Persistence of Vision ™ Ray-Tracer

POV-Ray ™ Version 3.1

User's Documentation

Draft date June 24, 1998

1 Introduction ... 10

1.1 Program Description ... 11
1.2 What is Ray-Tracing? ... 11
1.3 What is POV-Ray? ... 11
1.4 How Do I Begin? ... 12
1.5 Notation and Basic Assumptions ... 12
1.6 What's New in POV-Ray 3.1? ... 13
1.6.1 Media Replaces Halo & Atmosphere ... 13
1.6.2 New #macro Feature ... 14
1.6.3 Arrays Added ... 14
1.6.4 File I/O and other Directives ... 14
1.6.5 Additional New Features ... 14

2 Beginning Tutorial ... 16

2.1 Our First Image ... 16
2.1.1 Understanding POV-Ray's Coordinate System ... 16
2.1.2 Adding Standard Include Files ... 17
2.1.3 Adding a Camera ... 18
2.1.4 Describing an Object ... 18
2.1.5 Adding Texture to an Object ... 19
2.1.6 Defining a Light Source ... 19
2.2 Simple Shapes ... 20
2.2.1 Box Object ... 20
2.2.2 Cone Object ... 20
2.2.3 Cylinder Object ... 20
2.2.4 Plane Object ... 21
2.3 CSG Objects ... 21
2.3.1 What is CSG? ... 21
2.3.2 CSG Union ... 22
2.3.3 CSG Intersection ... 23
2.3.4 CSG Difference ... 23
2.3.5 CSG Merge ... 25
2.3.6 CSG Pitfalls ... 25
2.3.6.1 Coincidence Surfaces ... 25
2.4 Advanced Shapes ... 25
2.4.1 Bicubic Patch Object ... 26
2.4.2 Blob Object ... 32
2.4.2.1 Component Types and Other New Features ... 34
2.4.2.2 Complex Blob Constructs and Negative Strength ... 34

2.4.3 Height Field Object ... 36
2.4.4 Lathe Object ... 38
2.4.4.1 Understanding The Concept of Splines ... 39
2.4.5 Mesh Object ... 43
2.4.6 Polygon Object ... 44
2.4.7 Prism Object ... 46
2.4.7.1 Teaching An Old Spline New Tricks ... 47
2.4.7.2 Smooth Transitions ... 48
2.4.7.3 Multiple Sub-Shapes ... 48
2.4.7.4 Conic Sweeps And The Tapering Effect ... 49
2.4.8 Superquadric Ellipsoid Object ... 52
2.4.9 Surface of Revolution Object ... 55
2.4.10 Text Object ... 57
2.4.11 Torus Object ... 60
2.5 The Light Source ... 65
2.5.1 The Pointlight Source ... 65
2.5.2 The Spotlight Source ... 66
2.5.3 The Cylindrical Light Source ... 67
2.5.4 The Area Light Source ... 67
2.5.5 The Ambient Light Source ... 68
2.5.6 Light Source Specials ... 69
2.5.6.1 Using Shadowless Lights ... 69
2.5.6.2 Assigning an Object to a Light Source ... 69
2.5.6.3 Using Light Fading ... 70
2.6 Simple Texture Options ... 70
2.6.1 Surface Finishes ... 71
2.6.2 Adding Bumpiness ... 71
2.6.3 Creating Color Patterns ... 71
2.6.4 Pre-defined Textures ... 72
2.7 Advanced Texture Options ... 72
2.7.1 Pigments ... 73
2.7.1.1 Using Color List Pigments ... 73
2.7.1.2 Using Pigment and Patterns ... 73
2.7.1.3 Using Pattern Modifiers ... 74
2.7.1.4 Using Transparent Pigments and Layered Textures ... 75
2.7.1.5 Using Pigment Maps ... 76
2.7.2 Normals ... 77
2.7.2.1 Using Basic Normal Modifiers ... 77
2.7.2.2 Blending Normals ... 78
2.7.3 Finishes ... 80
2.7.3.1 Using Ambient ... 80
2.7.3.2 Using Surface Highlights ... 81
2.7.3.3 Using Reflection and Metallic ... 82
2.7.3.4 Using Iridescence ... 83
2.7.4 Working With Pigment Maps ... 83
2.7.5 Working With Normal Maps ... 84
2.7.6 Working With Texture Maps ... 85
2.7.7 Working With List Textures ... 85
2.7.8 What About Tiles? ... 86
2.7.9 Average Function ... 87
2.7.10 Working With Layered Textures ... 87
2.7.10.1 Declaring Layered Textures ... 89
2.7.10.2 Another Layered Textures Example ... 89
2.7.11 When All Else Fails: Material Maps ... 92
2.7.12 Limitations Of Special Textures ... 94
2.8 Using the Camera ... 95

2.8.1 Using Focal Blur ... 95
2.9 Using Atmospheric Effects ... 96
2.9.1 The Background ... 97
2.9.2 The Sky Sphere ... 97
2.9.2.1 Creating a Sky with a Color Gradient ... 97
2.9.2.2 Adding the Sun ... 99
2.9.2.3 Adding Some Clouds ... 100
2.9.3 The Fog ... 101
2.9.3.1 A Constant Fog ... 101
2.9.3.2 Setting a Minimum Translucency ... 102
2.9.3.3 Creating a Filtering Fog ... 103
2.9.3.4 Adding Some Turbulence to the Fog ... 103
2.9.3.5 Using Ground Fog ... 104
2.9.3.6 Using Multiple Layers of Fog ... 105
2.9.3.7 Fog and Hollow Objects ... 106
2.9.4 The Rainbow ... 106
2.9.4.1 Starting With a Simple Rainbow ... 106
2.9.4.2 Increasing the Rainbow's Translucency ... 108
2.9.4.3 Using a Rainbow Arc ... 109
2.9.5 Animation ... 110
2.9.5.1 The Clock Variable: Key To It All ... 110
2.9.5.2 Clock Dependant Variables And Multi-Stage Animations ... 112
2.9.5.3 The Phase Keyword ... 113
2.9.5.4 Do Not Use Jitter Or Crand ... 114
2.9.5.5 INI File Settings ... 114

3 POV-Ray Options ... 116

3.1 Setting POV-Ray Options ... 116
3.1.1 Command Line Switches ... 116
3.1.2 Using INI Files ... 116
3.1.3 Using the POVINI Environment Variable ... 118
3.2 Options Reference ... 118
3.2.1 Animation Options ... 118
3.2.1.1 External Animation Loop ... 119
3.2.1.2 Internal Animation Loop ... 119
3.2.1.3 Subsets of Animation Frames ... 120
3.2.1.4 Cyclic Animation ... 120
3.2.1.5 Field Rendering ... 121
3.2.2 Output Options ... 121
3.2.2.1 General Output Options ... 121
3.2.2.1.1 Height and Width of Output ... 121
3.2.2.1.2 Partial Output Options ... 121
3.2.2.1.3 Interrupting Options ... 122
3.2.2.1.4 Resuming Options ... 122
3.2.2.2 Display Output Options ... 123
3.2.2.2.1 Display Hardware Settings ... 123
3.2.2.2.2 Display Related Settings ... 124
3.2.2.2.3 Mosaic Preview ... 125
3.2.2.3 File Output Options ... 125
3.2.2.3.1 Output File Type ... 126
3.2.2.3.2 Output File Name ... 126
3.2.2.3.3 Output File Buffer ... 127
3.2.2.4 CPU Utilization Histogram ... 127
3.2.2.4.1 File Type ... 127
3.2.2.4.2 File Name ... 128
3.2.2.4.3 Grid Size ... 128

3.2.3 Scene Parsing Options ... 128
3.2.3.1 Input File Name ... 129
3.2.3.2 Library Paths ... 129
3.2.3.3 Language Version ... 129
3.2.4 Shell-out to Operating System ... 130
3.2.4.1 String Substitution in Shell Commands ... 130
3.2.4.2 Shell Command Sequencing ... 131
3.2.4.3 Shell Command Return Actions ... 131
3.2.5 Text Output ... 133
3.2.5.1 Text Streams ... 134
3.2.5.2 Console Text Output ... 134
3.2.5.3 Directing Text Streams to Files ... 135
3.2.5.4 Help Screen Switches ... 136
3.2.6 Tracing Options ... 136
3.2.6.1 Quality Settings ... 136
3.2.6.2 Radiosity Setting ... 137
3.2.6.3 Automatic Bounding Control ... 137
3.2.6.4 Removing User Bounding ... 138
3.2.6.5 Anti-Aliasing Options ... 138
4 Scene Description Language ... 142
4.1 Language Basics ... 142
4.1.1 Identifiers and Keywords ... 142
4.1.2 Comments ... 144
4.1.3 Float Expressions ... 145
4.1.3.1 Float Literals ... 146
4.1.3.2 Float Identifiers ... 146
4.1.3.3 Float Operators ... 147
4.1.3.4 Built-in Float Identifiers ... 147
4.1.3.5 Boolean Keywords ... 149
4.1.3.6 Float Functions ... 149
4.1.4 Vector Expressions ... 151
4.1.4.1 Vector Literals ... 152
4.1.4.2 Vector Identifiers ... 152
4.1.4.3 Vector Operators ... 152
4.1.4.4 Operator Promotion ... 153
4.1.4.5 Built-in Vector Identifiers ... 153
4.1.4.6 Vector Functions ... 154
4.1.5 Specifying Colors ... 154
4.1.5.1 Color Vectors ... 155
4.1.5.2 Color Keywords ... 156
4.1.5.3 Color Identifiers ... 156
4.1.5.4 Color Operators ... 157
4.1.5.5 Common Color Pitfalls ... 157
4.1.6 Strings ... 158
4.1.6.1 String Literals ... 158
4.1.6.2 String Identifiers ... 158
4.1.6.3 String Functions ... 159
4.1.7 Array Identifiers ... 160
4.1.7.1 Declaring Arrays ... 160
4.1.7.2 Array Initalizers ... 161
4.2 Language Directives ... 161
4.2.1 Include Files and the #include Directive. ... 162
4.2.2 The #declare and #local Directives ... 162
4.2.2.1 Declaring identifiers ... 162
4.2.2.2 #declare vs. #local ... 163

4.2.2.3 Identifier Name Collisions ... 164
4.2.2.4 Destroying Identifiers with #undef ... 165
4.2.3 File I/O Directives ... 165
4.2.3.1 The #fopen Directive ... 166
4.2.3.2 The #fclose Directive ... 166
4.2.3.3 The #read Directive ... 166
4.2.3.4 The #write Directive ... 167
4.2.4 The #default Directive ... 167
4.2.5 The #version Directive ... 168
4.2.6 Conditional Directives ... 169
4.2.6.1 The #if...#else...#end Directives ... 169
4.2.6.2 The #ifdef and #ifndef Directives ... 169
4.2.6.3 The #switch, #case, #range and #break Directives ... 170
4.2.6.4 The #while...#end Directive ... 171
4.2.7 User Message Directives ... 171
4.2.7.1 Text Message Streams ... 171
4.2.7.2 Text Formatting ... 172
4.2.8 User Defined Macros ... 173
4.2.8.1 The #macro Directive ... 173
4.2.8.2 Invoking Macros ... 173
4.2.8.3 Are POV-Ray Macros a Function or a Macro? ... 174
4.2.8.4 Returning a Value Like a Function ... 175
4.2.8.5 Returning Values Via Parameters ... 176
4.3 POV-Ray Coordinate System ... 176
4.3.1 Transformations ... 177
4.3.1.1 Translate ... 177
4.3.1.2 Scale ... 177
4.3.1.3 Rotate ... 178
4.3.1.4 Matrix Keyword ... 178
4.3.2 Transformation Order ... 179
4.3.3 Transform Identifiers ... 179
4.3.4 Transforming Textures and Objects ... 180
4.4 Camera ... 181
4.4.1 Placing the Camera ... 181
4.4.1.1 Location and Look_At ... 182
4.4.1.2 The Sky Vector ... 182
4.4.1.3 Angle ... 183
4.4.1.4 The Direction Vector ... 183
4.4.1.5 Up and Right Vectors ... 183
4.4.1.5.1 Aspect Ratio ... 184
4.4.1.5.2 Handedness ... 184
4.4.1.6 Transforming the Camera ... 185
4.4.2 Types of Projection ... 185
4.4.3 Focal Blur ... 186
4.4.4 Camera Ray Perturbation ... 187
4.4.5 Camera Identifiers ... 187
4.5 Objects ... 188
4.5.1 Finite Solid Primitives ... 189
4.5.1.1 Blob ... 189
4.5.1.2 Box ... 191
4.5.1.3 Cone ... 191
4.5.1.4 Cylinder ... 192
4.5.1.5 Height Field ... 193
4.5.1.6 Julia Fractal ... 195
4.5.1.7 Lathe ... 197
4.5.1.8 Prism ... 199

4.5.1.9 Sphere ... 201
4.5.1.10 Superquadric Ellipsoid ... 201
4.5.1.11 Surface of Revolution ... 202
4.5.1.12 Text ... 204
4.5.1.13 Torus ... 205
4.5.2 Finite Patch Primitives ... 205
4.5.2.1 Bicubic Patch ... 206
4.5.2.2 Disc ... 207
4.5.2.3 Mesh ... 207
4.5.2.4 Polygon ... 208
4.5.2.5 Triangle and Smooth Triangle ... 209
4.5.3 Infinite Solid Primitives ... 210
4.5.3.1 Plane ... 210
4.5.3.2 Poly, Cubic and Quartic ... 210
4.5.3.3 Quadric ... 212
4.5.4 Constructive Solid Geometry ... 213
4.5.4.1 Inside and Outside ... 213
4.5.4.2 Union ... 214
4.5.4.3 Intersection ... 215
4.5.4.4 Difference ... 215
4.5.4.5 Merge ... 216
4.5.5 Light Sources ... 217
4.5.5.1 Point Lights ... 217
4.5.5.2 Spotlights ... 217
4.5.5.3 Cylindrical Lights ... 221
4.5.5.4 Area Lights ... 221
4.5.5.5 Shadowless Lights ... 223
4.5.5.6 Looks_like ... 223
4.5.5.7 Light Fading ... 223
4.5.5.8 Atmospheric Media Interaction ... 224
4.5.5.9 Atmospheric Attenuation ... 224
4.5.6 Object Modifiers ... 224
4.5.6.1 Clipped_By ... 225
4.5.6.2 Bounded_By ... 226
4.5.6.3 Inverse ... 227
4.5.6.4 Hollow ... 227
4.5.6.5 No_Shadow ... 227
4.5.6.6 Sturm ... 228
4.6 Interior ... 228
4.6.1 Why are Interior and Media Necessary? ... 229
4.6.2 Empty and Solid Objects ... 229
4.6.3 Refraction ... 230
4.6.4 Attenuation ... 230
4.6.5 Faked Caustics ... 231
4.6.6 Object Media ... 231
4.7 Textures ... 232
4.7.1 Pigment ... 233
4.7.1.1 Solid Color Pigments ... 234
4.7.1.2 Color List Pigments ... 235
4.7.1.3 Color Maps ... 235
4.7.1.4 Pigment Maps and Pigment Lists ... 236
4.7.1.5 Image Maps ... 237
4.7.1.5.1 Specifying an Image Map ... 238
4.7.1.5.2 The Filter and Transmit Bitmap Modifiers ... 238
4.7.1.5.3 Using the Alpha Channel ... 239
4.7.1.6 Quick Color ... 239

4.7.2 Normal ... 240
4.7.2.1 Slope Maps ... 241
4.7.2.2 Normal Maps and Normal Lists ... 243
4.7.2.3 Bump Maps ... 244
4.7.2.3.1 Specifying a Bump Map ... 244
4.7.2.3.2 Bump_Size ... 245
4.7.2.3.3 Use_Index and Use_Color ... 245
4.7.3 Finish ... 245
4.7.3.1 Ambient ... 246
4.7.3.2 Diffuse Reflection Items ... 247
4.7.3.2.1 Diffuse ... 247
4.7.3.2.2 Brilliance ... 247
4.7.3.2.3 Crand Graininess ... 248
4.7.3.3 Highlights ... 248
4.7.3.3.1 Phong Highlights ... 248
4.7.3.3.2 Specular Highlight ... 248
4.7.3.3.3 Metallic Highlight Modifier ... 249
4.7.3.4 Specular Reflection ... 249
4.7.3.5 Iridescence ... 250
4.7.4 Halo ... 251
4.7.5 Patterned Textures ... 251
4.7.5.1 Texture Maps ... 251
4.7.5.2 Tiles ... 253
4.7.5.3 Material Maps ... 253
4.7.5.3.1 Specifying a Material Map ... 253
4.7.6 Layered Textures ... 255
4.7.7 Patterns ... 256
4.7.7.1 Agate ... 256
4.7.7.2 Average ... 256
4.7.7.3 Boxed ... 257
4.7.7.4 Bozo ... 258
4.7.7.5 Brick ... 258
4.7.7.6 Bumps ... 259
4.7.7.7 Checker ... 259
4.7.7.8 Crackle ... 259
4.7.7.9 Cylindrical ... 260
4.7.7.10 Density_File ... 260
4.7.7.11 Dents ... 260
4.7.7.12 Gradient ... 260
4.7.7.13 Granite ... 261
4.7.7.14 Hexagon ... 261
4.7.7.15 Leopard ... 262
4.7.7.16 Mandel ... 262
4.7.7.17 Marble ... 263
4.7.7.18 Onion ... 263
4.7.7.19 Planar ... 263
4.7.7.20 Quilted ... 264
4.7.7.21 Radial ... 266
4.7.7.22 Ripples ... 266
4.7.7.23 Spherical ... 267
4.7.7.24 Spiral1 ... 267
4.7.7.25 Spiral2 ... 267
4.7.7.26 Spotted ... 267
4.7.7.27 Waves ... 268
4.7.7.28 Wood ... 268
4.7.7.29 Wrinkles ... 268

4.7.8 Pattern Modifiers ... 268
4.7.8.1 Transforming Patterns ... 270
4.7.8.2 Frequency and Phase ... 270
4.7.8.3 Waveforms ... 271
4.7.8.4 Turbulence ... 271
4.7.8.5 Octaves ... 272
4.7.8.6 Lambda ... 273
4.7.8.7 Omega ... 273
4.7.8.8 Warps ... 273
4.7.8.8.1 Black Hole Warp ... 273
4.7.8.8.2 Repeat Warp ... 276
4.7.8.8.3 Turbulence Warp ... 277
4.7.8.9 Bitmap Modifiers ... 278
4.7.8.9.1 The once Option ... 278
4.7.8.9.2 The map_type Option ... 278
4.7.8.9.3 The interpolate Option ... 279
4.8 Media ... 279
4.8.1 Media Types ... 280
4.8.1.1 Absorption ... 280
4.8.1.2 Emission ... 281
4.8.1.3 Scattering ... 281
4.8.2 Sampling Parameters ... 283
4.8.3 Density ... 284
4.8.3.1 General Density Modifiers ... 284
4.8.3.2 Density with color_map ... 285
4.8.3.3 Density Maps and Density Lists ... 285
4.8.3.4 Multiple Density vs Multiple Media ... 286
4.9 Atmospheric Effects ... 287
4.9.1 Atmospheric Media ... 287
4.9.2 Background ... 287
4.9.3 Fog ... 287
4.9.4 Sky Sphere ... 288
4.9.5 Rainbow ... 289
4.10 Global Settings ... 290
4.10.1 ADC_Bailout ... 291
4.10.2 Ambient Light ... 291
4.10.3 Assumed_Gamma ... 291
4.10.3.1 Monitor Gamma ... 292
4.10.3.2 Image File Gamma ... 292
4.10.3.3 Scene File Gamma ... 293
4.10.4 HF_Gray_16 ... 293
4.10.5 Irid_Wavelength ... 294
4.10.6 Max_Trace_Level ... 294
4.10.7 Max_Intersections ... 295
4.10.8 Number_Of_Waves ... 295
4.10.9 Radiosity ... 295
4.10.9.1 How Radiosity Works ... 295
4.10.9.2 Adjusting Radiosity ... 296
4.10.9.2.1 brightness ... 296
4.10.9.2.2 count ... 296
4.10.9.2.3 distance_maximum ... 297
4.10.9.2.4 error_bound ... 297
4.10.9.2.5 gray_threshold ... 297
4.10.9.2.6 low_error_factor ... 297
4.10.9.2.7 minimum_reuse ... 298
4.10.9.2.8 nearest_count ... 298

4.10.9.2.9 recursion_limit ... 298
4.10.9.3 Tips on Radiosity ... 298
5 APPENDICES ... 299
5.1 Copyright, Legal Information and License -- POVLEGAL.DOC ... 299
5.1.1 General License Agreement -- POVLEGAL.DOC ... 299
5.1.2 Usage Provisions ... 299
5.1.3 General Rules For All Distribution ... 299
5.1.4 Definition Of "Full Package" ... 300
5.1.4.1 Conditions For Shareware/Freeware Distribution Companies ... 300
5.1.5 Conditions For On-Line Services And Bbs's Including Internet ... 300
5.1.6 Online Or Remote Execution Of POV-Ray ... 301
5.1.7 Permitted Modification And Custom Versions ... 301
5.1.8 Conditions For Distribution Of Custom Versions ... 302
5.1.9 Conditions For Commercial Bundling ... 302
5.1.10 POV-Team Endorsement Prohibitions ... 303
5.1.11 Retail Value Of This Software ... 303
5.1.12 Other Provisions ... 303
5.1.13 Revocation Of License ... 304
5.1.14 Disclaimer ... 304
5.1.15 Technical Support ... 304
5.2 Authors ... 304
5.2.1 Contacting the Authors ... 306
5.3 What to do if you don't have POV-Ray ... 306
5.3.1 Which Version of POV-Ray should you use? ... 307
5.3.1.1 Microsoft Windows 95/98/NT ... 307
5.3.1.2 MS-Dos & Windows 3.x ... 307
5.3.1.3 Linux for Intel x86 ... 308
5.3.1.4 Apple Macintosh ... 308
5.3.1.5 Amiga ... 309
5.3.1.6 SunOS ... 309
5.3.1.7 Generic Unix ... 310
5.3.1.8 All Versions ... 310
5.3.2 Where to Find POV-Ray Files ... 311
5.3.2.1 POV-Ray Forum on CompuServe ... 311
5.3.2.2 World Wide Website www.povray.org ... 311
5.3.2.3 Books, Magazines and CD-ROMs ... 311
5.4 Compiling POV-Ray ... 312
5.4.1 Directory Structure ... 312
5.4.2 Configuring POV-Ray Source ... 313
5.4.3 Conclusion ... 314
5.5 Suggested Reading ... 314
6 Index ... 315

1 Introduction

This document details the use of the Persistence of Vision ™ Ray-Tracer (POV-Ray ™). It is divided into five parts:

1) This introduction which explains what POV-Ray is and what ray-tracing is. It gives a brief overview of how to create ray-traced images.
2) A "Beginning Tutorial" which explains step by step how to use the different features of POV-Ray.
3) A complete reference on "Scene Description Language" in which you describe the scene.
4) A complete reference on "POV-Ray Options" which explains options (set either by command line switches or by INI file keywords) that tell POV-Ray how to render the scenes.
5) And in our "APPENDICES" you will find some tips and hints, where to get the latest version and versions for other platforms, information on compiling custom versions of POV-Ray, suggested reading, contact addresses and legal information.

POV-Ray runs on MS-Dos, Windows 3.x, Windows for Workgroups 3.11, Windows 95, Windows NT, Apple Macintosh 68k, Macintosh Power PC, Amiga, Linux, Sun-OS, UNIX and other platforms.

We assume that if you are reading this document then you already have POV-Ray installed and running. However the POV-Team does distribute this file by itself in various formats including online on the internet. If you don't have POV-Ray or aren't sure you have the official version or the latest version, see appendix "What to do if you don't have POV-Ray".

This document covers only the generic parts of the program which are common to each version. **Each version has platform-specific documentation not included here.** We recommend you finish reading this introductory section then read the platform-specific information before trying the tutorial here.

The platform-specific docs will show you how to render a sample scene and will give you detailed description of the platform-specific features.

The MS-Dos version documentation contains a plain text file `POVMSDOS.DOC` which contains its specific docs. It can be found in the main directory where you installed POV-Ray such as `C:\POVRAY3`.

The Windows version documentation is available on the POV-Ray program's Help menu or press the F1 key while in the program.

The Mac platform documentation consists of a self-displaying document called "`POV-Ray MacOS Read Me`" which contains information specific to the Mac version of POV-Ray. It is best to read this document first, to learn how to set up and start using the Mac version of POV-Ray. This document is in the "`Documentation`" folder in the main "`POV-Ray 3`" folder.

The Amiga version documentation is made up of Html documents, stored in the same place of general Pov docs; the 'root' document is "POVRAY3:POV-Reference/AmigaPOV.html" when the program is installed following the instruction given. Amiga specific documentation and Pov general docs are avaible pressing Help key in the Pov-Gui program; this feature is avaible in Pov-Gui since version 2.1

The Linux version documentation contains a plain text file `povlinux.doc` which contains its specific docs. It can be found in the main directory where you installed POV-Ray such as `/usr/povray3`.

The SunOS version documentation contains a plain text file `povsunos.doc` which contains its specific docs. It can be found in the main directory where you installed POV-Ray such as `/usr/povray3`.

The generic Unix version documentation contains a plain text file `povunix.doc` which contains its specific docs. It can be found in the main directory where you installed POV-Ray such as `/usr/povray3`.

1.1 Program Description

The Persistence of Vision™ Ray-Tracer creates three-dimensional, photo-realistic images using a rendering technique called ray-tracing. It reads in a text file containing information describing the objects and lighting in a scene and generates an image of that scene from the view point of a camera also described in the text file. Ray-tracing is not a fast process by any means, but it produces very high quality images with realistic reflections, shading, perspective and other effects.

1.2 What is Ray-Tracing?

Ray-tracing is a rendering technique that calculates an image of a scene by simulating the way rays of light travel in the real world. However it does its job backwards. In the real world, rays of light are emitted from a light source and illuminate objects. The light reflects off of the objects or passes through transparent objects. This reflected light hits our eyes or perhaps a camera lens. Because the vast majority of rays never hit an observer, it would take forever to trace a scene.

Ray-tracing programs like POV-Ray start with their simulated camera and trace rays backwards out into the scene. The user specifies the location of the camera, light sources, and objects as well as the surface texture properties of objects, their interiors (if transparent) and any atmospheric media such as fog, haze, or fire.

For every pixel in the final image one or more viewing rays are shot from the camera, into the scene to see if it intersects with any of the objects in the scene. These "viewing rays" originate from the viewer, represented by the camera, and pass through the viewing window (representing the final image).

Every time an object is hit, the color of the surface at that point is calculated. For this purpose rays are sent backwards to each light source to determine the amount of light coming from the source. These "shadow rays" are tested to tell whether the surface point lies in shadow or not. If the surface is reflective or transparent new rays are set up and traced in order to determine the contribution of the reflected and refracted light to the final surface color.

Special features like inter-diffuse reflection (radiosity), atmospheric effects and area lights make it necessary to shoot a lot of additional rays into the scene for every pixel.

1.3 What is POV-Ray?

The Persistence of Vision ™ Ray-Tracer was developed from DKBTrace 2.12 (written by David K. Buck and Aaron A. Collins) by a bunch of people, called the POV-Team ™, in their spare time. The headquarters of the POV-Team is in the POVRAY forum on CompuServe (see "POV-Ray Forum on CompuServe" for more details).

The POV-Ray ™ package includes detailed instructions on using the ray-tracer and creating scenes. Many stunning scenes are included with POV-Ray so you can start creating images immediately when you get the package. These scenes can be modified so you don't have to start from scratch.

In addition to the pre-defined scenes, a large library of pre-defined shapes and materials is provided. You can include these shapes and materials in your own scenes by just including the library file name at the top of your scene file, and by using the shape or material name in your scene.

Here are some highlights of POV-Ray's features:

* Easy to use scene description language.
* Large library of stunning example scene files.
* Standard include files that pre-define many shapes, colors and textures.
* Very high quality output image files (up to 48-bit color).
* 15 and 24 bit color display on many computer platforms using appropriate hardware.
* Create landscapes using smoothed height fields.
* Many camera types, including perspective, panorama, orthographic, fisheye, etc.
* Spotlights, cylindrical lights and area lights for sophisticated lighting.
* Phong and specular highlighting for more realistic-looking surfaces.

