (following all other mesh data, but before the object modifiers).

```
inside_vector <0, 0, 1>
```

3.2.4 Mesh2

The new mesh syntax is designed for use in conversion from other file formats.

```
MESH2 :
    mesh2{
      VECTORS...
      LISTS...   |
      INDICES... |
      MESH_MODIFIERS
    }
VECTORS :
```

```
    vertex_vectors
    {
      number_of_vertices,
      <vertex1>, <vertex2>, ...
    }|
    normal_vectors
    {
      number_of_normals,
      <normal1>, <normal2>, ...
    }|
    uv_vectors
    {
      number_of_uv_vectors,
      <uv_vect1>, <uv_vect2>, ...
    }
  LISTS :
    texture_list
    {
      number_of_textures,
      texture { Texture1 },
      texture { Texture2 }, ...
    }|
  INDICES :
    face_indices
    {
      number_of_faces,
      <index_a, index_b, index_c> [,texture_index [,
                                   texture_index, texture_index]],
      <index_d, index_e, index_f> [,texture_index [,
                                   texture_index, texture_index]],
      ...
    }|
    normal_indices
    {
      number_of_faces,
      <index_a, index_b, index_c>,
      <index_d, index_e, index_f>,
      ...
    }|
    uv_indices {
      number_of_faces,
      <index_a, index_b, index_c>,
      <index_d, index_e, index_f>,
      ...
    }
  MESH_MODIFIER :
    inside_vector <direction> | object_modifiers
```

`mesh2` has to be specified in the order *VECTORS...*, *LISTS...*, *INDICES...*. The `normal_vectors`, `uv_vectors`, and `texture_list` sections are optional. If the number of normals equals the number of vertices then the normal_indices section is optional and the indexes from the `face_indices` section are used instead. Likewise for the `uv_indices` section.

Note: that the numbers of uv_indices must equal number of faces.

The indexes are ZERO-BASED! So the first item in each list has an index of zero.

Smooth and Flat triangles in the same mesh

You can specify both flat and smooth triangles in the same mesh. To do this, specify the smooth triangles first in the `face_indices` section, followed by the flat triangles. Then, specify normal indices (in the `normal_indices` section) for only the smooth triangles. Any remaining triangles that do not have normal indices associated with them will be assumed to be flat triangles.

Mesh Triangle Textures

To specify a texture for an individual mesh triangle, specify a single integer texture index following the face-index vector for that triangle.

To specify three textures for vertex-texture interpolation, specify three integer texture indices (separated by commas) following the face-index vector for that triangle.

Vertex-texture interpolation and textures for an individual triangle can be mixed in the same mesh

3.2.5 Polygon

The `polygon` object is useful for creating rectangles, squares and other planar shapes with more than three edges. Their syntax is:

```
POLYGON:
    polygon
    {
        Number_Of_Points, <Point_1> <Point_2>... <Point_n>
        [OBJECT_MODIFIER...]
    }
```

The float *`Number_Of_Points`* tells how many points are used to define the polygon. The points <*`Point_1`*> through <*`Point_n`*> describe the polygon or polygons. A polygon can contain any number of sub-polygons, either overlapping or not. In places where an even number of polygons overlaps a hole appears. When you repeat the first point of a sub-polygon, it closes it and starts a new sub-polygon's point sequence. This means that all points of a sub-polygon are different.

If the last sub-polygon is not closed a warning is issued and the program automatically closes the polygon. This is useful because polygons imported from other programs may not be closed, i.e. their first and last point are not the same.

All points of a polygon are three-dimensional vectors that have to lay on the same plane. If this is not the case an error occurs. It is common to use two-dimensional vectors to describe the polygon. POV-Ray assumes that the z value is zero in this case.

A square polygon that matches the default planar image map is simply:

```
polygon {
  4,
  <0, 0>, <0, 1>, <1, 1>, <1, 0>
  texture {
    finish { ambient 1 diffuse 0 }
    pigment { image_map { gif "test.gif"  } }
  }
  //scale and rotate as needed here
}
```

The sub-polygon feature can be used to generate complex shapes like the letter "P", where a hole is cut into another polygon:

```
#declare P = polygon {
  12,
  <0, 0>, <0, 6>, <4, 6>, <4, 3>, <1, 3>, <1,0>, <0, 0>,
  <1, 4>, <1, 5>, <3, 5>, <3, 4>, <1, 4>
}
```

The first sub-polygon (on the first line) describes the outer shape of the letter "P". The second sub-polygon (on the second line) describes the rectangular hole that is cut in the top of the letter "P". Both rectangles are closed, i.e. their first and last points are the same.

The feature of cutting holes into a polygon is based on the polygon inside/outside test used. A point is considered to be inside a polygon if a straight line drawn from this point in an arbitrary direction crosses an odd number of edges (this is known as *Jordan's curve theorem*).

Another very complex example showing one large triangle with three small holes and three separate, small triangles is given below:

```
polygon {
  28,
  <0, 0> <1, 0> <0, 1> <0, 0>          // large outer triangle
  <.3, .7> <.4, .7> <.3, .8> <.3, .7>  // small outer triangle #1
  <.5, .5> <.6, .5> <.5, .6> <.5, .5>  // small outer triangle #2
  <.7, .3> <.8, .3> <.7, .4> <.7, .3>  // small outer triangle #3
  <.5, .2> <.6, .2> <.5, .3> <.5, .2>  // inner triangle #1
  <.2, .5> <.3, .5> <.2, .6> <.2, .5>  // inner triangle #2
  <.1, .1> <.2, .1> <.1, .2> <.1, .1>  // inner triangle #3
}
```

3.2.6 Triangle and Smooth Triangle

The `triangle` primitive is available in order to make more complex objects than the built-in shapes will permit. Triangles are usually not created by hand but are converted from other files or generated by utilities. A triangle is defined by

```
TRIANGLE:
    triangle
    {
        <Corner_1>, <Corner_2>, <Corner_3>
        [OBJECT_MODIFIER...]
    }
```

where <*Corner_n*> is a vector defining the x, y, z coordinates of each corner of the triangle.

Because triangles are perfectly flat surfaces it would require extremely large numbers of very small triangles to approximate a smooth, curved surface. However much of our perception of smooth surfaces is dependent upon the way light and shading is done. By artificially modifying the surface normals we can simulate a smooth surface and hide the sharp-edged seams between individual triangles.

The `smooth_triangle` primitive is used for just such purposes. The smooth triangles use a formula called Phong normal interpolation to calculate the surface normal for any point on the triangle based on normal vectors which you define for the three corners. This makes the triangle appear to be a smooth curved surface. A smooth triangle is defined by

```
SMOOTH_TRIANGLE:
    smooth_triangle
    {
        <Corner_1>, <Normal_1>, <Corner_2>,
        <Normal_2>, <Corner_3>, <Normal_3>
        [OBJECT_MODIFIER...]
    }
```

where the corners are defined as in regular triangles and <*Normal_n*> is a vector describing the direction of the surface normal at each corner.

These normal vectors are prohibitively difficult to compute by hand. Therefore smooth triangles are almost always generated by utility programs. To achieve smooth results, any triangles which share a common vertex should have the same normal vector at that vertex. Generally the smoothed normal should be the average of all the actual normals of the triangles which share that point.

The `mesh` object is a way to combine many `triangle` and `smooth_triangle` objects together in a very efficient way. See "Mesh" for details.

3.3 Infinite Solid Primitives

There are five polynomial primitive shapes that are possibly infinite and do not respond to automatic bounding. They are plane, cubic, poly, quadric and quartic. They do have a well defined inside and may be used in CSG and inside a `clipped_by` statement. As with all shapes they can be translated, rotated and scaled.

3.3.1 Plane

The `plane` primitive is a simple way to define an infinite flat surface. The plane is not a thin boundary or can be compared to a sheet of paper. A plane is a solid object of infinite size that divides POV-space in two parts, inside and outside the plane. The plane is specified as follows:

```
PLANE:
    plane
```

```
{
    <Normal>, Distance
    [OBJECT_MODIFIERS...]
}
```

The <*Normal*> vector defines the surface normal of the plane. A surface normal is a vector which points up from the surface at a 90 degree angle. This is followed by a float value that gives the distance along the normal that the plane is from the origin (that is only true if the normal vector has unit length; see below). For example:

```
plane { <0, 1, 0>, 4 }
```

This is a plane where straight up is defined in the positive y-direction. The plane is 4 units in that direction away from the origin. Because most planes are defined with surface normals in the direction of an axis you will often see planes defined using the `x`, `y` or `z` built-in vector identifiers. The example above could be specified as:

```
plane { y, 4 }
```

The plane extends infinitely in the x- and z-directions. It effectively divides the world into two pieces. By definition the normal vector points to the outside of the plane while any points away from the vector are defined as inside. This inside/outside distinction is important when using planes in CSG and `clipped_by`. It is also important when using fog or atmospheric media. If you place a camera on the "inside" half of the world, then the fog or media will not appear. Such issues arise in any solid object but it is more common with planes. Users typically know when they've accidentally placed a camera inside a sphere or box but "inside a plane" is an unusual concept. In general you can reverse the inside/outside properties of an object by adding the object modifier `inverse`. See "Inverse" and "Empty and Solid Objects" for details.

A plane is called a *polynomial* shape because it is defined by a first order polynomial equation. Given a plane:

```
plane { <A, B, C>, D }
```

it can be represented by the equation *`A*x + B*y + C*z - D*sqrt(A^2 + B^2 + C^2) = 0`*.

Therefore our example `plane{y,4}` is actually the polynomial equation y=4. You can think of this as a set of all x, y, z points where all have y values equal to 4, regardless of the x or z values.

This equation is a first order polynomial because each term contains only single powers of x, y or z. A second order equation has terms like x^2, y^2, z^2, xy, xz and yz. Another name for a 2nd order equation is a quadric equation. Third order polys are called cubics. A 4th order equation is a quartic. Such shapes are described in the sections below.

3.3.2 Poly, Cubic and Quartic

Higher order polynomial surfaces may be defined by the use of a `poly` shape. The syntax is

```
POLY:
    poly
    {
```

```
        Order, <A1, A2, A3,... An>
        [POLY_MODIFIERS...]
    }
POLY_MODIFIERS:
    sturm | OBJECT_MODIFIER
```

Poly default values:

```
sturm : off
```

where `Order` is an integer number from 2 to 15 inclusively that specifies the order of the equation. `A1, A2, ... An` are float values for the coefficients of the equation. There are `n` such terms where `n = ((Order+1)*(Order+2)*(Order+3))/6.` The `cubic` object is an alternate way to specify 3rd order polys. Its syntax is:

```
CUBIC:
    cubic
    {
        <A1, A2, A3,... A20>
        [POLY_MODIFIERS...]
    }
```

Also 4th order equations may be specified with the `quartic` object. Its syntax is:

```
QUARTIC:
    quartic
    {
        <A1, A2, A3,... A35>
        [POLY_MODIFIERS...]
    }
```

The following table shows which polynomial terms correspond to which x,y,z factors for the orders 2 to 7. Remember `cubic` is actually a 3rd order polynomial and `quartic` is 4th order.

Polynomial shapes can be used to describe a large class of shapes including the torus, the lemniscate, etc. For example, to declare a quartic surface requires that each of the coefficients (`A1 ... A35`) be placed in order into a single long vector of 35 terms. As an example let's define a torus the hard way. A Torus can be represented by the equation: $x^4 + y^4 + z^4 + 2 x^2 y^2 + 2 x^2 z^2 + 2 y^2 z^2 - 2 (r_02 + r_12) x^2 + 2 (r_02 - r_12) y^2 - 2 (r_02 + r_12) z^2 + (r_02 - r_12)^2 = 0$

Where r_0 is the major radius of the torus, the distance from the hole of the donut to the middle of the ring of the donut, and r_1 is the minor radius of the torus, the distance from the middle of the ring of the donut to the outer surface. The following object declaration is for a torus having major radius 6.3 minor radius 3.5 (Making the maximum width just under 20).

```
  // Torus having major radius sqrt(40), minor radius sqrt(12)
  quartic {
    < 1,   0,   0,   0,   2,   0,   0,   2,   0,
   -104,   0,   0,   0,   0,   0,   0,   0,   0,
      0,   0,   1,   0,   0,   2,   0,  56,   0,
      0,   0,   0,   1,   0, -104,   0, 784 >
    sturm
  }
```

	2nd	3rd	4th	5th	6th	7th		5th	6th	7th		6th	7th
A1	x^2	x^3	x^4	x^5	x^6	x^7	A41	y^3	xy^3	x^2y^3	A81	z^3	xz^3
A2	xy	x^2y	x^3y	x^4y	x^5y	x^6y	A42	y^2z^3	xy^2z^3	$x^2y^2z^3$	A82	z^2	xz^2
A3	xz	x^2z	x^3z	x^4z	x^5z	x^6z	A43	y^2z^2	xy^2z^2	$x^2y^2z^2$	A83	z	xz
A4	x	x^2	x^3	x^4	x^5	x^6	A44	y^2z	xy^2z	x^2y^2z	A84	1	x
A5	y^2	xy^2	x^2y^2	x^3y^2	x^4y^2	x^5y^2	A45	y^2	xy^2	x^2y^2	A85		y^7
A6	yz	xyz	x^2yz	x^3yz	x^4yz	x^5yz	A46	yz^4	xyz^4	x^2yz^4	A86		y^6z
A7	y	xy	x^2y	x^3y	x^4y	x^5y	A47	yz^3	xyz^3	x^2yz^3	A87		y^6
A8	z^2	xz^2	x^2z^2	x^3z^2	x^4z^2	x^5z^2	A48	yz^2	xyz^2	x^2yz^2	A88		y^5z^2
A9	z	xz	x^2z	x^3z	x^4z	x^5z	A49	yz	xyz	x^2yz	A89		y^5z
A10	1	x	x^2	x^3	x^4	x^5	A50	y	xy	x^2y	A90		y^5
A11		y^3	xy^3	x^2y^3	x^3y^3	x^4y^3	A51	z^5	xz^5	x^2z^5	A91		y^4z^3
A12		y^2z	xy^2z	x^2y^2z	x^3y^2z	x^4y^2z	A52	z^4	xz^4	x^2z^4	A92		y^4z^2
A13		y^2	xy^2	x^2y^2	x^3y^2	x^4y^2	A53	z^3	xz^3	x^2z^3	A93		y^4z
A14		yz^2	xyz^2	x^2yz^2	x^3yz^2	x^4yz^2	A54	z^2	xz^2	x^2z^2	A94		y^4
A15		yz	xyz	x^2yz	x^3yz	x^4yz	A55	z	xz	x^2z	A95		y^3z^4
A16		y	xy	x^2y	x^3y	x^4y	A56	1	x	x^2	A96		y^3z^3
A17		z^3	xz^3	x^2z^3	x^3z^3	x^4z^3	A57		y^6	xy^6	A97		y^3z^2
A18		z^2	xz^2	x^2z^2	x^3z^2	x^4z^2	A58		y^5z	xy^5z	A98		y^3z
A19		z	xz	x^2z	x^3z	x^4z	A59		y^5	xy^5	A99		y^3
A20		1	x	x^2	x^3	x^4	A60		y^4z^2	xy^4z^2	A100		y^2z^5
A21			y^4	xy^4	x^2y^4	x^3y^4	A61		y^4z	xy^4z	A101		y^2z^4
A22			y^3z	xy^3z	x^2y^3z	x^3y^3z	A62		y^4	xy^4	A102		y^2z^3
A23			y^3	xy^3	x^2y^3	x^3y^3	A63		y^3z^3	xy^3z^3	A103		y^2z^2
A24			y^2z^2	xy^2z^2	$x^2y^2z^2$	$x^3y^2z^2$	A64		y^3z^2	xy^3z^2	A104		y^2z
A25			y^2z	xy^2z	x^2y^2z	x^3y^2z	A65		y^3z	xy^3z	A105		y^2
A26			y^2	xy^2	x^2y^2	x^3y^2	A66		y^3	xy^3	A106		yz^6
A27			yz^3	xyz^3	x^2yz^3	x^3yz^3	A67		y^2z^4	xy^2z^4	A107		yz^5
A28			yz^2	xyz^2	x^2yz^2	x^3yz^2	A68		y^2z^3	xy^2z^3	A108		yz^4
A29			yz	xyz	x^2yz	x^3yz	A69		y^2z^2	xy^2z^2	A109		yz^3
A30			y	xy	x^2y	x^3y	A70		y^2z	xy^2z	A110		yz^2
A31			z^4	xz^4	x^2z^4	x^3z^4	A71		y^2	xy^2	A111		yz
A32			z^3	xz^3	x^2z^3	x^3z^3	A72		yz^5	xyz^5	A112		y
A33			z^2	xz^2	x^2z^2	x^3z^2	A73		yz^4	xyz^4	A113		z^7
A34			z	xz	x^2z	x^3z	A74		yz^3	xyz^3	A114		z^6
A35			1	x	x^2	x^3	A75		yz^2	xyz^2	A115		z^5
A36				y^5	xy^5	x^2y^5	A76		yz	xyz	A116		z^4
A37				y^4z	xy^4z	x^2y^4z	A77		y	xy	A117		z^3
A38				y^4	xy^4	x^2y^4	A78		z^6	xz^6	A118		z^2
A39				y^3z^2	xy^3z^2	$x^2y^3z^2$	A79		z^5	xz^5	A119		z
A40				y^3z	xy^3z	x^2y^3z	A80		z^4	xz^4	A120		1

Table 3.4: Cubic and quartic polynomial terms

Poly, cubic and quartics are just like quadrics in that you don't have to understand one to use one. The file `shapesq.inc` has plenty of pre-defined quartics for you to play with.

Polys use highly complex computations and will not always render perfectly. If the surface is not smooth, has dropouts, or extra random pixels, try using the optional keyword `sturm` in the definition. This will cause a slower but more accurate calculation method to be used. Usually, but not always, this will solve the problem. If sturm doesn't work, try rotating or translating the shape by some small amount.

There are really so many different polynomial shapes, we can't even begin to list or describe them all. We suggest you find a good reference or text book if you want to investigate the subject further.

3.3.3 Quadric

The `quadric` object can produce shapes like paraboloids (dish shapes) and hyperboloids (saddle or hourglass shapes). It can also produce ellipsoids, spheres, cones, and cylinders but you should use the `sphere`, `cone`, and `cylinder` objects built into POV-Ray because they are faster than the quadric version.

Note: do not confuse "quaDRic" with "quaRTic". A quadric is a 2nd order polynomial while a quartic is 4th order.

Quadrics render much faster and are less error-prone but produce less complex objects. The syntax is:

```
QUADRIC:
    quadric
    {
        <A,B,C>,<D,E,F>,<G,H,I>,J
        [OBJECT_MODIFIERS...]
    }
```

Although the syntax actually will parse 3 vector expressions followed by a float, we traditionally have written the syntax as above where *A* through *J* are float expressions. These 10 float that define a surface of x, y, z points which satisfy the equation $A x^2 + B y^2 + C z^2 + D xy + E xz + F yz + G x + H y + I z + J = 0$

Different values of *A, B, C, ... J* will give different shapes. If you take any three dimensional point and use its x, y and z coordinates in the above equation the answer will be 0 if the point is on the surface of the object. The answer will be negative if the point is inside the object and positive if the point is outside the object. Here are some examples:

$X^2 + Y^2 + Z^2 - 1 = 0$	Sphere
$X^2 + Y^2 - 1 = 0$	Infinite cylinder along the Z axis
$X^2 + Y^2 - Z^2 = 0$	Infinite cone along the Z axis

Table 3.5: Some quartic shapes

The easiest way to use these shapes is to include the standard file `shapes.inc` into your program. It contains several pre-defined quadrics and you can transform these

pre-defined shapes (using translate, rotate and scale) into the ones you want. For a complete list, see the file `shapes.inc`.

3.4 Isosurface Object

Details about many of the things that can be done with the isosurface object are discussed in the isosurface tutorial section. Below you will only find the syntax basics:

```
isosurface {
  function { FUNCTION_ITEMS }
  [contained_by { SPHERE | BOX }]
  [threshold FLOAT_VALUE]
  [accuracy FLOAT_VALUE]
  [max_gradient FLOAT_VALUE]
  [evaluate P0, P1, P2]
  [open]
  [max_trace INTEGER] | [all_intersections]
  [OBJECT_MODIFIERS...]
}
```

Isosurface default values:

```
contained_by : box{-1,1}
threshold    : 0.0
accuracy     : 0.001
max_gradient : 1.1
```

`function { ... }` This must be specified and be the first item of the `isosurface` statement. Here you place all the mathematical functions that will describe the surface.

`contained_by { ... }` The `contained_by` 'object' limits the area where POV-Ray samples for the surface of the function. This container can either be a sphere or a box, both of which use the standard POV-Ray syntax. If not specified a `box {<-1,-1,-1>, <1,1,1>}` will be used as default.

```
contained_by { sphere { CENTER, RADIUS } }
contained_by { box { CORNER1, CORNER2 } }
```

`threshold` This specifies how much strength, or substance to give the `isosurface`. The surface appears where the `function` value equals the `threshold` value. The default threshold is 0.

```
function = threshold
```

`accuracy` The isosurface finding method is a recursive subdivision method. This subdivision goes on until the length of the interval where POV-Ray finds a surface point is less than the specified `accuracy`. The default value is 0.001.
Smaller values produces more accurate surfaces, but it takes longer to render.

`max_gradient` POV-Ray can find the first intersecting point between a ray and the `isosurface` of any continuous function if the maximum gradient of the function is known. Therefore you can specify a `max_gradient` for the function. The default value is 1.1. When the `max_gradient` used to find the intersecting point is too high, the render slows down considerably. When it is too low, artefacts or holes may appear on the

isosurface. When it is way too low, the surface doesn't show at all. While rendering the isosurface POV-Ray records the found gradient values and prints a warning if these values are higher or much lower than the specified `max_gradient`:

```
Warning: The maximum gradient found was 5.257, but max_gradient of
the isosurface was set to 5.000. The isosurface may contain holes!
Adjust max_gradient to get a proper rendering of the isosurface.
```

```
Warning: The maximum gradient found was 5.257, but max_gradient of
the isosurface was set to 7.000. Adjust max_gradient to
get a faster rendering of the isosurface.
```

For best performance you should specify a value close to the real maximum gradient.

`evaluate` POV-Ray can also dynamically adapt the used max_gradient. To activate this technique you have to specify the `evaluate` keyword followed by three parameters:

- P0: the minimum max_gradient in the estimation process,
- P1: an over-estimating factor. This means that the max_gradient is multiplied by the P1 parameter.
- P2: an attenuation parameter (1 or less)

In this case POV-Ray starts with the `max_gradient` value `P0` and dynamically changes it during the render using `P1` and `P2`. In the evaluation process, the P1 and P2 parameters are used in quadratic functions. This means that over-estimation increases more rapidly with higher values and attenuation more rapidly with lower values. Also with dynamic `max_gradient`, there can be artefacts and holes.

If you are unsure what values to use, start a render without `evaluate` to get a value for `max_gradient`. Now you can use it with `evaluate` like this:

- P0 : found max_gradient * min_factor
  'min_factor' being a float between 0 and 1 to reduce the `max_gradient` to a 'minimum max_gradient'. The ideal value for P0 would be the average of the found max_gradients, but we do not have access to that information.
  A good starting point is 0.6 for the min_factor
- P1 : sqrt(found max_gradient/(found max_gradient * min_factor))
  'min_factor' being the same as used in P0 this will give an over-estimation factor of more than 1, based on your minimum max_gradient and the found max_gradient.
- P2 : 1 or less
  0.7 is a good starting point.

```
#declare Min_factor= 0.6;
isosurface {
   ...
   evaluate 356*Min_factor,  sqrt(356/(356*Min_factor)),  0.7
   //evaluate 213.6, 1.29, 0.7
   ...
 }
```

`open` When the isosurface isn't fully contained within the contained_by object, there will be a cross section. Where this happens, you will see the surface of the container.

With the open keyword, these cross section surfaces are removed. The inside of the isosurface becomes visible.

Note: that open slows down the render speed. Also, it is not recommended to use it with CSG operations.

max_trace Isosurfaces can be used in CSG shapes since they are solid finite objects - if not finite by themselves, they are through the cross section with the container.
By default POV-Ray searches only for the first surface which the ray intersects. But when using an isosurface in CSG operations, the other surfaces must also be found. Therefore, the keyword max_trace must be added to the isosurface statement. It must be followed by an integer value. To check for all surfaces, use the keyword all_intersections instead.
With all_intersections POV-Ray keeps looking until all surfaces are found. With a max_trace it only checks until that number is reached.

3.5 Parametric Object

Where the isosurface object uses implicit surface functions, F(x,y,z)=0, the parametric object is a set of equations for a surface expressed in the form of the parameters that locate points on the surface, x(u,v), y(u,v), z(u,v). Each pair of values for u and v gives a single point <x,y,z> in 3d space

The parametric object is not a solid object it is "hollow", like a thin shell.

Syntax:

```
parametric {
  function { FUNCTION_ITEMS },
  function { FUNCTION_ITEMS },
  function { FUNCTION_ITEMS }
  <u1,v1>, <u2,v2>
  [contained_by { SPHERE | BOX }]
  [max_gradient FLOAT_VALUE]
  [accuracy FLOAT_VALUE]
  [precompute DEPTH, VarList]
}
```

Parametric default values:

```
accuracy     : 0.001
```

The first function calculates the x value of the surface, the second y and the third the z value. Allowed is any function that results in a float.

<u1,v1>,<u2,v2> boundaries of the (u,v) space, in which the surface has to be calculated

contained_by { ... } The contained_by 'object' limits the area where POV-Ray samples for the surface of the function. This container can either be a sphere or a box, both of which use the standard POV-Ray syntax. If not specified a box {<-1,-1,-1>, <1,1,1>} will be used as default.

max_gradient, It's not really the maximum gradient. It's the maximum magnitude of all six partial derivatives over the specified ranges of u and v. That is, if you take dx/du, dx/dv, dy/du, dy/dv, dz/du, and dz/dv and calculate them over the entire range, the max_gradient is the maximum of the absolute values of all of those values.

accuracy The default value is 0.001. Smaller values produces more accurate surfaces, but take longer to render.

precompute can speedup rendering of parametric surfaces. It simply divides parametric surfaces into small ones (2^depth) and precomputes ranges of the variables(x,y,z) which you specify after depth. The maximum depth is 20. High values of depth can produce arrays that use a lot of memory, take longer to parse and render faster. If you declare a parametric surface with the precompute keyword and then use it twice, all arrays are in memory only once.

Example, a unit sphere:

```
parametric {
  function { sin(u)*cos(v) }
  function { sin(u)*sin(v) }
  function { cos(u) }

  <0,0>, <2*pi,pi>
  contained_by { sphere{0, 1.1} }
  max_gradient ??
  accuracy 0.0001
  precompute 10 x,y,z
  pigment {rgb 1}
}
```

3.6 Constructive Solid Geometry

In addition to all of the primitive shapes POV-Ray supports, you can also combine multiple simple shapes into complex shapes using *Constructive Solid Geometry* (CSG). There are four basic types of CSG operations: union, intersection, difference, and merge. CSG objects can be composed of primitives or other CSG objects to create more, and more complex shapes.

3.6.1 Inside and Outside

Most shape primitives, like spheres, boxes and blobs divide the world into two regions. One region is inside the object and one is outside. Given any point in space you can say it's either inside or outside any particular primitive object. Well, it could be exactly on the surface but this case is rather hard to determine due to numerical problems.

Even planes have an inside and an outside. By definition, the surface normal of the plane points towards the outside of the plane. You should note that triangles cannot be used as solid objects in CSG since they have no well defined inside and outside. Triangle-based shapes (mesh, mesh2) can only be used in CSG when they are closed objects and have an inside vector specified.

Note:: Although triangles, bicubic_patches and some other shapes have no well defined inside and outside, they have a front- and backside which makes it possible to use a texture on the front side and an interior_texture on the back side.

CSG uses the concepts of inside and outside to combine shapes together as explained in the following sections.

Imagine you have two objects that partially overlap like shown in the figure below. Four different areas of points can be distinguished: points that are neither in object `A` nor in object `B`, points that are in object `A` but not in object `B`, points that are not in object `A` but in object `B` and last not least points that are in object `A` and object `B`.

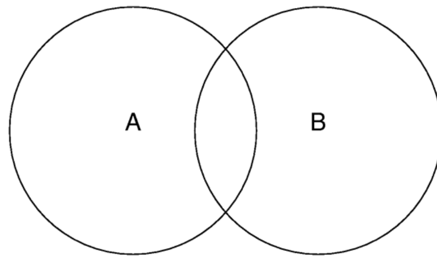

Figure 3.9: Two overlapping objects.

Keeping this in mind it will be quite easy to understand how the CSG operations work.

When using CSG it is often useful to invert an object so that it'll be inside-out. The appearance of the object is not changed, just the way that POV-Ray perceives it. When the `inverse` keyword is used the *inside* of the shape is flipped to become the *outside* and vice versa.

The inside/outside distinction is not important for a `union`, but is important for `intersection`, `difference`, and `merge`.Therefore any objects may be combined using `union` but only solid objects, i.e. objects that have a well-defined interior can be used in the other kinds of CSG. The objects described in "Finite Patch Primitives" have no well defined inside/outside. All objects described in the sections "Finite Solid Primitives" and "Infinite Solid Primitives".

3.6.2 Union

The simplest kind of CSG is the `union`. The syntax is:

```
UNION:
    union
    {
        OBJECTS...
        [OBJECT_MODIFIERS...]
    }
```

Unions are simply glue used to bind two or more shapes into a single entity that can be manipulated as a single object. The image above shows the union of `A` and `B`. The

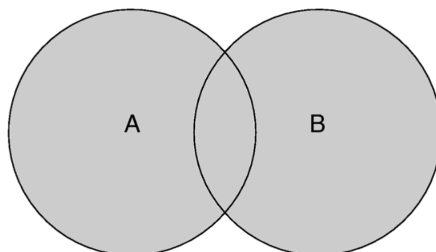

Figure 3.10: The union of two objects.

new object created by the union operation can be scaled, translated and rotated as a single shape. The entire union can share a single texture but each object contained in the union may also have its own texture, which will override any texture statements in the parent object.

You should be aware that the surfaces inside the union will not be removed. As you can see from the figure this may be a problem for transparent unions. If you want those surfaces to be removed you'll have to use the `merge` operations explained in a later section.

The following union will contain a box and a sphere.

```
union {
  box { <-1.5, -1, -1>, <0.5, 1, 1> }
  cylinder { <0.5, 0, -1>, <0.5, 0, 1>, 1 }
}
```

Earlier versions of POV-Ray placed restrictions on unions so you often had to combine objects with `composite` statements. Those earlier restrictions have been lifted so `composite` is no longer needed. It is still supported for backwards compatibility.

Split_Union

`split_union` is a boolean keyword that can be added to a union. It has two states `on`/`off`, its default is `on`.

`split_union` is used when photons are shot at the CSG-object. The object is split up in its compound parts, photons are shot at each part separately. This is to prevent photons from being shot at 'empty spaces' in the object, for example the holes in a grid. With compact objects, without 'empty spaces' `split_union off` can improve photon gathering.

```
union {
  object {...}
  object {...}
  split_union off
}
```

3.6.3 Intersection

The `intersection` object creates a shape containing only those areas where all components overlap. A point is part of an intersection if it is inside both objects, `A` and `B`, as show in the figure below.

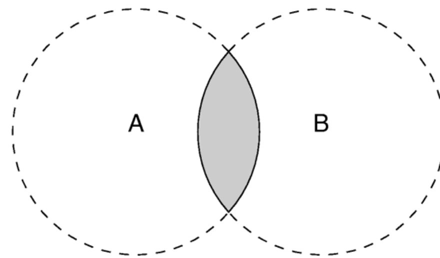

Figure 3.11: The intersection of two objects.

The syntax is:

```
INTERSECTION:
    intersection
    {
        SOLID_OBJECTS...
        [OBJECT_MODIFIERS...]
    }
```

The component objects must have well defined inside/outside properties. Patch objects are not allowed.

Note: if all components do not overlap, the intersection object disappears.

Here is an example that overlaps:

```
  intersection {
    box { <-1.5, -1, -1>, <0.5, 1, 1> }
    cylinder { <0.5, 0, -1>, <0.5, 0, 1>, 1 }
  }
```

3.6.4 Difference

The CSG `difference` operation takes the intersection between the first object and the inverse of all subsequent objects. Thus only points inside object `A` and outside object `B` belong to the difference of both objects.

The result is a subtraction of the 2nd shape from the first shape as shown in the figure below.

The syntax is:

```
DIFFERENCE:
    difference
    {
```

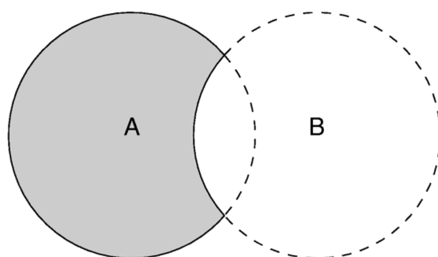

Figure 3.12: The difference between two objects.

```
        SOLID_OBJECTS...
        [OBJECT_MODIFIERS...]
    }
```

The component objects must have well defined inside/outside properties. Patch objects are not allowed.

Note: if the first object is entirely inside the subtracted objects, the difference object disappears.

Here is an example of a properly formed difference:

```
difference {
  box { <-1.5, -1, -1>, <0.5, 1, 1> }
  cylinder { <0.5, 0, -1>, <0.5, 0, 1>, 1 }
}
```

Note: internally, POV-Ray simply adds the `inverse` keyword to the second (and subsequent) objects and then performs an intersection.

The example above is equivalent to:

```
intersection {
  box { <-1.5, -1, -1>, <0.5, 1, 1> }
  cylinder { <0.5, 0, -1>, <0.5, 0, 1>, 1 inverse }
}
```

3.6.5 Merge

The `union` operation just glues objects together, it does not remove the objects' surfaces inside the `union`. Under most circumstances this doesn't matter. However if a transparent `union` is used, those interior surfaces will be visible. The `merge` operations can be used to avoid this problem. It works just like `union` but it eliminates the inner surfaces like shown in the figure below.

The syntax is:

```
MERGE:
    merge
    {
        SOLID_OBJECTS...
```

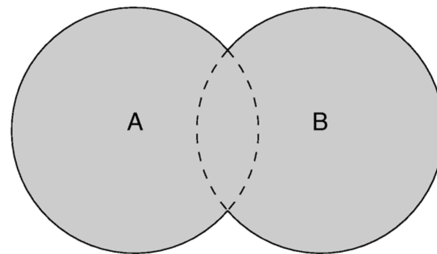

Figure 3.13: Merge removes inner surfaces.

```
        [OBJECT_MODIFIERS...]
    }
```

The component objects must have well defined inside/outside properties. Patch objects are not allowed.

Note: that in general `merge` is slower rendering than `union` when used with non transparent objects. A small test may be needed to determine what is the optimal solution regarding speed and visual result.

3.7 Light Sources

The `light_source` is not really an object. Light sources have no visible shape of their own. They are just points or areas which emit light. They are categorized as objects so that they can be combined with regular objects using `union`. Their full syntax is:

```
LIGHT_SOURCE:
    light_source
    {
        <Location>, COLOR
        [LIGHT_MODIFIERS...]
    }
LIGHT_MODIFIER:
    LIGHT_TYPE | SPOTLIGHT_ITEM | AREA_LIGHT_ITEMS |
    GENERAL_LIGHT_MODIFIERS
LIGHT_TYPE:
    spotlight | shadowless | cylinder | parallel
SPOTLIGHT_ITEM:
    radius Radius | falloff Falloff | tightness Tightness |
    point_at <Spot>
PARALLEL_ITEM:
    point_at <Spot>
AREA_LIGHT_ITEM:
    area_light <Axis_1>, <Axis_2>, Size_1, Size_2 |
    adaptive Adaptive | jitter Jitter | circular | orient
GENERAL_LIGHT_MODIFIERS:
    looks_like { OBJECT } |
    TRANSFORMATION fade_distance Fade_Distance |
```

```
    fade_power Fade_Power | media_attenuation [Bool] |
    media_interaction [Bool] | projected_through
```

Light source default values:

```
LIGHT_TYPE        : pointlight
falloff           : 70
media_interaction : on
media_attenuation : off
point_at          : <0,0,0>
radius            : 70
tightness         : 10
```

The different types of light sources and the optional modifiers are described in the following sections.

The first two items are common to all light sources. The <*Location*> vector gives the location of the light. The *COLOR* gives the color of the light. Only the red, green, and blue components are significant. Any transmit or filter values are ignored.

Note: you vary the intensity of the light as well as the color using this parameter. A color such as `rgb <0.5,0.5,0.5>` gives a white light that is half the normal intensity.

All of the keywords or items in the syntax specification above may appear in any order. Some keywords only have effect if specified with other keywords. The keywords are grouped into functional categories to make it clear which keywords work together. The *GENERAL_LIGHT_MODIFIERS* work with all types of lights and all options.

Note: that *TRANSFORMATIONS* such as `translate`, `rotate` etc. may be applied but no other *OBJECT_MODIFIERS* may be used.

There are three mutually exclusive light types. If no *LIGHT_TYPE* is specified it is a point light. The other choices are `spotlight` and `cylinder`.

3.7.1 Point Lights

The simplest kind of light is a point light. A point light source sends light of the specified color uniformly in all directions. The default light type is a point source. The <*Location*> and *COLOR* is all that is required. For example:

```
  light_source {
    <1000,1000,-1000>, rgb <1,0.75,0> //an orange light
  }
```

3.7.2 Spotlights

Normally light radiates outward equally in all directions from the source. However the `spotlight` keyword can be used to create a cone of light that is bright in the center and falls of to darkness in a soft fringe effect at the edge.

Although the cone of light fades to soft edges, objects illuminated by spotlights still cast hard shadows. The syntax is:

```
SPOTLIGHT_SOURCE:
    light_source
    {
        <Location>, COLOR spotlight
        [LIGHT_MODIFIERS...]
    }
LIGHT_MODIFIER:
    SPOTLIGHT_ITEM | AREA_LIGHT_ITEMS | GENERAL_LIGHT_MODIFIERS
SPOTLIGHT_ITEM:
    radius Radius | falloff Falloff | tightness Tightness |
    point_at <Spot>
```

```
  radius:    30 degrees
  falloff:   45 degrees
  tightness:  0
```

The `point_at` keyword tells the spotlight to point at a particular 3D coordinate. A line from the location of the spotlight to the `point_at` coordinate forms the center line of the cone of light. The following illustration will be helpful in understanding how these values relate to each other.

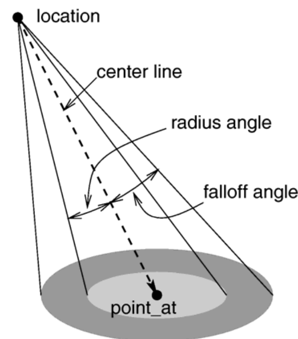

Figure 3.14: The geometry of a spotlight.

The `falloff`, `radius`, and `tightness` keywords control the way that light tapers off at the edges of the cone. These four keywords apply only when the `spotlight` or `cylinder` keywords are used.

The `falloff` keyword specifies the overall size of the cone of light. This is the point where the light falls off to zero intensity. The float value you specify is the angle, in degrees, between the edge of the cone and center line. The `radius` keyword specifies the size of the "hot-spot" at the center of the cone of light. The "hot-spot" is a brighter cone of light inside the spotlight cone and has the same center line. The `radius` value specifies the angle, in degrees, between the edge of this bright, inner cone and the center line. The light inside the inner cone is of uniform intensity. The light between the inner and outer cones tapers off to zero.

For example, assuming a `tightness 0`, with `radius 10` and `falloff 20` the light from the center line out to 10 degrees is full intensity. From 10 to 20 degrees from the center line the light falls off to zero intensity. At 20 degrees or greater there is no light.

Note: if the radius and falloff values are close or equal the light intensity drops rapidly

and the spotlight has a sharp edge.

The values for the `radius`, and `tightness` parameters are half the opening angles of the corresponding cones, both angles have to be smaller than 90 degrees. The light smoothly falls off between the radius and the falloff angle like shown in the figures below (as long as the radius angle is not negative).

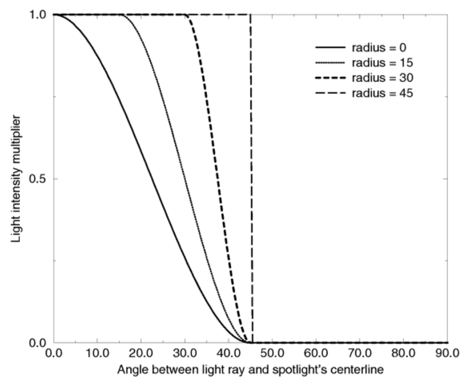

Figure 3.15: Intensity multiplier curve with a fixed falloff angle of 45 degrees.

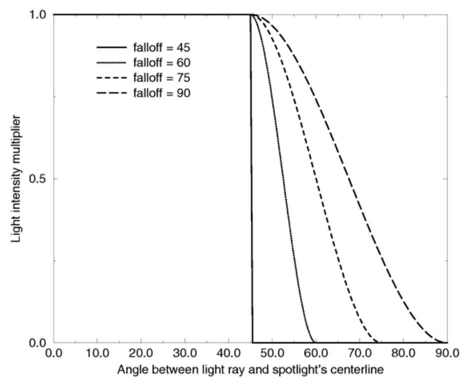

Figure 3.16: Intensity multiplier curve with a fixed radius angle of 45 degrees.

The `tightness` keyword is used to specify an *additional* exponential softening of the edges. A value other than 0, will affect light within the radius cone as well as light in the falloff cone. The intensity of light at an angle from the center line is given by: *`intensity * cos(angle)tightness`*. The default value for tightness is 0. Lower tightness values will make the spotlight brighter, making the spot wider and the edges sharper. Higher values will dim the spotlight, making the spot tighter and the edges softer. Values from 0 to 100 are acceptable.

You should note from the figures that the radius and falloff angles interact with the tightness parameter. To give the tightness value full control over the spotlight's appearance use radius 0 falloff 90. As you can see from the figure below. In that case the falloff angle has no effect and the lit area is only determined by the tightness parameter.

Spotlights may be used anyplace that a normal light source is used. Like any light sources, they are invisible. They may also be used in conjunction with area lights.

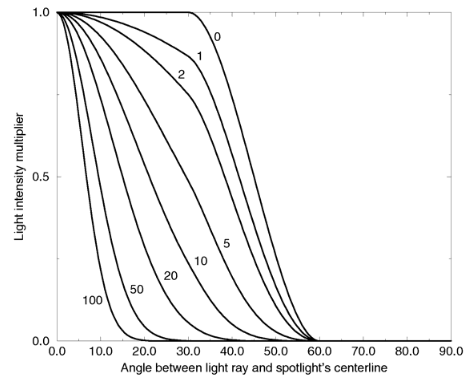

Figure 3.17: Intensity multiplier curve with fixed angle and falloff angles of 30 and 60 degrees respectively and different tightness values.

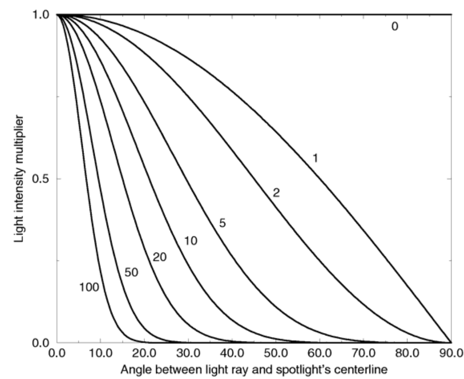

Figure 3.18: Intensity multiplier curve with a negative radius angle and different tightness values.

3.7.3 Cylindrical Lights

The `cylinder` keyword specifies a cylindrical light source that is great for simulating laser beams. Cylindrical light sources work pretty much like spotlights except that the light rays are constrained by a cylinder and not a cone. The syntax is:

```
CYLINDER_LIGHT_SOURCE:
    light_source
    {
        <Location>, COLOR cylinder
        [LIGHT_MODIFIERS...]
    }
LIGHT_MODIFIER:
    SPOTLIGHT_ITEM | AREA_LIGHT_ITEMS | GENERAL_LIGHT_MODIFIERS
SPOTLIGHT_ITEM:
    radius Radius | falloff Falloff | tightness Tightness |
    point_at <Spot>
```

```
  radius:     0.75 degrees
  falloff:    1    degrees
  tightness:  0
```

The `point_at`, `radius`, `falloff` and `tightness` keywords control the same features as with the spotlight. See ”Spotlights” for details.

You should keep in mind that the cylindrical light source is still a point light source. The rays are emitted from one point and are only constraint by a cylinder. The light rays are not parallel.

3.7.4 Parallel Lights

```
syntax:

light_source {
  LOCATION_VECTOR, COLOR
  [LIGHT_SOURCE_ITEMS...]
  parallel
  point_at VECTOR
}
```

The `parallel` keyword can be used with any type of light source.

Note: for normal point lights, `point_at` must come after `parallel`.

Parallel lights are useful for simulating very distant light sources, such as sunlight. As the name suggests, it makes the light rays parallel.

Technically this is done by shooting rays from the closest point on a plane to the object intersection point. The plane is determined by a perpendicular defined by the light `location` and the `point_at` vector.

Two things must be considered when choosing the light location (specifically, its distance):

1. Any parts of an object ”above” the light plane still get illuminated according to the light direction, but they will not cast or receive shadows.
2. `fade_distance` and `fade_power` use the light `location` to determine distance for light attenuation, so the attenuation still looks like that of a point source.
   Area light also uses the light location in its calculations.

3.7.5 Area Lights

Area light sources occupy a finite, one- or two-dimensional area of space. They can cast soft shadows because an object can partially block their light. Point sources are either totally blocked or not blocked.

The `area_light` keyword in POV-Ray creates sources that are rectangular in shape, sort of like a flat panel light. Rather than performing the complex calculations that would be required to model a true area light, it is approximated as an array of point light sources spread out over the area occupied by the light. The array-effect applies to shadows only. The object's illumination is still that of a point source. The intensity of each individual point light in the array is dimmed so that the total amount of light emitted by the light is equal to the light color specified in the declaration. The syntax is:

```
AREA_LIGHT_SOURCE:

  light_source {
    LOCATION_VECTOR, COLOR
    area_light
    AXIS_1_VECTOR, AXIS_2_VECTOR, Size_1, Size_2
    [adaptive Adaptive] [ jitter ]
    [ circular ] [ orient ]
    [ [LIGHT_MODIFIERS...]
  }
```

Any type of light source may be an area light.

The area_light command defines the location, the size and orientation of the area light as well as the number of lights in the light source array. The location vector is the centre of a rectangle defined by the two vectors <*Axis_1*> and <*Axis_2*>. These specify the lengths and directions of the edges of the light.

Since the area lights are rectangular in shape these vectors should be perpendicular to each other. The larger the size of the light the thicker the soft part of shadows will be. The integers Size_1 and Size_2 specify the number of rows and columns of point sources of the. The more lights you use the smoother your shadows will be but the longer they will take to render.

Note: it is possible to specify spotlight parameters along with the area light parameters to create area spotlights. Using area spotlights is a good way to speed up scenes that use area lights since you can confine the lengthy soft shadow calculations to only the parts of your scene that need them.

An interesting effect can be created using a linear light source. Rather than having a rectangular shape, a linear light stretches along a line sort of like a thin fluorescent

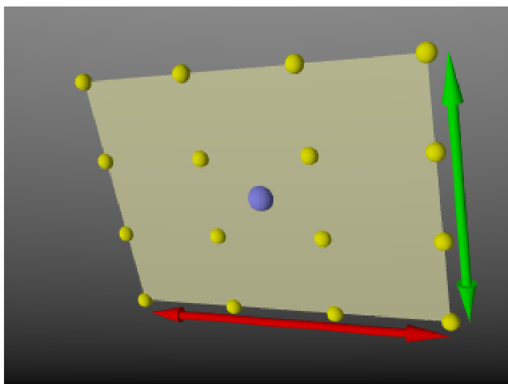

Figure 3.19: 4x4 Area light, location and vectors.

tube. To create a linear light just create an area light with one of the array dimensions set to 1.

The `jitter` command is optional. When used it causes the positions of the point lights in the array to be randomly jittered to eliminate any shadow banding that may occur. The jittering is completely random from render to render and should not be used when generating animations.

The `adaptive` command is used to enable adaptive sampling of the light source. By default POV-Ray calculates the amount of light that reaches a surface from an area light by shooting a test ray at every point light within the array. As you can imagine this is very slow. Adaptive sampling on the other hand attempts to approximate the same calculation by using a minimum number of test rays. The number specified after the keyword controls how much adaptive sampling is used. The higher the number the more accurate your shadows will be but the longer they will take to render. If you're not sure what value to use a good starting point is `adaptive 1`. The `adaptive` keyword only accepts integer values and cannot be set lower than 0.

When performing adaptive sampling POV-Ray starts by shooting a test ray at each of the four corners of the area light. If the amount of light received from all four corners is approximately the same then the area light is assumed to be either fully in view or fully blocked. The light intensity is then calculated as the average intensity of the light received from the four corners. However, if the light intensity from the four corners differs significantly then the area light is partially blocked. The area light is split into four quarters and each section is sampled as described above. This allows POV-Ray to rapidly approximate how much of the area light is in view without having to shoot a test ray at every light in the array. Visually the sampling goes like shown below.

While the adaptive sampling method is fast (relatively speaking) it can sometimes produce inaccurate shadows. The solution is to reduce the amount of adaptive sampling without completely turning it off. The number after the adaptive keyword adjusts the number of times that the area light will be split before the adaptive phase begins. For example if you use `adaptive 0` a minimum of 4 rays will be shot at the light. If you use `adaptive 1` a minimum of 9 rays will be shot (`adaptive 2` gives 25 rays, `adaptive 3` gives 81 rays, etc). Obviously the more shadow rays you shoot the slower the rendering will be so you should use the lowest value that gives acceptable results.

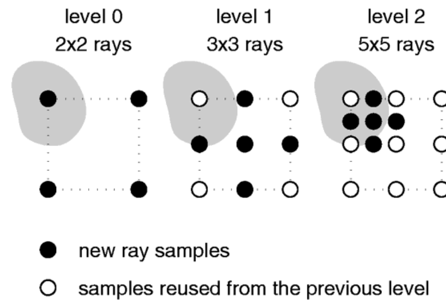

Figure 3.20: Area light adaptive samples.

The number of rays never exceeds the values you specify for rows and columns of points. For example `area_light x,y,4,4` specifies a 4 by 4 array of lights. If you specify `adaptive 3` it would mean that you should start with a 9 by 9 array. In this case no adaptive sampling is done. The 4 by 4 array is used.

The `circular` command has been added to area lights in order to better create circular soft shadows. With ordinary area lights the pseudo-lights are arranged in a rectangular grid and thus project partly rectangular shadows around all objects, including circular objects.
By including the `circular` tag in an area light, the light is stretched and squashed so that it looks like a circle: this way, circular or spherical light sources are better simulated.

A few things to remember:

- Circular area lights can be ellipses: the AXIS_1_VECTOR and AXIS_2_VECTOR define the shape and orientation of the circle; if the vectors are not equal, the light source is elliptical in shape.
- Rectangular artefacts may still show up with very large area grids.
- There is no point in using `circular` with linear area lights or area lights which have a 2x2 size.
- The area of a circular light is roughly 78.5 per cent of a similar size rectangular area light. Increase your axis vectors accordingly if you wish to keep the light source area constant.

The `orient` command has been added to area lights in order to better create soft shadows. Without this modifier, you have to take care when choosing the axis vectors of an area_light, since they define both its area and orientation.
Area lights are two dimensional: shadows facing the area light receive light from a larger surface area than shadows at the sides of the area light.

Actually, the area from which light is emitted at the sides of the area light is reduced to a single line, only casting soft shadows in one direction.

Between these two extremes the surface area emitting light progresses gradually.
By including the `orient` modifier in an area light, the light is rotated so that for every

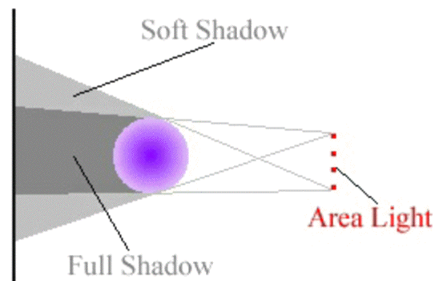

Figure 3.21: Area light facing object

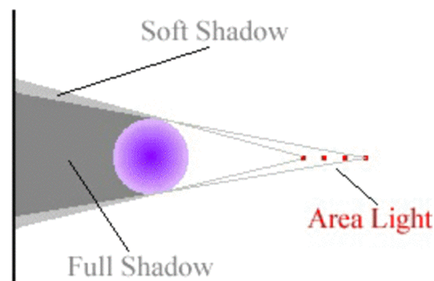

Figure 3.22: Area light not facing object

shadow test, it always faces the point being tested. The initial orientation is no longer important, so you only have to consider the desired dimensions (area) of the light source when specifying the axis vectors.
In effect, this makes the area light source appear 3-dimensional (e.g. an area_light with perpendicular axis vectors of the same size and dimensions using `circular` *and* `orient` simulates a spherical light source).

Orient has a few restrictions:

1. It can be used with "circular" lights only.
2. The two axes of the area light must be of equal length.
3. The two axes of the area light should use an equal number of samples, and that number should be greater than one

These three rules exist because without them, you can get unpredictable results from the orient feature.

If one of the first two rules is broken, POV will issue a warning and correct the problem. If the third rule is broken, you will only get the error message, and POV will not automatically correct the problem.

3.7.6 Shadowless Lights

Using the `shadowless` keyword you can stop a light source from casting shadows. These lights are sometimes called "fill lights". They are another way to simulate ambient light however shadowless lights have a definite source. The syntax is:

```
SHADOWLESS_LIGHT_SOURCE:
    light_source
    {
        <Location>, COLOR shadowless
        [LIGHT_MODIFIERS...]
    }
LIGHT_MODIFIER:
    AREA_LIGHT_ITEMS | GENERAL_LIGHT_MODIFIERS
```

`shadowless` may be used with all types of light sources. The only restriction is that `shadowless` should be before or after *all* spotlight or cylinder option keywords. Don't mix or you get the message "Keyword 'the one following shadowless' cannot be used with standard light source". Also note that shadowless lights will not cause highlights on the illuminated objects.

3.7.7 Looks_like

Normally the light source itself has no visible shape. The light simply radiates from an invisible point or area. You may give a light source any shape by adding a `looks_like` { *OBJECT* } statement.

There is an implied `no_shadow` attached to the `looks_like` object so that light is not blocked by the object. Without the automatic `no_shadow` the light inside the object would not escape. The object would, in effect, cast a shadow over everything.

If you want the attached object to block light then you should attach it with a `union` and not a `looks_like` as follows:

```
union {
  light_source { <100, 200, -300> color White }
  object { My_Lamp_Shape }
}
```

Presumably parts of the lamp shade are transparent to let some light out.

3.7.8 Projected_Through

Syntax:

```
light_source {
  LOCATION_VECTOR, COLOR
  [LIGHT_SOURCE_ITEMS...]
  projected_through { OBJECT }
}
```

Projected_through can be used with any type of light source. Any object can be used, provided it has been declared before.

Projecting a light through an object can be thought of as the opposite of shadowing: only the light rays that hit the projected_through object will contribute to the scene.
This also works with area_lights, producing spots of light with soft edges.
Any objects between the light and the projected through object will not cast shadows for this light. Also any surface within the projected through object will not cast shadows.
Any textures or interiors on the object will be stripped and the object will not show up in the scene.

3.7.9 Light Fading

By default POV-Ray does not diminish light from any light source as it travels through space. In order to get a more realistic effect `fade_distance` and `fade_power` keywords followed by float values can be used to model the distance based falloff in light intensity.

The `fade_distance` is used to specify the distance at which the full light intensity arrives, i. e. the intensity which was given by the *COLOR* specification. The actual attenuation is described by the `fade_power` *Fade_Power*, which determines the falloff rate. For example linear or quadratic falloff can be used by setting `fade_power` to 1 or 2 respectively. The complete formula to calculate the factor by which the light is attenuated is

$$attenuation = \frac{2}{1 + \left(\frac{d}{fade_distance}\right)^{fade_power}}$$

Equation 3.5:

with d being the distance the light has traveled.

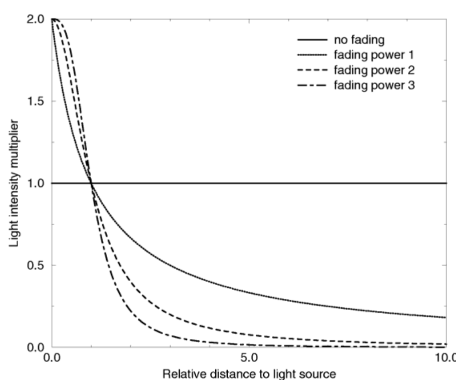

Figure 3.23: Light fading functions for different fading powers.

You should note two important facts: First, for *Fade_Distance* larger than one the light intensity at distances smaller than *Fade_Distance* actually increases. This is necessary to get the light source color if the distance traveled equals the *Fade_Distance*. Second,

only light coming directly from light sources is attenuated. Reflected or refracted light is not attenuated by distance.

3.7.10 Atmospheric Media Interaction

By default light sources will interact with an atmosphere added to the scene. This behavior can be switched off by using `media_interaction off` inside the light source statement.

Note: in POV-Ray 3.0 this feature was turned off and on with the atmosphere keyword.

3.7.11 Atmospheric Attenuation

Normally light coming from light sources is not influenced by fog or atmospheric media. This can be changed by turning the `media_attenuation on` for a given light source on. All light coming from this light source will now be diminished as it travels through the fog or media. This results in an distance-based, exponential intensity falloff ruled by the used fog or media. If there is no fog or media no change will be seen.

Note:in POV-Ray 3.0 this feature was turned off and on with the atmospheric_attenuation keyword.

3.8 Light Groups

Light groups make it possible to create a 'union' of light_sources and objects, where the objects in the group are illuminated by the lights in the group or, if desired, also by the global light_sources. The light_sources in the group can only illuminate the objects that are in the group.

Light_groups are for example useful when creating scenes in which some objects turn out to be too dark but the average light is exactly how it should be, as the light_sources in the group don't contribute to the global lighting.

Syntax :

```
light_group {
  LIGHT_GROUP LIGHT  |
  LIGHT_GROUP OBJECT |
  LIGHT_GROUP
  [LIGHT_GROUP MODIFIER]
}

LIGHT_GROUP LIGHT:
     light_source | light_source IDENTIFIER
LIGHT_GROUP OBJECT:
     OBJECT | OBJECT IDENTIFIER
LIGHT_GROUP MODIFIER:
     global_lights BOOL | TRANSFORMATION
```

global_lights. Add this command to the light_group to have objects in the group also be illuminated by global light sources.

Light groups may be nested. In this case light groups inherit the light sources of the light group they are contained by.

Light groups can be seen as a 'union of an object with light_source' and can be used in CSG.

Examples, simple light_group:

```
#declare RedLight = light_source {
  <-500,500,-500>
  rgb <1,0,0>
}

light_group {
  light_source {RedLight}
  sphere {0,1 pigment {rgb 1}}
  global_lights off
}
```

Nested light_group:

```
#declare L1 = light_group {
  light_source {<10,10,0>, rgb <1,0,0>}
  light_source {<0,0,-100>, rgb <0,0,1>}
  sphere {0,1 pigment {rgb 1}}
}

light_group {
  light_source {<0,100,0>, rgb 0.5}
  light_group {L1}
}
```

Light_groups in CSG:

```
difference {
  light_group {
    sphere {0,1 pigment {rgb 1}}
    light_source {<-100,0,-100> rgb <1,0,0>}
    global_lights off
  }
  light_group {
    sphere {<0,1,0>,1 pigment {rgb 1}}
    light_source {<100,100,0> rgb <0,0,1>}
    global_lights off
  }
  rotate <-45,0,0>
}
```

In the last example the result will be a sphere illuminated red, where the part that is differenced away is illuminated blue. In result comparable to the difference between two spheres with a different pigment.