* Inter-diffuse reflection (radiosity) for more realistic lighting.
* Atmospheric effects like atmosphere, ground-fog and rainbow.
* Particle media to model effects like clouds, dust, fire and steam.
* Several image file output formats including Targa, PNG and PPM.
* Basic shape primitives such as ... spheres, boxes, quadrics, cylinders, cones, triangles and planes.
* Advanced shape primitives such as ... Torii (donuts), bezier patches, height fields (mountains), blobs, quartics, smooth triangles, text, fractals, superquadrics, surfaces of revolution, prisms, polygons, lathes and fractals.
* Shapes can easily be combined to create new complex shapes using Constructive Solid Geometry (CSG). POV-Ray supports unions, merges, intersections and differences.
* Objects are assigned materials called textures (a texture describes the coloring and surface properties of a shape) and interior properties such as index of refraction and particle media (formerly known as "halos").
* Built-in color and normal patterns: Agate, Bozo, Bumps, Checker, Crackle, Dents, Granite, Gradient, Hexagon, Leopard, Mandel, Marble, Onion, Quilted, Ripples, Spotted, Spiral, Radial, Waves, Wood, Wrinkles and image file mapping.
* Users can create their own textures or use pre-defined textures such as ... Brass, Chrome, Copper, Gold, Silver, Stone, Wood.
* Combine textures using layering of semi-transparent textures or tiles of textures or material map files.
* Display preview of image while rendering (not available on all platforms).
* Halt and save a render part way through, and continue rendering the halted partial render later.

1.4 How Do I Begin?

POV-Ray scenes are described in a special text language called a "scene description language". You will type commands into a plain text file and POV-Ray will read it to create the image. The process of running POV-Ray is a little different on each platform or operating system. You should read the platform-specific documentation as suggested earlier in this introduction. It will tell you how to command POV-Ray to turn your text scene description into an image. You should try rendering several sample images before attempting to create your own.

Once you know how to run POV-Ray on your computer and your operating system, you can proceed with the tutorial which follows. The tutorial explains how to describe the scene using the POV-Ray language.

1.5 Notation and Basic Assumptions

Throughout the tutorial and reference section of this document, the following notation is used to mark keywords of the scene description language, command line switches, INI file keywords and file names.

`keyword`	mono-spaced bold	POV-Ray keywords and punctuation
`+W640 +H480`	mono-spaced bold	command-line switches
`C:\MYFILE.POV`	mono-spaced	file names, directories, paths
SYNTAX_ITEM	italics, all caps	required syntax item
[SYNTAX_ITEM]	italics, all caps, braces	optional syntax item
SYNTAX_ITEM...	italics, all caps, ellipsis	one or more syntax items
[SYNTAX_ITEM...]	italics, all caps, braces, ellipsis	zero or more syntax items
Value_1	italics, mixed case	a float value or expression
<Value_1>	italics, mixed case, angle braces	a vector value or expression
[*ITEM* **]**	bold square braces	ITEM enclosed in required braces

ITEM1 | *ITEM2*	vertical bar	choice of *ITEM1* or *ITEM2*

In the plain ASCII version of the document there is no visible difference between the different notations.

Note that POV-Ray is a command-line program on MS-Dos, Unix and other text-based operating system and is menu-driven on Windows and Macintosh platforms. Some of these operating systems use folders to store files while others use directories. Some separate the folders and sub-folders with a slash character (`/`), back-slash character (`\`), or others. We have tried to make this documentation as generic as possible but sometimes we have to refer to folders, files, options etc. and there's no way to escape it. Here are some assumptions we make...

1) You installed POV-Ray in the "`C:\POVRAY3`" directory. For MS-Dos this is probably true but for Unix it might be "`/usr/povray3`", or for Windows it might be "`C:\Program Files\POV-Ray for Windows`", for Mac it might be "`MyHD:Apps:POV-Ray 3:`", or you may have used some other drive or directory. So if we tell you that "Include files are stored in the `\povray3\include` directory," we assume you can translate that to something like "`::POVRAY3:INCLUDE`" or "`C:\Program Files\POV-Ray for Windows\include`" or whatever is appropriate for your platform, operating system and installation.

2) POV-Ray uses command-line switches and INI files to choose options in all versions but Windows and Mac also use dialog boxes or menu choices to set options. We will describe options assuming you are using switches or INI files when describing what the options do. We have taken care to use the same terminology in designing menus and dialogs as we use in describing switches or INI keywords. See your version-specific documentation on menu and dialogs.

3) Some of you are reading this using a help-reader, built-in help, web-browser, formatted printout, or plain text file. We assume you know how to get around in which ever medium you're using. We'll say "See the chapter on "Setting POV-Ray Option" we assume you can click, scroll, browse, flip pages or whatever to get there.

1.6 What's New in POV-Ray 3.1?

Here is an overview of what is new in POV-Ray 3.1 since version 3.0.

1.6.1 Media Replaces Halo & Atmosphere

The keywords **`halo`** and **`atmosphere`** have been totally eliminated with no backwards compatability of any kind provided. They have been replaced by a new feature called **`media`**. At the scene level, media acts as atmospheric media for fog, haze, dust, etc. On objects, media is not part of texture like halo was. Object media is now part of a new feature called **`interior`**. Media is not just a rename for halo. It is a new model with some similar features of halo. BECAUSE POV-Ray 3.1 DISCONTINUES SOME 3.0 FEATURES YOU MAY WISH TO KEEP 3.0 TO RENDER OLDER SCENES.

Any pattern type (**`bozo`**, **`wood`**, **`dents`**, etc.) may be used as a **`density`** function for media.

New patterns **`spherical`**, **`cylindrical`**, **`planar`**, and **`boxed`** added for **`pigment`**, **`normal`**, **`texture`**, and **`density`**.

New wave types **`cubic_wave`** and **`poly_wave`** *Float* have been added.

New object modifier **`interior{...}`**. Interior 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 **`finish`** items moved are **`ior`**, **`caustic`**, **`fade_power`**, and **`fade_depth`**. The **`refraction`** keyword is no longer necessary. Any **`ior`** other than 1.0 turns on refraction.

These 5 finish keywords which are now part of interior will still work in finish but will generate warnings. Some obscure **texture_map** statements with varying ior will not work.

Added **reflection_exponent** *Float* to **finish** to give more realistic reflection of very bright objects.

1.6.2 New #macro Feature

Add fully recursive and parameterized **#macro** directive. Define like this...

```
#macro MyMacro (P1,P2,P3)   ...  #end
```

Invoke like this...

```
MyMacro (5,x*5,MyTexture)
```

Note no '#' sign preceeds invocation. Macros can be invoked almost anywhere. Parameters must be identifiers or any item that can be declared, **MyMacro(pigment{Green},MyObject)** for example.

Added **#local** *IDENTIFIER***=** *STATEMENT* as alternative to **#declare** to create temporary local identifier in macros or include files.

1.6.3 Arrays Added

Added multi-dimension arrays

```
#declare MyArray=array[20]
```

or

```
#local PrivateArray=array[30]
```

or

```
#declare Rows=5; #declare Cols=4;
#declare Table=array[Rows][Cols]
```

Added optional initializer syntax for arrays.

```
#declare MyArray=array[2][3]{{1,2,3},{4,5,6}}
```

Subscripts start at 0. Anything that can be declared may be in an array. Arrays are initialized as null. You must later fill each element with values.

Added float functions for arrays. Given **#declare MyArray = array[4][5]** then **dimensions(MyArray)** is 2 and **dimension_size(MyArray,2)** is 5.

1.6.4 File I/O and other Directives

Added **#fopen**, **#fclose**, **#read**, and **#write** directives for user text files.

Added **#undef** identifier directive. Un-declares previously declared identifier. Works on locals or globals.

Added requirement that any directive which can end in a float or expression must be terminated by a semi-colon. Specifically this means any **#declare** or **#local** of float, vector or color or the **#version** directive.

1.6.5 Additional New Features

Added Bezier splines to **lathe** and **prism**. The spline is made of segments having four points each. Thus there are always four times the number of segments in a prism or lathe. A four point Bezier spline uses 3rd order Bernstein blending functions which are sufficient for smooth curves.

Added float constant **`clock_delta`** returns time between frames.

2 Beginning Tutorial

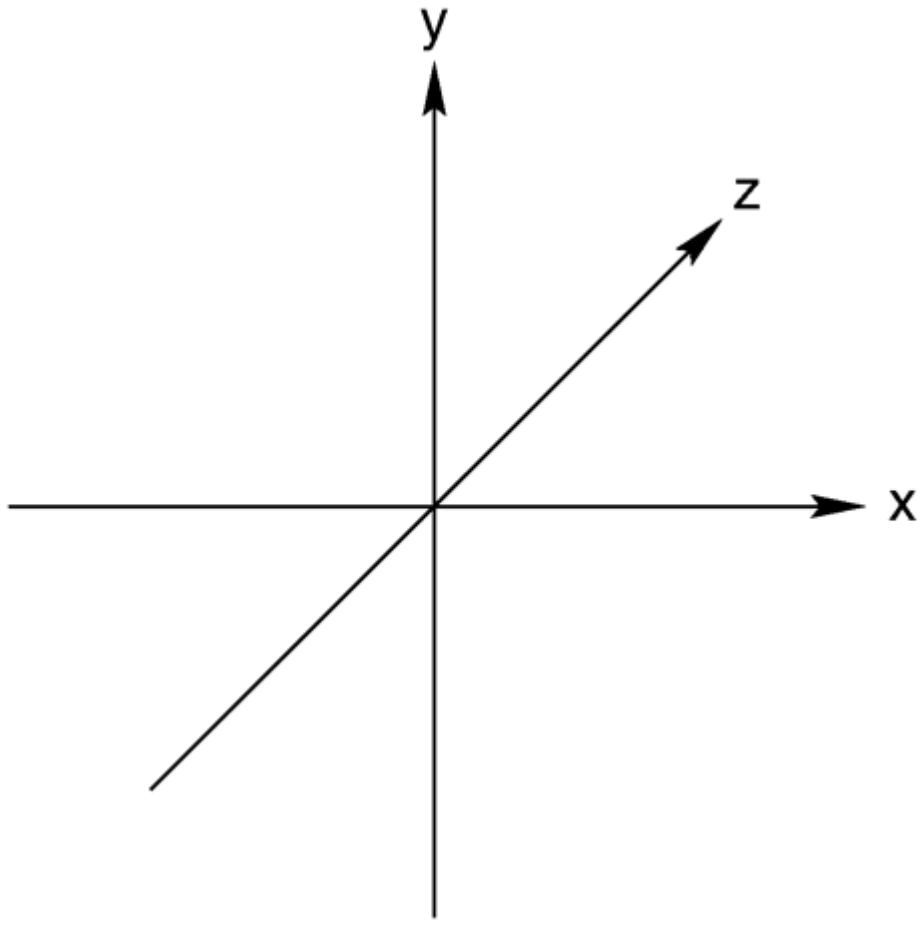
The beginning tutorial explains step by step how to use POV-Ray's scene description language to create own your scenes. The use of almost every feature of POV-Ray's language is explained in detail. We will learn basic things like placing cameras and light sources. We will also learn how to create a large variety of objects and how to assign different textures to them. The more sophisticated features like radiosity, interior, media and atmospheric effects will be explained in detail.

2.1 Our First Image

We will create the scene file for a simple picture. Since ray-tracers thrive on spheres, that is what we will render first.

2.1.1 Understanding POV-Ray's Coordinate System

First, we have to tell POV-Ray where our camera is and where it is looking. To do this, we use 3D coordinates. The usual coordinate system for POV-Ray has the positive y-axis pointing up, the positive x-axis pointing to the right, and the positive z-axis pointing into the screen as follows:

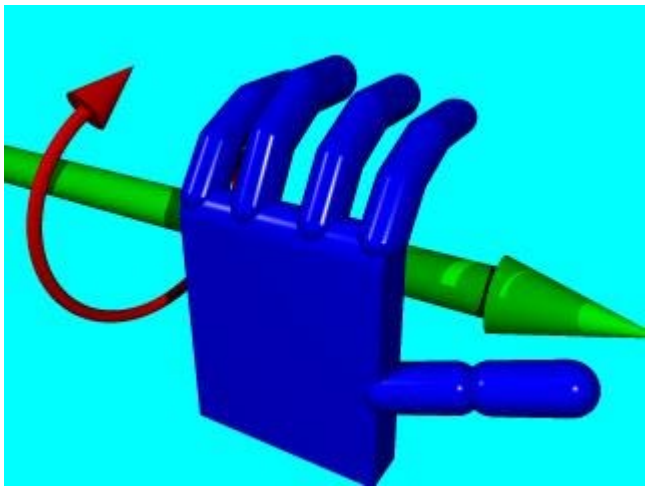


The left-handed coordinate system (the z-axis is pointing away)

This kind of coordinate system is called a left-handed coordinate system. If we use our left hand's fingers we can easily see why it is called left-handed. We just point our thumb in the direction of the positive x-axis (to the right),

the index finger in the direction of the positive y-axis (straight up) and the middle finger in the positive z-axis direction (forward). We can only do this with our left hand. If we had used our right hand we would not have been able to point the middle finger in the correct direction.

The left hand can also be used to determine rotation directions. To do this we must perform the famous "*Computer Graphics Aerobics*" exercise. We hold up our left hand and point our thumb in the positive direction of the axis of rotation. Our fingers will curl in the positive direction of rotation. Similarly if we point our thumb in the negative direction of the axis our fingers will curl in the negative direction of rotation.



"Computer Graphics Aerobics" to determine the rotation direction.

In the above illustration, the left hand is curling around the x-axis. The thumb points in the positive x direction and the fingers curl over in the positive rotation direction.

If we want to use a right-handed system, as some CAD systems and modelers do, the **`right`** vector in the camera specification needs to be changed. See the detailed description in "Handedness". In a right-handed system we use our right hand for the "Aerobics".

There is some controversy over whether POV-Ray's method of doing a right-handed system is really proper. To avoid problems we stick with the left-handed system which is not in dispute.

2.1.2 Adding Standard Include Files

Using our personal favorite text editor, we create a file called `demo.pov`. Note some versions of POV-Ray come with their own built-in text editor which may be easier to use. We then type in the following text. The input is case sensitive, so we have to be sure to get capital and lowercase letters correct.

```
#include "colors.inc"    // The include files contain
#include "stones.inc"    // pre-defined scene elements
```

The first include statement reads in definitions for various useful colors. The second include statement reads in a collection of stone textures.

POV-Ray comes with many standard include files. Others of interest are:

```
#include "textures.inc"    // pre-defined scene elements
#include "shapes.inc"
#include "glass.inc"
#include "metals.inc"
#include "woods.inc"
```

They read pre-defined textures, shapes, glass, metal, and wood textures. It is a good idea to have a look through them to see a few of the many possible shapes and textures available.

We should only include files we really need in our scene. Some of the include files coming with POV-Ray are quite large and we should better save the parsing time and memory if we don't need them. In the following examples we will only use the `colors.inc`, and `stones.inc` include files.

We may have as many include files as needed in a scene file. Include files may themselves contain include files, but we are limited to declaring includes nested only ten levels deep.

Filenames specified in the include statements will be searched for in the current directory first. If it fails to find your .Inc files in the current directory, POV-Ray searches any "library paths" that you have specified. Library paths are options set by the **+L** command-line switch or **Library_Path** option. See the chapter "Setting POV-Ray Options" for more information on library paths.

Because it is more useful to keep include files in a separate directory, standard installation of POV-Ray place these files in the \povray3\include directory. If you get an error message saying that POV-Ray cannot open "`colors.inc`" or other include files, make sure that you specify the library path properly.

2.1.3 Adding a Camera

The **camera** statement describes where and how the camera sees the scene. It gives x-, y- and z-coordinates to indicate the position of the camera and what part of the scene it is pointing at. We describe the coordinates using a three-part *vector*. A vector is specified by putting three numeric values between a pair of angle brackets and separating the values with commas.

We add the following camera statement to the scene.

```
camera {
  location <0, 2, -3>
  look_at  <0, 1,  2>
}
```

Briefly, **location <0,2,-3>** places the camera up two units and back three units from the center of the ray-tracing universe which is at <0,0,0>. By default +z is into the screen and -z is back out of the screen.

Also **look_at <0,1,2>** rotates the camera to point at the coordinates <0,1,2>. A point 1 unit up from the origin and 2 units away from the origin. This makes it 5 units in front of and 1 unit lower than the camera. The **look_at** point should be the center of attention of our image.

2.1.4 Describing an Object

Now that the camera is set up to record the scene, let's place a yellow sphere into the scene. We add the following to our scene file:

```
sphere {
  <0, 1, 2>, 2
  texture {
    pigment { color Yellow }
  }
}
```

The first vector specifies the center of the sphere. In this example the x coordinate is zero so it is centered left and right. It is also at y=1 or one unit up from the origin. The z coordinate is 2 which is five units in front of the camera, which is at z=-3. After the center vector is a comma followed by the radius which in this case is two units. Since the radius is half the width of a sphere, the sphere is four units wide.

2.1.5 Adding Texture to an Object

After we have defined the location and size of the sphere, we need to describe the appearance of the surface. The **texture** statement specifies these parameters. Texture blocks describe the color, bumpiness and finish properties of an object. In this example we will specify the color only. This is the minimum we must do. All other texture options except color will use default values.

The color we define is the way we want an object to look if fully illuminated. If we were painting a picture of a sphere we would use dark shades of a color to indicate the shadowed side and bright shades on the illuminated side. However ray-tracing takes care of that for you. We only need to pick the basic color inherent in the object and POV-Ray brightens or darkens it depending on the lighting in the scene. Because we are defining the basic color the object actually **has** rather than how it **looks** the parameter is called **pigment**.

Many types of color patterns are available for use in a pigment statement. The keyword **color** specifies that the whole object is to be one solid color rather than some pattern of colors. We can use one of the color identifiers previously defined in the standard include file `colors.inc`.

If no standard color is available for our needs, we may define our own color by using the color keyword followed by **red**, **green**, and **blue** keywords specifying the amount of red, green and blue to be mixed. For example a nice shade of pink can be specified by:

```
color red 1.0 green 0.8 blue 0.8
```

The values after each keyword should be in the range from 0.0 to 1.0. Any of the three components not specified will default to 0. A shortcut notation may also be used. The following produces the same shade of pink:

```
color rgb <1.0, 0.8, 0.8>
```

Colors are explained in more detail in section "Specifying Colors".

2.1.6 Defining a Light Source

One more detail is needed for our scene. We need a light source. Until we create one, there is no light in this virtual world. Thus we add the line

```
light_source { <2, 4, -3> color White}
```

to the scene file to get our first complete POV-Ray scene file as shown below.

```
#include "colors.inc"
background { color Cyan }
camera {
  location <0, 2, -3>
  look_at  <0, 1,  2>
}
sphere {
  <0, 1, 2>, 2
  texture {
    pigment { color Yellow }
  }
}
light_source { <2, 4, -3> color White}
```

The vector in the **light_source** statement specifies the location of the light as two units to our right, four units above the origin and three units back from the origin. The light source is an invisible tiny point that emits light. It has no physical shape, so no texture is needed.

That's it! We close the file and render a small picture of it using whatever methods you used for your particular platform. If you specified a preview display it will appear on your screen. If you specified an output file (the default is file output on), then POV-Ray also created a file. Note that if you do not have high color or true color

display hardware then the preview image may look poor but the full detail is written to the image file regardless of the type of display.

The scene we just traced isn't quite state of the art but we will have to start with the basics before we soon get to much more fascinating features and scenes.

2.2 Simple Shapes

So far we have just used the sphere shape. There are many other types of shapes that can be rendered by POV-Ray. The following sections will describe how to use some of the more simple objects as a replacement for the sphere used above.

2.2.1 Box Object

The **box** is one of the most common objects used. We try this example in place of the sphere:

```
box {
  <-1, 0,   -1>,  // Near lower left corner
  < 1, 0.5,  3>   // Far upper right corner
  texture {
    T_Stone25     // Pre-defined from stones.inc
    scale 4       // Scale by the same amount in all
                  // directions
  }
  rotate y*20     // Equivalent to "rotate <0,20,0>"
}
```

In the example we can see that a box is defined by specifying the 3D coordinates of its opposite corners. The first vector must be the minimum x-, y- and z-coordinates and the 2nd vector must be the maximum x-, y- and z-values. Box objects can only be defined parallel to the axes of the world coordinate system. We can later rotate them to any angle. Note that we can perform simple math on values and vectors. In the rotate parameter we multiplied the vector identifier **y** by 20. This is the same as <0,1,0>*20 or <0,20,0>.

2.2.2 Cone Object

Here's another example showing how to use a **cone**:

```
cone {
  <0, 1, 0>, 0.3    // Center and radius of one end
  <1, 2, 3>, 1.0    // Center and radius of other end
  texture { T_Stone25 scale 4 }
}
```

The cone shape is defined by the center and radius of each end. In this example one end is at location <0,1,0> and has a radius of 0.3 while the other end is centered at <1,2,3> with radius=1. If we want the cone to come to a sharp point we must use radius=0. The solid end caps are parallel to each other and perpendicular to the cone axis. If we want an open cone with no end caps we have to add the keyword **open** after the 2nd radius like this:

```
cone {
  <0, 1, 0>, 0.3    // Center and radius of one end
  <1, 2, 3>, 1.0    // Center and radius of other end
  open              // Removes end caps
  texture { T_Stone25 scale 4 }
}
```

2.2.3 Cylinder Object

We may also define a **cylinder** like this:

```
cylinder {
  <0, 1, 0>,     // Center of one end
  <1, 2, 3>,     // Center of other end
  0.5            // Radius
  open           // Remove end caps
  texture { T_Stone25 scale 4 }
}
```

2.2.4 Plane Object

Let's try out a computer graphics standard *"The Checkered Floor"*. We add the following object to the first version of the `demo.pov` file, the one including the sphere.

```
plane { <0, 1, 0>, -1
  pigment {
    checker color Red, color Blue
  }
}
```

The object defined here is an infinite plane. The vector <0,1,0> is the surface normal of the plane (i.e. if we were standing on the surface, the normal points straight up). The number afterward is the distance that the plane is displaced along the normal from the origin - in this case, the floor is placed at y=-1 so that the sphere at y=1, radius=2, is resting on it.

We note that even though there is no **`texture`** statement there is an implied texture here. We might find that continually typing statements that are nested like **`texture {pigment`**} can get to be tiresome so POV-Ray let's us leave out the **`texture`** statement under many circumstances. In general we only need the texture block surrounding a texture identifier (like the **`T_Stone25`** example above), or when creating layered textures (which are covered later).

This pigment uses the checker color pattern and specifies that the two colors red and blue should be used.

Because the vectors <1,0,0>, <0,1,0> and <0,0,1> are used frequently, POV-Ray has three built-in vector identifiers **`x`**, **`y`** and **`z`** respectively that can be used as a shorthand. Thus the plane could be defined as:

```
plane { y, -1
  pigment { ... }
}
```

Note that we do not use angle brackets around vector identifiers.

Looking at the floor, we notice that the ball casts a shadow on the floor. Shadows are calculated very accurately by the ray-tracer, which creates precise, sharp shadows. In the real world, penumbral or "soft" shadows are often seen. Later we will learn how to use extended light sources to soften the shadows.

2.3 CSG Objects

Constructive Solid Geometry, or CSG, is a powerful tool to combine primitive objects to create more complex objects as shown in the following sections.

2.3.1 What is CSG?

CSG stands for *Constructive Solid Geometry*. POV-Ray allows us to construct complex solids by combining primitive shapes in four different ways. In the **`union`** statement, two or more shapes are added together. With the **`intersection`** statement, two or more shapes are combined to make a new shape that consists of the area common to both shapes. The **`difference`** statement, an initial shape has all subsequent shapes subtracted from it.

And last not least **merge**, which is like a union where the surfaces inside the union are removed (useful in transparent CSG objects). We will deal with each of these in detail in the next few sections.

CSG objects can be extremely complex. They can be deeply nested. In other words there can be unions of differences or intersections of merges or differences of intersections or even unions of intersections of differences of merges... ad infinitum. CSG objects are (almost always) finite objects and thus respond to auto-bounding and can be transformed like any other POV primitive shape.

2.3.2 CSG Union

Let's try making a simple union. Create a file called `csgdemo.pov` and edit it as follows:

```
#include "colors.inc"
camera {
  location <0, 1, -10>
  look_at 0
  angle 36
}
light_source { <500, 500, -1000> White }
plane { y, -1.5
  pigment { checker Green White }
}
```

Let's add two spheres each translated 0.5 units along the x-axis in each direction. We color one blue and the other red.

```
sphere { <0, 0, 0>, 1
  pigment { Blue }
  translate -0.5*x
}
sphere { <0, 0, 0>, 1
  pigment { Red }
  translate 0.5*x
}
```

We trace this file and note the results. Now we place a union block around the two spheres. This will create a single CSG union out of the two objects.

```
union{
  sphere { <0, 0, 0>, 1
    pigment { Blue }
    translate -0.5*x
  }
  sphere { <0, 0, 0>, 1
    pigment { Red }
    translate 0.5*x
  }
}
```

We trace the file again. The union will appear no different from what each sphere looked like on its own, but now we can give the entire union a single texture and transform it as a whole. Let's do that now.

```
union{
  sphere { <0, 0, 0>, 1
    translate -0.5*x
  }
  sphere { <0, 0, 0>, 1
    translate 0.5*x
  }
  pigment { Red }
```

```
  scale <1, .25, 1>
  rotate <30, 0, 45>
}
```

We trace the file again. As we can see, the object has changed dramatically. We experiment with different values of scale and rotate and try some different textures.

There are many advantages of assigning only one texture to a CSG object instead of assigning the texture to each individual component. First, it is much easier to use one texture if our CSG object has a lot of components because changing the objects appearance involves changing only one single texture. Second, the file parses faster because the texture has to be parsed only once. This may be a great factor when doing large scenes or animations. Third, using only one texture saves memory because the texture is only stored once and referenced by all components of the CSG object. Assigning the texture to all n components means that it is stored n times.

2.3.3 CSG Intersection

Now let's use these same spheres to illustrate the next kind of CSG object, the **`intersection`**. We change the word **`union`** to **`intersection`** and delete the **`scale`** and **`rotate`** statements:

```
intersection {
  sphere { <0, 0, 0>, 1
    translate -0.5*x
  }
  sphere { <0, 0, 0>, 1
    translate 0.5*x
  }
  pigment { Red }
}
```

We trace the file and will see a lens-shaped object instead of the two spheres. This is because an intersection consists of the area shared by both shapes, in this case the lens-shaped area where the two spheres overlap. We like this lens-shaped object so we will use it to demonstrate differences.

2.3.4 CSG Difference

We rotate the lens-shaped intersection about the y-axis so that the broad side is facing the camera.

```
intersection{
  sphere { <0, 0, 0>, 1
    translate -0.5*x
  }
  sphere { <0, 0, 0>, 1
    translate 0.5*x
  }
  pigment { Red }
  rotate 90*y
}
```

Let's create a cylinder and stick it right in the middle of the lens.

```
cylinder { <0, 0, -1> <0, 0, 1>, .35
  pigment { Blue }
}
```

We render the scene to see the position of the cylinder. We will place a **`difference`** block around both the lens-shaped intersection and the cylinder like this:

```
difference {
  intersection {
    sphere { <0, 0, 0>, 1
```

```
      translate -0.5*x
    }
    sphere { <0, 0, 0>, 1
      translate 0.5*x
    }
    pigment { Red }
    rotate 90*y
  }
  cylinder { <0, 0, -1> <0, 0, 1>, .35
    pigment { Blue }
  }
}
```

We render the file again and see the lens-shaped intersection with a neat hole in the middle of it where the cylinder was. The cylinder has been **`subtracted`** from the intersection. Note that the pigment of the cylinder causes the surface of the hole to be colored blue. If we eliminate this pigment the surface of the hole will be red.

OK, let's get a little wilder now. Let's declare our perforated lens object to give it a name. Let's also eliminate all textures in the declared object because we will want them to be in the final union instead.

```
#declare Lens_With_Hole = difference {
  intersection {
    sphere { <0, 0, 0>, 1
      translate -0.5*x
    }
    sphere { <0, 0, 0>, 1
      translate 0.5*x
    }
    rotate 90*y
  }
  cylinder { <0, 0, -1> <0, 0, 1>, .35 }
}
```

Let's use a union to build a complex shape composed of copies of this object.

```
union {
  object { Lens_With_Hole translate <-.65, .65, 0> }
  object { Lens_With_Hole translate <.65, .65, 0> }
  object { Lens_With_Hole translate <-.65, -.65, 0> }
  object { Lens_With_Hole translate <.65, -.65, 0> }
  pigment { Red }
}
```

We render the scene. An interesting object to be sure. But let's try something more. Let's make it a partially-transparent object by adding some filter to the pigment block.

```
union {
  object { Lens_With_Hole translate <-.65, .65, 0> }
  object { Lens_With_Hole translate <.65, .65, 0> }
  object { Lens_With_Hole translate <-.65, -.65, 0> }
  object { Lens_With_Hole translate <.65, -.65, 0> }
  pigment { Red filter .5 }
}
```

We render the file again. This looks pretty good... only... we can see parts of each of the lens objects inside the union! This is not good.