3.9 Object Modifiers

A variety of modifiers may be attached to objects. The following items may be applied to any object:

```
OBJECT_MODIFIER:
   clipped_by { UNTEXTURED_SOLID_OBJECT... } |
   clipped_by { bounded_by }                 |
   bounded_by { UNTEXTURED_SOLID_OBJECT... } |
   bounded_by { clipped_by }                 |
   no_shadow                |
   no_image [ Bool ]        |
   no_reflection [ Bool ]   |
   inverse                  |
   sturm [ Bool ]           |
   hierarchy [ Bool ]       |
   double_illuminate [ Bool ] |
   hollow  [ Bool ]         |
   interior { INTERIOR_ITEMS... }                        |
   material { [MATERIAL_IDENTIFIER][MATERIAL_ITEMS...] } |
   texture { TEXTURE_BODY }   |
   interior_texture { TEXTURE_BODY } |
   pigment { PIGMENT_BODY }   |
   normal { NORMAL_BODY }     |
   finish { FINISH_ITEMS... } |
   photons { PHOTON_ITEMS...}
   TRANSFORMATION
```

Transformations such as translate, rotate and scale have already been discussed. The modifiers ”Textures” and its parts ”Pigment”, ”Normal”, and ”Finish” as well as ”Interior”, and ”Media” (which is part of interior) are each in major chapters of their own below. In the sub-sections below we cover several other important modifiers: `clipped_by`, `bounded_by`, `material`, `inverse`, `hollow`, `no_shadow`, `no_image`, `no_reflection`, `double_illuminate` and `sturm`. Although the examples below use object statements and object identifiers, these modifiers may be used on any type of object such as sphere, box etc.

3.9.1 Clipped_By

The `clipped_by` statement is technically an object modifier but it provides a type of CSG similar to CSG intersection. The syntax is:

```
CLIPPED_BY:
    clipped_by { UNTEXTURED_SOLID_OBJECT... } |
    clipped_by { bounded_by }
```

Where *UNTEXTURED_SOLID_OBJECT* is one or more solid objects which have had no texture applied. For example:

```
  object {
    My_Thing
    clipped_by{plane{y,0}}
  }
```

Every part of the object My_Thing that is inside the plane is retained while the remaining part is clipped off and discarded. In an intersection object the hole is closed off. With clipped_by it leaves an opening. For example the following figure shows object A being clipped by object B.

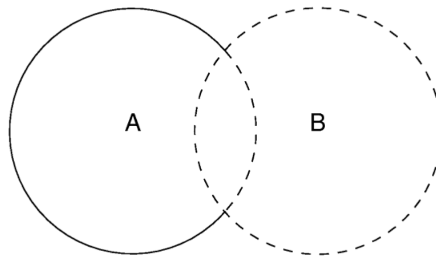

Figure 3.24: An object clipped by another object.

You may use clipped_by to slice off portions of any shape. In many cases it will also result in faster rendering times than other methods of altering a shape. Occasionally you will want to use the clipped_by and bounded_by options with the same object. The following shortcut saves typing and uses less memory.

```
object {
  My_Thing
  bounded_by { box { <0,0,0>, <1,1,1> } }
  clipped_by { bounded_by }
}
```

This tells POV-Ray to use the same box as a clip that was used as a bound.

3.9.2 Bounded_By

The calculations necessary to test if a ray hits an object can be quite time consuming. Each ray has to be tested against every object in the scene. POV-Ray attempts to speed up the process by building a set of invisible boxes, called bounding boxes, which cluster the objects together. This way a ray that travels in one part of the scene doesn't have to be tested against objects in another, far away part of the scene. When a large number of objects are present the boxes are nested inside each other. POV-Ray can use bounding boxes on any finite object and even some clipped or bounded quadrics. However infinite objects (such as a planes, quartic, cubic and poly) cannot be automatically bound. CSG objects are automatically bound if they contain finite (and in some cases even infinite) objects. This works by applying the CSG set operations to the bounding boxes of all objects used inside the CSG object. For difference and intersection operations this will hardly ever lead to an optimal bounding box. It's sometimes better (depending on the complexity of the CSG object) to have you place a bounding shape yourself using a bounded_by statement.

Normally bounding shapes are not necessary but there are cases where they can be used to speed up the rendering of complex objects. Bounding shapes tell the ray-tracer that the object is totally enclosed by a simple shape. When tracing rays, the ray is first tested

against the simple bounding shape. If it strikes the bounding shape the ray is further tested against the more complicated object inside. Otherwise the entire complex shape is skipped, which greatly speeds rendering. The syntax is:

```
BOUNDED_BY:
    bounded_by { UNTEXTURED_SOLID_OBJECT... } |
    bounded_by { clipped_by }
```

Where *UNTEXTURED_SOLID_OBJECT* is one or more solid objects which have had no texture applied. For example:

```
intersection {
  sphere { <0,0,0>, 2 }
  plane  { <0,1,0>, 0 }
  plane  { <1,0,0>, 0 }
  bounded_by { sphere { <0,0,0>, 2 } }
}
```

The best bounding shape is a sphere or a box since these shapes are highly optimized, although, any shape may be used. If the bounding shape is itself a finite shape which responds to bounding slabs then the object which it encloses will also be used in the slab system.

While it may a good idea to manually add a `bounded_by` to intersection, difference and merge, it is best to *never* bound a union. If a union has no `bounded_by` POV-Ray can internally split apart the components of a union and apply automatic bounding slabs to any of its finite parts. Note that some utilities such as `raw2pov` may be able to generate bounds more efficiently than POV-Ray's current system. However most unions you create yourself can be easily bounded by the automatic system. For technical reasons POV-Ray cannot split a merge object. It is maybe best to hand bound a merge, especially if it is very complex.

Note: if bounding shape is too small or positioned incorrectly it may clip the object in undefined ways or the object may not appear at all. To do true clipping, use `clipped_by` as explained in the previous section. Occasionally you will want to use the `clipped_by` and `bounded_by` options with the same object. The following shortcut saves typing and uses less memory.

```
object {
  My_Thing
  clipped_by{ box { <0,0,0>,<1,1,1 > }}
  bounded_by{ clipped_by }
}
```

This tells POV-Ray to use the same box as a bound that was used as a clip.

3.9.3 Material

One of the changes in POV-Ray 3.1 was the removal of several items from `texture {finish{...} }` and to move them to the new `interior` statement. The `halo` statement, formerly part of `texture`, is now renamed `media` and made a part of the `interior`.

This split was deliberate and purposeful (see "Why are Interior and Media Necessary?") however beta testers pointed out that it made it difficult to entirely describe

the surface properties and interior of an object in one statement that can be referenced by a single identifier in a texture library.

The result is that we created a "wrapper" around `texture` and `interior` which we call `material`.

The syntax is:

```
MATERIAL:
    material { [MATERIAL_IDENTIFIER][MATERIAL_ITEMS...] }
MATERIAL_ITEMS:
    TEXTURE | INTERIOR_TEXTURE | INTERIOR | TRANSFORMATIONS
```

For example:

```
#declare MyGlass=material{ texture{ Glass_T } interior{ Glass_I }}
object { MyObject material{ MyGlass}}
```

Internally, the "material" isn't attached to the object. The material is just a container that brings the texture and interior to the object. It is the texture and interior itself that is attached to the object. Users should still consider texture and interior as separate items attached to the object.

The material is just a "bucket" to carry them. If the object already has a texture, then the material texture is layered over it. If the object already has an interior, the material interior fully replaces it and the old interior is destroyed. Transformations inside the material affect only the textures and interiors which are inside the `material{}` wrapper and only those textures or interiors specified are affected. For example:

```
  object {
    MyObject
    material {
      texture { MyTexture }
      scale 4         //affects texture but not object or interior
      interior { MyInterior }
      translate 5*x   //affects texture and interior, not object
    }
  }
```

Note: The `material` statement has nothing to do with the `material_map` statement. A `material_map` is *not* a way to create patterned material. See "Material Maps" for explanation of this unrelated, yet similarly named, older feature.

3.9.4 Inverse

When using CSG it is often useful to invert an object so that it'll be inside-out. The appearance of the object is not changed, just the way that POV-Ray perceives it. When the `inverse` keyword is used the *inside* of the shape is flipped to become the *outside* and vice versa. For example:

```
  object { MyObject inverse }
```

The inside/outside distinction is also important when attaching `interior` to an object especially if `media` is also used. Atmospheric media and fog also do not work as

expected if your camera is inside an object. Using `inverse` is useful to correct that problem.

3.9.5 Hollow

POV-Ray by default assumes that objects are made of a solid material that completely fills the interior of an object. By adding the `hollow` keyword to the object you can make it hollow, also see the "Empty and Solid Objects" chapter. That is very useful if you want atmospheric effects to exist inside an object. It is even required for objects containing an interior media. The keyword may optionally be followed by a float expression which is interpreted as a boolean value. For example `hollow off` may be used to force it off. When the keyword is specified alone, it is the same as `hollow on`. By default `hollow` is `off` when not specified.

In order to get a hollow CSG object you just have to make the top level object hollow. All children will assume the same `hollow` state except when their state is explicitly set. The following example will set both spheres inside the union hollow

```
union {
  sphere { -0.5*x, 1 }
  sphere {  0.5*x, 1 }
  hollow
}
```

while the next example will only set the second sphere hollow because the first sphere was explicitly set to be not hollow.

```
union {
  sphere { -0.5*x, 1 hollow off }
  sphere {  0.5*x, 1 }
  hollow on
}
```

3.9.6 No_Shadow

You may specify the `no_shadow` keyword in an object to make that object cast no shadow. This is useful for special effects and for creating the illusion that a light source actually is visible. This keyword was necessary in earlier versions of POV-Ray which did not have the `looks_like` statement. Now it is useful for creating things like laser beams or other unreal effects. During test rendering it speeds things up if `no_shadow` is applied.

Simply attach the keyword as follows:

```
object {
  My_Thing
  no_shadow
}
```

3.9.7 No_Image, No_Reflection

```
OBJECT {
  [OBJECT_ITEMS...]
  no_image
  no_reflection
}
```

These two keywords are very similar in usage and function to the `no_shadow` keyword, and control an object's visibility.
You can use any combination of the three with your object.

When `no_image` is used, the object will not be seen by the camera, either directly or through transparent/refractive objects. However, it will still cast shadows, and show up in reflections (unless `no_reflection` and/or `no_shadow` is used also).

When `no_reflection` is used, the object will not show up in reflections. It will be seen by the camera (and through transparent/refractive objects) and cast shadows, unless `no_image` and/or `no_shadow` is used.

Using these three keywords you can produce interesting effects like a sphere casting a rectangular shadow, a cube that shows up as a cone in mirrors, etc.

3.9.8 Double_Illuminate

Syntax:

```
OBJECT {
  [OBJECT_ITEMS...]
  double_illuminate
}
```

A surface has two sides; usually, only the side facing the light source is illuminated, the other side remains in shadow. When `double_illuminate` is used, the other side is also illuminated.
This is useful for simulating effects like translucency (as in a lamp shade, sheet of paper, etc).

Note: `double_illuminate` only illuminates both sides of the same surface, so on a sphere, for example, you will not see the effect unless the sphere is either partially transparent, or if the camera is inside and the light source outside of the sphere (or vise versa).

3.9.9 Sturm

Some of POV-Ray's objects allow you to choose between a fast but sometimes inaccurate root solver and a slower but more accurate one. This is the case for all objects that involve the solution of a cubic or quartic polynomial. There are analytic mathematical solutions for those polynomials that can be used.

Lower order polynomials are trivial to solve while higher order polynomials require iterative algorithms to solve them. One of those algorithms is the Sturmian root solver. For example:

```
blob {
  threshold .65
  sphere { <.5,0,0>, .8, 1 }
  sphere { <-.5,0,0>,.8, 1 }
  sturm
}
```

The keyword may optionally be followed by a float expression which is interpreted as a boolean value. For example `sturm off` may be used to force it off. When the keyword is specified alone, it is the same as `sturm on`. By default `sturm` is `off` when not specified.

The following list shows all objects for which the Sturmian root solver can be used.

- blob
- cubic
- lathe (only with quadratic splines)
- poly
- prism (only with cubic splines)
- quartic
- sor

Chapter 4

Textures

The `texture` statement is an object modifier which describes what the surface of an object looks like, i.e. its material. Textures are combinations of pigments, normals, and finishes. Pigment is the color or pattern of colors inherent in the material. Normal is a method of simulating various patterns of bumps, dents, ripples or waves by modifying the surface normal vector. Finish describes the reflective properties of a material.

Note: that in previous versions of POV-Ray, the texture also contained information about the interior of an object. This information has been moved to a separate object modifier called `interior`. See "Interior" for details.

There are three basic kinds of textures: plain, patterned, and layered. A *plain texture* consists of a single pigment, an optional normal, and a single finish. A *patterned texture* combines two or more textures using a block pattern or blending function pattern. Patterned textures may be made quite complex by nesting patterns within patterns. At the innermost levels however, they are made up from plain textures. A *layered texture* consists of two or more semi-transparent textures layered on top of one another.

Note: although we call a plain texture *plain* it may be a very complex texture with patterned pigments and normals. The term *plain* only means that it has a single pigment, normal, and finish.

The syntax for `texture` is as follows:

```
TEXTURE:
    PLAIN_TEXTURE | PATTERNED_TEXTURE | LAYERED_TEXTURE
PLAIN_TEXTURE:
    texture
    {
        [TEXTURE_IDENTIFIER]
        [PNF_IDENTIFIER...]
        [PNF_ITEMS...]
    }
PNF_IDENTIFIER:
    PIGMENT_IDENTIFIER | NORMAL_IDENTIFIER | FINISH_IDENTIFIER
PNF_ITEMS:
    PIGMENT | NORMAL | FINISH | TRANSFORMATION
LAYERED_TEXTURE:
```

```
    NON_PATTERNED_TEXTURE...
PATTERNED_TEXTURE:
    texture
    {
        [PATTERNED_TEXTURE_ID]
        [TRANSFORMATIONS...]
    } |
    texture
    {
        PATTERN_TYPE
        [TEXTURE_PATTERN_MODIFIERS...]
    } |
    texture
    {
        tiles TEXTURE tile2 TEXTURE
        [TRANSFORMATIONS...]
    } |
    texture
    {
        material_map
        {
            BITMAP_TYPE "bitmap.ext"
            [MATERIAL_MODS...] TEXTURE... [TRANSFORMATIONS...]
        }
    }
TEXTURE_PATTERN_MODIFIER:
    PATTERN_MODIFIER | TEXTURE_LIST |
    texture_map { TEXTURE_MAP_BODY }
```

In the *PLAIN_TEXTURE*, each of the items are optional but if they are present the *TEXTURE_IDENTIFIER* must be first. If no texture identifier is given, then POV-Ray creates a copy of the default texture. See "The #default Directive" for details.

Next are optional pigment, normal, and/or finish identifiers which fully override any pigment, normal and finish already specified in the previous texture identifier or default texture. Typically this is used for backward compatibility to allow things like: `texture { MyPigment }` where `MyPigment` is a pigment identifier.

Finally we have optional `pigment`, `normal` or `finish` statements which modify any pigment, normal and finish already specified in the identifier. If no texture identifier is specified the `pigment`, `normal` and `finish` statements modify the current default values. This is the typical plain texture:

```
  texture {
    pigment { MyPigment }
    normal { MyNormal }
    finish { MyFinish }
    scale SoBig
    rotate SoMuch
    translate SoFar
}
```

The *TRANSFORMATIONS* may be interspersed between the pigment, normal and finish statements but are generally specified last. If they are interspersed, then they modify only those parts of the texture already specified. For example:

```
  texture {
    pigment { MyPigment }
    scale SoBig      //affects pigment only
    normal { MyNormal }
    rotate SoMuch    //affects pigment and normal
    finish { MyFinish }
    translate SoFar  //finish is never transformable no matter what.
                     //Therefore affects pigment and normal only
}
```

Texture identifiers may be declared to make scene files more readable and to parameterize scenes so that changing a single declaration changes many values. An identifier is declared as follows.

```
TEXTURE_DECLARATION:
    #declare IDENTIFIER = TEXTURE |
    #local IDENTIFIER = TEXTURE
```

Where *IDENTIFIER* is the name of the identifier up to 40 characters long and *TEXTURE* is any valid `texture` statement. See "#declare vs. #local" for information on identifier scope.

The sections below describe all of the options available in "Pigment", "Normal", and "Finish" which are the main part of plain textures.. There are also separate sections for "Patterned Textures" and "Layered Textures" which are made up of plain textures.

Note: the `tiles` and `material_map` versions of patterned textures are obsolete and are only supported for backwards compatibility.

4.1 Pigment

The color or pattern of colors for an object is defined by a `pigment` statement. All plain textures must have a pigment. If you do not specify one the default pigment is used. The color you define is the way you want the object to look if fully illuminated. You pick the basic color inherent in the object and POV-Ray brightens or darkens it depending on the lighting in the scene. The parameter is called `pigment` because we are defining the basic color the object actually is rather than how it looks.

The syntax for pigment is:

```
PIGMENT:
    pigment {
        [PIGMENT_IDENTIFIER]
        [PIGMENT_TYPE]
        [PIGMENT_MODIFIER...]
    }
PIGMENT_TYPE:
    PATTERN_TYPE | COLOR |
    image_map {
        BITMAP_TYPE "bitmap.ext" [IMAGE_MAP_MODS...]
    }
PIGMENT_MODIFIER:
    PATTERN_MODIFIER | COLOR_LIST | PIGMENT_LIST |
```

```
    color_map { COLOR_MAP_BODY } | colour_map { COLOR_MAP_BODY } |
    pigment_map { PIGMENT_MAP_BODY } | quick_color COLOR |
    quick_colour COLOR
```

Each of the items in a pigment are optional but if they are present, they must be in the order shown. Any items after the *PIGMENT_IDENTIFIER* modify or override settings given in the identifier. If no identifier is specified then the items modify the pigment values in the current default texture. The *PIGMENT_TYPE* fall into roughly four categories. Each category is discussed the sub-sections which follow. The four categories are solid color and image map patterns which are specific to `pigment` statements or color list patterns, color mapped patterns which use POV-Ray's wide selection of general patterns. See "Patterns" for details about specific patterns.

The pattern type is optionally followed by one or more pigment modifiers. In addition to general pattern modifiers such as transformations, turbulence, and warp modifiers, pigments may also have a *COLOR_LIST*, *PIGMENT_LIST*, `color_map`, `pigment_map`, and `quick_color` which are specific to pigments. See "Pattern Modifiers" for information on general modifiers. The pigment-specific modifiers are described in sub-sections which follow. Pigment modifiers of any kind apply only to the pigment and not to other parts of the texture. Modifiers must be specified last.

A pigment statement is part of a `texture` specification. However it can be tedious to use a `texture` statement just to add a color to an object. Therefore you may attach a pigment directly to an object without explicitly specifying that it as part of a texture. For example instead of this:

```
  object { My_Object texture {pigment { color Red } } }
```

you may shorten it to:

```
  object { My_Object pigment {color Red } }
```

Doing so creates an entire `texture` structure with default `normal` and `finish` statements just as if you had explicitly typed the full `texture {...}` around it.

Pigment identifiers may be declared to make scene files more readable and to parameterize scenes so that changing a single declaration changes many values. An identifier is declared as follows.

```
PIGMENT_DECLARATION:
    #declare IDENTIFIER = PIGMENT |
    #local IDENTIFIER = PIGMENT
```

Where *IDENTIFIER* is the name of the identifier up to 40 characters long and *PIGMENT* is any valid `pigment` statement. See "#declare vs. #local" for information on identifier scope.

4.1.1 Solid Color Pigments

The simplest type of pigment is a solid color. To specify a solid color you simply put a color specification inside a `pigment` statement. For example:

```
  pigment { color Orange }
```

A color specification consists of the optional keyword `color` followed by a color identifier or by a specification of the amount of red, green, blue, filtered and unfiltered transparency in the surface. See section "Specifying Colors" for more details about colors. Any pattern modifiers used with a solid color are ignored because there is no pattern to modify.

4.1.2 Color List Pigments

There are four color list patterns: `checker`, `hexagon`, `brick` and `object`. The result is a pattern of solid colors with distinct edges rather than a blending of colors as with color mapped patterns. Each of these patterns is covered in more detail in a later section. The syntax is:

```
COLOR_LIST_PIGMENT:
  pigment {brick [COLOR_1, [COLOR_2]] [PIGMENT_MODIFIERS...] }|
  pigment {checker [COLOR_1, [COLOR_2]] [PIGMENT_MODIFIERS...]}|
  pigment {
     hexagon [COLOR_1, [COLOR_2, [COLOR_3]]] [PIGMENT_MODIFIERS...]
  }|
  pigment {object OBJECT_IDENTIFIER | OBJECT {} [COLOR_1, COLOR_2]}
```

Each *COLOR_n* is any valid color specification. There should be a comma between each color or the `color` keyword should be used as a separator so that POV-Ray can determine where each color specification starts and ends. The `brick` and `checker` pattern expects two colors and `hexagon` expects three. If an insufficient number of colors is specified then default colors are used.

4.1.3 Color Maps

Most of the color patterns do not use abrupt color changes of just two or three colors like those in the brick, checker or hexagon patterns. They instead use smooth transitions of many colors that gradually change from one point to the next. The colors are defined in a pigment modifier called a `color_map` that describes how the pattern blends from one color to the next.

Each of the various pattern types available is in fact a mathematical function that takes any x, y, z location and turns it into a number between 0.0 and 1.0 inclusive. That number is used to specify what mix of colors to use from the color map.

The syntax for `color_map` is as follows:

```
COLOR_MAP:
    color_map { COLOR_MAP_BODY } | colour_map { COLOR_MAP_BODY }
COLOR_MAP_BODY:
    COLOR_MAP_IDENTIFIER | COLOR_MAP_ENTRY...
COLOR_MAP_ENTRY:
    [ Value COLOR ] |
    [ Value_1, Value_2 color COLOR_1 color COLOR_2 ]
```

Where each `Value_n` is a float values between 0.0 and 1.0 inclusive and each *COLOR_n*, is color specifications.

Note: the `[]` brackets are part of the actual *COLOR_MAP_ENTRY*. They are not notational symbols denoting optional parts. The brackets surround each entry in the color map.

There may be from 2 to 256 entries in the map. The alternate spelling `colour_map` may be used.

Here is an example:

```
sphere {
  <0,1,2>, 2
  pigment {
    gradient x       //this is the PATTERN_TYPE
    color_map {
      [0.1  color Red]
      [0.3  color Yellow]
      [0.6  color Blue]
      [0.6  color Green]
      [0.8  color Cyan]
    }
  }
}
```

The pattern function `gradient x` is evaluated and the result is a value from 0.0 to 1.0. If the value is less than the first entry (in this case 0.1) then the first color (red) is used. Values from 0.1 to 0.3 use a blend of red and yellow using linear interpolation of the two colors. Similarly values from 0.3 to 0.6 blend from yellow to blue.

The 3rd and 4th entries both have values of 0.6. This causes an immediate abrupt shift of color from blue to green. Specifically a value that is less than 0.6 will be blue but exactly equal to 0.6 will be green. Moving along, values from 0.6 to 0.8 will be a blend of green and cyan. Finally any value greater than or equal to 0.8 will be cyan.

If you want areas of unchanging color you simply specify the same color for two adjacent entries. For example:

```
color_map {
  [0.1  color Red]
  [0.3  color Yellow]
  [0.6  color Yellow]
  [0.8  color Green]
}
```

In this case any value from 0.3 to 0.6 will be pure yellow.

The first syntax version of *COLOR_MAP_ENTRY* with one float and one color is the current standard. The other double entry version is obsolete and should be avoided. The previous example would look as follows using the old syntax.

```
color_map {
  [0.0 0.1  color Red color Red]
  [0.1 0.3  color Red color Yellow]
  [0.3 0.6  color Yellow color Yellow]
  [0.6.0.8  color Yellow color Green]
  [0.8 1.0  color Green color Green]
}
```

You may use `color_map` with any patterns except `brick`, `checker`, `hexagon`, `object` and `image_map`. You may declare and use `color_map` identifiers. For example:

```
#declare Rainbow_Colors=
  color_map {
    [0.0   color Magenta]
    [0.33  color Yellow]
    [0.67  color Cyan]
    [1.0   color Magenta]
  }
object {
  My_Object
  pigment {
    gradient x
    color_map { Rainbow_Colors }
  }
}
```

4.1.4 Pigment Maps and Pigment Lists

In addition to specifying blended colors with a color map you may create a blend of pigments using a `pigment_map`. The syntax for a pigment map is identical to a color map except you specify a pigment in each map entry (and not a color).

The syntax for `pigment_map` is as follows:

```
PIGMENT_MAP:
    pigment_map { PIGMENT_MAP_BODY }
PIGMENT_MAP_BODY:
    PIGMENT_MAP_IDENTIFIER | PIGMENT_MAP_ENTRY...
PIGMENT_MAP_ENTRY:
    [ Value PIGMENT_BODY ]
```

Where *`Value`* is a float value between 0.0 and 1.0 inclusive and each *PIGMENT_BODY* is anything which can be inside a `pigment{...}` statement. The `pigment` keyword and {} braces need not be specified.

Note: that the `[]` brackets are part of the actual *PIGMENT_MAP_ENTRY*. They are not notational symbols denoting optional parts. The brackets surround each entry in the pigment map.

There may be from 2 to 256 entries in the map.

For example

```
sphere {
  <0,1,2>, 2
  pigment {
    gradient x       //this is the PATTERN_TYPE
    pigment_map {
      [0.3 wood scale 0.2]
      [0.3 Jade]    //this is a pigment identifier
      [0.6 Jade]
      [0.9 marble turbulence 1]
    }
```

```
    }
  }
```

When the `gradient x` function returns values from 0.0 to 0.3 the scaled wood pigment is used. From 0.3 to 0.6 the pigment identifier Jade is used. From 0.6 up to 0.9 a blend of Jade and a turbulent marble is used. From 0.9 on up only the turbulent marble is used.

Pigment maps may be nested to any level of complexity you desire. The pigments in a map may have color maps or pigment maps or any type of pigment you want. Any entry of a pigment map may be a solid color however if all entries are solid colors you should use a `color_map` which will render slightly faster.

Entire pigments may also be used with the block patterns such as `checker`, `hexagon` and `brick`. For example...

```
  pigment {
    checker
    pigment { Jade scale .8 }
    pigment { White_Marble scale .5 }
  }
```

Note: that in the case of block patterns the `pigment` wrapping is required around the pigment information.

A pigment map is also used with the `average` pigment type. See ”Average” for details.

You may not use `pigment_map` or individual pigments with an `image_map`. See section ”Texture Maps” for an alternative way to do this.

You may declare and use pigment map identifiers but the only way to declare a pigment block pattern list is to declare a pigment identifier for the entire pigment.

4.1.5 Image Maps

When all else fails and none of the above pigment pattern types meets your needs you can use an `image_map` to wrap a 2-D bit-mapped image around your 3-D objects.

Specifying an Image Map

The syntax for an `image_map` is:

```
IMAGE_MAP:
    pigment
    {
        image_map
        {
            [BITMAP_TYPE] "bitmap[.ext]"
            [IMAGE_MAP_MODS...]
        }
        [PIGMENT_MODFIERS...]
    }
BITMAP_TYPE:
    gif | tga | iff | ppm | pgm | png | jpeg | tiff | sys
```

```
IMAGE_MAP_MOD:
    map_type Type | once | interpolate Type |
    filter Palette, Amount | filter all Amount |
    transmit Palette, Amount | transmit all Amount
```

After the optional *BITMAP_TYPE* keyword is a string expression containing the name of a bitmapped image file of the specified type. If the *BITMAP_TYPE* is not given, the same type is expected as the type set for output.
Example:

```
plane {
  -z,0
  pigment {
    image_map {png "Eggs.png"}
  }
}

plane {
  -z,0
  pigment {
    image_map {"Eggs"}
  }
}
```

pngimage_map

Several optional modifiers may follow the file specification. The modifiers are described below.

Note: earlier versions of POV-Ray allowed some modifiers before the *BITMAP_TYPE* but that syntax is being phased out in favor of the syntax described here.

Note: sys format is a system-specific format such as BMP for Windows or Pict for Macintosh.

Filenames specified in the image_map statements will be searched for in the home (current) directory first and, if not found, will then be searched for in directories specified by any +L or Library_Path options active. This would facilitate keeping all your image maps files in a separate subdirectory and giving a Library_Path option to specify where your library of image maps are. See ”Library Paths” for details.

By default, the image is mapped onto the x-y-plane. The image is *projected* onto the object as though there were a slide projector somewhere in the -z-direction. The image exactly fills the square area from (x,y) coordinates (0,0) to (1,1) regardless of the image's original size in pixels. If you would like to change this default you may translate, rotate or scale the pigment or texture to map it onto the object's surface as desired.

In the section ”Checker”, the checker pigment pattern is explained. The checks are described as solid cubes of colored clay from which objects are carved. With image maps you should imagine that each pixel is a long, thin, square, colored rod that extends parallel to the z-axis. The image is made from rows and columns of these rods bundled together and the object is then carved from the bundle.

If you would like to change this default orientation you may translate, rotate or scale the pigment or texture to map it onto the object's surface as desired.

The file name is optionally followed by one or more *BITMAP_MODIFIERS*. The `filter`, `filter all`, `transmit`, and `transmit all` modifiers are specific to image maps and are discussed in the following sections. An `image_map` may also use generic bitmap modifiers `map_type`, `once` and `interpolate` described in "Bitmap Modifiers"

The Filter and Transmit Bitmap Modifiers

To make all or part of an image map transparent you can specify `filter` and/or `transmit` values for the color palette/registers of PNG, GIF or IFF pictures (at least for the modes that use palettes). You can do this by adding the keyword `filter` or `transmit` following the filename. The keyword is followed by two numbers. The first number is the palette number value and the second is the amount of transparency. The values should be separated by a comma. For example:

```
image_map {
  gif "mypic.gif"
  filter   0, 0.5 // Make color 0 50\% filtered transparent
  filter   5, 1.0 // Make color 5 100\% filtered transparent
  transmit 8, 0.3 // Make color 8 30\% non-filtered transparent
}
```

You can give the entire image a `filter` or `transmit` value using `filter all` *`Amount`* or `transmit all` *`Amount`*. For example:

```
image_map {
  gif "stnglass.gif"
  filter all 0.9
}
```

Note: early versions of POV-Ray used the keyword `alpha` to specify filtered transparency however that word is often used to describe non-filtered transparency. For this reason `alpha` is no longer used.

See section "Specifying Colors" for details on the differences between filtered and non-filtered transparency.

Using the Alpha Channel

Another way to specify non-filtered transmit transparency in an image map is by using the *alpha channel*. POV-Ray will automatically use the alpha channel for transmittance when one is stored in the image. PNG file format allows you to store a different transparency for each color index in the PNG file, if desired. If your paint programs support this feature of PNG you can do the transparency editing within your paint program rather than specifying transmit values for each color in the POV file. Since PNG and TGA image formats can also store full alpha channel (transparency) information you can generate image maps that have transparency which isn't dependent on the color of a pixel but rather its location in the image.

Although POV uses `transmit 0.0` to specify no transparency and `1.0` to specify full transparency, the alpha data ranges from 0 to 255 in the opposite direction. Alpha data 0 means the same as `transmit 1.0` and alpha data 255 produces `transmit 0.0`.

4.1.6 Quick Color

When developing POV-Ray scenes it's often useful to do low quality test runs that render faster. The +Q command line switch or Quality INI option can be used to turn off some time consuming color pattern and lighting calculations to speed things up. See "Quality Settings" for details. However all settings of +Q5 or Quality=5 or lower turns off pigment calculations and creates gray objects.

By adding a quick_color to a pigment you tell POV-Ray what solid color to use for quick renders instead of a patterned pigment. For example:

```
pigment {
  gradient x
  color_map{
    [0.0 color Yellow]
    [0.3 color Cyan]
    [0.6 color Magenta]
    [1.0 color Cyan]
  }
  turbulence 0.5
  lambda 1.5
  omega 0.75
  octaves 8
  quick_color Neon_Pink
}
```

This tells POV-Ray to use solid Neon_Pink for test runs at quality +Q5 or lower but to use the turbulent gradient pattern for rendering at +Q6 and higher. Solid color pigments such as

```
pigment {color Magenta}
```

automatically set the quick_color to that value. You may override this if you want. Suppose you have 10 spheres on the screen and all are yellow. If you want to identify them individually you could give each a different quick_color like this:

```
sphere {
  <1,2,3>,4
  pigment { color Yellow  quick_color Red }
}
sphere {
  <-1,-2,-3>,4
  pigment { color Yellow  quick_color Blue }
}
```

and so on. At +Q6 or higher they will all be yellow but at +Q5 or lower each would be different colors so you could identify them.

The alternate spelling quick_colour is also supported.

4.2 Normal

Ray-tracing is known for the dramatic way it depicts reflection, refraction and lighting effects. Much of our perception depends on the reflective properties of an object. Ray

tracing can exploit this by playing tricks on our perception to make us see complex details that aren't really there.

Suppose you wanted a very bumpy surface on the object. It would be very difficult to mathematically model lots of bumps. We can however simulate the way bumps look by altering the way light reflects off of the surface. Reflection calculations depend on a vector called a *surface normal* vector. This is a vector which points away from the surface and is perpendicular to it. By artificially modifying (or perturbing) this normal vector you can simulate bumps. This is done by adding an optional `normal` statement.

Note: that attaching a normal pattern does not really modify the surface. It only affects the way light reflects or refracts at the surface so that it looks bumpy.

The syntax is:

```
NORMAL:
  normal { [NORMAL_IDENTIFIER] [NORMAL_TYPE] [NORMAL_MODIFIER...] }
NORMAL_TYPE:
  PATTERN_TYPE Amount |
  bump_map { BITMAP_TYPE "bitmap.ext" [BUMP_MAP_MODS...]}
NORMAL_MODIFIER:
  PATTERN_MODIFIER | NORMAL_LIST | normal_map { NORMAL_MAP_BODY } |
  slope_map{ SLOPE_MAP_BODY } | bump_size Amount |
  no_bump_scale Bool | accuracy Float
```

Each of the items in a normal are optional but if they are present, they must be in the order shown. Any items after the *NORMAL_IDENTIFIER* modify or override settings given in the identifier. If no identifier is specified then the items modify the normal values in the current default texture. The *PATTERN_TYPE* may optionally be followed by a float value that controls the apparent depth of the bumps. Typical values range from 0.0 to 1.0 but any value may be used. Negative values invert the pattern. The default value if none is specified is 0.5.

There are four basic types of *NORMAL_TYPE*s. They are block pattern normals, continuous pattern normals, specialized normals and bump maps. They differ in the types of modifiers you may use with them. The pattern type is optionally followed by one or more normal modifiers. In addition to general pattern modifiers such as transformations, turbulence, and warp modifiers, normals may also have a *NORMAL_LIST*, `slope_map`, `normal_map`, and `bump_size` which are specific to normals. See "Pattern Modifiers" for information on general modifiers. The normal-specific modifiers are described in sub-sections which follow. Normal modifiers of any kind apply only to the normal and not to other parts of the texture. Modifiers must be specified last.

Originally POV-Ray had some patterns which were exclusively used for pigments while others were exclusively used for normals. Since POV-Ray 3.0 you can use any pattern for either pigments or normals. For example it is now valid to use `ripples` as a pigment or `wood` as a normal type. The patterns `bumps`, `dents`, `ripples`, `waves`, `wrinkles`, and `bump_map` were once exclusively normal patterns which could not be used as pigments. Because these six types use specialized normal modification calculations they cannot have `slope_map`, `normal_map` or wave shape modifiers. All other normal pattern types may use them. Because block patterns `checker`, `hexagon`, `object` and `brick` do not return a continuous series of values, they cannot use these modifiers either. See "Patterns" for details about specific patterns.

A normal statement is part of a texture specification. However it can be tedious to use a texture statement just to add bumps to an object. Therefore you may attach a normal directly to an object without explicitly specifying that it as part of a texture. For example instead of this:

```
object  {My_Object texture { normal { bumps 0.5 } } }
```

you may shorten it to:

```
object { My_Object normal { bumps 0.5 } }
```

Doing so creates an entire texture structure with default pigment and finish statements just as if you had explicitly typed the full texture {...} around it. Normal identifiers may be declared to make scene files more readable and to parameterize scenes so that changing a single declaration changes many values. An identifier is declared as follows.

```
NORMAL_DECLARATION:
    #declare IDENTIFIER = NORMAL |
    #local IDENTIFIER = NORMAL
```

Where *IDENTIFIER* is the name of the identifier up to 40 characters long and *NORMAL* is any valid normal statement. See "#declare vs. #local" for information on identifier scope.

4.2.1 Slope Maps

A slope_map is a normal pattern modifier which gives the user a great deal of control over the exact shape of the bumpy features. Each of the various pattern types available is in fact a mathematical function that takes any x, y, z location and turns it into a number between 0.0 and 1.0 inclusive. That number is used to specify where the various high and low spots are. The slope_map lets you further shape the contours. It is best illustrated with a gradient normal pattern. Suppose you have...

```
plane{ z, 0
  pigment{ White }
  normal { gradient x }
}
```

This gives a ramp wave pattern that looks like small linear ramps that climb from the points at x=0 to x=1 and then abruptly drops to 0 again to repeat the ramp from x=1 to x=2. A slope map turns this simple linear ramp into almost any wave shape you want. The syntax is as follows...

```
SLOPE_MAP:
    slope_map { SLOPE_MAP_BODY }
SLOPE_MAP_BODY:
    SLOPE_MAP_IDENTIFIER | SLOPE_MAP_ENTRY...
SLOPE_MAP_ENTRY:
    [ Value, <Height, Slope> ]
```

Note: the [] brackets are part of the actual *SLOPE_MAP_ENTRY*. They are not notational symbols denoting optional parts. The brackets surround each entry in the slope map.

There may be from 2 to 256 entries in the map.

Each `Value` is a float value between 0.0 and 1.0 inclusive and each <*Height*, *Slope*> is a 2 component vector such as <0,1> where the first value represents the apparent height of the wave and the second value represents the slope of the wave at that point. The height should range between 0.0 and 1.0 but any value could be used.

The slope value is the change in height per unit of distance. For example a slope of zero means flat, a slope of 1.0 means slope upwards at a 45 degree angle and a slope of -1 means slope down at 45 degrees. Theoretically a slope straight up would have infinite slope. In practice, slope values should be kept in the range -3.0 to +3.0. Keep in mind that this is only the visually apparent slope. A normal does not actually change the surface.

For example here is how to make the ramp slope up for the first half and back down on the second half creating a triangle wave with a sharp peak in the center.

```
normal {
  gradient x       // this is the PATTERN_TYPE
  slope_map {
    [0   <0, 1>]   // start at bottom and slope up
    [0.5 <1, 1>]   // halfway through reach top still climbing
    [0.5 <1,-1>]   // abruptly slope down
    [1   <0,-1>]   // finish on down slope at bottom
  }
}
```

The pattern function is evaluated and the result is a value from 0.0 to 1.0. The first entry says that at x=0 the apparent height is 0 and the slope is 1. At x=0.5 we are at height 1 and slope is still up at 1. The third entry also specifies that at x=0.5 (actually at some tiny fraction above 0.5) we have height 1 but slope -1 which is downwards. Finally at x=1 we are at height 0 again and still sloping down with slope -1.

Although this example connects the points using straight lines the shape is actually a cubic spline. This example creates a smooth sine wave.

```
normal {
  gradient x          // this is the PATTERN_TYPE
  slope_map {
    [0    <0.5, 1>]   // start in middle and slope up
    [0.25 <1.0, 0>]   // flat slope at top of wave
    [0.5  <0.5,-1>]   // slope down at mid point
    [0.75 <0.0, 0>]   // flat slope at bottom
    [1    <0.5, 1>]   // finish in middle and slope up
  }
}
```

This example starts at height 0.5 sloping up at slope 1. At a fourth of the way through we are at the top of the curve at height 1 with slope 0 which is flat. The space between these two is a gentle curve because the start and end slopes are different. At half way we are at half height sloping down to bottom out at 3/4ths. By the end we are climbing at slope 1 again to complete the cycle. There are more examples in `slopemap.pov` in the sample scenes.

A `slope_map` may be used with any pattern except `brick`, `checker`, `object`, `hexagon`, `bumps`, `dents`, `ripples`, `waves`, `wrinkles` and `bump_map`.

You may declare and use slope map identifiers. For example:

```
#declare Fancy_Wave =
  slope_map {       // Now let's get fancy
    [0.0  <0, 1>]   // Do tiny triangle here
    [0.2  <1, 1>]   //  down
    [0.2  <1,-1>]   //     to
    [0.4  <0,-1>]   //      here.
    [0.4  <0, 0>]   // Flat area
    [0.5  <0, 0>]   //   through here.
    [0.5  <1, 0>]   // Square wave leading edge
    [0.6  <1, 0>]   //   trailing edge
    [0.6  <0, 0>]   // Flat again
    [0.7  <0, 0>]   //   through here.
    [0.7  <0, 3>]   // Start scallop
    [0.8  <1, 0>]   //   flat on top
    [0.9  <0,-3>]   //     finish here.
    [0.9  <0, 0>]   // Flat remaining through 1.0
  }
object{ My_Object
  pigment { White }
  normal {
    wood
    slope_map { Fancy_Wave }
  }
}
```

Normals, Accuracy

Surface normals that use patterns that were not designed for use with normals (anything other than bumps, dents, waves, ripples, and wrinkles) uses a `slope_map` whether you specify one or not. To create a perturbed normal from a pattern, POV-Ray samples the pattern at four points in a pyramid surrounding the desired point to determine the gradient of the pattern at the center of the pyramid. The distance that these points are from the center point determines the accuracy of the approximation. Using points too close together causes floating-point inaccuracies. However, using points too far apart can lead to artefacts as well as smoothing out features that should not be smooth.

Usually, points very close together are desired. POV-Ray currently uses a delta or accuracy distance of 0.02. Sometimes it is necessary to decrease this value to get better accuracy if you are viewing a close-up of the texture. Other times, it is nice to increase this value to smooth out sharp edges in the normal (for example, when using a 'solid' crackle pattern). For this reason, a new property, `accuracy`, has been added to normals. It only makes a difference if the normal uses a `slope_map` (either specified or implied).

You can specify the value of this accuracy (which is the distance between the sample points when determining the gradient of the pattern for slope_map) by adding `accuracy <float>` to your normal. For all patterns, the default is 0.02.

4.2.2 Normal Maps and Normal Lists

Most of the time you will apply single normal pattern to an entire surface but you may also create a pattern or blend of normals using a `normal_map`. The syntax for a `normal_map` is identical to a `pigment_map` except you specify a `normal` in each map entry. The syntax for `normal_map` is as follows:

```
NORMAL_MAP:
    normal_map { NORMAL_MAP_BODY }
NORMAL_MAP_BODY:
    NORMAL_MAP_IDENTIFIER | NORMAL_MAP_ENTRY...
NORMAL_MAP_ENTRY:
    [ Value NORMAL_BODY ]
```

Where *Value* is a float value between 0.0 and 1.0 inclusive and each *NORMAL_BODY* is anything which can be inside a `normal{...}` statement. The `normal` keyword and `{}` braces need not be specified.

Note: that the `[]` brackets are part of the actual *NORMAL_MAP_ENTRY*. They are not notational symbols denoting optional parts. The brackets surround each entry in the normal map.

There may be from 2 to 256 entries in the map.

For example

```
  normal {
    gradient x       //this is the PATTERN_TYPE
    normal_map {
      [0.3  bumps scale 2]
      [0.3  dents]
      [0.6  dents]
      [0.9  marble turbulence 1]
    }
  }
```

When the `gradient x` function returns values from 0.0 to 0.3 then the scaled bumps normal is used. From 0.3 to 0.6 dents pattern is used. From 0.6 up to 0.9 a blend of dents and a turbulent marble is used. From 0.9 on up only the turbulent marble is used.

Normal maps may be nested to any level of complexity you desire. The normals in a map may have slope maps or normal maps or any type of normal you want.

A normal map is also used with the `average` normal type. See ”Average” for details.

Entire normals in a normal list may also be used with the block patterns such as `checker`, `hexagon` and `brick`. For example...

```
  normal {
    checker
    normal { gradient x scale .2 }
    normal { gradient y scale .2 }
  }
```

Note: in the case of block patterns the `normal` wrapping is required around the normal information.

You may not use normal_map or individual normals with a bump_map. See section "Texture Maps" for an alternative way to do this.

You may declare and use normal map identifiers but the only way to declare a normal block pattern list is to declare a normal identifier for the entire normal.

4.2.3 Bump Maps

When all else fails and none of the above normal pattern types meets your needs you can use a bump_map to wrap a 2-D bit-mapped bump pattern around your 3-D objects.

Instead of placing the color of the image on the shape like an image_map a bump_map perturbs the surface normal based on the color of the image at that point. The result looks like the image has been embossed into the surface. By default, a bump map uses the brightness of the actual color of the pixel. Colors are converted to gray scale internally before calculating height. Black is a low spot, white is a high spot. The image's index values may be used instead (see section "Use_Index and Use_Color" below).

Specifying a Bump Map

The syntax for a bump_map is:

```
BUMP_MAP:
    normal
    {
        bump_map
        {
            BITMAP_TYPE "bitmap.ext"
            [BUMP_MAP_MODS...]
        }
        [NORMAL_MODFIERS...]
    }
BITMAP_TYPE:
    gif | tga | iff | ppm | pgm | png | jpeg | tiff | sys
BUMP_MAP_MOD:
    map_type Type | once | interpolate Type | use_color |
    use_colour | bump_size Value
```

After the required *BITMAP_TYPE* keyword is a string expression containing the name of a bitmapped bump file of the specified type. Several optional modifiers may follow the file specification. The modifiers are described below.

Note: earlier versions of POV-Ray allowed some modifiers before the *BITMAP_TYPE* but that syntax is being phased out in favor of the syntax described here.

Note: sys format is a system-specific format such as BMP for Windows or Pict for Macintosh.

Filenames specified in the bump_map statements will be searched for in the home (current) directory first and, if not found, will then be searched for in directories specified by any +L or Library_Path options active. This would facilitate keeping all your

bump maps files in a separate subdirectory and giving a `Library_Path` option to specify where your library of bump maps are. See "Library Paths" for details.

By default, the bump pattern is mapped onto the x-y-plane. The bump pattern is *projected* onto the object as though there were a slide projector somewhere in the -z-direction. The pattern exactly fills the square area from (x,y) coordinates (0,0) to (1,1) regardless of the pattern's original size in pixels. If you would like to change this default you may translate, rotate or scale the pigment or texture to map it onto the object's surface as desired. If you would like to change this default orientation you may translate, rotate or scale the pigment or texture to map it onto the object's surface as desired.

The file name is optionally followed by one or more *BITMAP_MODIFIERS*. The `bump_size`, `use_color` and `use_index` modifiers are specific to bump maps and are discussed in the following sections. See section "Bitmap Modifiers" for the generic bitmap modifiers `map_type`, `once` and `interpolate` described in "Bitmap Modifiers"

Bump_Size

The relative bump size can be scaled using the `bump_size` modifier. The bump size number can be any number other than 0 but typical values are from about 0.1 to as high as 4.0 or 5.0.

```
normal {
  bump_map {
    gif "stuff.gif"
    bump_size 5.0
  }
}
```

Originally `bump_size` could only be used inside a bump map but it can now be used with any normal. Typically it is used to override a previously defined size. For example:

```
normal {
  My_Normal   //this is a previously defined normal identifier
  bump_size 2.0
}
```

Use_Index and Use_Color

Usually the bump map converts the color of the pixel in the map to a gray scale intensity value in the range 0.0 to 1.0 and calculates the bumps based on that value. If you specify `use_index`, the bump map uses the color's palette number to compute as the height of the bump at that point. So, color number 0 would be low and color number 255 would be high (if the image has 256 palette entries). The actual color of the pixels doesn't matter when using the index. This option is only available on palette based formats. The `use_color` keyword may be specified to explicitly note that the color methods should be used instead. The alternate spelling `use_colour` is also valid. These modifiers may only be used inside the `bump_map` statement.

4.2.4 Scaling normals

When scaling a normal, or when scaling an object after a normal is applied to it, the depth of the normal is affected by the scaling. This is not always wanted. If you want to turn off bump scaling for a texture or normal, you can do this by adding the keyword `no_bump_scale` to the texture's or normal's modifiers. This modifier will get passed on to all textures or normals contained in that texture or normal. Think of this like the way no_shadow gets passed on to objects contained in a CSG.

It is also important to note that if you add `no_bump_scale` to a normal or texture that is contained within another pattern (such as within a `texture_map` or `normal_map`), then the only scaling that will be ignored is the scaling of that texture or normal. Scaling of the parent texture or normal or of the object will affect the depth of the bumps, unless `no_bump_scale` is specified at the top-level of the texture (or normal, if the normal is not wrapped in a texture).

4.3 Finish

The finish properties of a surface can greatly affect its appearance. How does light reflect? What happens in shadows? What kind of highlights are visible. To answer these questions you need a `finish`.

The syntax for `finish` is as follows:

```
FINISH:
    finish { [FINISH_IDENTIFIER] [FINISH_ITEMS...] }
FINISH_ITEMS:
    ambient COLOR | diffuse Amount | brilliance Amount |
    phong Amount | phong_size Amount | specular Amount |
    roughness Amount | metallic [Amount] | reflection COLOR |
    crand Amount | conserve_energy BOOL_ON_OF |
    reflection { Color_Reflecting_Min [REFLECTION_ITEMS...] }|
    irid { Irid_Amount [IRID_ITEMS...] }
REFLECTION_ITEMS:
    COLOR_REFLECTION_MAX | fresnel BOOL_ON_OFF |
    falloff FLOAT_FALLOFF | exponent FLOAT_EXPONENT |
    metallic FLOAT_METALLIC
IRID_ITEMS:
    thickness Amount | turbulence Amount
```

The *FINISH_IDENTIFIER* is optional but should proceed all other items. Any items after the *FINISH_IDENTIFIER* modify or override settings given in the *FINISH_IDENTIFIER*. If no identifier is specified then the items modify the finish values in the current default texture.

Note: transformations are not allowed inside a finish because finish items cover the entire surface uniformly. Each of the *FINISH_ITEMS* listed above is described in sub-sections below.

In earlier versions of POV-Ray, the `refraction`, `ior`, and `caustics` keywords were part of the `finish` statement but they are now part of the `interior` statement. They are still supported under `finish` for backward compatibility but the results may not be

100% identical to previous versions. See ”Why are Interior and Media Necessary?” for details.

A `finish` statement is part of a `texture` specification. However it can be tedious to use a `texture` statement just to add a highlights or other lighting properties to an object. Therefore you may attach a finish directly to an object without explicitly specifying that it as part of a texture. For example instead of this:

```
object { My_Object texture { finish { phong 0.5 } } }
```

you may shorten it to:

```
object { My_Object finish { phong 0.5 } }
```

Doing so creates an entire `texture` structure with default `pigment` and `normal` statements just as if you had explicitly typed the full `texture {...}` around it.

Finish identifiers may be declared to make scene files more readable and to parameterize scenes so that changing a single declaration changes many values. An identifier is declared as follows.

```
FINISH_DECLARATION:
    #declare IDENTIFIER = FINISH |
    #local IDENTIFIER = FINISH
```

Where *IDENTIFIER* is the name of the identifier up to 40 characters long and *FINISH* is any valid `finish` statement. See ”#declare vs. #local” for information on identifier scope.

4.3.1 Ambient

The light you see in dark shadowed areas comes from diffuse reflection off of other objects. This light cannot be directly modeled using ray-tracing. However we can use a trick called *ambient lighting* to simulate the light inside a shadowed area.

Ambient light is light that is scattered everywhere in the room. It bounces all over the place and manages to light objects up a bit even where no light is directly shining. Computing real ambient light would take far too much time, so we simulate ambient light by adding a small amount of white light to each texture whether or not a light is actually shining on that texture.

This means that the portions of a shape that are completely in shadow will still have a little bit of their surface color. It’s almost as if the texture glows, though the ambient light in a texture only affects the shape it is used on.

The `ambient` keyword controls the amount of ambient light. Usually a single float value is specified even though the syntax calls for a color. For example a float value of `0.3` gets promoted to the full color vector `<0.3,0.3,0.3,0.3,0.3>` which is acceptable because only the red, green and blue parts are used.

The default value is 0.1 which gives very little ambient light. The value can range from 0.0 to 1.0. Ambient light affects both shadowed and non-shadowed areas so if you turn up the `ambient` value you may want to turn down the `diffuse` and `reflection` values.

Note: that this method doesn’t account for the color of surrounding objects. If you walk into a room that has red walls, floor and ceiling then your white clothing will

look pink from the reflected light. POV-Ray's ambient shortcut doesn't account for this. There is also no way to model specular reflected indirect illumination such as the flashlight shining in a mirror.

You may color the ambient light using one of two methods. You may specify a color rather than a float after the ambient keyword in each finish statement. For example

```
finish { ambient rgb <0.3,0.1,0.1> } //a pink ambient
```

You may also specify the overall ambient light source used when calculating the ambient lighting of an object using the global `ambient_light` setting. The formula is given by *Ambient = Finish_Ambient * Global_Ambient_Light_Source* See section "Ambient Light" for details.

4.3.2 Diffuse Reflection Items

When light reflects off of a surface the laws of physics say that it should leave the surface at the exact same angle it came in. This is similar to the way a billiard ball bounces off a bumper of a pool table. This perfect reflection is called *specular reflection*. However only very smooth polished surfaces reflect light in this way. Most of the time, light reflects and is scattered in all directions by the roughness of the surface. This scattering is called *diffuse reflection* because the light diffuses or spreads in a variety of directions. It accounts for the majority of the reflected light we see.

Diffuse

The keyword `diffuse` is used in a `finish` statement to control how much of the light coming directly from any light sources is reflected via diffuse reflection. For example

```
finish { diffuse 0.7 }
```

means that 70% of the light seen comes from direct illumination from light sources. The default value is `diffuse 0.6`.

Brilliance

The amount of direct light that diffuses from an object depends upon the angle at which it hits the surface. When light hits at a shallow angle it illuminates less. When it is directly above a surface it illuminates more. The `brilliance` keyword can be used in a `finish` statement to vary the way light falls off depending upon the angle of incidence. This controls the tightness of the basic diffuse illumination on objects and slightly adjusts the appearance of surface shininess. Objects may appear more metallic by increasing their brilliance. The default value is 1.0. Higher values from 5.0 to about 10.0 cause the light to fall off less at medium to low angles. There are no limits to the brilliance value. Experiment to see what works best for a particular situation. This is best used in concert with highlighting.

Crand Graininess

Very rough surfaces, such as concrete or sand, exhibit a dark graininess in their apparent color. This is caused by the shadows of the pits or holes in the surface. The `crand` keyword can be added to a `finish` to cause a minor random darkening in the diffuse reflection of direct illumination. Typical values range from `crand 0.01` to `crand 0.5` or higher. The default value is 0. For example:

```
finish { crand 0.05 }
```

This feature is carried over from the earliest versions of POV-Ray and is considered obsolete. This is because the grain or noise introduced by this feature is applied on a pixel-by-pixel basis. This means that it will look the same on far away objects as on close objects. The effect also looks different depending upon the resolution you are using for the rendering.

Note: this should not be used when rendering animations. This is the one of a few truly random features in POV-Ray and will produce an annoying flicker of flying pixels on any textures animated with a `crand` value. For these reasons it is not a very accurate way to model the rough surface effect.

4.3.3 Highlights

Highlights are the bright spots that appear when a light source reflects off of a smooth object. They are a blend of specular reflection and diffuse reflection. They are specular-like because they depend upon viewing angle and illumination angle. However they are diffuse-like because some scattering occurs. In order to exactly model a highlight you would have to calculate specular reflection off of thousands of microscopic bumps called micro facets. The more that micro facets are facing the viewer the shinier the object appears and the tighter the highlights become. POV-Ray uses two different models to simulate highlights without calculating micro facets. They are the *specular* and *Phong* models.

Note: specular and Phong highlights are *not* mutually exclusive. It is possible to specify both and they will both take effect. Normally, however, you will only specify one or the other.

Phong Highlights

The `phong` keyword in the `finish` statement controls the amount of Phong highlighting on the object. It causes bright shiny spots on the object that are the color of the light source being reflected.

The Phong method measures the average of the facets facing in the mirror direction from the light sources to the viewer.

Phong's value is typically from 0.0 to 1.0, where 1.0 causes complete saturation to the light source's color at the brightest area (center) of the highlight. The default `phong 0.0` gives no highlight.

The size of the highlight spot is defined by the phong_size value. The larger the phong size the tighter, or smaller, the highlight and the shinier the appearance. The smaller the phong size the looser, or larger, the highlight and the less glossy the appearance.

Typical values range from 1.0 (very dull) to 250 (highly polished) though any values may be used. Default phong size is 40 (plastic) if phong_size is not specified. For example:

```
finish { phong 0.9 phong_size 60 }
```

If phong is not specified phong_size has no effect.

Specular Highlight

The specular keyword in a finish statement produces a highlight which is very similar to Phong highlighting but it uses slightly different model. The specular model more closely resembles real specular reflection and provides a more credible spreading of the highlights occurring near the object horizons.

The specular value is typically from 0.0 to 1.0, where 1.0 causes complete saturation to the light source's color at the brightest area (center) of the highlight. The default specular 0.0 gives no highlight.

The size of the spot is defined by the value given the roughness keyword. Typical values range from 1.0 (very rough - large highlight) to 0.0005 (very smooth - small highlight). The default value, if roughness is not specified, is 0.05 (plastic).

It is possible to specify wrong values for roughness that will generate an error when you try to render the file. Don't use 0 and if you get errors check to see if you are using a very, very small roughness value that may be causing the error. For example:

```
finish { specular 0.9 roughness 0.02 }
```

If specular is not specified roughness has no effect.

Note: that when light is reflected by a surface such as a mirror, it is called *specular reflection* however such reflection is not controlled by the specular keyword. The reflection keyword controls mirror-like specular reflection.

Metallic Highlight Modifier

The keyword metallic may be used with phong or specular highlights. This keyword indicates that the color of the highlights will be calculated by an empirical function that models the reflectivity of metallic surfaces.

Normally highlights are the color of the light source. Adding this keyword filters the highlight so that white light reflected from a metallic surface takes the color specified by the pigment

The metallic keyword may optionally be follow by a numeric value to specify the influence the amount of the effect. If no keyword is specified, the default value is zero. If the keyword is specified without a value, the default value is one. For example:

```
  finish {
    phong 0.9
    phong_size 60
    metallic
  }
```

If phong or specular keywords are not specified then metallic has no effect.

4.3.4 Specular Reflection

When light does not diffuse and it *does* reflect at the same angle as it hits an object, it is called *specular reflection*. Such mirror-like reflection is controlled by the reflection {...} block in a finish statement.

Syntax:

```
finish {
    reflection {
      [COLOR_REFLECTION_MIN,] COLOR_REFLECTION_MAX
      [fresnel BOOL_ON_OFF]
      [falloff FLOAT_FALLOFF]
      [exponent FLOAT_EXPONENT]
      [metallic FLOAT_METALLIC]
    }
  }
[interior { ior IOR }]
```

The simplest use would be a perfect mirror:

```
  finish { reflection {1.0} ambient 0 diffuse 0 }
```

This gives the object a mirrored finish. It will reflect all other elements in the scene. Usually a single float value is specified after the keyword even though the syntax calls for a color. For example a float value of 0.3 gets promoted to the full color vector <0.3,0.3,0.3,0.3,0.3> which is acceptable because only the red, green and blue parts are used.

The value can range from 0.0 to 1.0. By default there is no reflection.

Note:

- Adding reflection to a texture makes it take longer to render because an additional ray must be traced.
- The reflected light may be tinted by specifying a color rather than a float.
  For example:
  `finish { reflection rgb <1,0,0> }`
  gives a red mirror that only reflects red light.
- Although such reflection is called specular it is not controlled by the specular keyword. That keyword controls a specular highlight.
- The old syntax for simple reflection: "reflection COLOR" and "reflection_exponent Float" without braces is still supported for backward compatibility.

falloff sets a falloff exponent in the variable reflection. This is the exponent telling how fast the reflectivity will fall off, i.e. linear, squared, cubed, etc.

The metallic keyword is similar in function to the "metallic" keyword used for highlights in finishes: it simulates the reflective properties of metallic surfaces, where reflected light takes on the colour of the surface. When metallic is used, the "reflection" color is multiplied by the pigment color at each point. You can specify an optional float value, which is the amount of influence the metallic keyword has on the reflected color. metallic uses the Fresnel equation so that the color of the light is reflected at glancing angles, and the color of the metal is reflected for angles close to the surface's normal.

exponent
POV-Ray uses a limited light model that cannot distinguish between objects which are simply brightly colored and objects which are extremely bright. A white piece of paper, a light bulb, the sun, and a supernova, all would be modeled as rgb<1,1,1> and slightly off-white objects would be only slightly darker. It is especially difficult to model partially reflective surfaces in a realistic way. Middle and lower brightness objects typically look too bright when reflected. If you reduce the reflection value, it tends to darken the bright objects too much. Therefore the optional exponent keyword has been added. It produces non-linear reflection intensities. The default value of 1.0 produces a linear curve. Lower values darken middle and low intensities and keeps high intensity reflections bright. This is a somewhat experimental feature designed for artistic use. It does not directly correspond to any real world reflective properties.