2.3.5 CSG Merge

This brings us to the fourth kind of CSG object, the **merge**. Merges are the same as unions, but the geometry of the objects in the CSG that is inside the merge is not traced. This should eliminate the problem with our object. Let's try it.

```
merge {
  object { Lens_With_Hole translate <-.65, .65, 0> }
  object { Lens_With_Hole translate <.65, .65, 0> }
  object { Lens_With_Hole translate <-.65, -.65, 0> }
  object { Lens_With_Hole translate <.65, -.65, 0> }
  pigment { Red filter .5 }
}
```

Sure enough, it does!

2.3.6 CSG Pitfalls

There is a severe pitfall in the CSG code that we have to be aware of.

2.3.6.1 Coincidence Surfaces

POV-Ray uses inside/outside tests to determine the points at which a ray intersects a CSG object. A problem arises when the surfaces of two different shapes coincide because there is no way (due to numerical problems) to tell whether a point on the coincident surface belongs to one shape or the other.

Look at the following example where a cylinder is used to cut a hole in a larger box.

```
difference {
  box { -1, 1 pigment { Red } }
  cylinder { -z, z, 0.5 pigment { Green } }
}
```

Note that the vectors -1 and 1 in the box definition expand to <-1,-1,-1> and <1,1,1> respectively.

If we trace this object we see red speckles where the hole is supposed to be. This is caused by the coincident surfaces of the cylinder and the box. One time the cylinder's surface is hit first by a viewing ray, resulting in the correct rendering of the hole, and another time the box is hit first, leading to a wrong result where the hole vanishes and red speckles appear.

This problem can be avoided by increasing the size of the cylinder to get rid of the coincidence surfaces. This is done by:

```
difference {
  box { -1, 1 pigment { Red } }
  cylinder { -1.001*z, 1.001*z, 0.5 pigment { Green } }
}
```

In general we have to make the subtracted object a little bit larger in a CSG difference. We just have to look for coincident surfaces and increase the subtracted object appropriately to get rid of those surfaces.

The same problem occurs in CSG intersections and is also avoided by scaling some of the involved objects.

2.4 Advanced Shapes

After we have gained some experience with the simpler shapes available in POV-Ray it is time to go on to the more advanced, thrilling shapes.

We should be aware that the shapes described below are not trivial to understand. We needn't be worried though if we do not know how to use them or how they work. We just try the examples and play with the features described in the reference chapter. There is nothing better than learning by doing.

You may wish to skip to the chapter "Simple Texture Options" before proceeding with these advanced shapes.

2.4.1 Bicubic Patch Object

Bicubic or Bezier patches are useful surface representations because they allow an easy definition of surfaces using only a few control points. The control points serve to determine the shape of the patch. Instead of defining the vertices of triangles, we simply give the coordinates of the control points. A single patch has 16 control points, one at each corner, and the rest positioned to divide the patch into smaller sections. For ray-tracing (or rendering) the patches are approximated using triangles.

Bezier patches are almost always created using a third party modeler so for this tutorial, we will use `moray` (any other modeler that supports Bezier patches and POV-Ray can also be used). We will use `moray` only to create the patch itself, not the other elements of the scene.

Note that `moray` is not included with POV-Ray. It is a separate shareware program that currently only runs on MS-Dos and Win95/NT but this tutorial assumes you are using the MS-Dos version. If you do not have `moray` or are not on MS-Dos, you can still render the sample scene described below, even though you cannot see how `moray` created it. Simply type in the **`bicubic_patch`** declaration listed below.

Bezier patches are actually very useful and, with a little practice, some pretty amazing things can be created with them. For our first tutorial, let's make a sort of a teepee/tent shape using a single sheet patch.

First, we start `moray` and, from the main edit screen, we click on "CREATE". We Name our object **`Teepee`**. The "CREATE BEZIER PATCH" dialogue box will appear. We have to make sure that "SHEET" is depressed. We click on "OK, CREATE". At the bottom of the main edit screen, we click on "EXTENDED EDIT".

We hold the cursor over the "TOP" view and right click to make the pop-up menu appear. We click on "MAXIMIZE". We [ALT]-drag to zoom in a little. We click on "MARK ALL", and under the transformation mode box, "UFRM SCL". We drag the mouse to scale the patch until it is approximately four units wide. We click on "TRANSLATE", and move the patch so that its center is over the origin. We right click "MINIMIZE" and "UNMARK ALL".

We [SHIFT]-drag a box around the lower right control point to mark it. We [ALT]-zoom into the "FRONT" view so that we can see the patch better. In the "FRONT" view, we "TRANSLATE" that point 10 units along the negative z-axis (we note that in MORAY z is up). We "UNMARK ALL". We repeat this procedure for each of the other three corner points. We make sure we remember to "UNMARK ALL" once each point has been translated. We should have a shape that looks as though it is standing on four pointed legs. We "UNMARK ALL".

Working once again in the "TOP" view, we [SHIFT]-drag a box around the four center control points to mark them. We right-click over the "TOP" view and "MAXIMIZE". We click on "UFRM SCL" and drag the mouse to scale the four points close together. We [ALT]-drag to zoom closer and get them as close together as we can. We [ALT]-drag to zoom out, right click and "MINIMIZE".

In the "FRONT" view, we "TRANSLATE" the marked points 10 units along the positive z-axis. We "UNMARK ALL". The resulting shape is quite interesting, was simple to model, and could not be produced using CSG primitives. Now let's use it in a scene.

We click on "DONE" to return to the main edit screen. We note that **`u_steps`** and **`v_steps`** are both set to 3 and flatness is set to 0.01. We leave them alone for now. We click on "FILES" and then "SAVE SEL" (save selection). We name our new file `teepee1.mdl`. We press [F3] and open `teepee1.mdl`. There is no need to save the original file. When `teepee1` is open, we create a quick "dummy" texture (`moray` will not allow us to export data

without a texture). We use white with default finish and name it **TeePeeTex**. We apply it to the object, save the file and press [CTRL-F9]. moray will create two files: teepee1.inc and teepee1.pov.

We exit moray and copy teepee1.inc and teepee1.pov into our working directory where we are doing these tutorials. We create a new file called bezdemo.pov and edit it as follows:

```
#include "colors.inc"
camera {
  location <0, .1, -60>
  look_at 0
  angle 40
}
background { color Gray25 }  //to make the patch easier to see
light_source { <300, 300, -700> White }
plane { y, -12
  texture {
    pigment {
      checker
      color Green
      color Yellow
    }
  }
}
```

Using a text editor, we create and declare a simple texture for our teepee object:

```
#declare TeePeeTex = texture {
  pigment {
    color rgb <1, 1, 1,>
  }
  finish {
    ambient .2
    diffuse .6
  }
}
```

We paste in the bezier patch data from teepee1.pov (the additional object keywords added by moray were removed):

```
bicubic_patch {
  type 1 flatness 0.0100 u_steps 3 v_steps 3,
  <-5.174134, 5.528420, -13.211995>,
  <-1.769023, 5.528420, 0.000000>,
  <1.636088, 5.528420, 0.000000>,
  <5.041199, 5.528420, -13.003932>,
  <-5.174134, 1.862827, 0.000000>,
  <0.038471, 0.031270, 18.101474>,
  <0.036657, 0.031270, 18.101474>,
  <5.041199, 1.862827, 0.000000>,
  <-5.174134, -1.802766, 0.000000>,
  <0.038471, 0.028792, 18.101474>,
  <0.036657, 0.028792, 18.101474>,
  <5.041199, -1.802766, 0.000000>,
  <-5.174134, -5.468359, -13.070366>,
  <-1.769023, -5.468359, 0.000000>,
  <1.636088, -5.468359, 0.000000>,
  <4.974128, -5.468359, -12.801446>
  texture {
    TeePeeTex
```

```
    }
    rotate -90*x  // to orient the object to LHC
    rotate 25*y   // to see the four "legs" better
  }
```

We add the above rotations so that the patch is oriented to POV-Ray's left-handed coordinate system (remember the patch was made in moray in a right handed coordinate system), so we can see all four legs. Rendering this at 200x150 -a we see pretty much what we expect, a white teepee over a green and yellow checkered plane. Let's take a little closer look. We render it again, this time at 320x200.

Now we see that something is amiss. There appears to be sharp angling, almost like facing, especially near the top. This is indeed a kind of facing and is due to the **u_steps** and **v_steps** parameters. Let's change these from 3 to 4 and see what happens.

That's much better, but it took a little longer to render. This is an unavoidable tradeoff. If we want even finer detail, we must use a **u_steps** and **v_steps** value of 5 and set flatness to 0. But we must expect to use lots of memory and an even longer tracing time.

Well, we can't just leave this scene without adding a few items just for interest. We declare the patch object and scatter a few of them around the scene:

```
#declare TeePee = bicubic_patch {
  type 1 flatness 0.0100 u_steps 3 v_steps 3,
  <-5.174134, 5.528420, -13.211995>,
  <-1.769023, 5.528420, 0.000000>,
  <1.636088, 5.528420, 0.000000>,
  <5.041199, 5.528420, -13.003932>,
  <-5.174134, 1.862827, 0.000000>,
  <0.038471, 0.031270, 18.101474>,
  <0.036657, 0.031270, 18.101474>,
  <5.041199, 1.862827, 0.000000>,
  <-5.174134, -1.802766, 0.000000>,
  <0.038471, 0.028792, 18.101474>,
  <0.036657, 0.028792, 18.101474>,
  <5.041199, -1.802766, 0.000000>,
  <-5.174134, -5.468359, -13.070366>,
  <-1.769023, -5.468359, 0.000000>,
  <1.636088, -5.468359, 0.000000>,
  <4.974128, -5.468359, -12.801446>
  texture {
    TeePeeTex
   }
   rotate -90*x // to orient the object to LHC
   rotate 25*y  // to see the four "legs" better
}
object { TeePee }
object { TeePee translate <8, 0, 8> }
object { TeePee translate <-9, 0, 9> }
object { TeePee translate <18, 0, 24> }
object { TeePee translate <-18, 0, 24> }
```

That looks good. Let's do something about that boring gray background. We delete the background declaration and replace it with:

```
plane { y, 500
  texture {
    pigment { SkyBlue }
    finish { ambient 1 diffuse 0}
   }
```

```
    texture {
      pigment {
        bozo
        turbulence .5
        color_map {
          [0 White]
          [1 White filter 1]
        }
      }
      finish { ambient 1 diffuse 0 }
      scale <1000, 250, 250>
      rotate <5, 45, 0>
    }
  }
```

This adds a pleasing cirrus-cloud filled sky. Now, let's change the checkered plane to rippled sand dunes:

```
  plane {y,-12
    texture {
      pigment {
        color <.85, .5, .15>
      }
      finish {
        ambient .25
        diffuse .6
        crand .5
      }
      normal {
        ripples .35
        turbulence .25
        frequency 5
      }
      scale 10
      translate 50*x
    }
  }
```

We render this. Not bad! Let's just add one more element. Let's place a golden egg under each of the teepees. And since this is a bezier patch tutorial, let's make the eggs out of bezier patches.

We return to `moray` and create another bezier patch. We name it **`Egg1`** and select "CYLINDRICAL 2 - PATCH" from the "CREATE BEZIER PATCH" dialogue box. We click on "EXTENDED EDIT". We "MARK ALL" and rotate the patch so that the cylinder lays on its side. We "UNMARK ALL". In the "FRONT" view, we [SHIFT]-drag a box around the four points on the right end to mark them. In the "SIDE" view, we right click and "MAXIMIZE". We [ALT]-drag to zoom in a little closer. We "UFRM SCL" the points together as close as possible. We zoom in closer to get them nice and tight. We zoom out, right click and "MINIMIZE".

We click on "TRANSLATE" and drag the points to the left so that they are aligned on the z-axis with the next group of four points. This should create a blunt end to the patch. We repeat this procedure for the other end. We "UNMARK ALL".

In the "FRONT" view, the control grid should be a rectangle now and the patch should be an ellipsoid. We [SHIFT]-drag a box around the upper right corner of the control grid to mark those points. We then [SHIFT]-drag a box around the lower right corner to mark those points as well. In the "SIDE" view, we "UFRM SCL" the points apart a little to make that end of the egg a little wider than the other. We "UNMARK ALL".

The egg may need a little proportional adjustment. We should be able to "MARK ALL" and "LOCAL SCL" in the three views until we get it to look like an egg. When we are satisfied that it does, we "UNMARK ALL" and click on done. Learning from our teepee object, we now go ahead and change **`u_steps`** and **`v_steps`** to 4.

We create a dummy texture, white with default finish, name it **`EggTex`** and apply it to the egg. From the FILES menu, we "SAVE SEL" to filename `egg1.mdl`. We load this file and export ([CTRL F9]). We exit `moray` and copy the files `egg1.inc` and `egg1.pov` into our working directory.

Back in `bezdemo.pov`, we create a nice, shiny gold texture:

```
#declare EggTex = texture {
  pigment { BrightGold }
  finish {
    ambient .1
    diffuse .4
    specular 1
    roughness 0.001
    reflection .5
    metallic
  }
}
```

And while we're at it, let's dandy up our **`TeePeeTex`** texture:

```
#declare TeePeeTex = texture {
  pigment { Silver }
  finish {
    ambient .1
    diffuse .4
    specular 1
    roughness 0.001
    reflection .5
    metallic
  }
}
```

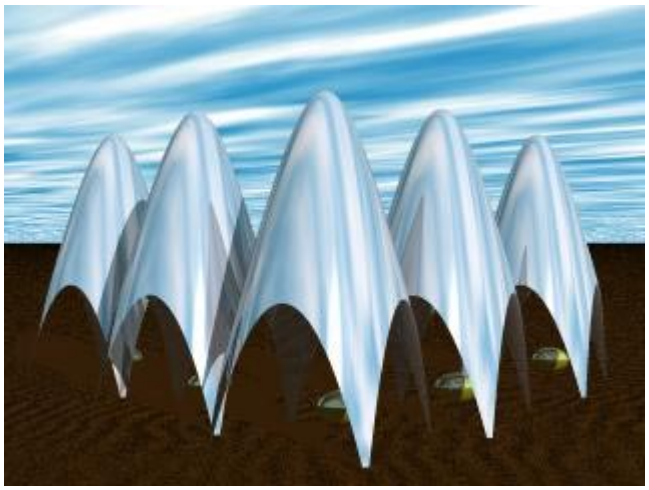
Now we paste in our egg patch data and declare our egg:

```
#declare Egg = union { // Egg1
  bicubic_patch {
    type 1 flatness 0.0100 u_steps 4 v_steps 4,
    <2.023314, 0.000000, 4.355987>,
    <2.023314, -0.000726, 4.355987>,
    <2.023312, -0.000726, 4.356867>,
    <2.023312, 0.000000, 4.356867>,
    <2.032037, 0.000000, 2.734598>,
    <2.032037, -1.758562, 2.734598>,
    <2.027431, -1.758562, 6.141971>,
    <2.027431, 0.000000, 6.141971>,
    <-1.045672, 0.000000, 3.281572>,
    <-1.045672, -1.758562, 3.281572>,
    <-1.050279, -1.758562, 5.414183>,
    <-1.050279, 0.000000, 5.414183>,
    <-1.044333, 0.000000, 4.341816>,
    <-1.044333, -0.002947, 4.341816>,
    <-1.044341, -0.002947, 4.345389>,
    <-1.044341, 0.000000, 4.345389>
  }
  bicubic_patch {
```

```
      type 1 flatness 0.0100 u_steps 4 v_steps 4,
      <2.023312, 0.000000, 4.356867>,
      <2.023312, 0.000726, 4.356867>,
      <2.023314, 0.000726, 4.355987>,
      <2.023314, 0.000000, 4.355987>,
      <2.027431, 0.000000, 6.141971>,
      <2.027431, 1.758562, 6.141971>,
      <2.032037, 1.758562, 2.734598>,
      <2.032037, 0.000000, 2.734598>,
      <-1.050279, 0.000000, 5.414183>,
      <-1.050279, 1.758562, 5.414183>,
      <-1.045672, 1.758562, 3.281572>,
      <-1.045672, 0.000000, 3.281572>,
      <-1.044341, 0.000000, 4.345389>,
      <-1.044341, 0.002947, 4.345389>,
      <-1.044333, 0.002947, 4.341816>,
      <-1.044333, 0.000000, 4.341816>
    }
    texture { EggTex }
    translate <0.5, 0, -5>  // centers the egg around the origin
    translate -9.8*y        // places the egg on the ground
  }
```

We now place a copy of the egg under each teepee. This should require only the x- and z-coordinates of each teepee to be changed:

```
  object { Egg }
  object { Egg translate <8, 0, 8> }
  object { Egg translate <-9, 0, 9> }
  object { Egg translate <18, 0, 24> }
  object { Egg translate <-18, 0, 24> }
```



Scene build with different Bezier patches.

We render this at low resolution such as 320x240. Everything looks good so we run it again at 640x480. Now we see that there is still some facing near the top of the teepees and on the eggs as well. The only solution is to raise **`u_steps`** and **`v_steps`** from 4 to 5 and set flatness to 0 for all our bezier objects. We make the changes and render it again at 640x480. The facets are gone.

2.4.2 Blob Object

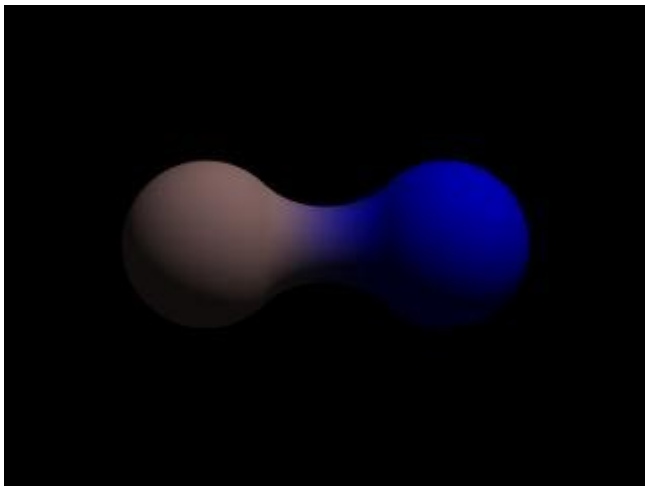
Blobs are described as spheres and cylinders covered with "goo" which stretches to smoothly join them (see section " an interior for these objects.

Blob"). Ideal for modeling atoms and molecules, blobs are also powerful tools for creating many smooth flowing "organic" shapes.

A slightly more mathematical way of describing a blob would be to say that it is one object made up of two or more component pieces. Each piece is really an invisible field of force which starts out at a particular strength and falls off smoothly to zero at a given radius. Where ever these components overlap in space, their field strength gets added together (and yes, we can have negative strength which gets subtracted out of the total as well). We could have just one component in a blob, but except for seeing what it looks like there is little point, since the real beauty of blobs is the way the components interact with one another.

Let us take a simple example blob to start. Now, in fact there are a couple different types of components but we will look at them a little later. For the sake of a simple first example, let us just talk about spherical components. Here is a sample POV-Ray code showing a basic camera, light, and a simple two component blob (this scene is called `blobdem1.pov`):

```
#include "colors.inc"
camera {
  angle 15
  location <0,2,-10>
  look_at <0,0,0>
}
light_source { <10, 20, -10> color White }
blob {
  threshold .65
  sphere { <.5,0,0>, .8, 1 pigment {Blue} }
  sphere { <-.5,0,0>,.8, 1 pigment {Pink} }
  finish { phong 1 }
}
```



A simple, two-part blob.

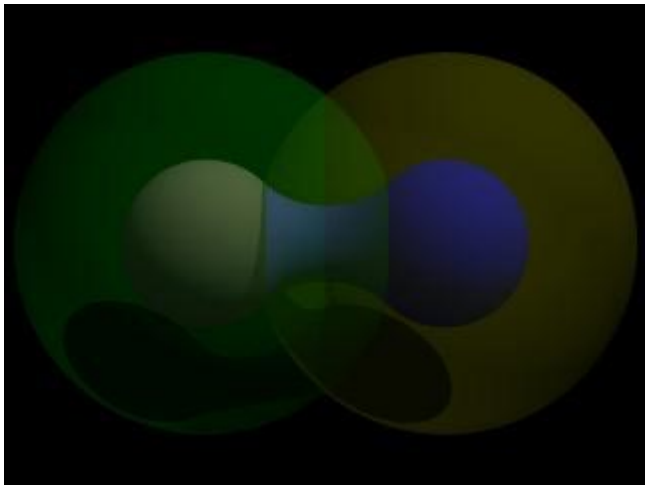
The threshold is simply the overall strength value at which the blob becomes visible. Any points within the blob where the strength matches the threshold exactly form the surface of the blob shape. Those less than the threshold are *outside* and those greater than are *inside* the blob.

We note that the spherical component looks a lot like a simple sphere object. We have the sphere keyword, the vector representing the location of the center of the sphere and the float representing the radius of the sphere. But what is that last float value? That is the individual strength of that component. In a spherical component, that is how strong the component's field is at the center of the sphere. It will fall off in a linear progression until it reaches exactly zero at the radius of the sphere.

Before we render this test image, we note that we have given each component a different pigment. POV-Ray allows blob components to be given separate textures. We have done this here to make it clearer which parts of the blob are which. We can also texture the whole blob as one, like the finish statement at the end, which applies to all components since it appears at the end, outside of all the components. We render the scene and get a basic kissing spheres type blob.

The image we see shows the spheres on either side, but they are smoothly joined by that bridge section in the center. This bridge represents where the two fields overlap, and therefore stay above the threshold for longer than elsewhere in the blob. If that is not totally clear, we add the following two objects to our scene and re-render (see file `blobdem2.pov`). We note that these are meant to be entered as separate sphere objects, not more components in the blob.

```
sphere { <.5,0,0>, .8
  pigment { Yellow transmit .75 }
}
sphere { <-.5,0,0>, .8
  pigment { Green transmit .75 }
}
```



The spherical components made visible.

Now the secrets of the kissing spheres are laid bare. These semi-transparent spheres show where the components of the blob actually are. If we have not worked with blobs before, we might be surprised to see that the spheres we just added extend way farther out than the spheres that actually show up on the blobs. That of course is because our spheres have been assigned a starting strength of one, which gradually fades to zero as we move away from the sphere's center. When the strength drops below the threshold (in this case 0.65) the rest of the sphere becomes part of the outside of the blob and therefore is not visible.

See the part where the two transparent spheres overlap? We note that it exactly corresponds to the bridge between the two spheres. That is the region where the two components are both contributing to the overall strength of the blob at that point. That is why the bridge appears: that region has a high enough strength to stay over the threshold, due to the fact that the combined strength of two spherical components is overlapping there.

2.4.2.1 Component Types and Other New Features

The shape shown so far is interesting, but limited. POV-Ray has a few extra tricks that extend its range of usefulness however. For example, as we have seen, we can assign individual textures to blob components, we can also apply individual transformations (translate, rotate and scale) to stretch, twist, and squash pieces of the blob as we require. And perhaps most interestingly, the blob code has been extended to allow cylindrical components.

Before we move on to cylinders, it should perhaps be mentioned that the old style of components used in previous versions of POV-Ray still work. Back then, all components were spheres, so it was not necessary to say sphere or cylinder. An old style component had the form:

component *Strength*, *Radius*, *<Center>*

This has the same effect as a spherical component, just as we already saw above. This is only useful for backwards compatibility. If we already have POV-Ray files with blobs from earlier versions, this is when we would need to recognize these components. We note that the old style components did not put braces around the strength, radius and center, and of course, we cannot independently transform or texture them, so if we are modifying an older work into a new version, it may arguably be of benefit to convert old style components into spherical components anyway.

Now for something new and different: cylindrical components. It could be argued that all we ever needed to do to make a roughly cylindrical portion of a blob was string a line of spherical components together along a straight line. Which is fine, if we like having extra to type, and also assuming that the cylinder was oriented along an axis. If not, we would have to work out the mathematical position of each component to keep it is a straight line. But no more! Cylindrical components have arrived.

We replace the blob in our last example with the following and re-render. We can get rid of the transparent spheres too, by the way.

```
blob {
  threshold .65
  cylinder { <-.75,-.75,0>, <.75,.75,0>, .5, 1 }
  pigment { Blue }
  finish { phong 1 }
}
```

We only have one component so that we can see the basic shape of the cylindrical component. It is not quite a true cylinder - more of a sausage shape, being a cylinder capped by two hem-spheres. We think of it as if it were an array of spherical components all closely strung along a straight line.

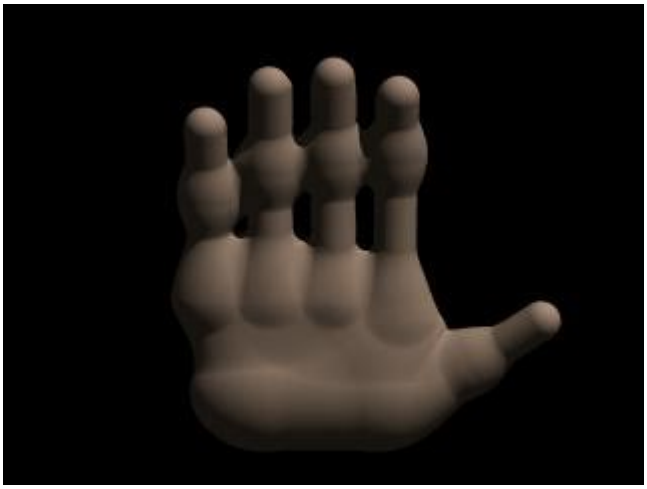
As for the component declaration itself: simple, logical, exactly as we would expect it to look (assuming we have been awake so far): it looks pretty much like the declaration of a cylinder object, with vectors specifying the two endpoints and a float giving the radius of the cylinder. The last float, of course, is the strength of the component. Just as with spherical components, the strength will determine the nature and degree of this component's interaction with its fellow components. In fact, next let us give this fellow something to interact with, shall we?

2.4.2.2 Complex Blob Constructs and Negative Strength

Beginning a new POV-Ray file called `blobdem3.pov`, we enter this somewhat more complex example:

```
#include "colors.inc"
camera {
  angle 20
  location<0,2,-10>
  look_at<0,0,0>
}
light_source { <10, 20, -10> color White }
blob {
  threshold .65
```

```
  sphere { <-.23,-.32,0>,.43, 1 scale <1.95,1.05,.8> }   //palm
  sphere { <+.12,-.41,0>,.43, 1 scale <1.95,1.075,.8> }  //palm
  sphere { <-.23,-.63,0>, .45, .75 scale <1.78, 1.3,1> } //midhand
  sphere { <+.19,-.63,0>, .45, .75 scale <1.78, 1.3,1> } //midhand
  sphere { <-.22,-.73,0>, .45, .85 scale <1.4, 1.25,1> } //heel
  sphere { <+.19,-.73,0>, .45, .85 scale <1.4, 1.25,1> } //heel
  cylinder { <-.65,-.28,0>, <-.65,.28,-.05>, .26, 1 }    //lower pinky
  cylinder { <-.65,.28,-.05>, <-.65, .68,-.2>, .26, 1 }  //upper pinky
  cylinder { <-.3,-.28,0>, <-.3,.44,-.05>, .26, 1 }      //lower ring
  cylinder { <-.3,.44,-.05>, <-.3, .9,-.2>, .26, 1 }     //upper ring
  cylinder { <.05,-.28,0>, <.05, .49,-.05>, .26, 1 }     //lower middle
  cylinder { <.05,.49,-.05>, <.05, .95,-.2>, .26, 1 }    //upper middle
  cylinder { <.4,-.4,0>, <.4, .512, -.05>, .26, 1 }      //lower index
  cylinder { <.4,.512,-.05>, <.4, .85, -.2>, .26, 1 }    //upper index
  cylinder { <.41, -.95,0>, <.85, -.68, -.05>, .25, 1 }  //lower thumb
  cylinder { <.85,-.68,-.05>, <1.2, -.4, -.2>, .25, 1 }  //upper thumb
  pigment { Flesh }
}
```



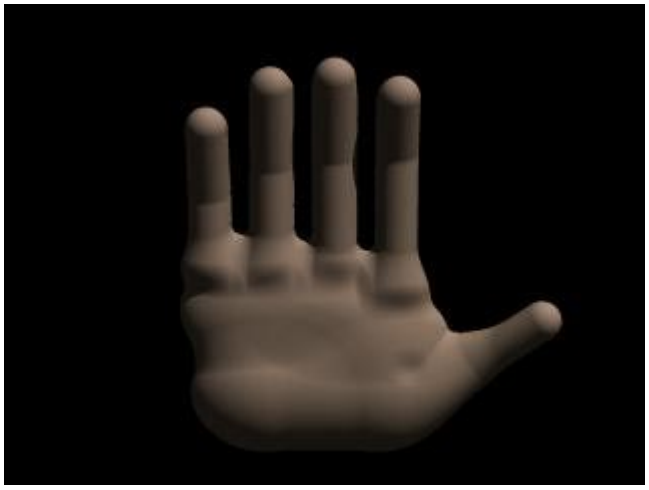
A hand made with blobs.