Variable reflection
Many materials, such as water, ceramic glaze, and linoleum are more reflective when viewed at shallow angles. This can be simulated by also specifying a minimum reflection in the reflection {...} statement.
For example:

```
finish { reflection { 0.03, 1 }}
```

uses the same function as the standard reflection, but the first parameter sets the minimum reflectivity. It could be a color vector or a float (which is automatically promoted to a gray vector). This minimum value is how reflective the surface will be when viewed from a direction parallel to its normal.
The second parameter sets the maximum reflectivity, which could also be a color vector or a float (which is automatically promoted to a gray vector). This maximum parameter is how reflective the surface will be when viewed at a 90-degree angle to its normal.

Note: You can make maximum reflection less than minimum reflection if you want, although the result is something that doesn't occur in nature.

When adding the fresnel keyword, the Fresnel reflectivity function is used instead of standard reflection. It calculates reflectivity using the finish's IOR. So with a fresnel reflection_type an interior { ior IOR } statement is required, even with opaque pigments. Remember that in real life many opaque objects have a thin layer of transparent glaze on their surface, and it is the glaze (which -does- have an IOR) that is reflective.

4.3.5 Conserve Energy for Reflection

One of the features in POV-Ray is variable reflection, including realistic Fresnel reflection (see section "Variable Reflection "). Unfortunately, when this is coupled with constant transmittance, the texture can look unrealistic. This unrealism is caused by the scene breaking the law of conservation of energy. As the amount of light reflected changes, the amount of light transmitted should also change (in a give-and-take relationship).

This can be achieved by adding the `conserve_energy` keyword to the object's `finish` {}.
When conserve_energy is enabled, POV-Ray will multiply the amount filtered and transmitted by what is left over from reflection (for example, if reflection is 80%, filter/transmit will be multiplied by 20%).

4.3.6 Iridescence

Iridescence, or Newton's thin film interference, simulates the effect of light on surfaces with a microscopic transparent film overlay. The effect is like an oil slick on a puddle of water or the rainbow hues of a soap bubble. This effect is controlled by the `irid` statement specified inside a `finish` statement.

This parameter modifies the surface color as a function of the angle between the light source and the surface. Since the effect works in conjunction with the position and angle of the light sources to the surface it does not behave in the same ways as a procedural pigment pattern.

The syntax is:

```
IRID:
    irid { Irid_Amount [IRID_ITEMS...] }
IRID_ITEMS:
    thickness Amount | turbulence Amount
```

The required *`Irid_Amount`* parameter is the contribution of the iridescence effect to the overall surface color. As a rule of thumb keep to around 0.25 (25% contribution) or less, but experiment. If the surface is coming out too white, try lowering the `diffuse` and possibly the `ambient` values of the surface.

The `thickness` keyword represents the film's thickness. This is an awkward parameter to set, since the thickness value has no relationship to the object's scale. Changing it affects the scale or *busy-ness* of the effect. A very thin film will have a high frequency of color changes while a thick film will have large areas of color. The default value is zero.

The thickness of the film can be varied with the `turbulence` keyword. You can only specify the amount of turbulence with iridescence. The octaves, lambda, and omega values are internally set and are not adjustable by the user at this time. This parameter varies only a single value: the thickness. Therefore the value must be a single float value. It cannot be a vector as in other uses of the `turbulence` keyword.

In addition, perturbing the object's surface normal through the use of bump patterns will affect iridescence.

For the curious, thin film interference occurs because, when the ray hits the surface of the film, part of the light is reflected from that surface, while a portion is transmitted into the film. This *subsurface* ray travels through the film and eventually reflects off the opaque substrate. The light emerges from the film slightly out of phase with the ray that was reflected from the surface.

This phase shift creates interference, which varies with the wavelength of the component colors, resulting in some wavelengths being reinforced, while others are cancelled out. When these components are recombined, the result is iridescence. See also the global setting "Irid_Wavelength".

The concept used for this feature came from the book *Fundamentals of Three-Dimensional Computer Graphics* by Alan Watt (Addison-Wesley).

4.4 Halo

Earlier versions of POV-Ray used a feature called `halo` to simulate fine particles such as smoke, steam, fog, or flames. The `halo` statement was part of the `texture` statement. This feature has been discontinued and replaced by the `interior` and `media` statements which are object modifiers outside the `texture` statement.

See "Why are Interior and Media Necessary?" for a detailed explanation on the reasons for the change. See "Media" for details on `media`.

4.5 Patterned Textures

Patterned textures are complex textures made up of multiple textures. The component textures may be plain textures or may be made up of patterned textures. A plain texture has just one pigment, normal and finish statement. Even a pigment with a pigment map is still one pigment and thus considered a plain texture as are normals with normal map statements.

Patterned textures use either a `texture_map` statement to specify a blend or pattern of textures or they use block textures such as `checker` with a texture list or a bitmap similar to an image map called a *material map* specified with a `material_map` statement.

The syntax is...

```
PATTERNED_TEXTURE:
    texture
    {
        [PATTERNED_TEXTURE_ID]
        [TRANSFORMATIONS...]
    } |
    texture
    {
        PATTERN_TYPE
        [TEXTURE_PATTERN_MODIFIERS...]
    } |
    texture
```

```
    {
        tiles TEXTURE tile2 TEXTURE
        [TRANSFORMATIONS...]
    } |
    texture
    {
        material_map
        {
            BITMAP_TYPE "bitmap.ext"
            [BITMAP_MODS...] TEXTURE... [TRANSFORMATIONS...]
        }
    }
TEXTURE_PATTERN_MODIFIER:
    PATTERN_MODIFIER | TEXTURE_LIST |
    texture_map { TEXTURE_MAP_BODY }
```

There are restrictions on using patterned textures. A patterned texture may not be used as a default texture (see section ”The #default Directive”). A patterned texture cannot be used as a layer in a layered texture however you may use layered textures as any of the textures contained within a patterned texture.

4.5.1 Texture Maps

In addition to specifying blended color with a color map or a pigment map you may create a blend of textures using `texture_map`. The syntax for a texture map is identical to the pigment map except you specify a texture in each map entry.

The syntax for `texture_map` is as follows:

```
TEXTURE_MAP:
    texture_map { TEXTURE_MAP_BODY }
TEXTURE_MAP_BODY:
  TEXTURE_MAP_IDENTIFIER | TEXTURE_MAP_ENTRY...
TEXTURE_MAP_ENTRY:
    [ Value TEXTURE_BODY ]
```

Where *`Value`* is a float value between 0.0 and 1.0 inclusive and each *TEXTURE_BODY* is anything which can be inside a `texture{...}` statement. The `texture` keyword and `{}` braces need not be specified.

Note: the `[]` brackets are part of the actual *TEXTURE_MAP_ENTRY*. They are not notational symbols denoting optional parts. The brackets surround each entry in the texture map.

There may be from 2 to 256 entries in the map.

For example:

```
  texture {
    gradient x       //this is the PATTERN_TYPE
    texture_map {
      [0.3  pigment{Red} finish{phong 1}]
      [0.3  T_Wood11]    //this is a texture identifier
      [0.6  T_Wood11]
      [0.9  pigment{DMFWood4} finish{Shiny}]
```

```
    }
  }
```

When the `gradient x` function returns values from 0.0 to 0.3 the red highlighted texture is used. From 0.3 to 0.6 the texture identifier `T_Wood11` is used. From 0.6 up to 0.9 a blend of `T_Wood11` and a shiny `DMFWood4` is used. From 0.9 on up only the shiny wood is used.

Texture maps may be nested to any level of complexity you desire. The textures in a map may have color maps or texture maps or any type of texture you want.

The blended area of a texture map works by fully calculating both contributing textures in their entirety and then linearly interpolating the apparent colors. This means that reflection, refraction and lighting calculations are done twice for every point. This is in contrast to using a pigment map and a normal map in a plain texture, where the pigment is computed, then the normal, then reflection, refraction and lighting are calculated once for that point.

Entire textures may also be used with the block patterns such as `checker`, `hexagon` and `brick`. For example...

```
  texture {
    checker
      texture { T_Wood12 scale .8 }
      texture {
        pigment { White_Marble }
        finish { Shiny }
        scale .5
      }
    }
  }
```

Note: that in the case of block patterns the `texture` wrapping is required around the texture information. Also note that this syntax prohibits the use of a layered texture however you can work around this by declaring a texture identifier for the layered texture and referencing the identifier.

A texture map is also used with the `average` texture type. See ”Average” for details.

You may declare and use texture map identifiers but the only way to declare a texture block pattern list is to declare a texture identifier for the entire texture.

4.5.2 Tiles

Earlier versions of POV-Ray had a patterned texture called a *tiles texture*. It used the `tiles` and `tile2` keywords to create a checkered pattern of textures.

```
TILES_TEXTURE:
    texture
    {
        tiles TEXTURE tile2 TEXTURE
        [TRANSFORMATIONS...]
    }
```

Although it is still supported for backwards compatibility you should use a checker block texture pattern described in section "Texture Maps" rather than tiles textures.

4.5.3 Material Maps

The material_map patterned texture extends the concept of image maps to apply to entire textures rather than solid colors. A material map allows you to wrap a 2-D bit-mapped texture pattern around your 3-D objects.

Instead of placing a solid color of the image on the shape like an image map, an entire texture is specified based on the index or color of the image at that point. You must specify a list of textures to be used like a *texture palette* rather than the usual color palette.

When used with mapped file types such as GIF, and some PNG and TGA images, the index of the pixel is used as an index into the list of textures you supply. For unmapped file types such as some PNG and TGA images the 8 bit value of the red component in the range 0-255 is used as an index.

If the index of a pixel is greater than the number of textures in your list then the index is taken modulo N where N is the length of your list of textures.

Note: The material_map statement has nothing to do with the material statement. A material_map is *not* a way to create patterned material. See "Material" for explanation of this unrelated, yet similarly named, older feature.

Specifying a Material Map

The syntax for a material_map is:

```
MATERIAL_MAP:
    texture
    {
        material_map
        {
            BITMAP_TYPE "bitmap.ext"
            [BITMAP_MODS...] TEXTURE... [TRANSFORMATIONS...]
        }
    }
BITMAP_TYPE:
    gif | tga | iff | ppm | pgm | png | jpeg | tiff | sys
BITMAP_MOD:
    map_type Type | once | interpolate Type
```

After the required *BITMAP_TYPE* keyword is a string expression containing the name of a bitmapped material file of the specified type. Several optional modifiers may follow the file specification. The modifiers are described below.

Note: earlier versions of POV-Ray allowed some modifiers before the *BITMAP_TYPE* but that syntax is being phased out in favor of the syntax described here.

Note: sys format is a system-specific format such as BMP for Windows or Pict for Macintosh.

Filenames specified in the `material_map` statements will be searched for in the home (current) directory first and, if not found, will then be searched for in directories specified by any `+L` or `Library_Path` options active. This would facilitate keeping all your material maps files in a separate subdirectory and giving a `Library_Path` option to specify where your library of material maps are. See "Library Paths" for details.

By default, the material is mapped onto the x-y-plane. The material is *projected* onto the object as though there were a slide projector somewhere in the -z-direction. The material exactly fills the square area from (x,y) coordinates (0,0) to (1,1) regardless of the material's original size in pixels. If you would like to change this default you may translate, rotate or scale the texture to map it onto the object's surface as desired.

The file name is optionally followed by one or more *BITMAP_MODIFIERS*. There are no modifiers which are unique to a `material_map`. It only uses the generic bitmap modifiers `map_type`, `once` and `interpolate` described in "Bitmap Modifiers".

Although `interpolate` is legal in material maps, the color index is interpolated before the texture is chosen. It does not interpolate the final color as you might hope it would. In general, interpolation of material maps serves no useful purpose but this may be fixed in future versions.

Next is one or more `texture` statements. Each texture in the list corresponds to an index in the bitmap file. For example:

```
texture {
   material_map {
      png "povmap.png"
      texture {  //used with index 0
         pigment {color red 0.3 green 0.1 blue 1}
         normal  {ripples 0.85 frequency 10 }
         finish  {specular 0.75}
         scale 5
      }
      texture {  //used with index 1
         pigment {White}
         finish {
           ambient 0 diffuse 0
           reflection 0.9 specular 0.75
         }
      }
      // used with index 2
      texture {pigment{NeonPink} finish{Luminous}}
      texture {  //used with index 3
         pigment {
            gradient y
            color_map {
               [0.00 rgb < 1 , 0 , 0>]
               [0.33 rgb < 0 , 0 , 1>]
               [0.66 rgb < 0 , 1 , 0>]
               [1.00 rgb < 1 , 0 , 0>]
            }
         }
         finish{specular 0.75}
         scale 8
      }
```

```
    }
    scale 30
    translate <-15, -15, 0>
}
```

After a material_map statement but still inside the texture statement you may apply any legal texture modifiers.

Note: no other pigment, normal, or finish statements may be added to the texture outside the material map.

The following is illegal:

```
texture {
  material_map {
    gif "matmap.gif"
    texture {T1}
    texture {T2}
    texture {T3}
  }
  finish {phong 1.0}
}
```

The finish must be individually added to each texture. Earlier versions of POV-Ray allowed such specifications but they were ignored. The above restrictions on syntax were necessary for various bug fixes. This means some POV-Ray 1.0 scenes using material maps many need minor modifications that cannot be done automatically with the version compatibility mode.

If particular index values are not used in an image then it may be necessary to supply dummy textures. It may be necessary to use a paint program or other utility to examine the map file's palette to determine how to arrange the texture list.

The textures within a material map texture may be layered but material map textures do not work as part of a layered texture. To use a layered texture inside a material map you must declare it as a texture identifier and invoke it in the texture list.

4.6 Layered Textures

It is possible to create a variety of special effects using layered textures. A layered texture consists of several textures that are partially transparent and are laid one on top of the other to create a more complex texture. The different texture layers show through the transparent portions to create the appearance of one texture that is a combination of several textures.

You create layered textures by listing two or more textures one right after the other. The last texture listed will be the top layer, the first one listed will be the bottom layer. All textures in a layered texture other than the bottom layer should have some transparency. For example:

```
object {
  My_Object
  texture {T1}  // the bottom layer
  texture {T2}  // a semi-transparent layer
```

```
    texture {T3}  // the top semi-transparent layer
  }
```

In this example T2 shows only where T3 is transparent and T1 shows only where T2 and T3 are transparent.

The color of underlying layers is filtered by upper layers but the results do not look exactly like a series of transparent surfaces. If you had a stack of surfaces with the textures applied to each, the light would be filtered twice: once on the way in as the lower layers are illuminated by filtered light and once on the way out. Layered textures do not filter the illumination on the way in. Other parts of the lighting calculations work differently as well. The results look great and allow for fantastic looking textures but they are simply different from multiple surfaces. See `stones.inc` in the standard include files directory for some magnificent layered textures.

Note: in versions predating POV-Ray 3.5, `filter` used to work the same as `transmit` in layered textures. It has been changed to work as filter should. This can change the appearance of "pre 3.5" textures a lot. The `#version` directive can be used to get the "pre 3.5" behaviour.

Note: layered textures must use the `texture` wrapped around any pigment, normal or finish statements. Do not use multiple pigment, normal or finish statements without putting them inside the texture statement.

Layered textures may be declared. For example

```
  #declare Layered_Examp =
    texture {T1}
    texture {T2}
    texture {T3}
```

may be invoked as follows:

```
  object {
    My_Object
    texture {
      Layer_Examp
      // Any pigment, normal or finish here
      // modifies the bottom layer only.
    }
  }
```

Note: No macros are allowed in layered textures. The problem is that if a macro would contain a declare the parser could no longer guess that two or more texture identifiers are supposed to belong to the layered texture and not some other declare.

If you wish to use a layered texture in a block pattern, such as `checker`, `hexagon`, or `brick`, or in a `material_map`, you must declare it first and then reference it inside a single texture statement. A patterned texture cannot be used as a layer in a layered texture however you may use layered textures as any of the textures contained within a patterned texture.

4.7 UV Mapping

All textures in POV-Ray are defined in 3 dimensions. Even planar image mapping is done this way. However, it is sometimes more desirable to have the texture defined for the surface of the object. This is especially true for bicubic_patch objects and mesh objects, that can be stretched and compressed. When the object is stretched or compressed, it would be nice for the texture to be *glued* to the object's surface and follow the object's deformations.

When uv_mapping is used, then that object's texture will be mapped to it using surface coordinates (u and v) instead of spatial coordinates (x, y, and z). This is done by taking a slice of the object's regular 3D texture from the XY plane (Z=0) and wrapping it around the surface of the object, following the object's surface coordinates.

Note: some textures should be rotated to fit the slice in the XY plane.

Syntax:

```
texture {
 uv_mapping pigment{PIGMENT_BODY} | pigment{uv_mapping PIGMENT_BODY}
 uv_mapping normal {NORMAL_BODY } | normal {uv_mapping NORMAL_BODY }
 uv_mapping texture{TEXTURE_BODY} | texture{uv_mapping TEXTURE_BODY)
}
```

4.7.1 Supported Objects

Surface mapping is currently defined for the following objects:

- **bicubic_patch** : UV coordinates are based on the patch's parametric coordinates. They stretch with the control points. The default range is (0..1) and can be changed.

- **mesh, mesh2** : UV coordinates are defined for each vertex and interpolated between.

- **lathe, sor** : modified spherical mapping... the u coordinate (0..1) wraps around the y axis, while the v coordinate is linked to the object's control points (also ranging 0..1).
  Surface of Revolution also has special disc mapping on the end caps if the object is not 'open'.

- **sphere** : boring spherical mapping.

- **box** : the image is *wrapped* around the box, as shown below.

- **parametric** : In this case the map is not taken from a "fixed" set of coordinates but the map is taken from the area defined by the boundaries of the uv-space, in

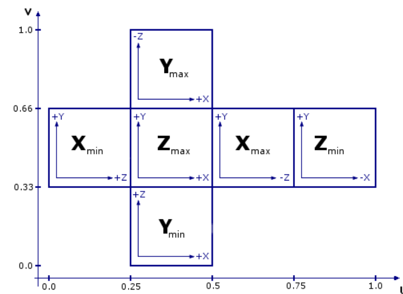

Figure 4.1: UV Boxmap

which the parametric surface has to be calculated.

- **torus** : The map is taken from the area <0,0><1,1> where the u-coordinate is wrapped around the major radius and the the v-coordinate is wrapped around the minor radius.

4.7.2 UV Vectors

With the keyword `uv_vectors`, the UV coordinates of the corners can be controlled for bicubic patches and standard triangle mesh.

For bicubic patches the UV coordinates can be specified for each of the four corners of the patch. This goes right before the control points.
The syntax is:

```
uv_vectors <corner1>,<corner2>,<corner3>, <corner4>
```

with default

```
uv_vectors <0,0>,<1,0>,<1,1>,<0,1>
```

For standard triangle meshes (not mesh2) you can specify the UV coordinates for each of the three vertices `uv_vectors <uv1>,<uv2>,<uv3>` inside each mesh triangle. This goes right after the coordinates (or coordinates & normals with smooth triangles) and right before the texture.
Example:

```
mesh {
  triangle {
    <0,0,0>, <0.5,0,0>, <0.5,0.5,0>
    uv_vectors <0,0>, <1,0>, <1,1>
  }
  triangle {
    <0,0,0>, <0.5,0.5,0>, <0,0.5,0>
    uv_vectors <0,0>, <1,1>, <0,1>
  }
  texture {
    uv_mapping pigment {
      image_map {
        sys "SomeImage"
```

```
          map_type 0
          interpolate 0
        }
      }
    }
  }
```

4.8 Triangle Texture Interpolation

This feature is utilized in a number of visualization approaches: triangles with individual textures for each vertex, which are interpolated during rendering.

Syntax:

```
MESH_TRIANGLE:
  triangle {
    <Corner_1>,
    <Corner_2>,
    <Corner_3>
    [MESH_TEXTURE]
  }   |
  smooth_triangle {
    <Corner_1>, <Normal_1>,
    <Corner_2>, <Normal_2>,
    <Corner_3>, <Normal_3>
    [MESH_TEXTURE]
  }

MESH_TEXTURE:
  texture { TEXTURE_IDENTIFIER } |
  texture_list {
    TEXTURE_IDENTIFIER TEXTURE_IDENTIFIER TEXTURE_IDENTIFIER
  }
```

To specify three vertex textures for the triangle, simply use `texture_list` instead of texture.

4.9 Interior Texture

Syntax:

```
object {
  texture { TEXTURE_ITEMS... }
  interior_texture { TEXTURE_ITEMS...}
}
```

All surfaces have an exterior and interior surface. The `interior_texture` simply allows to specify a separate texture for the interior surface of the object. For objects with no well defined inside/outside (bicubic_patch, triangle, ...) the `interior_texture` is applied to the backside of the surface. Interior surface textures use exactly the same

syntax and should work in exactly the same way as regular surface textures, except that they use the keyword `interior_texture` instead of `texture`.

Note: Do not confuse `interior_texture {}` with `interior {}`: the first one specifies surface properties, the second one specifies volume properties.

4.10 Cutaway Textures

Syntax:

```
difference | intersection {
  OBJECT_1_<strong>WITH</strong>_TEXTURES
  OBJECT_2_<strong>WITH_NO</strong>_TEXTURE
  cutaway_textures
}
```

When using a CSG difference or intersection to *cut* away parts of an object, it is sometimes desirable to allow the object to retain its original texture. Generally, however, the texture of the surface that was used to do the cutting will be displayed.
Also, if the cutting object was not given a texture by the user, the default texture is assigned to it.

By using the `cutaway_textures` keyword in a CSG difference or intersection, you specify that you do not want the default texture on the intersected surface, but instead, the textures of the parent objects in the CSG should be used.
POV-Ray will determine which texture(s) to use by doing insidedness tests on the objects in the difference or intersection. If the intersection point is inside an object, that object's texture will be used (and evaluated at the interior point).
If the parent object is a CSG of objects with different textures, then the textures on overlapping parts will be averaged together.

4.11 Patterns

POV-Ray uses a method called *three-dimensional solid texturing* to define the color, bumpiness and other properties of an object. You specify the way that the texture varies over a surface by specifying a *pattern*. Patterns are used in pigments, normals and texture maps as well as media density.

All patterns in POV-Ray are three dimensional. For every point in space, each pattern has a unique value. Patterns do not wrap around a surface like putting wallpaper on an object. The patterns exist in 3d and the objects are carved from them like carving an object from a solid block of wood or stone.

Consider a block of wood. It contains light and dark bands that are concentric cylinders being the growth rings of the wood. On the end of the block you see these concentric circles. Along its length you see lines that are the veins. However the pattern exists throughout the entire block. If you cut or carve the wood it reveals the pattern inside. Similarly an onion consists of concentric spheres that are visible only when you slice it. Marble stone consists of wavy layers of colored sediments that harden into rock.

These solid patterns can be simulated using mathematical functions. Other random patterns such as granite or bumps and dents can be generated using a random number system and a noise function.

In each case, the x, y, z coordinate of a point on a surface is used to compute some mathematical function that returns a float value. When used with color maps or pigment maps, that value looks up the color of the pigment to be used. In normal statements the pattern function result modifies or perturbs the surface normal vector to give a bumpy appearance. Used with a texture map, the function result determines which combinations of entire textures to be used. When used with media density it specifies the density of the particles or gasses.

The following sections describe each pattern. See the sections "Pigment", "Normal" "Patterned Textures" and "Density" for more details on how to use patterns. Unless mentioned otherwise, all patterns use the `ramp_wave` wave type by default but may use any wave type and may be used with `color_map`, `pigment_map`, `normal_map`, `slope_map`, `texture_map`, `density`, and `density_map`.

Note: Some patterns have a built in default color_map that does not result in a grey-scale pattern. This may lead to unexpected results when one of these patterns is used without a user specified color_map, for example in functions or media.

These patterns are:

- `agate`
- `bozo`
- `brick`
- `checker`
- `mandel`
- `hexagon`
- `marble`
- `radial`
- `wood`

4.11.1 Agate

The `agate` pattern is a banded pattern similar to marble but it uses a specialized built-in turbulence function that is different from the traditional turbulence. The traditional turbulence can be used as well but it is generally not necessary because agate is already very turbulent. You may control the amount of the built-in turbulence by adding the optional `agate_turb` keyword followed by a float value. For example:

```
pigment {
  agate
  agate_turb 0.5
  color_map {MyMap}
}
```

The agate pattern has a default color_map built in that results in a brown and white pattern with smooth transitions.

Agate as used in a normal:

```
normal {
  agate [Bump_Size]
  [MODIFIERS...]
}
```

4.11.2 Average

Technically average is not a pattern type but it is listed here because the syntax is similar to other patterns. Typically a pattern type specifies how colors or normals are chosen from a pigment_map, texture_map, density_map, or normal_map , however average tells POV-Ray to average together all of the patterns you specify. Average was originally designed to be used in a normal statement with a normal_map as a method of specifying more than one normal pattern on the same surface. However average may be used in a pigment statement with a pigment_map or in a texture statement with a texture_map or media density with density_map to average colors too.

When used with pigments, the syntax is:

```
AVERAGED_PIGMENT:
    pigment
    {
        pigment_map
        {
            PIGMENT_MAP_ENTRY...
        }
    }
PIGMENT_MAP_ENTRY:
    [ [Weight] PIGMENT_BODY ]
```

Where *Weight* is an optional float value that defaults to 1.0 if not specified. This weight value is the relative weight applied to that pigment. Each *PIGMENT_BODY* is anything which can be inside a pigment{...} statement. The pigment keyword and {} braces need not be specified.

Note: that the [] brackets are part of the actual *PIGMENT_MAP_ENTRY*. They are not notational symbols denoting optional parts. The brackets surround each entry in the pigment_map.

There may be from 2 to 256 entries in the map.

For example

```
pigment {
  average
  pigment_map {
    [1.0  Pigment_1]
    [2.0  Pigment_2]
    [0.5  Pigment_3]
  }
}
```

All three pigments are evaluated. The weight values are multiplied by the resulting color. It is then divided by the total of the weights which, in this example is 3.5. When used with `texture_map` or `density_map` it works the same way.

When used with a `normal_map` in a normal statement, multiple copies of the original surface normal are created and are perturbed by each pattern. The perturbed normals are then weighted, added and normalized.

See the sections "Pigment Maps and Pigment Lists", "Normal Maps and Normal Lists", "Texture Maps", and "Density Maps and Density Lists" for more information.

4.11.3 Boxed

The `boxed` pattern creates a 2x2x2 unit cube centered at the origin. It is computed by: *value =1.0- min(1, max(abs(X), abs(Y), abs(Z)))* It starts at 1.0 at the origin and decreases to a minimum value of 0.0 as it approaches any plane which is one unit from the origin. It remains at 0.0 for all areas beyond that distance. This pattern was originally created for use with `halo` or `media` but it may be used anywhere any pattern may be used.

4.11.4 Bozo

The `bozo` pattern is a very smooth, random noise function that is traditionally used with some turbulence to create clouds. The `spotted` pattern is identical to `bozo` but in early versions of POV-Ray spotted did not allow turbulence to be added. Turbulence can now be added to any pattern so these are redundant but both are retained for backwards compatibility. The `bumps` pattern is also identical to `bozo` when used anywhere except in a `normal` statement. When used as a normal pattern, `bumps` uses a slightly different method to perturb the normal with a similar noise function.

The `bozo` noise function has the following properties:

1. It's defined over 3D space i.e., it takes x, y, and z and returns the noise value there.

2. If two points are far apart, the noise values at those points are relatively random.

3. If two points are close together, the noise values at those points are close to each other.

You can visualize this as having a large room and a thermometer that ranges from 0.0 to 1.0. Each point in the room has a temperature. Points that are far apart have relatively random temperatures. Points that are close together have close temperatures. The temperature changes smoothly but randomly as we move through the room.

Now let's place an object into this room along with an artist. The artist measures the temperature at each point on the object and paints that point a different color depending on the temperature. What do we get? A POV-Ray bozo texture!

The `bozo` pattern has a default color_map built in that results in a green, blue, red and white pattern with sharp transitions.

Note: The appearance of the bozo pattern depends on the noise_generator used. The default type is 2. This may be changed using the `noise_generator` keyword (See section "Pattern Modifiers / Noise_generator").

4.11.5 Brick

The `brick` pattern generates a pattern of bricks. The bricks are offset by half a brick length on every other row in the x- and z-directions. A layer of mortar surrounds each brick. The syntax is given by

```
pigment {
  brick COLOR_1, COLOR_2
  [brick_size <Size>] [mortar Size]
}
```

where *COLOR_1* is the color of the mortar and *COLOR_2* is the color of the brick itself. If no colors are specified a default deep red and dark gray are used. The default size of the brick and mortar together is <8, 3, 4.5> units. The default thickness of the mortar is 0.5 units. These values may be changed using the optional `brick_size` and `mortar` pattern modifiers. You may also use pigment statements in place of the colors. For example:

```
pigment {
  brick pigment{Jade}, pigment{Black_Marble}
}
```

This example uses normals:

```
normal { brick 0.5 }
```

The float value is an optional bump size. You may also use full normal statements. For example:

```
normal {
  brick normal{bumps 0.2}, normal{granite 0.3}
}
```

When used with textures, the syntax is

```
texture {
  brick texture{T_Gold_1A}, texture{Stone12}
}
```

This is a block pattern which cannot use wave types, `color_map`, or `slope_map` modifiers.

The `brick` pattern has a default color_map built in that results in red bricks and grey mortar.

4.11.6 Bumps

The `bumps` pattern was originally designed only to be used as a normal pattern. It uses a very smooth, random noise function that creates the look of rolling hills when scaled

large or a bumpy orange peel when scaled small. Usually the bumps are about 1 unit apart.

When used as a normal pattern, this pattern uses a specialized normal perturbation function. This means that the pattern cannot be used with `normal_map`, `slope_map` or wave type modifiers in a `normal` statement.

When used as a pigment pattern or texture pattern, the `bumps` pattern is identical to `bozo` or `spotted` and is similar to normal bumps but is not identical as are most normals when compared to pigments.

Note: The appearance of the bumps pattern depends on the noise_generator used. The default type is 2. This may be changed using the `noise_generator` keyword (See section "Pattern Modifiers / Noise_generator").

4.11.7 Cells

The `cells` pattern fills 3d space with unit cubes. Each cube gets a random value from 0 to 1.

`cells` is not very suitable as a normal as it has no smooth transitions of one grey value to another.

4.11.8 Checker

The `checker` pattern produces a checkered pattern consisting of alternating squares of two colors. The syntax is:

```
pigment { checker [COLOR_1 [, COLOR_2]] [PATTERN_MODIFIERS...] }
```

If no colors are specified then default blue and green colors are used.

The checker pattern is actually a series of cubes that are one unit in size. Imagine a bunch of 1 inch cubes made from two different colors of modeling clay. Now imagine arranging the cubes in an alternating check pattern and stacking them in layer after layer so that the colors still alternate in every direction. Eventually you would have a larger cube. The pattern of checks on each side is what the POV-Ray checker pattern produces when applied to a box object. Finally imagine cutting away at the cube until it is carved into a smooth sphere or any other shape. This is what the checker pattern would look like on an object of any kind.

You may also use pigment statements in place of the colors. For example:

```
pigment { checker pigment{Jade}, pigment{Black_Marble} }
```

This example uses normals:

```
normal { checker 0.5 }
```

The float value is an optional bump size. You may also use full normal statements. For example:

```
normal {
  checker normal{gradient x scale .2},
          normal{gradient y scale .2}
```

```
}
```

When used with textures, the syntax is

```
texture { checker texture{T_Wood_3A},texture{Stone12} }
```

The `checker` pattern has a default color_map built in that results in blue and green tiles.

This use of checker as a texture pattern replaces the special tiles texture in previous versions of POV-Ray. You may still use `tiles` but it may be phased out in future versions so checker textures are best.

This is a block pattern which cannot use wave types, `color_map`, or `slope_map` modifiers.

4.11.9 Crackle Patterns

The `crackle` pattern is a set of random tiled multifaceted cells.

There is a choice between different types:

Standard Crackle
Mathematically, the set crackle(p)=0 is a 3D Voronoi diagram of a field of semi random points and crackle(p) < 0 is the distance from the set along the shortest path (a Voronoi diagram is the locus of points equidistant from their two nearest neighbors from a set of disjoint points, like the membranes in suds are to the centers of the bubbles).

With a large scale and no turbulence it makes a pretty good stone wall or floor.
With a small scale and no turbulence it makes a pretty good crackle ceramic glaze.
Using high turbulence it makes a good marble that avoids the problem of apparent parallel layers in traditional marble.

Form

```
pigment {
  crackle form <FORM_VECTOR>
  [PIGMENT_ITEMS ...]
}
normal {
  crackle [Bump_Size]
  form <FORM_VECTOR>
  [NORMAL_ITEMS ...]
}
```

Form determines the linear combination of distances used to create the pattern. Form is a vector.
The first component determines the multiple of the distance to the closest point to be used in determining the value of the pattern at a particular point.
The second component determines the coefficient applied to the second-closest distance.
The third component corresponds to the third-closest distance.

The standard form is <-1,1,0> (also the default), corresponding to the difference in the distances to the closest and second-closest points in the cell array. Another commonly-used form is <1,0,0>, corresponding to the distance to the closest point, which pro-

duces a pattern that looks roughly like a random collection of intersecting spheres or cells.
Other forms can create very interesting effects, but it's best to keep the sum of the coefficients low.
If the final computed value is too low or too high, the resultant pigment will be saturated with the color at the low or high end of the color_map. In this case, try multiplying the form vector by a constant.

Metric

```
pigment {
  crackle metric METRIC_VALUE
  [PIGMENT_ITEMS ...]
}
normal {
  crackle [Bump_Size]
  metric METRIC_VALUE
  [NORMAL_ITEMS ...]
}
```

Changing the metric changes the function used to determine which cell center is closer, for purposes of determining which cell a particular point falls in. The standard Euclidean distance function has a metric of 2. Changing the metric value changes the boundaries of the cells. A metric value of 3, for example, causes the boundaries to curve, while a very large metric constrains the boundaries to a very small set of possible orientations.
The default for metric is 2, as used by the standard crackle texture.
Metrics other than 1 or 2 can lead to substantially longer render times, as the method used to calculate such metrics is not as efficient.

Offset

```
pigment {
  crackle offset OFFSET_VALUE
  [PIGMENT_ITEMS ...]
}
normal {
  crackle [Bump_Size]
  offset OFFSET_VALUE
  [NORMAL_ITEMS ...]
}
```

The offset is used to displace the pattern from the standard xyz space along a fourth dimension.
It can be used to round off the "pointy" parts of a cellular normal texture or procedural heightfield by keeping the distances from becoming zero.
It can also be used to move the calculated values into a specific range if the result is saturated at one end of the color_map.
The default offset is zero.

Solid

```
pigment {
  crackle solid
  [PIGMENT_ITEMS ...]
```

```
  }
  normal {
    crackle [Bump_Size]
    solid
    [NORMAL_ITEMS ...]
  }
```

Causes the same value to be generated for every point within a specific cell. This has practical applications in making easy stained-glass windows or flagstones. There is no provision for mortar, but mortar may be created by layering or texture-mapping a standard crackle texture with a solid one.
The default for this parameter is off.

4.11.10 Cylindrical

The `cylindrical` pattern creates a one unit radius cylinder along the Y axis. It is computed by: *value = 1.0-min(1, sqrt(X^2 + Z^2))* It starts at 1.0 at the origin and decreases to a minimum value of 0.0 as it approaches a distance of 1 unit from the Y axis. It remains at 0.0 for all areas beyond that distance. This pattern was originally created for use with `halo` or `media` but it may be used anywhere any pattern may be used.

4.11.11 Density_File

The `density_file` pattern is a 3-D bitmap pattern that occupies a unit cube from location <0,0,0> to <1,1,1>. The data file is a raw binary file format created for POV-Ray called `df3` format. The syntax provides for the possibility of implementing other formats in the future. This pattern was originally created for use with `halo` or `media` but it may be used anywhere any pattern may be used. The syntax is:

```
pigment
{
    density_file df3 "filename.df3"
    [interpolate Type] [PIGMENT_MODIFIERS...]
}
```

where *"filename.df3"* is a file name of the data file.

As a normal pattern, the syntax is

```
normal
{
    density_file df3 "filename.df3" [, Bump_Size]
    [interpolate Type]
    [NORMAL_MODIFIERS...]
}
```

The optional float *Bump_Size* should follow the file name and any other modifiers follow that.

The density pattern occupies the unit cube regardless of the dimensions in voxels. It remains at 0.0 for all areas beyond the unit cube. The data in the range of 0 to 255, in case of 8 bit resolution, are scaled into a float value in the range 0.0 to 1.0.

The interpolate keyword may be specified to add interpolation of the data. The default value of zero specifies no interpolation. A value of one specifies tri-linear interpolation, a value of two specifies tri-cubic interpolation

See the sample scenes for data file include\spiral.df3,and the scenes which use it: scenes\textures\patterns\densfile.pov, scenes\interior\media\galaxy.pov for examples.

df3 file format

Header:

The df3 format consists of a 6 byte header of three 16-bit integers with high order byte first. These three values give the x,y,z size of the data in pixels (or more appropriately called *voxels*).

Data:

The header is followed by x*y*z unsigned integer bytes of data with a resolution of 8, 16 or 32 bit. The data are written with high order byte first (big-endian). The resolution of the data is determined by the size of the df3-file. That is, if the file is twice (minus header, of course) as long as an 8 bit file then it is assumed to contain 16 bit ints and if it is four times as long 32 bit ints.

4.11.12 Dents

The dents pattern was originally designed only to be used as a normal pattern. It is especially interesting when used with metallic textures. It gives impressions into the metal surface that look like dents have been beaten into the surface with a hammer. Usually the dents are about 1 unit apart.

When used as a normal pattern, this pattern uses a specialized normal perturbation function. This means that the pattern cannot be used with normal_map, slope_map or wave type modifiers in a normal statement.

When used as a pigment pattern or texture pattern, the dents pattern is similar to normal dents but is not identical as are most normals when compared to pigments.

4.11.13 Facets

```
normal {
  facets [coords SCALE_VALUE | size FACTOR]
  [NORMAL_ITEMS...]
}
```

The facets pattern is designed to be used as a normal, it is not suitable for use as a pigment: it will cause an error.
There are two forms of the facets pattern. One is most suited for use with rounded surfaces, and one is most suited for use with flat surfaces.

If `coords` is specified, the facets pattern creates facets with a size on the same order as the specified SCALE_VALUE. This version of facets is most suited for use with flat surfaces, but will also work with curved surfaces. The boundaries of the facets coincide with the boundaries of the cells in the standard crackle pattern. The coords version of this pattern may be quite similar to a crackle normal pattern with solid specified.

If `size` is specified, the facets texture uses a different function that creates facets only on curved surfaces. The FACTOR determines how many facets are created, with smaller values creating more facets, but it is not directly related to any real-world measurement. The same factor will create the same pattern of facets on a sphere of any size.
This pattern creates facets by snapping normal vectors to the closest vectors in a perturbed grid of normal vectors. Because of this, if a surface has normal vectors that do not vary along one or more axes, there will be no facet boundaries along those axes.

4.11.14 Fractal Patterns

Fractal patterns supported in POV-Ray:

- The Mandelbrot set with exponents up to 33.(The formula for these is: `z(n+1) = z(n)^p + c`, where `p` is the correspondent exponent.)
- The equivalent Julia sets.
- The magnet1 and magnet2 fractals (which are derived from some magnetic renormalization transformations; see the fractint help for more details).
  Both 'Mandelbrot' and 'Julia' versions of them are supported.

For the Mandelbrot and Julia sets, higher exponents will be slower for two reasons:

1. For the exponents 2,3 and 4 an optimized algorithm is used. Higher exponents use a generic algorithm for raising a complex number to an integer exponent, and this is a bit slower than an optimized version for a certain exponent.
2. The higher the exponent, the slower it will be. This is because the amount of operations needed to raise a complex number to an integer exponent is directly proportional to the exponent. This means that exponent 10 will be (very) roughly twice as slow as exponent 5.

Syntax:

```
MANDELBROT:
  mandel ITERATIONS [, BUMP_SIZE]
  [exponent EXPONENT]
  [exterior EXTERIOR_TYPE, FACTOR]
  [interior INTERIOR_TYPE, FACTOR]

JULIA:
  julia COMPLEX, ITERATIONS [, BUMP_SIZE]
  [exponent EXPONENT]
  [exterior EXTERIOR_TYPE, FACTOR]
  [interior INTERIOR_TYPE, FACTOR]

MAGNET MANDEL:
  magnet MAGNET_TYPE mandel ITERATIONS [, BUMP_SIZE]
```

```
  [exterior EXTERIOR_TYPE, FACTOR]
  [interior INTERIOR_TYPE, FACTOR]

MAGNET JULIA:
  magnet MAGNET_TYPE julia COMPLEX, ITERATIONS [, BUMP_SIZE]
  [exterior EXTERIOR_TYPE, FACTOR]
  [interior INTERIOR_TYPE, FACTOR]
```

Where:

`ITERATIONS` is the number of times to iterate the algorithm.

`COMPLEX` is a 2D vector denoting a complex number.

`MAGNET_TYPE` is either 1 or 2.

`exponent` is an integer between 2 and 33. If not given, the default is 2.

`interior` and `exterior` specify special coloring algorithms. You can specify one of them or both at the same time. They only work with the fractal patterns.
`EXTERIOR_TYPE` and `INTERIOR_TYPE` are integer values between 0 and 6 (inclusive). When not specified, the default value of INTERIOR_TYPE is 0 and for EXTERIOR_TYPE 1.
`FACTOR` is a float. The return value of the pattern is multiplied by `FACTOR` before returning it. This can be used to scale the value range of the pattern when using interior and exterior coloring (this is often needed to get the desired effect). The default value of FACTOR is 1.

The different values of EXTERIOR_TYPE and INTERIOR_TYPE have the following meaning:

- 0 : Returns just 1
- 1 : For exterior: The number of iterations until bailout divided by ITERATIONS. Note: this is not scaled by FACTOR (since it's internally scaled by 1/ITERATIONS instead).
  For interior: The absolute value of the smallest point in the orbit of the calculated point
- 2 : Real part of the last point in the orbit
- 3 : Imaginary part of the last point in the orbit
- 4 : Squared real part of the last point in the orbit
- 5 : Squared imaginary part of the last point in the orbit
- 6 : Absolute value of the last point in the orbit

Example:

```
box {
  <-2, -2, 0>, <2, 2, 0.1>
  pigment {
    julia <0.353, 0.288>, 30
    interior 1, 1
    color_map {
  [0 rgb 0]
      [0.2 rgb x]
```

```
        [0.4 rgb x+y]
        [1 rgb 1]
        [1 rgb 0]
      }
    }
  }
```

4.11.15 Function as pattern

Allows you to use a function { } block as pattern.

```
pigment {
   function { USER_DEFINED_FUNCTIONS }
   [PIGMENT_MODIFIERS...]
}
```

Declaring a function:
By default a function takes three parameters (x,y,z) and you do not have to explicitly specify the parameter names when declaring it. When using the identifier, the parameters must be specified.

```
#declare Foo = function { x + y + z}
pigment {
  function { Foo(x, y, z) }
    [PIGMENT_MODIFIERS...]
}
```

On the other hand, if you need more or less than three parameters when declaring a function, you also have to explicitly specify the parameter names.

```
#declare Foo = function(x,y,z,t) { x + y + z + t}
pigment {
  function { Foo(x, y, z, 4) }
    [PIGMENT_MODIFIERS...]
}
```

Using function in a normal:

```
#declare Foo = function { x + y + z}
normal {
  function { Foo(x, y, z) } [Bump_Size]
  [MODIFIERS...]
}
```

What can be used

All float expressions and operators (see section ”User-Defined Functions”) which are legal in POV-Ray. Of special interest here is the `pattern` option, that makes it possible to use patterns as functions

```
#declare FOO = function {
   pattern {
     checker
   }
```

```
}
```

User defined functions (like equations).

Since pigments can be declared as functions, they can also be used in functions. They must be declared first. When using the identifier, you have to specify which component of the color vector should be used. To do this, the dot notation is used: Function(x,y,z).red

```
#declare FOO = function {pigment { checker } }
pigment {
   function { FOO(x,y,z).green }
   [PIGMENT_MODIFIERS...]
}
```

POV-Ray has a large amount of pre-defined functions. These are mainly algebraic surfaces but there is also a mesh function and noise3d function. See section "Internal Functions" for a complete list and some explanation on the parameters to use. These internal functions can be included through the functions.inc include file.

```
#include "functions.inc"
#declare FOO = function {pigment { checker } }
pigment {
   function { FOO(x,y,z).green & f_noise3d(x*2, y*3,z)}
   [PIGMENT_MODIFIERS...]
}
```

4.11.16 Function Image

Syntax :`function Width, Height { FUNCTION_BODY }`

Not a real pattern, but listed here for convenience. This keyword defines a new 'internal' bitmap image type. The pixels of the image are derived from the Function_Body, with Function_Body either being a regular function, a pattern function or a pigment function. In case of a pigment function the output image will be in color, in case of a pattern or regular function the output image will be grayscale. All variants of grayscale pigment functions are available using the regular function syntax, too. In either case the image will use 16 bit per component

Note: functions are evaluated on the x-y plane. This is different from the pattern image type for the reason that it makes using uv functions easier.

Width and Height specify the resolution of the resulting 'internal' bitmap image. The image is taken from the square region `<0,0,0>, <1,1,0>`

The `function` statement can be used wherever an image specifier like `tga` or `png` may be used. Some uses include creating heightfields from procedural textures or wrapping a slice of a 3d texture or function around a cylinder or extrude it along an axis.

Examples:

```
plane {
   y, -1
   pigment {
      image_map {
```

```
          function 10,10 {
             pigment { checker 1,0 scale .5  }
          }
       }
       rotate x*90
    }
 }

 height_field {
    function 200,200 {
       pattern {
          bozo
       }
    }
    translate -0.5
    scale 10
    pigment {rgb 1}
 }
```

Note: that for height fields and other situations where color is not needed it is easier to use `function n,n {pattern{...}}` than `function n,n {pigment{...}}`. The pattern functions are returning a scalar, not a color vector, thus a pattern is grayscale.

4.11.17 Gradient

One of the simplest patterns is the `gradient` pattern. It is specified as

```
pigment {
    gradient <Orientation>
    [PIGMENT_MODIFIERS...]
}
```

where <*Orientation*> is a vector pointing in the direction that the colors blend. For example

```
pigment { gradient x } // bands of color vary as you move
                       // along the "x" direction.
```

produces a series of smooth bands of color that look like layers of colors next to each other. Points at x=0 are the first color in the color map. As the x location increases it smoothly turns to the last color at x=1. Then it starts over with the first again and gradually turns into the last color at x=2. In POV-Ray versions older than 3.5 the pattern reverses for negative values of x. As per POV-Ray 3.5 this is not the case anymore [1]. Using `gradient y` or `gradient z` makes the colors blend along the y- or z-axis. Any vector may be used but x, y and z are most common.

As a normal pattern, gradient generates a saw-tooth or ramped wave appearance. The syntax is

```
normal {
    gradient <Orientation> [, Bump_Size]
    [NORMAL_MODIFIERS...]
}
```

where the vector <*Orientation*> is a required parameter but the float *Bump_Size* which follows is optional.

Note: the comma is required especially if *Bump_Size* is negative.

[1] If only the range -1 to 1 was used of the old gradient, for example in a `sky_sphere`, it can be replaced by the `planar` or `marble` pattern and revert the color_map. Also rotate the pattern for other orientations than `y`. A more general solution is to use `function{abs(x)}` as a pattern instead of `gradient x` and similar for `gradient y` and `gradient z`.

4.11.18 Granite

The `granite` pattern uses a simple 1/f fractal noise function to give a good granite pattern. This pattern is used with creative color maps in `stones.inc` to create some gorgeous layered stone textures.

As a normal pattern it creates an extremely bumpy surface that looks like a gravel driveway or rough stone.

Note: The appearance of the granite pattern depends on the noise_generator used. The default type is 2. This may be changed using the `noise_generator` keyword (See section "Pattern Modifiers / Noise_generator").

4.11.19 Hexagon

The `hexagon` pattern is a block pattern that generates a repeating pattern of hexagons in the x-z-plane. In this instance imagine tall rods that are hexagonal in shape and are parallel to the y-axis and grouped in bundles like shown in the example image. Three separate colors should be specified as follows:

```
pigment {
    hexagon [COLOR_1 [, COLOR_2 [, COLOR_3]]]
    [PATTERN_MODIFIERS...]
}
```

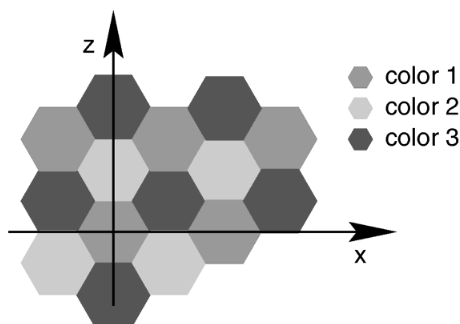

Figure 4.2: The hexagon pattern.

The three colors will repeat the hexagonal pattern with hexagon *COLOR_1* centered at the origin, *COLOR_2* in the +z-direction and *COLOR_3* to either side. Each side of

the hexagon is one unit long. The hexagonal rods of color extend infinitely in the +y- and -y-directions. If no colors are specified then default blue, green and red colors are used.

You may also use pigment statements in place of the colors. For example:

```
pigment {
  hexagon
  pigment { Jade },
  pigment { White_Marble },
  pigment { Black_Marble }
}
```

This example uses normals:

```
normal { hexagon 0.5 }
```

The float value is an optional bump size. You may also use full normal statements. For example:

```
normal {
  hexagon
    normal { gradient x scale .2 },
    normal { gradient y scale .2 },
    normal { bumps scale .2 }
}
```

When used with textures, the syntax is...

```
texture {
  hexagon
    texture { T_Gold_3A },
    texture { T_Wood_3A },
    texture { Stone12 }
}
```

The `hexagon` pattern has a default color_map built in that results in red, blue and green tiles.

This is a block pattern which cannot use wave types, `color_map`, or `slope_map` modifiers.

4.11.20 Image Pattern

Instead of placing the color of the image on the object like an image_map an image_pattern specifies an entire texture item (color, pigment, normal or texture) based on the gray value at that point.
This gray-value is checked against a list and the corresponding item is then used for the texture at that particular point. For values between listed items, an averaged texture is calculated.
It takes a standard image specification and has one option, `use_alpha` which works similar to `use_color` or `use_index`.

Syntax:

```
PIGMENT:
```

```
  pigment {
    IMAGE_PATTERN
    color_map { COLOR_MAP_BODY } |
    colour_map { COLOR_MAP_BODY } |
    pigment_map { PIGMENT_MAP_BODY }
  }

NORMAL:
  normal {
    IMAGE_PATTERN [Bump_Size]
    normal_map { NORMAL_MAP_BODY }
  }

TEXTURE:
  texture {
    IMAGE_PATTERN
    texture_map { TEXTURE_MAP_BODY }
  }

IMAGE_PATTERN
    image_pattern {
      BITMAP_TYPE "bitmap.ext"
      [IMAGE_MAP_MODS...]
    }
IMAGE_MAP_MOD:
    map_type Type | once | interpolate Type | use_alpha
ITEM_MAP_BODY:
    ITEM_MAP_IDENTIFIER | ITEM_MAP_ENTRY...
ITEM_MAP_ENTRY:
    [ GRAY_VALUE  ITEM_MAP_ENTRY... ]
```

It is also useful for creating texture "masks", like the following:

```
 texture {
   image_pattern { tga "image.tga" use_alpha }
   texture_map {
     [0 Mytex ]
     [1 pigment { transmit 1 } ]
   }
 }
```

Note: This pattern uses an image to get the gray values from. If you want exactly the same possibilities but need to get gray values from a pigment, you can use the pigment_pattern.

4.11.21 Leopard

Leopard creates regular geometric pattern of circular spots. The formula used is: *value = Sqr((sin(x)+sin(y)+sin(z))/3)*

4.11.22 Marble

The marble pattern is very similar to the gradient x pattern. The gradient pattern uses a default ramp_wave wave type which means it uses colors from the color map from 0.0 up to 1.0 at location x=1 but then jumps back to the first color for x > 1 and repeats the pattern again and again. However the marble pattern uses the triangle_wave wave type in which it uses the color map from 0 to 1 but then it reverses the map and blends from 1 back to zero. For example:

```
pigment {
  gradient x
  color_map {
    [0.0  color Yellow]
    [1.0  color Cyan]
  }
}
```

This blends from yellow to cyan and then it abruptly changes back to yellow and repeats. However replacing gradient x with marble smoothly blends from yellow to cyan as the x coordinate goes from 0.0 to 0.5 and then smoothly blends back from cyan to yellow by x=1.0.

Earlier versions of POV-Ray did not allow you to change wave types. Now that wave types can be changed for most any pattern, the distinction between marble and gradient x is only a matter of default wave types.

When used with turbulence and an appropriate color map, this pattern looks like veins of color of real marble, jade or other types of stone. By default, marble has no turbulence.

The marble pattern has a default color_map built in that results in a red, black and white pattern with smooth and sharp transitions.

4.11.23 Object Pattern

The object pattern takes an object as input. It generates a, two item, color list pattern. Whether a point is assigned to one item or the other depends on whether it is inside the specified object or not.

Object's used in the object pattern cannot have a texture and must be solid - these are the same limitations as for bounded_by and clipped_by.

Syntax:

```
object {
  OBJECT_IDENTIFIER | OBJECT {}
  LIST_ITEM_A, LIST_ITEM_B
}
```

Where OBJ_IDENTIFIER is the target object (which must be declared), or use the full object syntax. LIST_ITEM_A and LIST_ITEM_B are the colors, pigments, or whatever the pattern is controlling. LIST_ITEM_A is used for all points outside the object, and LIST_ITEM_B is used for all points inside the object.

Example:

```
pigment {
   object {
      myTextObject
      color White
      color Red
   }
   turbulence 0.15
}
```

Note: This is a block pattern which cannot use wave types, color_map, or slope_map modifiers.

4.11.24 Onion

The `onion` is a pattern of concentric spheres like the layers of an onion. *Value = mod(sqrt(Sqr(X)+Sqr(Y)+Sqr(Z)), 1.0)* Each layer is one unit thick.

4.11.25 Pigment Pattern

Use any pigment as a pattern. Instead of using the pattern directly on the object, a pigment_pattern converts the pigment to gray-scale first. For each point, the gray-value is checked against a list and the corresponding item is then used for the texture at that particular point. For values between listed items, an averaged texture is calculated.
Texture items can be color, pigment, normal or texture and are specified in a color_map, pigment_map, normal_map or texture_map.
It takes a standard pigment specification.

Syntax:

```
PIGMENT:
  pigment {
    pigment_pattern { PIGMENT_BODY }
    color_map { COLOR_MAP_BODY } |
    colour_map { COLOR_MAP_BODY } |
    pigment_map { PIGMENT_MAP_BODY }
  }

NORMAL:
  normal {
    pigment_pattern { PIGMENT_BODY } [Bump_Size]
    normal_map { NORMAL_MAP_BODY }
  }

TEXTURE:
  texture {
    pigment_pattern { PIGMENT_BODY }
    texture_map { TEXTURE_MAP_BODY }
  }

ITEM_MAP_BODY:
```

```
    ITEM_MAP_IDENTIFIER | ITEM_MAP_ENTRY...
ITEM_MAP_ENTRY:
    [ GRAY_VALUE  ITEM_MAP_ENTRY... ]
```

This pattern is also useful when parent and children patterns need to be transformed independently from each other. Transforming the pigment_pattern will not affect the child textures. When any of the child textures should be transformed, apply it to the specific MAP_ENTRY.

This can be used with any pigments, ranging from a simple checker to very complicated nested pigments. For example:

```
pigment {
  pigment_pattern {
    checker White, Black
    scale 2
    turbulence .5
  }
  pigment_map {
    [ 0, checker Red, Green scale .5 ]
    [ 1, checker Blue, Yellow scale .2 ]
  }
}
```

Note: This pattern uses a pigment to get the gray values from. If you want to get the pattern from an image, you should use the image_pattern.

4.11.26 Planar

The `planar` pattern creates a horizontal stripe plus or minus one unit above and below the X-Z plane. It is computed by: *value =1.0- min(1, abs(Y))* It starts at 1.0 at the origin and decreases to a minimum value of 0.0 as the Y values approaches a distance of 1 unit from the X-Z plane. It remains at 0.0 for all areas beyond that distance. This pattern was originally created for use with `halo` or `media` but it may be used anywhere any pattern may be used.

4.11.27 Quilted

The `quilted` pattern was originally designed only to be used as a normal pattern. The quilted pattern is so named because it can create a pattern somewhat like a quilt or a tiled surface. The squares are actually 3-D cubes that are 1 unit in size.

When used as a normal pattern, this pattern uses a specialized normal perturbation function. This means that the pattern cannot be used with `normal_map`, `slope_map` or wave type modifiers in a `normal` statement.

When used as a pigment pattern or texture pattern, the `quilted` pattern is similar to normal quilted but is not identical as are most normals when compared to pigments.

The two parameters `control0` and `control1` are used to adjust the curvature of the *seam* or *gouge* area between the `quilts`.

The syntax is:

```
    pigment { quilted [QUILTED_MODIFIERS...] }
QUILTED_MODIFIERS:
  control0 Value_0 | control1 Value_1 | PIGMENT_MODIFIERS
```

The values should generally be kept to around the 0.0 to 1.0 range. The default value is 1.0 if none is specified. Think of this gouge between the tiles in cross-section as a sloped line.

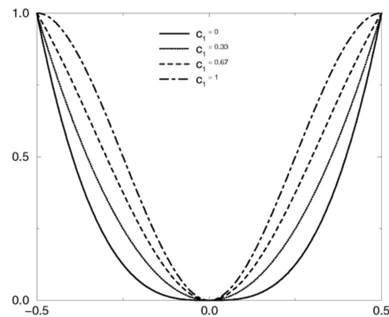

Figure 4.3: Quilted pattern with c0=0 and different values for c1.

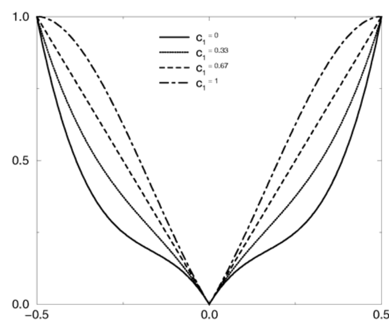

Figure 4.4: Quilted pattern with c0=0.33 and different values for c1.

This straight slope can be made to curve by adjusting the two control values. The control values adjust the slope at the top and bottom of the curve. A control values of 0 at both ends will give a linear slope, as shown above, yielding a hard edge. A control value of 1 at both ends will give an ”s” shaped curve, resulting in a softer, more rounded edge.

The syntax for use as a normal is:

```
  normal {
    quilted [Bump_Size]
    [QUILTED_MODIFIERS...]
  }
QUILTED_MODIFIERS:
  control0 Value_0 | control1 Value_1 | PIGMENT_MODIFIERS
```

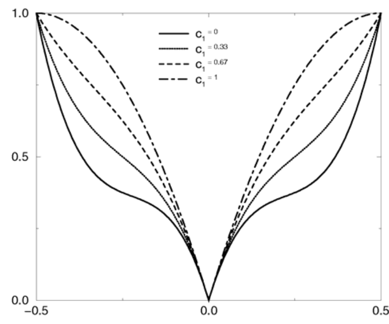

Figure 4.5: Quilted pattern with c0=0.67 and different values for c1.

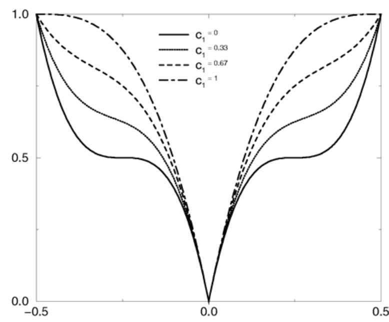

Figure 4.6: Quilted pattern with c0=1 and different values for c1.

4.11.28 Radial

The `radial` pattern is a radial blend that wraps around the +y-axis. The color for value 0.0 starts at the +x-direction and wraps the color map around from east to west with 0.25 in the -z-direction, 0.5 in -x, 0.75 at +z and back to 1.0 at +x. Typically the pattern is used with a `frequency` modifier to create multiple bands that radiate from the y-axis. For example:

```
pigment {
  radial color_map{[0.5 Black][0.5 White]}
  frequency 10
}
```

creates 10 white bands and 10 black bands radiating from the y axis.

The `radial` pattern has a default color_map built in that results in a yellow, magenta and cyan pattern with smooth transitions.

4.11.29 Ripples

The `ripples` pattern was originally designed only to be used as a normal pattern. It makes the surface look like ripples of water. The ripples radiate from 10 random locations inside the unit cube area <0,0,0> to <1,1,1>. Scale the pattern to make the centers closer or farther apart.

Usually the ripples from any given center are about 1 unit apart. The `frequency` keyword changes the spacing between ripples. The `phase` keyword can be used to move the ripples outwards for realistic animation.

The number of ripple centers can be changed with the global parameter global_settings{number_of_waves Count }

somewhere in the scene. This affects the entire scene. You cannot change the number of wave centers on individual patterns. See section "Number_Of_Waves" for details.

When used as a normal pattern, this pattern uses a specialized normal perturbation function. This means that the pattern cannot be used with `normal_map`, `slope_map` or wave type modifiers in a `normal` statement.

When used as a pigment pattern or texture pattern, the `ripples` pattern is similar to normal ripples but is not identical as are most normals when compared to pigments.

4.11.30 Slope

The `slope` pattern uses the normal of a surface to calculate the slope at a given point. It then creates the pattern value dependent on the slope and optionally the altitude. It can be used for pigments, normals and textures, but not for media densities. For pigments the syntax is:

```
pigment {
  slope {
    <Direction> [, Lo_slope, Hi_slope ]
    [ altitude <Altitude> [, Lo_alt, Hi_alt ]]
```

```
    }
    [PIGMENT_MODIFIERS...]
  }
```

The slope value at a given point is dependent on the angle between the `<Direction>` vector and the normal of the surface at that point. For example:
- When the surface normal points in the opposite direction of the `<Direction>` vector (180 degrees), the slope is 0.0.
- When the surface normal is perpendicular to the `<Direction>` vector (90 degrees), the slope is 0.5.
- When the surface normal is parallel to the `<Direction>` vector (0 degrees), the slope is 1.0.

When using the simplest variant of the syntax:

```
  slope { <Direction> }
```

the pattern value for a given point is the same as the slope value. `<Direction>` is a 3-dimensional vector and will usually be `<0,-1,0>` for landscapes, but any direction can be used.

By specifying `Lo_slope` and `Hi_slope` you get more control:

```
  slope { <Direction>, Lo_slope, Hi_slope }
```

`Lo_slope` and `Hi_slope` specifies which range of slopes are used, so you can control which slope values return which pattern values. `Lo_slope` is the slope value that returns 0.0 and `Hi_slope` is the slope value that returns 1.0.

For example, if you have a height_field and `<Direction>` is set to `<0,-1,0>`, then the slope values would only range from 0.0 to 0.5 because height_fields can't have overhangs. If you don't specify `Lo_slope` and `Hi_slope`, you should keep in mind that the texture for the flat (horizontal) areas must be set at 0.0 and the texture for the steep (vertical) areas at 0.5 when designing the texture_map. The part from 0.5 up to 1.0 is not used then. But, by setting `Lo_slope` and `Hi_slope` to 0.0 and 0.5 respectively, the slope range will be stretched over the entire map, and the texture_map can then be defined from 0.0 to 1.0.

By adding an optional `<Altitude>` vector:

```
  slope {
    <Direction>
    altitude <Altitude>
  }
```

the pattern will be influenced not only by the slope but also by a special gradient. `<Altitude>` is a 3-dimensional vector that specifies the direction of the gradient. When `<Altitude>` is specified, the pattern value is a weighted average of the slope value and the gradient value. The weights are the lengths of the vectors `<Direction>` and `<Altitude>`. So if `<Direction>` is much longer than `<Altitude>` it means that the slope has greater effect on the results than the gradient. If on the other hand `<Altitude>` is longer, it means that the gradient has more effect on the results than the slope.

When adding the `<Altitude>` vector, the default gradient is defined from 0 to 1 units along the specified axis. This is fine when your object is defined within this range,

otherwise a correction is needed. This can be done with the optional Lo_alt and Hi_alt parameters:

```
slope {
  <Direction>
  altitude <Altitude>, Lo_alt, Hi_alt
}
```

They define the range of the gradient along the axis defined by the <Altitude> vector.

For example, with an <Altitude> vector set to y and an object going from -3 to 2 on the y axis, the Lo_alt and Hi_alt parameters should be set to -3 and 2 respectively.

Note:

- You may use the turbulence keyword inside slope pattern definitions but it may cause unexpected results. Turbulence is a 3-dimensional distortion of a pattern. Since slope is only defined on surfaces of objects, a 3-dimensional turbulence is not applicable to the slope component. However, if you are using altitude, the altitude component of the pattern will be affected by turbulence.
- If your object is larger than the range of altitude you have specified, you may experience unexpected discontinuities. In that case it's best to adjust the Lo_alt and Hi_alt values so they fit to your object.
- The slope pattern doesn't work for the sky_sphere, because the sky_sphere is a background feature and does not have a surface. similarly, it does not work for media densities.

4.11.31 Spherical

The spherical pattern creates a one unit radius sphere, with its center at the origin. It is computed by: *value = 1.0-min(1, sqrt(X^2 + Y^2 + Z^2))* It starts at 1.0 at the origin and decreases to a minimum value of 0.0 as it approaches a distance of 1 unit from the origin in any direction. It remains at 0.0 for all areas beyond that distance. This pattern was originally created for use with halo or media but it may be used anywhere any pattern may be used.

4.11.32 Spiral1

The spiral1 pattern creates a spiral that winds around the z-axis similar to a screw. When viewed sliced in the x-y plane, it looks like the spiral arms of a galaxy. Its syntax is:

```
pigment
{
    spiral1 Number_of_Arms
    [PIGMENT_MODIFIERS...]
}
```

The *Number_of_Arms* value determines how may arms are winding around the z-axis.

As a normal pattern, the syntax is

```
normal
{
    spiral1 Number_of_Arms [, Bump_Size]
    [NORMAL_MODIFIERS...]
}
```

where the `Number_of_Arms` value is a required parameter but the float *`Bump_Size`* which follows is optional.

Note: the comma is required especially if *Bump_Size* is negative.

The pattern uses the `triangle_wave` wave type by default but may use any wave type.

4.11.33 Spiral2

The `spiral2` pattern creates a double spiral that winds around the z-axis similar to `spiral1` except that it has two overlapping spirals which twist in opposite directions. The result sometimes looks like a basket weave or perhaps the skin of pineapple. The center of a sunflower also has a similar double spiral pattern. Its syntax is:

```
pigment
{
    spiral2 Number_of_Arms
    [PIGMENT_MODIFIERS...]
}
```

The *`Number_of_Arms`* value determines how may arms are winding around the z-axis. As a normal pattern, the syntax is

```
normal
{
    spiral2 Number_of_Arms [, Bump_Size]
    [NORMAL_MODIFIERS...]
}
```

where the `Number_of_Arms` value is a required parameter but the float *`Bump_Size`* which follows is optional.

Note: the comma is required especially if *Bump_Size* is negative. The pattern uses the `triangle_wave` wave type by default but may use any wave type.

4.11.34 Spotted

The `spotted` pattern is identical to the `bozo` pattern. Early versions of POV-Ray did not allow turbulence to be used with spotted. Now that any pattern can use turbulence there is no difference between `bozo` and `spotted`. See section ”Bozo” for details.

4.11.35 Waves

The `waves` pattern was originally designed only to be used as a normal pattern. It makes the surface look like waves on water. The `waves` pattern looks similar to the `ripples` pattern except the features are rounder and broader. The effect is to make waves that

look more like deep ocean waves. The waves radiate from 10 random locations inside the unit cube area <0,0,0> to <1,1,1>. Scale the pattern to make the centers closer or farther apart.

Usually the waves from any given center are about 1 unit apart. The `frequency` keyword changes the spacing between waves. The `phase` keyword can be used to move the waves outwards for realistic animation.

The number of wave centers can be changed with the global parameter

```
global_settings { number_of_waves Count }
```

somewhere in the scene. This affects the entire scene. You cannot change the number of wave centers on individual patterns. See section "Number_Of_Waves" for details.

When used as a normal pattern, this pattern uses a specialized normal perturbation function. This means that the pattern cannot be used with `normal_map`, `slope_map` or wave type modifiers in a `normal` statement.

When used as a pigment pattern or texture pattern, the `waves` pattern is similar to normal waves but is not identical as are most normals when compared to pigments.

4.11.36 Wood

The `wood` pattern consists of concentric cylinders centered on the z-axis. When appropriately colored, the bands look like the growth rings and veins in real wood. Small amounts of turbulence should be added to make it look more realistic. By default, wood has no turbulence.

Unlike most patterns, the `wood` pattern uses the `triangle_wave` wave type by default. This means that like marble, wood uses color map values 0.0 to 1.0 then repeats the colors in reverse order from 1.0 to 0.0. However you may use any wave type.

The `wood` pattern has a default color_map built in that results in a light and dark brown pattern with sharp transitions.

4.11.37 Wrinkles

The `wrinkles` pattern was originally designed only to be used as a normal pattern. It uses a 1/f noise pattern similar to granite but the features in wrinkles are sharper. The pattern can be used to simulate wrinkled cellophane or foil. It also makes an excellent stucco texture.

When used as a normal pattern, this pattern uses a specialized normal perturbation function. This means that the pattern cannot be used with `normal_map`, `slope_map` or wave type modifiers in a `normal` statement.

When used as a pigment pattern or texture pattern, the `wrinkles` pattern is similar to normal wrinkles but is not identical as are most normals when compared to pigments.

Note: The appearance of the wrinkles pattern depends on the noise_generator used. The default type is 2. This may be changed using the `noise_generator` keyword (See section "Pattern Modifiers / Noise_generator").

4.12 Pattern Modifiers

Pattern modifiers are statements or parameters which modify how a pattern is evaluated or tells what to do with the pattern. The complete syntax is:

```
PATTERN_MODIFIER:
    BLEND_MAP_MODIFIER | AGATE_MODIFIER | DENSITY_FILE_MODIFIER |
    QUILTED_MODIFIER | BRICK_MODIFIER | SLOPE_MODIFIER |
    noise_generator Number| turbulence <Amount> |
    octaves Count | omega Amount | lambda Amount |
    warp { [WARP_ITEMS...] } | TRANSFORMATION
BLEND_MAP_MODIFIER:
    frequency Amount | phase Amount | ramp_wave | triangle_wave |
    sine_wave | scallop_wave | cubic_wave | poly_wave [Exponent]
AGATE_MODIFIER:
    agate_turb Value
BRICK_MODIFIER:
    brick_size Size | mortar Size
DENSITY_FILE_MODIFIER:
    interpolate Type
SLOPE_MODIFIERS:
<Altitude>
<Lo_slope,Hi_slope>
    <Lo_alt,Hi_alt>
QUILTED_MODIFIER:
    control0 Value | control1 Value
PIGMENT_MODIFIER:
    PATTERN_MODIFIER | COLOR_LIST | PIGMENT_LIST |
    color_map { COLOR_MAP_BODY } | colour_map { COLOR_MAP_BODY } |
    pigment_map{ PIGMENT_MAP_BODY } | quick_color COLOR |
    quick_colour COLOR
COLOR NORMAL_MODIFIER:
    PATTERN_MODIFIER | NORMAL_LIST |
    normal_map { NORMAL_MAP_BODY } | slope_map{ SLOPE_MAP_BODY } |
    bump_size Amount
TEXTURE_PATTERN_MODIFIER:
    PATTERN_MODIFIER | TEXTURE_LIST |
    texture_map{ TEXTURE_MAP_BODY }
DENSITY_MODIFIER:
    PATTERN_MODIFIER | DENSITY_LIST | COLOR_LIST |
    color_map { COLOR_MAP_BODY } | colour_map { COLOR_MAP_BODY } |
    density_map { DENSITY_MAP_BODY }
```

Default values for pattern modifiers:

```
dist_exp        : 0
falloff         : 2.0
frequency       : 1.0
lambda          : 2.0
major_radius    : 1
map_type        : 0
noise_generator : 2
octaves         : 6
omega           : 0.5
orientation     : <0,0,1>
```

```
phase          : 0.0
poly_wave      : 1.0
strength       : 1.0
turbulence     : <0,0,0>
```

The modifiers *PIGMENT_LIST*, `quick_color`, and `pigment_map` apply only to pigments. See section "Pigment" for details on these pigment-specific pattern modifiers.

The modifiers *COLOR_LIST* and `color_map` apply only to pigments and densities. See sections "Pigment" and "Density" for details on these pigment-specific pattern modifiers.

The modifiers *NORMAL_LIST*, `bump_size`, `slope_map` and `normal_map` apply only to normals. See section "Normal" for details on these normal-specific pattern modifiers.

The *TEXTURE_LIST* and `texture_map` modifiers can only be used with patterned textures. See section "Texture Maps" for details.

The *DENSITY_LIST* and `density_map` modifiers only work with `media{density{..}}` statements. See "Density" for details.

The `agate_turb` modifier can only be used with the `agate` pattern. See "Agate" for details.

The `brick_size` and `mortar` modifiers can only be used with the `brick` pattern. See "Brick" for details.

The `control0` and `control1` modifiers can only be used with the `quilted` pattern. See "Quilted" for details.

The `interpolate` modifier can only be used with the `density_file` pattern. See "Density_File" for details.

The general purpose pattern modifiers in the following sections can be used with `pigment`, `normal`, `texture`, or `density` patterns.

4.12.1 Transforming Patterns

The most common pattern modifiers are the transformation modifiers `translate`, `rotate`, `scale`, `transform`, and `matrix`. For details on these commands see section "Transformations".

These modifiers may be placed inside pigment, normal, texture, and density statements to change the position, size and orientation of the patterns.

Transformations are performed in the order in which you specify them. However in general the order of transformations relative to other pattern modifiers such as `turbulence`, `color_map` and other maps is not important. For example scaling before or after turbulence makes no difference. The turbulence is done first, then the scaling regardless of which is specified first. However the order in which transformations are performed relative to `warp` statements is important. See "Warps" for details.

4.12.2 Frequency and Phase

The frequency and phase modifiers act as a type of scale and translate modifiers for various blend maps. They only have effect when blend maps are used. Blend maps are color_map, pigment_map, normal_map, slope_map, density_map, and texture_map. This discussion uses a color map as an example but the same principles apply to the other blend map types.

The frequency keyword adjusts the number of times that a color map repeats over one cycle of a pattern. For example gradient covers color map values 0 to 1 over the range from x=0 to x=1. By adding frequency 2.0 the color map repeats twice over that same range. The same effect can be achieved using scale 0.5*x so the frequency keyword isn't that useful for patterns like gradient.

However the radial pattern wraps the color map around the +y-axis once. If you wanted two copies of the map (or 3 or 10 or 100) you'd have to build a bigger map. Adding frequency 2.0 causes the color map to be used twice per revolution. Try this:

```
pigment {
  radial
  color_map{[0.5 color Red][0.5 color White]}
  frequency 6
}
```

The result is six sets of red and white radial stripes evenly spaced around the object.

The float after frequency can be any value. Values greater than 1.0 causes more than one copy of the map to be used. Values from 0.0 to 1.0 cause a fraction of the map to be used. Negative values reverses the map.

The phase value causes the map entries to be shifted so that the map starts and ends at a different place. In the example above if you render successive frames at phase 0 then phase 0.1, phase 0.2, etc. you could create an animation that rotates the stripes. The same effect can be easily achieved by rotating the radial pigment using rotate y*Angle but there are other uses where phase can be handy.

Sometimes you create a great looking gradient or wood color map but you want the grain slightly adjusted in or out. You could re-order the color map entries but that's a pain. A phase adjustment will shift everything but keep the same scale. Try animating a mandel pigment for a color palette rotation effect.

These values work by applying the following formula

*New_Value = fmod (Old_Value * Frequency + Phase, 1.0).*

The frequency and phase modifiers have no effect on block patterns checker, brick, and hexagon nor do they effect image_map, bump_map or material_map. They also have no effect in normal statements when used with bumps, dents, quilted or wrinkles because these normal patterns cannot use normal_map or slope_map.

They can be used with normal patterns ripples and waves even though these two patterns cannot use normal_map or slope_map either. When used with ripples or waves, frequency adjusts the space between features and phase can be adjusted from 0.0 to 1.0 to cause the ripples or waves to move relative to their center for animating the features.

4.12.3 Waveforms

POV-Ray allows you to apply various wave forms to the pattern function before applying it to a blend map. Blend maps are `color_map`, `pigment_map`, `normal_map`, `slope_map`, `density_map`, and `texture_map`.

Most of the patterns which use a blend map, use the entries in the map in order from 0.0 to 1.0. The effect can most easily be seen when these patterns are used as normal patterns with no maps. Patterns such as `gradient` or `onion` generate a groove or slot that looks like a ramp that drops off sharply. This is called a `ramp_wave` wave type and it is the default wave type for most patterns. However the `wood` and `marble` patterns use the map from 0.0 to 1.0 and then reverses it and runs it from 1.0 to 0.0. The result is a wave form which slopes upwards to a peak, then slopes down again in a `triangle_wave`. In earlier versions of POV-Ray there was no way to change the wave types. You could simulate a triangle wave on a ramp wave pattern by duplicating the map entries in reverse, however there was no way to use a ramp wave on wood or marble.

Now any pattern that takes a map can have the default wave type overridden. For example:

```
pigment { wood color_map { MyMap } ramp_wave }
```

Also available are `sine_wave`, `scallop_wave`, `cubic_wave` and `poly_wave` types. These types are of most use in normal patterns as a type of built-in slope map. The `sine_wave` takes the zig-zag of a ramp wave and turns it into a gentle rolling wave with smooth transitions. The `scallop_wave` uses the absolute value of the sine wave which looks like corduroy when scaled small or like a stack of cylinders when scaled larger. The `cubic_wave` is a gentle cubic curve from 0.0 to 1.0 with zero slope at the start and end. The `poly_wave` is an exponential function. It is followed by an optional float value which specifies exponent. For example `poly_wave 2` starts low and climbs rapidly at the end while `poly_wave 0.5` climbs rapidly at first and levels off at the end. If no float value is specified, the default is 1.0 which produces a linear function identical to `ramp_wave`.

Although any of these wave types can be used for pigments, normals, textures, or density the effect of many of the wave types are not as noticeable on pigments, textures, or density as they are for normals.

Wave type modifiers have no effect on block patterns `checker`, `brick`, `object` and `hexagon` nor do they effect `image_map`, `bump_map` or `material_map`. They also have no effect in normal statements when used with `bumps`, `dents`, `quilted`, `ripples`, `waves`, or `wrinkles` because these normal patterns cannot use `normal_map` or `slope_map`.

4.12.4 Noise Generators

There are three noise generators implemented. Changing the `noise_generator` will change the appearance of noise based patterns, like bozo and granite.

- `noise_generator 1` the noise that was used in POV_Ray 3.1
- `noise_generator 2` 'range corrected' version of the old noise, it does not show the plateaus seen with `noise_generator 1`

- noise_generator 3 generates Perlin noise

The default is noise_generator 2

Note: The noise_generator can also be set in global_settings

4.12.5 Turbulence

The turbulence pattern modifier is still supported for compatibility issues, but it's better nowadays to use the warp {turbulence} feature, which doesn't have turbulence's limitation in transformation order (turbulence is always applied first, before any scale, translate or rotate, whatever the order you specify). For a detailed discussion see 'Turbulence versus Turbulence Warp'

The old-style turbulence is handled slightly differently when used with the agate, marble, spiral1, spiral2, and wood textures.

4.12.6 Warps

The warp statement is a pattern modifier that is similar to turbulence. Turbulence works by taking the pattern evaluation point and pushing it about in a series of random steps. However warps push the point in very well-defined, non-random, geometric ways. The warp statement also overcomes some limitations of traditional turbulence and transformations by giving the user more control over the order in which turbulence, transformation and warp modifiers are applied to the pattern.

Currently there are seven types of warps but the syntax was designed to allow future expansion. The turbulence warp provides an alternative way to specify turbulence. The others modify the pattern in geometric ways.

The syntax for using a warp statement is:

```
WARP:
    warp { WARP_ITEM }
WARP_ITEM:
    repeat <Direction> [REPEAT_ITEMS...] |
    black_hole <Location>, Radius [BLACK_HOLE_ITEMS...] |
    turbulence <Amount> [TURB_ITEMS...]
    cylindrical  [ orientation VECTOR | dist_exp FLOAT ]
    spherical  [ orientation VECTOR | dist_exp FLOAT ]
    toroidal  [ orientation VECTOR | dist_exp FLOAT |
                major_radius FLOAT ]
    planar [ VECTOR , FLOAT ]
REPEAT_ITEMS:
    offset <Amount> |
    flip <Axis>
BLACK_HOLE_ITEMS:
    strength Strength | falloff Amount | inverse |
    repeat <Repeat> | turbulence <Amount>
TURB_ITEMS:
    octaves Count | omega Amount | lambda Amount
```

You may have as many separate warp statements as you like in each pattern. The placement of warp statements relative to other modifiers such as `color_map` or `turbulence` is not important. However placement of warp statements relative to each other and to transformations is significant. Multiple warps and transformations are evaluated in the order in which you specify them. For example if you translate, then warp or warp, then translate, the results can be different.

Black Hole Warp

A `black_hole` warp is so named because of its similarity to real black holes. Just like the real thing, you cannot actually see a black hole. The only way to detect its presence is by the effect it has on things that surround it.

Take, for example, a wood grain. Using POV-Ray's normal turbulence and other texture modifier functions, you can get a nice, random appearance to the grain. But in its randomness it is regular - it is regularly random! Adding a black hole allows you to create a localized disturbance in a wood grain in either one or multiple locations. The black hole can have the effect of either *sucking* the surrounding texture into itself (like the real thing) or *pushing* it away. In the latter case, applied to a wood grain, it would look to the viewer as if there were a knothole in the wood. In this text we use a wood grain regularly as an example, because it is ideally suitable to explaining black holes. However, black holes may in fact be used with any texture or pattern. The effect that the black hole has on the texture can be specified. By default, it *sucks* with the strength calculated exponentially (inverse-square). You can change this if you like.

Black holes may be used anywhere a warp is permitted. The syntax is:

```
BLACK_HOLE_WARP:
    warp
    {
        black_hole <Location>, Radius
        [BLACK_HOLE_ITEMS...]
    }
BLACK_HOLE_ITEMS:
    strength Strength | falloff Amount | inverse | type Type |
    repeat <Repeat> | turbulence <Amount>
```

The minimal requirement is the `black_hole` keyword followed by a vector <*Location*> followed by a comma and a float *Radius*. Black holes effect all points within the spherical region around the location and within the radius. This is optionally followed by any number of other keywords which control how the texture is warped.

The `falloff` keyword may be used with a float value to specify the power by which the effect of the black hole falls off. The default is two. The force of the black hole at any given point, before applying the `strength` modifier, is as follows.

First, convert the distance from the point to the center to a proportion (0 to 1) that the point is from the edge of the black hole. A point on the perimeter of the black hole will be 0.0; a point at the center will be 1.0; a point exactly halfway will be 0.5, and so forth. Mentally you can consider this to be a closeness factor. A closeness of 1.0 is as close as you can get to the center (i.e. at the center), a closeness of 0.0 is as far away as you can get from the center and still be inside the black hole and a closeness of 0.5 means the point is exactly halfway between the two.

Call this value c. Raise c to the power specified in `falloff`. By default Falloff is 2, so this is c^2 or c squared. The resulting value is the force of the black hole at that exact location and is used, after applying the `strength` scaling factor as described below, to determine how much the point is perturbed in space. For example, if c is 0.5 the force is 0.5^2 or 0.25. If c is 0.25 the force is 0.125. But if c is exactly 1.0 the force is 1.0. Recall that as c gets smaller the point is farther from the center of the black hole. Using the default power of 2, you can see that as c reduces, the force reduces exponentially in an inverse-square relationship. Put in plain English, it means that the force is much stronger (by a power of two) towards the center than it is at the outside.

By increasing `falloff`, you can increase the magnitude of the falloff. A large value will mean points towards the perimeter will hardly be affected at all and points towards the center will be affected strongly. A value of 1.0 for `falloff` will mean that the effect is linear. A point that is exactly halfway to the center of the black hole will be affected by a force of exactly 0.5. A value of `falloff` of less than one but greater than zero means that as you get closer to the outside, the force increases rather than decreases. This can have some uses but there is a side effect. Recall that the effect of a black hole ceases outside its perimeter. This means that points just within the perimeter will be affected strongly and those just outside not at all. This would lead to a visible border, shaped as a sphere. A value for `falloff` of 0 would mean that the force would be 1.0 for all points within the black hole, since any number larger 0 raised to the power of 0 is 1.0.

The `strength` keyword may be specified with a float value to give you a bit more control over how much a point is perturbed by the black hole. Basically, the force of the black hole (as determined above) is multiplied by the value of `strength`, which defaults to 1.0. If you set strength to 0.5, for example, all points within the black hole will be moved by only half as much as they would have been. If you set it to 2.0 they will be moved twice as much.

There is a rider to the latter example, though - the movement is clipped to a maximum of the original distance from the center. That is to say, a point that is 0.75 units from the center may only be moved by a maximum of 0.75 units either towards the center or away from it, regardless of the value of `strength`. The result of this clipping is that you will have an exclusion area near the center of the black hole where all points whose final force value exceeded or equaled 1.0 were moved by a fixed amount.

If the `inverse` keyword is specified then the points *pushed* away from the center instead of being pulled in.

The `repeat` keyword followed by a vector, allows you to simulate the effect of many black holes without having to explicitly declare them. Repeat is a vector that tells POV-Ray to use this black hole at multiple locations. Using `repeat` logically divides your scene up into cubes, the first being located at <0,0,0> and going to <*Repeat*>. Suppose your repeat vector was <1,5,2>. The first cube would be from <0,0,0> to < 1,5,2>. This cube repeats, so there would be one at < -1,-5,-2>, <1,5,2>, <2,10,4> and so forth in all directions, ad infinitum.

When you use `repeat`, the center of the black hole does not specify an absolute location in your scene but an offset into each block. It is only possible to use positive offsets. Negative values will produce undefined results.

Suppose your center was <0.5,1,0.25> and the repeat vector is <2,2,2>. This gives

us a block at < 0,0,0> and <2,2,2>, etc. The centers of the black hole's for these blocks would be <0,0,0> + < 0.5,1.0,0.25>, i. e. <0.5,1.0,0.25>, and < 2,2,2> + <0.5,1.0,0.25>, i. e. < 2,5,3.0,2.25>.

Due to the way repeats are calculated internally, there is a restriction on the values you specify for the repeat vector. Basically, each black hole must be totally enclosed within each block (or cube), with no part crossing into a neighboring one. This means that, for each of the x, y and z dimensions, the offset of the center may not be less than the radius, and the repeat value for that dimension must be >=the center plus the radius since any other values would allow the black hole to cross a boundary. Put another way, for each of x, y and z

Radius <= Offset or Center <= Repeat - Radius.

If the repeat vector in any dimension is too small to fit this criteria, it will be increased and a warning message issued. If the center is less than the radius it will also be moved but no message will be issued.

Note that none of the above should be read to mean that you can't overlap black holes. You most certainly can and in fact this can produce some most useful effects. The restriction only applies to elements of the `same` black hole which is repeating. You can declare a second black hole that also repeats and its elements can quite happily overlap the first and causing the appropriate interactions. It is legal for the repeat value for any dimension to be 0, meaning that POV-Ray will not repeat the black hole in that direction.

The `turbulence` can only be used in a black hole with `repeat`. It allows an element of randomness to be inserted into the way the black holes repeat, to cause a more natural look. A good example would be an array of knotholes in wood - it would look rather artificial if each knothole were an exact distance from the previous.

The `turbulence` vector is a measurement that is added to each individual black hole in an array, after each axis of the vector is multiplied by a different random amount ranging from 0 to 1. The resulting actual position of the black hole's center for that particular repeat element is random (but consistent, so renders will be repeatable) and somewhere within the above coordinates. There is a rider on the use of turbulence, which basically is the same as that of the repeat vector. You can't specify a value which would cause a black hole to potentially cross outside of its particular block.

In summary: For each of x, y and z the offset of the center must be >=radius and the value of the repeat must be >= center + radius + turbulence. The exception being that repeat may be 0 for any dimension, which means do not repeat in that direction.

Some examples are given by

```
warp {
  black_hole <0, 0, 0>, 0.5
}
warp {
  black_hole <0.15, 0.125, 0>, 0.5
  falloff 7
  strength 1.0
  repeat <1.25, 1.25, 0>
  turbulence <0.25, 0.25, 0>
  inverse
```

```
  }
  warp {
    black_hole <0, 0, 0>, 1.0
    falloff 2
    strength 2
    inverse
  }
```

Repeat Warp

The `repeat` warp causes a section of the pattern to be repeated over and over. It takes a slice out of the pattern and makes multiple copies of it side-by-side. The warp has many uses but was originally designed to make it easy to model wood veneer textures. Veneer is made by taking very thin slices from a log and placing them side-by-side on some other backing material. You see side-by-side nearly identical ring patterns but each will be a slice perhaps 1/32th of an inch deeper.

The syntax for a repeat warp is

```
REPEAT_WARP:
    warp { repeat <Direction> [REPEAT_ITEMS...] }
REPEAT_ITEMS:
    offset <Amount> | flip <Axis>
```

The `repeat` vector specifies the direction in which the pattern repeats and the width of the repeated area. This vector must lie entirely along an axis. In other words, two of its three components must be 0. For example

```
  pigment {
    wood
    warp { repeat 2*x }
  }
```

which means that from x=0 to x=2 you get whatever the pattern usually is. But from x=2 to x=4 you get the same thing exactly shifted two units over in the x-direction. To evaluate it you simply take the x-coordinate modulo 2. Unfortunately you get exact duplicates which isn't very realistic. The optional `offset` vector tells how much to translate the pattern each time it repeats. For example

```
  pigment {
    wood
    warp {repeat x*2  offset z*0.05}
  }
```

means that we slice the first copy from x=0 to x=2 at z=0 but at x=2 to x=4 we offset to z=0.05. In the 4 to 6 interval we slice at z=0.10. At the n-th copy we slice at 0.05 n z. Thus each copy is slightly different. There are no restrictions on the offset vector.

Finally the `flip` vector causes the pattern to be flipped or mirrored every other copy of the pattern. The first copy of the pattern in the positive direction from the axis is not flipped. The next farther is, the next is not, etc. The flip vector is a three component x, y, z vector but each component is treated as a boolean value that tells if you should or should not flip along a given axis. For example

```
  pigment {
```

```
    wood
    warp {repeat 2*x  flip <1,1,0>}
  }
```

means that every other copy of the pattern will be mirrored about the x- and y- axis but not the z-axis. A non-zero value means flip and zero means do not flip about that axis. The magnitude of the values in the flip vector doesn't matter.

Turbulence versus Turbulence Warp

The POV-Ray language contains an ambiguity and limitation on the way you specify `turbulence` and transformations such as `translate`, `rotate`, `scale`, `matrix`, and `transform` transforms. Usually the turbulence is done first. Then all translate, rotate, scale, matrix, and transform operations are always done after turbulence regardless of the order in which you specify them. For example this

```
pigment {
  wood
  scale .5
  turbulence .2
}
```

works exactly the same as

```
pigment {
  wood
  turbulence .2
  scale .5
}
```

The turbulence is always first. A better example of this limitation is with uneven turbulence and rotations.

```
pigment {
  wood
  turbulence 0.5*y
  rotate z*60
}
// as compared to
pigment {
 wood
 rotate z*60
 turbulence 0.5*y
}
```

The results will be the same either way even though you'd think it should look different.

We cannot change this basic behavior in POV-Ray now because lots of scenes would potentially render differently if suddenly the order transformation vs. turbulence mattered when in the past, it didn't.

However, by specifying our turbulence inside warp statement you tell POV-Ray that the order in which turbulence, transformations and other warps are applied is significant. Here's an example of a turbulence warp.

```
warp { turbulence <0,1,1> octaves 3 lambda 1.5 omega 0.3 }
```

The significance is that this

```
pigment {
  wood
  translate <1,2,3> rotate x*45 scale 2
  warp { turbulence <0,1,1> octaves 3 lambda 1.5 omega 0.3 }
}
```

produces *different results* than this...

```
pigment {
  wood
  warp { turbulence <0,1,1> octaves 3 lambda 1.5 omega 0.3 }
  translate <1,2,3> rotate x*45 scale 2
}
```

You may specify turbulence without using a warp statement. However you cannot control the order in which they are evaluated unless you put them in a warp.

The evaluation rules are as follows:

1. First any turbulence not inside a warp statement is applied regardless of the order in which it appears relative to warps or transformations.
2. Next each warp statement, translate, rotate, scale or matrix one-by-one, is applied in the order the user specifies. If you want turbulence done in a specific order, you simply specify it inside a warp in the proper place.

Turbulence Warp

Inside the `warp` statement, the keyword `turbulence` followed by a float or vector may be used to stir up any `pigment`, `normal` or `density`. A number of optional parameters may be used with turbulence to control how it is computed. The syntax is:

```
TURBULENCE_ITEM:
    turbulence <Amount> | octaves Count |
    omega Amount | lambda Amount
```

Typical turbulence values range from the default 0.0, which is no turbulence, to 1.0 or more, which is very turbulent. If a vector is specified different amounts of turbulence are applied in the x-, y- and z-direction. For example

```
turbulence <1.0, 0.6, 0.1>
```

has much turbulence in the x-direction, a moderate amount in the y-direction and a small amount in the z-direction.

Turbulence uses a random noise function called *DNoise*. This is similar to the noise used in the `bozo` pattern except that instead of giving a single value it gives a direction. You can think of it as the direction that the wind is blowing at that spot. Points close together generate almost the same value but points far apart are randomly different.

Turbulence uses *DNoise* to push a point around in several steps called `octaves`. We locate the point we want to evaluate, then push it around a bit using turbulence to get to a different point then look up the color or pattern of the new point.

It says in effect *"Don't give me the color at this spot... take a few random steps in different directions and give me that color"*. Each step is typically half as long as the one before. For example:

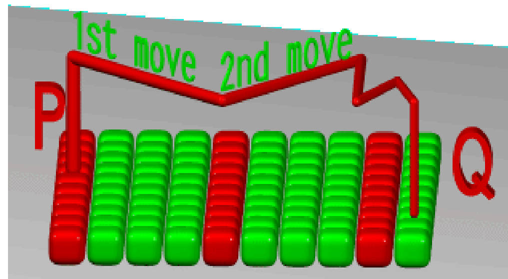

Figure 4.7: Turbulence random walk.

The magnitude of these steps is controlled by the turbulence value. There are three additional parameters which control how turbulence is computed. They are `octaves`, `lambda` and `omega`. Each is optional. Each is followed by a single float value. Each has no effect when there is no turbulence.

Octaves

The `octaves` keyword may be followed by an integer value to control the number of steps of turbulence that are computed. Legal values range from 1 to <10. The default value of 6 is a fairly high value; you won't see much change by setting it to a higher value because the extra steps are too small. Float values are truncated to integer. Smaller numbers of octaves give a gentler, wavy turbulence and computes faster. Higher octaves create more jagged or fuzzy turbulence and takes longer to compute.

Lambda

The `lambda` parameter controls how statistically different the random move of an octave is compared to its previous octave. The default value is 2.0 which is quite random. Values close to lambda 1.0 will straighten out the randomness of the path in the diagram above. The zig-zag steps in the calculation are in nearly the same direction. Higher values can look more *swirly* under some circumstances.

Omega

The `omega` value controls how large each successive octave step is compared to the previous value. Each successive octave of turbulence is multiplied by the omega value. The default `omega 0.5` means that each octave is 1/2 the size of the previous one. Higher omega values mean that 2nd, 3rd, 4th and up octaves contribute more turbulence

giving a sharper, *crinkly* look while smaller omegas give a fuzzy kind of turbulence that gets blurry in places.

Mapping using warps

Syntax:

```
CYLINDRICAL_WARP:
  warp { cylindrical [CYLINDRICAL_ITEMS...]}
CYLINDRICAL_ITEMS:
  orientation VECTOR | dist_exp FLOAT
SPHERICAL_WARP:
  warp { spherical [SPHERICAL_ITEMS...]}
SPHERICAL_ITEMS:
  orientation VECTOR | dist_exp FLOAT
TOROIDAL_WARP:
  warp { toroidal [TOROIDAL_ITEMS...]}
TOROIDAL_ITEMS:
  orientation VECTOR | dist_exp FLOAT | major_radius FLOAT
PLANAR_WARP:
  warp { planar [ VECTOR , FLOAT ]}
```

With the `cylindrical, spherical` and `toroidal` warps you can wrap checkers, bricks and other patterns around cylinders, spheres, toruses and other objects. In essence, these warps use the same mapping as the image maps use.

However it does 3D mapping and some concession had to be made on depth. This is controllable by `dist_exp` (distance exponent). In the default of 0, imagine a box <0,0> to <1,1> (actually it is <0,0>, `<dist^dist_exp,dist^dist_exp>`) stretching to infinity along the orientation vector. The warp takes its points from that box.

For a sphere `distance` is distance from origin, cylinder is distance from y-axis, torus is distance from major radius. (or distance is minor radius if you prefer to look at it that way)

Defaults: `orientation <0,0,1>`
`dist_exp 0`
`major_radius 1`

Examples:

```
torus {
  1, 0.5
  pigment {
    hexagon
    scale 0.1
    warp {
      toroidal
      orientation y
      dist_exp 1
      major_radius 1
    }
  }
```

```
}
sphere {
  0,1
  pigment {
    hexagon
    scale <0.5/pi,0.25/pi,1>*0.1
    warp {
      spherical
      orientation y
      dist_exp 1
    }
  }
}
cylinder {
  -y, y, 1
  pigment {
    hexagon
    scale <0.5/pi, 1, 1>*0.1
    warp {
      cylindrical
      orientation y
      dist_exp 1
    }
  }
}
```

The `planar` warp was made to make a pattern act like an image_map, of infinite size and can be useful in combination with other mapping-warps. By default the pigment in the XY-plane is extruded along the Z-axis. The pigment can be taken from an other plane, by specifying the optional vector (normal of the plane) and float (distance along the normal). The result, again, is extruded along the Z-axis.

4.12.7 Bitmap Modifiers

A bitmap modifier is a modifier used inside an `image_map`, `bump_map` or `material_map` to specify how the 2-D bitmap is to be applied to the 3-D surface. Several bitmap modifiers apply to specific kinds of maps and they are covered in the appropriate sections. The bitmap modifiers discussed in the following sections are applicable to all three types of bitmaps.

The once Option

Normally there are an infinite number of repeating image maps, bump maps or material maps created over every unit square of the x-y-plane like tiles. By adding the `once` keyword after a file name you can eliminate all other copies of the map except the one at (0,0) to (1,1). In image maps, areas outside this unit square are treated as fully transparent. In bump maps, areas outside this unit square are left flat with no normal modification. In material maps, areas outside this unit square are textured with the first texture of the texture list.

For example:

```
image_map {
  gif "mypic.gif"
  once
}
```

The map_type Option

The default projection of the image onto the x-y-plane is called a *planar map type*. This option may be changed by adding the `map_type` keyword followed by an integer number specifying the way to wrap the image around the object.

A `map_type 0` gives the default planar mapping already described.

A `map_type 1` gives a spherical mapping. It assumes that the object is a sphere of any size sitting at the origin. The y-axis is the north/south pole of the spherical mapping. The top and bottom edges of the image just touch the pole regardless of any scaling. The left edge of the image begins at the positive x-axis and wraps the image around the sphere from west to east in a -y-rotation. The image covers the sphere exactly once. The `once` keyword has no meaning for this mapping type.

With `map_type 2` you get a cylindrical mapping. It assumes that a cylinder of any diameter lies along the y-axis. The image wraps around the cylinder just like the spherical map but the image remains one unit tall from y=0 to y=1. This band of color is repeated at all heights unless the `once` keyword is applied.

Finally `map_type 5` is a torus or donut shaped mapping. It assumes that a torus of major radius one sits at the origin in the x-z-plane. The image is wrapped around similar to spherical or cylindrical maps. However the top and bottom edges of the map wrap over and under the torus where they meet each other on the inner rim.

Types 3 and 4 are still under development.

Note: that the `map_type` option may also be applied to `bump_map` and `material_map` statements.

For example:

```
sphere{<0,0,0>,1
  pigment{
    image_map {
      gif "world.gif"
      map_type 1
    }
  }
}
```

The interpolate Option

Adding the `interpolate` keyword can smooth the jagged look of a bitmap. When POV-Ray checks a color for an image map or a bump amount for a bump map, it often checks a point that is not directly on top of one pixel but sort of between several differently colored pixels. Interpolations return an in-between value so that the steps between the pixels in the map will look smoother.

Although interpolate is legal in material maps, the color index is interpolated before the texture is chosen. It does not interpolate the final color as you might hope it would. In general, interpolation of material maps serves no useful purpose but this may be fixed in future versions.

There are currently two types of interpolation: interpolate 2 gives bilinear interpolation while interpolate 4 gives normalized distance. For example:

```
image_map {
  gif "mypic.gif"
  interpolate 2
}
```

Default is no interpolation. Normalized distance is the slightly faster of the two, bilinear does a better job of picking the between color. Normally bilinear is used.

If your map looks jaggy, try using interpolation instead of going to a higher resolution image. The results can be very good.

Chapter 5

Interior & Media & Photons

5.1 Interior

Introduced in POV-Ray 3.1 is an object modifier statement called `interior`. The syntax is:

```
INTERIOR:
    interior { [INTERIOR_IDENTIFIER] [INTERIOR_ITEMS...] }
INTERIOR_ITEM:
    ior Value | caustics Value | dispersion Value |
    dispersion_samples Samples | fade_distance Distance |
    fade_power Power | fade_color <Color>
    MEDIA...
```

Interior default values:

```
ior                : 1.0
caustics           : 0.0
dispersion         : 1.0
dispersion_samples : 7
fade_distance      : 0.0
fade_power         : 0.0
fade_color         : <0,0,0>
```

The `interior` contains items which describe the properties of the interior of the object. This is in contrast to the `texture` and `interior_texture` which describe the surface properties only. The interior of an object is only of interest if it has a transparent texture which allows you to see inside the object. It also applies only to solid objects which have a well-defined inside/outside distinction.

Note: the `open` keyword, or `clipped_by` modifier also allows you to see inside but interior features may not render properly. They should be avoided if accurate interiors are required.

Interior identifiers may be declared to make scene files more readable and to parameterize scenes so that changing a single declaration changes many values. An identifier is declared as follows.

```
INTERIOR_DECLARATION:
    #declare IDENTIFIER = INTERIOR |
    #local IDENTIFIER = INTERIOR
```

Where *IDENTIFIER* is the name of the identifier up to 40 characters long and *INTERIOR* is any valid `interior` statement. See "#declare vs. #local" for information on identifier scope.

5.1.1 Why are Interior and Media Necessary?

In previous versions of POV-Ray, most of the items in the `interior` statement were previously part of the `finish` statement. Also the `halo` statement which was once part of the `texture` statement has been discontinued and has been replaced by the `media` statement which is part of `interior`.

You are probably asking **WHY?** As explained earlier, the `interior` contains items which describe the properties of the interior of the object. This is in contrast to the `texture` which describes the surface properties only. However this is not just a philosophical change. There were serious inconsistencies in the old model.

The main problem arises when a `texture_map` or other patterned texture is used. These features allow you to create textures that are a blend of two textures and which vary the entire texture from one point to another. It does its blending by fully evaluating the apparent color as though only one texture was applied and then fully reevaluating it with the other texture. The two final results are blended.

It is totally illogical to have a ray enter an object with one index or refraction and then recalculate with another index. The result is not an average of the two ior values. Similarly it makes no sense to have a ray enter at one ior and exit at a different ior without transitioning between them along the way. POV-Ray only calculates refraction as the ray enters or leaves. It cannot incrementally compute a changing ior through the interior of an object. Real world objects such as optical fibers or no-line bifocal eyeglasses can have variable iors but POV-Ray cannot simulate them.

Similarly the `halo` calculations were not performed as the syntax implied. Using a `halo` in such multi-textured objects did not vary the `halo` through the interior of the object. Rather, it computed two separate halos through the whole object and averaged the results. The new design for `media` which replaces `halo` makes it possible to have media that varies throughout the interior of the object according to a pattern but it does so independently of the surface texture. Because there are other changes in the design of this feature which make it significantly different, it was not only moved to the `interior` but the name was changed.

During our development, someone asked if we will create patterned interiors or a hypothetical `interior_map` feature. We will not. That would defeat the whole purpose of moving these features in the first place. They cannot be patterned and have logical or self-consistent results.

5.1.2 Empty and Solid Objects

It is very important that you know the basic concept behind empty and solid objects in POV-Ray to fully understand how features like interior and translucency are used. Objects in POV-Ray can either be solid, empty or filled with (small) particles.

A solid object is made from the material specified by its pigment and finish statements (and to some degree its normal statement). By default all objects are assumed to be solid. If you assign a stone texture to a sphere you'll get a ball made completely of stone. It's like you had cut this ball from a block of stone. A glass ball is a massive sphere made of glass. You should be aware that solid objects are conceptual things. If you clip away parts of the sphere you'll clearly see that the interior is empty and it just has a very thin surface.

This is not contrary to the concept of a solid object used in POV-Ray. It is assumed that all space inside the sphere is covered by the sphere's `interior`. Light passing through the object is affected by attenuation and refraction properties. However there is no room for any other particles like those used by fog or interior media.

Empty objects are created by adding the `hollow` keyword (see "Hollow") to the object statement. An empty (or hollow) object is assumed to be made of a very thin surface which is of the material specified by the pigment, finish and normal statements. The object's interior is empty, it normally contains air molecules.

An empty object can be filled with particles by adding fog or atmospheric media to the scene or by adding an interior media to the object. It is very important to understand that in order to fill an object with any kind of particles it first has to be made hollow.

There is a pitfall in the empty/solid object implementation that you have to be aware of.

In order to be able to put solid objects inside a media or fog, a test has to be made for every ray that passes through the media. If this ray travels through a solid object the media will not be calculated. This is what anyone will expect. A solid glass sphere in a fog bank does not contain fog.

The problem arises when the camera ray is inside any non-hollow object. In this case the ray is already traveling through a solid object and even if the media's container object is hit and it is hollow, the media will not be calculated. There is no way of telling between these two cases.

POV-Ray has to determine whether the camera is inside any object prior to tracing a camera ray in order to be able to correctly render medias when the camera is inside the container object. There's no way around doing this.

The solution to this problem (that will often happen with infinite objects like planes) is to make those objects hollow too. Thus the ray will travel through a hollow object, will hit the container object and the media will be calculated.

5.1.3 Scaling objects with an interior

All the statements that can be put in an interior represent aspects of the matter that an object is made of. Scaling an object, changing its size, doesn't change its matter.

Two pieces of the same quality steel, one twice as big as the other, both have the same density. The bigger piece is quite a bit heavier though.

So, in POV-Ray, if you design a lens from a glass with an ior of 1.5 and you scale it bigger, the focal distance of the lens will get longer as the ior stays the same. For light attenuation it means that an object will be "darker" after being scaled up. The light intensity decreases a certain amount per pov-unit. The object has become bigger, more pov-units, so more light is faded. The `fade_distance, fade_power` themselves have not been changed.

The same applies to media. Imagine media as a density of particles, you specify 100 particles per cubic pov-unit. If we scale a 1 cubic pov-unit object to be twice as big in every direction, we will have a total of 800 particles in the object. The object will look different, as we have more particles to look through. Yet the objects density is still 100 particles per cubic pov-unit. In media this "particle density" is set by the color after `emission`, `absorption`, or in the `scattering` statement

```
#version 3.5;
global_settings {assumed_gamma 1.0}
camera {location <0, 0,-12.0> look_at 0 angle 30 }
#declare Container_T= texture {
   pigment {rgbt <1,1,1,1>}
   finish {ambient 0 diffuse 0}
}

#declare Scale=2;

box {                  //The reference
   <-1,-1,0>,<1,1,.3>
   hollow
   texture {Container_T}
   interior {
      media {
         intervals 1
         samples 1,1
         emission 1
      }
   }
   translate <-2.1,0,0>
}

box {                  //Object scaled twice as big
   <-1,-1,0>,<1,1,.3>  //looks different but same
   hollow                          //particle density
   texture {Container_T}
   interior {
      media {
         intervals 1
         samples 1,1
         emission 1
      }
   }
   scale Scale
   translate<0,0,12>
```

```
}

box {                   //Object scaled twice as big
   <-1,-1,0>,<1,1,.3>  //looks the same but particle
   hollow                           //density scaled down
   texture {Container_T}
   interior {
      media {
         intervals 1
         samples 1,1
         emission 1/Scale
      }
   }
   scale Scale
   translate<0,0,12>
   translate<4.2,0,0>
}
```

The third object in the scene above, shows what to do, if you want to scale the object *and* want it to keep the same look as before. The interior feature has to be divided by the same amount, that the object was scaled by. This is only possible when the object is scaled uniform.

In general, the correct approach is to scale the media density proportionally to the change in container volume. For non-uniform scaling to get an unambiguous result, that can be explained in physical terms, we need to do:

```
Density*sqrt(3)/vlength(Scale)
```

where Density is your original media density and Scale is the scaling vector applied to the container.

Note: the density modifiers inside the `density{}` statement are scaled along with the object.

5.1.4 Refraction

When light passes through a surface either into or out of a dense medium the path of the ray of light is bent. Such bending is called *refraction*. The amount of bending or refracting of light depends upon the density of the material. Air, water, crystal and diamonds all have different densities and thus refract differently. The *index of refraction* or *ior* value is used by scientists to describe the relative density of substances. The `ior` keyword is used in POV-Ray in the `interior` to turn on refraction and to specify the ior value. For example:

```
object { MyObject pigment {Clear } interior { ior 1.5 } }
```

The default ior value of 1.0 will give no refraction. The index of refraction for air is 1.0, water is 1.33, glass is 1.5 and diamond is 2.4.

Normally transparent or semi-transparent surfaces in POV-Ray do not refract light. Earlier versions of POV-Ray required you to use the `refraction` keyword in the `finish` statement to turn on refraction. This is no longer necessary. Any non-zero `ior` value now turns refraction on.

In addition to turning refraction on or off, the old `refraction` keyword was followed by a float value from 0.0 to 1.0. Values in between 0.0 and 1.0 would darken the refracted light in ways that do not correspond to any physical property. Many POV-Ray scenes were created with intermediate refraction values before this bug was discovered so the feature has been maintained. A more appropriate way to reduce the brightness of refracted light is to change the `filter` or `transmit` value in the colors specified in the pigment statement or to use the `fade_power` and `fade_distance` keywords. See "Attenuation".

Note: neither the `ior` nor `refraction` keywords cause the object to be transparent. Transparency only occurs if there is a non-zero `filter` or `transmit` value in the color.

The `refraction` and `ior` keywords were originally specified in `finish` but are now properly specified in `interior`. They are accepted in `finish` for backward compatibility and generate a warning message.

5.1.5 Dispersion

For all materials with a ior different from 1.0 the refractive index isn't constant throughout the spectrum. It changes as a function of wavelength. Generally the refractive index decreases as the wavelength increases. Therefore light passing through a material will be separated according to wavelength. This is known as chromatic dispersion.

By default POV-Ray does not calculate dispersion as light travels through a transparent object. In order to get a more realistic effect the `dispersion` and `dispersion_samples` keywords can be added to the `interior{}` block. They will simulate dispersion by creating a prismatic color effect in the object.

The `dispersion` value is the ratio of refractive indices for violet to red. It controls the strength of dispersion (how much the colors are spread out) used. A DISPERSION_VALUE of 1 will give no dispersion, good values are 1.01 to 1.1.

Note: there will be no dispersion, unless the `ior` keyword has been specified in `interior{ }`. An ior of 1 is legal. The ior has no influence on the dispersion strength, only on the angle of refraction.

As POV-Ray does not use wavelengths for raytracing, a spectrum is simulated. The `dispersion_samples` value controls the amount of color-steps and smoothness in the spectrum. The default value is 7, the minimum is 2. Values up to 100 or higher may be needed to get a very smooth result.

Dispersion & Caustics

Dispersion only affects the interior of an object and has no effect on faked caustics (See "Faked Caustics").
To see the effects of dispersion in caustics, photon mapping is needed (See the sections "Photons" and "Dispersion & Photons").

5.1.6 Attenuation

Light attenuation is used to model the decrease in light intensity as the light travels through a transparent object. The keywords `fade_power`, `fade_distance` and `fade_color` are specified in the `interior` statement.

The `fade_distance` value determines the distance the light has to travel to reach half intensity while the `fade_power` value determines how fast the light will fall off. `fade_color` colorizes the attenuation. For realistic effects a fade power of 1 to 2 should be used. Default values for `fade_power` and `fade_distance` is 0.0 which turns this feature off. Default for `fade_color` is `<0,0,0>`, if `fade_color` is `<1,1,1>` there is no attenuation. The actual colors give colored attenuation. `<1,0,0>` looks red, not cyan as in media.

The attenuation is calculated by a formula similar to that used for light source attenuation.

$$attenuation = \frac{1}{1 + \left(\frac{d}{fade_distance}\right)^{fade_power}}$$

Equation 5.1:

If you set fade_power in the interior of an object at 1000 or above, a realistic exponential attenuation function will be used:

```
Attenuation = exp(-depth/fade_dist)
```

The `fade_power` and `fade_distance` keywords were originally specified in `finish` but are now properly specified in `interior`. They are accepted in `finish` for backward compatibility and generate a warning message.

5.1.7 Simulated Caustics

Caustics are light effects that occur if light is reflected or refracted by specular reflective or refractive surfaces. Imagine a glass of water standing on a table. If sunlight falls onto the glass you will see spots of light on the table. Some of the spots are caused by light being reflected by the glass while some of them are caused by light being refracted by the water in the glass.

Since it is a very difficult and time-consuming process to actually calculate those effects (though it is not impossible, see the sections "Photons") POV-Ray uses a quite simple method to simulate caustics caused by refraction. The method calculates the angle between the incoming light ray and the surface normal. Where they are nearly parallel it makes the shadow brighter. Where the angle is greater, the effect is diminished. Unlike real-world caustics, the effect does not vary based on distance. This caustic effect is limited to areas that are shaded by the transparent object. You'll get no caustic effects from reflective surfaces nor in parts that are not shaded by the object.

The `caustics` *`Power`* keyword controls the effect. Values typically range from 0.0 to 1.0 or higher. Zero is the default which is no caustics. Low, non-zero values give broad hot-spots while higher values give tighter, smaller simulated focal points.

The caustics keyword was originally specified in finish but is now properly specified in interior. It is accepted in finish for backward compatibility and generates a warning message.

5.1.8 Object-Media

The interior statement may contain one or more media statements. Media is used to simulate suspended particles such as smoke, haze, or dust. Or visible gasses such as steam or fire and explosions. When used with an object interior, the effect is constrained by the object's shape. The calculations begin when the ray enters an object and ends when it leaves the object. This section only discusses media when used with object interior. The complete syntax and an explanation of all of the parameters and options for media is given in the section ”Media”.

Typically the object itself is given a fully transparent texture however media also works in partially transparent objects. The texture pattern itself does not effect the interior media except perhaps to create shadows on it. The texture pattern of an object applies only to the surface shell. Any interior media patterns are totally independent of the texture.

In previous versions of POV-Ray, this feature was called halo and was part of the texture specification along with pigment, normal, and finish. See ”Why are Interior and Media Necessary?” for an explanation of the reasons for the change.

Media may also be specified outside an object to simulate atmospheric media. There is no constraining object in this case. If you only want media effects in a particular area, you should use object media rather than only relying upon the media pattern. In general it will be faster and more accurate because it only calculates inside the constraining object. See ”Atmospheric Media” for details on unconstrained uses of media.

You may specify more than one media statement per interior statement. In that case, all of the media participate and where they overlap, they add together.

Any object which is supposed to have media effects inside it, whether those effects are object media or atmospheric media, must have the hollow on keyword applied. Otherwise the media is blocked. See ”Empty and Solid Objects” for details.

5.2 Media

The media statement is used to specify particulate matter suspended in a medium such air or water. It can be used to specify smoke, haze, fog, gas, fire, dust etc. Previous versions of POV-Ray had two incompatible systems for generating such effects. One was halo for effects enclosed in a transparent or semi-transparent object. The other was atmosphere for effects that permeated the entire scene. This duplication of systems was complex and unnecessary. Both halo and atmosphere have been eliminated. See ”Why are Interior and Media Necessary?” for further details on this change. See ”Object Media” for details on how to use media with objects. See ”Atmospheric Media” for details on using media for atmospheric effects outside of objects. This section and

the sub-sections which follow explains the details of the various `media` options which are useful for either object media or atmospheric media.

Media works by sampling the density of particles at some specified number of points along the ray's path. Sub-samples are also taken until the results reach a specified confidence level. POV-Ray provides three methods of sampling. When used in an object's `interior` statement, sampling only occurs inside the object. When used for atmospheric media, the samples run from the camera location until the ray strikes an object. Therefore for localized effects, it is best to use an enclosing object even though the density pattern might only produce results in a small area whether the media was enclosed or not.

The complete syntax for a `media` statement is as follows:

```
MEDIA:
    media { [MEDIA_IDENTIFIER] [MEDIA_ITEMS...] }
MEDIA_ITEMS:
    method Number | intervals Number | samples Min, Max |
    confidence Value  | variance Value | ratio Value |
    absorption COLOR | emission COLOR | aa_threshold Value |
    aa_level Value |
    scattering {
       Type, COLOR [ eccentricity Value ] [ extinction Value ]
    }  |
    density {
       [DENSITY_IDENTIFIER] [PATTERN_TYPE] [DENSITY_MODIFIER...]
    }   |
    TRANSFORMATIONS
DENSITY_MODIFIER:
    PATTERN_MODIFIER | DENSITY_LIST | COLOR_LIST |
    color_map { COLOR_MAP_BODY } | colour_map { COLOR_MAP_BODY } |
    density_map { DENSITY_MAP_BODY }
```

Media default values:

```
aa_level     : 4
aa_threshold : 0.1
absorption   : <0,0,0>
confidence   : 0.9
emission     : <0,0,0>
intervals    : 10
method       : 3
ratio        : 0.9
samples      : Min 1, Max 1
variance     : 1/128
SCATTERING
  COLOR        : <0,0,0>
  eccentricity : 0.0
  extinction   : 1.0
```

If a media identifier is specified, it must be the first item. All other media items may be specified in any order. All are optional. You may have multiple `density` statements in a single `media` statement. See "Multiple Density vs. Multiple Media" for details. Transformations apply only the `density` statements which have been already specified. Any `density` after a transformation is not affected. If the `media` has no `density` state-

ments and none was specified in any media identifier, then the transformation has no effect. All other media items except for `density` and transformations override default values or any previously set values for this `media` statement.

Note: some media effects depend upon light sources. However the participation of a light source depends upon the `media_interaction` and `media_attenuation` keywords. See "Atmospheric Media Interaction" and "Atmospheric Attenuation" for details.

Note: In the POV-Ray 3.1 documentation it said: "Note a strange design side-effect was discovered during testing and it was too difficult to fix. If the enclosing object uses `transmit` rather than `filter` for transparency, then the `media` casts no shadows."This is not the case anymore in POV-Ray 3.5. Whether you specify `transmit` or `filter` to create a transparent container object, the `media` will always cast a shadow. If a shadow is not desired, use the `no_shadow` keyword for the container object.

5.2.1 Media Types

There are three types of particle interaction in `media`: absorbing, emitting, and scattering. All three activities may occur in a single media. Each of these three specifications requires a color. Only the red, green, and blue components of the color are used. The filter and transmit values are ignored. For this reason it is permissible to use one float value to specify an intensity of white color. For example the following two lines are legal and produce the same results:

```
emission 0.75
emission rgb<0.75,0.75,0.75>
```

Absorption

The `absorption` keyword specifies a color of light which is absorbed when looking through the media. For example `absorption rgb<0,1,0>` blocks the green light but permits red and blue to get through. Therefore a white object behind the media will appear magenta.

The default value is `rgb<0,0,0>` which means no light is absorbed – all light passes through normally.