As we can guess from the comments, we are building a hand here. After we render this image, we can see there are a few problems with it. The palm and heel of the hand would look more realistic if we used a couple dozen smaller components rather than the half dozen larger ones we have used, and each finger should have three segments instead of two, but for the sake of a simplified demonstration, we can overlook these points. But there is one thing we really need to address here: This poor fellow appears to have horrible painful swelling of the joints!

A review of what we know of blobs will quickly reveal what went wrong. The joints are places where the blob components overlap, therefore the combined strength of both components at that point causes the surface to extend further out, since it stays over the threshold longer. To fix this, what we need are components corresponding to the overlap region which have a negative strength to counteract part of the combined field strength. We add the following components to our blob (see file `blobdem4.pov`).

```
sphere { <-.65,.28,-.05>, .26, -1 } //counteract pinky knuckle bulge
sphere { <-.65,-.28,0>, .26, -1 }   //counteract pinky palm bulge
sphere { <-.3,.44,-.05>, .26, -1 }  //counteract ring knuckle bulge
sphere { <-.3,-.28,0>, .26, -1 }    //counteract ring palm bulge
sphere { <.05,.49,-.05>, .26, -1 }  //counteract middle knuckle bulge
sphere { <.05,-.28,0>, .26, -1 }    //counteract middle palm bulge
sphere { <.4,.512,-.05>, .26, -1 }  //counteract index knuckle bulge
sphere { <.4,-.4,0>, .26, -1 }      //counteract index palm bulge
```

```
sphere { <.85,-.68,-.05>, .25, -1 } //counteract thumb knuckle bulge
sphere { <.41,-.7,0>, .25, -.89 }   //counteract thumb heel bulge
```



The hand without the swollen joints.

Much better! The negative strength of the spherical components counteracts approximately half of the field strength at the points where to components overlap, so the ugly, unrealistic (and painful looking) bulging is cut out making our hand considerably improved. While we could probably make a yet more realistic hand with a couple dozen additional components, what we get this time is a considerable improvement. Any by now, we have enough basic knowledge of blob mechanics to make a wide array of smooth, flowing organic shapes!

2.4.3 Height Field Object

A **`height_field`** is an object that has a surface that is determined by the color value or palette index number of an image designed for that purpose. With height fields, realistic mountains and other types of terrain can easily be made. First, we need an image from which to create the height field. It just so happens that POV-Ray is ideal for creating such an image.

We make a new file called `image.pov` and edit it to contain the following:

```
#include "colors.inc"
global_settings {
  assumed_gamma 2.2
  hf_gray_16
}
```

The **`hf_gray_16`** keyword causes the output to be in a special 16 bit grayscale that is perfect for generating height fields. The normal 8 bit output will lead to less smooth surfaces.

Now we create a camera positioned so that it points directly down the z-axis at the origin.

```
camera {
  location <0, 0, -10>
  look_at 0
}
```

We then create a plane positioned like a wall at z=0. This plane will completely fill the screen. It will be colored with white and gray wrinkles.

```
plane { z, 10
  pigment {
    wrinkles
```

```
      color_map {
       [0 0.3*White]
       [1 White]
      }
    }
  }
```

Finally, create a light source.

```
  light_source { <0, 20, -100> color White }
```

We render this scene at 640x480 **+A0.1 +FT**. We will get an image that will produce an excellent height field. We create a new file called `hfdemo.pov` and edit it as follows:

```
  #include "colors.inc"
```

We add a camera that is two units above the origin and ten units back ...

```
  camera{
    location <0, 2, -10>
    look_at 0
    angle 30
  }
```

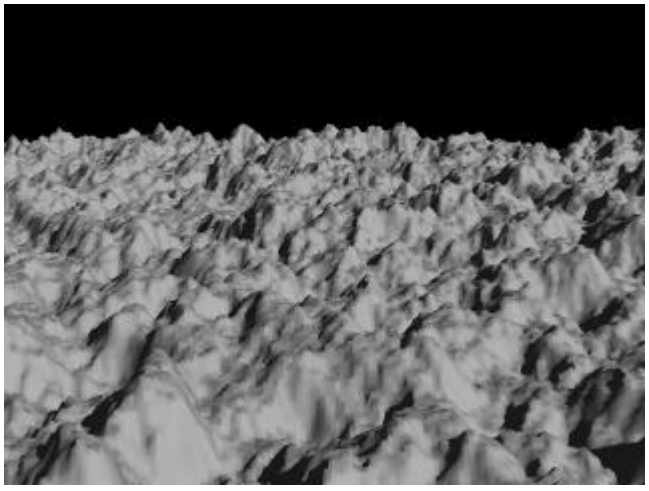
... and a light source.

```
  light_source{ <1000,1000,-1000> White }
```

Now we add the height field. In the following syntax, a Targa image file is specified, the height field is smoothed, it is given a simple white pigment, it is translated to center it around the origin and it is scaled so that it resembles mountains and fills the screen.

```
  height_field {
    tga "image.tga"
    smooth
    pigment { White }
    translate <-.5, -.5, -.5>
    scale <17, 1.75, 17>
  }
```

We save the file and render it at 320x240 **-A**. Later, when we are satisfied that the height field is the way we want it, we render it at a higher resolution with anti-aliasing.



A height field created completely with POV-Ray.

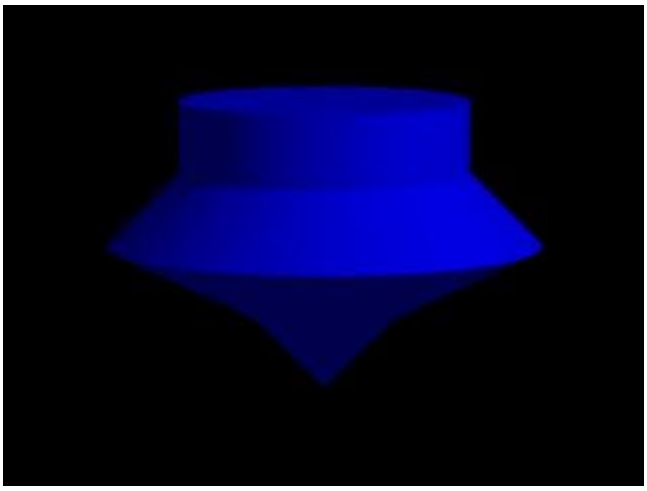
Wow! The Himalayas have come to our computer screen!

2.4.4 Lathe Object

In the real world, **lathe** refers to a process of making patterned rounded shapes by spinning the source material in place and carving pieces out as it turns. The results can be elaborate, smoothly rounded, elegant looking artifacts such as table legs, pottery, etc. In POV-Ray, a lathe object is used for creating much the same kind of items, although we are referring to the object itself rather than the means of production.

Here is some source for a really basic lathe (called `lathdem1.pov`).

```
#include "colors.inc"
camera {
  angle 10
  location <1, 9, -50>
  look_at <0, 2, 0>
}
light_source {
  <20, 20, -20> color White
}
lathe {
  linear_spline
  6,
  <0,0>, <1,1>, <3,2>, <2,3>, <2,4>, <0,4>
  pigment { Blue }
  finish {
    ambient .3
    phong .75
  }
}
```



A simple lathe object.

We render this, and what we see is a fairly simply type of lathe, which looks like a child's top. Let's take a look at how this code produced the effect.

First, a set of six points are declared which the raytracer connects with lines. We note that there are only two components in the vectors which describe these points. The lines that are drawn are assumed to be in the x-y-plane, therefore it is as if all the z-components were assumed to be zero. The use of a two-dimensional vector is mandatory (Attempting to use a 3D vector would trigger an error... with one exception, which we will explore later in the discussion of splines).

Once the lines are determined, the ray-tracer rotates this line around the y-axis, and we can imagine a trail being left through space as it goes, with the surface of that trail being the surface of our object.

The specified points are connected with straight lines because we used the **`linear_spline`** keyword. There are other types of splines available with the lathe, which will result in smooth curving lines, and even rounded curving points of transition, but we will get back to that in a moment.

First, we would like to digress a moment to talk about the difference between a lathe and a surface of revolution object (SOR). The SOR object, described in a separate tutorial, may seem terribly similar to the lathe at first glance. It too declares a series of points and connects them with curving lines and then rotates them around the y-axis. The lathe has certain advantages, such as different kinds of splines, linear, quadratic and cubic, and one more thing:

The simpler mathematics used by a SOR doesn't allow the curve to double back over the same y-coordinates, thus, if using a SOR, any sudden twist which cuts back down over the same heights that the curve previously covered will trigger an error. For example, suppose we wanted a lathe to arc up from <0,0> to <2,2>, then to dip back down to <4,0>. Rotated around the y-axis, this would produce something like a gelatin mold - a rounded semi torus, hollow in the middle. But with the SOR, as soon as the curve doubled back on itself in the y-direction, it would become an illegal declaration.

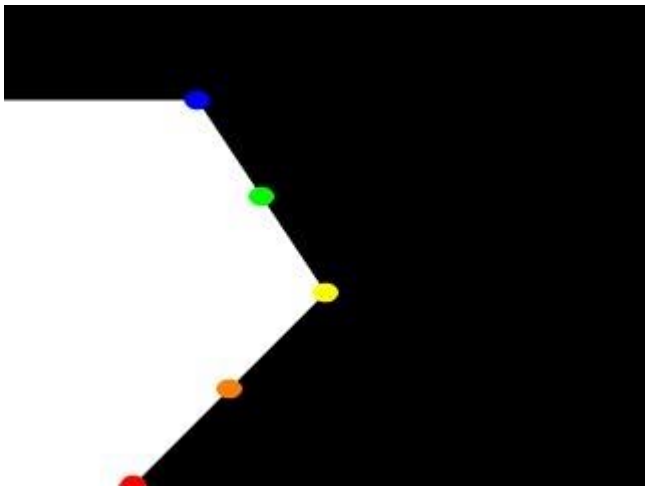
Still, the SOR has one powerful strong point: because it uses simpler order mathematics, it generally tends to render faster than an equivalent lathe. So in the end, its a matter of: we use a SOR if its limitations will allow, but when we need a more flexible shape, we go with the lathe instead.

2.4.4.1 Understanding The Concept of Splines

It would be helpful, in order to understand splines, if we had a sort of *Spline Workshop* where we could practice manipulating types and points of splines and see what the effects were like. So let's make one! Now that we know how to create a basic lathe, it will be easy (see file `lathdem2.pov`):

```
#include "colors.inc"
camera {
  orthographic
  up <0, 5, 0>
  right <5, 0, 0>
  location <2.5, 2.5, -100>
  look_at <2.5, 2.5, 0>
}
/* set the control points to be used */
#declare Red_Point    = <1.00, 0.00, 0>;
#declare Orange_Point = <1.75, 1.00, 0>;
#declare Yellow_Point = <2.50, 2.00, 0>;
#declare Green_Point  = <2.00, 3.00, 0>;
#declare Blue_Point   = <1.50, 4.00, 0>;
/* make the control points visible */
cylinder { Red_Point, Red_Point - 20*z, .1
  pigment { Red }
  finish { ambient 1 }
}
cylinder { Orange_Point, Orange_Point - 20*z, .1
  pigment { Orange }
  finish { ambient 1 }
}
cylinder { Yellow_Point, Yellow_Point - 20*z, .1
  pigment { Yellow }
  finish { ambient 1 }
}
cylinder { Green_Point, Green_Point - 20*z, .1
```

```
  pigment { Green }
  finish { ambient 1 }
}
cylinder { Blue_Point, Blue_Point- 20*z, .1
  pigment { Blue }
  finish { ambient 1 }
}
/* something to make the curve show up */
lathe {
  linear_spline
  5,
  Red_Point,
  Orange_Point,
  Yellow_Point,
  Green_Point,
  Blue_Point
  pigment { White }
  finish { ambient 1 }
}
```



A simple "Spline Workshop".

Now, we take a deep breath. We know that all looks a bit weird, but with some simple explanations, we can easily see what all this does.

First, we are using the orthographic camera. If we haven't read up on that yet, a quick summary is: it renders the scene *flat*, eliminating perspective distortion so that in a side view, the objects look like they were drawn on a piece of graph paper (like in the side view of a modeler or CAD package). There are several uses for this practical new type of camera, but here it is allowing us to see our lathe and cylinders *edge on*, so that what we see is almost like a cross section of the curve which makes the lathe, rather than the lathe itself. To further that effect, we eliminated shadowing with the **`ambient 1`** finish, which of course also eliminates the need for lighting. We have also positioned this particular side view so that <0,0> appears at the lower left of our scene.

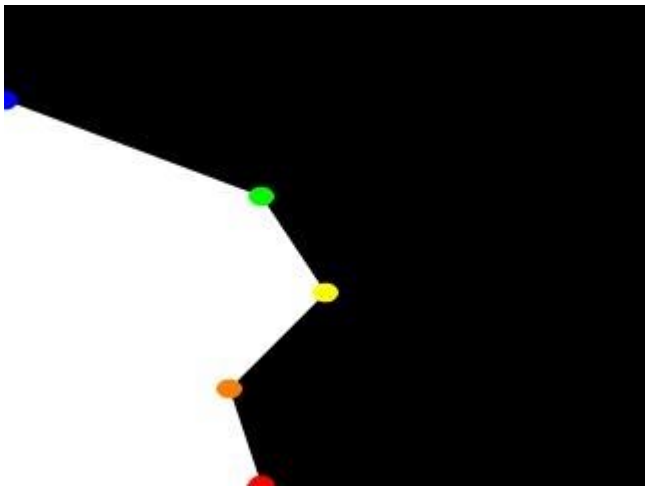
Next, we declared a set of points. We note that we used 3D vectors for these points rather than the 2D vectors we expect in a lathe. That's the exception we mentioned earlier. When we declare a 3D point, then use it in a lathe, the lathe only uses the first two components of the vector, and whatever is in the third component is simply ignored. This is handy here, since it makes this example possible.

Next we do two things with the declared points. First we use them to place small diameter cylinders at the locations of the points with the circular caps facing the camera. Then we re-use those same vectors to determine the lathe.

Since trying to declare a 2D vector can have some odd results, and isn't really what our cylinder declarations need anyway, we can take advantage of the lathe's tendency to ignore the third component by just setting the z-coordinate in these 3D vectors to zero.

The end result is: when we render this code, we see a white lathe against a black background showing us how the curve we've declared looks, and the circular ends of the cylinders show us where along the x-y-plane our control points are. In this case, it's very simple. The linear spline has been used so our curve is just straight lines zig-zagging between the points. We change the declarations of **`Red_Point`** and **`Blue_Point`** to read as follows (see file `lathdem3.pov`).

```
#declare Red_Point  = <2.00, 0.00, 0>;
#declare Blue_Point = <0.00, 4.00, 0>;
```

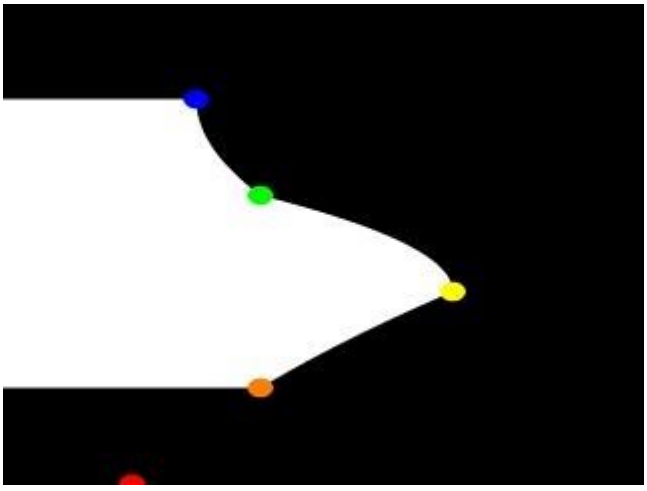


Moving some points of the spline.

We re-render and, as we can see, all that happens is that the straight line segments just move to accommodate the new position of the red and blue points. Linear splines are so simple, we could manipulate them in our sleep, no?

Let's try something different. First, we change the points to the following (see file `lathdem4.pov`).

```
#declare Red_Point    = <1.00, 0.00, 0>;
#declare Orange_Point = <2.00, 1.00, 0>;
#declare Yellow_Point = <3.50, 2.00, 0>;
#declare Green_Point  = <2.00, 3.00, 0>;
#declare Blue_Point   = <1.50, 4.00, 0>;
```



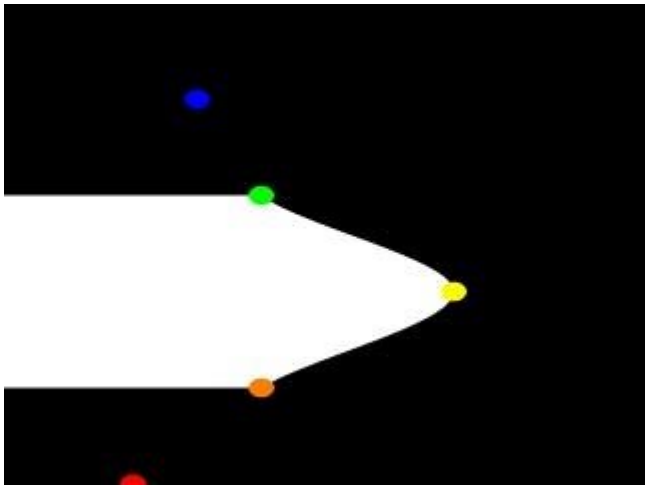
A quadratic spline lathe.

We then go down to the lathe declaration and change **linear_spline** to **quadratic_spline**. We re-render and what do we have? Well, there's a couple of things worthy of note this time. First, we will see that instead of straight lines we have smooth arcs connecting the points. These arcs are made from quadratic curves, so our lathe looks much more interesting this time. Also, **Red_Point** is no longer connected to the curve. What happened?

Well, while any two points can determine a straight line, it takes three to determine a quadratic curve. POV-Ray looks not only to the two points to be connected, but to the point immediately preceding them to determine the formula of the quadratic curve that will be used to connect them. The problem comes in at the beginning of the curve. Beyond the first point in the curve there is no *previous* point. So we need to declare one. Therefore, when using a quadratic spline, we must remember that the first point we specify is only there so that POV-Ray can determine what curve to connect the first two points with. It will not show up as part of the actual curve.

There's just one more thing about this lathe example. Even though our curve is now put together with smooth curving lines, the transitions between those lines is... well, kind of choppy, no? This curve looks like the lines between each individual point have been terribly mismatched. Depending on what we are trying to make, this could be acceptable, or, we might long for a more smoothly curving shape. Fortunately, if the latter is true, we have another option.

The quadratic spline takes longer to render than a linear spline. The math is more complex. Still longer needs the cubic spline, yet, for a really smoothed out shape, this is the only way to go. We go back into our example, and simply replace **quadratic_spline** with **cubic_spline** (see file `lathdem5.pov`). We render one more time, and take a look at what we have.



A cubic spline lathe.

While a quadratic spline takes three points to determine the curve, a cubic needs four. So, as we might expect, **Blue_Point** has now dropped out of the curve, just as **Red_Point** did, as the first and last points of our curve are now only control points for shaping the curves between the remaining points. But look at the transition from **Orange_Point** to **Yellow_Point** and then back to **Green_Point**. Now, rather than looking mismatched, our curve segments look like one smoothly joined curve.

The concept of splines is a handy and necessary one, which will be seen again in the prism and polygon objects. But with a little tinkering we can quickly get a feel for working with them.

2.4.5 Mesh Object

Mesh objects are very useful because they allow us to create objects containing hundreds or thousands of triangles. Compared to a simple union of triangles the mesh object stores the triangles more efficiently. Copies of mesh objects need only a little additional memory because the triangles are stored only once.

Almost every object can be approximated using triangles but we may need a lot of triangles to create more complex shapes. Thus we will only create a very simple mesh example. This example will show a very useful feature of the triangles meshes though: a different texture can be assigned to each triangle in the mesh.

Now let's begin. We will create a simple box with differently colored sides. We create an empty file called `meshdemo.pov` and add the following lines.

```
camera {
  location <20, 20, -50>
  look_at <0, 5, 0>
}
light_source { <50, 50, -50> color rgb<1, 1, 1> }
#declare Red = texture {
  pigment { color rgb<0.8, 0.2, 0.2> }
  finish { ambient 0.2 diffuse 0.5 }
}
#declare Green = texture {
  pigment { color rgb<0.2, 0.8, 0.2> }
  finish { ambient 0.2 diffuse 0.5 }
}
#declare Blue = texture {
  pigment { color rgb<0.2, 0.2, 0.8> }
  finish { ambient 0.2 diffuse 0.5 }
}
```

We must declare all textures we want to use inside the mesh before the mesh is created. Textures cannot be specified inside the mesh due to the poor memory performance that would result.

Now we add the mesh object. Three sides of the box will use individual textures while the other will use the *global* mesh texture.

```
mesh {
  /* top side */
  triangle { <-10, 10, -10>, <10, 10, -10>, <10, 10, 10>
    texture { Red }
  }
  triangle { <-10, 10, -10>, <-10, 10, 10>, <10, 10, 10>
    texture { Red }
  }
  /* bottom side */
  triangle { <-10, -10, -10>, <10, -10, -10>, <10, -10, 10> }
  triangle { <-10, -10, -10>, <-10, -10, 10>, <10, -10, 10> }
  /* left side */
  triangle { <-10, -10, -10>, <-10, -10, 10>, <-10, 10, 10> }
  triangle { <-10, -10, -10>, <-10, 10, -10>, <-10, 10, 10> }
  /* right side */
  triangle { <10, -10, -10>, <10, -10, 10>, <10, 10, 10>
    texture { Green }
  }
  triangle { <10, -10, -10>, <10, 10, -10>, <10, 10, 10>
    texture { Green }
  }
  /* front side */
```

```
    triangle { <-10, -10, -10>, <10, -10, -10>, <-10, 10, -10>
      texture { Blue }
    }
    triangle { <-10, 10, -10>, <10, 10, -10>, <10, -10, -10>
      texture { Blue }
    }
    /* back side */
    triangle { <-10, -10, 10>, <10, -10, 10>, <-10, 10, 10> }
    triangle { <-10, 10, 10>, <10, 10, 10>, <10, -10, 10> }
    texture {
      pigment { color rgb<0.9, 0.9, 0.9> }
      finish { ambient 0.2 diffuse 0.7 }
    }
  }
```

Tracing the scene at 320x240 we will see that the top, right and front side of the box have different textures. Though this is not a very impressive example it shows what we can do with mesh objects. More complex examples, also using smooth triangles, can be found under the scene directory as `chesmsh.pov` and `robotmsh.pov`.

2.4.6 Polygon Object

The **`polygon`** object can be used to create any planar, n-sided shapes like squares, rectangles, pentagons, hexagons, octagons, etc.

A polygon is defined by a number of points that describe its shape. Since polygons have to be closed the first point has to be repeated at the end of the point sequence.

In the following example we will create the word "POV" using just one polygon statement.

We start with thinking about the points we need to describe the desired shape. We want the letters to lie in the x-y-plane with the letter O being at the center. The letters extend from y=0 to y=1. Thus we get the following points for each letter (the z coordinate is automatically set to zero).

Letter P (outer polygon):
<-0.8, 0.0>, <-0.8, 1.0>,
<-0.3, 1.0>, <-0.3, 0.5>,
<-0.7, 0.5>, <-0.7, 0.0>
Letter P (inner polygon):
<-0.7, 0.6>, <-0.7, 0.9>,
<-0.4, 0.9>, <-0.4, 0.6>
Letter O (outer polygon):
<-0.25, 0.0>, <-0.25, 1.0>,
< 0.25, 1.0>, < 0.25, 0.0>
Letter O (inner polygon):
<-0.15, 0.1>, <-0.15, 0.9>,
< 0.15, 0.9>, < 0.15, 0.1>
Letter V:
<0.45, 0.0>, <0.30, 1.0>,
<0.40, 1.0>, <0.55, 0.1>,
<0.70, 1.0>, <0.80, 1.0>,
<0.65, 0.0>

Both letters P and O have a hole while the letter V consists of only one polygon. We'll start with the letter V because it is easier to define than the other two letters.

We create a new file called `polygdem.pov` and add the following text.

```
  camera {
```

```
    orthographic
    location <0, 0, -10>
    right 1.3 * 4/3 * x
    up 1.3 * y
    look_at <0, 0.5, 0>
  }
  light_source { <25, 25, -100> color rgb 1 }
  polygon {
    8,
    <0.45, 0.0>, <0.30, 1.0>, // Letter "V"
    <0.40, 1.0>, <0.55, 0.1>,
    <0.70, 1.0>, <0.80, 1.0>,
    <0.65, 0.0>,
    <0.45, 0.0>
    pigment { color rgb <1, 0, 0> }
  }
```

As noted above the polygon has to be closed by appending the first point to the point sequence. A closed polygon is always defined by a sequence of points that ends when a point is the same as the first point.

After we have created the letter V we'll continue with the letter P. Since it has a hole we have to find a way of cutting this hole into the basic shape. This is quite easy. We just define the outer shape of the letter P, which is a closed polygon, and add the sequence of points that describes the hole, which is also a closed polygon. That's all we have to do. There'll be a hole where both polygons overlap.

In general we will get holes whenever an even number of sub-polygons inside a single polygon statement overlap. A sub-polygon is defined by a closed sequence of points.

The letter P consists of two sub-polygons, one for the outer shape and one for the hole. Since the hole polygon overlaps the outer shape polygon we'll get a hole.

After we have understood how multiple sub-polygons in a single polygon statement work, it is quite easy to add the missing O letter.

Finally, we get the complete word POV.

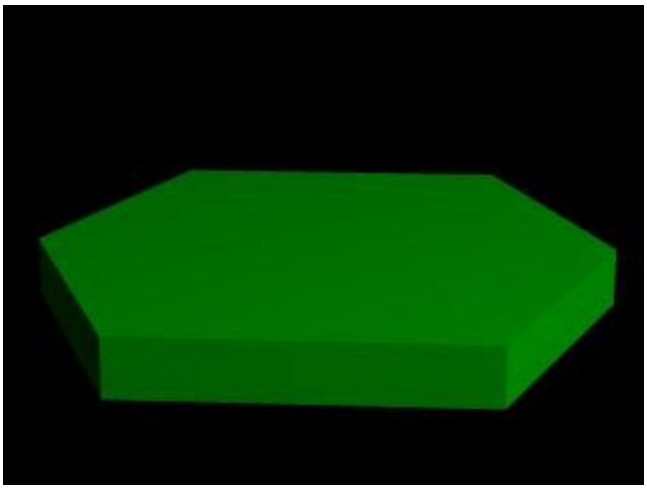
```
  polygon {
    30,
    <-0.8, 0.0>, <-0.8, 1.0>,    // Letter "P"
    <-0.3, 1.0>, <-0.3, 0.5>,    // outer shape
    <-0.7, 0.5>, <-0.7, 0.0>,
    <-0.8, 0.0>,
    <-0.7, 0.6>, <-0.7, 0.9>,    // hole
    <-0.4, 0.9>, <-0.4, 0.6>,
    <-0.7, 0.6>
    <-0.25, 0.0>, <-0.25, 1.0>,  // Letter "O"
    < 0.25, 1.0>, < 0.25, 0.0>,  // outer shape
    <-0.25, 0.0>,
    <-0.15, 0.1>, <-0.15, 0.9>,  // hole
    < 0.15, 0.9>, < 0.15, 0.1>,
    <-0.15, 0.1>,
    <0.45, 0.0>, <0.30, 1.0>,    // Letter "V"
    <0.40, 1.0>, <0.55, 0.1>,
    <0.70, 1.0>, <0.80, 1.0>,
    <0.65, 0.0>,
    <0.45, 0.0>
    pigment { color rgb <1, 0, 0> }
  }
```

The word "POV" made with one polygon statement.

2.4.7 Prism Object

The prism is essentially a polygon or closed curve which is swept along a linear path. We can imagine the shape so swept leaving a trail in space, and the surface of that trail is the surface of our prism. The curve or polygon making up a prism's face can be a composite of any number of sub-shapes, can use any kind of three different splines, and can either keep a constant width as it is swept, or slowly tapering off to a fine point on one end. But before this gets too confusing, let's start one step at a time with the simplest form of prism. We enter and render the following POV code (see file `prismdm1.pov`).

```
#include "colors.inc"
camera {
  angle 20
  location <2, 10, -30>
  look_at <0, 1, 0>
}
light_source { <20, 20, -20> color White }
prism {
  linear_sweep
  linear_spline
  0, // sweep the following shape from here ...
  1, // ... up through here
  7, // the number of points making up the shape ...
  <3,5>, <-3,5>, <-5,0>, <-3,-5>, <3, -5>, <5,0>, <3,5>
  pigment { Green }
}
```



A hexagonal prism shape.

This produces a hexagonal polygon, which is then swept from y=0 through y=1. In other words, we now have an extruded hexagon. One point to note is that although this is a six sided figure, we have used a total of seven points. That is because the polygon is supposed to be a closed shape, which we do here by making the final point the same as the first. Technically, with linear polygons, if we didn't do this, POV-Ray would automatically join the two ends with a line to force it to close, although a warning would be issued. However, this only works with linear splines, so we mustn't get too casual about those warning messages!

2.4.7.1 Teaching An Old Spline New Tricks

If we followed the section on splines covered under the lathe tutorial (see section "Understanding The Concept of Splines"), we know that there are two additional kinds of splines besides linear: the quadratic and the cubic spline. Sure enough, we can use these with prisms to make a more free form, smoothly curving type of prism.

There is just one catch, and we should read this section carefully to keep from tearing our hair out over mysterious "too few points in prism" messages which keep our prism from rendering. We can probably guess where this is heading: how to close a non-linear spline. Unlike the linear spline, which simply draws a line between the last and first points if we forget to make the last point equal to the first, quadratic and cubic splines are a little more fussy.

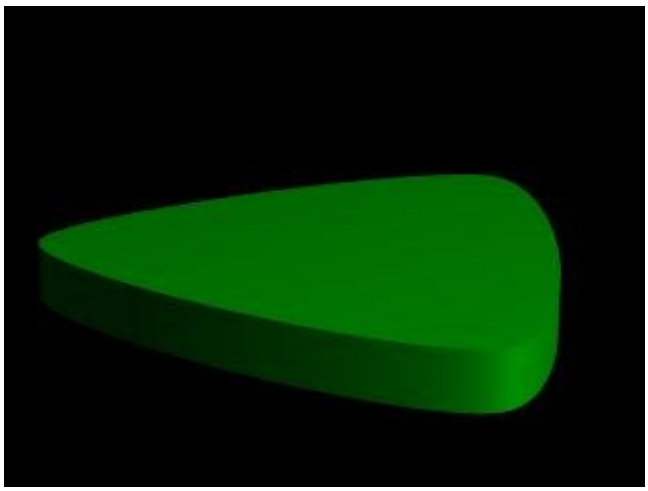
First of all, we remember that quadratic splines determine the equation of the curve which connects any two points based on those two points and the previous point, so the first point in any quadratic spline is just *control point* and won't actually be part of the curve. What this means is: when we make our shape out of a quadratic spline, we must match the second point to the last, since the first point is not on the curve - it's just a control point needed for computational purposes.

Likewise, cubic splines need both the first and last points to be control points, therefore, to close a shape made with a cubic spline, we must match the second point to the second from last point. If we don't match the correct points on a quadratic or cubic shape, that's when we will get the "too few points in prism" error. POV-Ray is still waiting for us to close the shape, and when it runs out of points without seeing the closure, an error is issued.

Confused? Okay, how about an example? We replace the prism in our last bit of code with this one (see file `prismdm2.pov`).

```
prism {
  cubic_spline
  0, // sweep the following shape from here ...
  1, // ... up through here
  6, // the number of points making up the shape ...
  < 3, -5>, // point#1 (control point... not on curve)
```

```
  < 3,  5>, // point#2  ... THIS POINT ...
  <-5,  0>, // point#3
  < 3, -5>, // point#4
  < 3,  5>, // point#5 ... MUST MATCH THIS POINT
  <-5,  0>  // point#6 (control point... not on curve)
  pigment { Green }
}
```



A cubic, triangular prism shape.

This simple prism produces what looks like an extruded triangle with its corners sanded smoothly off. Points two, three and four are the corners of the triangle and point five closes the shape by returning to the location of point two. As for points one and six, they are our control points, and aren't part of the shape - they're just there to help compute what curves to use between the other points.

2.4.7.2 Smooth Transitions

Now a handy thing to note is that we have made point one equal point four, and also point six equals point three. Yes, this is important. Although this prism would still be legally closed if the control points were not what we've made them, the curve transitions between points would not be as smooth. We change points one and six to <4,6> and <0,7> respectively and re-render to see how the back edge of the shape is altered (see file `prismdm3.pov`).

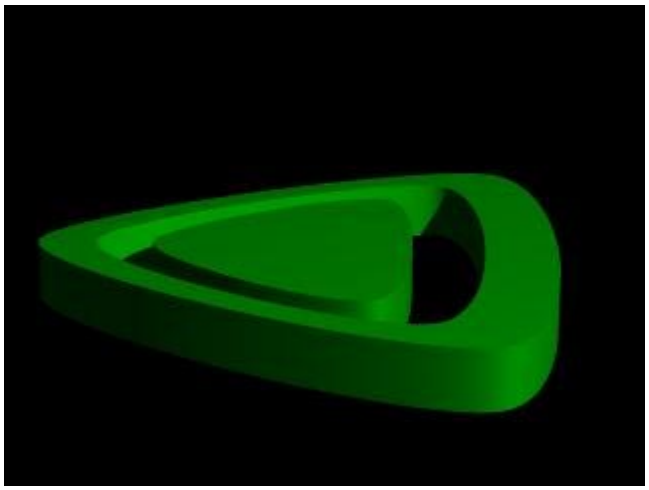
To put this more generally, if we want a smooth closure on a cubic spline, we make the first control point equal to the third from last point, and the last control point equal to the third point. On a quadratic spline, the trick is similar, but since only the first point is a control point, make that equal to the second from last point.

2.4.7.3 Multiple Sub-Shapes

Just as with the polygon object (see section "Polygon Object") the prism is very flexible, and allows us to make one prism out of several sub-prisms. To do this, all we need to do is keep listing points after we have already closed the first shape. The second shape can be simply an add on going off in another direction from the first, but one of the more interesting features is that if any even number of sub-shapes overlap, that region where they overlap behaves as though it has been cut away from both sub-shapes. Let's look at another example. Once again, same basic code as before for camera, light and so forth, but we substitute this complex prism (see file `prismdm4.pov`).

```
prism {
  linear_sweep
  cubic_spline
  0,  // sweep the following shape from here ...
  1,  // ... up through here
```

```
  18, // the number of points making up the shape ...
  <3,-5>, <3,5>, <-5,0>, <3, -5>, <3,5>, <-5,0>, // sub-shape #1
  <2,-4>, <2,4>, <-4,0>, <2,-4>, <2,4>, <-4,0>,  // sub-shape #2
  <1,-3>, <1,3>, <-3,0>, <1, -3>, <1,3>, <-3,0>  // sub-shape #3
  pigment { Green }
}
```



Using sub-shapes to create a more complex shape.

For readability purposes, we have started a new line every time we moved on to a new sub-shape, but the ray-tracer of course tells where each shape ends based on whether the shape has been closed (as described earlier). We render this new prism, and look what we've got. It's the same familiar shape, but it now looks like a smaller version of the shape has been carved out of the center, then the carved piece was sanded down even smaller and set back in the hole.

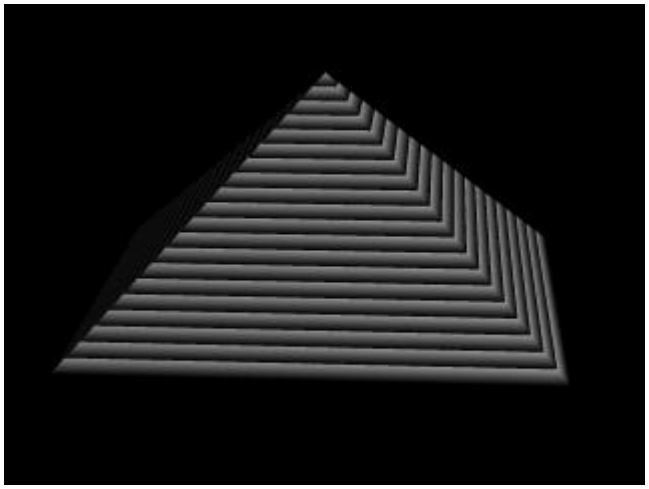
Simply, the outer rim is where only sub-shape one exists, then the carved out part is where sub-shapes one and two overlap. In the extreme center, the object reappears because sub-shapes one, two, and three overlap, returning us to an odd number of overlapping pieces. Using this technique we could make any number of extremely complex prism shapes!

2.4.7.4 Conic Sweeps And The Tapering Effect

In our original prism, the keyword **`linear_sweep`** is actually optional. This is the default sweep assumed for a prism if no type of sweep is specified. But there is another, extremely useful kind of sweep: the conic sweep. The basic idea is like the original prism, except that while we are sweeping the shape from the first height through the second height, we are constantly expanding it from a single point until, at the second height, the shape has expanded to the original points we made it from. To give a small idea of what such effects are good for, we replace our existing prism with this (see file `prismdm4.pov`):

```
prism {
  conic_sweep
  linear_spline
  0, // height 1
  1, // height 2
  5, // the number of points making up the shape...
  <4,4>,<-4,4>,<-4,-4>,<4,-4>,<4,4>
  rotate <180, 0, 0>
  translate <0, 1, 0>
  scale <1, 4, 1>
  pigment { gradient y scale .2 }
```

```
}
```



Creating a pyramid using conic sweeping.

The gradient pigment was selected to give some definition to our object without having to fix the lights and the camera angle right at this moment, but when we render it, we what we've created? A horizontally striped pyramid! By now we can recognize the linear spline connecting the four points of a square, and the familiar final point which is there to close the spline.

Notice all the transformations in the object declaration. That's going to take a little explanation. The rotate and translate are easy. Normally, a conic sweep starts full sized at the top, and tapers to a point at y=0, but of course that would be upside down if we're making a pyramid. So we flip the shape around the x-axis to put it right side up, then since we actually orbited around the point, we translate back up to put it in the same position it was in when we started.

The scale is to put the proportions right for this example. The base is eight units by eight units, but the height (from y=1 to y=0) is only one unit, so we've stretched it out a little. At this point, we're probably thinking, "why not just sweep up from y=0 to y=4 and avoid this whole scaling thing?"

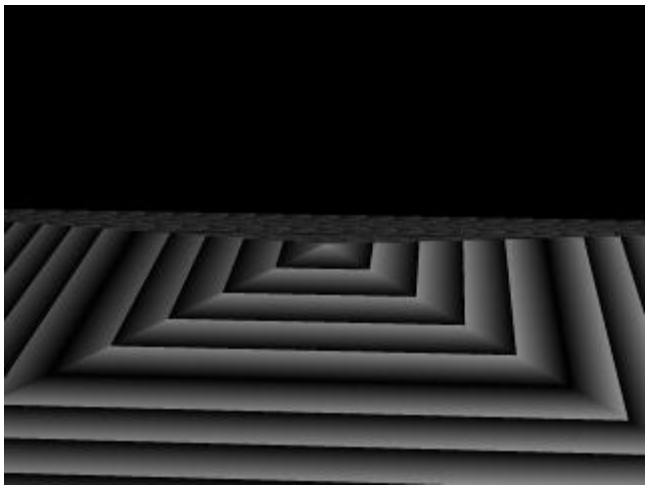
That is a very important gotcha! with conic sweeps. To see what's wrong with that, let's try and put it into practice (see file `prismdm5.pov`). We must make sure to remove the scale statement, and then replace the line which reads

```
1, // height 2
```

with

```
4, // height 2
```

This sets the second height at y=4, so let's re-render and see if the effect is the same.



Choosing a second height larger than one for the conic sweep.

Whoa! Our height is correct, but our pyramid's base is now huge! What went wrong here? Simple. The base, as we described it with the points we used actually occurs at y=1 no matter what we set the second height for. But if we do set the second height higher than one, once the sweep passes y=1, it keeps expanding outward along the same lines as it followed to our original base, making the actual base bigger and bigger as it goes.

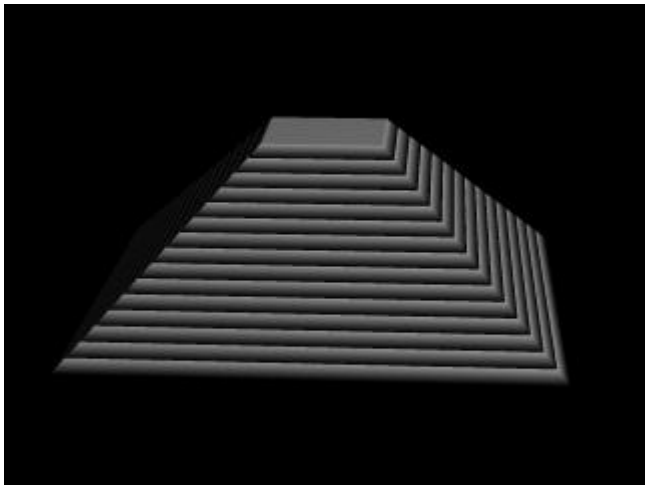
To avoid losing control of a conic sweep prism, it is usually best to let the second height stay at y=1, and use a scale statement to adjust the height from its unit size. This way we can always be sure the base's corners remain where we think they are.

That leads to one more interesting thing about conic sweeps. What if we for some reason don't want them to taper all the way to a point? What if instead of a complete pyramid, we want more of a ziggurat step? Easily done. After putting the second height back to one, and replacing our scale statement, we change the line which reads

```
  0, // height 1
```

to

```
  0.251, // height 1
```



Increasing the first height for the conic sweep.

When we re-render, we see that the sweep stops short of going all the way to its point, giving us a pyramid without a cap. Exactly how much of the cap is cut off depends on how close the first height is to the second height.

2.4.8 Superquadric Ellipsoid Object

Sometimes we want to make an object that does not have perfectly sharp edges like a box does. Then, the superquadric ellipsoid shape made by the **`superellipsoid`** is a useful object. It is described by the simple syntax:

`superellipsoid {` **`<`***Value_E*`,`** ***Value_N*** **`>}`**

Where *Value_E* and *Value_N* are float values greater than zero and less than or equal to one. Let's make a superellipsoid and experiment with the values of *Value_E* and *Value_N* to see what kind of shapes we can make.

We create a file called `supellps.pov` and edit it as follows:

```
#include "colors.inc"
camera {
  location <10, 5, -20>
  look_at 0
  angle 15
}
background { color rgb <.5, .5, .5> }
light_source { <10, 50, -100> White }
```

The addition of a gray background makes it a little easier to see our object. We now type:

```
superellipsoid { <.25, .25>
  pigment { Red }
}
```

We save the file and trace it at 200x150 **`-A`** to see the shape. It will look like a box, but the edges will be rounded off. Now let's experiment with different values of *Value_E* and *Value_N*. For the next trace, try <1, 0.2>. The shape now looks like a cylinder, but the top edges are rounded. Now try <0.1, 1>. This shape is an odd one! We don't know exactly what to call it, but it is interesting. Finally, lets try <1, 1>. Well, this is more familiar... a sphere!

There are a couple of facts about superellipsoids we should know. First, we should not use a value of 0 for either *Value_E* nor *Value_N*. This will cause POV-Ray to incorrectly make a black box instead of our desired shape. Second, very small values of *Value_E* and *Value_N* may yield strange results so they should be avoided. Finally, the Sturmian root solver will not work with superellipsoids.

Superellipsoids are finite objects so they respond to auto-bounding and can be used in CSG.

Now let's use the superellipsoid to make something that would be useful in a scene. We will make a tiled floor and place a couple of superellipsoid objects hovering over it. We can start with the file we have already made.

We rename it to `tiles.pov` and edit it so that it reads as follows:

```
#include "colors.inc"
#include "textures.inc"
camera {
  location <10, 5, -20>
  look_at 0
  angle 15
}
background { color rgb <.5, .5, .5> }
light_source{ <10, 50, -100> White }
```

Note that we have added **`#include "textures.inc"`** so we can use pre-defined textures. Now we want to define the superellipsoid which will be our tile.

```
#declare Tile = superellipsoid { <0.5, 0.1>
  scale <1, .05, 1>
}
```

Superellipsoids are roughly 2*2*2 units unless we scale them otherwise. If we wish to lay a bunch of our tiles side by side, they will have to be offset from each other so they don't overlap. We should select an offset value that is slightly more than 2 so that we have some space between the tiles to fill with grout. So we now add this:

```
#declare Offset = 2.1;
```

We now want to lay down a row of tiles. Each tile will be offset from the original by an ever-increasing amount in both the +z and -z directions. We refer to our offset and multiply by the tile's rank to determine the position of each tile in the row. We also union these tiles into a single object called **Row** like this:

```
#declare Row = union {
  object { Tile }
  object { Tile translate z*Offset }
  object { Tile translate z*Offset*2 }
  object { Tile translate z*Offset*3 }
  object { Tile translate z*Offset*4 }
  object { Tile translate z*Offset*5 }
  object { Tile translate z*Offset*6 }
  object { Tile translate z*Offset*7 }
  object { Tile translate z*Offset*8 }
  object { Tile translate z*Offset*9 }
  object { Tile translate z*Offset*10 }
  object { Tile translate -z*Offset }
  object { Tile translate -z*Offset*2 }
  object { Tile translate -z*Offset*3 }
  object { Tile translate -z*Offset*4 }
  object { Tile translate -z*Offset*5 }
  object { Tile translate -z*Offset*6 }
}
```

This gives us a single row of 17 tiles, more than enough to fill the screen. Now we must make copies of the **Row** and translate them, again by the offset value, in both the +x and -x directions in ever increasing amounts in the same manner.

```
object { Row }
object { Row translate x*Offset }
object { Row translate x*Offset*2 }
object { Row translate x*Offset*3 }
object { Row translate x*Offset*4 }
object { Row translate x*Offset*5 }
object { Row translate x*Offset*6 }
object { Row translate x*Offset*7 }
object { Row translate -x*Offset }
object { Row translate -x*Offset*2 }
object { Row translate -x*Offset*3 }
object { Row translate -x*Offset*4 }
object { Row translate -x*Offset*5 }
object { Row translate -x*Offset*6 }
object { Row translate -x*Offset*7 }
```

Finally, our tiles are complete. But we need a texture for them. To do this we union all of the **Rows** together and apply a **White Marble** pigment and a somewhat shiny reflective surface to it:

```
union{
  object { Row }
  object { Row translate x*Offset }
  object { Row translate x*Offset*2 }
  object { Row translate x*Offset*3 }
  object { Row translate x*Offset*4 }
  object { Row translate x*Offset*5 }
```

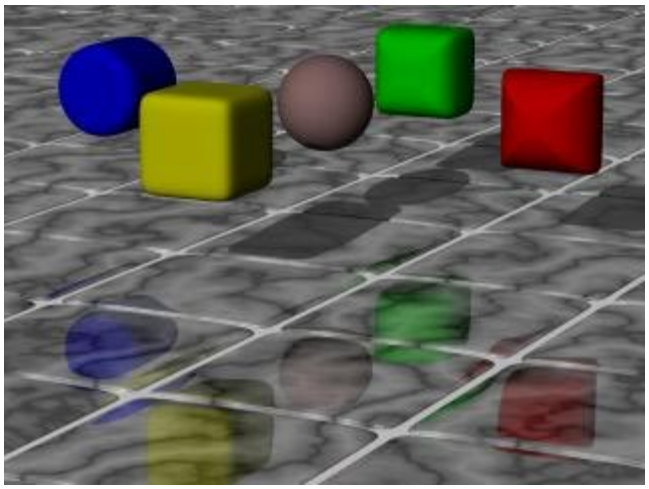
```
    object { Row translate x*Offset*6 }
    object { Row translate x*Offset*7 }
    object { Row translate -x*Offset }
    object { Row translate -x*Offset*2 }
    object { Row translate -x*Offset*3 }
    object { Row translate -x*Offset*4 }
    object { Row translate -x*Offset*5 }
    object { Row translate -x*Offset*6 }
    object { Row translate -x*Offset*7 }
    pigment { White_Marble }
    finish { phong 1 phong_size 50 reflection .35 }
  }
```

We now need to add the grout. This can simply be a white plane. We have stepped up the ambient here a little so it looks whiter.

```
  plane { y, 0  //this is the grout
    pigment { color White }
    finish { ambient .4 diffuse .7 }
  }
```

To complete our scene, let's add five different superellipsoids, each a different color, so that they hover over our tiles and are reflected in them.

```
  superellipsoid {
    <0.1, 1>
    pigment { Red }
    translate <5, 3, 0>
    scale .45
  }
  superellipsoid {
    <1, 0.25>
    pigment { Blue }
    translate <-5, 3, 0>
    scale .45
  }
  superellipsoid {
    <0.2, 0.6>
    pigment { Green }
    translate <0, 3, 5>
    scale .45
  }
  superellipsoid {
    <0.25, 0.25>
    pigment { Yellow }
    translate <0, 3, -5>
    scale .45
  }
  superellipsoid {
    <1, 1>
    pigment { Pink }
    translate y*3
    scale .45
  }
```



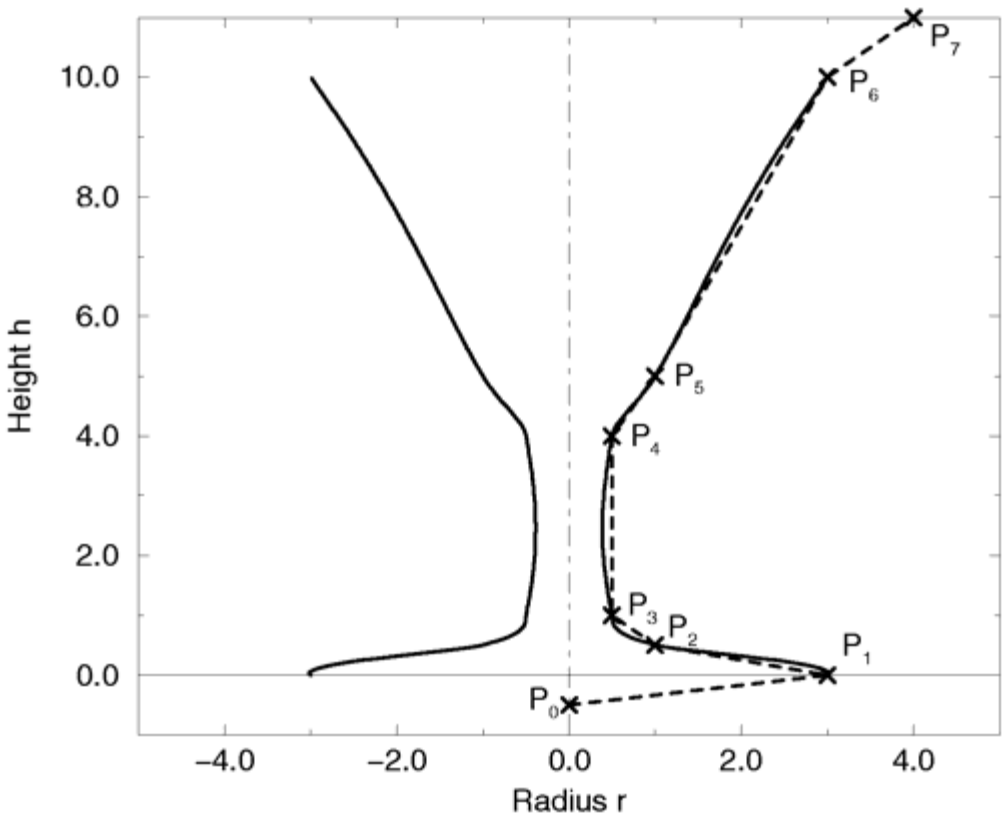
Some superellipsoids hovering above a tiled floor.

We trace the scene at 320x200 **-A** to see the result. If we are happy with that, we do a final trace at 640x480 **+A0.2**.

2.4.9 Surface of Revolution Object

Bottles, vases and glasses make nice objects in ray-traced scenes. We want to create a golden cup using the *surface of revolution* object (SOR object).

We first start by thinking about the shape of the final object. It is quite difficult to come up with a set of points that describe a given curve without the help of a modeling program supporting POV-Ray's surface of revolution object. If such a program is available we should take advantage of it.



The point configuration of our cup object.

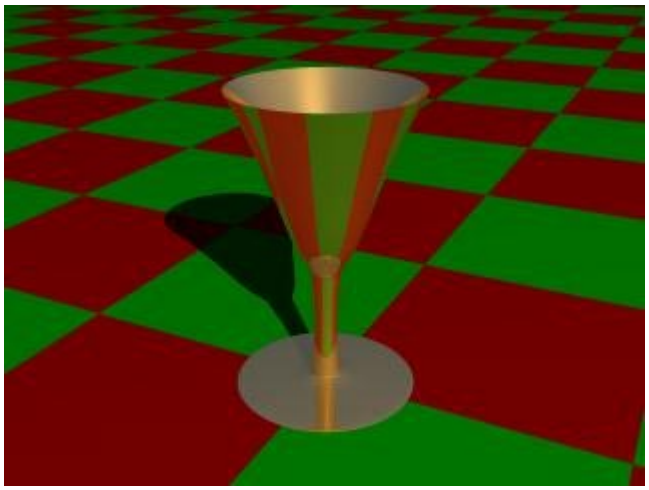
We will use the point configuration shown in the figure above. There are eight points describing the curve that will be rotated about the y-axis to get our cup. The curve was calculated using the method described in the reference section (see "Surface of Revolution").

Now it is time to come up with a scene that uses the above SOR object. We edit a file called `sordemo.pov` and enter the following text.

```
#include "colors.inc"
#include "golds.inc"
global_settings { assumed_gamma 2.2 }
camera {
  location <10, 15, -20>
  look_at <0, 5, 0>
  angle 45
}
background { color rgb<0.2, 0.4, 0.8>  }
light_source { <100, 100, -100> color rgb 1 }
plane { y, 0
  pigment { checker color Red, color Green scale 10 }
}
sor {
  8,
  <0.0,  -0.5>,
  <3.0,   0.0>,
  <1.0,   0.2>,
  <0.5,   0.4>,
```

```
    <0.5,   4.0>,
    <1.0,   5.0>,
    <3.0,  10.0>,
    <4.0,  11.0>
    texture { T_Gold_1B }
  }
```

The scene contains our cup object resting on a checkered plane. Tracing this scene results in the image below.



A surface of revolution object.

The surface of revolution is described by starting with the number of points followed by the points with ascending heights. Each point determines the radius of the curve for a given height. E. g. the first point tells POV-Ray that at height -0.5 the radius is 0. We should take care that each point has a larger height than its predecessor. If this is not the case the program will abort with an error message.

2.4.10 Text Object

Creating **`text`** objects using POV-Ray always used to mean that the letters had to be built either from CSG, a painstaking process or by using a black and white image of the letters as a height field, a method that was only somewhat satisfactory. Now, for POV-Ray 3.0, a new primitive has been introduced that can use any TrueType font to create text objects. These objects can be used in CSG, transformed and textured just like any other POV primitive.

For this tutorial, we will make two uses of the text object. First, let's just make some block letters sitting on a checkered plane. Any TTF font should do, but for this tutorial, we will use the ones bundled with POV-Ray 3.0.

We create a file called `textdemo.pov` and edit it as follows:

```
#include "colors.inc"
camera {
  location <0, 1, -10>
  look_at 0
  angle 35
}
light_source { <500,500,-1000> White }
plane { y,0
  pigment { checker Green White }
}
```

Now let's add the text object. We will use the font `timrom.ttf` and we will create the string "POV-RAY 3.0". For now, we will just make the letters red. The syntax is very simple. The first string in quotes is the font name, the

second one is the string to be rendered. The two floats are the thickness and offset values. The thickness float determines how thick the block letters will be. Values of .5 to 2 are usually best for this. The offset value will add to the kerning distance of the letters. We will leave this a 0 for now.

```
text { ttf "timrom.ttf" "POV-RAY 3.0" 1, 0
  pigment { Red }
}
```

Rendering this at 200x150 **-A**, we notice that the letters are off to the right of the screen. This is because they are placed so that the lower left front corner of the first letter is at the origin. To center the string we need to translate it -x some distance. But how far? In the docs we see that the letters are all 0.5 to 0.75 units high. If we assume that each one takes about 0.5 units of space on the x-axis, this means that the string is about 6 units long (12 characters and spaces). Let's translate the string 3 units along the negative x-axis.

```
text { ttf "timrom.ttf" "POV-RAY 3.0" 1, 0
  pigment { Red }
  translate -3*x
}
```

That's better. Now let's play around with some of the parameters of the text object. First, let's raise the thickness float to something outlandish... say 25!

```
text { ttf "timrom.ttf" "POV-RAY 3.0" 25, 0
  pigment { Red }
  translate -2.25*x
}
```

Actually, that's kind of cool. Now let's return the thickness value to 1 and try a different offset value. Change the offset float from 0 to 0.1 and render it again.