Emission

The `emission` keyword specifies a color of the light emitted from the particles. Although we say they "emit" light, this only means that they are visible without any illumination shining on them. They do not really emit light that is cast on to nearby objects. This is similar to an object with high `ambient` values. The default value is `rgb<0,0,0>` which means no light is emitted.

Scattering

The syntax of a `scattering` statement is:

```
SCATTERING:
    scattering {
        Type, COLOR [ eccentricity Value ] [ extinction Value ]
    }
```

The first float value specifies the type of scattering. This is followed by the color of the scattered light. The default value if no `scattering` statement is given is `rgb<0,0,0>` which means no scattering occurs.

The scattering effect is only visible when light is shining on the media from a light source. This is similar to `diffuse` reflection off of an object. In addition to reflecting light, a scattering media also absorbs light like an `absorption` media. The balance between how much absorption occurs for a given amount of scattering is controlled by the optional `extinction` keyword and a single float value. The default value of 1.0 gives an extinction effect that matches the scattering. Values such as `extinction 0.25` give 25% the normal amount. Using `extinction 0.0` turns it off completely. Any value other than the 1.0 default is contrary to the real physical model but decreasing extinction can give you more artistic flexibility.

The integer value *`Type`* specifies one of five different scattering phase functions representing the different models: isotropic, Mie (haze and murky atmosphere), Rayleigh, and Henyey-Greenstein.

Type 1, *isotropic scattering* is the simplest form of scattering because it is independent of direction. The amount of light scattered by particles in the atmosphere does not depend on the angle between the viewing direction and the incoming light.

Types 2 and 3 are *Mie haze* and *Mie murky* scattering which are used for relatively small particles such as minuscule water droplets of fog, cloud particles, and particles responsible for the polluted sky. In this model the scattering is extremely directional in the forward direction i.e. the amount of scattered light is largest when the incident light is anti-parallel to the viewing direction (the light goes directly to the viewer). It is smallest when the incident light is parallel to the viewing direction. The haze and murky atmosphere models differ in their scattering characteristics. The murky model is much more directional than the haze model.

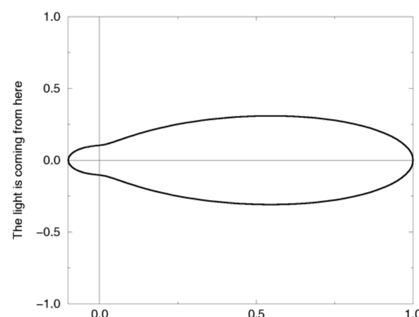

Figure 5.1: The Mie

Type 4 *Rayleigh scattering* models the scattering for extremely small particles such as molecules of the air. The amount of scattered light depends on the incident light angle.

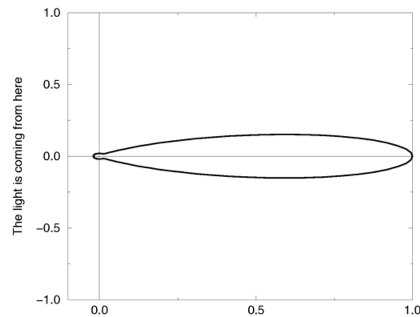

Figure 5.2: The Mie

It is largest when the incident light is parallel or anti-parallel to the viewing direction and smallest when the incident light is perpendicular to the viewing direction. You should note that the Rayleigh model used in POV-Ray does not take the dependency of scattering on the wavelength into account.

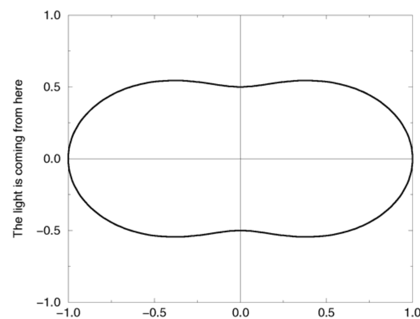

Figure 5.3: The Rayleigh scattering function.

Type 5 is the *Henyey-Greenstein scattering* model. It is based on an analytical function and can be used to model a large variety of different scattering types. The function models an ellipse with a given eccentricity e. This eccentricity is specified by the optional keyword `eccentricity` which is only used for scattering type five. The default eccentricity value of zero defines isotropic scattering while positive values lead to scattering in the direction of the light and negative values lead to scattering in the opposite direction of the light. Larger values of e (or smaller values in the negative case) increase the directional property of the scattering.

5.2.2 Sampling Parameters & Methods

Media effects are calculated by sampling the media along the path of the ray. It uses a method called *Monte Carlo integration.* The `intervals` keyword may be used to specify the integer number of intervals used to sample the ray. The default number

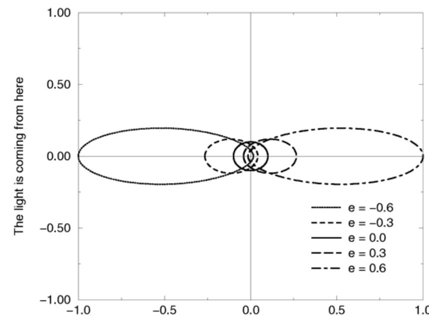

Figure 5.4: The Henyey-Greenstein scattering function for different eccentricity values.

of intervals is 10. For object media the intervals are spread between the entry and exit points as the ray passes through the container object. For atmospheric media, the intervals spans the entire length of the ray from its start until it hits an object. For media types which interact with spotlights or cylinder lights, the intervals which are not illuminated by these light types are weighted differently than the illuminated intervals when distributing samples.

The `ratio` keyword distributes intervals differently between lit and unlit areas. The default value of `ratio 0.9` means that lit intervals get more samples than unlit intervals. Note that the total number of intervals must exceed the number of illuminated intervals. If a ray passes in and out of 8 spotlights but you've only specified 5 intervals then an error occurs.

The `samples` *`Min`*, *`Max`* keyword specifies the minimum and maximum number of samples taken per interval. The default values are `samples 1,1`.

As each interval is sampled, the variance is computed. If the variance is below a threshold value, then no more samples are needed. The `variance` and `confidence` keywords specify the permitted variance allowed and the confidence that you are within that variance. The exact calculations are quite complex and involve chi-squared tests and other statistical principles too messy to describe here. The default values are `variance 1.0/128` and `confidence 0.9`. For slower more accurate results, decrease the variance and increase the confidence.

Note: the maximum number of samples limits the calculations even if the proper variance and confidence are never reached.

The `method` keyword lets you specify what sampling method is used, POV-Ray provides three. `Method 1` is the method described above.

Sample `method 2` distributes samples evenly along the viewing ray or light ray. The latter can make things look smoother sometimes. If you specify a max samples higher than the minimum samples, POV will take additional samples, but they will be random, just like in method 1. Therefore, it is suggested you set the max samples equal to the minimum samples. `jitter` will cause method 2 to look similar to method 1. It should be followed by a float, and a value of 1 will stagger the samples in the full range between samples.

Sample method 3 uses adaptive sampling (similar to adaptive anti-aliasing) which is very much like the sampling method used in POV-Ray 3.0's atmosphere. This code was written from the ground-up to work with media, however. Adaptive sampling works by taking another sample between two existing samples if there is too much variance in the original two samples. This leads to fewer samples being taken in areas where the effect from the media remains constant. The adaptive sampling is only performed if the minimum samples are set to 3 or more.

You can specify the anti-aliasing recursion depth using the aa_level keyword followed by an integer. You can specify the anti-aliasing threshold by using the aa_threshold followed by a float. The default for aa_level is 4 and the default aa_threshold is 0.1. jitter also works with method 3. Sample method 3 ignores the maximum samples value. It's usually best to only use one interval with method 3. Too many intervals can lead to artefacts, and POV will create more intervals if it needs them.

5.2.3 Density

Particles of media are normally distributed in constant density throughout the media. However the density statement allows you to vary the density across space using any of POV-Ray's pattern functions such as those used in textures. If no density statement is given then the density remains a constant value of 1.0 throughout the media. More than one density may be specified per media statement. See "Multiple Density vs. Multiple Media". The syntax for density is:

```
DENSITY:
    density
    {
        [DENSITY_IDENTIFIER]
        [DENSITY_TYPE]
        [DENSITY_MODIFIER...]
    }
DENSITY_TYPE:
    PATTERN_TYPE | COLOR
DENSITY_MODIFIER:
    PATTERN_MODIFIER | DENSITY_LIST | color_map { COLOR_MAP_BODY } |
    colour_map { COLOR_MAP_BODY } | density_map { DENSITY_MAP_BODY }
```

The density statement may begin with an optional density identifier. All subsequent values modify the defaults or the values in the identifier. The next item is a pattern type. This is any one of POV-Ray's pattern functions such as bozo, wood, gradient, waves, etc. Of particular usefulness are the spherical, planar, cylindrical, and boxed patterns which were previously available only for use with our discontinued halo feature. All patterns return a value from 0.0 to 1.0. This value is interpreted as the density of the media at that particular point. See "Patterns" for details on particular pattern types. Although a solid *COLOR* pattern is legal, in general it is used only when the density statement is inside a density_map.

General Density Modifiers

A `density` statement may be modified by any of the general pattern modifiers such as transformations, `turbulence` and `warp`. See "Pattern Modifiers" for details. In addition there are several density-specific modifiers which can be used.

Density with color_map

Typically a `media` uses just one constant color throughout. Even if you vary the density, it is usually just one color which is specified by the `absorption`, `emission`, or `scattering` keywords. However when using `emission` to simulate fire or explosions, the center of the flame (high density area) is typically brighter and white or yellow. The outer edge of the flame (less density) fades to orange, red, or in some cases deep blue. To model the density-dependent change in color which is visible, you may specify a `color_map`. The pattern function returns a value from 0.0 to 1.0 and the value is passed to the color map to compute what color or blend of colors is used. See "Color Maps" for details on how pattern values work with `color_map`. This resulting color is multiplied by the `absorption`, `emission` and `scattering` color. Currently there is no way to specify different color maps for each media type within the same `media` statement.

Consider this example:

```
media{
  emission 0.75
  scattering {1, 0.5}
  density { spherical
    color_map {
      [0.0 rgb <0,0,0.5>]
      [0.5 rgb <0.8, 0.8, 0.4>]
      [1.0 rgb <1,1,1>]
    }
  }
}
```

The color map ranges from white at density 1.0 to bright yellow at density 0.5 to deep blue at density 0. Assume we sample a point at density 0.5. The emission is 0.75*<0.8,0.8,0.4> or <0.6,0.6,0.3>. Similarly the scattering color is 0.5*<0.8,0.8,0.4> or <0.4,0.4,0.2>.

For block pattern types `checker`, `hexagon`, and `brick` you may specify a color list such as this:

```
density{
  checker
  density {rgb<1,0,0>}
  density {rgb<0,0,0>}
}
```

See "Color List Pigments" which describes how `pigment` uses a color list. The same principles apply when using them with `density`.

Density Maps and Density Lists

In addition to specifying blended colors with a color map you may create a blend of densities using a `density_map`. The syntax for a density map is identical to a color map except you specify a density in each map entry (and not a color).

The syntax for `density_map` is as follows:

```
DENSITY_MAP:
    density_map { DENSITY_MAP_BODY }
DENSITY_MAP_BODY:
    DENSITY_MAP_IDENTIFIER | DENSITY_MAP_ENTRY...
DENSITY_MAP_ENTRY:
    [ Value DENSITY_BODY ]
```

Where *`Value`* is a float value between 0.0 and 1.0 inclusive and each *DENSITY_BODY* is anything which can be inside a `density{...}` statement. The `density` keyword and `{}` braces need not be specified.

Note: the `[]` brackets are part of the actual *DENSITY_MAP_ENTRY*. They are not notational symbols denoting optional parts. The brackets surround each entry in the density map.

There may be from 2 to 256 entries in the map.

Density maps may be nested to any level of complexity you desire. The densities in a map may have color maps or density maps or any type of density you want.

Entire densities may also be used with the block patterns such as `checker`, `hexagon` and `brick`. For example...

```
  density {
    checker
    density { Flame scale .8 }
    density { Fire scale .5 }
  }
```

Note: in the case of block patterns the `density` wrapping is required around the density information.

A density map is also used with the `average` density type. See ”Average” for details.

You may declare and use density map identifiers but the only way to declare a density block pattern list is to declare a density identifier for the entire density.

Multiple Density vs. Multiple Media

It is possible to have more than one `media` specified per object and it is legal to have more than one `density` per `media`. The effects are quite different. Consider this example:

```
  object {
    MyObject
    pigment { rgbf 1 }
    interior {
      media {
```

```
      density { Some_Density }
      density { Another_Density }
    }
  }
}
```

As the media is sampled, calculations are performed for each density pattern at each sample point. The resulting samples are multiplied together. Suppose one density returned `rgb<.8,.8,.4>` and the other returned `rgb<.25,.25,0>`. The resulting color is `rgb<.2,.2,0>`.

Note: in areas where one density returns zero, it will wipe out the other density. The end result is that only density areas which overlap will be visible. This is similar to a CSG intersection operation. Now consider

```
object {
  MyObject
  pigment { rgbf 1 }
  interior {
    media {
      density { Some_Density }
    }
    media {
      density { Another_Density }
    }
  }
}
```

In this case each media is computed independently. The resulting colors are added together. Suppose one density and media returned `rgb<.8,.8,.4>` and the other returned `rgb<.25,.25,0>`. The resulting color is `rgb<1.05,1.05,.4>`. The end result is that density areas which overlap will be especially bright and all areas will be visible. This is similar to a CSG union operation. See the sample scene `scenes\interior\media\media4.pov` for an example which illustrates this.

5.3 Photons

5.3.1 Overview

The basic goal of this implementation of the photon map is to render true reflective and refractive caustics. The photon map was first introduced by Henrik Wann Jensen (see Suggested Reading).

Photon mapping is a technique which uses a forward ray-tracing pre-processing step to render refractive and reflective caustics realistically. This means that mirrors can reflect light rays and lenses can focus light.

Photon mapping works by shooting packets of light (photons) from light sources into the scene. The photons are directed towards specific objects. When a photon hits an object after passing through (or bouncing off of) the target object, the ray intersection is stored in memory. This data is later used to estimate the amount of light contributed by reflective and refractive caustics.

Examples

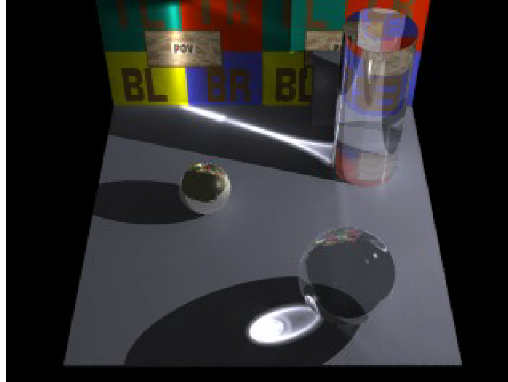

Figure 5.5: Reflective caustics

This image shows refractive caustics from a sphere and a cylinder. Both use an index of refraction of `1.2`. Also visible is a small amount of reflective caustics from the metal sphere, and also from the clear cylinder and sphere.

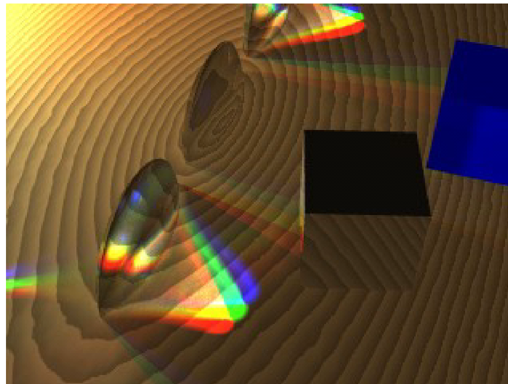

Figure 5.6: Photons used for lenses and caustics

Here we have three lenses and three light sources. The middle lens has photon mapping turned off. You can also see some reflective caustics from the brass box (some light reflects and hits the blue box, other light bounces through the nearest lens and is focused in the lower left corner of the image).

5.3.2 Using Photon Mapping in Your Scene

When designing a scene with photons, it helps to think of the scene objects in two categories. Objects in the first category will show photon caustics when hit by photons. Objects in the second category cause photon caustics by reflecting or refracting photons. Some objects may be in both categories, and some objects may be in neither category.

Category 1 - Objects that show photon caustics

By default, all objects are in the first category. Whenever a photon hits an object, the photon is stored and will later be used to render caustics on that object. This means that, by default, caustics from photons can appear on any surface. To speed up rendering, you can take objects out of this category. You do this with the line: `photons{collect off}`. If you use this syntax, caustics from photons will not appear on the object. This will save both memory and computational time during rendering.

Category 2 - Objects that cause photon caustics

By default, there are no objects in the second category. If you want your object to cause caustics, you need to do two things. First, make your object into a "target." You do this with the `target` keyword. This enables light sources to shoot photons at your object. Second, you need to specify if your object reflects photons, refracts photons, or both. This is done with the `reflection on` and `refraction on` keywords. To allow an object to reflect and refract photons, you would use the following lines of code inside the object:

```
photons{
  target
  reflection on
  refraction on
}
```

Generally speaking, you don't want an object to be in both categories. Most objects that cause photon caustics do not themselves have much color or brightness. Usually they simply refract or reflect their surroundings. For this reason, it is usually a waste of time to display photon caustics on such surfaces. Even if computed, the effects from the caustics would be so dim that they would go unnoticed.

Sometimes, you may also wish to add `photons{collect off}` to other clear or reflective objects, even if they are not photon targets. Again, this is done to prevent unnecessary computation of caustic lighting.

Finally, you may wish to enable photon reflection and refraction for a surface, even if it is not a target. This allows indirect photons (photons that have already hit a target and been reflected or refracted) to continue their journey after hitting this object.

Photon Global Settings

```
global_photon_block:
photons {
  spacing <photon_spacing> | count <photons_to_shoot>

  [gather <min_gather>, <max_gather>]
  [media <max_steps> [,<factor>]]
  [jitter <jitter_amount>]
  [max_trace_level <photon_trace_level>]
  [adc_bailout <photon_adc_bailout>]
  [save_file "filename" | load_file "filename"]
  [autostop <autostop_fraction>]
  [expand_thresholds <percent_increase>, <expand_min>]
  [radius <gather_radius>,<multiplier>,
          <gather_radius_media>,<multiplier>]
```

```
}
```

All photons default values:

```
Global :
  expand_min    : 40
  gather        : 20, 100
  jitter        : 0.4
  media         : 0

Object :
  collect       : on
  refraction    : off
  reflection    : off
  split_union   : on
  target        : 1.0

Light_source:
  area_light    : off
  refraction    : off
  reflection    : off
```

To specify photon gathering and storage options you need to add a photons block to the global_settings section of your scene.

For example:

```
global_settings {
  photons {
    count 20000
    autostop 0
    jitter .4
  }
}
```

The number of photons generated can be set using either the spacing or count keywords:

- If spacing is used, it specifies approximately the average distance between photons on surfaces. If you cut the spacing in half, you will get four times as many surface photons, and eight times as many media photons.
- If count is used, POV-Ray will shoot the approximately number of photons specified. The actual number of photons that result from this will almost always be at least slightly different from the number specified. Still, if you double the photons_to_shoot value, then twice as many photons will be shot. If you cut the value in half, then half the number of photons will be shot.
  - It may be less, because POV shoots photons at a target object's bounding box, which means that some photons will miss the target object.
  - On the other hand, may be more, because each time one object hits an object that has both reflection and refraction, two photons are created (one for reflection and one for refraction).

- POV will attempt to compensate for these two factors, but it can only estimate how many photons will actually be generated. Sometimes this estimation is rather poor, but the feature is still usable.

The keyword `gather` allows you to specify how many photons are gathered at each point during the regular rendering step. The first number (default 20) is the minimum number to gather, while the second number (default 100) is the maximum number to gather. These are good values and you should only use different ones if you know what you’re doing.

The keyword `media` turns on media photons. The parameter `max_steps` specifies the maximum number of photons to deposit over an interval. The optional parameter factor specifies the difference in media spacing compared to surface spacing. You can increase factor and decrease max_steps if too many photons are being deposited in media.

The keyword `jitter` specifies the amount of jitter used in the sampling of light rays in the pre-processing step. The default value is good and usually does not need to be changed.

The keywords `max_trace_level` and `adc_bailout` allow you to specify these attributes for the photon-tracing step. If you do not specify these, the values for the primary ray-tracing step will be used.

The keywords `save_file` and `load_file` allow you to save and load photon maps. If you load a photon map, no photons will be shot. The photon map file contains all surface (caustic) and media photons.

`radius` is used for gathering photons. The larger the radius, the longer it takes to gather photons. But if you use too small of a radius, you might not get enough photons to get a good estimate. Therefore, choosing a good radius is important. Normally POV-Ray looks through the photon map and uses some ad-hoc statistical analysis to determine a reasonable radius. Sometimes it does a good job, sometimes it does not. The radius keyword lets you override or adjust POV-Ray's guess.

`radius` parameters (all are optional):

1. Manually set the gather radius for surface photons. If this is either zero or if you leave it out, POV-Ray will analyze and guess.
2. Adjust the radius for surface photons by setting a multiplier. If POV-Ray, for example, is picking a radius that you think is too big (render is too slow), you can use "`radius ,0.5`" to lower the radius (multiply by 0.5) and speed up the render at the cost of quality.
3. Manually set the gather radius for media photons.
4. Adjust the radius for media photons by setting a multiplier.

The keywords `autostop` and `expand_thresholds` will be explained later.

Shooting Photons at an Object

```
object_photon_block:
photons {
  [target [<spacing_multiplier>]]
```

```
  [refraction on|off]
  [reflection on|off]
  [collect on|off]
  [pass_through]
}
```

To shoot photons at an object, you need to tell POV that the object receives photons. To do this, create a `photons { }` block within the object. For example:

```
object {
  MyObject
  photons {
    target
    refraction on
    reflection on
    collect off
  }
}
```

In this example, the object both reflects and refracts photons. Either of these options could be turned off (by specifying reflection off, for example). By using this, you can have an object with a reflective finish which does not reflect photons for speed and memory reasons.

The keyword `target` makes this object a target.

The density of the photons can be adjusted by specifying the `spacing_multiplier`. If, for example, you specify a `spacing_multiplier` of 0.5, then the spacing for photons hitting this object will be 1/2 of the distance of the spacing for other objects.

Note: This means four times as many surface photons, and eight times as many media photons.

The keyword `collect off` causes the object to ignore photons. Photons are neither deposited nor gathered on that object.

The keyword `pass_through` causes photons to pass through the object **unaffected** on their way to a target object. Once a photon hits the target object, it will ignore the `pass_through` flag. This is basically a photon version of the `no_shadow` keyword, with the exception that media within the object will still be affected by the photons (unless that media specifies collect off). If you use the `no_shadow` keyword, the object will be tagged as `pass_through` automatically. You can then turn off `pass_through` if necessary by simply using `photons { pass_through off }`.

Note: Photons will not be shot at an object unless you specify the `target` keyword. Simply turning refraction on will not suffice.

When shooting photons at a CSG-union, it may sometimes be of advantage to use `split_union off` inside the union. POV-Ray will be forced to shoot at the whole object, instead of splitting it up and shooting photons at its compound parts.

Photons and Light Sources

```
light_photon_block:
photons {
```

```
  [refraction on | off]
  [reflection on | off]
  [area_light]
}
```

Example:

```
light_source {
  MyLight
  photons {
    refraction on
    reflection on
  }
}
```

Sometimes, you want photons to be shot from one light source and not another. In that case, you can turn photons on for an object, but specify `photons { reflection off refraction off }` in the light source’s definition. You can also turn off only reflection or only refraction for any light source.

Photons and Media

```
global_settings {
  photons {
    count 10000
    media 100
  }
}
```

Photons also interact fully with media. This means that volumetric photons are stored in scattering media. This is enabled by using the keyword media within the photons block.

To store photons in media, POV deposits photons as it steps through the media during the photon-tracing phase of the render. It will deposit these photons as it traces caustic photons, so the number of media photons is dependent on the number of caustic photons. As a light ray passes through a section of media, the photons are deposited, separated by approximately the same distance that separates surface photons.

You can specify a factor as a second optional parameter to the media keyword. If, for example, factor is set to 2.0, then photons will be spaced twice as far apart as they would otherwise have been spaced.

Sometimes, however, if a section of media is very large, using these settings could create a large number of photons very fast and overload memory. Therefore, following the media keyword, you must specify the maximum number of photons that are deposited for each ray that travels through each section of media. A setting of 100 should probably work in most cases.

You can put `collect off` into media to make that media ignore photons. Photons will neither be deposited nor gathered in a media that is ignoring them. Photons will also not be gathered nor deposited in non-scattering media. However, if multiple medias exist in the same space, and at least one does not ignore photons and is scattering, then

photons will be deposited in that interval and will be gathered for use with all media in that interval.

5.3.3 Photons FAQ

I made an object with IOR 1.0 and the shadows look weird.

If the borders of your shadows look odd when using photon mapping, do not be alarmed. This is an unfortunate side-effect of the method. If you increase the density of photons (by decreasing spacing and gather radius) you will notice the problem diminish. We suggest not using photons if your object does not cause much refraction (such as with a window pane or other flat piece of glass or any objects with an IOR very close to 1.0).

My scene takes forever to render.

When POV-Ray builds the photon maps, it continually displays in the status bar the number of photons that have been shot. Is POV-Ray stuck in this step and does it keep shooting lots and lots of photons?

yes

If you are shooting photons at an infinite object (like a plane), then you should expect this. Either be patient or do not shoot photons at infinite objects.

Are you shooting photons at a CSG difference? Sometimes POV-Ray does a bad job creating bounding boxes for these objects. And since photons are shot at the bounding box, you could get bad results. Try manually bounding the object. You can also try the autostop feature (try `autostop 0`). See the docs for more info on autostop.

no

Does your scene have lots of glass (or other clear objects)? Glass is slow and you need to be patient.

My scene has polka dots but renders really quickly. Why?

You should increase the number of photons (or decrease the spacing).

The photons in my scene show up only as small, bright dots. How can I fix this?

The automatic calculation of the gather radius is probably not working correctly, most likely because there are many photons not visible in your scene which are affecting the statistical analysis.

You can fix this by either reducing the number of photons that are in your scene but not visible to the camera (which confuse the auto-computation), or by specifying the initial gather radius manually by using the keyword radius. If you must manually specify a gather radius, it is usually best to also use spacing instead of count, and then set radius and spacing to a 5:1 (radius:spacing) ratio.

Adding photons slowed down my scene a lot, and I see polka dots.

This is usually caused by having both high- and low-density photons in the same scene. The low density ones cause polka dots, while the high density ones slow down the scene. It is usually best if the all photons are on the same order of magnitude for

spacing and brightness. Be careful if you are shooting photons objects close to and far from a light source. There is an optional parameter to the target keyword which allows you to adjust the spacing of photons at the target object. You may need to adjust this factor for objects very close to or surrounding the light source.

I added photons, but I don’t see any caustics.

When POV-Ray builds the photon maps, it continually displays in the status bar the number of photons that have been shot. Did it show any photons being shot?

no

Try avoiding `autostop`, or you might want to bound your object manually.

Try increasing the number of photons (or decreasing the spacing).

yes

Were any photons stored (the number after `total` in the rendering message as POV-Ray shoots photons)?

no

It is possible that the photons are not hitting the target object (because another object is between the light source and the other object).

yes

The photons may be diverging more than you expect. They are probably there, but you cannot see them since they are spread out too much

The base of my glass object is really bright.

Use `collect off` with that object.

Will area lights work with photon mapping?

Photons do work with area lights. However, normally photon mapping ignores all area light options and treats all light sources as point lights. If you would like photon mapping to use your area light options, you must specify the "area_light" keyword **within** the `photons { }` block in your light source's code. Doing this will not increase the number of photons shot by the light source, but it might cause regular patterns to show up in the rendered caustics (possibly splotchiness).

What do the stats mean?

In the stats, `photons shot` means how many light rays were shot from the light sources. `photons stored` means how many photons are deposited on surfaces in the scene. If you turn on reflection and refraction, you could get more photons stored than photons shot, since the each ray can get split into two.

5.3.4 Photon Tips

– Use `collect off` in objects that photons do not hit. Just put `photons { collect off }` in the object’s definition.

– Use `collect off` in glass objects.

– Use `autostop` unless it causes problems.

– A big tip is to make sure that all of the final densities of photons are of the same general magnitude. You do not want spots with really high density photons and another area with really low density photons. You will always have some variation (which is a good thing), but having really big differences in photon density is what causes some scenes to take many hours to render.

5.3.5 Advanced Techniques

Autostop

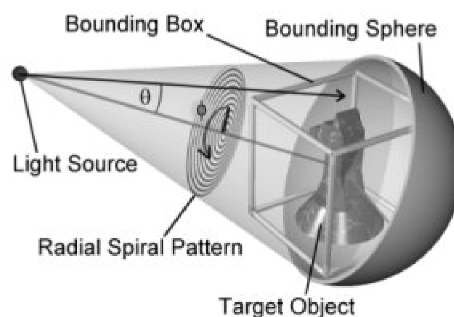

Figure 5.7: Example of the photon autostop option

To understand the `autostop` option, you need to understand the way photons are shot from light sources. Photons are shot in a spiral pattern with uniform angular density. Imagine a sphere with a spiral starting at one of the poles and spiraling out in ever-increasing circles to the equator. Two angles are involved here. The first, phi, is the how far progress has been made in the current circle of the spiral. The second, theta, is how far we are from the pole to the equator. Now, imagine this sphere centered at the light source with the pole where the spiral starts pointed towards the center of the object receiving photons. Now, photons are shot out of the light in this spiral pattern.

Normally, POV does not stop shooting photons until the target object’s entire bounding box has been thoroughly covered. Sometimes, however, an object is much smaller than its bounding box. At these times, we want to stop shooting if we do a complete circle in the spiral without hitting the object. Unfortunately, some objects (such as copper rings), have holes in the middle. Since we start shooting at the middle of the object, the photons just go through the hole in the middle, thus fooling the system into thinking that it is done. To avoid this, the `autostop` keyword lets you specify how far the system must go before this auto-stopping feature kicks in. The value specified is a fraction of the object's bounding box. Valid values are 0.0 through 1.0 (0% through 100%). POV will continue to shoot photons until the spiral has exceeded this value or the bounding box is completely covered. If a complete circle of photons fails to hit the target object after the spiral has passed the autostop threshold, POV will then stop shooting photons.

The `autostop` feature will also not kick in until at least one photon has hit the object. This allows you to use `autostop 0` even with objects that have holes in the middle.

Note: If the light source is within the object’s bounding box, the photons are shot in all directions from the light source.

Adaptive Search Radius

Unless photons are interacting with media, POV-Ray uses an adaptive search radius while gathering photons. If the minimum number of photons is not found in the original search radius, the radius is expanded and searched again. Using this adaptive search radius can both decrease the amount of time it takes to render the image, and sharpen the borders in the caustic patterns.

Sometimes this adaptive search technique can create unwanted artefacts at borders. To remove these artefacts, a few thresholds are used, which can be specified by `expand_thresholds`. For example, if expanding the radius increases the estimated density of photons by too much (threshold is percent_increase, default is 20%, or 0.2), the expanded search is discarded and the old search is used instead. However, if too few photons are gathered in the expanded search (`expand_min`, default is 40), the new search will be used always, even if it means more than a 20% increase in photon density.

Photons and Dispersion

When dispersion is specified for interior of a transparent object, photons will make use of that and show ”colored” caustics.

Saving and Loading Photon Maps

It is possible to save and load photon maps to speed up rendering. The photon map itself is view-independent, so if you want to animate a scene that contains photons and you know the photon map will not change during the animation, you can save it on the first frame and then load it for all subsequent frames.

To save the photon map, put the line

```
save_file "myfile.ph"
```

into the `photons { }` block inside the `global_settings` section.

Loading the photon map is the same, but with `load_file` instead of `save_file`. You cannot both load and save a photon map in the POV file. If you load the photon map, it will load all of the photons. No photons will be shot if the map is loaded from a file. All other options (such as gather radius) must still be specified in the POV scene file and are not loaded with the photon map.

When can you safely re-use a saved photon map?

- Moving the camera is *always* safe.
- Moving lights that do not cast photons is *always* safe.

- Moving objects that do not have photons shot at them, that do not receive photons, and would not receive photons in the new location is *always* safe.
- Moving an object that recieves photons to a new location where it does not receive photons is *sometimes* safe.
- Moving an object to a location where it recieves photons is *not* safe
- Moving an object that has photons shot at it is *not* safe
- Moving a light that casts photons is *not* safe.
- Changing the texture of an object that recieves photons is safe.
- Changing the texture of an object that has photons shot at it produces results that are not realistic, but can be useful sometimes.

Chapter 6

Include Files

The "Standard Include File" section describes the include files that can be found in every standard distribution of POV-Ray. It is supposed to be used as a reference for looking up things. It does not contain detailed explanations on how scenes are written or how POV-Ray is used. It just explains all features, their syntax, applications, limits, drawbacks, etc.

6.1 arrays.inc

This file contains macros for manipulating arrays.

`Rand_Array_Item(Array, Stream)`. Randomly Picks an item from a 1D array.
Parameters:

- `Array` = The array from which to choose the item.
- `Stream` = A random number stream.

`Resize_Array(Array, NewSize)`. Resize a 1D array, retaining its contents.
Parameters:

- `Array` = The array to be resized.
- `NewSize` = The desired new size of the array.

`Reverse_Array(Array)`. Reverses the order of items in a 1D array.
Parameters:

- `Array` = The array to be reversed.

`Sort_Compare(Array, IdxA, IdxB)`. This macro is used by the `Sort_Array()` and `Sort_Partial_Array()` macros. The given macro works for 1D arrays of floats, but you can redefine it in your scene file for more complex situations, arrays of vectors or multidimensional arrays for example. Just make sure your macro returns true if the item at IdxA < the item at IdxB, and otherwise returns false.
Parameters:

- `Array` = The array containing the data being sorted.

- `IdxA, IdxB` = The array offsets of the data elements being compared.

`Sort_Swap_Data(Array, IdxA, IdxB)`. This macro is used by the `Sort_Array()` and `Sort_Partial_Array()` macros. The given macro works for 1D arrays and floats only, but you can redefine it in your scene file for more complex situations, arrays of vectors or multidimensional arrays for example. The only requirement is that your macro swaps the data at IdxA with that at IdxB.
Parameters:

- `Array` = The array containing the data being sorted.
- `IdxA, IdxB` = The array offsets of the data elements being swapped.

`Sort_Array(Array)`. This macro sorts a 1D array of floats, though you can redefine the `Sort_Compare()` and `Sort_Swap_Data()` macros to handle multidimensional arrays and other data types.
Parameters:

- `Array` = The array to be sorted.

`Sort_Partial_Array(Array, FirstInd, LastInd)`. This macro is like `Sort_Array()`, but sorts a specific range of an array instead of the whole array.
Parameters:

- `Array` = The array to be sorted.
- `FirstInd, LastInd` = The start and end indices of the range being sorted.

6.2 chars.inc

This file includes 26 upper-case letter and other characters defined as objects. The size of all characters is 4 * 5 * 1. The center of the bottom side of a character face is set to the origin, so you may need to translate a character appropriately before rotating it about the x or z axes.

Letters:
```
char_A, char_B, char_C,
char_D, char_E, char_F,
char_G, char_H, char_I,
char_J, char_K, char_L,
char_M, char_N, char_O,
char_P, char_Q, char_R,
char_S, char_T, char_U,
char_V, char_W, char_X,
char_Y, char_Z
```

Numerals:
```
char_0, char_1,
char_2, char_3,
char_4, char_5,
char_6, char_7,
```

char_8, char_9

Symbols:
char_Dash, char_Plus, char_ExclPt,
char_Amps, char_Num, char_Dol,
char_Perc, char_Astr, char_Hat,
char_LPar, char_RPar, char_AtSign,
char_LSqu, char_RSqu

Usage:

```
#include "chars.inc"
.
.
object {char_A ...}
```

6.3 colors.inc

This file is mainly a list of predefined colors, but also has a few color manipulation macros.

6.3.1 Predefined colors

Primary colors

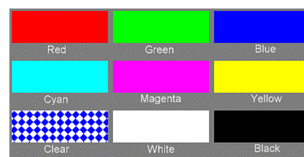

Figure 6.1: Primary Colors

Shades of gray...from 5% to 95%, in 5% increments

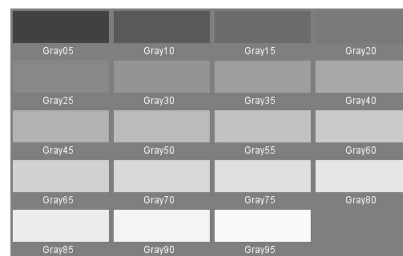

Figure 6.2: Shades of Gray

Misc. other shades of gray, with both spelling variants

Figure 6.3: Shades of Gray

Misc. colors

Figure 6.4: Misc. Colors Part 1

6.3.2 Color macros

In POV-Ray all colors are handled in RGB color space with a component for the amount of red, green and blue light. However, not everybody thinks this is the most intuitive way to specify colors. For your convenience there are macros included in colors.inc that converts between a few different types of color spaces.
The three supported color spaces:

- `RGB` = < Red, Green, Blue, Filter, Transmit >
- `HSL` = < Hue, Saturation, Lightness, Filter, Transmit >
- `HSV` = < Hue, Saturation, Value, Filter, Transmit >

`CHSL2RGB(Color)`. Converts a color given in `HSL` space to one in `RGB` space.
Parameters:

Figure 6.5: Misc. Colors Part 2

- `Color` = `HSL` color to be converted.

`CRGB2HSL(Color)`. Converts a color given in `RGB` space to one in `HSL` space.
Parameters:

- `Color` = `RGB` color to be converted.

`CHSV2RGB(Color)`. Converts a color given in `HSV` space to one in `RGB` space.
Parameters:

- `Color` = `HSV` color to be converted.

`CRGB2HSV(Color)`. Converts a color given in `RGB` space to one in `HSV` space.
Parameters:

- `Color` = `RGB` color to be converted.

`Convert_Color(SourceType, DestType, Color)`. Converts a color from one color space to another. Color spaces available are: `RGB`, `HSL`, and `HSV`.
Parameters:

- `SourceType` = Color space of input color.
- `DestType` = Desired output color space.
- `Color` = Color to be converted, in SourceType color space.

6.4 consts.inc

This file defines a number of constants, including things such as mapping types and ior definitions.

6.4.1 Vector constants

`o` = < 0, 0, 0> (origin)

`xy` = < 1, 1, 0>

`yz` = < 0, 1, 1>

`xz` = < 1, 0, 1>

6.4.2 Map type constants

`Plane_Map` = 0

`Sphere_Map` = 1

`Cylinder_Map` = 2

`Torus_Map` = 5

6.4.3 Interpolation type constants

`Bi` = 2

`Norm` = 4

6.4.4 Fog type constants

`Uniform_Fog` = 1

`Ground_Fog` = 2

6.4.5 Focal blur hexgrid constants

`Hex_Blur1` = 7

`Hex_Blur2` = 19

`Hex_Blur3` = 37

6.4.6 IORs

Air_Ior = 1.000292

Amethyst_Ior = 1.550

Apatite_Ior = 1.635

Aquamarine_Ior = 1.575

Beryl_Ior = 1.575

Citrine_Ior = 1.550

Crown_Glass_Ior = 1.51

Corundum_Ior = 1.765

Diamond_Ior = 2.47

Emerald_Ior = 1.575

Flint_Glass_Ior = 1.71

Flint_Glass_Heavy_Ior = 1.8

Flint_Glass_Medium_Ior = 1.63

Flint_Glass_Light_Ior = 1.6

Fluorite_Ior = 1.434

Gypsum_Ior = 1.525

Ice_Ior = 1.31

Plexiglas_Ior = 1.5

Quartz_Ior = 1.550

Quartz_Glass_Ior = 1.458

Ruby_Ior = 1.765

Salt_Ior = 1.544

Sapphire_Ior = 1.765

Topaz_Ior = 1.620

Tourmaline_Ior = 1.650

Water_Ior = 1.33

6.4.7 Dispersion amounts

Quartz_Glass_Dispersion = 1.012

Water_Dispersion = 1.007

Diamond_Dispersion = 1.035

Sapphire_Dispersion = 1.015

6.4.8 Scattering media type constants

ISOTROPIC_SCATTERING = 1;

MIE_HAZY_SCATTERING = 2;

MIE_MURKY_SCATTERING = 3;

RAYLEIGH_SCATTERING = 4;

HENYEY_GREENSTEIN_SCATTERING = 5;

6.5 debug.inc

This file contains a set of macros designed to make debugging easier. It also functions like the old debug.inc, with the exception that you have to call the Debug_Inc_Stack() macro to get the include stack output.

Debug_Inc_Stack(). Activates include file tracking, each included file will send a debug message when it is included.
Parameters: None.

Set_Debug(Bool). Activate or deactivate the debugging macros.
Parameters:

- Bool = A boolean (true/false) value.

Debug_Message(Str). If debugging, sends the message to the debug stream.
Parameters:

- Str = The desired message.

Debug(Condition, Message)
Warning(Condition, Message)
Error(Condition, Message)
These macros send a message to the #debug, #warning, and #error streams depending on a given condition. They are just a shortcut for an #if()...#end block, intended to make scenes easier to read.
Parameters:

- Condition = Any boolean expression.
- Message = The message to be sent if Condition evaluates as ”true”.

6.6 finish.inc

This file contains some predefined finishes.

Dull

Dull, with a large, soft specular highlight.

Shiny

Shiny, with a small, tight specular highlight.

`Glossy`

Very shiny with very tight specular highlights and a fair amount of reflection.

`Phong_Dull`

Dull, with a large, soft phong highlight.

`Phong_Shiny`

Shiny, with a small, tight phong highlight.

`Phong_Glossy`

Very shiny with very tight phong highlights and a fair amount of reflection.

`Luminous`

A glowing surface, unaffected by light_sources.

`Mirror`

A perfectly reflective surface, no highlights or shading.

6.7 functions.inc

This include file contains interfaces to internal functions as well as several predefined functions. The ID's used to access the internal functions through calls to "internal(XX)", are not guaranteed to stay the same between POV-Ray versions, so users are encouraged to use the functions declared here.

The number of required parameters and what they control are also given in the include file, this chapter gives more information.
For starter values of the parameters, check the "i_internal.pov" demo file.

Syntax to be used:

```
#include "functions.inc"
isosurface {
  function { f_torus_gumdrop(x,y,z, P0) }
  ...
}

pigment {
  function { f_cross_ellipsoids(x,y,z, P0, P1, P2, P3) }
  COLOR_MAP ...
)
```

Some special parameters are found in several of these functions. These are described in the next section and later referred to as "Cross section type", "Field Strength", "Field Limit", "SOR" parameters.

6.7.1 Common Parameters

Cross Section Type:
In the helixes and spiral functions, the 9th parameter is the cross section type.
Some shapes are:

`0` :
: square

`0.0 to 1.0` :
: rounded squares

`1` :
: circle

`1.0 to 2.0` :
: rounded diamonds

`2` :
: diamond

`2.0 to 3.0` :
: partially concave diamonds

`3` :
: concave diamond

Field Strength

The numerical value at a point in space generated by the function is multiplied by the Field Strength. The set of points where the function evaluates to zero are unaffected by any positive value of this parameter, so if you're just using the function on its own with threshold = 0, the generated surface is still the same.
In some cases, the field strength has a considerable effect on the speed and accuracy of rendering the surface. In general, increasing the field strength speeds up the rendering, but if you set the value too high the surface starts to break up and may disappear completely.
Setting the field strength to a negative value produces the inverse of the surface, like making the function negative.

Field Limit

This won't make any difference to the generated surface if you're using threshold that's within the field limit (and will kill the surface completely if the threshold is greater than the field limit). However, it may make a huge difference to the rendering times.
If you use the function to generate a pigment, then all points that are a long way from the surface will have the same color, the color that corresponds to the numerical value of the field limit.

SOR Switch

If greater than zero, the curve is swept out as a surface of revolution (SOR).
If the value is zero or negative, the curve is extruded linearly in the Z direction.

SOR Offset

If the SOR switch is on, then the curve is shifted this distance in the X direction before being swept out.

SOR Angle

If the SOR switch is on, then the curve is rotated this number of degrees about the Z axis before being swept out.

Invert Isosurface

Sometimes, when you render a surface, you may find that you get only the shape of the container. This could be caused by the fact that some of the build in functions are defined inside out.
We can invert the isosurface by negating the whole function:
`-(function) - threshold`

6.7.2 Internal Functions

Here is a list of the internal functions in the order they appear in the "functions.inc" include file

`f_algbr_cyl1(x,y,z, P0, P1, P2, P3, P4)`. An algebraic cylinder is what you get if you take any 2d curve and plot it in 3d. The 2d curve is simply extruded along the third axis, in this case the z axis.
With the SOR Switch switched on, the figure-of-eight curve will be rotated around the Y axis instead of being extruded along the Z axis.

- `P0` : Field Strength
- `P1` : Field Limit
- `P2` : SOR Switch
- `P3` : SOR Offset
- `P4` : SOR Angle

`f_algbr_cyl2(x,y,z, P0, P1, P2, P3, P4)`. An algebraic cylinder is what you get if you take any 2d curve and plot it in 3d. The 2d curve is simply extruded along the third axis, in this case the z axis.
With the SOR Switch switched on, the cross section curve will be rotated around the Y axis instead of being extruded along the Z axis.

- `P0` : Field Strength (Needs a negative field strength or a negated function)
- `P1` : Field Limit
- `P2` : SOR Switch
- `P3` : SOR Offset
- `P4` : SOR Angle

`f_algbr_cyl3(x,y,z, P0, P1, P2, P3, P4)`. An algebraic cylinder is what you get if you take any 2d curve and plot it in 3d. The 2d curve is simply extruded along the third axis, in this case the Z axis.
With the SOR Switch switched on, the cross section curve will be rotated around the Y axis instead of being extruded along the Z axis.

- `P0` : Field Strength (Needs a negative field strength or a negated function)
- `P1` : Field Limit
- `P2` : SOR Switch
- `P3` : SOR Offset
- `P4` : SOR Angle

`f_algbr_cyl4(x,y,z, P0, P1, P2, P3, P4)`. An algebraic cylinder is what you get if you take any 2d curve and plot it in 3d. The 2d curve is simply extruded along the third axis, in this case the z axis.
With the SOR Switch switched on, the cross section curve will be rotated around the Y axis instead of being extruded along the Z axis.

- `P0` : Field Strength (Needs a negative field strength or a negated function)
- `P1` : Field Limit
- `P2` : SOR Switch
- `P3` : SOR Offset
- `P4` : SOR Angle

`f_bicorn(x,y,z, P0, P1)`. The surface is a surface of revolution.

- `P0` : Field Strength (Needs a negative field strength or a negated function)
- `P1` : Scale. The mathematics of this surface suggest that the shape should be different for different values of this parameter. In practice the difference in shape is hard to spot. Setting the scale to 3 gives a surface with a radius of about 1 unit

`f_bifolia(x,y,z, P0, P1)`. The bifolia surface looks something like the top part of a a paraboloid bounded below by another paraboloid.

- `P0` : Field Strength (Needs a negative field strength or a negated function)
- `P1` : Scale. The surface is always the same shape. Changing this parameter has the same effect as adding a scale modifier. Setting the scale to 1 gives a surface with a radius of about 1 unit

`f_blob(x,y,z, P0, P1, P2, P3, P4)`. This function generates blobs that are similar to a CSG blob with two spherical components. This function only seems to work with negative threshold settings.

- `P0` : X distance between the two components
- `P1` : Blob strength of component 1
- `P2` : Inverse blob radius of component 1
- `P3` : Blob strength of component 2
- `P4` : Inverse blob radius of component 2

`f_blob2(x,y,z, P0, P1, P2, P3)`. The surface is similar to a CSG blob with two spherical components.

- `P0` : Separation. One blob component is at the origin, and the other is this distance away on the X axis
- `P1` : Inverse size. Increase this to decrease the size of the surface
- `P2` : Blob strength
- `P3` : Threshold. Setting this parameter to 1 and the threshold to zero has exactly the same effect as setting this parameter to zero and the threshold to -1

`f_boy_surface(x,y,z, P0, P1)`. For this surface, it helps if the field strength is set low, otherwise the surface has a tendency to break up or disappear entirely. This has the side effect of making the rendering times extremely long.

- `P0` : Field Strength (Needs a negative field strength or a negated function)
- `P1` : Scale. The surface is always the same shape. Changing this parameter has the same effect as adding a scale modifier

`f_comma(x,y,z, P0)`. The 'comma' surface is very much like a comma-shape.

- `P0` : Scale

`f_cross_ellipsoids(x,y,z, P0, P1, P2, P3)`. The 'cross ellipsoids' surface is like the union of three crossed ellipsoids, one oriented along each axis.

- `P0` : Eccentricity. When less than 1, the ellipsoids are oblate, when greater than 1 the ellipsoids are prolate, when zero the ellipsoids are spherical (and hence the whole surface is a sphere)
- `P1` : Inverse size. Increase this to decrease the size of the surface
- `P2` : Diameter. Increase this to increase the size of the ellipsoids
- `P3` : Threshold. Setting this parameter to 1 and the threshold to zero has exactly the same effect as setting this parameter to zero and the threshold to -1

`f_crossed_trough(x,y,z, P0)`

- `P0` : Field Strength (Needs a negative field strength or a negated function)

`f_cubic_saddle(x,y,z, P0)`. For this surface, it helps if the field strength is set quite low, otherwise the surface has a tendency to break up or disappear entirely.

- `P0` : Field Strength (Needs a negative field strength or a negated function)

`f_cushion(x,y,z, P0)`

- `P0` : Field Strength (Needs a negative field strength or a negated function)

`f_devils_curve(x,y,z, P0)`

- `P0` : Field Strength (Needs a negative field strength or a negated function)

`f_devils_curve_2d(x,y,z, P0, P1, P2, P3, P4, P5)`. The `f_devils_curve_2d` curve can be extruded along the z axis, or using the SOR parameters it can be made into a surface of revolution. The X and Y factors control the size of the central feature.

- `P0` : Field Strength (Needs a negative field strength or a negated function)
- `P1` : X factor
- `P2` : Y factor
- `P3` : SOR Switch
- `P4` : SOR Offset
- `P5` : SOR Angle

`f_dupin_cyclid(x,y,z, P0, P1, P2, P3, P4, P5)`

- `P0` : Field Strength (Needs a negative field strength or a negated function)
- `P1` : Major radius of torus
- `P2` : Minor radius of torus
- `P3` : X displacement of torus
- `P4` : Y displacement of torus
- `P5` : Radius of inversion

`f_ellipsoid(x,y,z, P0, P1, P2)`. `f_ellipsoid` generates spheres and ellipsoids. Needs "threshold 1".
Setting these scaling parameters to 1/n gives exactly the same effect as performing a scale operation to increase the scaling by n in the corresponding direction.

- `P0` : X scale (inverse)
- `P1` : Y scale (inverse)
- `P2` : Z scale (inverse)

`f_enneper(x,y,z, P0)`

- `P0` : Field Strength (Needs a negative field strength or a negated function)

`f_flange_cover(x,y,z, P0, P1, P2, P3)`

- `P0` : Spikiness. Set this to very low values to increase the spikes. Set it to 1 and you get a sphere
- `P1` : Inverse size. Increase this to decrease the size of the surface. (The other parameters also drastically affect the size, but this parameter has no other effects)
- `P2` : Flange. Increase this to increase the flanges that appear between the spikes. Set it to 1 for no flanges

- `P3` : Threshold. Setting this parameter to 1 and the threshold to zero has exactly the same effect as setting this parameter to zero and the threshold to -1

`f_folium_surface(x,y,z, P0, P1, P2)`. A 'folium surface' looks something like a paraboloid glued to a plane.

- `P0` : Field Strength (Needs a negative field strength or a negated function)
- `P1` : Neck width factor - the larger you set this, the narrower the neck where the paraboloid meets the plane
- `P2` : Divergence - the higher you set this value, the wider the paraboloid gets

`f_folium_surface_2d(x,y,z, P0, P1, P2, P3, P4, P5)`. The `f_folium_surface_2d` curve can be rotated around the X axis to generate the same 3d surface as the `f_folium_surface`, or it can be extruded in the Z direction (by switching the SOR switch off)

- `P0` : Field Strength (Needs a negative field strength or a negated function)
- `P1` : Neck width factor - same as the 3d surface if you're revolving it around the Y axis
- `P2` : Divergence - same as the 3d surface if you're revolving it around the Y axis
- `P3` : SOR Switch
- `P4` : SOR Offset
- `P5` : SOR Angle

`f_glob(x,y,z, P0)`. One part of this surface would actually go off to infinity if it were not restricted by the contained_by shape.

- `P0` : Field Strength (Needs a negative field strength or a negated function)

`f_heart(x,y,z, P0)`

- `P0` : Field Strength (Needs a negative field strength or a negated function)

`f_helical_torus(x,y,z, P0, P1, P2, P3, P4, P5, P6, P7, P8, P9)`. With some sets of parameters, it looks like a torus with a helical winding around it. The winding optionally has grooves around the outside.

- `P0` : Major radius
- `P1` : Number of winding loops
- `P2` : Twistiness of winding. When zero, each winding loop is separate. When set to one, each loop twists into the next one. When set to two, each loop twists into the one after next
- `P3` : Fatness of winding?
- `P4` : Threshold. Setting this parameter to 1 and the threshold to zero has s similar effect as setting this parameter to zero and the threshold to 1
- `P5` : Negative minor radius? Reducing this parameter increases the minor radius of the central torus. Increasing it can make the torus disappear and be replaced by a vertical column. The value at which the surface switches from one form to the other depends on several other parameters

- `P6` : Another fatness of winding control?
- `P7` : Groove period. Increase this for more grooves
- `P8` : Groove amplitude. Increase this for deeper grooves
- `P9` : Groove phase. Set this to zero for symmetrical grooves

`f_helix1(x,y,z, P0, P1, P2, P3, P4, P5, P6)`

- `P0` : Number of helixes - e.g. 2 for a double helix
- `P1` : Period - is related to the number of turns per unit length
- `P2` : Minor radius (major radius > minor radius)
- `P3` : Major radius
- `P4` : Shape parameter. If this is greater than 1 then the tube becomes fatter in the y direction
- `P5` : Cross section type
- `P6` : Cross section rotation angle (degrees)

`f_helix2(x,y,z, P0, P1, P2, P3, P4, P5, P6)`. Needs a negated function

- `P0` : Not used
- `P1` : Period - is related to the number of turns per unit length
- `P2` : Minor radius (minor radius > major radius)
- `P3` : Major radius
- `P4` : Not used
- `P5` : Cross section type
- `P6` : Cross section rotation angle (degrees)

`f_hex_x(x,y,z, P0)`. This creates a grid of hexagonal cylinders stretching along the z-axis. The fatness is controlled by the threshold value. When this value equals 0.8660254 or cos(30) the sides will touch, because this is the distance between centers. Negating the function will inverse the surface and create a honey-comb structure. This function is also useful as pigment function.

- `P0` : No effect (but the syntax requires at least one parameter)

`f_hex_y(x,y,z, P0)`. This is function forms a lattice of infinite boxes stretching along the z-axis. The fatness is controlled by the threshold value. These boxes are rotated 60 degrees around centers, which are 0.8660254 or cos(30) away from each other. This function is also useful as pigment function.

- `P0` : No effect (but the syntax requires at least one parameter)

`f_hetero_mf(x,y,z, P0, P1, P2, P3, P4, P5)`. `f_hetero_mf (x,0,z)` makes multifractal height fields and patterns of '1/f' noise
'Multifractal' refers to their characteristic of having a fractal dimension which varies with altitude. Built from summing noise of a number of frequencies, the hetero_mf parameters determine how many, and which frequencies are to be summed.
An advantage to using these instead of a height_field {} from an image (a number of

height field programs output multifractal types of images) is that the hetero_mf function domain extends arbitrarily far in the x and z directions so huge landscapes can be made without losing resolution or having to tile a height field. Other functions of interest are `f_ridged_mf` and `f_ridge`.

- `P0` : H is the negative of the exponent of the basis noise frequencies used in building these functions (each frequency f's amplitude is weighted by the factor f - H). In landscapes, and many natural forms, the amplitude of high frequency contributions are usually less than the lower frequencies.
  When H is 1, the fractalization is relatively smooth ("1/f noise").
  As H nears 0, the high frequencies contribute equally with low frequencies as in "white noise".
- `P1` : Lacunarity' is the multiplier used to get from one 'octave' to the next. This parameter affects the size of the frequency gaps in the pattern. Make this greater than 1.0
- `P2` : Octaves is the number of different frequencies added to the fractal. Each 'Octave' frequency is the previous one multiplied by 'Lacunarity', so that using a large number of octaves can get into very high frequencies very quickly.
- `P3` : Offset is the 'base altitude' (sea level) used for the heterogeneous scaling
- `P4` : T scales the 'heterogeneity' of the fractal. T=0 gives 'straight 1/f' (no heterogeneous scaling). T=1 suppresses higher frequencies at lower altitudes
- `P5` : Generator type used to generate the noise3d. 0, 1, 2 and 3 are legal values.

`f_hunt_surface(x,y,z, P0)`

- `P0` : Field Strength (Needs a negative field strength or a negated function)

`f_hyperbolic_torus(x,y,z, P0, P1, P2)`

- `P0` : Field Strength (Needs a negative field strength or a negated function)
- `P1` : Major radius: separation between the centers of the tubes at the closest point
- `P2` : Minor radius: thickness of the tubes at the closest point

`f_isect_ellipsoids(x,y,z, P0, P1, P2, P3)`. The 'isect ellipsoids' surface is like the intersection of three crossed ellipsoids, one oriented along each axis.

- `P0` : Eccentricity. When less than 1, the ellipsoids are oblate, when greater than 1 the ellipsoids are prolate, when zero the ellipsoids are spherical (and hence the whole surface is a sphere)
- `P1` : Inverse size. Increase this to decrease the size of the surface
- `P2` : Diameter. Increase this to increase the size of the ellipsoids
- `P3` : Threshold. Setting this parameter to 1 and the threshold to zero has exactly the same effect as setting this parameter to zero and the threshold to -1

`f_kampyle_of_eudoxus(x,y,z, P0, P1, P2)`. The 'kampyle of eudoxus' is like two infinite planes with a dimple at the center.

- `P0` : Field Strength (Needs a negative field strength or a negated function)

- P1 : Dimple: When zero, the two dimples punch right through and meet at the center. Non-zero values give less dimpling
- P2 : Closeness: Higher values make the two planes become closer

f_kampyle_of_eudoxus_2d(x,y,z, P0, P1, P2, P3, P4, P5)The 2d curve that generates the above surface can be extruded in the Z direction or rotated about various axes by using the SOR parameters.

- P0 : Field Strength (Needs a negative field strength or a negated function)
- P1 : Dimple: When zero, the two dimples punch right through and meet at the center. Non-zero values give less dimpling
- P2 : Closeness: Higher values make the two planes become closer
- P3 : SOR Switch
- P4 : SOR Offset
- P5 : SOR Angle

f_klein_bottle(x,y,z, P0)

- P0 : Field Strength (Needs a negative field strength or a negated function)

f_kummer_surface_v1(x,y,z, P0). The Kummer surface consists of a collection of radiating rods.

- P0 : Field Strength (Needs a negative field strength or a negated function)

f_kummer_surface_v2(x,y,z, P0, P1, P2, P3). Version 2 of the kummer surface only looks like radiating rods when the parameters are set to particular negative values. For positive values it tends to look rather like a superellipsoid.

- P0 : Field Strength (Needs a negative field strength or a negated function)
- P1 : Rod width (negative): Setting this parameter to larger negative values increases the diameter of the rods
- P2 : Divergence (negative): Setting this number to -1 causes the rods to become approximately cylindrical. Larger negative values cause the rods to become fatter further from the origin. Smaller negative numbers cause the rods to become narrower away from the origin, and have a finite length
- P3 : Influences the length of half of the rods. Changing the sign affects the other half of the rods. 0 has no effect

f_lemniscate_of_gerono(x,y,z, P0). The "Lemniscate of Gerono" surface is an hourglass shape. Two teardrops with their ends connected.

- P0 : Field Strength (Needs a negative field strength or a negated function)

f_lemniscate_of_gerono_2d(x,y,z, P0, P1, P2, P3, P4, P5). The 2d version of the Lemniscate can be extruded in the Z direction, or used as a surface of revolution to generate the equivalent of the 3d version, or revolved in different ways.