Wait a minute?! The letters go wandering off up at an angle! That is not what the docs describe! It almost looks as if the offset value applies in both the x- and y-axis instead of just the x axis like we intended. Could it be that a vector is called for here instead of a float? Let's try it. We replace **`0.1`** with **`0.1*x`** and render it again.

That works! The letters are still in a straight line along the x-axis, just a little further apart. Let's verify this and try to offset just in the y-axis. We replace **`0.1*x`** with **`0.1*y`**. Again, this works as expected with the letters going up to the right at an angle with no additional distance added along the x-axis. Now let's try the z-axis. We replace **`0.1*y`** with **`0.1*z`**. Rendering this yields a disappointment. No offset occurs! The offset value can only be applied in the x- and y-directions.

Let's finish our scene by giving a fancier texture to the block letters, using that cool large thickness value, and adding a slight y-offset. For fun, we will throw in a sky sphere, dandy up our plane a bit, and use a little more interesting camera viewpoint (we render the following scene at 640x480 **+A0.2**):

```
#include "colors.inc"
camera {
  location <-5,.15,-2>
  look_at <.3,.2,1>
  angle 35
}
light_source { <500,500,-1000> White }
plane { y,0
  texture {
    pigment { SeaGreen }
    finish { reflection .35 specular 1 }
    normal { ripples .35 turbulence .5 scale .25 }
  }
}
text { ttf "timrom.ttf" "POV-RAY 3.0" 25, 0.1*y
  pigment { BrightGold }
```

```
    finish { reflection .25 specular 1 }
    translate -3*x
  }
  #include "skies.inc"
  sky_sphere { S_Cloud5 }
```

Let's try using text in a CSG object. We will attempt to create an inlay in a stone block using a text object. We create a new file called `textcsg.pov` and edit it as follows:

```
  #include "colors.inc"
  #include "stones.inc"
  background { color rgb 1 }
  camera {
    location <-3, 5, -15>
    look_at 0
    angle 25
  }
  light_source { <500,500,-1000> White }
```

Now let's create the block. We want it to be about eight units across because our text string "POV-RAY 3.0" is about six units long. We also want it about four units high and about one unit deep. But we need to avoid a potential coincident surface with the text object so we will make the first z-coordinate 0.1 instead of 0. Finally, we will give this block a nice stone texture.

```
  box { <-3.5, -1, 0.1>, <3.5, 1, 1>
    texture { T_Stone10 }
  }
```

Next, we want to make the text object. We can use the same object we used in the first tutorial except we will use slightly different thickness and offset values.

```
  text { ttf "timrom.ttf" "POV-RAY 3.0" 0.15, 0
    pigment { BrightGold }
    finish { reflection .25 specular 1 }
    translate -3*x
  }
```

We remember that the text object is placed by default so that its front surface lies directly on the x-y-plane. If the front of the box begins at z=0.1 and thickness is set at 0.15, the depth of the inlay will be 0.05 units. We place a difference block around the two objects.

```
  difference {
    box { <-3.5, -1, 0.1>, <3.5, 1, 1>
      texture { T_Stone10 }
    }
    text { ttf "timrom.ttf" "POV-RAY 3.0" 0.15, 0
      pigment { BrightGold }
      finish { reflection .25 specular 1 }
      translate -3*x
    }
  }
```

Text carved from stone.

We render this at 200x150 **-A**. We can see the inlay clearly and that it is indeed a bright gold color. We re-render at 640x480 **+A0.2** to see the results more clearly, but be forewarned... this trace will take a little time.

2.4.11 Torus Object

A **torus** can be thought of as a donut or an inner-tube. It is a shape that is vastly useful in many kinds of CSG so POV-Ray has adopted this 4th order quartic polynomial as a primitive shape. The syntax for a torus is so simple that it makes it a very easy shape to work with once we learn what the two float values mean. Instead of a lecture on the subject, let's create one and do some experiments with it.

We create a file called `tordemo.pov` and edit it as follows:

```
#include "colors.inc"
camera {
  location <0, .1, -25>
  look_at 0
  angle 30
}
background { color Gray50 } // to make the torus easy to see
light_source{ <300, 300, -1000> White }
torus { 4, 1        // major and minor radius
  rotate -90*x      // so we can see it from the top
  pigment { Green }
}
```

We trace the scene. Well, it's a donut alright. Let's try changing the major and minor radius values and see what happens. We change them as follows:

```
torus { 5, .25      // major and minor radius
```

That looks more like a hula-hoop! Let's try this:

```
torus { 3.5, 2.5    // major and minor radius
```

Whoa! A donut with a serious weight problem!

With such a simple syntax, there isn't much else we can do to a torus besides change its texture... or is there? Let's see...

Torii are very useful objects in CSG. Let's try a little experiment. We make a difference of a torus and a box:

```
difference {
  torus { 4, 1
    rotate x*-90  // so we can see it from the top
  }
  box { <-5, -5, -1>, <5, 0, 1> }
  pigment { Green }
}
```

Interesting... a half-torus. Now we add another one flipped the other way. Only, let's declare the original half-torus and the necessary transformations so we can use them again:

```
#declare Half_Torus = difference {
  torus { 4, 1
    rotate -90*x  // so we can see it from the top
  }
  box { <-5, -5, -1>, <5, 0, 1> }
  pigment { Green }
}
#declare Flip_It_Over = 180*x;
#declare Torus_Translate = 8;  // twice the major radius
```

Now we create a union of two **`Half_Torus`** objects:

```
union {
  object { Half_Torus }
  object { Half_Torus
    rotate Flip_It_Over
    translate Torus_Translate*x
  }
}
```

This makes an S-shaped object, but we can't see the whole thing from our present camera. Let's add a few more links, three in each direction, move the object along the +z-direction and rotate it about the +y-axis so we can see more of it. We also notice that there appears to be a small gap where the half Torii meet. This is due to the fact that we are viewing this scene from directly on the x-z-plane. We will change the camera's y-coordinate from 0 to 0.1 to eliminate this.

```
union {
  object { Half_Torus }
  object { Half_Torus
    rotate Flip_It_Over
    translate x*Torus_Translate
  }
  object { Half_Torus
    translate x*Torus_Translate*2
  }
  object { Half_Torus
    rotate Flip_It_Over
    translate x*Torus_Translate*3
  }
  object { Half_Torus
    rotate Flip_It_Over
    translate -x*Torus_Translate
  }
  object { Half_Torus
    translate -x*Torus_Translate*2
  }
  object { Half_Torus
    rotate Flip_It_Over
    translate -x*Torus_Translate*3
```

```
    }
    object { Half_Torus
      translate -x*Torus_Translate*4
    }
    rotate y*45
    translate z*20
  }
```

Rendering this we see a cool, undulating, snake-like something-or-other. Neato. But we want to model something useful, something that we might see in real life. How about a chain?

Thinking about it for a moment, we realize that a single link of a chain can be easily modeled using two half tori and two cylinders. We create a new file. We can use the same camera, background, light source and declared objects and transformations as we used in `tordemo.pov`:

```
#include "colors.inc"
camera {
  location <0, .1, -25>
  look_at 0
  angle 30
}
background { color Gray50 }
light_source{ <300, 300, -1000> White }
#declare Half_Torus = difference {
  torus { 4,1
    sturm
    rotate x*-90  // so we can see it from the top
  }
  box { <-5, -5, -1>, <5, 0, 1> }
  pigment { Green }
}
#declare Flip_It_Over = x*180;
#declare Torus_Translate = 8;
```

Now, we make a complete torus of two half toruses:

```
union {
  object { Half_Torus }
  object { Half_Torus rotate Flip_It_Over }
}
```

This may seem like a wasteful way to make a complete torus, but we are really going to move each half apart to make room for the cylinders. First, we add the declared cylinder before the union:

```
#declare Chain_Segment = cylinder { <0, 4, 0>, <0, -4, 0>, 1
  pigment { Green }
}
```

We then add two **`Chain_Segments`** to the union and translate them so that they line up with the minor radius of the torus on each side:

```
union {
  object { Half_Torus }
  object { Half_Torus rotate Flip_It_Over }
  object { Chain_Segment translate  x*Torus_Translate/2 }
  object { Chain_Segment translate -x*Torus_Translate/2 }
}
```

Now we translate the two half toruses +y and -y so that the clipped ends meet the ends of the cylinders. This distance is equal to half of the previously declared **`Torus_Translate`**:

```
union {
```

```
    object { Half_Torus
      translate y*Torus_Translate/2
    }
    object { Half_Torus
      rotate Flip_It_Over
      translate -y*Torus_Translate/2
    }
    object { Chain_Segment
      translate x*Torus_Translate/2
    }
    object { Chain_Segment
      translate -x*Torus_Translate/2
    }
  }
```

We render this and viola! A single link of a chain. But we aren't done yet! Whoever heard of a green chain? We would rather use a nice metallic color instead. First, we remove any pigment blocks in the declared torsos and cylinders. Then we add the following before the union:

```
  #declare Chain_Gold = texture {
    pigment { BrightGold }
    finish {
      ambient .1
      diffuse .4
      reflection .25
      specular 1
      metallic
    }
  }
```

We then add the texture to the union and declare the union as a single link:

```
  #declare Link = union {
    object { Half_Torus
      translate y*Torus_Translate/2
    }
    object { Half_Torus
      rotate Flip_It_Over
      translate -y*Torus_Translate/2
    }
    object { Chain_Segment
      translate x*Torus_Translate/2
    }
    object { Chain_Segment
      translate -x*Torus_Translate/2
    }
    texture { Chain_Gold }
  }
```

Now we make a union of two links. The second one will have to be translated +y so that its inner wall just meets the inner wall of the other link, just like the links of a chain. This distance turns out to be double the previously declared **`Torus_Translate`** minus 2 (twice the minor radius). This can be described by the expression:

```
  Torus_Translate*2-2*y
```

We declare this expression as follows:

```
  #declare Link_Translate = Torus_Translate*2-2*y;
```

In the object block, we will use this declared value so that we can multiply it to create other links. Now, we rotate the second link **`90*y`** so that it is perpendicular to the first, just like links of a chain. Finally, we scale the union by 1/4 so that we can see the whole thing:

```
union {
  object { Link }
  object { Link translate y*Link_Translate rotate y*90 }
  scale .25
}
```

We render this and we will see a very realistic pair of links. If we want to make an entire chain, we must declare the above union and then create another union of this declared object. We must be sure to remove the scaling from the declared object:

```
#declare Link_Pair =
union {
  object { Link }
  object { Link translate y*Link_Translate rotate y*90 }
}
```

Now we declare our chain:

```
#declare Chain = union {
  object { Link_Pair}
  object { Link_Pair translate  y*Link_Translate*2 }
  object { Link_Pair translate  y*Link_Translate*4 }
  object { Link_Pair translate  y*Link_Translate*6 }
  object { Link_Pair translate -y*Link_Translate*2 }
  object { Link_Pair translate -y*Link_Translate*4 }
  object { Link_Pair translate -y*Link_Translate*6 }
}
```

And finally we create our chain with a couple of transformations to make it easier to see. These include scaling it down by a factor of 1/10, and rotating it so that we can clearly see each link:

```
object { Chain scale .1 rotate <0, 45, -45> }
```

The torus object can be used to create chains.

We render this and we should see a very realistic gold chain stretched diagonally across the screen.

2.5 The Light Source

In any ray-traced scene, the light needed to illuminate our objects and their surfaces must come from a light source. There are many kinds of light sources available in POV-Ray and careful use of the correct kind can yield very impressive results. Let's take a moment to explore some of the different kinds of light sources and their various parameters.

2.5.1 The Pointlight Source

Pointlights are exactly what the name indicates. A pointlight has no size, is invisible and illuminates everything in the scene equally no matter how far away from the light source it may be (this behavior can be changed). This is the simplest and most basic light source. There are only two important parameters, location and color. Let's design a simple scene and place a pointlight source in it.

We create a new file and name it `litedemo.pov`. We edit it as follows:

```
#include "colors.inc"
#include "textures.inc"
camera {
  location  <-4, 3, -9>
  look_at   <0, 0, 0>
  angle 48
}
```

We add the following simple objects:

```
plane { y, -1
  texture {
    pigment {
      checker
      color rgb<0.5, 0, 0>
      color rgb<0, 0.5, 0.5>
    }
    finish {
      diffuse 0.4
      ambient 0.2
      phong 1
      phong_size 100
      reflection 0.25
    }
  }
}
torus { 1.5, 0.5
  texture { Brown_Agate }
  rotate <90, 160, 0>
  translate  <-1, 1, 3>
}
box { <-1, -1, -1>, <1, 1, 1>
  texture { DMFLightOak }
  translate  <2, 0, 2.3>
}
cone { <0,1,0>, 0, <0,0,0>, 1
  texture { PinkAlabaster }
  scale <1, 3, 1>
  translate  <-2, -1, -1>
}
sphere { <0,0,0>,1
  texture { Sapphire_Agate }
```

```
    translate  <1.5, 0, -2>
  }
```

Now we add a pointlight:

```
  light_source {
    <2, 10, -3>
    color White
  }
```

We render this at 200x150 **-A** and see that the objects are clearly visible with sharp shadows. The sides of curved objects nearest the light source are brightest in color with the areas that are facing away from the light source being darkest. We also note that the checkered plane is illuminated evenly all the way to the horizon. This allows us to see the plane, but it is not very realistic.

2.5.2 The Spotlight Source

Spotlights are a very useful type of light source. They can be used to add highlights and illuminate features much as a photographer uses spots to do the same thing. To create a spotlight simply add the **spotlight** keyword to a regular point light. There are a few more parameters with spotlights than with pointlights. These are **radius**, **falloff**, **tightness** and **point_at**. The **radius** parameter is the angle of the fully illuminated cone. The **falloff** parameter is the angle of the *umbra* cone where the light falls off to darkness. The **tightness** is a parameter that determines the rate of the light falloff. The **point_at** parameter is just what it says, the location where the spotlight is pointing to. Let's change the light in our scene as follows:

```
  light_source {
    <0, 10, -3>
    color White
    spotlight
    radius 15
    falloff 20
    tightness 10
    point_at <0, 0, 0>
  }
```

We render this at 200x150 **-A** and see that only the objects are illuminated. The rest of the plane and the outer portions of the objects are now unlit. There is a broad falloff area but the shadows are still razor sharp. Let's try fiddling with some of these parameters to see what they do. We change the falloff value to 16 (it must always be larger than the radius value) and render again. Now the falloff is very narrow and the objects are either brightly lit or in total darkness. Now we change falloff back to 20 and change the tightness value to 100 (higher is tighter) and render again. The spotlight appears to have gotten much smaller but what has really happened is that the falloff has become so steep that the radius actually appears smaller.

We decide that a tightness value of 10 (the default) and a falloff value of 18 are best for this spotlight and we now want to put a few spots around the scene for effect. Let's place a slightly narrower blue and a red one in addition to the white one we already have:

```
  light_source {
    <10, 10, -1>
    color Red
    spotlight
    radius 12
    falloff 14
    tightness 10
    point_at <2, 0, 0>
  }
  light_source {
    <-12, 10, -1>
    color Blue
```

```
    spotlight
    radius 12
    falloff 14
    tightness 10
    point_at <-2, 0, 0>
  }
```

Rendering this we see that the scene now has a wonderfully mysterious air to it. The three spotlights all converge on the objects making them blue on one side and red on the other with enough white in the middle to provide a balance.

2.5.3 The Cylindrical Light Source

Spotlights are cone shaped, meaning that their effect will change with distance. The farther away from the spotlight an object is, the larger the apparent radius will be. But we may want the radius and falloff to be a particular size no matter how far away the spotlight is. For this reason, cylindrical light sources are needed. A cylindrical light source is just like a spotlight, except that the radius and falloff regions are the same no matter how far from the light source our object is. The shape is therefore a cylinder rather than a cone. We can specify a cylindrical light source by replacing the **`spotlight`** keyword with the **`cylinder`** keyword. We try this now with our scene by replacing all three spotlights with cylinder lights and rendering again. We see that the scene is much dimmer. This is because the cylindrical constraints do not let the light spread out like in a spotlight. Larger radius and falloff values are needed to do the job. We try a radius of 20 and a falloff of 30 for all three lights. That's the ticket!

2.5.4 The Area Light Source

So far all of our light sources have one thing in common. They produce sharp shadows. This is because the actual light source is a point that is infinitely small. Objects are either in direct sight of the light, in which case they are fully illuminated, or they are not, in which case they are fully shaded. In real life, this kind of stark light and shadow situation exists only in outer space where the direct light of the sun pierces the total blackness of space. But here on Earth, light bends around objects, bounces off objects, and usually the source has some dimension, meaning that it can be partially hidden from sight (shadows are not sharp anymore). They have what is known as an *umbra*, or an area of fuzziness where there is neither total light or shade. In order to simulate these *soft* shadows, a ray-tracer must give its light sources dimension. POV-Ray accomplishes this with a feature known as an area light.

Area lights have dimension in two axis'. These are specified by the first two vectors in the area light syntax. We must also specify how many lights are to be in the array. More will give us cleaner soft shadows but will take longer to render. Usually a 3*3 or a 5*5 array will suffice. We also have the option of specifying an adaptive value. The **`adaptive`** keyword tells the ray-tracer that it can adapt to the situation and send only the needed rays to determine the value of the pixel. If adaptive is not used, a separate ray will be sent for every light in the area light. This can really slow things down. The higher the adaptive value the cleaner the umbra will be but the longer the trace will take. Usually an adaptive value of 1 is sufficient. Finally, we probably should use the **`jitter`** keyword. This tells the ray-tracer to slightly move the position of each light in the area light so that the shadows appear truly soft instead of giving us an umbra consisting of closely banded shadows.

OK, let's try one. We comment out the cylinder lights and add the following:

```
  light_source {
    <2, 10, -3>
    color White
    area_light <5, 0, 0>, <0, 0, 5>, 5, 5
    adaptive 1
    jitter
  }
```

This is a white area light centered at <2,10,-3>. It is 5 units (along the x-axis) by 5 units (along the z-axis) in size and has 25 (5*5) lights in it. We have specified adaptive 1 and jitter. We render this at 200x150 **`-A`**.

Right away we notice two things. The trace takes quite a bit longer than it did with a point or a spotlight and the shadows are no longer sharp! They all have nice soft umbrae around them. Wait, it gets better.

Spotlights and cylinder lights can be area lights too! Remember those sharp shadows from the spotlights in our scene? It would not make much sense to use a 5*5 array for a spotlight, but a smaller array might do a good job of giving us just the right amount of umbra for a spotlight. Let's try it. We comment out the area light and change the cylinder lights so that they read as follows:

```
light_source {
  <2, 10, -3>
  color White
  spotlight
  radius 15
  falloff 18
  tightness 10
  area_light <1, 0, 0>, <0, 0, 1>, 2, 2
  adaptive 1
  jitter
  point_at <0, 0, 0>
}
light_source {
  <10, 10, -1>
  color Red
  spotlight
  radius 12
  falloff 14
  tightness 10
  area_light <1, 0, 0>, <0, 0, 1>, 2, 2
  adaptive 1
  jitter
  point_at <2, 0, 0>
}
light_source {
  <-12, 10, -1>
  color Blue
  spotlight
  radius 12
  falloff 14
  tightness 10
  area_light <1, 0, 0>, <0, 0, 1>, 2, 2
  adaptive 1
  jitter
  point_at <-2, 0, 0>
}
```

We now have three area-spotlights, one unit square consisting of an array of four (2*2) lights, three different colors, all shining on our scene. We render this at 200x150 **-A**. It appears to work perfectly. All our shadows have small, tight umbrae, just the sort we would expect to find on an object under a real spotlight.

2.5.5 The Ambient Light Source

The *ambient light source* is used to simulate the effect of inter-diffuse reflection. If there wasn't inter-diffuse reflection all areas not directly lit by a light source would be completely dark. POV-Ray uses the **ambient** keyword to determine how much light coming from the ambient light source is reflected by a surface.

By default the ambient light source, which emits its light everywhere and in all directions, is pure white (**rgb <1,1,1>**). Changing its color can be used to create interesting effects. First of all the overall light level of the

scene can be adjusted easily. Instead of changing all ambient values in every finish only the ambient light source is modified. By assigning different colors we can create nice effects like a moody reddish ambient lighting. For more details about the ambient light source see "Ambient Light".

Below is an example of a red ambient light source.

```
global_settings { ambient_light rgb<1, 0, 0> }
```

2.5.6 Light Source Specials

2.5.6.1 Using Shadowless Lights

Light sources can be assigned the **shadowless** keyword and no shadows will be cast due to its presence in a scene. Sometimes, scenes are difficult to illuminate properly using the lights we have chosen to illuminate our objects. It is impractical and unrealistic to apply a higher ambient value to the texture of every object in the scene. So instead, we would place a couple of *fill lights* around the scene. Fill lights are simply dimmer lights with the **shadowless** keyword that act to boost the illumination of other areas of the scene that may not be lit well. Let's try using one in our scene.

Remember the three colored area spotlights? We go back and un-comment them and comment out any other lights we have made. Now we add the following:

```
light_source {
  <0, 20, 0>
  color Gray50
  shadowless
}
```

This is a fairly dim light 20 units over the center of the scene. It will give a dim illumination to all objects including the plane in the background. We render it and see.

2.5.6.2 Assigning an Object to a Light Source

Light sources are invisible. They are just a location where the light appears to be coming from. They have no true size or shape. If we want our light source to be a visible shape, we can use the **looks_like** keyword. We can specify that our light source can look like any object we choose. When we use **looks_like**, then **no_shadow** is applied to the object automatically. This is done so that the object will not block any illumination from the light source. If we want some blocking to occur (as in a lampshade), it is better to simply use a union to do the same thing. Let's add such an object to our scene. Here is a light bulb we have made just for this purpose:

```
#declare Lightbulb = union {
  merge {
    sphere { <0,0,0>,1 }
    cylinder { <0,0,1>, <0,0,0>, 1
      scale <0.35, 0.35, 1.0>
      translate  0.5*z
    }
    texture {
      pigment {color rgb <1, 1, 1>}
      finish {ambient .8 diffuse .6}
    }
  }
  cylinder { <0,0,1>, <0,0,0>, 1
    scale <0.4, 0.4, 0.5>
    texture { Brass_Texture }
    translate  1.5*z
  }
  rotate -90*x
```

```
    scale .5
  }
```

Now we add the light source:

```
  light_source {
    <0, 2, 0>
    color White
    looks_like { Lightbulb }
  }
```

Rendering this we see that a fairly believable light bulb now illuminates the scene. However, if we do not specify a high ambient value, the light bulb is not lit by the light source. On the plus side, all of the shadows fall away from the light bulb, just as they would in a real situation. The shadows are sharp, so let's make our bulb an area light:

```
  light_source {
    <0, 2, 0>
    color White
    area_light <1, 0, 0>, <0, 1, 0>, 2, 2
    adaptive 1
    jitter
    looks_like { Lightbulb }
  }
```

We note that we have placed this area light in the x-y-plane instead of the x-z-plane. We also note that the actual appearance of the light bulb is not affected in any way by the light source. The bulb must be illuminated by some other light source or by, as in this case, a high ambient value.

2.5.6.3 Using Light Fading

If it is realism we want, it is not realistic for the plane to be evenly illuminated off into the distance. In real life, light gets scattered as it travels so it diminishes its ability to illuminate objects the farther it gets from its source. To simulate this, POV-Ray allows us to use two keywords: **`fade_distance`**, which specifies the distance at which full illumination is achieved, and **`fade_power`**, an exponential value which determines the actual rate of attenuation. Let's apply these keywords to our fill light.

First, we make the fill light a little brighter by changing **`Gray50`** to **`Gray75`**. Now we change that fill light as follows:

```
  light_source {
    <0, 20, 0>
    color Gray75
    fade_distance 5
    fade_power 1
    shadowless
  }
```

This means that the full value of the fill light will be achieved at a distance of 5 units away from the light source. The fade power of 1 means that the falloff will be linear (the light falls of at a constant rate). We render this to see the result.

That definitely worked! Now let's try a fade power of 2 and a fade distance of 10. Again, this works well. The falloff is much faster with a fade power of 2 so we had to raise the fade distance to 10.

2.6 Simple Texture Options

The pictures rendered so far where somewhat boring regarding the appearance of the objects. Let's add some fancy features to the texture.

2.6.1 Surface Finishes

One of the main features of a ray-tracer is its ability to do interesting things with surface finishes such as highlights and reflection. Let's add a nice little Phong highlight (shiny spot) to the sphere. To do this we need to add a **finish** keyword followed by a parameter. We change the definition of the sphere to this:

```
sphere { <0, 1, 2>, 2
  texture {
    pigment { color Yellow } // Yellow is pre-defined in COLORS.INC
    finish { phong 1 }
  }
}
```

We render the scene. The **phong** keyword adds a highlight the same color of the light shining on the object. It adds a lot of credibility to the picture and makes the object look smooth and shiny. Lower values of phong will make the highlight less bright (values should be between 0 and 1).

2.6.2 Adding Bumpiness

The highlight we have added illustrates how 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 we 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*. This is a vector which points away from the surface and is perpendicular to it. By artificially modifying (or perturbing) this normal vector we can simulate bumps. We change the scene to read as follows and render it:

```
sphere { <0, 1, 2>, 2
  texture {
    pigment { color Yellow }
    normal { bumps 0.4 scale 0.2 }
    finish { phong 1}
  }
}
```

This tells POV-Ray to use the **bumps** pattern to modify the surface normal. The value 0.4 controls the apparent depth of the bumps. Usually the bumps are about 1 unit wide which doesn't work very well with a sphere of radius 2. The scale makes the bumps 1/5th as wide but does not affect their depth.

2.6.3 Creating Color Patterns

We can do more than assigning a solid color to an object. We can create complex patterns in the pigment block like in this example:

```
sphere { <0, 1, 2>, 2
  texture {
    pigment {
      wood
      color_map {
        [0.0 color DarkTan]
        [0.9 color DarkBrown]
        [1.0 color VeryDarkBrown]
      }
      turbulence 0.05
      scale <0.2, 0.3, 1>
    }
```

```
    finish { phong 1 }
  }
}
```

The keyword **wood** specifies a pigment pattern of concentric rings like rings in wood. The **color_map** keyword specifies that the color of the wood should blend from **DarkTan** to **DarkBrown** over the first 90% of the vein and from **DarkBrown** to **VeryDarkBrown** over the remaining 10%. The **turbulence** keyword slightly stirs up the pattern so the veins aren't perfect circles and the **scale** keyword adjusts the size of the pattern.

Most patterns are set up by default to give us one *feature* across a sphere of radius 1.0. A feature is very roughly defined as a color transition. For example, a wood texture would have one band on a sphere of radius 1.0. In this example we scale the pattern using the **scale** keyword followed by a vector. In this case we scaled 0.2 in the x direction, 0.3 in the y direction and the z direction is scaled by 1, which leaves it unchanged. Scale values larger than one will stretch an element. Scale values smaller than one will squish an element. A scale value of one will leave an element unchanged.

2.6.4 Pre-defined Textures

POV-Ray has some very sophisticated textures pre-defined in the standard include files `glass.inc`, `metals.inc`, `stones.inc` and `woods.inc`. Some are entire textures with pigment, normal and/or finish parameters already defined. Some are just pigments or just finishes. We change the definition of our sphere to the following and then re-render it:

```
sphere { <0, 1, 2>, 2
  texture {
    pigment {
      DMFWood4       // pre-defined in textures.inc
      scale 4        // scale by the same amount in all
                     // directions
    }
    finish { Shiny } // pre-defined in finish.inc
  }
}
```

The pigment identifier **DMFWood4** has already been scaled down quite small when it was defined. For this example we want to scale the pattern larger. Because we want to scale it uniformly we can put a single value after the scale keyword rather than a vector of x, y, z scale factors.