- P0 : Field Strength (Needs a negative field strength or a negated function)
- P1 : Size: increasing this makes the 2d curve larger and less rounded

- `P2` : Width: increasing this makes the 2d curve fatter
- `P3` : SOR Switch
- `P4` : SOR Offset
- `P5` : SOR Angle

`f_mesh1(x,y,z, P0, P1, P2, P3, P4)` The overall thickness of the threads is controlled by the isosurface threshold, not by a parameter. If you render a mesh1 with zero threshold, the threads have zero thickness and are therefore invisible. Parameters P2 and P4 control the shape of the thread relative to this threshold parameter.

- `P0` : Distance between neighboring threads in the x direction
- `P1` : Distance between neighboring threads in the z direction
- `P2` : Relative thickness in the x and z directions
- `P3` : Amplitude of the weaving effect. Set to zero for a flat grid
- `P4` : Relative thickness in the y direction

`f_mitre(x,y,z, P0)`. The 'Mitre' surface looks a bit like an ellipsoid which has been nipped at each end with a pair of sharp nosed pliers.

- `P0` : Field Strength (Needs a negative field strength or a negated function)

`f_nodal_cubic(x,y,z, P0)`. The 'Nodal Cubic' is something like what you'd get if you were to extrude the Stophid2D curve along the X axis and then lean it over.

- `P0` : Field Strength (Needs a negative field strength or a negated function)

`f_noise3d(x,y,z)`

`f_noise_generator(x,y,z, P0)`

- `P0` : Noise generator number

`f_odd(x,y,z, P0)`

- `P0` : Field Strength (Needs a negative field strength or a negated function)

`f_ovals_of_cassini(x,y,z, P0, P1, P2, P3)`. The Ovals of Cassini are a generalization of the torus shape.

- `P0` : Field Strength (Needs a negative field strength or a negated function)
- `P1` : Major radius - like the major radius of a torus
- `P2` : Filling. Set this to zero, and you get a torus. Set this to a higher value and the hole in the middle starts to heal up. Set it even higher and you get an ellipsoid with a dimple
- `P3` : Thickness. The higher you set this value, the plumper is the result

`f_paraboloid(x,y,z, P0)`. This paraboloid is the surface of revolution that you get if you rotate a parabola about the Y axis.

- `P0` : Field Strength (Needs a negative field strength or a negated function)

`f_parabolic_torus(x,y,z, P0, P1, P2)`

- `P0` : Field Strength (Needs a negative field strength or a negated function)
- `P1` : Major radius
- `P2` : Minor radius

`f_ph(x,y,z)` = atan2(sqrt(x*x + z*z), y)
When used alone, the "PH" function gives a surface that consists of all points that are at a particular latitude, i.e. a cone. If you use a threshold of zero (the default) this gives a cone of width zero, which is invisible. Also look at `f_th` and `f_r`

`f_pillow(x,y,z, P0)`

- `P0` : Field Strength

`f_piriform(x,y,z, P0)`. The piriform surface looks rather like half a lemniscate.

- `P0` : Field Strength

`f_piriform_2d(x,y,z, P0, P1, P2, P3, P4, P5, P6)`. The 2d version of the "Piriform" can be extruded in the Z direction, or used as a surface of revolution to generate the equivalent of the 3d version.

- `P0` : Field Strength (Needs a negative field strength or a negated function)
- `P1` : Size factor 1: increasing this makes the curve larger
- `P2` : Size factor 2: making this less negative makes the curve larger but also thinner
- `P3` : Fatness: increasing this makes the curve fatter
- `P4` : SOR Switch
- `P5` : SOR Offset
- `P6` : SOR Angle

`f_poly4(x,y,z, P0, P1, P2, P3, P4)`. This `f_poly4` can be used to generate the surface of revolution of any polynomial up to degree 4.
To put it another way: If we call the parameters A, B, C, D, E; then this function generates the surface of revolution formed by revolving "x = A + By + Cy2 + Dy3 + Ey4" around the Y axis.

- `P0` : Constant
- `P1` : Y coefficient
- `P2` : Y2 coefficient
- `P3` : Y3 coefficient
- `P4` : Y4 coefficient

`f_polytubes(x,y,z, P0, P1, P2, P3, P4, P5)`. The 'Polytubes' surface consists of a number of tubes. Each tube follows a 2d curve which is specified by a polynomial of degree 4 or less. If we look at the parameters, then this function generates "P0" tubes which all follow the equation " x = P1 + P2y + P3y2 + P4y3 + P5y4 " arranged around the Y axis.
This function needs a positive threshold (fatness of the tubes).

- P0 : Number of tubes
- P1 : Constant
- P2 : Y coefficient
- P3 : Y2 coefficient
- P4 : Y3 coefficient
- P5 : Y4 coefficient

f_quantum(x,y,z, P0). It resembles the shape of the electron density cloud for one of the d orbitals.

- P0 : Not used, but required

f_quartic_paraboloid(x,y,z, P0). The 'Quartic Paraboloid' is similar to a paraboloid, but has a squarer shape.

- P0 : Field Strength (Needs a negative field strength or a negated function)

f_quartic_saddle(x,y,z, P0). The 'Quartic saddle' is similar to a saddle, but has a squarer shape.

- P0 : Field Strength

f_quartic_cylinder(x,y,z, P0, P1, P2). The 'Quartic cylinder' looks a bit like a cylinder that's swallowed an egg.

- P0 : Field Strength (Needs a negative field strength or a negated function)
- P1 : Diameter of the "egg"
- P2 : Controls the width of the tube and the vertical scale of the "egg"

f_r(x,y,z) = sqrt(x*x + y*y + z*z)
When used alone, the "R" function gives a surface that consists of all the points that are a specific distance (threshold value) from the origin, i.e. a sphere. Also look at f_ph and f_th

f_ridge(x,y,z, P0, P1, P2, P3, P4, P5). This function is mainly intended for modifying other surfaces as you might use a height field or to use as pigment function. Other functions of interest are f_hetero_mf and f_ridged_mf.

- P0 : Lambda
- P1 : Octaves
- P2 : Omega
- P3 : Offset
- P4 : Ridge
- P5 : Generator type used to generate the noise3d. 0, 1, 2 and 3 are legal values.

f_ridged_mf(x,y,z, P0, P1, P2, P3, P4, P5). The "Ridged Multifractal" surface can be used to create multifractal height fields and patterns. 'Multifractal' refers to their characteristic of having a fractal dimension which varies with altitude. They are built from summing noise of a number of frequencies. The f_ridged_mf parameters determine how many, and which frequencies are to be summed, and how the different

frequencies are weighted in the sum.
An advantage to using these instead of a `height_field{}` from an image is that the ridged_mf function domain extends arbitrarily far in the x and z directions so huge landscapes can be made without losing resolution or having to tile a height field. Other functions of interest are `f_hetero_mf` and `f_ridge`.

- `P0` : H is the negative of the exponent of the basis noise frequencies used in building these functions (each frequency f's amplitude is weighted by the factor fE- H). When H is 1, the fractalization is relatively smooth. As H nears 0, the high frequencies contribute equally with low frequencies
- `P1` : Lacunarity is the multiplier used to get from one "octave" to the next in the "fractalization".
  This parameter affects the size of the frequency gaps in the pattern. (Use values greater than 1.0)
- `P2` : Octaves is the number of different frequencies added to the fractal. Each octave frequency is the previous one multiplied by "Lacunarity". So, using a large number of octaves can get into very high frequencies very quickly
- `P3` : Offset gives a fractal whose fractal dimension changes from altitude to altitude. The high frequencies at low altitudes are more damped than at higher altitudes, so that lower altitudes are smoother than higher areas
- `P4` : Gain weights the successive contributions to the accumulated fractal result to make creases stick up as ridges
- `P5` : Generator type used to generate the noise3d. 0, 1, 2 and 3 are legal values.

`f_rounded_box(x,y,z, P0, P1, P2, P3)`. The Rounded Box is defined in a cube from <-1, -1, -1> to <1, 1, 1>. By changing the " Scale" parameters, the size can be adjusted, without affecting the Radius of curvature.

- `P0` : Radius of curvature. Zero gives square corners, 0.1 gives corners that match "sphere {0, 0.1}"
- `P1` : Scale x
- `P2` : Scale y
- `P3` : Scale z

`f_sphere(x,y,z, P0)`

- `P0`: radius of the sphere

`f_spikes(x,y,z, P0, P1, P2, P3, P4)`

- `P0` : Spikiness. Set this to very low values to increase the spikes. Set it to 1 and you get a sphere
- `P1` : Hollowness. Increasing this causes the sides to bend in more
- `P2` : Size. Increasing this increases the size of the object
- `P3` : Roundness. This parameter has a subtle effect on the roundness of the spikes
- `P4` : Fatness. Increasing this makes the spikes fatter

`f_spikes_2d(x,y,z, P0, P1, P2, P3)` =2-D function : f = f(x, z) - y

- `P0` : Height of central spike
- `P1` : Frequency of spikes in the X direction
- `P2` : Frequency of spikes in the Z direction
- `P3` : Rate at which the spikes reduce as you move away from the center

`f_spiral(x,y,z, P0, P1, P2, P3, P4, P5)`

- `P0` : Distance between windings
- `P1` : Thickness
- `P2` : Outer diameter of the spiral. The surface behaves as if it is contained_by a sphere of this diameter
- `P3` : Not used
- `P4` : Not used
- `P5` : Cross section type

`f_steiners_roman(x,y,z, P0)`. The "Steiners Roman" is composed of four identical triangular pads which together make up a sort of rounded tetrahedron. There are creases along the X, Y and Z axes where the pads meet.

- `P0` : Field Strength (Needs a negative field strength or a negated function)

`f_strophoid(x,y,z, P0, P1, P2, P3)`. The "Strophoid" is like an infinite plane with a bulb sticking out of it.

- `P0` : Field Strength (Needs a negative field strength or a negated function)
- `P1` : Size of bulb. Larger values give larger bulbs. Negative values give a bulb on the other side of the plane
- `P2` : Sharpness. When zero, the bulb is like a sphere that just touches the plane. When positive, there is a crossover point. When negative the bulb simply bulges out of the plane like a pimple
- `P3` : Flatness. Higher values make the top end of the bulb fatter

`f_strophoid_2d(x,y,z, P0, P1, P2, P3, P4, P5, P6)`. The 2d strophoid curve can be extruded in the Z direction or rotated about various axes by using the SOR parameters.

- `P0` : Field Strength
- `P1` : Size of bulb. Larger values give larger bulbs. Negative values give a bulb on the other side of the plane
- `P2` : Sharpness. When zero, the bulb is like a sphere that just touches the plane. When positive, there is a crossover point. When negative the bulb simply bulges out of the plane like a pimple
- `P3` : Fatness. Higher values make the top end of the bulb fatter
- `P4` : SOR Switch
- `P5` : SOR Offset

- P6 : SOR Angle

f_superellipsoid(x,y,z, P0, P1). Needs a negative field strength or a negated function.

- P0 : east-west exponentx
- P1 : north-south exponent

f_th(x,y,z) = atan2(x, z)
f_th() is a function that is only useful when combined with other surfaces.
It produces a value which is equal to the "theta" angle, in radians, at any point. The theta angle is like the longitude coordinate on the Earth. It stays the same as you move north or south, but varies from east to west. Also look at f_ph and f_r

f_torus(x,y,z, P0, P1)

- P0 : Major radius
- P1 : Minor radius

f_torus2(x,y,z, P0, P1, P2). This is different from the f_torus function which just has the major and minor radii as parameters.

- P0 : Field Strength (Needs a negative field strength or a negated function)
- P1 : Major radius
- P2 : Minor radius

f_torus_gumdrop(x,y,z, P0). The "Torus Gumdrop" surface is something like a torus with a couple of gumdrops hanging off the end.

- P0 : Field Strength (Needs a negative field strength or a negated function)

f_umbrella(x,y,z, P0)

- P0 : Field Strength (Needs a negative field strength or a negated function)

f_witch_of_agnesi(x,y,z, P0, P1, P2, P3, P4, P5). The "Witch of Agnesi" surface looks something like a witches hat.

- P0 : Field Strength (Needs a negative field strength or a negated function)
- P1 : Controls the width of the spike. The height of the spike is always about 1 unit

f_witch_of_agnesi_2d(x,y,z, P0, P1, P2, P3, P4, P5). The 2d version of the "Witch of Agnesi" curve can be extruded in the Z direction or rotated about various axes by use of the SOR parameters.

- P0 : Field Strength (Needs a negative field strength or a negated function)
- P1 : Controls the size of the spike
- P2 : Controls the height of the spike
- P3 : SOR Switch
- P4 : SOR Offset
- P5 : SOR Angle

6.7.3 Pre defined functions

`eval_pigment(Pigm, Vect)`, This macro evaluates the color of a pigment at a specific point. Some pigments require more information than simply a point, slope pattern based pigments for example, and will not work with this macro. However, most pigments will work fine.
Parameters:

- `Vect` = The point at which to evaluate the pigment.
- `Pigm` = The pigment to evaluate.

`f_snoise3d(x, y, z)`. Just like f_noise3d(), but returns values in the range [-1, 1].

`f_sine_wave(val, amplitude, frequency)`. Turns a ramping waveform into a sine waveform.

`f_scallop_wave(val, amplitude, frequency)`. Turns a ramping waveform into a "scallop_wave" waveform.

Pattern functions

Predefined pattern functions, useful for building custom function patterns or performing "displacement mapping" on isosurfaces. Many of them aren't really useful for these purposes, they are simply included for completeness.

Some are not implemented at all because they require special parameters that must be specified in the definition, or information that isn't available to pattern functions. For this reason, you probably would want to define your own versions of these functions.

All of these functions take three parameters, the XYZ coordinates of the point to evaluate the pattern at.

`f_agate(x, y, z)`

`f_boxed(x, y, z)`

`f_bozo(x, y, z)`

`f_brick(x, y, z)`

`f_bumps(x, y, z)`

`f_checker(x, y, z)`

`f_crackle(x, y, z)`

This pattern has many more options in POV 3.5 than in previous versions, this function uses the defaults.

`f_cylindrical(x, y, z)`

`f_dents(x, y, z)`

`f_gradientX(x, y, z)`

`f_gradientY(x, y, z)`

`f_gradientZ(x, y, z)`

f_granite(x, y, z)

f_hexagon(x, y, z)

f_leopard(x, y, z)

f_mandel(x, y, z)

Only the basic mandel pattern is implemented, its variants and the other fractal patterns are not implemented.

f_marble(x, y, z)

f_onion(x, y, z)

f_planar(x, y, z)

f_radial(x, y, z)

f_ripples(x, y, z)

f_spherical(x, y, z)

f_spiral1(x, y, z)

f_spiral2(x, y, z)

f_spotted(x, y, z)

f_waves(x, y, z)

f_wood(x, y, z)

f_wrinkles(x, y, z)

6.8 glass.inc, glass_old.inc

This file contains glass materials using new features in POV 3.1 and 3.5. The old glass.inc file is still included for backwards compatibility (it is named glass_old.inc, and is included by glass.inc, so you don't need to change any scenes), but these materials will give more realistic results.

6.8.1 Glass colors (with transparency)

Col_Glass_Beerbottle	Col_Glass_General	Col_Glass_Ruby
Col_Glass_Bluish	Col_Glass_Green	Col_Glass_Vicksbottle
Col_Glass_Clear	Col_Glass_Old	Col_Glass_Winebottle
Col_Glass_Dark_Green	Col_Glass_Orange	Col_Glass_Yellow

Table 6.1: glass.inc glass colors with transparency

Col_Amber_01	Col_Amber_06	Col_Amethyst_02
Col_Amber_02	Col_Amber_07	Col_Amethyst_03
Col_Amber_03	Col_Amber_08	Col_Amethyst_04
Col_Amber_04	Col_Amber_09	Col_Amethyst_05
Col_Amber_05	Col_Amethyst_01	Col_Amethyst_06
Col_Apatite_01	Col_Aquamarine_01	Col_Aquamarine_06
Col_Apatite_02	Col_Aquamarine_02	Col_Azurite_01
Col_Apatite_03	Col_Aquamarine_03	Col_Azurite_02
Col_Apatite_04	Col_Aquamarine_04	Col_Azurite_03
Col_Apatite_05	Col_Aquamarine_05	Col_Azurite_04
Col_Beerbottle	Col_Citrine_01	Col_Emerald_04
Col_Blue_01	Col_Dark_Green	Col_Emerald_05
Col_Blue_02	Col_Emerald_01	Col_Emerald_06
Col_Blue_03	Col_Emerald_02	Col_Emerald_07
Col_Blue_04	Col_Emerald_03	Col_Fluorite_01
Col_Fluorite_02	Col_Fluorite_07	Col_Green_02
Col_Fluorite_03	Col_Fluorite_08	Col_Green_03
Col_Fluorite_04	Col_Fluorite_09	Col_Green_04
Col_Fluorite_05	Col_Green	Col_Gypsum_01
Col_Fluorite_06	Col_Green_01	Col_Gypsum_02
Col_Gypsum_03	Col_Red_01	Col_Ruby_01
Col_Gypsum_04	Col_Red_02	Col_Ruby_02
Col_Gypsum_05	Col_Red_03	Col_Ruby_03
Col_Gypsum_06	Col_Red_04	Col_Ruby_04
Col_Orange	Col_Ruby	Col_Ruby_05
Col_Sapphire_01	Col_Topaz_03	Col_Tourmaline_05
Col_Sapphire_02	Col_Tourmaline_01	Col_Tourmaline_06
Col_Sapphire_03	Col_Tourmaline_02	Col_Vicksbottle
Col_Topaz_01	Col_Tourmaline_03	Col_Winebottle
Col_Topaz_02	Col_Tourmaline_04	Col_Yellow
Col_Yellow_01		
Col_Yellow_02		
Col_Yellow_03		
Col_Yellow_04		

Table 6.2: glass.inc glass colors without transparency for fade_color

6.8.2 Glass colors (without transparency, for fade_color)

6.8.3 Glass finishes

F_Glass5, ..., F_Glass10

6.8.4 Glass interiors

I_Glass1, ..., I_Glass4

I_Glass_Fade_Sqr1 (identical to I_Glass1)

I_Glass_Fade_Exp1 (identical to I_Glass2)

I_Glass_Fade_Exp2 (identical to I_Glass3)

I_Glass_Fade_Exp3 (identical to I_Glass4)

Glass interiors with various fade_power settings.

I_Glass_Dispersion1, I_Glass_Dispersion2

Glass interiors with dispersion. I_Glass_Dispersion1 has an approximately natural glass dispersion. I_Glass_Dispersion2 is exaggerated.

I_Glass_Caustics1, I_Glass_Caustics2

Glass interiors with caustics.

6.8.5 Glass interior macros

I_Glass_Exp(Distance) and I_Glass_Sqr(Distance).
These macros return an interior with either exponential or fade_power 2 falloff, and a fade_distance of Distance.

6.8.6 glass_old.inc

This file contains glass textures for POV-Ray versions 3.1 and earlier. These textures do not take advantage of the new features of POV-Ray 3.5 and are included for backwards compatability, you will get better results with the materials in glass.inc.

These textures are designed to be used with the I_Glass interior, also defined in this file.

Glass finishes

F_Glass1, ..., F_Glass4

Glass textures

`T_Glass1`

Simple clear glass.

`T_Glass2`

More like an acrylic plastic.

`T_Glass3`

An excellent lead crystal glass.

`T_Glass4`

`T_Old_Glass`

`T_Winebottle_Glass`

`T_Beerbottle_Glass`

`T_Ruby_Glass`

`T_Green_Glass`

`T_Dark_Green_Glass`

`T_Yellow_Glass`

`T_Orange_Glass`

Orange/amber glass.

`T_Vicksbottle_Glass`

6.9 math.inc

This file contains many general math functions and macros.

6.9.1 Float functions and macros

`even(N)`. A function to test whether N is even, returns 1 when true, 0 when false.
Parameters

- `N` = Input value

`odd(N)`. A function to test whether N is odd, returns 1 when true, 0 when false.
Parameters

- `N` = Input value

`Interpolate(GC, GS, GE, TS, TE, Method)`. Interpolation macro, interpolates between the float values `TS` and `TE`. The method of interpolation is cosine, linear or exponential. The position where to evaluate the interpolation is determined by the position of `GC` in the range `GS` - `GE`. See example.
Parameters:

- `GC` = global current, float value within the range GS - GE
- `GS` = global start
- `GE` = global end
- `TS` = target start
- `TE` = target end
- `Method` = interpolation method, float value:
  - `Method` < 0 : exponential, using the value of Method as exponent.
  - `Method` = 0 : cosine interpolation.
  - `Method` > 0 : exponential, using the value of Method as exponent.
    - `Method` = 1 : linear interpolation,

```
#declare A = Interpolate(0.5, 0, 1, 0, 10, 1);
#debug str(A,0,2)
// result A = 5.00

#declare A = Interpolate(0.0,-2, 2, 0, 10, 1);
#debug str(A,0,2)
// result A = 5.00

#declare A = Interpolate(0.5, 0, 1, 0, 10, 2);
#debug str(A,0,2)
// result A = 2.50
```

`Mean(A)`. A macro to compute the average of an array of values.
Parameters:

- `A` = An array of float or vector values.

`Std_Dev(A, M)`. A macro to compute the standard deviation.
Parameters:

- `A` = An array of float values.
- `M` = Mean of the floats in the array.

`GetStats(ValArr)`. This macro declares a global array named "`StatisticsArray`" containing: N, Mean, Min, Max, and Standard Deviation
Parameters:

- `A` = An array of float values.

`Histogram(ValArr, Intervals)`. This macro declares a global, 2D array named "`HistogramArray`". The first value in the array is the center of the interval/bin, the second the number of values in that interval.
Parameters:

- `ValArr` = An array with values.
- `Intervals` = The desired number of intervals/bins.

`sind(v), cosd(v), tand(v), asind(v), acosd(v), atan2d(a, b)`. These functions are versions of the trigonometric functions using degrees, instead of radians, as the angle unit.
Parameters:
The same as for the analogous built-in trig function.

`max3(a, b, c)`. A function to find the largest of three numbers.
Parameters:

- `a, b, c` = Input values.

`min3(a, b, c)`. A function to find the smallest of three numbers.
Parameters:

- `a, b, c` = Input values.

`f_sqr(v)`. A function to square a number.
Parameters:

- `v` = Input value.

`sgn(v)`. A function to show the sign of the number. Returns -1 or 1 depending on the sign of v.
Parameters:

- `v` = Input value.

`clip(V, Min, Max)`. A function that limits a value to a specific range, if it goes outside that range it is "clipped". Input values larger than `Max` will return `Max`, those less than `Min` will return `Min`.
Parameters:

- `V` = Input value.
- `Min` = Minimum of output range.
- `Max` = Maximum of output range.

`clamp(V, Min, Max)`. A function that limits a value to a specific range, if it goes outside that range it is "clamped" to this range, wrapping around. As the input increases or decreases outside the given range, the output will repeatedly sweep through that range, making a "sawtooth" waveform.
Parameters:

- `V` = Input value.
- `Min` = Minimum of output range.
- `Max` = Maximum of output range.

`adj_range(V, Min, Max)`. A function that adjusts input values in the range [0, 1] to a given range. An input value of 0 will return `Min`, 1 will return `Max`, and values outside the [0, 1] range will be linearly extrapolated (the graph will continue in a straight line).
Parameters:

- `V` = Input value.
- `Min` = Minimum of output range.
- `Max` = Maximum of output range.

`adj_range2(V, InMin, InMax, OutMin, OutMax)`. Like `f_range()`, but adjusts input values in the range `[InMin, InMax]` to the range `[OutMin, OutMax]`.
Parameters:

- `V` = Input value.
- `InMin` = Minimum of input range.
- `InMax` = Maximum of input range.
- `OutMin` = Minimum of output range.
- `OutMax` = Maximum of output range.

6.9.2 Vector functions and macros

These are all macros in the current version because functions can not take vector parameters, but this may change in the future.

`VSqr(V)`. Square each individual component of a vector, equivalent to `V*V`.
Parameters:

- `V` = Vector to be squared.

`VPow(V, P)`, `VPow5D(V, P)`. Raise each individual component of a vector to a given power.
Parameters:

- `V` = Input vector.
- `P` = Power.

`VEq(V1, V2)`. Tests for equal vectors, returns true if all three components of `V1`equal the respective components of `V2`.
Parameters:

- `V1, V2` = The vectors to be compared.

`VEq5D(V1, V2)`. A 5D version of `VEq()`. Tests for equal vectors, returns true if all 5 components of `V1` equal the respective components of `V2`.
Parameters:

- `V1, V2` = The vectors to be compared.

`VZero(V)`. Tests for a < 0, 0, 0> vector.
Parameters:

- `V` = Input vector.

`VZero5D(V)`. Tests for a < 0, 0, 0, 0, 0> vector.
Parameters:

- `V` = Input vector.

`VLength5D(V)`. Computes the length of a 5D vector.
Parameters:

- `V` = Input vector.

`VNormalize5D(V)`. Normalizes a 5D vector.
Parameters:

- `V` = Input vector.

`VDot5D(V1, V2)`. Computes the dot product of two 5D vectors. See vdot() for more information on dot products.
Parameters:

- `V` = Input vector.

`VCos_Angle(V1, V2)`. Compute the cosine of the angle between two vectors.
Parameters:

- `V1, V2` = Input vectors.

`VAngle(V1, V2), VAngleD(V1, V2)`. Compute the angle between two vectors. `VAngle()` returns the angle in radians, `VAngleD()` in degrees.
Parameters:

- `V1, V2` = Input vectors.

`VRotation(V1, V2, Axis), VRotationD(V1, V2, Axis)`.Compute the rotation angle from V1 to V2 around Axis. Axis should be perpendicular to both V1 and V2. The output will be in the range between -pi and pi radians or between -180 degrees and 180 degrees if you are using the degree version. However, if Axis is set to <0,0,0> the output will always be positive or zero, the same result you will get with the VAngle() macros.
Parameters:

- `V1, V2` = Input vectors.

`VDist(V1, V2)`. Compute the distance between two points.
Parameters:

- `V1, V2` = Input vectors.

`VPerp_To_Vector(V)`. Find a vector perpendicular to the given vector.
Parameters:

- `V` = Input vector.

`VPerp_To_Plane(V1, V2)`. Find a vector perpendicular to both given vectors. In other words, perpendicular to the plane defined by the two input vectors
Parameters:

- `V1, V2` = Input vectors.

`VPerp_Adjust(V1, Axis)`. Find a vector perpendicular to Axis and in the plane of V1 and Axis. In other words, the new vector is a version of V1 adjusted to be perpendicular to Axis.
Parameters:

- `V1, Axis` = Input vectors.

`VProject_Plane(V1, Axis)`. Project vector V1 onto the plane defined by Axis.
Parameters:

- `V1` = Input vectors.

- Axis = Normal of the plane.

VProject_Axis(V1, Axis). Project vector V1 onto the axis defined by Axis.
Parameters:

- V1, Axis = Input vectors.

VMin(V), VMax(V). Find the smallest or largest component of a vector.
Parameters:

- V = Input vector.

VWith_Len(V, Len). Create a vector parallel to a given vector but with a given length.
Parameters:

- V = Direction vector.
- Len = Length of desired vector.

6.9.3 Vector Analysis

SetGradientAccuracy(Value): all below macros make use of a constant named '__Gradient_Fn_Accuracy_' for numerical approximation of the derivatives. This constant can be changed with the macro, the default value is 0.001.

fn_Gradient(Fn): macro calculating the gradient of a function as a function.
Parameters:

- Fn = function to calculate the gradient from.

fn_Gradient_Directional(Fn, Dir): macro calculating the gradient of a function in one direction as a function.
Parameters:

- Fn = function to calculate the gradient from.
- Dir = direction to calculate the gradient.

fn_Divergence(Fnx, Fny, Fnz): macro calculating the divergence of a (vector) function as a function.
Parameters:

- Fnx, Fny, Fnz= x, y and z components of a vector function.

vGradient(Fn, p0): macro calculating the gradient of a function as a vector expression.
Parameters:

- Fn = function to calculate the gradient from.
- p0 = point where to calculate the gradient.

vCurl(Fnx, Fny, Fnz, p0): macro calculating the curl of a (vector) function as a vector expression
Parameters:

- Fnx, Fny, Fnz = x, y and z components of a vector function.
- p0 = point where to calculate the gradient.

`Divergence(Fnx, Fny, Fnz, p0)`: macro calculating the divergence of a (vector) function as a float expression
Parameters:

- `Fnx, Fny, Fnz` = x, y and z components of a vector function.
- `p0` = point where to calculate the gradient.

`Gradient_Length(Fn, p0)`: macro calculating the length of the gradient of a function as a float expression.
Parameters:

- `Fn` = function to calculate the gradient from.
- `p0` = point where to calculate the gradient.

`Gradient_Directional(Fn, p0, Dir)`: macro calculating the gradient of a function in one direction as a float expression.
Parameters:

- `Fn` = function to calculate the gradient from.
- `p0` = point where to calculate the gradient.
- `Dir` = direction to calculate the gradient.

6.10 metals.inc, golds.inc

These files define several metal textures. The file metals.inc contains copper, silver, chrome, and brass textures, and golds.inc contains the gold textures.
Rendering the demo files will come in useful in using these textures.

6.10.1 metals.inc

Colors:

`P_Brass1`

Dark brown bronze.

`P_Brass2`

Somewhat lighter brown than Brass4. Old penny, in soft finishes.

`P_Brass3`

Used by Steve Anger's Polished_Brass. Slightly coppery.

`P_Brass4`

A little yellower than Brass1.

`P_Brass5`

Very light bronze, ranges from med tan to almost white.

`P_Copper1`

Bronze-like. Best in finish #C.

P_Copper2

Slightly brownish copper/bronze. Best in finishes #B-#D.

P_Copper3

Reddish-brown copper. Best in finishes #C-#E.

P_Copper4

Pink copper, like new tubing. Best in finishes #C-#E.

P_Copper5

Bronze in softer finishes, gold in harder finishes.

P_Chrome1

20% Gray. Used in Steve Anger's Polished_Chrome.

P_Chrome2

Slightly blueish 60% gray. Good steel w/finish #A.

P_Chrome3

50% neutral gray.

P_Chrome4

75% neutral gray.

P_Chrome5

95% neutral gray.

P_Silver1

Yellowish silverplate. Somewhat tarnished looking.

P_Silver2

Not quite as yellowish as Silver1 but more so than Silver3.

P_Silver3

Reasonably neutral silver.

P_Silver4

P_Silver5

Finishes:

F_MetalA

Very soft and dull.

F_MetalB

Fairly soft and dull.

F_MetalC

Medium reflectivity. Holds color well.

F_MetalD

Very hard and highly polished. High reflectivity.

F_MetalE

Very highly polished and reflective.

Textures:

T_Brass_1A to T_Brass_5E

T_Copper_1A to T_Copper_5E

T_Chrome_1A to T_Chrome_5E

T_Silver_1A to T_Silver_5E

6.10.2 golds.inc

This file has its own versions of F_MetalA through F_MetalB.

The gold textures themselves are T_Gold_1A through T_Gold_5E.

6.11 rand.inc

A collection of macros for generating random numbers, as well as 4 predefined random number streams: RdmA, RdmB, RdmC, and RdmD. There are macros for creating random numbers in a flat distribution (all numbers equally likely) in various ranges, and a variety of other distributions.

6.11.1 Flat Distributions

SRand(Stream). "Signed rand()", returns random numbers in the range [-1, 1].
Parameters:

- Stream = Random number stream.

RRand(Min, Max, Stream). Returns random numbers in the range [Min, Max].
Parameters:

- Min = The lower end of the output range.
- Max = The upper end of the output range.
- Stream = Random number stream.

VRand(Stream). Returns random vectors in a box from < 0, 0, 0> to < 1, 1, 1>
Parameters:

- Stream = Random number stream.

`VRand_In_Box(PtA, PtB, Stream)`. Like VRand(), this macro returns a random vector in a box, but this version lets you specify the two corners of the box.
Parameters:

- `PtA` = Lower-left-bottom corner of box.
- `PtB` = Upper-right-top corner of box.
- `Stream` = Random number stream.

`VRand_In_Sphere(Stream)`. Returns a random vector in a unit-radius sphere located at the origin.
Parameters:

- `Stream` = Random number stream.

`VRand_On_Sphere(Stream)`. Returns a random vector on the surface of a unit-radius sphere located at the origin.
Parameters:

- `Stream` = Random number stream.

`VRand_In_Obj(Object, Stream)` This macro takes a solid object and returns a random point that is inside it. It does this by randomly sampling the bounding box of the object, and can be quite slow if the object occupies a small percentage of the volume of its bounding box (because it will take more attempts to find a point inside the object). This macro is best used on finite, solid objects (non-solid objects, such as meshes and bezier patches, don't have a defined "inside", and will not work).
Parameters:

- `Object` = The object the macro chooses the points from.
- `Stream` = Random number stream.

6.11.2 Other Distributions

Continuous Symmetric Distributions

`Rand_Cauchy(Mu, Sigma, Stream)`. Cauchy distribution.
Parameters:

- `Mu` = Mean.
- `Sigma` = Standard deviation.
- `Stream` = Random number stream.

`Rand_Student(N, Stream)`. Student's-t distribution.
Parameters:

- `N` = degrees of freedom.
- `Stream` = Random number stream.

`Rand_Normal(Mu, Sigma, Stream)`. Normal distribution.
Parameters:

- `Mu` = Mean.

- `Sigma` = Standard deviation.
- `Stream` = Random number stream.

`Rand_Gauss(Mu, Sigma, Stream)`. Gaussian distribution. Like Rand_Normal(), but a bit faster.
Parameters:

- `Mu` = Mean.
- `Sigma` = Standard deviation.
- `Stream` = Random number stream.

Continuous Skewed Distributions

`Rand_Spline(Spline, Stream)`. This macro takes a spline describing the desired distribution. The T value of the spline is the output value, and the .y value its chance of occuring.
Parameters:

- `Spline` = A spline determining the distribution.
- `Stream` = Random number stream.

`Rand_Gamma(Alpha, Beta, Stream)`. Gamma distribution.
Parameters:

- `Alpha` = Shape parameter > 0.
- `Beta` = Scale parameter > 0.
- `Stream` = Random number stream.

`Rand_Beta(Alpha, Beta, Stream)`. Beta variate.
Parameters:

- `Alpha` = Shape Gamma1.
- `Beta` = Scale Gamma2.
- `Stream` = Random number stream.

`Rand_Chi_Square(N, Stream)`. Chi Square random variate.
Parameters:

- `N` = Degrees of freedom (integer).
- `Stream` = Random number stream.

`Rand_F_Dist(N, M, Stream)`. F-distribution.
Parameters:

- `N, M` = Degrees of freedom.
- `Stream` = Random number stream.

`Rand_Tri(Min, Max, Mode, Stream)`. Triangular distribution
Parameters:

- `Min, Max, Mode`: Min < Mode < Max.
- `Stream` = Random number stream.

`Rand_Erlang(Mu, K, Stream)`. Erlang variate.
Parameters:

- `Mu` = Mean >= 0.
- `K` = Number of exponential samples.
- `Stream` = Random number stream.

`Rand_Exp(Lambda, Stream)`. Exponential distribution.
Parameters:

- `Lambda` = rate = 1/mean.
- `Stream` = Random number stream.

`Rand_Lognormal(Mu, Sigma, Stream)`. Lognormal distribution.
Parameters:

- `Mu` = Mean.
- `Sigma` = Standard deviation.
- `Stream` = Random number stream.

`Rand_Pareto(Alpha, Stream)`. Pareto distribution.
Parameters:

- `Alpha` = ?
- `Stream` = Random number stream.

`Rand_Weibull(Alpha, Beta, Stream)`. Weibull distribution.
Parameters:

- `Alpha` = ?
- `Beta` = ?
- `Stream` = Random number stream.

Discrete Distributions

`Rand_Bernoulli(P, Stream)` and `Prob(P, Stream)`. Bernoulli distribution. Output is true with probability equal to the value of P and false with a probability of 1 - P.
Parameters:

- `P` = probability range (0-1).
- `Stream` = Random number stream.

`Rand_Binomial(N, P, Stream)`. Binomial distribution.
Parameters:

- `N` = Number of trials.
- `P` = Probability (0-1)

- `Stream` = Random number stream.

`Rand_Geo(P, Stream)`. Geometric distribution.
Parameters:

- `P` = Probability (0-1).
- `Stream` = Random number stream.

`Rand_Poisson(Mu, Stream)`. Poisson distribution.
Parameters:

- `Mu` = Mean.
- `Stream` = Random number stream.

6.12 shapes.inc, shapes_old.inc, shapes2.inc, shapesq.inc

These files contain predefined shapes and shape-generation macros.

”shapes.inc” includes ”shapes_old.inc” and contains many macros for working with objects, and for creating special objects, such as bevelled text, spherical height fields, and rounded shapes.

Many of the objects in ”shapes_old.inc” are not very useful in the newer versions of POV-Ray, and are kept for backwards compatability with old scenes written for versions of POV-Ray that lacked primitives like cones, disks, planes, etc.

The file ”shapes2.inc” contains some more useful shapes, including regular polyhedrons, and ”shapesq.inc” contains several quartic and cubic shape definitions.

Some of the shapes in ”shapesq.inc” would be much easier to generate, more flexible, and possibly faster rendering as isosurfaces, but are still useful for two reasons: backwards compatability, and the fact that isosurfaces are always finite.

6.12.1 shapes.inc

`Isect(Pt, Dir, Obj, OPt)` and `IsectN(Pt, Dir, Obj, OPt, ONorm)`
These macros are interfaces to the trace() function. Isect() only returns the intersection point, IsectN() returns the surface normal as well. These macros return the point and normal information through their parameters, and true or false depending on whether an intersection was found:
If an intersection is found, they return true and set OPt to the intersection point, and ONorm to the normal. Otherwise they return false, and do not modify OPt or ONorm.
Parameters:

- `Pt` = The origin (starting point) of the ray.
- `Dir` = The direction of the ray.
- `Obj` = The object to test for intersection with.
- `OPt` = A declared variable, the macro will set this to the intersection point.

- `ONorm` = A declared variable, the macro will set this to the surface normal at the intersection point.

`Extents(Obj, Min, Max)`. This macro is a shortcut for calling both min_extent() and max_extent() to get the corners of the bounding box of an object. It returns these values through the Min and Max parameters.
Parameters:

- `Obj` = The object you are getting the extents of.
- `Min` = A declared variable, the macro will set this to the min_extent of the object.
- `Max` = A declared variable, the macro will set this to the max_extent of the object.

`Center_Object(Object, Axis)`. A shortcut for using the Center_Trans() macro with an object.
Parameters:

- `Object` = The object to be centered.
- `Axis` = See Center_Trans() in the transforms.inc documentation.

`Align_Object(Object, Axis, Pt)`. A shortcut for using the Align_Trans() macro with an object.
Parameters:

- `Object` = The object to be aligned.
- `Axis` = See Align_Trans() in the transforms.inc documentation.
- `Point` = The point to which to align the bounding box of the object.

`Bevelled_Text(Font, String, Cuts, BevelAng, BevelDepth, Depth, Offset, UseMerge)`. This macro attempts to "bevel" the front edges of a text object. It accomplishes this by making an intersection of multiple copies of the text object, each sheared in a different direction. The results are no perfect, but may be entirely acceptable for some purposes. Warning: the object generated may render considerably more slowly than an ordinary text object.
Parameters:

- `Font` = A string specifying the font to use.
- `String` = The text string the object is generated from.
- `Cuts` = The number of intersections to use in bevelling the text. More cuts give smoother results, but take more memory and are slower rendering.
- `BevelAng` = The angle of the bevelled edge.
- `BevelDepth` = The thickness of the bevelled portion.
- `Depth` = The total thickness of the resulting text object.
- `Offset` = The offset parameter for the text object. The z value of this vector will be ignored, because the front faces of all the letters need to be coplanar for the bevelling to work.
- `UseMerge` = Switch between merge (1) and union (0).

`Text_Space(Font, String, Size, Spacing)`. Computes the width of a text string, including "white space", it returns the advance widths of all n letters. Text_Space gives the space a text, or a glyph, occupies in regard to its surroundings.
Parameters:

- `Font` = A string specifying the font to use.
- `String` = The text string the object is generated from.
- `Size` = A scaling value.
- `Spacing` = The amount of space to add between the characters.

`Text_Width(Font, String, Size, Spacing)`. Computes the width of a text string, it returns the advance widths of the first n-1 letters, plus the glyph width of the last letter. Text_Width gives the "physical" width of the text and if you use only one letter the "fysical" width of one glyph.
Parameters:

- `Font` = A string specifying the font to use.
- `String` = The text string the object is generated from.
- `Size` = A scaling value.
- `Spacing` = The amount of space to add between the characters.

`Align_Left, Align_Right, Align_Center`. These constants are used by the `Circle_Text()` macro.

`Circle_Text(Font, String, Size, Spacing, Depth, Radius, Inverted, Justification, Angle)`. Creates a text object with the bottom (or top) of the character cells aligned with all or part of a circle. This macro should be used inside an `object{...}` block.
Parameters:

- `Font` = A string specifying the font to use.
- `String` = The text string the object is generated from.
- `Size` = A scaling value.
- `Spacing` = The amount of space to add between the characters.
- `Depth` = The thickness of the text object.
- `Radius` = The radius of the circle the letters are aligned to.
- `Inverted` = Controls what part of the text faces "outside". If this parameter is nonzero, the tops of the letters will point toward the center of the circle. Otherwise, the bottoms of the letters will do so.
- `Justification` = Align_Left, Align_Right, or Align_Center.
- `Angle` = The point on the circle from which rendering will begin. The +x direction is 0 and the +y direction is 90 (i.e. the angle increases anti-clockwise).

`Wedge(Angle)`. This macro creates an infinite wedge shape, an intersection of two planes. It is mainly useful in CSG, for example to obtain a specific arc of a torus. The edge of the wedge is positioned along the y axis, and one side is fixed to the zy plane,

the other side rotates clockwise around the y axis.
Parameters:

- `Angle` = The angle, in degrees, between the sides of the wedge shape.

`Spheroid(Center, Radius)`. This macro creates an unevenly scaled sphere. Radius is a vector where each component is the radius along that axis.
Parameters:

- `Center` = Center of the spheroid.
- `Radius` = A vector specifying the radii of the spheroid.

`Supertorus(MajorRadius, MinorRadius, MajorControl, MinorControl, Accuracy, MaxGradient)`. This macro creates an isosurface of the torus equivalent of a superellipsoid. If you specify a MaxGradient of less than 1, evaluate will be used. You will have to adjust MaxGradient to fit the parameters you choose, a squarer supertorus will have a higher gradient. You may want to use the function alone in your own isosurface.
Parameters:

- `MajorRadius, MinorRadius` = Base radii for the torus.
- `MajorControl, MinorControl` = Controls for the roundness of the supertorus. Use numbers in the range [0, 1].
- `Accuracy` = The accuracy parameter.
- `MaxGradient` = The max_gradient parameter.

`Supercone(EndA, A, B, EndB, C, D)`. This macro creates an object similar to a cone, but where the end points are ellipses. The actual object is an intersection of a quartic with a cylinder.
Parameters:

- `EndA` = Center of end A.
- `A, B` = Controls for the radii of end A.
- `EndB` = Center of end B.
- `C, D` = Controls for the radii of end B.

`Connect_Spheres(PtA, RadiusA, PtB, RadiusB)`. This macro creates a cone that will smoothly join two spheres. It creates only the cone object, however, you will have to supply the spheres yourself or use the Round_Cone2() macro instead.
Parameters:

- `PtA` = Center of sphere A.
- `RadiusA` = Radius of sphere A.
- `PtB` = Center of sphere B.
- `RadiusB` = Radius of sphere B.

`Wire_Box_Union(PtA, PtB, Radius)`,
`Wire_Box_Merge(PtA, PtB, Radius)`,
`Wire_Box(PtA, PtB, Radius, UseMerge)`. Creates a wire-frame box from cylinders and spheres. The resulting object will fit entirely within a box object with the same

corner points.
Parameters:

- `PtA` = Lower-left-front corner of box.
- `PtB` = Upper-right-back corner of box.
- `Radius` = The radius of the cylinders and spheres composing the object.
- `UseMerge` = Whether or not to use a merge.

`Round_Box_Union(PtA, PtB, EdgeRadius)`,
`Round_Box_Merge(PtA, PtB, EdgeRadius)`,
`Round_Box(PtA, PtB, EdgeRadius, UseMerge)`. Creates a box with rounded edges from boxes, cylinders and spheres. The resulting object will fit entirely within a box object with the same corner points. The result is slightly different from a superellipsoid, which has no truely flat areas.
Parameters:

- `PtA` = Lower-left-front corner of box.
- `PtB` = Upper-right-back corner of box.
- `EdgeRadius` = The radius of the edges of the box.
- `UseMerge` = Whether or not to use a merge.

`Round_Cylinder_Union(PtA, PtB, Radius, EdgeRadius)`,
`Round_Cylinder_Merge(PtA, PtB, Radius, EdgeRadius)`,
`Round_Cylinder(PtA, PtB, Radius, EdgeRadius, UseMerge)`. Creates a cylinder with rounded edges from cylinders and tori. The resulting object will fit entirely within a cylinder object with the same end points and radius. The result is slightly different from a superellipsoid, which has no truely flat areas.
Parameters:

- `PtA, PtB` = The end points of the cylinder.
- `Radius` = The radius of the cylinder.
- `EdgeRadius` = The radius of the edges of the cylinder.
- `UseMerge` = Whether or not to use a merge.

`Round_Cone_Union(PtA, RadiusA, PtB, RadiusB, EdgeRadius)`,
`Round_Cone_Merge(PtA, RadiusA, PtB, RadiusB, EdgeRadius)`,
`Round_Cone(PtA, RadiusA, PtB, RadiusB, EdgeRadius, UseMerge)` Creates a cone with rounded edges from cones and tori. The resulting object will fit entirely within a cone object with the same end points and radii.
Parameters:

- `PtA, PtB` = The end points of the cone.
- `RadiusA, RadiusB` = The radii of the cone.
- `EdgeRadius` = The radius of the edges of the cone.
- `UseMerge` = Whether or not to use a merge.

`Round_Cone2_Union(PtA, RadiusA, PtB, RadiusB)`,
`Round_Cone2_Merge(PtA, RadiusA, PtB, RadiusB)`,
`Round_Cone2(PtA, RadiusA, PtB, RadiusB, UseMerge)`. Creates a cone with rounded edges from a cone and two spheres. The resulting object will not fit entirely within a cone object with the same end points and radii because of the spherical caps. The end points are not used for the conical portion, but for the spheres, a suitable cone is then generated to smoothly join them.
Parameters:

- `PtA, PtB` = The centers of the sphere caps.
- `RadiusA, RadiusB` = The radii of the sphere caps.
- `UseMerge` = Whether or not to use a merge.

`Round_Cone3_Union(PtA, RadiusA, PtB, RadiusB)`,
`Round_Cone3_Merge(PtA, RadiusA, PtB, RadiusB)`
`Round_Cone3(PtA, RadiusA, PtB, RadiusB, UseMerge)`. Like Round_Cone2(), this creates a cone with rounded edges from a cone and two spheres, and the resulting object will not fit entirely within a cone object with the same end points and radii because of the spherical caps. The difference is that this macro takes the end points of the conical portion and moves the spheres to be flush with the surface, instead of putting the spheres at the end points and generating a cone to join them.
Parameters:

- `PtA, PtB` = The end points of the cone.
- `RadiusA, RadiusB` = The radii of the cone.
- `UseMerge` = Whether or not to use a merge.

`Quad(A, B, C, D)` and `Smooth_Quad(A, NA, B, NB, C, NC, D, ND)`. These macros create "quads", 4-sided polygonal objects, using triangle pairs.
Parameters:

- `A, B, C, D` = Vertices of the quad.
- `NA, NB, NC, ND` = Vertex normals of the quad.

The HF Macros

There are several HF macros in shapes.inc, which generate meshes in various shapes. All the HF macros have these things in common:

- The HF macros do not directly use an image for input, but evaluate a user-defined function. The macros deform the surface based on the function values.
- The macros can either write to a file to be included later, or create an object directly. If you want to output to a file, simply specify a filename. If you want to create an object directly, specify "" as the file name (an empty string).
- The function values used for the heights will be taken from the square that goes from <0,0,0> to <1,1,0> if UV height mapping is on. Otherwise the function values will be taken from the points where the surface is (before the deformation).

- The texture you apply to the shape will be evaluated in the square that goes from <0,0,0> to <1,1,0> if UV texture mapping is on. Otherwise the texture is evaluated at the points where the surface is (after the deformation.

The usage of the different HF macros is described below.

`HF_Square (Function, UseUVheight, UseUVtexture, Res, Smooth, FileName, MnExt, MxExt)`. This macro generates a mesh in the form of a square height field, similar to the built-in height_field primitive. Also see the general description of the HF macros above.
Parameters:

- `Function` = The function to use for deforming the height field.
- `UseUVheight` = A boolean value telling the macro whether or not to use UV height mapping.
- `UseUVtexture` = A boolean value telling the macro whether or not to use UV texture mapping.
- `Res` = A 2D vector specifying the resolution of the generated mesh.
- `Smooth` = A boolean value telling the macro whether or not to smooth the generated mesh.
- `FileName` = The name of the output file.
- `MnExt` = Lower-left-front corner of a box containing the height field.
- `MxExt` = Upper-right-back corner of a box containing the height field.

`HF_Sphere(Function, UseUVheight, UseUVtexture, Res, Smooth, FileName, Center, Radius, Depth)`. This macro generates a mesh in the form of a spherical height field. When UV-mapping is used, the UV square will be wrapped around the sphere starting at +x and going anti-clockwise around the y axis. Also see the general description of the HF macros above. Parameters:

- `Function` = The function to use for deforming the height field.
- `UseUVheight` = A boolean value telling the macro whether or not to use UV height mapping.
- `UseUVtexture` = A boolean value telling the macro whether or not to use UV texture mapping.
- `Res` = A 2D vector specifying the resolution of the generated mesh.
- `Smooth` = A boolean value telling the macro whether or not to smooth the generated mesh.
- `FileName` = The name of the output file.
- `Center` = The center of the height field before being displaced, the displacement can, and most likely will, make the object off-center.
- `Radius` = The starting radius of the sphere, before being displaced.
- `Depth` = The depth of the height field.

`HF_Cylinder(Function, UseUVheight, UseUVtexture, Res, Smooth, FileName, EndA, EndB, Radius,Depth)`. This macro generates a mesh in the form of an open-ended cylindrical height field. When UV-mapping is used, the UV square will be wrapped around the cylinder. Also see the general description of the HF macros above.
Parameters:

- `Function` = The function to use for deforming the height field.
- `UseUVheight` = A boolean value telling the macro whether or not to use UV height mapping.
- `UseUVtexture` = A boolean value telling the macro whether or not to use UV texture mapping.
- `Res` = A 2D vector specifying the resolution of the generated mesh.
- `Smooth` = A boolean value telling the macro whether or not to smooth the generated mesh.
- `FileName` = The name of the output file.
- `EndA, EndB` = The end points of the cylinder.
- `Radius` = The (pre-displacement) radius of the cylinder.
- `Depth` = The depth of the height field.

`HF_Torus (Function, UseUVheight, UseUVtexture, Res, Smooth, FileName, Major, Minor, Depth)`. This macro generates a mesh in the form of a torus-shaped height field. When UV-mapping is used, the UV square is wrapped around similar to spherical or cylindrical mapping. However the top and bottom edges of the map wrap over and under the torus where they meet each other on the inner rim. Also see the general description of the HF macros above.
Parameters:

- `Function` = The function to use for deforming the height field.
- `UseUVheight` = A boolean value telling the macro whether or not to use UV height mapping.
- `UseUVtexture` = A boolean value telling the macro whether or not to use UV texture mapping.
- `Res` = A 2D vector specifying the resolution of the generated mesh.
- `Smooth` = A boolean value telling the macro whether or not to smooth the generated mesh.
- `FileName` = The name of the output file.
- `Major` = The major radius of the torus.
- `Minor` = The minor radius of the torus.

6.12.2 shapes_old.inc

```
Ellipsoid, Sphere
```

Unit-radius sphere at the origin.

`Cylinder_X, Cylinder_Y, Cylinder_Z`

Infinite cylinders.

`QCone_X, QCone_Y, QCone_Z`

Infinite cones.

`Cone_X, Cone_Y, Cone_Z`

Closed capped cones: unit-radius at -1 and 0 radius at +1 along each axis.

`Plane_YZ, Plane_XZ, Plane_XY`

Infinite planes passing through the origin.

`Paraboloid_X, Paraboloid_Y, Paraboloid_Z`

y^2 + z^2 - x = 0

`Hyperboloid, Hyperboloid_Y`

y - x^2 + z^2 = 0

`UnitBox, Cube`

A cube 2 units on each side, centered on the origin.

`Disk_X, Disk_Y, Disk_Z`

"Capped" cylinders, with a radius of 1 unit and a length of 2 units, centered on the origin.

6.12.3 shapes2.inc

`Tetrahedron`

4-sided regular polyhedron.

`Octahedron`

8-sided regular polyhedron.

`Dodecahedron`

12-sided regular polyhedron.

`Icosahedron`

20-sided regular polyhedron.

`Rhomboid`

Three dimensional 4-sided diamond, basically a sheared box.

`Hexagon`

6-sided regular polygonal solid, axis along x.

`HalfCone_Y`

Convenient finite cone primitive, pointing up in the Y axis.

`Pyramid`

4-sided pyramid (union of triangles, can not be used in CSG).

`Pyramid2`

4-sided pyramid (intersection of planes, can be used in CSG).

`Square_X, Square_Y, Square_Z`

Finite planes stretching 1 unit along each axis. In other words, 2X2 unit squares.

6.12.4 shapesq.inc

`Bicorn`

This curve looks like the top part of a paraboloid, bounded from below by another paraboloid. The basic equation is:

y^2 - (x^2 + z^2) y^2 - (x^2 + z^2 + 2 y - 1)^2 =

`Crossed_Trough`

This is a surface with four pieces that sweep up from the x-z plane.

The equation is: y = x^2 z^2

`Cubic_Cylinder`

A drop coming out of water? This is a curve formed by using the equation:

y = 1/2 x^2 (x + 1)

as the radius of a cylinder having the x-axis as its central axis. The final form of the equation is:

y^2 + z^2 = 0.5 (x^3 + x^2)

`Cubic_Saddle_1`

A cubic saddle. The equation is: z = x^3 - y^3

`Devils_Curve`

Variant of a devil's curve in 3-space. This figure has a top and bottom part that are very similar to a hyperboloid of one sheet, however the central region is pinched in the middle leaving two teardrop shaped holes. The equation is:

x^4 + 2 x^2 z^2 - 0.36 x^2 - y^4 + 0.25 y^2 + z^4 = 0

`Folium`

This is a folium rotated about the x-axis. The formula is:

2 x^2 - 3 x y^2 - 3 x z^2 + y^2 + z^2 = 0

`Glob_5`

Glob - sort of like basic teardrop shape. The equation is:

y^2 + z^2 = 0.5 x^5 + 0.5 x^4

`Twin_Glob`

Variant of a lemniscate - the two lobes are much more teardrop-like.

`Helix, Helix_1`

Approximation to the helix z = arctan(y/x). The helix can be approximated with an algebraic equation (kept to the range of a quartic) with the following steps:

tan(z) = y/x => sin(z)/cos(z) = y/x =>
(1) x sin(z) - y cos(z) = 0 Using the taylor expansions for sin, cos about z = 0,
sin(z) = z - z^3/3! + z^5/5! - ...
cos(z) = 1 - z^2/2! + z^6/6! - ...
Throwing out the high order terms, the expression (1) can be written as:
x (z - z^3/6) - y (1 + z^2/2) = 0, or

(2) -1/6 x z^3 + x z + 1/2 y z^2 - y = 0
This helix (2) turns 90 degrees in the range 0 <= z <= sqrt(2)/2. By using scale <2 2 2>, the helix defined below turns 90 degrees in the range 0 <= z <= sqrt(2) = 1.4042.

`Hyperbolic_Torus`

Hyperbolic Torus having major radius sqrt(40), minor radius sqrt(12). This figure is generated by sweeping a circle along the arms of a hyperbola. The equation is:

x^4 + 2 x^2 y^2 - 2 x^2 z^2 - 104 x^2 + y^4 - 2 y^2 z^2 + 56 y^2 + z^4 + 104 z^2 + 784 = 0

`Lemniscate`

Lemniscate of Gerono. This figure looks like two teardrops with their pointed ends connected. It is formed by rotating the Lemniscate of Gerono about the x-axis. The formula is:

x^4 - x^2 + y^2 + z^2 = 0

`Quartic_Loop_1`

This is a figure with a bumpy sheet on one side and something that looks like a paraboloid (but with an internal bubble). The formula is:

(x^2 + y^2 + a c x)^2 - (x^2 + y^2)(c - a x)^2

-99*x^4+40*x^3-98*x^2*y^2-98*x^2*z^2+99*x^2+40*x*y^2

+40*x*z^2+y^4+2*y^2*z^2-y^2+z^4-z^2

`Monkey_Saddle`

This surface has three parts that sweep up and three down. This gives a saddle that has a place for two legs and a tail... The equation is:

```
z = c (x^3 - 3 x y^2)
```

The value c gives a vertical scale to the surface - the smaller the value of c, the flatter the surface will be (near the origin).

`Parabolic_Torus_40_12`

Parabolic Torus having major radius sqrt(40), minor radius sqrt(12). This figure is generated by sweeping a circle along the arms of a parabola. The equation is:

$x^4 + 2 x^2 y^2 - 2 x^2 z - 104 x^2 + y^4 - 2 y^2 z + 56 y^2 + z^2 + 104 z + 784 = 0$

`Piriform`

This figure looks like a hersheys kiss. It is formed by sweeping a Piriform about the x-axis. A basic form of the equation is:

$(x^4 - x^3) + y^2 + z^2 = 0.$

`Quartic_Paraboloid`

Quartic parabola - a 4th degree polynomial (has two bumps at the bottom) that has been swept around the z axis. The equation is:

$0.1 x^4 - x^2 - y^2 - z^2 + 0.9 = 0$

`Quartic_Cylinder`

Quartic Cylinder - a Space Needle?

`Steiner_Surface`

Steiners quartic surface

`Torus_40_12`

Torus having major radius sqrt(40), minor radius sqrt(12).

`Witch_Hat`

Witch of Agnesi.

`Sinsurf`

Very rough approximation to the sin-wave surface z = sin(2 pi x y).

In order to get an approximation good to 7 decimals at a distance of 1 from the origin would require a polynomial of degree around 60, which would require around 200,000 coefficients. For best results, scale by something like <1 1 0.2>.

6.13 skies.inc, stars.inc

These files contain some predefined skies for you to use in your scenes.

skies.inc:
There are textures and pigment definitions in this file. All pigment definitions start with

"P_", all sky_spheres start with "S_", all textures start with "T_", and all objects start with "O_".

stars.inc:
This file contains predefined starfield textures. The starfields become denser and more colorful with the number, with Starfield6 being the densest and most colorful.

6.13.1 skies.inc

Pigments:

`P_Cloud1`

`P_Cloud2`

`P_Cloud3`

Sky Spheres:

`S_Cloud1`

This sky_sphere uses P_Cloud2 and P_Cloud3.

`S_Cloud2`

This sky_sphere uses P_Cloud4.

`S_Cloud3`

This sky_sphere uses P_Cloud2.

`S_Cloud4`

This sky_sphere uses P_Cloud3.

`S_Cloud5`

This sky_sphere uses a custom pigment.

Textures:

`T_Cloud1`

2-layer texture using P_Cloud1 pigment, contains clear regions.

`T_Cloud2`

1-layer texture, contains clear regions.

`T_Cloud3`

2-layer texture, contains clear regions.

Objects:

`O_Cloud1`

Sphere, radius 10000 with T_Cloud1 texture.

`O_Cloud2`

Union of 2 planes, with T_Cloud2 and T_Cloud3.

6.13.2 stars.inc

`Starfield1`

`Starfield2`

`Starfield3`

`Starfield4`

`Starfield5`

`Starfield6`

6.14 stones.inc, stones1.inc, stones2.inc, stoneold.inc

The two files stones1.inc and stones2.inc contain lists of predefined stone textures.

The file ”stones1.inc” contains texture definitions for T_Grnt0 to T_Grnt29, T_Grnt1a to T_Grnt24a, and T_Stone0 to T_Stone24.

The T_GrntXX, T_GrntXXa, and CrackX textures are ”building blocks that are used to create the final ”usable” T_StoneX textures (and other textures that *you* design, of course!)