We look through the file `textures.inc` to see what pigments and finishes are defined and try them out. We just insert the name of the new pigment where **DMFWood4** is now or try a different finish in place of **Shiny** and re-render our file.

Here is an example of using a complete texture identifier rather than just the pieces.

```
sphere { <0, 1, 2>, 2
  texture { PinkAlabaster }
}
```

2.7 Advanced Texture Options

The extremely powerful texturing ability is one thing that really sets POV-Ray apart from other raytracers. So far we have not really tried anything too complex but by now we should be comfortable enough with the program's syntax to try some of the more advanced texture options.

Obviously, we cannot try them all. It would take a tutorial a lot more pages to use every texturing option available in POV-Ray. For this limited tutorial, we will content ourselves to just trying a few of them to give an idea of how textures are created. With a little practice, we will soon be creating beautiful textures of our own.

Note that early versions of POV-Ray made a distinction between pigment and normal patterns, i. e. patterns that could be used inside a **normal** or **pigment** statement. With POV-Ray 3.0 this restriction was removed so that all patterns listed in section "Patterns" can be used as a pigment or normal pattern.

2.7.1 Pigments

Every surface must have a color. In POV-Ray this color is called a **pigment**. It does not have to be a single color. It can be a color pattern, a color list or even an image map. Pigments can also be layered one on top of the next so long as the uppermost layers are at least partially transparent so the ones beneath can show through. Let's play around with some of these kinds of pigments.

We create a file called `texdemo.pov` and edit it as follows:

```
#include "colors.inc"
camera {
  location <1, 1, -7>
  look_at 0
  angle 36
}
light_source { <1000, 1000, -1000> White }
plane { y, -1.5
  pigment { checker Green, White }
}
sphere { <0,0,0>, 1
  pigment { Red }
}
```

Giving this file a quick test render at 200x150 **-A** we see that it is a simple red sphere against a green and white checkered plane. We will be using the sphere for our textures.

2.7.1.1 Using Color List Pigments

Before we begin we should note that we have already made one kind of pigment, the color list pigment. In the previous example we have used a checkered pattern on our plane. There are two other kinds of color list pigments, **brick** and **hexagon**. Let's quickly try each of these. First, we change the plane's pigment as follows:

```
pigment { hexagon Green, White, Yellow }
```

Rendering this we see a three-color hexagonal pattern. Note that this pattern requires three colors. Now we change the pigment to...

```
pigment { brick Gray75, Red rotate -90*x scale .25 }
```

Looking at the resulting image we see that the plane now has a brick pattern. We note that we had to rotate the pattern to make it appear correctly on the flat plane. This pattern normally is meant to be used on vertical surfaces. We also had to scale the pattern down a bit so we could see it more easily. We can play around with these color list pigments, change the colors, etc. until we get a floor that we like.

2.7.1.2 Using Pigment and Patterns

Let's begin texturing our sphere by using a pattern and a color map consisting of three colors. We replace the pigment block with the following.

```
pigment {
  gradient x
  color_map {
    [0.00 color Red]
    [0.33 color Blue]
    [0.66 color Yellow]
```

```
    [1.00 color Red]
  }
}
```

Rendering this we see that the **gradient** pattern gives us an interesting pattern of vertical stripes. We change the gradient direction to y. The stripes are horizontal now. We change the gradient direction to z. The stripes are now more like concentric rings. This is because the gradient direction is directly away from the camera. We change the direction back to x and add the following to the pigment block.

```
pigment {
  gradient x
  color_map {
    [0.00 color Red]
    [0.33 color Blue]
    [0.66 color Yellow]
    [1.00 color Red]
  }
  rotate -45*z          // <- add this line
}
```

The vertical bars are now slanted at a 45 degree angle. All patterns can be rotated, scaled and translated in this manner. Let's now try some different types of patterns. One at a time, we substitute the following keywords for **gradient x** and render to see the result: **bozo**, **marble**, **agate**, **granite**, **leopard**, **spotted** and **wood** (if we like we can test all patterns listed in section "Patterns").

Rendering these we see that each results in a slightly different pattern. But to get really good results each type of pattern requires the use of some pattern modifiers.

2.7.1.3 Using Pattern Modifiers

Let's take a look at some pattern modifiers. First, we change the pattern type to bozo. Then we add the following change.

```
pigment {
  bozo
  frequency 3            // <- add this line
  color_map {
    [0.00 color Red]
    [0.33 color Blue]
    [0.66 color Yellow]
    [1.00 color Red]
  }
  rotate -45*z
}
```

The **frequency** modifier determines the number of times the color map repeats itself per unit of size. This change makes the **bozo** pattern we saw earlier have many more bands in it. Now we change the pattern type to **marble**. When we rendered this earlier, we saw a banded pattern similar to **gradient y** that really did not look much like marble at all. This is because marble really is a kind of gradient and it needs another pattern modifier to look like marble. This modifier is called **turbulence**. We change the line **frequency 3** to **turbulence 1** and render again. That's better! Now let's put **frequency 3** back in right after the turbulence and take another look. Even more interesting!

But wait, it gets better! Turbulence itself has some modifiers of its own. We can adjust the turbulence several ways. First, the float that follows the **turbulence** keyword can be any value with higher values giving us more turbulence. Second, we can use the keywords **omega**, **lambda** and **octaves** to change the turbulence parameters. Let's try this now:

```
pigment {
```

```
    marble
    turbulence 0.5
    lambda 1.5
    omega 0.8
    octaves 5
    frequency 3
    color_map {
      [0.00 color Red]
      [0.33 color Blue]
      [0.66 color Yellow]
      [1.00 color Red]
    }
    rotate 45*z
  }
```

Rendering this we see that the turbulence has changed and the pattern looks different. We play around with the numerical values of turbulence, lambda, omega and octaves to see what they do.

2.7.1.4 Using Transparent Pigments and Layered Textures

Pigments are described by numerical values that give the rgb value of the color to be used (like **`color rgb<1,0,0>`** giving us a red color). But this syntax will give us more than just the rgb values. We can specify filtering transparency by changing it as follows: **`color rgbf<1,0,0,1>`**. The *f* stands for **`filter`**, POV-Ray's word for filtered transparency. A value of one means that the color is completely transparent, but still filters the light according to what the pigment is. In this case, the color will be a transparent red, like red cellophane.

There is another kind of transparency in POV-Ray. It is called *transmittance* or non-filtering transparency (the keyword is **`transmit`**). It is different from **`filter`** in that it does not filter the light according to the pigment color. It instead allows all the light to pass through unchanged. It can be specified like this: **`rgbt <1,0,0,1>`**.

Let's use some transparent pigments to create another kind of texture, the layered texture. Returning to our previous example, declare the following texture.

```
  #declare LandArea = texture {
      pigment {
        agate
        turbulence 1
        lambda 1.5
        omega .8
        octaves 8
        color_map {
          [0.00 color rgb <.5, .25, .15>]
          [0.33 color rgb <.1, .5, .4>]
          [0.86 color rgb <.6, .3, .1>]
          [1.00 color rgb <.5, .25, .15>]
        }
      }
    }
```

This texture will be the land area. Now let's make the oceans by declaring the following.

```
  #declare OceanArea = texture {
      pigment {
        bozo
        turbulence .5
        lambda 2
        color_map {
          [0.00, 0.33 color rgb <0, 0, 1>
```

```
                color rgb <0, 0, 1>]
        [0.33, 0.66 color rgbf <1, 1, 1, 1>
                color rgbf <1, 1, 1, 1>]
        [0.66, 1.00 color rgb <0, 0, 1>
                color rgb <0, 0, 1>]
      }
    }
  }
```

Note how the ocean is the opaque blue area and the land is the clear area which will allow the underlying texture to show through.

Now, let's declare one more texture to simulate an atmosphere with swirling clouds.

```
#declare CloudArea = texture {
  pigment {
    agate
    turbulence 1
    lambda 2
    frequency 2
    color_map {
      [0.0 color rgbf <1, 1, 1, 1>]
      [0.5 color rgbf <1, 1, 1, .35>]
      [1.0 color rgbf <1, 1, 1, 1>]
    }
  }
}
```

Now apply all of these to our sphere.

```
sphere { <0,0,0>, 1
  texture { LandArea }
  texture { OceanArea }
  texture { CloudArea }
}
```

We render this and have a pretty good rendition of a little planetoid. But it could be better. We don't particularly like the appearance of the clouds. There is a way they could be done that would be much more realistic.

2.7.1.5 Using Pigment Maps

Pigments may be blended together in the same way as the colors in a color map using the same pattern keywords and a **`pigment_map`**. Let's just give it a try.

We add the following declarations, making sure they appear before the other declarations in the file.

```
#declare Clouds1 = pigment {
    bozo
    turbulence 1
    color_map {
      [0.0 color White filter 1]
      [0.5 color White]
      [1.0 color White filter 1]
    }
  }
#declare Clouds2 = pigment {
  agate
  turbulence 1
  color_map {
    [0.0 color White filter 1]
```

```
    [0.5 color White]
    [1.0 color White filter 1]
    }
  }
#declare Clouds3 = pigment {
  marble
  turbulence 1
  color_map {
    [0.0 color White filter 1]
    [0.5 color White]
    [1.0 color White filter 1]
  }
}
#declare Clouds4 = pigment {
  granite
  turbulence 1
  color_map {
    [0.0 color White filter 1]
    [0.5 color White]
    [1.0 color White filter 1]
  }
}
```

Now we use these declared pigments in our cloud layer on our planetoid. We replace the declared cloud layer with.

```
#declare CloudArea = texture {
  pigment {
    gradient y
    pigment_map {
      [0.00 Clouds1]
      [0.25 Clouds2]
      [0.50 Clouds3]
      [0.75 Clouds4]
      [1.00 Clouds1]
    }
  }
}
```

We render this and see a remarkable pattern that looks very much like weather patterns on the planet earth. They are separated into bands, simulating the different weather types found at different latitudes.

2.7.2 Normals

Objects in POV-Ray have very smooth surfaces. This is not very realistic so there are several ways to disturb the smoothness of an object by perturbing the surface normal. The surface normal is the vector that is perpendicular to the angle of the surface. By changing this normal the surface can be made to appear bumpy, wrinkled or any of the many patterns available. Let's try a couple of them.

2.7.2.1 Using Basic Normal Modifiers

We comment out the planetoid sphere for now and, at the bottom of the file, create a new sphere with a simple, single color texture.

```
sphere { <0,0,0>, 1
  pigment { Gray75 }
  normal { bumps 1 scale .2 }
}
```

Here we have added a **normal** block in addition to the **pigment** block (note that these do not have to be included in a **texture** block unless they need to be transformed together or need to be part of a layered texture). We render this to see what it looks like. Now, one at a time, we substitute for the keyword **bumps** the following keywords: **dents**, **wrinkles**, **ripples** and **waves** (we can also use any of the patterns listed in "Patterns"). We render each to see what they look like. We play around with the float value that follows the keyword. We also experiment with the scale value.

For added interest, we change the plane texture to a single color with a normal as follows.

```
plane { y, -1.5
  pigment { color rgb <.65, .45, .35> }
  normal { dents .75 scale .25 }
}
```

2.7.2.2 Blending Normals

Normals can be layered similar to pigments but the results can be unexpected. Let's try that now by editing the sphere as follows.

```
sphere { <0,0,0>, 1
  pigment { Gray75 }
    normal { radial frequency 10 }
    normal { gradient y scale .2 }
}
```

As we can see, the resulting pattern is neither a radial nor a gradient. It is instead the result of first calculating a radial pattern and then calculating a gradient pattern. The results are simply additive. This can be difficult to control so POV-Ray gives the user other ways to blend normals.

One way is to use normal maps. A normal map works the same way as the pigment map we used earlier. Let's change our sphere texture as follows.

```
sphere { <0,0,0>, 1
  pigment { Gray75 }
  normal {
    gradient y
    frequency 3
    turbulence .5
    normal_map {
      [0.00 granite]
      [0.25 spotted turbulence .35]
      [0.50 marble turbulence .5]
      [0.75 bozo turbulence .25]
      [1.00 granite]
    }
  }
}
```

Rendering this we see that the sphere now has a very irregular bumpy surface. The gradient pattern type separates the normals into bands but they are turbulated, giving the surface a chaotic appearance. But this give us an idea.

Suppose we use the same pattern for a normal map that we used to create the oceans on our planetoid and applied it to the land areas. Does it follow that if we use the same pattern and modifiers on a sphere the same size that the shape of the pattern would be the same? Wouldn't that make the land areas bumpy while leaving the oceans smooth? Let's try it. First, let's render the two spheres side-by-side so we can see if the pattern is indeed the same. We un-comment the planetoid sphere and make the following changes.

```
sphere { <0,0,0>, 1
  texture { LandArea }
  texture { OceanArea }
```

```
    //texture { CloudArea }  // <-comment this out
    translate -x             // <- add this transformation
  }
```

Now we change the gray sphere as follows.

```
  sphere { <0,0,0>, 1
    pigment { Gray75 }
    normal {
      bozo
      turbulence .5
      lambda 2
      normal_map {
        [0.4 dents .15 scale .01]
        [0.6 agate turbulence 1]
        [1.0 dents .15 scale .01]
      }
    }
    translate x // <- add this transformation
  }
```

We render this to see if the pattern is the same. We see that indeed it is. So let's comment out the gray sphere and add the **`normal`** block it contains to the land area texture of our planetoid. We remove the transformations so that the planetoid is centered in the scene again.

```
  #declare LandArea = texture {
    pigment {
      agate
      turbulence 1
      lambda 1.5
      omega .8
      octaves 8
      color_map {
        [0.00 color rgb <.5, .25, .15>]
        [0.33 color rgb <.1, .5, .4>]
        [0.86 color rgb <.6, .3, .1>]
        [1.00 color rgb <.5, .25, .15>]
      }
    }
    normal {
      bozo
      turbulence .5
      lambda 2
      normal_map {
        [0.4 dents .15 scale .01]
        [0.6 agate turbulence 1]
        [1.0 dents .15 scale .01]
      }
    }
  }
```

Looking at the resulting image we see that indeed our idea works! The land areas are bumpy while the oceans are smooth. We add the cloud layer back in and our planetoid is complete.

There is much more that we did not cover here due to space constraints. On our own, we should take the time to explore slope maps, average and bump maps.

2.7.3 Finishes

The final part of a POV-Ray texture is the **finish**. It controls the properties of the surface of an object. It can make it shiny and reflective, or dull and flat. It can also specify what happens to light that passes through transparent pigments, what happens to light that is scattered by less-than-perfectly-smooth surfaces and what happens to light that is reflected by surfaces with thin-film interference properties. There are twelve different properties available in POV-Ray to specify the finish of a given object. These are controlled by the following keywords: **ambient**, **diffuse**, **brilliance**, **phong**, **specular**, **metallic**, **reflection**, **crand** and **iridescence**. Let's design a couple of textures that make use of these parameters.

2.7.3.1 Using Ambient

Since objects in POV-Ray are illuminated by light sources, the portions of those objects that are in shadow would be completely black were it not for the first two finish properties, **ambient** and **diffuse**. Ambient is used to simulate the light that is scattered around the scene that does not come directly from a light source. Diffuse determines how much of the light that is seen comes directly from a light source. These two keywords work together to control the simulation of ambient light. Let's use our gray sphere to demonstrate this. Let's also change our plane back to its original green and white checkered pattern.

```
plane {y,-1.5
  pigment {checker Green, White}
}
sphere { <0,0,0>, 1
  pigment {Gray75}
  finish {
    ambient .2
    diffuse .6
}
```

In the above example, the default values for ambient and diffuse are used. We render this to see what the effect is and then make the following change to the finish.

```
ambient 0
diffuse 0
```

The sphere is black because we have specified that none of the light coming from any light source will be reflected by the sphere. Let's change **diffuse** back to the default of 0.6.

Now we see the gray surface color where the light from the light source falls directly on the sphere but the shaded side is still absolutely black. Now let's change **diffuse** to 0.3 and **ambient** to 0.3.

The sphere now looks almost flat. This is because we have specified a fairly high degree of ambient light and only a low amount of the light coming from the light source is diffusely reflected towards the camera. The default values of **ambient** and **diffuse** are pretty good averages and a good starting point. In most cases, an ambient value of 0.1 ... 0.2 is sufficient and a diffuse value of 0.5 ... 0.7 will usually do the job. There are a couple of exceptions. If we have a completely transparent surface with high refractive and/or reflective values, low values of both ambient and diffuse may be best. Here is an example:

```
sphere { <0,0,0>, 1
   pigment { White filter 1 }
   finish {
      ambient 0
      diffuse 0
      reflection .25
      specular 1
      roughness .001
   }
   interior{ior 1.33}
```

```
}
```

This is glass, obviously. Glass is a material that takes nearly all of its appearance from its surroundings. Very little of the surface is seen because it transmits or reflects practically all of the light that shines on it. See `glass.inc` for some other examples.

If we ever need an object to be completely illuminated independently of the lighting situation in a given scene we can do this artificially by specifying an **`ambient`** value of 1 and a **`diffuse`** value of 0. This will eliminate all shading and simply give the object its fullest and brightest color value at all points. This is good for simulating objects that emit light like light bulbs and for skies in scenes where the sky may not be adequately lit by any other means.

Let's try this with our sphere now.

```
sphere { <0,0,0>, 1
   pigment { White }
   finish {
      ambient 1
      diffuse 0
   }
}
```

Rendering this we get a blinding white sphere with no visible highlights or shaded parts. It would make a pretty good streetlight.

2.7.3.2 Using Surface Highlights

In the glass example above, we noticed that there were bright little *hotspots* on the surface. This gave the sphere a hard, shiny appearance. POV-Ray gives us two ways to specify surface specular highlights. The first is called *Phong highlighting.* Usually, Phong highlights are described using two keywords: **`phong`** and **`phong_size`**. The float that follows **`phong`** determines the brightness of the highlight while the float following **`phong_size`** determines its size. Let's try this.

```
sphere { <0,0,0>, 1
  pigment { Gray50 }
  finish {
    ambient .2
    diffuse .6
    phong .75
    phong_size 25
  }
}
```

Rendering this we see a fairly broad, soft highlight that gives the sphere a kind of plastic appearance. Now let's change **`phong_size`** to 150. This makes a much smaller highlight which gives the sphere the appearance of being much harder and shinier.

There is another kind of highlight that is calculated by a different means called *specular highlighting*. It is specified using the keyword **`specular`** and operates in conjunction with another keyword called **`roughness`**. These two keywords work together in much the same way as **`phong`** and **`phong_size`** to create highlights that alter the apparent shininess of the surface. Let's try using specular in our sphere.

```
sphere { <0,0,0>, 1
   pigment { Gray50 }
   finish {
      ambient .2
      diffuse .6
      specular .75
      roughness .1
```

```
   }
}
```

Looking at the result we see a broad, soft highlight similar to what we had when we used **phong_size** of 25. Change **roughness** to .001 and render again. Now we see a small, tight highlight similar to what we had when we used **phong_size** of 150. Generally speaking, specular is slightly more accurate and therefore slightly more realistic than phong but you should try both methods when designing a texture. There are even times when both phong and specular may be used on a finish.

2.7.3.3 Using Reflection and Metallic

There is another surface parameter that goes hand in hand with highlights, **reflection**. Surfaces that are very shiny usually have a degree of reflection to them. Let's take a look at an example.

```
sphere { <0,0,0>, 1
   pigment { Gray50 }
   finish {
      ambient .2
      diffuse .6
      specular .75
      roughness .001
      reflection .5
   }
}
```

We see that our sphere now reflects the green and white checkered plane and the black background but the gray color of the sphere seems out of place. This is another time when a lower diffuse value is needed. Generally, the higher **reflection** is the lower **diffuse** should be. We lower the diffuse value to 0.3 and the ambient value to 0.1 and render again. That is much better. Let's make our sphere as shiny as a polished gold ball bearing.

```
sphere { <0,0,0>, 1
   pigment { BrightGold }
   finish {
      ambient .1
      diffuse .1
      specular 1
      roughness .001
      reflection .75
   }
 }
```

That is very close but there is something wrong with the highlight. To make the surface appear more like metal the keyword **metallic** is used. We add it now to see the difference.

```
sphere { <0,0,0>, 1
   pigment { BrightGold }
   finish {
      ambient .1
      diffuse .1
      specular 1
      roughness .001
      reflection .75
      metallic
   }
}
```

We see that the highlight has taken on the color of the surface rather than the light source. This gives the surface a more metallic appearance.

2.7.3.4 Using Iridescence

Iridescence is what we see on the surface of an oil slick when the sun shines on it. The rainbow effect is created by something called *thin-film interference* (read section "Iridescence" for details). For now let's just try using it. Iridescence is specified by the **irid** statement and three values: amount, **thickness** and **turbulence**. The amount is the contribution to the overall surface color. Usually 0.1 to 0.5 is sufficient here. The thickness affects the "busyness" of the effect. Keep this between 0.25 and 1 for best results. The turbulence is a little different from pigment or normal turbulence. We cannot set **octaves**, **lambda** or **omega** but we can specify an amount which will affect the thickness in a slightly different way from the thickness value. Values between 0.25 and 1 work best here too. Finally, iridescence will respond to the surface normal since it depends on the angle of incidence of the light rays striking the surface. With all of this in mind, let's add some iridescence to our glass sphere.

```
sphere { <0,0,0>, 1
     pigment { White filter 1 }
     finish {
        ambient .1
        diffuse .1
        reflection .2
        specular 1
        roughness .001
        irid {
          0.35
          thickness .5
          turbulence .5
        }
     }
     interior{
        ior 1.5
        fade_distance 5
        fade_power 1
        caustics 1
     }
}
```

We try to vary the values for amount, thickness and turbulence to see what changes they make. We also try to add a **normal** block to see what happens.

2.7.4 Working With Pigment Maps

Let's look at the pigment map. We must not confuse this with a color map, as color maps can only take individual colors as entries in the map, while pigment maps can use entire other pigment patterns. To get a feel for these, let's begin by setting up a basic plane with a simple pigment map. Now, in the following example, we are going to declare each of the pigments we are going to use before we actually use them. This isn't strictly necessary (we could put an entire pigment description in each entry of the map) but it just makes the whole thing more readable.

```
  // simple Black on White checkboard... it's a classic
  #declare Pigment1 = pigment {
    checker color Black color White
    scale .1
  }
  // kind of a "psychedelic rings" effect
  #declare Pigment2 = pigment {
    wood
    color_map {
      [ 0.0 Red ]
      [ 0.3 Yellow ]
      [ 0.6 Green ]
      [ 1.0 Blue ]
```

```
    }
  }
  plane { -z, 0
    pigment {
      gradient x
      pigment_map {
        [ 0.0 Pigment1 ]
        [ 0.5 Pigment2 ]
        [ 1.0 Pigment1 ]
      }
    }
  }
```

Okay, what we have done here is very simple, and probably quite recognizable if we have been working with color maps all along anyway. All we have done is substituted a pigment map where a color map would normally go, and as the entries in our map, we have referenced our declared pigments. When we render this example, we see a pattern which fades back and forth between the classic checkerboard, and those colorful rings. Because we fade from Pigment1 to Pigment2 and then back again, we see a clear blending of the two patterns at the transition points. We could just as easily get a sudden transition by amending the map to read.

```
  pigment_map {
    [ 0.0 Pigment1 ]
    [ 0.5 Pigment1 ]
    [ 0.5 Pigment2 ]
    [ 1.0 Pigment2 ]
  }
```

Blending individual pigment patterns is just the beginning.

2.7.5 Working With Normal Maps

For our next example, we replace the plane in the scene with this one.

```
  plane { -z, 0
    pigment { White }
    normal {
      gradient x
      normal_map {
        [ 0.0 bumps 1 scale .1]
        [ 1.0 ripples 1 scale .1]
      }
    }
  }
```

First of all, we have chosen a solid white color to show off all bumping to best effect. Secondly, we notice that our map blends smoothly from all bumps at 0.0 to all ripples at 1.0, but because this is a default gradient, it falls off abruptly back to bumps at the beginning of the next cycle. We Render this and see just enough sharp transitions to clearly see where one normal gives over to another, yet also an example of how two normal patterns look while they are smoothly blending into one another.

The syntax is the same as we would expect. We just changed the type of map, moved it into the normal block and supplied appropriate bump types. It is important to remember that as of POV-Ray 3, all patterns that work with pigments work as normals as well (and vice versa, of course) so we could just as easily have blended from wood to granite, or any other pattern we like. We experiment a bit and get a feel for what the different patterns look like.

After seeing how interesting the various normals look blended, we might like to see them completely blended all the way through rather than this business of fading from one to the next. Well, that is possible too, but we would be getting ahead of ourselves. That is called the **`average`** function, and we will return to it a little bit further down the page.

2.7.6 Working With Texture Maps

We know how to blend colors, pigment patterns, and normals, and we are probably thinking what about finishes? What about whole textures? Both of these can be kind of covered under one topic. While there is no finish map per se, there are texture maps, and we can easily adapt these to serve as finish maps, simply by putting the same pigment and/or normal in each of the texture entries of the map. Here is an example. We eliminate the declared pigments we used before and the previous plane, and add the following.

```
#declare Texture1 = texture {
  pigment { Grey }
  finish { reflection 1 }
}
#declare Texture2 = texture {
  pigment { Grey }
  finish { reflection 0 }
}
cylinder { <-2, 5, -2>, <-2, -5, -2>, 1
  pigment { Blue }
}
plane { -z, 0
  rotate y * 30
  texture {
    gradient y
    texture_map {
      [ 0.0 Texture1 ]
      [ 0.4 Texture1 ]
      [ 0.6 Texture2 ]
      [ 1.0 Texture2 ]
    }
    scale 2
  }
}
```

Now, what have we done here? The background plane alternates vertically between two textures, identical except for their finishes. When we render this, the cylinder has a reflection part of the way down the plane, and then stops reflecting, then begins and then stops again, in a gradient pattern down the surface of the plane. With a little adaptation, this could be used with any pattern, and in any number of creative ways, whether we just wanted to give various parts of an object different finishes, as we are doing here, or whole different textures altogether.

One might ask: if there is a texture map, why do we need pigment and normal maps? Fair question. The answer: speed of calculation. If we use a texture map, for every in-between point, POV-Ray must make multiple calculations for each texture element, and then run a weighted average to produce the correct value for that point. Using just a pigment map (or just a normal map) decreases the overall number of calculations, and our texture renders a bit faster in the bargain. As a rule of thumb: we use pigment or normal maps where we can and only fall back on texture maps if we need the extra flexibility.

2.7.7 Working With List Textures

If we have followed the corresponding tutorials on simple pigments, we know that there are three patterns called *color list* patterns, because rather than using a color map, these simple but useful patterns take a list of colors immediately following the pattern keyword. We're talking about checker, hexagon, and, new to POV-Ray 3, the brick pattern.

Naturally they also work with whole pigments, normals, and entire textures, just as the other patterns do above. The only difference is that we list entries in the pattern (as we would do with individual colors) rather than using a map of entries. Here is an example. We strike the plane and any declared pigments we had left over in our last example, and add the following to our basic file.

```
#declare Pigment1 = pigment {
  hexagon
  color Yellow color Green color Grey
  scale .1
}
#declare Pigment2 = pigment {
  checker
  color Red color Blue
  scale .1
}
#declare Pigment3 = pigment {
  brick
  color White color Black
  rotate -90*x
  scale .1
}
box { -5, 5
  pigment {
    hexagon
    pigment {Pigment1}
    pigment {Pigment2}
    pigment {Pigment3}
    rotate 90*x
  }
}
```

We begin by declaring an example of each of the color list patterns as individual pigments. Then we use the hexagon pattern as a *pigment list* pattern, simply feeding it a list of pigments rather than colors as we did above. There are two rotate statements throughout this example, because bricks are aligned along the z-direction, while hexagons align along the y-direction, and we wanted everything to face toward the camera we originally declared out in the -z-direction so we can really see the patterns within patterns effect here.

Of course color list patterns used to be only for pigments, but as of POV-Ray 3, everything that worked for pigments can now also be adapted for normals or entire textures. A couple of quick examples might look like

```
normal {
  brick
  normal { granite .1 }
  normal { bumps 1 scale .1 }
}
```

or...

```
texture {
  checker
  texture { Gold_Metal }
  texture { Silver_Metal }
}
```

2.7.8 What About Tiles?

In earlier versions of POV-Ray, there was a texture pattern called **`tiles`**. By simply using a checker texture pattern (as we just saw above), we can achieve the same thing as tiles used to do, so it is now obsolete. It is still supported by POV-Ray 3 for backwards compatibility with old scene files, but now is a good time to get in the habit of using a checker pattern instead.