The T_GrntXX textures generally contain no transparency, but the T_GrntXXa textures do contain transparency. The CrackX textures are clear with thin opaque bands, simulating cracks.

The file ”stones2.inc” provides additional stone textures, and contains texture definitions for T_Stone25 to T_Stone44.

The file ”stones.inc” simply includes both ”stones1.inc” and ”stones2.inc”, and the file ”stoneold.inc” provides backwards compatability for old scenes, the user is advised to use the textures in ”stones1.inc” instead.

6.14.1 stones1.inc

`T_Grnt0`

Gray/Tan with Rose.

`T_Grnt1`

Creamy Whites with Yellow & Light Gray.

`T_Grnt2`

Deep Cream with Light Rose, Yellow, Orchid, & Tan.

`T_Grnt3`

Warm tans olive & light rose with cream.

`T_Grnt4`

Orchid, Sand & Mauve.

T_Grnt5

Medium Mauve Med.Rose & Deep Cream.

T_Grnt6

Med. Orchid, Olive & Dark Tan ”mud pie”.

T_Grnt7

Dark Orchid, Olive & Dark Putty.

T_Grnt8

Rose & Light Cream Yellows

T_Grnt9

Light Steely Grays

T_Grnt10

Gray Creams & Lavender Tans

T_Grnt11

Creams & Grays Kahki

T_Grnt12

Tan Cream & Red Rose

T_Grnt13

Cream Rose Orange

T_Grnt14

Cream Rose & Light Moss w/Light Violet

T_Grnt15

Black with subtle chroma

T_Grnt16

White Cream & Peach

T_Grnt17

Bug Juice & Green

T_Grnt18

Rose & Creamy Yellow

T_Grnt19

Gray Marble with White feather Viens

T_Grnt20

White Marble with Gray feather Viens

`T_Grnt21`

Green Jade

`T_Grnt22`

Clear with White feather Viens (has some transparency)

`T_Grnt23`

Light Tan to Mauve

`T_Grnt24`

Light Grays

`T_Grnt25`

Moss Greens & Tan

`T_Grnt26`

Salmon with thin Green Viens

`T_Grnt27`

Dark Green & Browns

`T_Grnt28`

Red Swirl

`T_Grnt29`

White, Tan, w/ thin Red Viens

`T_Grnt0a`

Translucent T_Grnt0

`T_Grnt1a`

Translucent T_Grnt1

`T_Grnt2a`

Translucent T_Grnt2

`T_Grnt3a`

Translucent T_Grnt3

`T_Grnt4a`

Translucent T_Grnt4

`T_Grnt5a`

Translucent T_Grnt5

`T_Grnt6a`

Translucent T_Grnt6

`T_Grnt7a`

Translucent T_Grnt7

`T_Grnt8a`

Aqua Tints

`T_Grnt9a`

Transmit Creams With Cracks

`T_Grnt10a`

Transmit Cream Rose & light yellow

`T_Grnt11a`

Transmit Light Grays

`T_Grnt12a`

Transmit Creams & Tans

`T_Grnt13a`

Transmit Creams & Grays

`T_Grnt14a`

Cream Rose & light moss

`T_Grnt15a`

Transmit Sand & light Orange

`T_Grnt16a`

Cream Rose & light moss (again?)

`T_Grnt17a`

???

`T_Grnt18a`

???

`T_Grnt19a`

Gray Marble with White feather Viens with Transmit

`T_Grnt20a`

White Feather Viens

`T_Grnt21a`

Thin White Feather Viens

`T_Grnt22a`

???

`T_Grnt23a`

Transparent Green Moss

T_Grnt24a

???

T_Crack1

T_Crack & Red Overtint

T_Crack2

Translucent Dark T_Cracks

T_Crack3

Overtint Green w/ Black T_Cracks

T_Crack4

Overtint w/ White T_Crack

The StoneXX textures are the complete textures, ready to use.

T_Stone1

Deep Rose & Green Marble with large White Swirls

T_Stone2

Light Greenish Tan Marble with Agate style veining

T_Stone3

Rose & Yellow Marble with fog white veining

T_Stone4

Tan Marble with Rose patches

T_Stone5

White Cream Marble with Pink veining

T_Stone6

Rose & Yellow Cream Marble

T_Stone7

Light Coffee Marble with darker patches

T_Stone8

Gray Granite with white patches

T_Stone9

White & Light Blue Marble with light violets

T_Stone10

Dark Brown & Tan swirl Granite with gray undertones

T_Stone11

Rose & White Marble with dark tan swirl

T_Stone12

White & Pinkish Tan Marble

T_Stone13

Medium Gray Blue Marble

T_Stone14

Tan & Olive Marble with gray white veins

T_Stone15

Deep Gray Marble with white veining

T_Stone16

Peach & Yellow Marble with white veining

T_Stone17

White Marble with gray veining

T_Stone18

Green Jade with white veining

T_Stone19

Peach Granite with white patches & green trim

T_Stone20

Brown & Olive Marble with white veining

T_Stone21

Red Marble with gray & white veining

T_Stone22

Dark Tan Marble with gray & white veining

T_Stone23

Peach & Cream Marble with orange veining

T_Stone24

Green & Tan Moss Marble

6.14.2 stones2.inc

T_Stone25, ..., T_Stone44

6.15 stdinc.inc

This file simply includes the most commonly used include files, so you can get all of them with a single #include. The files included are:

- colors.inc
- shapes.inc
- transforms.inc
- consts.inc
- functions.inc
- math.inc
- rand.inc

6.16 strings.inc

This include contains macros for manipulating and generating text strings.

`CRGBStr(C, MinLen, Padding)` and `CRGBFTStr(C, MinLen, Padding)`
These macros convert a color to a string. The format of the output string is ”rgb < R, G, B>” or ”rgbft < R, G, B, F, T>”, depending on the macro being called.
Parameters:

- `C` = The color to be turned into a string.
- `MinLen` = The minimum length of the individual components, analogous to the second parameter of str().
- `Padding` = The padding to use for the components, see the third parameter of the str() function for details.

`Str(A)`. This macro creates a string containing a float with the systems default precision. It is a shortcut for using the str() function.
Parameters:

- `A` = The float to be converted to a string.

`VStr2D(V), VStr(V)`. These macros create strings containing vectors using POV syntax (<X,Y,Z>) with the default system precision. VStr2D() works with 2D vectors, VStr() with 3D vectors. They are shortcuts for using the `vstr()` function.
Parameters:

- `V` = The vector to be converted to a string.

`Vstr2D(V,L,P), Vstr(V,L,P)`. These macros create strings containing vectors using POV syntax (<X,Y,Z>) with user specified precision. Vstr2D() works with 2D vectors, Vstr() with 3D vectors. They are shortcuts for using the vstr() function. The function of L and P is the same as in `vstr` specified in String Functions.
Parameters:

- `V` = The vector to be converted to a string.

- `L` = Minimum length of the string and the type of left padding used if the string's representation is shorter than the minimum.
- `P` = Number of digits after the decimal point."

`Triangle_Str(A, B, C)` and `Smooth_Triangle_Str(A, NA, B, NB, C, NC)`
These macros take vertex and normal information and return a string representing a triangle in POV-Ray syntax. They are mainly useful for generating mesh files.
Parameters:

- `A, B, C` = Triangle vertex points.
- `NA, NB, NC` = Triangle vertex normals (Smooth_Triangle_Str() only).

`Parse_String(String)`. This macro takes a string, writes it to a file, and then includes that file. This has the effect of parsing that string: "`Parse_String("MyColor")`" will be seen by POV-Ray as "`MyColor`".
Parameters:

- `String` = The string to be parsed.

6.17 textures.inc

This file contains many predefined textures, including wood, glass, and metal textures, and a few texture/pattern generation macros.

6.17.1 Stones

Stone Pigments:

`Jade_Map, Jade`

Drew Wells' superb Jade. Color map works nicely with other textures, too.

`Red_Marble_Map, Red_Marble`

Classic white marble with red veins. Over-worked, like checkers.

`White_Marble_Map, White_Marble`

White marble with black veins.

`Blood_Marble_Map, Blood_Marble`

Light blue and black marble with a thin red vein.

`Blue_Agate_Map, Blue_Agate`

A grey blue agate – kind of purplish.

`Sapphire_Agate_Map, Sapphire_Agate`

Deep blue agate – almost glows.

`Brown_Agate_Map, Brown_Agate`

Brown and white agate – very pretty.

`Pink_Granite_Map, Pink_Granite`

Umm, well, pink granite.

Stone textures:

`PinkAlabaster`

Gray-pink alabaster or marble. Layers are scaled for a unit object and relative to each other.

Note: This texture has very tiny dark blue specks that are often mistaken for rendering errors. They are not errors. Just a strange texture design.

Underlying surface is very subtly mottled with bozo.

Second layer texture has some transmit values, yet a fair amount of color.

Veining is kept quite thin in color map and by the largish scale.

6.17.2 Skies

Sky pigments:

`Blue_Sky_Map, Blue_Sky`

Basic blue sky with clouds.

`Bright_Blue_Sky`

Bright blue sky with very white clouds.

`Blue_Sky2`

Another sky.

`Blue_Sky3`

Small puffs of white clouds.

`Blood_Sky`

Red sky with yellow clouds – very surreal.

`Apocalypse`

Black sky with red and purple clouds.

Try adding turbulence values from 0.1 - 5.0 – CdW

`Clouds`

White clouds with transparent sky.

`FBM_Clouds`

`Shadow_Clouds`

A multilayered cloud texture (a real texture, not a pigment).

6.17.3 Woods

Wood pigments:

Several wooden pigments by Tom Price:

`Cherry_Wood`

A light reddish wood.

`Pine_Wood`

A light tan wood whiteish rings.

`Dark_Wood`

Dark wood with a,ish hue to it.

`Tan_Wood`

Light tan wood with brown rings.

`White_Wood`

A very pale wood with tan rings – kind of balsa-ish.

`Tom_Wood`

Brown wood - looks stained.

`DMFWood1, DMFWood2, DMFWood3, DMFWood4, DMFWood5`

The scaling in these definitions is relative to a unit-sized object (radius 1).

> **Note:** woods are functionally equivalent to a log lying along the z axis. For best results, think like a woodcutter trying to extract the nicest board out of that log. A little tilt along the x axis will give elliptical rings of grain like you'd expect to find on most boards. Experiment.

Wood textures:

`DMFWood6`

This is a three-layer wood texture. Renders rather slowly because of the transparent layers and the two layers of turbulence, but it looks great. Try other colors of "varnish" for simple variations.

`DMFLightOak`

Is this really oak? I dunno. Quite light, maybe more like spruce.

`DMFDarkOak`

Looks like old desk oak if used correctly.

`EMBWood1`

Wood by Eric Barish

Doug Otwell woods:

`Yellow_Pine`

Yellow pine, close grained.

`Rosewood`

`Sandalwood`

makes a great burled maple, too

6.17.4 Glass

`Glass_Finish` is a generic glass finish, `Glass_Interior` is a generic glass interior, it just adds an ior of 1.5.

Glass materials:

`M_Glass`

Just glass.

`M_Glass2`

Probably more of a ”Plexiglas” than glass.

`M_Glass3`

An excellent lead crystal glass!

`M_Green_Glass`

Glass textures contributed by Norm Bowler, of Richland WA. NBglass_finish is used by these materials.

`M_NBglass`

`M_NBoldglass`

`M_NBwinebottle`

`M_NBbeerbottle`

A few color variations on Norm's glass.

`M_Ruby_Glass`

`M_Dark_Green_Glass`

`M_Yellow_Glass`

`M_Orange_Glass`

`M_Vicks_Bottle_Glass`

6.17.5 Metals

Metal finishes:

`Metal`

Generic metal finish.

`SilverFinish`

Basic silver finish

`Metallic_Finish`

`Chrome_Metal, Brass_Metal, Bronze_Metal, Gold_Metal, Silver_Metal, Copper_Metal`

A series of metallic textures using the Metal finish (except for Chrome_Metal, which has a custom finish). There are identical textures ending in _Texture instead of _Metal, but use of those names is discouraged.

`Polished_Chrome`

A highly reflective Chrome texture.

`Polished_Brass`

A highly reflective brass texture.

`New_Brass`

Beautiful military brass texture!

`Spun_Brass`

Spun Brass texture for cymbals & such

`Brushed_Aluminum`

Brushed aluminum (brushed along X axis)

`Silver1`

`Silver2`

`Silver3`

`Brass_Valley`

Sort of a ”Black Hills Gold”, black, white, and orange specks or splotches.

`Rust`

`Rusty_Iron`

`Soft_Silver`

`New_Penny`

`Tinny_Brass`

`Gold_Nugget`

`Aluminum`

`Bright_Bronze`

6.17.6 Special textures

`Candy_Cane`

Red & white stripes - Looks best on a y axis Cylinder.

It "spirals" because it's gradient on two axis.

`Peel`

Orange and Clear stripes spiral around the texture to make an object look like it was "Peeled". Now, you too can be M.C. Escher!

`Y_Gradient`

`X_Gradient`

`M_Water`

Wavy water material. Requires a sub-plane, and may require scaling to fit your scene.

WARNING: Water texture has been changed to M_Water material, see explanation in the "glass" section of this file.

`Cork`

`Lightning_CMap1, Lightning1, and Lightning_CMap2, Lightning2`

These are just lightning textures, they look like arcing electricity...earlier versions misspelled them as "Lightening".

`Starfield`

A starfield texture by Jeff Burton

6.17.7 Texture and pattern macros

`Irregular_Bricks_Ptrn (Mortar Thickness, X-scaling, Variation, Roundness)`. This function pattern creates a pattern of bricks of varying lengths on the x-y plane. This can be useful in building walls that don't look like they were built by a computer. Note that mortar thickness between bricks can vary somewhat, too.
Parameters:

- `Mortar Thickness` = Thickness of the mortar (0-1).
- `X-scaling` = The scaling of the bricks (but not the mortar) in the x direction.
- `Variation` = The amount by which brick lengths will vary (0=none, 1=100%).
- `Roundness` = The roundness of the bricks (0.01=almost rectangular, 1=very round).

`Tiles_Ptrn()`. This macro creates a repeating box pattern on the x-y plane. It can be useful for creating grids. The cells shade continuously from the center to the edges.
Parameters: None.

`Hex_Tiles_Ptrn()`. This macro creates a pattern that is a sort of cross between the hexagon pattern and a repeating box pattern. The hexagonal cells shade continuously from the center to the edges.
Parameters: None.

`Star_Ptrn (Radius, Points, Skip)`. This macro creates a pattern that resembles a star. The pattern is in the x-y plane, centered around the origin.
Parameters:

- Radius = The radius of a circle drawn through the points of the star.
- Points = The number of points on the star.
- Skip = The number of points to skip when drawing lines between points to form the star. A normal 5-pointed star skips 2 points. A Star of David also skips 2 points. Skip must be less than Points/2 and greater than 0. Integers are preferred but not required. Skipping 1 point makes a regular polygon with Points sides.
- Pigment = The pigment to be applied to the star.
- Background = The pigment to be applied to the background.

6.18 transforms.inc

Several useful transformation macros. All these macros produce transformations, you can use them anywhere you can use scale, rotate, etc. The descriptions will assume you are working with an object, but the macros will work fine for textures, etc.

`Shear_Trans(A, B, C)`. This macro reorients and deforms an object so its original XYZ axes point along A, B, and C, resulting in a shearing effect when the vectors are not perpendicular. You can also use vectors of different lengths to affect scaling, or use perpendicular vectors to reorient the object.
Parameters:

- A, B, C = Vectors representing the new XYZ axes for the transformation.

`Matrix_Trans(A, B, C, D)`. This macro provides a way to specify a matrix transform with 4 vectors. The effects are very similar to that of the Shear_Trans() macro, but the fourth parameter controls translation.
Parameters:

- A, B, C, D = Vectors for each row of the resulting matrix.

`Axial_Scale_Trans(Axis, Amt)`. A kind of directional scale, this macro will ”stretch” an object along a specified axis.
Parameters:

- Axis = A vector indicating the direction to stretch along.
- Amt = The amount to stretch.

`Axis_Rotate_Trans(Axis, Angle)`. This is equivalent to the transformation done by the vaxis_rotate() function, it rotates around an arbitrary axis.
Parameters:

- Axis = A vector representing the axis to rotate around.
- Angle = The amount to rotate by.

`Rotate_Around_Trans(Rotation, Point)`. Ordinary rotation operates around the origin, this macro rotates around a specific point.
Parameters:

- Rotation = The rotation vector, the same as the parameter to the rotate keyword.

- `Point` = The point to rotate around.

`Reorient_Trans(Axis1, Axis2)`. This aligns `Axis1` to `Axis2` by rotating the object around a vector perpendicular to both axis1 and axis2.
Parameters:

- `Axis1` = Vector to be rotated.
- `Axis2` = Vectors to be rotated towards.

`Point_At_Trans(YAxis)`. This macro is similar to Reorient_Trans(), but it points the y axis along Axis.
Parameters:

- `YAxis` = The direction to point the y axis in.

`Center_Trans(Object, Axis)`. Calculates a transformation which will center an object along a specified axis. You indicate the axes you want to center along by adding ”x”, ”y”, and ”z” together in the Axis parameter.

Note: this macro actually computes the transform to center the bounding box of the object, which may not be entirely accurate. There is no way to define the ”center” of an arbitrary object.

Parameters:

- `Object` = The object the center transform is being computed for.
- `Axis` = The axes to center the object on.

```
object {MyObj Center_Trans(MyObj, x)} //center along x axis
```

```
object {MyObj Center_Trans(MyObj, x+y)} //center along x and y axis
```

`Align_Trans(Object, Axis, Pt)`. Calculates a transformation which will align the sides of the bounding box of an object to a point. Negative values on Axis will align to the sides facing the negative ends of the coordinate system, positive values will align to the opposite sides, 0 means not to do any alignment on that axis.
Parameters:

- `Object` = The object being aligned.
- `Axis` = A combination of +x, +y, +z, -x, -y, and -z, or a vector where each component is -1, 0, or +1 specifying the faces of the bounding box to align to the point.
- `Point` = The point to which to align the bounding box of the object.

```
object {
   MyObj
   Align_Trans(MyObj, x, Pt) //Align right side of object to be
                             //coplanar with Pt
   Align_Trans(MyObj,-y, Pt) //Align bottom of object to be
                             // coplanar with Pt
}
```

vtransform(Vect, Trans) and vinv_transform(Vect, Trans).
The vtransform() macro takes a transformation (rotate, scale, translate, etc...) and a point, and returns the result of applying the transformation to the point. The vinv_transform() macro is similar, but applies the inverse of the transform, in effect "undoing" the transformation. You can combine transformations by enclosing them in a transform block.
Parameters:

- Vect = The vector to which to apply the transformation.
- Trans = The transformation to apply to Vect.

Spline_Trans(Spline, Time, SkyVector, ForeSight, Banking). This macro aligns an object to a spline for a given time value. The Z axis of the object will point in the forward direction of the spline and the Y axis of the object will point upwards.
Parameters:

- Spline = The spline that the object is aligned to.
- Time = The time value to feed to the spline, for example clock.
- Sky = The vector that is upwards in your scene, usually y.
- Foresight = A positive value that controls how much in advance the object will turn and bank. Values close to 0 will give precise results, while higher values give smoother results. It will not affect parsing speed, so just find the value that looks best.
- Banking = How much the object tilts when turning. The amount of tilting is equally much controlled by the ForeSight value.

```
object {MyObj Spline_Trans(MySpline, clock, y, 0.1, 0.5)}
```

6.19 woodmaps.inc, woods.inc

The file woodmaps.inc contains color_maps designed for use in wood textures. The M_WoodXA maps are intended to be used in the first layer of a multilayer texture, but can be used in single-layer textures. The M_WoodXB maps contain transparent areas, and are intended to be used in upper texture layers.

The file woods.inc contains predefined wood textures and pigments.

The pigments are prefixed with P_, and do not have color_maps, allowing you to specify a color map from woodmaps.inc or create your own. There are two groups, "A" and "B": the A series is designed to work better on the bottom texture layer, and the B series is designed for the upper layers, with semitransparent color maps. The pigments with the same number were designed to work well together, but you don't necessarily have to use them that way.

The textures are prefixed with T_, and are ready to use. They are designed with the major axis of the woodgrain "cylinder" aligned along the Z axis. With the exception of the few of the textures which have a small amount of rotation built-in, the textures will exhibit a very straight grain pattern unless you apply a small amount of x-axis rotation to them (generally 2 to 4 degrees seems to work well).

6.19.1 woodmaps.inc

Color maps:

`M_Wood1A, ..., M_Wood19A`

`M_Wood1B, ..., M_Wood19B`

6.19.2 woods.inc

Pigments:

`P_WoodGrain1A, ..., P_WoodGrainA`

`P_WoodGrain1B, ..., P_WoodGrainB`

Textures:

`T_Wood1`
: Natural oak (light)

`T_Wood2`
: Dark brown

`T_Wood3`
: Bleached oak (white)

`T_Wood4`
: Mahogany (purplish-red)

`T_Wood5`
: Dark yellow with reddish overgrain

`T_Wood6`
: Cocabola (red)

`T_Wood7`
: Yellow pine (ragged grain)

`T_Wood8`
: Dark brown. Walnut?

`T_Wood9`
: Yellowish-brown burl (heavily turbulated)

`T_Wood10`
: Soft pine (light yellow, smooth grain)

`T_Wood11`
: Spruce (yellowish, very straight, fine grain)

`T_Wood12`

Another very dark brown. Walnut-stained pine, perhaps?

`T_Wood13`

Very straight grained, whitish

`T_Wood14`

Red, rough grain

`T_Wood15`

Medium brown

`T_Wood16`

Medium brown

`T_Wood17`

Medium brown

`T_Wood18`

Orange

`T_Wood19, ..., T_Wood30`

Golden Oak.

`T_Wood31`

A light tan wood - heavily grained (variable coloration)

`T_Wood32`

A rich dark reddish wood, like rosewood, with smooth-flowing grain

`T_Wood33`

Similar to T_WoodB, but brighter

`T_Wood34`

Reddish-orange, large, smooth grain.

`T_Wood35`

Orangish, with a grain more like a veneer than a plank

6.20 Other files

There are various other files in the include collection, including font files, color maps, and images for use in height fields or image_maps, and includes that are too small to have their own section.

6.20.1 logo.inc

The official POV-Ray logo designed by Chris Colefax, in two versions

`Povray_Logo`

The POV-Ray logo object

`Povray_Logo_Prism`

The POV-Ray logo as a prism

`Povray_Logo_Bevel`

The POV-Ray logo as a beveled prism

6.20.2 rad_def.inc

This file defines a macro that sets some common radiosity settings. These settings are extremely general and are intended for ease of use, and don't necessarily give the best results.

Usage:

```
#include "rad_def.inc"
global_settings {
   ...
   radiosity {
      Rad_Settings(Setting, Normal, Media)
   }
}
```

Parameters:

- Setting = Quality setting. Use one of the predefined constants:
  - Radiosity_Default
  - Radiosity_Debug
  - Radiosity_Fast
  - Radiosity_Normal
  - Radiosity_2Bounce
  - Radiosity_Final
  - Radiosity_OutdoorLQ
  - Radiosity_OutdoorHQ
  - Radiosity_OutdoorLight
  - Radiosity_IndoorLQ
  - Radiosity_IndoorHQ

- Normal = Boolean value, whether or not to use surface normal modifiers for radiosity samples.
- Media = Boolean value, whether or not to calculate media for radiosity samples.

6.20.3 screen.inc

Screen.inc will enable you to place objects and textures right in front of the camera. When you move the camera, the objects placed with screen.inc will follow the movement and stay in the same position on the screen. One use of this is to place your signature or a logo in the corner of the image.

You can only use screen.inc with the perspective camera. Screen.inc will automatically create a default camera definition for you when it is included. All aspects of the camera can than be changed, by invoking the appropriate 'Set_Camera_...' macros in your scene. After calling these setup macros you can use the macros Screen_Object and Screen_Plane.

Note: even though objects aligned using screen.inc follow the camera, they are still part of the scene. That means that they will be affected by perspective, lighting, the surroundings etc.

For an example of use, see the screen.pov demo file.

`Set_Camera_Location(Loc)` Changes the position of the default camera to a new location as specified by the `Loc` vector.

`Set_Camera_Look_At(LookAt)` Changes the position the default camera looks at to a new location as specified by the `LookAt` vector.

`Set_Camera_Aspect_Ratio(Aspect)` Changes the default aspect ratio, `Aspect` is a float value, usually width divided by the height of the image.

`Set_Camera_Aspect(Width,Height)` Changes the default aspect ratio of the camera.

`Set_Camera_Sky(Sky)` Sets a new Sky-vector for the camera.

`Set_Camera_Zoom(Zoom)` The amount to zoom in or out, `Zoom` is a float.

`Set_Camera_Angle(Angle)` Sets a new camera angle.

`Set_Camera(Location, LookAt, Angle)` Set `location`, `look_at` and `angle` in one go.

`Reset_Camera()` Resets the camera to its default values.

`Screen_Object (Object, Position, Spacing, Confine, Scaling)` Puts an object in front of the camera.
Parameters:

- `Object` = The object to place in front of the screen.
- `Position` = UV coordinates for the object. <0,0> is lower left corner of the screen and <1,1> is upper right corner.
- `Spacing` = Float describing minimum distance from object to the borders. UV vector can be used to get different horizontal and vertical spacing.

- `Confine` = Set to true to confine objects to visible screen area. Set to false to allow objects to be outside visible screen area.
- `Scaling` = If the object intersects or interacts with the scene, try to move it closer to the camera by decreasing Scaling.

`Screen_Plane (Texture, Scaling, BLCorner, TRCorner)` Screen_Plane is a macro that will place a texture of your choice on a plane right in front of the camera.
Parameters:

- `Texture` = The texture to be displayed on the camera plane. <0,0,0> is lower left corner and <1,1,0> is upper right corner.
- `Scaling` = If the plane intersects or interacts with the scene, try to move it closer to the camera by decreasing Scaling.
- `BLCorner` = The bottom left corner of the Screen_Plane.
- `TRCorner` = The top right corner of the Screen_Plane.

6.20.4 stdcam.inc

This file simply contains a camera, a light_source, and a ground plane.

6.20.5 stage1.inc

This file simply contains a camera, a light_source, and a ground plane, and includes colors.inc, textures.inc, and shapes.inc.

6.20.6 sunpos.inc

This file only contains the sunpos() macro

`sunpos(Year, Month, Day, Hour, Minute, Lstm, LAT, LONG)`. The macro returns the position of the sun, for a given date, time, and location on earth. The suns position is also globally declared as the vector `SolarPosition`. Two other declared vectors are the `Az` (Azimuth) and `Al` (Altitude), these can be useful for aligning an object (media container) with the sunlight.
Assumption: in the scene north is in the +Z direction, south is -Z.
Parameters:

- `Year`= The year in four digits.
- `Month`= The month number (1-12).
- `Day`= The day number (1-31).
- `Hour`= The hour of day in 24 hour format (0-23).
- `Minute`= The minutes (0-59).
- `Lstm`= Meridian of your local time zone in degrees (+1 hour = +15 deg, east = positive, west = negative)

- `LAT`= Lattitude in degrees.decimal, northern hemisphere = positive, southern = negative
- `LONG`= Longitude in degrees.decimal, east = positive, west is negative

Use :

```
#include "sunpos.inc"

light_source {
   //Greenwich, noon on the longest day of 2000
   SunPos(2000, 6, 21, 12, 2, 0, 51.4667, 0.00)
   rgb 1
}

cylinder{
   <-2,0,0>,<2,0,0>,0.1
   rotate <0, Az-90, Al>  //align cylinder with sun
   texture {...}
}
```

6.20.7 font files (*.ttf)

The fonts cyrvetic.ttf and timrom.ttf were donated to the POV-Team by their creator, Ted Harrison (CompuServe:70220,344) and were built using his FontLab for Windows by SoftUnion, Ltd. of St. Petersburg, Russia.

The font crystal.ttf was donated courtesy of Jerry Fitzpatrick, Red Mountain Corporation, redmtn@ix.netcom.com

The font povlogo.ttf is created by Fabien Mosen and based on the POV-Ray logo design by Chris Colefax.

`crystal.ttf`

A fixed space programmer's font.

`cyrvetic.ttf`

A proportional spaces sans-serif font.

`timrom.ttf`

A proportional spaces serif font.

`povlogo.ttf`

Only contains the POV-Ray logo.

6.20.8 color_map files (*.map)

These are 255-color color_maps, and are in individual files because of their size.

`ash.map`

`benediti.map`

`bubinga.map`

`cedar.map`

`marbteal.map`

`orngwood.map`

`pinkmarb.map`

`rdgranit.map`

`teak.map`

`whiteash.map`

6.20.9 image files (*.png, *.pot, *.df3)

`bumpmap_.png`

A color mandelbrot fractal image, presumably intended for use as a bumpmap.

`fract003.png`

Some kind of fractal landscape, with color for blue water, brown land, and white peaks.

`maze.png`

A maze.

`mtmand.pot`

A grayscale mandelbrot fractal.

`mtmandj.png`

A 2D color julia fractal.

`plasma2.png, plasma3.png`

”Plasma fractal” images, mainly useful for landscape height fields. The file plasma3.png is a smoother version of plasma2.png, plasma1.png does not exist.

`povmap.png`

The text ”Persistance of Vision” in green on a blue background, framed in black and red.

`test.png`

A ”test image”, the image is divided into 4 areas of different colors (magenta, yellow, cyan, red) with black text on them, and the text ”POV-Ray” is centered on the image in white.

`spiral.df3`

A 3D bitmap density file. A spiral, ”galaxy” shape.

Chapter 7

Quick Reference

This is a consolidation of the entire syntax for the POV-Ray's Scene Description Language. Note that the syntax conventions used here are slightly different than those used in the user documentation.

The following syntax conventions are used:

`ITEM`

An item not in brackets indicates that it is a required item.

`[ITEM]`

Brackets surround an optional item. If brackets are part of the item, that is noted where applicable.

`ITEM...`

An ellipsis indicates an item that may be used one or more times.

`[ITEM...]`

An ellipsis within brackets indicates an item that may be used zero or more times.

`ITEM ITEM`

Two or more juxtaposed items indicates that they should be used in the given order.

`ITEM | ITEM`

A pipe separates two or more alternatives from which only one item should be used.

`ITEM & ITEM`

An ampersand separates two or more items that may be used in any order.

Juxtaposition has precedence over the pipe or ampersand. In the following example, you would select one of the keyword and vector pairs. For that last pair, the keyword itself is optional.

rgb 3D_VECTOR — **rgbf** 4D_VECTOR — **rgbt** 4D_VECTOR — [**rgbft**] 5D_VECTOR

Some item names are simply descriptive in nature. An indication of the item's type is given by a prefix on the item name, as follows:

`F_`

A FLOAT item

`I_`

An INT item

`V_`

A VECTOR item

`V4_`

A 4-D VECTOR item

NOTE: this document provides only the syntax of the Scene Description Language (SDL). The intent is to provide a single reference for all statements and keywords. It does not provide definitions for the numerous keywords nor explain their usage.

7.1 Quick Reference Contents

7.2 The Scene

Describe a POV-Ray scene:

SCENE:

```
SCENE_ITEM...
```

SCENE_ITEM:

```
LANGUAGE_DIRECTIVE | CAMERA | LIGHT | OBJECT | ATMOSPHERIC_EFFECT | GLOBAL_SETTINGS
```

Quick Reference Contents

7.3 Language Basics

7.3.1 Floats

Float Expressions

FLOAT:

```
NUMERIC_TERM [SIGN NUMERIC_TERM]...
```

SIGN:

```
+ | -
```

The Scene	Object Modifiers
Language Basics	UV Mapping
Floats	Material
Vectors	Interior
Colors	Interior Texture
User-Defined Functions	Texture
Strings	Plain Texture
Arrays	Layered Texture
Splines	Patterned Texture
Language Directives	Pigment
File Inclusion	Normal
Identifier Declaration	Finish
File Input/Output	Pattern
Default Texture	Pattern Modifiers
Version Compatibility	Media
Conditional Directives	Atmospheric Effects
Message Streams	Background
Macros	Fog
Embedded Directives	Sky Sphere
Transformations	Rainbow
Camera	Global Settings
Lights	Radiosity
Light Source	Photons
Light Group	
Objects	
Finite Solid Objects	
Finite Patch Objects	
Infinite Solid Objects	
Isosurface	
Parametric	
Constructive Solid Geometry	

Table 7.1: Quick Reference Overview

NUMERIC_TERM:

    NUMERIC_FACTOR [MULT NUMERIC_FACTOR]...

MULT:

    * | /

NUMERIC_EXPRESSION:

    FLOAT_LITERAL | FLOAT_IDENTIFIER | SIGN NUMERIC_EXPRESSION | FLOAT_FUNCTION
        | FLOAT_BUILT_IN_IDENT | (FULL_EXPRESSION) | ! NUMERIC_EXPRESSION
        | VECTOR.DOT_ITEM | FLOAT_FUNCTION_INVOCATION

FLOAT_LITERAL:

    [DIGIT...][.]DIGIT...[EXP[SIGN]DIGIT...]

DIGIT:

    0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

EXP:

    e | E

FLOAT_FUNCTION:

    abs (FLOAT) | acos (FLOAT) | acosh (FLOAT) | asc (STRING) | asin (FLOAT)
        | asinh (FLOAT) | atan (FLOAT) | atanh (FLOAT) | atan2 (FLOAT,FLOAT)
        | ceil (FLOAT) | cos (FLOAT) | cosh (FLOAT) | defined (IDENTIFIER)
        | degrees (FLOAT) | dimensions (ARRAY_IDENTIFIER) | dimension_size
        (ARRAY_IDENTIFIER,INT) | div (FLOAT,FLOAT) | exp (FLOAT) | file_exists
        (STRING) | floor (FLOAT) | int (FLOAT) | inside (SOLID_OBJECT_IDENT,
        VECTOR) | ln (FLOAT) | log (FLOAT) | max (FLOAT,FLOAT[,FLOAT]...)
        | min (FLOAT,FLOAT[,FLOAT]...) | mod (FLOAT,FLOAT) | pow (FLOAT,FLOAT)
        | radians (FLOAT) | rand (FLOAT) | seed (FLOAT) | select (FLOAT,FLOAT,FLOAT,[FLOAT])
        | sin (FLOAT) | sinh (FLOAT) | sqrt (FLOAT) | strcmp (STRING,STRING)
        | strlen (STRING) | tan (FLOAT) | tanh (FLOAT) | val (STRING) | vdot
        (VECTOR,VECTOR) | vlength (VECTOR)

FLOAT_BUILT_IN_IDENT:

    BOOLEAN_KEYWORD | clock | clock_delta | clock_on | final_clock | final_frame
        | frame_number | image_height | image_width | initial_clock | initial_frame
        | pi | version

BOOLEAN_KEYWORD:

    true | yes | on | false | no | off

FULL_EXPRESSION:

    LOGICAL_EXPRESSION [? FULL_EXPRESSION : FULL_EXPRESSION]

LOGICAL_EXPRESSION:

    REL_TERM [LOGICAL_OPERATOR REL_TERM]...

LOGICAL_OPERATOR:

& | **|**

REL_TERM:

FLOAT [REL_OPERATOR FLOAT]...

REL_OPERATOR:

< | <= | = | >= | > | **!=**

DOT_ITEM:

x | **y** | **z** | **t** | **u** | **v** | **red** | **green** | **blue** | **filter** | **transmit** | **gray**

INT:

FLOAT

Any fractional part is discarded.

BOOL:

BOOLEAN_KEYWORD | LOGICAL_EXPRESSION

Quick Reference Contents

7.3.2 Vectors

Vector Expressions

VECTOR:

VECTOR_TERM [SIGN VECTOR_TERM]...

VECTOR_TERM:

VECTOR_EXPRESSION [MULT VECTOR_EXPRESSION]...

VECTOR_EXPRESSION:

VECTOR_LITERAL | VECTOR_IDENTIFIER | SIGN VECTOR_EXPRESSION | VECTOR_FUNCTION
| VECTOR_BUILT_IN_IDENT | **!** VECTOR_EXPRESSION | FLOAT | VECTOR_FUNCTION_INVOCATION
| COLOR_FUNCTION_INVOCATION | SPLINE_INVOCATION

VECTOR_LITERAL:

< FLOAT, FLOAT [, FLOAT [, FLOAT [, FLOAT]]] >

VECTOR_FUNCTION:

min_extent (OBJECT_IDENTIFIER) | **max_extent** (OBJECT_IDENTIFIER) | **trace**
(OBJECT_IDENTIFIER,VECTOR,VECTOR[,VECTOR_IDENTIFIER]) | **vaxis_rotate**
(VECTOR,VECTOR,FLOAT) | **vcross** (VECTOR,VECTOR) | **vrotate** (VECTOR,VECTOR)
| **vnormalize** (VECTOR) | **vturbulence** (FLOAT,FLOAT,FLOAT,VECTOR)

VECTOR_BUILT_IN_IDENT:

x | **y** | **z** | **t** | **u** | **v**

Quick Reference Contents

7.3.3 Colors

Color Expressions

COLOR:

[**color**] COLOR_BODY | **colour** COLOR_BODY

COLOR_BODY:

COLOR_VECTOR | COLOR_KEYWORD_GROUP | COLOR_IDENTIFIER

COLOR_VECTOR:

rgb 3D_VECTOR | **rgbf** 4D_VECTOR | **rgbt** 4D_VECTOR | [**rgbft**] 5D_VECTOR

COLOR_KEYWORD_GROUP:

[COLOR_IDENTIFIER] COLOR_KEYWORD_ITEMS

COLOR_KEYWORD_ITEMS:

[**red** FLOAT] & [**green** FLOAT] & [**blue** FLOAT] & [**filter** FLOAT] & [**transmit** FLOAT]

Quick Reference Contents

7.3.4 User defined Functions

User-Defined Functions

USER_FUNCTION:

FLOAT_USER_FUNCTION | VECTOR_USER_FUNCTION | COLOR_USER_FUNCTION

FLOAT_USER_FUNCTION:

function { FN_FLOAT } | **function** (IDENT_LIST) { FN_FLOAT } | **function** { **pattern** { PATTERN [PATTERN_MODIFIERS]} }

IDENT_LIST:

IDENT_ITEM [, IDENT_LIST]

The maximum number of parameter identifiers is 56. An identifier may not be repeated in the list.

IDENT_ITEM:

x | **y** | **z** | **u** | **v** | PARAM_IDENTIFIER

PATTERN:

MAP_PATTERN | **brick** [BRICK_ITEM] | **checker** | **hexagon** | **object** { LIST_OBJECT }

VECTOR_USER_FUNCTION:

function { SPECIAL_VECTOR_FUNCTION }

SPECIAL_VECTOR_FUNCTION:

```
TRANSFORM | SPLINE
```

COLOR_USER_FUNCTION:

```
function { PIGMENT }
```

Specify a float expression in a user-defined function:

FN_FLOAT

```
LOGIC_AND [OR LOGIC_AND]
```

OR:

```
|
```

LOGIC_AND:

```
REL_TERM [AND REL_TERM]
```

AND:

```
&
```

REL_TERM:

```
TERM [REL_OPERATOR TERM]
```

REL_OPERATOR:

```
< | <= | = | >= | > | !=
```

TERM:

```
FACTOR [SIGN FACTOR]
```

SIGN:

```
+ | -
```

FACTOR:

```
EXPRESSION [MULT EXPRESSION]
```

MULT:

```
* | /
```

EXPRESSION:

```
FLOAT_LITERAL | FLOAT_IDENTIFIER | FN_FLOAT_FUNCTION | FLOAT_BUILT_IN_IDENT
   | ( FN_FLOAT ) | IDENT_ITEM | SIGN EXPRESSION | VECTOR_FUNCTION_INVOCATION.FN_DOT_ITEM
   | COLOR_FUNCTION_INVOCATION.FN_DOT_ITEM | FLOAT_FUNCTION_INVOCATION
```

FN_DOT_ITEM:

```
DOT_ITEM | hf
```

FN_FLOAT_FUNCTION:

```
abs (FN_FLOAT) | acos (FN_FLOAT) | acosh (FN_FLOAT) | asin (FN_FLOAT)
   | asinh (FN_FLOAT) | atan (FN_FLOAT) | atanh (FN_FLOAT) | atan2 (FN_FLOAT,FN_FLOAT)
   | ceil (FN_FLOAT) | cos (FN_FLOAT) | cosh (FN_FLOAT) | degrees (FN_FLOAT)
   | exp (FN_FLOAT) | floor (FN_FLOAT) | int (FN_FLOAT) | ln (FN_FLOAT)
```

| **log** (FN_FLOAT) | **max** (FN_FLOAT,FN_FLOAT[,FN_FLOAT]...) | **min** (FN_FLOAT,FN_FLOAT[,FN_
| **mod** (FN_FLOAT,FN_FLOAT) | **pow** (FN_FLOAT,FN_FLOAT) | **prod** (IDENTIFIER,
FN_FLOAT, FN_FLOAT, FN_FLOAT) | **radians** (FN_FLOAT) | **sin** (FN_FLOAT)
| **sinh** (FN_FLOAT) | **sqrt** (FN_FLOAT) | **sum** (IDENTIFIER, FN_FLOAT, FN_FLOAT,
FN_FLOAT) |**tan** (FN_FLOAT) | **tanh** (FN_FLOAT) | **select** (FN_FLOAT,FN_FLOAT,FN_FLOAT
[,FN_FLOAT])

Create an identifier for a user-defined function:

USER_FUNCTION_DECLARATION:

#declare FLOAT_FUNCTION_IDENTIFIER = FLOAT_USER_FUNCTION |
#local FLOAT_FUNCTION_IDENTIFIER = FLOAT_USER_FUNCTION |
#declare VECTOR_FUNCTION_IDENTIFIER = VECTOR_USER_FUNCTION |
#local VECTOR_FUNCTION_IDENTIFIER = VECTOR_USER_FUNCTION |
#declare COLOR_FUNCTION_IDENTIFIER = COLOR_USER_FUNCTION |
#local COLOR_FUNCTION_IDENTIFIER = COLOR_USER_FUNCTION

Reference a user-defined function:

FLOAT_FUNCTION_INVOCATION:

FLOAT_FUNCTION_IDENTIFIER (FN_PARAM_LIST)

VECTOR_FUNCTION_INVOCATION:

VECTOR_FUNCTION_IDENTIFIER (FN_PARAM_LIST)

COLOR_FUNCTION_INVOCATION:

COLOR_FUNCTION_IDENTIFIER (FN_PARAM_LIST)

FN_PARAM_LIST:

FN_PARAM_ITEM [, FN_PARAM_LIST]

FN_PARAM_ITEM:

x | **y** | **z** | **u** | **v** | FLOAT

Quick Reference Contents

7.3.5 Strings

String Expressions

STRING:

STRING_FUNCTION | STRING_IDENTIFIER | STRING_LITERAL

STRING_FUNCTION:

chr (INT) | **concat** (STRING,STRING[,STRING]...) | **str** (FLOAT,INT,INT)
| **strlwr** (STRING) | **strupr** (STRING) | **substr** (STRING,INT,INT) | **vstr**
(INT,VECTOR,STRING,INT,INT)

STRING_LITERAL:

QUOTE [CHARACTER...] QUOTE

Limited to 256 characters.

QUOTE:

"

CHARACTER:

Any ASCII or Unicode character, depending on the **charset** setting in **global_settings**. The following escape sequences might be useful when writing to files or message streams:

\a - alarm
\b - backspace
\f - form feed
\n - new line
\r - carriage return
\t - horizontal tab
\uNNNN - unicode character four-digit code
\v - vertical tab
\\ - backslash
\' - single quote
\" - double quote

Quick Reference Contents

7.3.6 Arrays

Define an array:

ARRAY_DECLARATION:

```
#declare ARRAY_IDENTIFIER = array DIMENSION... [ARRAY_INITIALIZER] |
  #local ARRAY_IDENTIFIER = array DIMENSION... [ARRAY_INITIALIZER]
```

Limited to five dimensions.

DIMENSION:

```
[ INT ]
```

The brackets here are part of the dimension specification. The integer must be greater than zero.

ARRAY_INITIALIZER:

```
{ ARRAY_INITIALIZER [, ARRAY_INITIALIZER]... } |
  { RVALUE [, RVALUE]... }
```

Place a value into an array:

ARRAY_ELEMENT_ASSIGNMENT:

```
#declare ARRAY_REFERENCE =RVALUE [;] |
  #local ARRAY_REFERENCE = RVALUE [;]
```

The semicolon is required for a FLOAT, VECTOR or COLOR assignment.

Reference an array:

ARRAY_REFERENCE:

```
ARRAY_IDENTIFIER ELEMENT...
```

ELEMENT:

```
[ INT ]
```

The brackets here are part of the element specification.

Quick Reference Contents

7.3.7 Splines

Define a spline:

SPLINE:

```
spline { SPLINE_ITEMS }
```

SPLINE_ITEMS

```
[SPLINE_TYPE] PATH_LIST | SPLINE_IDENTIFIER [SPLINE_TYPE][PATH_LIST]
```

SPLINE_TYPE:

```
linear_spline | quadratic_spline | cubic_spline | natural_spline
```

PATH_LIST:

```
FLOAT, VECTOR [[,] PATH_LIST]
```

Reference a spline:

SPLINE_INVOCATION:

```
SPLINE_IDENTIFIER ( FLOAT [, SPLINE_TYPE] )
```

Quick Reference Contents

7.4 Language Directives

Control the parsing of sections of the scene file:

LANGUAGE_DIRECTIVE:

```
INCLUDE_DIRECTIVE | IDENTIFIER_DECLARATION | UNDEF_DIRECTIVE | FOPEN_DIRECTIVE
   | FCLOSE_DIRECTIVE | READ_DIRECTIVE | WRITE_DIRECTIVE | DEFAULT_DIRECTIVE
   | VERSION_DIRECTIVE | IF_DIRECTIVE | IFDEF_DIRECTIVE | IFNDEF_DIRECTIVE
   | SWITCH_DIRECTIVE | WHILE_DIRECTIVE | TEXT_STREAM_DIRECTIVE | MACRO_DEFINITION
```

Quick Reference Contents

7.4.1 File Inclusion

Insert content of another scene file:

INCLUDE_DIRECTIVE:

#include FILE_NAME

File inclusion may be nested at most 10 levels deep.

FILE_NAME:

STRING

Quick Reference Contents

7.4.2 Identifier Declaration

Create an identifier for a value, object, etc.

IDENTIFIER_DECLARATION:

```
#declare IDENTIFIER = RVALUE [;] |
   #local IDENTIFIER = RVALUE [;]
```

Up to 127 characters, starting with a letter, consisting of letters, digits and/or the underscore. The semicolon is required for a FLOAT, VECTOR or COLOR declaration.

RVALUE:

```
FLOAT | VECTOR | COLOR | USER_FUNCTION | STRING | ARRAY_REFERENCE | SPLINE
   | TRANSFORM | CAMERA | LIGHT | OBJECT | MATERIAL | INTERIOR | TEXTURE
   | TEXTURE_MAP | PIGMENT | COLOR_MAP | PIGMENT_MAP | NORMAL | SLOPE_MAP
   | NORMAL_MAP | FINISH | MEDIA | DENSITY | DENSITY_MAP | FOG | RAINBOW
   | SKY_SPHERE
```

Destroy an identifier:

UNDEF_DIRECTIVE:

#undef IDENTIFIER

Quick Reference Contents

7.4.3 File Input/Output

Open a text file:

FOPEN_DIRECTIVE:

#fopen FILE_HANDLE_IDENTIFIER FILE_NAME OPEN_TYPE

OPEN_TYPE:

read | **write** | **append**

Close a text file:

FCLOSE_DIRECTIVE:

```
#fclose FILE_HANDLE_IDENTIFIER
```

Read string, float and/or vector values from a text file:

READ_DIRECTIVE:

```
#read ( FILE_HANDLE_IDENTIFIER, DATA_IDENTIFIER [, DATA_IDENTIFIER]...
    )
```

Use **defined**(FILE_HANDLE_IDENTIFIER) to detect end-of-file after a read.

DATA_IDENTIFIER:

```
UNDECLARED_IDENTIFIER | FLOAT_IDENTIFIER | VECTOR_IDENTIFIER | STRING_IDENTIFIER
    | ARRAY_REFERENCE
```

May read a value into an array reference if the array element's type has already been established.

Write string, float and/or vector values to a text file:

WRITE_DIRECTIVE:

```
#write ( FILE_HANDLE_IDENTIFIER, DATA_ITEM [, DATA_ITEM]...  )
```

DATA_ITEM:

```
FLOAT | VECTOR | STRING
```

Quick Reference Contents

7.4.4 Default Texture

Specify a default texture, pigment, normal or finish:

DEFAULT_DIRECTIVE:

```
#default { DEFAULT_ITEM }
```

DEFAULT_ITEM:

```
PLAIN_TEXTURE | PIGMENT | NORMAL | FINISH
```

Quick Reference Contents

7.4.5 Version Identfier

Specify the POV-Ray compatibility version number:

VERSION_DIRECTIVE:

```
#version FLOAT;
```

Quick Reference Contents

7.4.6 Control Flow Directives

Conditionally parse a section of the scene file, depending on a boolean expression:

IF_DIRECTIVE:

```
#if ( BOOL ) TOKENS [#else TOKENS] #end
```

TOKENS:

Any number of POV-Ray keywords, identifiers, values and/or punctuation.

Conditionally parse a section of the scene file, depending on the existence of an identifier:

IFDEF_DIRECTIVE:

```
#ifdef ( IDENTIFIER ) TOKENS [#else TOKENS] #end
```

IFNDEF_DIRECTIVE:

```
#ifndef ( IDENTIFIER ) TOKENS [#else TOKENS] #end
```

Conditionally parse a section of the scene file, depending on the value of a float expression:

SWITCH_DIRECTIVE:

```
#switch ( FLOAT ) SWITCH_CLAUSE...  [#else TOKENS] #end
```

SWITCH_CLAUSE:

```
#case ( FLOAT ) TOKENS [#break] |
   #range ( F_LOW, F_HIGH ) TOKENS [#break]
```

Repeat a section of the scene file while a boolean condition is true:

WHILE_DIRECTIVE:

```
#while ( LOGICAL_EXPRESSION ) TOKENS #end
```

Quick Reference Contents

7.4.7 Message Streams

Send a message to a text stream:

TEXT_STREAM_DIRECTIVE:

```
#debug STRING | #error STRING | #warning STRING
```

Quick Reference Contents

7.4.8 Macro

Define a macro:

MACRO_DEFINITION:

```
#macro MACRO_IDENTIFIER ( [PARAM_IDENTIFIER [, PARAM_IDENTIFIER]...]
   ) TOKENS #end
```

A parameter identifier may not be repeated in the list.

Invoke a macro:

MACRO_INVOCATION:

```
MACRO_IDENTIFIER ( [ACTUAL_PARAM [, ACTUAL_PARAM]...] )
```

ACTUAL_PARAM:

```
IDENTIFIER | RVALUE
```

Quick Reference Contents

7.4.9 Embedded Directives

Some directives may be embedded in CAMERA, LIGHT, OBJECT and ATMOSPHERIC_EFFECT statements. However, the directives should only include items (if any) that are valid for a given statement. Also, they should not disrupt the required order of items, where applicable.

EMBEDDED_DIRECTIVE:

```
IDENTIFIER_DECLARATION | UNDEF_DIRECTIVE | READ_DIRECTIVE | WRITE_DIRECTIVE
   | IF_DIRECTIVE | IFDEF_DIRECTIVE | IFNDEF_DIRECTIVE | SWITCH_DIRECTIVE
   | WHILE_DIRECTIVE | TEXT_STREAM_DIRECTIVE
```

Quick Reference Contents

7.5 Transformations

Rotate, resize, move, or otherwise manipulate the coordinates of an object or texture

TRANSFORMATION:

```
rotate VECTOR | scale VECTOR | translate VECTOR | TRANSFORM | MATRIX
```

TRANSFORM:

```
transform TRANSFORM_IDENTIFIER | transform { [TRANSFORM_ITEM...] }
```

TRANSFORM_ITEM:

```
TRANSFORM_IDENTIFIER | TRANSFORMATION | inverse
```

MATRIX:

```
matrix < F_VAL00, F_VAL01, F_VAL02, F_VAL10, F_VAL11, F_VAL12, F_VAL20,
   F_VAL21, F_VAL22, F_VAL30, F_VAL31, F_VAL32 >
```

Quick Reference Contents

7.6 Camera

Describe the position, projection type and properties of the camera viewing the scene

CAMERA:

Jump to SDL

camera { [CAMERA_TYPE] [CAMERA_ITEMS] [CAMERA_MODIFIERS] } |

camera { CAMERA_IDENTIFIER [TANSFORMATIONS ...] }

CAMERA_TYPE:

perspective | **orthographic** | **fisheye** | **ultra_wide_angle** | **omnimax** | **panoramic** | **spherical** | **cylinder** CYLINDER_TYPE

CYLINDER_TYPE:

1 | **2** | **3** | **4**

CAMERA_ITEMS:

[**location** VECTOR] & [**right** VECTOR] & [**up** VECTOR] & [**direction** VECTOR] & [**sky** VECTOR]

CAMERA_MODIFIERS:

[**angle** [angle F_HORIZONTAL] [,F_VERTICAL]] & [**look_at** VECTOR] & [FOCAL_BLUR] & [NORMAL] & [TRANSFORMATION...]

FOCAL_BLUR:

aperture FLOAT & **blur_samples** INT & [**focal_point** VECTOR] & [**confidence** FLOAT] & [**variance** FLOAT]

Quick Reference Contents

7.7 Lights

Specify light sources for the scene or for specific objects

LIGHT:

LIGHT_SOURCE | LIGHT_GROUP

Describe the position, type and properties of a light source for the scene:

LIGHT_SOURCE:

Jump to SDL

light_source { V_LOCATION, COLOR [LIGHT_SOURCE_ITEMS] }

LIGHT_SOURCE_ITEMS:

[LIGHT_TYPE] & [AREA_LIGHT_ITEMS] & [LIGHT_MODIFIERS]

LIGHT_TYPE:

spotlight [SPOTLIGHT_ITEMS] | **cylinder** [SPOTLIGHT_ITEMS]

SPOTLIGHT_ITEMS:

[**radius** FLOAT] & [**falloff** FLOAT] & [**tightness** FLOAT] & [**point_at** VECTOR]

AREA_LIGHT_ITEMS:

area_light V_AXIS1, V_AXIS2, I_SIZE1, I_SIZE2 [AREA_LIGHT_MODIFIERS]

AREA_LIGHT_MODIFIERS:

[**adaptive** INT] & [**jitter**] & [**circular**] & [**orient**]

LIGHT_MODIFIERS:

[LIGHT_PHOTONS] & [**looks_like** { OBJECT }] & [TRANSFORMATION...] & [**fade_distance** FLOAT] & [**fade_power** FLOAT] & [**media_attenuation** [BOOL]] & [**media_interaction** [BOOL]] & [**shadowless**] & [**projected_through** { OBJECT_IDENTIFIER }] & [**parallel** [**point_at** VECTOR]]

Specify how a light source should interact with photons:

LIGHT_PHOTONS:

photons { LIGHT_PHOTON_ITEMS }

LIGHT_PHOTON_ITEMS:

[**refraction** BOOL] & [**reflection** BOOL] & [**area_light**]

Quick Reference Contents

7.7.1 Lightgroup

Assign objects to specific light sources:

LIGHT_GROUP:

Jump to SDL

light_group { LIGHT_GROUP_ITEM... [LIGHT_GROUP_MODIFIERS] }

LIGHT_GROUP_ITEM:

LIGHT_SOURCE | OBJECT | LIGHT_GROUP

LIGHT_GROUP_MODIFIERS:

[**global_lights** BOOL] & [TRANSFORMATION...]

Quick Reference Contents

7.8 Objects

Describe an object in the scene

OBJECT:

FINITE_SOLID_OBJECT | FINITE_PATCH_OBJECT | INFINITE_SOLID_OBJECT | ISOSURFACE
| PARAMETRIC | CSG_OBJECT | OBJECT_STATEMENT

OBJECT_STATEMENT:

object { OBJECT_IDENTIFIER [OBJECT_MODIFIERS] }

Quick Reference Contents

7.8.1 Finite Solid Objects

Describe a solid finite shape:

FINITE_SOLID_OBJECT:

BLOB | BOX | CONE | CYLINDER | HEIGHT_FIELD | JULIA_FRACTAL | LATHE |
PRISM | SPHERE | SPHERE_SWEEP | SUPERELLIPSOID | SOR | TEXT | TORUS

The blob object:

BLOB:

Jump to SDL

blob { [**threshold** FLOAT] BLOB_ITEM... [BLOB_MODIFIERS] }

BLOB_ITEM:

sphere { V_CENTER, F_RADIUS, [**strength**] F_STRENGTH [COMPONENT_MODIFIERS]
} |
cylinder { V_END1, V_END2, F_RADIUS, [**strength**] F_STRENGTH [COMPONENT_MODIFIERS]
}

COMPONENT_MODIFIERS:

[TEXTURE] & [PIGMENT] & [NORMAL] & [FINISH] & [TRANSFORMATION...]

BLOB_MODIFIERS:

[**hierarchy** [BOOL]] & [**sturm** [BOOL]] & [OBJECT_MODIFIERS]

The box object:

BOX:

Jump to SDL

box { V_CORNER1, V_CORNER2 [BOX_MODIFIERS] }

BOX_MODIFIERS:

[UV_MAPPING] & [OBJECT_MODIFIERS]

The cone object:

CONE:

Jump to SDL

cone { V_BASE_CENTER, F_BASE_RADIUS, V_CAP_CENTER, F_CAP_RADIUS [**open**]
[OBJECT_MODIFIERS] }

The cylinder object:

CYLINDER:

Jump to SDL

cylinder { V_BASE_CENTER, V_CAP_CENTER, F_RADIUS [**open**] [OBJECT_MODIFIERS] }

The height field object:

HEIGHT_FIELD:

Jump to SDL

height_field { HF_IMAGE [HF_MODIFIERS] }

HF_IMAGE:

FUNCTION_IMAGE | [HF_TYPE] FILE_NAME

HF_TYPE:

gif | **tga** | **pot** | **png** | **pgm** | **ppm** | **jpeg** | **tiff** | **sys**

HF_MODIFIERS:

[**hierarchy** [BOOL]] & [**smooth**] & [**water_level** FLOAT] & [OBJECT_MODIFIERS]

The Julia fractal object:

JULIA_FRACTAL:

Jump to SDL

julia_fractal { 4D_VECTOR [JF_ITEMS] [OBJECT_MODIFIERS] }

JF_ITEMS:

[ALGEBRA_ITEM] & [**max_iteration** INT] & [**precision** FLOAT] & [**slice** V4_NORMAL, F_DISTANCE]

ALGEBRA_ITEM:

quaternion [QUATER_FUNCTION] | **hypercomplex** [HYPER_FUNCTION]

QUATER_FUNCTION:

sqr | **cube**

HYPER_FUNCTION:

sqr | **cube** | **exp** | **reciprocal** | **sin** | **asin** | **sinh** | **asinh** | **cos** | **acos** | **cosh** | **acosh** | **tan** | **atan** | **tanh** | **atanh** | **ln** | **pwr** (FLOAT,FLOAT)

The lathe object:

LATHE:

Jump to SDL

lathe { [LATHE_SPLINE_TYPE] I_NUM_POINTS, POINT_LIST [LATHE_MODIFIERS] }

LATHE_SPLINE_TYPE:

linear_spline | **quadratic_spline** | **cubic_spline** | **bezier_spline**

POINT_LIST:

2D_VECTOR [, 2D_VECTOR]...

The quantity of 2D_VECTORs is specified by the I_NUM_POINTS value.

LATHE_MODIFIERS:

[**sturm** [BOOL]] & [UV_MAPPING] & [OBJECT_MODIFIERS]

The prism object:

PRISM:

Jump to SDL

prism { [PRISM_ITEMS] F_HEIGHT1, F_HEIGHT2, I_NUM_POINTS, POINT_LIST [**open**] [PRISM_MODIFIERS] }

PRISM_ITEMS:

[PRISM_SPLINE_TYPE] & [PRISM_SWEEP_TYPE]

PRISM_SPLINE_TYPE:

linear_spline | **quadratic_spline** | **cubic_spline** | **bezier_spline**

PRISM_SWEEP_TYPE:

linear_sweep | **conic_sweep**

PRISM_MODIFIERS:

[**sturm** [BOOL]] & [OBJECT_MODIFIERS]

The sphere object:

SPHERE:

Jump to SDL

sphere { V_CENTER, F_RADIUS [SPHERE_MODIFIERS] }

SPHERE_MODIFIERS:

[UV_MAPPING] & [OBJECT_MODIFIERS]

The sphere sweep object:

SPHERE_SWEEP:

Jump to SDL

sphere_sweep { SWEEP_SPLINE_TYPE I_NUM_SPHERES, SPHERE_LIST [**tolerance** F_DEPTH_TOLERANCE] [OBJECT_MODIFIERS] }

SWEEP_SPLINE_TYPE:

linear_spline | **b_spline** | **cubic_spline**

SPHERE_LIST:

V_CENTER, F_RADIUS [, SPHERE_LIST]

The quantity of V_CENTER, F_RADIUS pairs is specified by the I_NUM_SPHERES value.

The superquadric ellipsoid object:

SUPERELLIPSOID:

Jump to SDL

superellipsoid { < FLOAT, FLOAT > [OBJECT_MODIFIERS] }

The surface of revolution object:

SOR:

Jump to SDL

sor { I_NUM_POINTS, POINT_LIST [**open**] [SOR_MODIFIERS] }

SOR_MODIFIERS:

[**sturm** [BOOL]] & [UV_MAPPING] & [OBJECT_MODIFIERS]

The text object:

TEXT:

Jump to SDL

text { **ttf** FILE_NAME STRING F_THICKNESS, V_OFFSET [OBJECT_MODIFIERS] }

The torus object:

TORUS:

Jump to SDL

torus { F_MAJOR_RADIUS, F_MINOR_RADIUS [TORUS_MODIFIERS] }

TORUS_MODIFIERS:

[**sturm** [BOOL]] & [UV_MAPPING] & [OBJECT_MODIFIERS]

Quick Reference Contents

7.8.2 Finite Patch Objects

Describe a totally thin, finite shape:

FINITE_PATCH_OBJECT:

Jump to SDL

BICUBIC_PATCH | DISC | MESH | MESH2 | POLYGON | TRIANGLE | SMOOTH_TRIANGLE

The bicubic patch object:

BICUBIC_PATCH:

Jump to SDL

bicubic_patch { PATCH_ITEMS [PATCH_UV_VECTORS] CONTROL_POINTS [BICUBIC_PATCH_MODIFIERS] }

PATCH_ITEMS:

`type PATCH_TYPE & [u_steps INT] & [v_steps INT] & [flatness FLOAT]`

PATCH_TYPE:

`0 | 1`

PATCH_UV_VECTORS:

`uv_vectors V2_CORNER1, V2_CORNER2, V2_CORNER3, V2_CORNER4`

CONTROL_POINTS:

16 VECTORs, optionally separated by commas.

BICUBIC_PATCH_MODIFIERS:

`[UV_MAPPING] & [OBJECT_MODIFIERS]`

The disc object:

DISC:

Jump to SDL

```
disc { V_CENTER, V_NORMAL, F_RADIUS [, F_HOLE_RADIUS] [OBJECT_MODIFIERS]
    }
```

The mesh object:

MESH:

Jump to SDL

`mesh { MESH_TRIANGLE... [MESH_MODIFIERS] }`

MESH_TRIANGLE:

```
triangle { V_CORNER1, V_CORNER2, V_CORNER3 [MESH_UV_VECTORS] [MESH_TEXTURE]
    } |
    smooth_triangle { V_CORNER1, V_NORMAL1, V_CORNER2, V_NORMAL2, V_CORNER3,
    V_NORMAL3 [MESH_UV_VECTORS] [MESH_TEXTURE] }
```

MESH_UV_VECTORS:

`uv_vectors V2_CORNER1, V2_CORNER2, V2_CORNER3`

MESH_TEXTURE:

```
texture { TEXTURE_IDENTIFIER } |
    texture_list { TEXTURE_IDENTIFIER TEXTURE_IDENTIFIER TEXTURE_IDENTIFIER
    }
```

MESH_MODIFIERS:

`[inside_vector V_DIRECTION] & [hierarchy [BOOL]] & [UV_MAPPING] & [OBJECT_MODIFIERS]`

The mesh2 object:

MESH2:

Jump to SDL

mesh2 { MESH2_VECTORS [TEXTURE_LIST] MESH2_INDICES [MESH2_MODIFIERS] }

MESH2_VECTORS:

VERTEX_VECTORS [NORMAL_VECTORS] [UV_VECTORS]

VERTEX_VECTORS:

vertex_vectors { I_NUM_VERTICES, VECTOR [, VECTOR]... }

NORMAL_VECTORS:

normal_vectors { I_NUM_NORMALS, VECTOR [, VECTOR]... }

UV_VECTORS:

uv_vectors { I_NUM_UV_VECTORS, 2D_VECTOR [, 2D_VECTOR]... }

TEXTURE_LIST:

texture_list { I_NUM_TEXTURES, TEXTURE [, TEXTURE]... }

MESH2_INDICES:

FACE_INDICES [NORMAL_INDICES] [UV_INDICES]

FACE_INDICES:

face_indices { I_NUM_FACES, FACE_INDICES_ITEM [, FACE_INDICES_ITEM]... }

FACE_INDICES_ITEM:

VECTOR [, I_TEXTURE_INDEX [, I_TEXTURE_INDEX, I_TEXTURE_INDEX]]

NORMAL_INDICES:

normal_indices { I_NUM_FACES, VECTOR [, VECTOR]... }

UV_INDICES:

uv_indices { I_NUM_FACES, VECTOR [, VECTOR]... }

MESH2_MODIFIERS:

[**inside_vector** V_DIRECTION] & [UV_MAPPING] & [OBJECT_MODIFIERS]

The polygon object:

POLYGON:

Jump to SDL

polygon { I_NUM_POINTS, V_POINT [, V_POINT]... [OBJECT_MODIFIERS] }

The quantity of V_POINTs is specified by the I_NUM_POINTS value.

The triangle object:

TRIANGLE:

Jump to SDL

triangle { V_CORNER1, V_CORNER2, V_CORNER3 [OBJECT_MODIFIERS] }

The smooth triangle object:

SMOOTH_TRIANGLE:

Jump to SDL

smooth_triangle { V_CORNER1, V_NORMAL1, V_CORNER2, V_NORMAL2, V_CORNER3, V_NORMAL3 [OBJECT_MODIFIERS] }

Quick Reference Contents

7.8.3 Infinite Solid Objects

Describe a solid, possibly infinite, shape:

INFINITE_SOLID_OBJECT:

PLANE | POLY | CUBIC | QUARTIC | QUADRIC

The plane object:

PLANE:

Jump to SDL

plane { V_NORMAL, F_DISTANCE [OBJECT_MODIFIERS] }

The poly object:

POLY:

Jump to SDL

poly { ORDER, < POLY_COEFFICIENTS > [POLY_MODIFIERS] }

ORDER:

An integer value between 2 and 15 inclusive.

POLY_COEFFICIENTS:

A quantity **n** of FLOATs separated by commas, where **n** is ((ORDER+1)*(ORDER+2)*(ORDER+3))/6.

POLY_MODIFIERS:

[**sturm** [BOOL]] & [OBJECT_MODIFIERS]

The cubic object:

CUBIC:

cubic { < CUBIC_COEFFICIENTS > [POLY_MODIFIERS] }

CUBIC_COEFFICIENTS:

20 FLOATs separated by commas.

The quartic object:

QUARTIC:

quartic { < QUARTIC_COEFFICIENTS > [POLY_MODIFIERS] }

QUARTIC_COEFFICIENTS:

35 FLOATs separated by commas.

The quadric object:

QUADRIC:

Jump to SDL

quadric { < FLOAT, FLOAT, FLOAT >, < FLOAT, FLOAT, FLOAT >, < FLOAT, FLOAT, FLOAT >, FLOAT [OBJECT_MODIFIERS] }

Quick Reference Contents

7.8.4 Isosurface

Describe a surface via a mathematical function:

ISOSURFACE:

Jump to SDL

isosurface { FLOAT_USER_FUNCTION [ISOSURFACE_ITEMS] [OBJECT_MODIFIERS] }

ISOSURFACE_ITEMS:

[**contained_by** { CONTAINER }] & [**threshold** FLOAT] & [**accuracy** FLOAT] & [**max_gradient** FLOAT [**evaluate** F_MIN_ESTIMATE, F_MAX_ESTIMATE, F_ATTENUATION]] & [**open**] & [INTERSECTION_LIMIT]

CONTAINER:

sphere { V_CENTER, F_RADIUS } | **box** { V_CORNER1, V_CORNER2 }

INTERSECTION_LIMIT:

max_trace INT | **all_intersections**

Quick Reference Contents

7.8.5 Parametric

Describe a surface using functions to locate points on the surface:

PARAMETRIC:

Jump to SDL

parametric { FLOAT_USER_FUNCTION, FLOAT_USER_FUNCTION, FLOAT_USER_FUNCTION 2D_VECTOR, 2D_VECTOR [PARAMETRIC_ITEMS] [UV_MAPPING] & [OBJECT_MODIFIERS] }

PARAMETRIC_ITEMS:

[**contained_by** { CONTAINER }] & [**max_gradient** FLOAT] & [**accuracy** FLOAT] & [**precompute** I_DEPTH, **x, y, z**]

CONTAINER:

sphere { V_CENTER, F_RADIUS } | **box** { V_CORNER1, V_CORNER2 }

Quick Reference Contents

7.8.6 CSG

Describe one complex shape from multiple shapes:

CSG_OBJECT:

Jump to SDL

UNION | INTERSECTION | DIFFERENCE | MERGE

Combine multiple shapes into one:

UNION:

union { UNION_OBJECT UNION_OBJECT... [UNION_MODIFIERS] }

UNION_OBJECT:

OBJECT | LIGHT

UNION_MODIFIERS:

[**split_union** BOOL] & [OBJECT_MODIFIERS]

Create a new shape from the overlapping portions of multiple shapes:

INTERSECTION:

intersection { SOLID_OBJECT SOLID_OBJECT... [INTERSECTION_MODIFIERS] }

SOLID_OBJECT:

FINITE_SOLID_OBJECT | INFINITE_SOLID_OBJECT | ISOSURFACE | CSG_OBJECT

INTERSECTION_MODIFIERS:

[**cutaway_textures**] & [OBJECT_MODIFIERS]

Subtract one or more shapes from another:

DIFFERENCE:

difference { SOLID_OBJECT SOLID_OBJECT... [DIFFERENCE_MODIFIERS] }

DIFFERENCE_MODIFIERS:

[**cutaway_textures**] & [OBJECT_MODIFIERS]

Combine multiple shapes into one, removing internal surfaces:

MERGE:

merge { SOLID_OBJECT SOLID_OBJECT... [OBJECT_MODIFIERS] }

Quick Reference Contents

7.9 Object Modifiers

Manipulate the appearance of an object

OBJECT_MODIFIERS:

```
[OBJECT_PHOTONS] & [CLIPPED_BY] & [BOUNDED_BY] & [MATERIAL] & [INTERIOR]
   & [INTERIOR_TEXTURE] & [TEXTURE] & [PIGMENT] & [NORMAL] & [FINISH]
   & [TRANSFORMATION...]  & [no_shadow] & [no_image[BOOL]] & [no_reflection{BOOL]]
   & [inverse] & [double_illuminate[BOOL]] & [hollow [BOOL]]
```

Specify how an object should interact with photons:

OBJECT_PHOTONS:

Jump to SDL

```
photons { OBJECT_PHOTON_ITEMS }
```

OBJECT_PHOTON_ITEMS:

```
[target [F_SPACING_MULT]] & [refraction BOOL] & [reflection BOOL] & [collect
   BOOL] & [pass_through [BOOL]]
```

Slice a portion of a shape:

CLIPPED_BY:

```
clipped_by { UNTEXTURED_SOLID_OBJECT...  } |
   clipped_by { bounded_by }
```

UNTEXTURED_SOLID_OBJECT:

```
FINITE_SOLID_OBJECT | INFINITE_SOLID_OBJECT
```

Note, neither with a texture applied.

Specify a bounding shape for an object:

BOUNDED_BY:

```
bounded_by { UNTEXTURED_SOLID_OBJECT...  } |
   bounded_by { clipped_by }
```

Quick Reference Contents

7.9.1 UV Mapping

Map a texture to an object using surface coordinates:

UV_MAPPING:

Jump to SDL

```
uv_mapping PIGMENT | pigment { uv_mapping PIGMENT_BODY } |
   uv_mapping NORMAL | normal { uv_mapping NORMAL_BODY } |
   uv_mapping TEXTURE | texture { uv_mapping TEXTURE_BODY }
```

Quick Reference Contents

7.9.2 Material

Group together surface textures and interior properties:

MATERIAL:

material { [MATERIAL_IDENTIFIER] [MATERIAL_ITEM ...] }

MATERIAL_ITEMS:

TEXTURE | INTERIOR_TEXTURE | INTERIOR | TRANSFORMATION

Quick Reference Contents

7.9.3 Interior

Describe the interior of an object:

INTERIOR:

Jump to SDL

interior { [INTERIOR_IDENTIFIER] [INTERIOR_ITEMS] }

INTERIOR_ITEMS:

[**ior** FLOAT] & [**dispersion** FLOAT] & [**dispersion_samples** INT] & [**caustics** FLOAT] & [**fade_distance** FLOAT] & [**fade_power** FLOAT] & [**fade_color** COLOR] & [MEDIA...]

Quick Reference Contents

7.9.4 Interior Texture

Describe the interior surface of an object:

INTERIOR_TEXTURE:

interior_texture { TEXTURE_BODY }

Quick Reference Contents

7.10 Texture

Describe the surface of an object

TEXTURE:

PLAIN_TEXTURE | LAYERED_TEXTURE | PATTERNED_TEXTURE

Quick Reference Contents

7.10.1 Plain Texture

Describe a texture consisting of a single pigment, normal and finish:

PLAIN_TEXTURE:

```
texture { PLAIN_TEXTURE_BODY }
```

PLAIN_TEXTURE_BODY:

```
[PLAIN_TEXTURE_IDENT] [PNF_IDENTIFIERS] [PNF_ITEMS]
```

PNF_IDENTIFIERS:

```
[PIGMENT_IDENTIFIER] & [NORMAL_IDENTIFIER] & [FINISH_IDENTIFIER]
```

PNF_ITEMS:

```
[PIGMENT] & [NORMAL] & [FINISH] & [TRANSFORMATION...]
```

Quick Reference Contents

7.10.2 Layered Texture

Describe a texture consisting of two or more semi-transparent layers:

LAYERED_TEXTURE:

Jump to SDL

```
texture { LAYERED_TEXTURE_IDENT } |
   PLAIN_TEXTURE PLAIN_TEXTURE...
```

Quick Reference Contents

7.10.3 Patterned Texture

Describe a texture using a pattern or blending function:

PATTERNED_TEXTURE:

Jump to SDL

```
texture { PATTERNED_TEXTURE_BODY }
```

PATTERNED_TEXTURE_BODY:

```
PATTERNED_TEXTURE_IDENT [TRANSFORMATION...]  | TEXTURE_PATTERN [PATTERN_MODIFIERS]
   | MATERIAL_MAP [TRANSFORMATION...]
```

TEXTURE_PATTERN:

```
TEXTURE_LIST_PATTERN | MAP_PATTERN TEXTURE_MAP
```

TEXTURE_LIST_PATTERN:

brick TEXTURE, TEXTURE [BRICK_ITEMS] |
checker TEXTURE, TEXTURE |
hexagon TEXTURE, TEXTURE, TEXTURE |
object { LIST_OBJECT TEXTURE, TEXTURE }

BRICK_ITEMS:

[**brick_size** VECTOR] & [**mortar** FLOAT]

LIST_OBJECT:

UNTEXTURED_SOLID_OBJECT | UNTEXTURED_SOLID_OBJECT_IDENT

TEXTURE_MAP:

texture_map { TEXTURE_MAP_BODY } [BLEND_MAP_MODIFIERS]

TEXTURE_MAP_BODY:

TEXTURE_MAP_IDENTIFIER | TEXTURE_MAP_ENTRY...

There may be from 2 to 256 map entries.

TEXTURE_MAP_ENTRY:

[FLOAT TEXTURE_BODY **]**

The brackets here are part of the map entry.

TEXTURE_BODY:

PLAIN_TEXTURE_BODY | LAYERED_TEXTURE_IDENT | PATTERNED_TEXTURE_BODY

MATERIAL_MAP:

material_map { BITMAP_IMAGE [BITMAP_MODIFIERS] TEXTURE... }

Quick Reference Contents

7.10.4 Pigment

Describe a color or pattern of colors for a texture:

PIGMENT:

Jump to SDL

pigment { PIGMENT_BODY }

PIGMENT_BODY:

[PIGMENT_IDENTIFIER] [PIGMENT_TYPE] [PIGMENT_MODIFIERS]

PIGMENT_TYPE:

COLOR | COLOR_LIST_PATTERN | PIGMENT_LIST_PATTERN | IMAGE_MAP | MAP_PATTERN [COLOR_MAP] | MAP_PATTERN PIGMENT_MAP

COLOR_LIST_PATTERN:

brick [COLOR [, COLOR]] [BRICK_ITEMS] |
checker [COLOR [, COLOR]] |
hexagon [COLOR [, COLOR [, COLOR]]] |
object { LIST_OBJECT [COLOR [, COLOR]] }

PIGMENT_LIST_PATTERN:

brick PIGMENT, PIGMENT [BRICK_ITEMS] |
checker PIGMENT, PIGMENT |
hexagon PIGMENT, PIGMENT, PIGMENT |
object { LIST_OBJECT PIGMENT, PIGMENT }

IMAGE_MAP:

image_map {BITMAP_IMAGE [IMAGE_MAP_MODIFIER...] [BITMAP_MODIFIERS] }

IMAGE_MAP_MODIFIER:

filter I_PALETTE, F_AMOUNT | **filter all** F_AMOUNT | **transmit** I_PALETTE, F_AMOUNT | **transmit all** F_AMOUNT

COLOR_MAP:

color_map { COLOR_MAP_BODY } [BLEND_MAP_MODIFIERS] |
colour_map { COLOR_MAP_BODY } [BLEND_MAP_MODIFIERS]

COLOR_MAP_BODY:

COLOR_MAP_IDENTIFIER | COLOR_MAP_ENTRY...

There may be from 2 to 256 map entries.

COLOR_MAP_ENTRY:

[FLOAT COLOR **]**

The brackets here are part of the map entry.

PIGMENT_MAP:

pigment_map { PIGMENT_MAP_BODY } [BLEND_MAP_MODIFIERS]

PIGMENT_MAP_BODY:

PIGMENT_MAP_IDENTIFIER | PIGMENT_MAP_ENTRY...

There may be from 2 to 256 map entries.

PIGMENT_MAP_ENTRY:

[FLOAT PIGMENT_BODY **]**

The brackets here are part of the map entry.

PIGMENT_MODIFIERS:

[QUICK_COLOR] & [PATTERN_MODIFIERS]

QUICK_COLOR:

quick_color COLOR | **quick_colour** COLOR

7.10.5 Normal

Simulate the visual or tactile surface characteristics of a texture:

NORMAL:

Jump to SDL

normal { NORMAL_BODY }

NORMAL_BODY:

[NORMAL_IDENTIFIER] [NORMAL_TYPE] [NORMAL_MODIFIERS]

NORMAL_TYPE:

NORMAL_PATTERN | BUMP_MAP

NORMAL_PATTERN:

NORMAL_LIST_PATTERN |
MAP_PATTERN [F_DEPTH] [SLOPE_MAP] |
MAP_PATTERN NORMAL_MAP

NORMAL_LIST_PATTERN:

brick NORMAL, NORMAL [BRICK_ITEMS] | **brick** [F_DEPTH] [BRICK_ITEMS] |
checker NORMAL, NORMAL | **checker** [F_DEPTH] |
hexagon NORMAL, NORMAL, NORMAL | **hexagon** [F_DEPTH] |
object { LIST_OBJECT NORMAL, NORMAL } | **object** { LIST_OBJECT } [F_DEPTH]

NORMAL_MAP:

normal_map { NORMAL_MAP_BODY } [BLEND_MAP_MODIFIERS]

NORMAL_MAP_BODY:

NORMAL_MAP_IDENTIFIER | NORMAL_MAP_ENTRY...

There may be from 2 to 256 map entries.

NORMAL_MAP_ENTRY:

[FLOAT NORMAL_BODY **]**

The brackets here are part of the map entry.

SLOPE_MAP:

slope_map { SLOPE_MAP_BODY } [BLEND_MAP_MODIFIERS]

SLOPE_MAP_BODY:

SLOPE_MAP_IDENTIFIER | SLOPE_MAP_ENTRY...

There may be from 2 to 256 map entries.

SLOPE_MAP_ENTRY:

[FLOAT, < F_HEIGHT, F_SLOPE > **]**

The brackets here are part of the map entry.

BUMP_MAP:

bump_map { BITMAP_IMAGE [BUMP_MAP_MODIFIERS] }

BUMP_MAP_MODIFIERS:

[BITMAP_MODIFIERS] & [BUMP_METHOD] & [**bump_size** FLOAT]

BUMP_METHOD:

use_index | **use_color** | **use_colour**

NORMAL_MODIFIERS:

[PATTERN_MODIFIERS] & [**bump_size** FLOAT] & [**no_bump_scale** [BOOL]] & [**accuracy** FLOAT]

Quick Reference Contents

7.10.6 Finish

Describe the reflective properties of a surface:

FINISH:

Jump to SDL

finish { [FINISH_IDENTIFIER] [FINISH_ITEMS] }

FINISH_ITEMS:

[**ambient** COLOR] & [**diffuse** FLOAT] & [**brilliance** FLOAT] & [PHONG] & [SPECULAR] & [REFLECTION] & [IRID] & [**crand** FLOAT] & [**conserve_energy** [BOOL]]

PHONG:

phong FLOAT & [**phong_size** FLOAT] & [**metallic** [FLOAT]]

SPECULAR:

specular FLOAT & [**roughness** FLOAT] & [**metallic** [FLOAT]]

REFLECTION:

reflection COLOR [**reflection_exponent** FLOAT] |
reflection { [COLOR,] COLOR [REFLECTION_ITEMS] }

REFLECTION_ITEMS:

[**fresnel** BOOL] & [**falloff** FLOAT] & [**exponent** FLOAT] & [**metallic** [FLOAT]]

Must also use **interior** {**ior** FLOAT} in the object when **fresnel** is used.

IRID:

irid { F_AMOUNT [IRID_ITEMS] }

IRID_ITEMS:

[**thickness** FLOAT] & [**turbulence** FLOAT]

Quick Reference Contents

7.10.7 Pattern

Specify a pattern function for a texture, pigment, normal or density:

MAP_PATTERN:

Jump to SDL

AGATE | **average** | **boxed** | **bozo** | **bumps** | **cells** | CRACKLE | **cylindrical**
| DENSITY_FILE | **dents** | FACETS | FRACTAL | **function** { FN_FLOAT }
| **gradient** VECTOR | **granite** | IMAGE_PATTERN | **leopard** | **marble** |
onion | **pigment_pattern** { PIGMENT_BODY } | **planar** | QUILTED | **radial**
| **ripples** | SLOPE | **spherical** | **spiral1** I_NUM_ARMS | **spiral2** I_NUM_ARMS
| **spotted** | **waves** | **wood** | **wrinkles**

AGATE:

agate [**agate_turb** FLOAT]

CRACKLE:

crackle [CRACKLE_TYPES]

CRACKLE_TYPES:

[**form** VECTOR] & [**metric** FLOAT] & [**offset** FLOAT] & [**solid**]

DENSITY_FILE:

density_file df3 FILE_NAME [**interpolate** DENSITY_INTERPOLATE]

DENSITY_INTERPOLATE:

0 | **1** | **2**

FACETS:

facets FACETS_TYPE

Note, **facets** can only be used as a **normal** pattern.

FACETS_TYPE:

coords F_SCALE | **size** F_SIZE_FACTOR

FRACTAL:

MANDELBROT_FRACTAL | JULIA_FRACTAL | MAGNET_MANDEL_FRACTAL | MAGNET_JULIA_FRACTAL

MANDELBROT_FRACTAL:

mandel I_ITERATIONS [**exponent** INT] [**exterior** EXTERIOR_TYPE, F_FACTOR]
[**interior** INTERIOR_TYPE, F_FACTOR]

JULIA_FRACTAL:

julia V2_COMPLEX, I_ITERATIONS [**exponent** INT] [**exterior** EXTERIOR_TYPE,
F_FACTOR] [**interior** INTERIOR_TYPE, F_FACTOR]

MAGNET_MANDEL_FRACTAL:

magnet MAGNET_TYPE **mandel** I_ITERATIONS [**exterior** EXTERIOR_TYPE, F_FACTOR] [**interior** INTERIOR_TYPE, F_FACTOR]

MAGNET_TYPE:

1 | **2**

MAGNET_JULIA_FRACTAL:

magnet MAGNET_TYPE **julia** V2_COMPLEX, I_ITERATIONS [**exterior** EXTERIOR_TYPE, F_FACTOR] [**interior** INTERIOR_TYPE, F_FACTOR]

EXTERIOR_TYPE:

0 | **1** | **2** | **3** | **4** | **5** | **6**

INTERIOR_TYPE:

0 | **1** | **2** | **3** | **4** | **5** | **6**

IMAGE_PATTERN:

image_pattern {BITMAP_IMAGE [IMAGE_PATTERN_MODIFIERS] }

IMAGE_PATTERN_MODIFIERS:

[BITMAP_MODIFIERS] & [**use_alpha**]

QUILTED:

quilted [**control0** FLOAT] [**control1** FLOAT]

SLOPE:

slope { V_DIRECTION [, F_LOW_SLOPE, F_HIGH_SLOPE] [**altitude** VECTOR [, F_LOW_ALT, F_HIGH_ALT]] }

The **slope** pattern does not work in media densities.

Quick Reference Contents

7.10.8 Pattern Modifiers

Modify the evaluation of a pattern function:

PATTERN_MODIFIERS:

Jump to SDL

[TURBULENCE] & [WARP...] & [TRANSFORMATION...] & [**noise_generator** NG_TYPE]

NG_TYPE:

1 | **2** | **3**

TURBULENCE:

turbulence VECTOR & [**octaves** INT] & [**omega** FLOAT] & [**lambda** FLOAT]

WARP:

warp { WARP_ITEM }

WARP_ITEM:

REPEAT_WARP | BLACK_HOLE_WARP | TURBULENCE | CYLINDRICAL_WARP | SPHERICAL_WARP | TOROIDAL_WARP | PLANAR_WARP

REPEAT_WARP:

repeat VECTOR [**offset** VECTOR] [**flip** VECTOR]

BLACK_HOLE_WARP:

black_hole V_LOCATION, F_RADIUS [BLACK_HOLE_ITEMS]

BLACK_HOLE_ITEMS:

[**strength** FLOAT] & [**falloff** FLOAT] & [**inverse**] & [**repeat** VECTOR [**turbulence** VECTOR]]

CYLINDRICAL_WARP:

cylindrical [**orientation** VECTOR] [**dist_exp** FLOAT]

SPHERICAL_WARP:

spherical [**orientation** VECTOR] [**dist_exp** FLOAT]

TOROIDAL_WARP:

toroidal [**orientation** VECTOR] [**dist_exp** FLOAT] [**major_radius** FLOAT]

PLANAR_WARP:

planar [V_NORMAL, F_DISTANCE]

Modify the usage of a blend map:

BLEND_MAP_MODIFIERS:

Jump to SDL

frequency FLOAT & [**phase** FLOAT] & [WAVEFORM]

WAVEFORM:

Jump to SDL

ramp_wave | **triangle_wave** | **sine_wave** | **scallop_wave** | **cubic_wave** | **poly_wave** [F_EXPONENT]

Specify a two-dimensional bitmap image for a pattern:

BITMAP_IMAGE:

FUNCTION_IMAGE | BITMAP_TYPE FILE_NAME

FUNCTION_IMAGE:

Jump to SDL

function I_WIDTH, I_HEIGHT { FUNCTION_IMAGE_BODY }

FUNCTION_IMAGE_BODY:

PIGMENT | FN_FLOAT | **pattern** { PATTERN [PATTERN_MODIFIERS] }

PATTERN:

MAP_PATTERN | **brick** [BRICK_ITEMS] | **checker** | **hexagon** | **object** { LIST_OBJECT }

BITMAP_TYPE:

Jump to SDL

gif | **tga** | **iff** | **ppm** | **pgm** | **png** | **jpeg** | **tiff** | **sys**

Modify how a 2-D bitmap is to be applied to a 3-D surface:

BITMAP_MODIFIERS:

Jump to SDL

[**once**] & [**map_type** MAP_TYPE] & [**interpolate** INTERPOLATE_TYPE]

MAP_TYPE:

0 | **1** | **2** | **5**

INTERPOLATE_TYPE:

2 | **4**

Quick Reference Contents

7.11 Media

Describe particulate matter

MEDIA:

Jump to SDL

media { [MEDIA_IDENTIFIER] [MEDIA_ITEMS] }

MEDIA_ITEMS:

[**method** METHOD_TYPE] & [**intervals** INT] & [**samples** I_MIN, I_MAX] & [**confidence** FLOAT] & [**variance** FLOAT] & [**ratio** FLOAT] & [**absorption** COLOR] & [**emission** COLOR] & [**aa_threshold** FLOAT] & [**aa_level** INT] & [SCATTERING] & [DENSITY...] & [TRANSFORMATION...] & [**collect** BOOL]

METHOD_TYPE:

1 | **2** | **3**

SCATTERING:

scattering { SCATTERING_TYPE, COLOR [**eccentricity** FLOAT] [**extinction** FLOAT] }

SCATTERING_TYPE:

1 | **2** | **3** | **4** | **5**

DENSITY:

```
density { DENSITY_BODY }
```

DENSITY_BODY:

```
[DENSITY_IDENTIFIER] [DENSITY_TYPE] [PATTERN_MODIFIERS]
```

DENSITY_TYPE:

```
COLOR | COLOR_LIST_PATTERN | DENSITY_LIST_PATTERN | MAP_PATTERN [COLOR_MAP]
   | MAP_PATTERN DENSITY_MAP
```

DENSITY_LIST_PATTERN:

```
brick DENSITY, DENSITY [BRICK_ITEMS] |
   checker DENSITY, DENSITY |
   hexagon DENSITY, DENSITY, DENSITY |
   object { LIST_OBJECT DENSITY, DENSITY }
```

DENSITY_MAP:

```
density_map { DENSITY_MAP_BODY } [BLEND_MAP_MODIFIERS]
```

DENSITY_MAP_BODY:

```
DENSITY_MAP_IDENTIFIER | DENSITY_MAP_ENTRY...
```

There may be from 2 to 256 map entries.

DENSITY_MAP_ENTRY:

```
[ FLOAT DENSITY_BODY ]
```

The brackets here are part of the map entry.

Quick Reference Contents

7.12 Atmospheric Effects

Describe various background and atmospheric features

ATMOSPHERIC_EFFECT:

```
MEDIA | BACKGROUND | FOG | SKY_SPHERE | RAINBOW
```

Quick Reference Contents

7.12.1 Background

Specify a background color for the scene:

BACKGROUND:

```
background { COLOR }
```

Quick Reference Contents

7.12.2 Fog

Simulate a hazy or foggy atmosphere:

FOG:

Jump to SDL

`CONSTANT_FOG | GROUND_FOG`

CONSTANT_FOG:

fog { [FOG_IDENTIFIER] [**fog_type 1**] FOG_ITEMS }

FOG_ITEMS:

distance FLOAT & COLOR & [TURBULENCE [**turb_depth** FLOAT]]

GROUND_FOG:

fog { [FOG_IDENTIFIER] **fog_type 2** GROUND_FOG_ITEMS }

GROUND_FOG_ITEMS:

FOG_ITEMS & **fog_offset** FLOAT & **fog_alt** FLOAT & [**up** VECTOR [TRANSFORMATION...]]

Quick Reference Contents

7.12.3 Sky Sphere

Specify a sky pigment:

SKY_SPHERE:

sky_sphere { [SKY_SPHERE_IDENTIFIER] [SKY_SPHERE_ITEM...] }

SKY_SPHERE_ITEM:

`PIGMENT | TRANSFORMATION`

Quick Reference Contents

7.12.4 Rainbow

Specify a rainbow arc:

RAINBOW:

Jump to SDL

rainbow { [RAINBOW_IDENTIFIER] [RAINBOW_ITEMS] }

RAINBOW_ITEMS:

direction VECTOR & **angle** FLOAT & **width** FLOAT & **distance** FLOAT & COLOR_MAP & [**jitter** FLOAT] & [**up** VECTOR] & [**arc_angle** FLOAT] & [**falloff_angle** FLOAT]

Quick Reference Contents

7.13 Global Settings

Specify various settings that apply to the entire scene

GLOBAL_SETTINGS:

Jump to SDL

global_settings { GLOBAL_SETTING_ITEMS }

GLOBAL_SETTING_ITEMS:

[**adc_bailout** FLOAT] & [**ambient_light** COLOR] & [**assumed_gamma** FLOAT] & [**hf_gray_16** [BOOL]] & [**irid_wavelength** COLOR] & [**charset** GLOBAL_CHARSET] & [**max_intersections** INT] & [**max_trace_level** INT] & [**number_of_waves** INT] & [**noise_generator** NG_TYPE] & [RADIOSITY] & [PHOTONS]

GLOBAL_CHARSET:

ascii | **utf8** | **sys**

NG_TYPE:

1 | **2** | **3**

Quick Reference Contents

7.13.1 Radiosity

Enable radiosity to compute diffuse inter-reflection of light:

RADIOSITY:

Jump to SDL

radiosity { [RADIOSITY_ITEMS] }

RADIOSITY_ITEMS:

[**adc_bailout** FLOAT] & [**always_sample** BOOL] & [**brightness** FLOAT] & [**count** INT] & [**error_bound** FLOAT] & [**gray_threshold** FLOAT] & [**load_file** FILE_NAME] & [**low_error_factor** FLOAT] & [**max_sample** FLOAT] & [**media** BOOL] & [**minimum_reuse** FLOAT] & [**nearest_count** INT] & [**normal** BOOL] & [**pretrace_end** FLOAT] & [**pretrace_start** FLOAT] & [**recursion_limit** INT] & [**save_file** FILE_NAME]

Quick Reference Contents

7.13.2 Photons

Enable photon mapping to render reflective and refractive caustics:

PHOTONS:

Jump to SDL

photons { PHOTON_QUANTITY [PHOTON_ITEMS] }

PHOTON_QUANTITY:

spacing FLOAT | **count** INT

PHOTON_ITEMS:

[**gather** I_MIN, I_MAX] & [**media** I_MAX_STEPS [, F_FACTOR]] & [**jitter** FLOAT] & [**max_trace_level** INT] & [**adc_bailout** FLOAT] & [**save_file** FILE_NAME] & [**load_file** FILE_NAME] & [**autostop** FLOAT] & [**expand_thresholds** F_INCREASE, F_MIN] & [**radius** [FLOAT, FLOAT, FLOAT, FLOAT]]

Quick Reference Contents

Index

+a, 105
+am, 105
+b, 88
+c, 81
+d, 82
+ec, 80
+ef, 78
+ep, 85
+er, 80
+f, 86
+fc, 86
+fn, 86
+fp, 86
+fs, 86
+ga, 100
+gd, 100
+gf, 100
+gi, 81
+gr, 100
+gs, 100
+gw, 100
+h, 79, 102
+hi, 91
+hn, 89
+hs, 90
+ht, 89
+htc, 89
+htn, 89
+htp, 89
+hts, 89
+htt, 89
+htx, 89
+i, 91
+j, 105
+k, 76
+kc, 78
+kff, 76
+kfi, 76
+ki, 76
+l, 92
+mb, 103
+mv, 92
+o, 88
+p, 84
+q, 103
+r, 105
+sc, 80
+sf, 78
+sp, 85
+sr, 80
+su, 104
+ua, 86
+ud, 84
+uf, 79
+ul, 103
+uo, 79
+ur, 104
+uv, 103
+v, 84
+w, 79
+wl, 102
+x, 81
-a, 105
-b, 88
-c, 81
-d, 82
-f, 86
-ga, 100
-gd, 100
-gf, 100
-gr, 100
-gs, 100
-gw, 100
-j, 105
-kc, 78
-mb, 103
-p, 84
-su, 104
-ua, 86
-ud, 84
-uf, 79
-ul, 103

-uo, 79
-ur, 104
-uv, 103
-v, 84
-x, 81
#break, 66
#case, 66
#debug, 68
#declare, 56
#default, 63
#else, 65
#end, 65
#error, 68
#fclose, 61
#fopen, 60
#if, 65
#ifdef, 66
#ifndef, 66
#include, 56
#local, 56
#macro, 70
#range, 66
#read, 61
#render, 68
#statistics, 68
#switch, 66
#undef, 60
#version, 64
#warning, 68
#while, 67
#write, 62

aa_level, 295
aa_threshold, 295
abs, 25
accuracy, 173
    normals, 217
    parametric, 176
acos, 25
    julia, 149
acosd, 341
acosh, 25
    julia, 149
adaptive, 188
adc_bailout
    global_settings, 124
    photons, 303
    radiosity, 132
adj_range, 341
adj_range2, 342
agate_turb, 240
Align_Object, 352
Align_Trans, 378
all, 212
All_Console, 100
All_File, 101
all_intersections, 175
alpha, 212
altitude, 262
angle, 110
    camera, 110
    rainbow, 122
Animation
    cyclic, 78
    external loop, 76
    field rendering, 79
    internal loop, 76
    options, 76
    subsets of frames, 78
Antialias, 105
Antialias_Depth, 105
Antialias_Threshold, 105
aperture, 116
append, 60
arc_angle, 123
area_light, 187
Array
    declaring, 51
    identifiers, 51
    initialization, 52
array, 51
    quickref, 395
arrays
    quickref, 395
asc, 25
ascii, 129
asin, 25
    julia, 149
asind, 341
asinh, 26
    julia, 149
atan, 149
atan2, 26
atan2d, 341
atanh, 26
    julia, 149
atmosphere, 290
atmospheric effects
    quickref, 423
Axial_Scale_Trans, 377

Axis_Rotate_Trans, 377

b_spline, 154
background
  quickref, 423
Bevelled_Text, 352
bezier, 161
Bezier Patch, 161
bezier_spline, 150
bicubic_patch, 161
Bits_Per_Color, 86
black_hole, 272
blob
  component, 140
  component, cylinder, 139
  component, sphere, 139
Blobs, 138
blue, 40
Blur, 116
blur_samples, 117
BMP output, 86
boolean, 28
Bounding, 103
Bounding_Threshold, 103
break, 66
brick_size, 243
Buffer_Output, 88
Buffer_Size, 88
bump_map, 219

Camera
  coordinate system, 113
  focal blur, 116
  placing, 109
  types of, 114
case, 66
Caustics, 289
  simulated, 289
ceil, 26
Center_Object, 352
Center_Trans, 378
chr, 50
CHSL2RGB, 314
CHSV2RGB, 315
Circle_Text, 353
circular, 189
clamp, 341
clip, 341
clock, 29
clock_delta, 29
clock_on, 30
collect, 304
Color
  common pitfalls, 41
  functions, user-defined, 47
  identifiers, 40
  keywords, 40
  operators, 41
  specifying, 37
  vectors, 39
color, 38
  quickref, 392
color_map, 207
  density, 297
  rainbow, 122
colors
  quickref, 392
colour, 38
  quickref, 392
colour_map, 207
component, 140
composite, 178
concat, 50
conditional directives
  quickref, 399
confidence, 117
  focal blur, 117
  media, 295
conic_sweep, 153
Connect_Spheres, 354
conserve_energy, 228
constant fog, 120
Constructive Solid Geometry
  quickref, 411
contained_by, 173
  isosurface, 173
  parametric, 175
contents
  quickref, 388
Continue_Trace, 81
control0, 259
control1, 259
Convert_Color, 315
Coordinate system
  camera, 113
coords, 248
cos, 26
  julia, 149
cosd, 341
cosh, 26

julia, 149
crackle, 245
crand, 224
Create_Ini, 81
CRGB2HSL, 315
CRGB2HSV, 315
CRGBStr, 370
Cross Section Type, 320
CSG, 176
difference, 179
intersection, 179
merge, 180
union, 177
cube, 148
cubic, 169
cubic_spline, 150
cubic_wave, 270
cutaway_textures, 239
Cyclic_Animation, 78
cylinder
blob component, 139
light_source, 186
cylindrical
projection, 115
warp, 279

debug, 68
debug.inc, 318
Debug_Console, 100
Debug_File, 101
declare, 56
Declaring
arrays, 51
default, 63
Default Output Directory, 88
default texture
quickref, 398
Default values
bicubic_patch, 161
blob, 139
camera, 108
disc, 162
fog, 119
global settings, 124
height_field, 143
interior, 283
isosurface, 173
julia fractal, 147
lathe, 149
light_source, 182
media, 291
mesh, 163
parametric, 175
pattern modifiers, 267
photons, 302
poly, 170
prism, 151
rainbow, 122
sor, 156
sphere_sweep, 154
torus, 160
defined, 26
degrees, 26
density_map, 298
Depth of field, 116
df3, 247
dimension_size, 26
dimensions, 26
direction, 111
rainbow, 122
Directives
#language, #declare vs. #local, 58
language, 55
language, #declare, 56
language, #local, 56
language, #macro, 70
language, #version, 64
language, conditional, 65
language, default texture, 63
language, file I/O, 60
language, identifiers, 56
language, identifiers, destroying, 60
language, user messages, 68
Directory
default output, 88
dispersion
photons, 309
dispersion_samples, 288
Display, 82
Display_Gamma, 82
dist_exp, 279
distance, 120
fog, 120
rainbow, 122
div, 26
Divergence, 345
Draw_Vistas, 84
dynamic max_gradient, 174

eccentricity, 294
else, 65
end, 65
End_Column, 80
End_Row, 80
error, 68
    debug.inc, 318
eval_pigment, 335
evaluate, 174
even, 339
exp, 26
    julia, 149
expand_thresholds, 309
exponent, 227
    fractal pattern, 249
    reflection, 227
Expressions
    float, 22
    vector, 32
Extents, 352
exterior, 249
extinction, 293

f_algbr_cyl1, 321
f_algbr_cyl2, 321
f_algbr_cyl3, 322
f_algbr_cyl4, 322
f_bicorn, 322
f_bifolia, 322
f_blob, 323
f_blob2, 323
f_boy_surface, 323
f_comma, 323
f_cross_ellipsoids, 323
f_crossed_trough, 323
f_cubic_saddle, 323
f_cushion, 324
f_devils_curve, 324
f_devils_curve_2d, 324
f_dupin_cyclid, 324
f_ellipsoid, 324
f_enneper, 324
f_flange_cover, 324
f_folium_surface, 325
f_folium_surface_2d, 325
f_glob, 325
f_heart, 325
f_helical_torus, 325
f_helix1, 326
f_helix2, 326
f_hetero_mf, 326
f_hex_x, 326
f_hex_y, 326
f_hunt_surface, 327
f_hyperbolic_torus, 327
f_isect_ellipsoids, 327
f_kampyle_of_eudoxus, 327
f_kampyle_of_eudoxus_2d, 328
f_klein_bottle, 328
f_kummer_surface_v1, 328
f_kummer_surface_v2, 328
f_lemniscate_of_gerono, 328
f_lemniscate_of_gerono_2d, 328
f_mesh1, 329
f_mitre, 329
f_nodal_cubic, 329
f_noise3d, 329
f_noise_generator, 329
f_odd, 329
f_ovals_of_cassini, 329
f_parabolic_torus, 329
f_paraboloid, 329
f_ph, 330
f_pillow, 330
f_piriform, 330
f_piriform_2d, 330
f_poly4, 330
f_polytubes, 330
f_quantum, 331
f_quartic_cylinder, 331
f_quartic_paraboloid, 331
f_quartic_saddle, 331
f_r, 331
f_ridge, 331
f_ridged_mf, 331
f_rounded_box, 332
f_scallop_wave, 335
f_sine_wave, 335
f_snoise3d, 335
f_sphere, 332
f_spikes, 332
f_spikes_2d, 332
f_spiral, 333
f_sqr, 341
f_steiners_roman, 333
f_strophoid, 333
f_strophoid_2d, 333
f_superellipsoid, 334
f_th, 334
f_torus, 334

f_torus2, 334
f_torus_gumdrop, 334
f_umbrella, 334
f_witch_of_agnesi, 334
f_witch_of_agnesi_2d, 334
face_indices, 164
fade_color, 289
fade_colour, 289
fade_distance, 192
    interior, 289
    light_source, 192
fade_power, 192
    interior, 289
    light_source, 192
falloff, 183
    light_source, 183
    reflection, 227
    warp, 272
falloff_angle, 123
false, 28
Fatal_Console, 100
Fatal_Error_Command, 93
Fatal_Error_Return, 95
Fatal_File, 101
fclose, 61
Field_Render, 79
file i/o
    quickref, 397
file inclusion
    quickref, 397
file_exists, 26
filter, 38
    bitmap modfier, 212
Final_Clock
    ini-option, 76
final_clock, 30
Final_Frame
    ini-option, 76
final_frame, 30
Finding include files, 92
finish
    quickref, 418
fisheye, 115
flatness, 161
flip, 275
Float
    boolean, 28
    expressions, 22
    functions, 25
    functions, user-defined, 46
    identifiers, 23
    literals, 23
    operators, 24
float
    quickref, 388
float expressions
    quickref, 388
floats
    quickref, 388
floor, 26
fn_Divergence, 344
fn_Gradient, 344
fn_Gradient_Directional, 344
focal_point, 116
fog
    quickref, 424
fog_alt, 120
fog_offset, 120
fog_type, 120
fopen, 60
form, 245
fractal, 146
Fractal Object, 146
frame_number, 30
frequency, 269
fresnel, 227
function, 43
    as pattern, 251
    internal bitmap, 252
Functions, 43
    float, 25
    internal, 48
    string, 50
    user-defined, 43
    user-defined, color, 47
    user-defined, float, 46
    user-defined, vector, 47
    vector, 34

Gamma
    image file, 127
    monitor, 126
    scene file, 127
gamma
    determining your display, 83
    test image, 83
gather, 303
GetStats, 340
gif, 211
global settings

quickref, 425
global_lights, 193
global_settings, 123
Gradient_Directional, 345
Gradient_Length, 345
gray, 41
green, 40
ground fog, 120

Height, 79
height_field, 143
Hex_Tiles_Ptrn, 376
HF_Cylinder, 358
HF_Sphere, 357
HF_Square, 357
HF_Torus, 358
hierarchy, 140
blob, 140
height_field, 146
mesh, 164
Histogram, 340
Histogram_Grid_Size, 90
Histogram_Name, 89
Histogram_Type, 89
hypercomplex, 148

identifier
declaration, quickref, 397
Identifiers
array, 51
color, 40
declaring, 56
destroying, 60
float, 23
string, 49
vector, 33
if, 65
ifdef, 66
iff, 211
ifndef, 66
image_height, 30
image_map, 210
image_pattern, 255
image_width, 30
include, 56
standard files, 311
Include Files
finding, 92
Include Path, 92
Include_Header, 91
ini files
constant, 90
Initial_Clock
ini-option, 76
initial_clock, 30
Initial_Frame
ini-option, 76
initial_frame, 30
Initialization
arrays, 52
Input_File_Name, 91
inside, 26
inside_vector, 164
int, 26
interior
fade_distance, 289
fade_power, 289
fractal pattern, 249
quickref, 413
interior texture, 413
interior_texture, 238
internal, 48
functions.inc, 319
Interpolate
macro, 339
interpolate, 281
intervals, 294
ior, 287
irid, 228
Irregular_Bricks_Ptrn, 376
Isect, 351

jitter, 188
anti-aliasing , 105
area_light, 188
photons, 303
rainbow, 122
Jitter_Amount, 105
jpeg, 211
julia, 249
julia_fractal, 146

Keywords
color, 40

Language
identifiers, camera, 117
language
basics, quickref, 388
directives, quickref, 396

language basics
 quickref, 388
language directives
 quickref, 396
layered texture
 quickref, 414
Library_Path, 92
Light Sources
 and photons, 304
Light_Buffer, 103
light_group, 193
light_source, 181
 area_light, 187
 area_light, adaptive, 188
 area_light, circular, 189
 area_light, jitter, 188
 area_light, orient, 189
 cylinder, 186
 fade_distance, 192
 fade_power, 192
 looks_like, 191
 media_attenuation, 193
 media_interaction, 193
 parallel, 186
 projected_through, 191
 shadowless, 191
 spotlight, 182
 spotlight, falloff, 183
 spotlight, radius, 183
 spotlight, tightness, 183
lightgroup
 quickref, 402
lights
 quickref, 401
linear_spline, 150
linear_sweep, 152
Literals
 float, 23
 string, 48
ln, 26
 julia, 149
load_file, 135
 photons, 303
 radiosity, 135
local, 56
location, 109
log, 26
look_at, 109

macro, 70
 quickref, 399
Macros, 70
 declaring, 70
 invoking, 70
 return values in parameters, 74
 returning values from, 72
macros
 quickref, 399
magnet, 249
major_radius, 279
mandel, 249
mandelbrot, 249
map_type, 281
material
 quickref, 413
material_map, 232
Matrix_Trans, 377
max, 26
max3, 341
max_extent, 35
max_gradient, 173
 isosurface, 173
 parametric, 175
max_iteration, 147
max_trace, 175
max_trace_level
 photons, 303
Mean, 340
Media
 and photons, 305
media
 photons, 303
 quickref, 422
media_attenuation, 193
media_interaction, 193
message streams
 quickref, 399
metallic, 225
 highlight, 225
 reflection, 227
method, 295
metric, 245
min, 27
min3, 341
min_extent, 35
mod, 27
mortar, 243
Mountains
 generating with a height field, 143
Moving

camera, 109

natural_spline, 53
no, 28
no image, 200
no reflection, 200
no_bump_scale, 221
no_image, 200
no_reflection, 200
noise generator, 270
normal
quickref, 417
normal_indices, 164
normal_map, 218
normal_vectors, 164

object, 137
modifiers, quickref, 412
pattern, 257
object media, 290
object modifiers
quickref, 412
Objects
empty and solid, 285
objects
csg, quickref, 411
finite patch, quickref, 406
finite solid, quickref, 403
infinite solid, quickref, 409
isosurface, quickref, 410
parametric, quickref, 410
quickref, 402
odd, 339
Odd_Field, 79
off, 28
offset, 275
crackle, 245
repeat warp, 275
omnimax, 115
on, 28
once, 280
open, 283
isosurface, 174
Operators
color, 41
float, 24
promotion, 34
vector, 34
Options
animation, 76
anti-aliasing, 105
bounding, automatic, 103
bounding, manual, 104
display, 82
general output, 79
height and width, 79
help screen, 102
interruption, 80
partial output, 79
rendering, 102
resuming, 81
text output, 98
tracing, 102
Optios
parsing, 90
orient, 189
orientation, 279
orthographic, 114
Output
BMP, 86
PNG, 86
PPM, 86
system-specific, 86
Targa, compressed, 86
Targa, uncompressed, 86
Output File
placing in a default directory, 88
output formats, 86
Output_Alpha, 86
Output_File_Name, 88
Output_File_Type, 86
Output_to_File, 86

Palette, 82
panoramic, 115
parallel, 186
parametric, 175
Parse_String, 371
pass_through, 304
Path
includ files, 92
pattern, 46
quickref, 419
pattern modifiers
quickref, 420
patterned texture
quickref, 414
Pause_When_Done, 84
perspective, 114
Perturbation

camera ray, 117
pgm, 211
phase, 269
phong, 224
phong_size, 224
photons
dispersion, 309
media, 303
quickref, 425
pi, 28
pigment
quickref, 415
pigment_map, 209
pigment_pattern, 258
Pigments
color list, 207
color maps, 207
pigment list, 209
solid color, 206
Pitfalls
color, 41
plain texture
quickref, 414
planar
warp, 279
png, 211
PNG output, 86
point_at, 183
parallel, 186
spotlight, 183
Point_At_Trans, 378
poly, 169
poly_wave, 270
Post_Frame_Command, 93
Post_Frame_Return, 95
Post_Scene_Command, 93
Post_Scene_Return, 95
pot, 145
pow, 27
ppm, 211
PPM output, 86
Pre_Frame_Command, 93
Pre_Frame_Return, 95
Pre_Scene_Command, 93
Pre_Scene_Return, 95
precision, 147
precompute, 176
pretrace_end, 135
pretrace_start, 135
Preview_End_Size, 85
Preview_Start_Size, 85
prod, 44
projection
cylindrical, 115
fisheye, 115
omnimax, 115
orthographic, 114
panoramic, 115
perspective, 114
spherical, 116
ultra_wide_angle, 115
pwr, 149

Quad, 356
quadratic_spline, 150
Quality, 103
quartic, 169
quaternion, 148
quick_color, 213
quick_colour, 213
quickref, 387
arrays, 395
atmospheric effects, 423
background, 423
bitmap, 421
blend_map_modifiers, 421
brick_item, 415
camera, 401
colors, 392
conditional directives, 399
contents, 388
csg objects, 411
default texture, 398
dot_item, 391
embedded directives, 400
file i/o, 397
file inclusion, 397
finish, 418
finite patch objects, 406
finite solid objects, 403
floats, 388
fog, 424
function_invocation, 394
global settings, 425
identifier declaration, 397
infinite solid objects, 409
interior, 413
interior texture, 413
isosurface objects, 410
language basics, 388

language directives, 396
layered texture, 414
lightgroup, 402
lights, 401
list_object, 415
logical_expression, 390
macros, 399
material, 413
media, 422
message streams, 399
normal, 417
object modifiers, 412
objects, 402
parametric objects, 410
pattern, 419
pattern modifiers, 420
patterned texture, 414
photons, 425
pigment, 415
plain texture, 414
radiosity, 425
rainbow, 424
scene, 388
sky sphere, 424
splines, 396
strings, 394
texture, 413
transformations, 400
user-defined functions, 392
uv_mapping, 412
vectors, 391
version, 398

radians, 27
Radiosity
adjusting, 132
how it works, 131
tips, 136
radiosity
quickref, 425
radius, 183
light_source, 183
photons, 303
rainbow
quickref, 424
ramp_wave, 270
rand, 27
Rand_Array_Item, 311
Rand_Bernoulli, 350
Rand_Beta, 349
Rand_Binomial, 350
Rand_Cauchy, 348
Rand_Chi_Square, 349
Rand_Erlang, 350
Rand_Exp, 350
Rand_F_Dist, 349
Rand_Gamma, 349
Rand_Gauss, 349
Rand_Geo, 351
Rand_Lognormal, 350
Rand_Normal, 348
Rand_Pareto, 350
Rand_Poisson, 351
Rand_Spline, 349
Rand_Student, 348
Rand_Tri, 349
Rand_Weibull, 350
range, 66
ratio, 295
read, 61
reciprocal, 149
red, 40
refelection
metallic, 227
reflection, 226
exponent, 227
falloff, 227
reflection_exponent, 227
Remove_Bounds, 104
render, 68
Render_Console, 100
Render_File, 101
Reorient_Trans, 378
repeat, 275
Resize_Array, 311
Resolution, 79
Reverse_Array, 311
rgb, 39
rgbf, 40
rgbft, 40
rgbt, 40
right, 111
Rotate_Around_Trans, 377
roughness, 225
Round_Box_Union, 355
Round_Cone2_Union, 356
Round_Cone3_Union, 356
Round_Cone_Union, 355
Round_Cylinder_Union, 355
RRand, 347

samples, 295
Sampling_Method, 105
save_file, 135
  photons, 303
  radiosity, 135
scallop_wave, 270
scene
  quickref, 388
Scene Description Language, 15
Search Path, 92
seed, 27
select, 27
SetGradientAccuracy, 344
Settings
  global, 123
sgn, 341
shadowless, 191
Shear_Trans, 377
sin, 27
  julia, 149
sind, 341
sine_wave, 270
sinh, 27
  julia, 149
size, 248
sky, 110
sky sphere
  quickref, 424
sky_sphere, 121
slice, 147
slope_map, 215
smooth, 146
smooth_triangle, 167
solid, 245
solid triangle mesh, 164
sor, 156
Sort_Array, 312
Sort_Compare, 311
Sort_Partial_Array, 312
Sort_Swap_Data, 312
spacing, 302
spacing_multiplier, 304
specular, 225
sphere
  blob component, 139
sphere_sweep, 154
spherical
  projection, 116
  warp, 279
Spheroid, 354
spiral, 264
spline, 53
  quickref, 396
spline_trans, 379
splines
  quickref, 396
Split_Unions, 104
spotlight, 182
sqr, 148
sqrt, 28
SRand, 347
Star_Ptrn, 376
Start_Column, 80
Start_Row, 80
Statistic_Console, 100
Statistic_File, 101
statistics, 68
Std_Dev, 340
Str
  strings.inc, 370
str, 50
strcmp, 27
stream, 98
strength, 140
  black_hole warp, 272
  blob, 140
String
  functions, 50
  identifiers, 49
  literals, 48
string
  quickref, 394
String Literals, 48
strings
  quickref, 394
strlen, 27
strlwr, 50
strupr, 51
sturm
  lathe, 151
  prism, 153
  sor, 157
  torus, 160
Subset_End_Frame, 78
Subset_Start_Frame, 78
substr, 51
sum, 44
sunpos, 384
Supercone, 354
superellipsoid, 155

superquadric, 155
Supertorus, 354
switch, 66
sys, 129
    image_map, 211
System-specific output, 86

t, 34
tan, 28
    julia, 149
tand, 341
tanh, 28
    julia, 149
Targa output
    compressed, 86
    uncompressed, 86
target, 304
Test_Abort, 81
Test_Abort_Count, 81
Text_Space, 353
Text_Width, 353
texture, 203
    layered, quickref, 414
    patterned, quickref, 414
    plain, quickref, 414
    quickref, 413
texture-list, 164
texture_list, 238
texture_map, 230
Textures
    default, 63
tga, 211
The scene
    quickref, 388
thickness, 228
threshold, 139
    isosurface, 173
tiff, 211
tightness, 183
tile2, 231
Tiles_Ptrn, 376
tolerance, 154
toroidal, 279
    warp, 279
trace, 35
transformations
    quickref, 400
transmit, 38
    bitmap modifier, 212
triangle, 167
Triangle_Str, 371
triangle_Wave, 267
true, 28
ttf, 159
turb_depth, 121
turbulence
    fog, 121
    irid, 228
    warp, 276
type, 161

u, 34
u_steps, 161
ultra_wide_angle, 115
undef, 60
up, 111, 123
use_alpha, 255
use_color, 220
use_colour, 220
use_index, 220
user-defined functions
    quickref, 392
User_Abort_Command, 93
User_Abort_Return, 95
utf8, 129
uv_indices, 164
uv_mapping, 236
    quickref, 412
uv_vectors, 164

v, 34
v_steps, 161
val, 28
VAngle, 343
variable reflection, 227
variance, 117
    focal blur, 117
    media, 295
vaxis_rotate, 36
VCos_Angle, 343
vcross, 36
vCurl, 344
VDist, 343
vdot, 28
VDot5D, 343
Vector, 32
    built-in identifiers, 36
    color, 39
    expressions, 32
    functions, 34

functions, user-defined, 47
identifiers, 33
operators, 34
Vectors
direction, 111
sky, 110
VEq, 342
VEq5D, 342
Verbose, 84
Version
ini-option, 92
version, 64
quickref, 398
version identfier
quickref, 398
vertex_vectors, 164
vGradient, 344
Video_Mode, 82
Vista_Buffer, 103
vlength, 28
VLength5D, 342
VMin, 344
vnormalize, 36
VNormalize5D, 343
VPerp_Adjust, 343
VPerp_To_Plane, 343
VPerp_To_Vector, 343
VPow, 342
VPow5D, 342
VProject_Axis, 344
VProject_Plane, 343
VRand, 347
VRand_In_Box, 348
VRand_In_Obj, 348
VRand_In_Sphere, 348
VRand_On_Sphere, 348
vrotate, 36
VRotation, 343
VSqr, 342
Vstr
strings.inc, 370
vstr, 51
VStr2D, 370
Vstr2D, 370
vtransform, 379
vturbulence, 36
VWith_Len, 344
VZero, 342
VZero5D, 342
warning, 68
debug.inc, 318
Warning_Console, 100
Warning_File, 101
Warning_Level, 102
warp, 271
cylindrical, 279
falloff, 272
planar, 279
spherical, 279
toroidal, 279
water_level, 146
Wedge, 353
while, 67
Width, 79
width
rainbow, 122
Wire_Box_Union, 354
write, 62

x, 34

y, 34
yes, 28

z, 34