2.7.9 Average Function

Now things get interesting. Above, we began to see how pigments and normals can fade from one to the other when we used them in maps. But how about if we want a smooth blend of patterns all the way through? That is where a new feature called **average** can come in very handy. Average works with pigment, normal, and texture maps, although the syntax is a little bit different, and when we are not expecting it, the change can be confusing. Here is a simple example. We use our standard includes, camera and light source from above, and enter the following object.

```
plane { -z, 0
  pigment { White }
  normal {
    average
    normal_map {
      [ gradient x ]
      [ gradient y ]
    }
  }
}
```

What we have done here is pretty self explanatory as soon as we render it. We have combined a vertical with a horizontal gradient bump pattern, creating crisscrossing gradients. Actually, the crisscrossing effect is a smooth blend of gradient x with gradient y all the way across our plane. Now, what about that syntax difference?

We see how our normal map has changed from earlier examples. The floating point value to the left-hand side of each map entry has been removed. That value usually helps in procedurally mapping each entry to the pattern we have selected, but average is a smooth blend all the way through, not a pattern, so it cannot use those values. In fact, including them may sometimes lead to unexpected results, such as entries being lost or misrepresented in some way. To ensure that we'll get the pattern blend we anticipate, we leave off the floating point value.

2.7.10 Working With Layered Textures

With the multitudinous colors, patterns, and options for creating complex textures in POV-Ray, we can easily become deeply engrossed in mixing and tweaking just the right textures to apply to our latest creations. But as we go, sooner or later there is going to come that *special* texture. That texture that is sort of like wood, only varnished, and with a kind of spotty yellow streaking, and some vertical gray flecks, that looks like someone started painting over it all, and then stopped, leaving part of the wood visible through the paint.

Only... now what? How do we get all that into one texture? No pattern can do that many things. Before we panic and say image map there is at least one more option: *layered textures.*

With layered textures, we only need to specify a series of textures, one after the other, all associated with the same object. Each texture we list will be applied one on top of the other, from bottom to top in the order they appear.

It is very important to note that we must have some degree of transparency (filter or transmit) in the pigments of our upper textures, or the ones below will get lost underneath. We won't receive a warning or an error - technically it is legal to do this: it just doesn't make sense. It is like spending hours sketching an elaborate image on a bare wall, then slapping a solid white coat of latex paint over it.

Let's design a very simple object with a layered texture, and look at how it works. We create a file called LAYTEX.POV and add the following lines.

```
#include "colors.inc"
#include "textures.inc"
camera {
  location <0, 5, -30>
  look_at <0, 0, 0>
}
light_source { <-20, 30, -50> color White }
```

```
plane { y, 0 pigment { checker color Green color Yellow  } }
background { rgb <.7, .7, 1> }
box { <-10, 0, -10>, <10, 10, 10>
  texture {
    Silver_Metal // a metal object ...
    normal {     // ... which has suffered a beating
      dents 2
      scale 1.5
    }
  } // (end of base texture)
  texture { // ... has some flecks of rust ...
    pigment {
      granite
      color_map {
        [0.0 rgb <.2, 0, 0> ]
        [0.2 color Brown ]
        [0.2 rgbt <1, 1, 1, 1> ]
        [1.0 rgbt <1, 1, 1, 1> ]
      }
      frequency 16
    }
  } // (end rust fleck texture)
  texture { // ... and some sooty black marks
    pigment {
      bozo
      color_map {
        [0.0 color Black ]
        [0.2 color rgbt <0, 0, 0, .5> ]
        [0.4 color rgbt <.5, .5, .5, .5> ]
        [0.5 color rgbt <1, 1, 1, 1> ]
        [1.0 color rgbt <1, 1, 1, 1> ]
      }
      scale 3
    }
  } // (end of sooty mark texture)
} // (end of box declaration)
```

Whew. This gets complicated, so to make it easier to read, we have included comments showing what we are doing and where various parts of the declaration end (so we don't get lost in all those closing brackets!). To begin, we created a simple box over the classic checkerboard floor, and give the background sky a pale blue color. Now for the fun part...

To begin with we made the box use the **`Silver_Metal`** texture as declared in textures.inc (for bonus points, look up `textures.inc` and see how this standard texture was originally created sometime). To give it the start of its abused state, we added the dents normal pattern, which creates the illusion of some denting in the surface as if our mysterious metal box had been knocked around quite a bit.

The flecks of rust are nothing but a fine grain granite pattern fading from dark red to brown which then abruptly drops to fully transparent for the majority of the color map. True, we could probably come up with a more realistic pattern of rust using pigment maps to cluster rusty spots, but pigment maps are a subject for another tutorial section, so let's skip that just now.

Lastly, we have added a third texture to the pot. The randomly shifting **`bozo`** texture gradually fades from blackened centers to semi-transparent medium gray, and then ultimately to fully transparent for the latter half of its color map. This gives us a look of sooty burn marks further marring the surface of the metal box. The final result leaves our mysterious metal box looking truly abused, using multiple texture patterns, one on top of the other, to produce an effect that no single pattern could generate!

2.7.10.1 Declaring Layered Textures

In the event we want to reuse a layered texture on several objects in our scene, it is perfectly legal to declare a layered texture. We won't repeat the whole texture from above, but the general format would be something like this:

```
#declare Abused_Metal =
  texture { /* insert your base texture here... */ }
  texture { /* and your rust flecks here... */ }
  texture { /* and of course, your sooty burn marks here */ }
```

POV-Ray has no problem spotting where the declaration ends, because the textures follow one after the other with no objects or directives in between. The layered texture to be declared will be assumed to continue until it finds something other than another texture, so any number of layers can be added in to a declaration in this fashion.

One final word about layered textures: whatever layered texture we create, whether declared or not, we must not leave off the texture wrapper. In conventional single textures a common shorthand is to have just a pigment, or just a pigment and finish, or just a normal, or whatever, and leave them outside of a texture statement. This shorthand does not extend to layered textures. As far as POV-Ray is concerned we can layer entire textures, but not individual pieces of textures. For example

```
#declare Bad_Texture =
  texture { /* insert your base texture here... */ }
  pigment { Red filter .5 }
  normal { bumps 1 }
```

will not work. The pigment and the normal are just floating there without being part of any particular texture. Inside an object, with just a single texture, we can do this sort of thing, but with layered textures, we would just generate an error whether inside the object or in a declaration.

2.7.10.2 Another Layered Textures Example

To further explain how layered textures work another example is described in detail. A tablecloth is created to be used in a picnic scene. Since a simple red and white checkered cloth looks entirely too new, too flat, and too much like a tiled floor, layered textures are used to stain the cloth.

We're going to create a scene containing four boxes. The first box has that plain red and white texture we started with in our picnic scene, the second adds a layer meant to realistically fade the cloth, the third adds some wine stains, and the final box adds a few wrinkles (not another layer, but we must note when and where adding changes to the surface normal have an effect in layered textures).

We start by placing a camera, some lights, and the first box. At this stage, the texture is plain tiling, not layered. See file `layered1.pov`.

```
#include "colors.inc"
camera {
  location <0, 0, -6>
  look_at <0, 0, 0>
}
light_source { <-20, 30, -100> color White }
light_source { <10, 30, -10> color White }
light_source { <0, 30, 10> color White }
#declare PLAIN_TEXTURE =
  // red/white check
  texture {
    pigment {
      checker
      color rgb<1.000, 0.000, 0.000>
      color rgb<1.000, 1.000, 1.000>
      scale <0.2500, 0.2500, 0.2500>
```

```
    }
  }
// plain red/white check box
box { <-1, -1, -1>, <1, 1, 1>
  texture {
    PLAIN_TEXTURE
  }
  translate  <-1.5, 1.2, 0>
}
```

We render this scene. It is not particularly interesting, isn't it? That is why we will use some layered textures to make it more interesting.

First, we add a layer of two different, partially transparent greys. We tile them as we had tiled the red and white colors, but we add some turbulence to make the fading more realistic. We add following box to the previous scene and re-render (see file `layered2.pov`).

```
#declare FADED_TEXTURE =
  // red/white check texture
  texture {
    pigment {
      checker
      color rgb<0.920, 0.000, 0.000>
      color rgb<1.000, 1.000, 1.000>
      scale <0.2500, 0.2500, 0.2500>
    }
  }
  // greys to fade red/white
  texture {
    pigment {
      checker
      color rgbf<0.632, 0.612, 0.688, 0.698>
      color rgbf<0.420, 0.459, 0.520, 0.953>
      turbulence 0.500
      scale <0.2500, 0.2500, 0.2500>
    }
  }
// faded red/white check box
box { <-1, -1, -1>, <1, 1, 1>
  texture {
    FADED_TEXTURE
  }
  translate  <1.5, 1.2, 0>
}
```

Even though it is a subtle difference, the red and white checks no longer look quite so new.

Since there is a bottle of wine in the picnic scene, we thought it might be a nice touch to add a stain or two. While this effect can almost be achieved by placing a flattened blob on the cloth, what we really end up with is a spill effect, not a stain. Thus it is time to add another layer.

Again, we add another box to the scene we already have scripted and re-render (see file `layered3.pov`).

```
#declare STAINED_TEXTURE =
  // red/white check
  texture {
    pigment {
      checker
      color rgb<0.920, 0.000, 0.000>
```

```
      color rgb<1.000, 1.000, 1.000>
      scale <0.2500, 0.2500, 0.2500>
    }
  }
  // greys to fade check
  texture {
    pigment {
      checker
      color rgbf<0.634, 0.612, 0.688, 0.698>
      color rgbf<0.421, 0.463, 0.518, 0.953>
      turbulence 0.500
      scale <0.2500, 0.2500, 0.2500>
    }
  }
  // wine stain
  texture {
    pigment {
      spotted
      color_map {
        [ 0.000  color rgb<0.483, 0.165, 0.165> ]
        [ 0.329  color rgbf<1.000, 1.000, 1.000, 1.000> ]
        [ 0.734  color rgbf<1.000, 1.000, 1.000, 1.000> ]
        [ 1.000  color rgb<0.483, 0.165, 0.165> ]
      }
      turbulence 0.500
      frequency 1.500
    }
  }
// stained box
box { <-1, -1, -1>, <1, 1, 1>
  texture {
    STAINED_TEXTURE
  }
  translate  <-1.5, -1.2, 0>
}
```

Now there's a tablecloth texture with personality.

Another touch we want to add to the cloth are some wrinkles as if the cloth had been rumpled. This is not another texture layer, but when working with layered textures, we must keep in mind that changes to the surface normal must be included in the uppermost layer of the texture. Changes to lower layers have no effect on the final product (no matter how transparent the upper layers are).

We add this final box to the script and re-render (see file `layered4.pov`)

```
#declare WRINKLED_TEXTURE =
  // red and white check
  texture {
    pigment {
      checker
      color rgb<0.920, 0.000, 0.000>
      color rgb<1.000, 1.000, 1.000>
      scale <0.2500, 0.2500, 0.2500>
    }
  }
  // greys to "fade" checks
  texture {
    pigment {
```

```
      checker
      color rgbf<0.632, 0.612, 0.688, 0.698>
      color rgbf<0.420, 0.459, 0.520, 0.953>
      turbulence 0.500
      scale <0.2500, 0.2500, 0.2500>
    }
  }
  // the wine stains
  texture {
    pigment {
      spotted
      color_map {
        [ 0.000  color rgb<0.483, 0.165, 0.165> ]
        [ 0.329  color rgbf<1.000, 1.000, 1.000, 1.000> ]
        [ 0.734  color rgbf<1.000, 1.000, 1.000, 1.000> ]
        [ 1.000  color rgb<0.483, 0.165, 0.165> ]
      }
      turbulence 0.500
      frequency 1.500
    }
    normal {
      wrinkles 5.0000
    }
  }
// wrinkled box
box { <-1, -1, -1>, <1, 1, 1>
  texture {
    WRINKLED_TEXTURE
  }
  translate  <1.5, -1.2, 0>
}
```

Well, this may not be the tablecloth we want at any picnic we're attending, but if we compare the final box to the first, we see just how much depth, dimension, and personality is possible just by the use of creative texturing.

One final note: the comments concerning the surface normal do not hold true for finishes. If a *lower* layer contains a specular finish and an *upper* layer does not, any place where the upper layer is transparent, the specular will show through.

2.7.11 When All Else Fails: Material Maps

We have some pretty powerful texturing tools at our disposal, but what if we want a more free form arrangement of complex textures? Well, just as image maps do for pigments, and bump maps do for normals, whole textures can be mapped using a material map, should the need arise.

Just as with image maps and bump maps, we need a source image in bitmapped format which will be called by POV-Ray to serve as the map of where the individual textures will go, but this time, we need to specify what texture will be associated with which palette index. To make such an image, we can use a paint program which allows us to select colors by their palette index number (the actual color is irrelevant, since it is only a map to tell POV-Ray what texture will go at that location). Now, if we have the complete package that comes with POV-Ray, we have in our include files an image called `povmap.gif` which is a bitmapped image that uses only the first four palette indices to create a bordered square with the words "Persistence of Vision" in it. This will do just fine as a sample map for the following example. Using our same include files, the camera and light source, we enter the following object.

```
plane { -z, 0
  texture {
    material_map {
```

```
      gif "povmap.gif"
      interpolate 2
      once
      texture { PinkAlabaster }          // the inner border
      texture { pigment { DMFDarkOak } } // outer border
      texture { Gold_Metal }             // lettering
      texture { Chrome_Metal }           // the window panel
    }
    translate <-0.5, -0.5, 0>
    scale 5
  }
}
```

The position of the light source and the lack of foreground objects to be reflected do not show these textures off to their best advantage. But at least we can see how the process works. The textures have simply been placed according to the location of pixels of a particular palette index. By using the **`once`** keyword (to keep it from tiling), and translating and scaling our map to match the camera we have been using, we get to see the whole thing laid out for us.

Of course, that is just with palette mapped image formats, such as GIF and certain flavors of PNG. Material maps can also use non-paletted formats, such as the TGA files that POV-Ray itself outputs. That leads to an interesting consequence: We can use POV-Ray to produce source maps for POV-Ray! Before we wrap up with some of the limitations of special textures, let's do one more thing with material maps, to show how POV-Ray can make its own source maps.

To begin with, if using an non-paletted image, POV-Ray looks at the 8 bit red component of the pixel's color (which will be a value from 0 to 255) to determine which texture from the list to use. So to create a source map, we need to control very precisely what the red value of a given pixel will be. We can do this by

1.) Using an rgb statement to choose our color such as **`rgb <N/255,0,0>`**, where "N" is the red value we want to assign that pigment, and then...
2.) Use no light sources and apply a finish of **`finish{ambient 1}`** to all objects, to ensure that highlighting and shadowing will not interfere.

Confused? Alright, here is an example, which will generate a map very much like `povmap.gif` which we used earlier, except in TGA file format. We notice that we have given the pigments blue and green components too. POV-Ray will ignore that in our final map, so this is really for us humans, whose unaided eyes cannot tell the difference between red variances of 0 to 4/255ths. Without those blue and green variances, our map would look to our eyes like a solid black screen. That may be a great way to send secret messages using POV-Ray (plug it into a material map to decode) but it is no use if we want to see what our source map looks like to make sure we have what we expected to.

We render the following code, and name the resulting file `povmap.tga`.

```
camera {
  orthographic
  up <0, 5, 0>
  right <5, 0, 0>
  location <0, 0, -25>
  look_at <0, 0, 0>
}
plane { -z, 0
  pigment { rgb <1/255, 0, 0.5> }
  finish { ambient 1 }
}
box { <-2.3, -1.8, -0.2>, <2.3, 1.8, -0.2>
  pigment { rgb <0/255, 0, 1> }
  finish { ambient 1 }
}
```

```
box { <-1.95, -1.3, -0.4>, <1.95, 1.3, -0.3>
  pigment { rgb <2/255, 0.5, 0.5> }
  finish { ambient 1 }
}
text { ttf "crystal.ttf", "The vision", 0.1, 0
  scale <0.7, 1, 1>
  translate <-1.8, 0.25, -0.5>
  pigment { rgb <3/255, 1, 1> }
  finish { ambient 1 }
}
text { ttf "crystal.ttf", "Persists!", 0.1, 0
  scale <0.7, 1, 1>
  translate <-1.5, -1, -0.5>
  pigment { rgb <3/255, 1, 1> }
  finish { ambient 1 }
}
```

All we have to do is modify our last material map example by changing the material map from GIF to TGA and modifying the filename. When we render using the new map, the result is extremely similar to the palette mapped GIF we used before, except that we didn't have to use an external paint program to generate our source: POV-Ray did it all!

2.7.12 Limitations Of Special Textures

There are a couple limitations to all of the special textures we have seen (from textures, pigment and normal maps through material maps). First, if we have used the default directive to set the default texture for all items in our scene, it will not accept any of the special textures discussed here. This is really quite minor, since we can always declare such a texture and apply it individually to all objects. It doesn't actually prevent us from doing anything we couldn't otherwise do.

The other is more limiting, but as we will shortly see, can be worked around quite easily. If we have worked with layered textures, we have already seen how we can pile multiple texture patterns on top of one another (as long as one texture has transparency in it). This very useful technique has a problem incorporating the special textures we have just seen as a layer. But there is an answer!

For example, say we have a layered texture called **`Speckled_Metal`**, which produces a silver metallic surface, and then puts tiny specks of rust all over it. Then we decide, for a really rusty look, we want to create patches of concentrated rust, randomly over the surface. The obvious approach is to create a special texture pattern, with transparency to use as the top layer. But of course, as we have seen, we wouldn't be able to use that texture pattern as a layer. We would just generate an error message. The solution is to turn the problem inside out, and make our layered texture part of the texture pattern instead, like this

```
// This part declares a pigment for use
// in the rust patch texture pattern
#declare Rusty = pigment {
  granite
  color_map {
    [ 0 rgb <0.2, 0, 0> ]
    [ 1 Brown ]
  }
  frequency 20
}
// And this part applies it
// Notice that our original layered texture
// "Speckled_Metal" is now part of the map
#declare Rust_Patches = texture {
  bozo
```

```
    texture_map {
      [ 0.0  pigment {Rusty} ]
      [ 0.75 Speckled_Metal ]
      [ 1.0  Speckled_Metal ]
    }
  }
```

And the ultimate effect is the same as if we had layered the rust patches on to the speckled metal anyway.

With the full array of patterns, pigments, normals, finishes, layered and special textures, there is now practically nothing we cannot create in the way of amazing textures. An almost infinite number of new possibilities are just waiting to be created!

2.8 Using the Camera

2.8.1 Using Focal Blur

Let's construct a simple scene to illustrate the use of focal blur. For this example we will use a pink sphere, a green box and a blue cylinder with the sphere placed in the foreground, the box in the center and the cylinder in the background. A checkered floor for perspective and a couple of light sources will complete the scene. We create a new file called `focaldem.pov` and enter the following text

```
#include "colors.inc"
#include "shapes.inc"
#include "textures.inc"
sphere { <1, 0, -6>, 0.5
  finish {
    ambient 0.1
    diffuse 0.6
  }
  pigment { NeonPink }
}
box { <-1, -1, -1>, < 1,  1,  1>
  rotate <0, -20, 0>
  finish {
    ambient 0.1
    diffuse 0.6
  }
  pigment { Green }
}
cylinder { <-6, 6, 30>, <-6, -1, 30>, 3
  finish {
    ambient 0.1
    diffuse 0.6
  }
  pigment {NeonBlue}
}
plane { y, -1.0
  pigment {
    checker color Gray65 color Gray30
  }
}
light_source { <5, 30, -30> color White }
light_source { <-5, 30, -30> color White }
```

Now we can proceed to place our focal blur camera to an appropriate viewing position. Straight back from our three objects will yield a nice view. Adjusting the focal point will move the point of focus anywhere in the scene. We just add the following lines to the file:

```
camera {
  location <0.0, 1.0, -10.0>
  look_at  <0.0, 1.0,  0.0>
//  focal_point <-6, 1, 30>    // blue cylinder in focus
//  focal_point < 0, 1,  0>    // green box in focus
  focal_point < 1, 1, -6>    // pink sphere in focus
  aperture 0.4     // a nice compromise
//  aperture 0.05    // almost everything is in focus
//  aperture 1.5     // much blurring
//  blur_samples 4       // fewer samples, faster to render
  blur_samples 20      // more samples, higher quality image
}
```

The focal point is simply the point at which the focus of the camera is at its sharpest. We position this point in our scene and assign a value to the aperture to adjust how close or how far away we want the focal blur to occur from the focused area.

The aperture setting can be considered an *area of focus*. Opening up the aperture has the effect of making the area of focus smaller while giving the aperture a smaller value makes the area of focus larger. This is how we control where focal blur begins to occur around the focal point.

The blur samples setting determines how many rays are used to sample each pixel. Basically, the more rays that are used the higher the quality of the resultant image, but consequently the longer it takes to render. Each scene is different so we have to experiment. This tutorial has examples of 4 and 20 samples but we can use more for high resolution images. We should not use more samples than is necessary to achieve the desired quality - more samples take more time to render. The confidence and variance settings are covered in section "Focal Blur".

We experiment with the focal point, aperture, and blur sample settings. The scene has lines with other values that we can try by commenting out the default line with double slash marks and un-commenting the line we wish to try out. We make only one change at a time to see the effect on the scene.

Two final points when tracing a scene using a focal blur camera. We needn't specify anti-aliasing because the focal blur code uses its one sampling method that automatically takes care of anti-aliasing. Focal blur can only be used with the perspective camera.

2.9 Using Atmospheric Effects

POV-Ray offers a variety of atmospheric effects, i. e. features that affect the background of the scene or the air by which everything is surrounded.

It is easy to assign a simple color or a complex color pattern to a virtual sky sphere. You can create anything from a cloud free, blue summer sky to a stormy, heavy clouded sky. Even starfields can easily be created.

You can use different kinds of fog to create foggy scenes. Multiple fog layers of different colors can add an eerie touch to your scene.

A much more realistic effect can be created by using an atmosphere, a constant fog that interacts with the light coming from light sources. Beams of light become visible and objects will cast shadows into the fog.

Last but not least you can add a rainbow to your scene.

2.9.1 The Background

The **background** feature is used to assign a color to all rays that don't hit any object. This is done in the following way.

```
camera {
  location <0, 0, -10>
  look_at <0, 0, 0>
}
background { color rgb <0.2, 0.2, 0.3> }
sphere { 0, 1
  pigment { color rgb <0.8, 0.5, 0.2> }
}
```

The background color will be visible if a sky sphere is used and if some translucency remains after all sky sphere pigment layers are processed.

2.9.2 The Sky Sphere

The **sky_sphere** can be used to easily create a cloud covered sky, a nightly star sky or whatever sky you have in mind.

In the following examples we'll start with a very simple sky sphere that will get more and more complex as we add new features to it.

2.9.2.1 Creating a Sky with a Color Gradient

Beside the single color sky sphere that is covered with the background feature the simplest sky sphere is a color gradient.

You may have noticed that the color of the sky varies with the angle to the earth's surface normal. If you look straight up the sky normally has a much deeper blue than it has at the horizon.

We want to model this effect using the sky sphere as shown in the scene below (`skysph1.pov`).

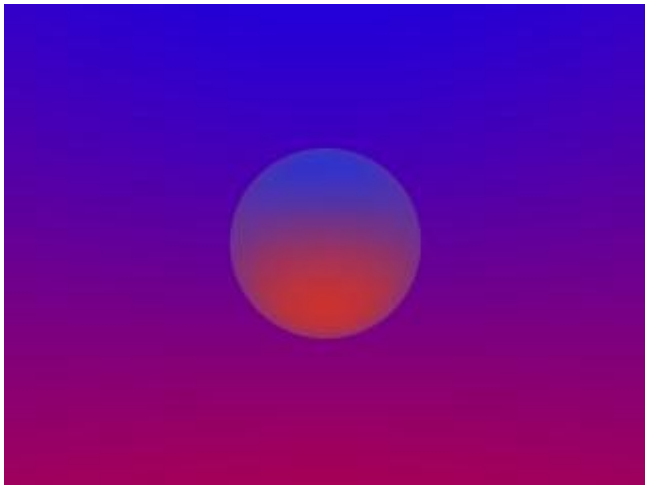
```
#include "colors.inc"
camera {
  location <0, 1, -4>
  look_at <0, 2, 0>
  angle 80
}
light_source { <10, 10, -10> White }
sphere { 2*y, 1
  pigment { color rgb <1, 1, 1> }
  finish { ambient 0.2 diffuse 0 reflection 0.6 }
}
sky_sphere {
  pigment {
    gradient y
    color_map {
      [0 color Red]
      [1 color Blue]
    }
    scale 2
    translate -1
  }
}
```

The interesting part is the sky sphere statement. It contains a pigment that describe the look of the sky sphere. We want to create a color gradient along the viewing angle measured against the earth's surface normal. Since the ray direction vector is used to calculate the pigment colors we have to use the y-gradient.

The scale and translate transformation are used to map the points derived from the direction vector to the right range. Without those transformations the pattern would be repeated twice on the sky sphere. The **`scale`** statement is used to avoid the repetition and the **`translate -1`** statement moves the color at index zero to the bottom of the sky sphere (that's the point of the sky sphere you'll see if you look straight down).

After this transformation the color entry at position 0 will be at the bottom of the sky sphere, i. e. below us, and the color at position 1 will be at the top, i. e. above us.

The colors for all other positions are interpolated between those two colors as you can see in the resulting image.



A simple gradient sky sphere.

If you want to start one of the colors at a specific angle you'll first have to convert the angle to a color map index. This is done by using the formula

color_map_index = (1 - cos(angle)) / 2

where the angle is measured against the negated earth's surface normal. This is the surface normal pointing towards the center of the earth. An angle of 0 degrees describes the point below us while an angle of 180 degrees represents the zenith.

In POV-Ray you first have to convert the degree value to radian values as it is shown in the following example.

```
sky_sphere {
  pigment {
    gradient y
    color_map {
      [(1-cos(radians( 30)))/2 color Red]
      [(1-cos(radians(120)))/2 color Blue]
    }
    scale 2
    translate -1
  }
}
```

This scene uses a color gradient that starts with a red color at 30 degrees and blends into the blue color at 120 degrees. Below 30 degrees everything is red while above 120 degrees all is blue.

2.9.2.2 Adding the Sun

In the following example we will create a sky with a red sun surrounded by a red color halo that blends into the dark blue night sky. We'll do this using only the sky sphere feature.

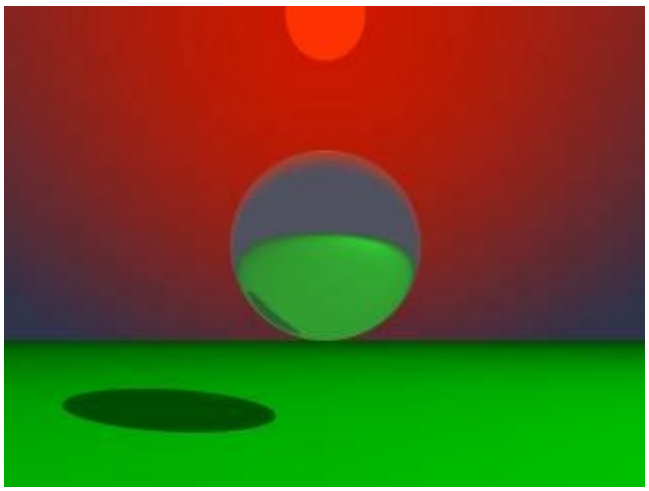
The sky sphere we use is shown below. A ground plane is also added for greater realism (`skysph2.pov`).

```
sky_sphere {
  pigment {
    gradient y
    color_map {
      [0.000 0.002 color rgb <1.0, 0.2, 0.0>
                   color rgb <1.0, 0.2, 0.0>]
      [0.002 0.200 color rgb <0.8, 0.1, 0.0>
                   color rgb <0.2, 0.2, 0.3>]
    }
    scale 2
    translate -1
  }
  rotate -135*x
}
plane { y, 0
  pigment { color Green }
  finish { ambient .3 diffuse .7 }
}
```

The gradient pattern and the transformation inside the pigment are the same as in the example in the previous section.

The color map consists of three colors. A bright, slightly yellowish red that is used for the sun, a darker red for the halo and a dark blue for the night sky. The sun's color covers only a very small portion of the sky sphere because we don't want the sun to become too big. The color is used at the color map values 0.000 and 0.002 to get a sharp contrast at value 0.002 (we don't want the sun to blend into the sky). The darker red color used for the halo blends into the dark blue sky color from value 0.002 to 0.200. All values above 0.200 will reveal the dark blue sky.

The **`rotate -135*x`** statement is used to rotate the sun and the complete sky sphere to its final position. Without this rotation the sun would be at 0 degrees, i.e. right below us.



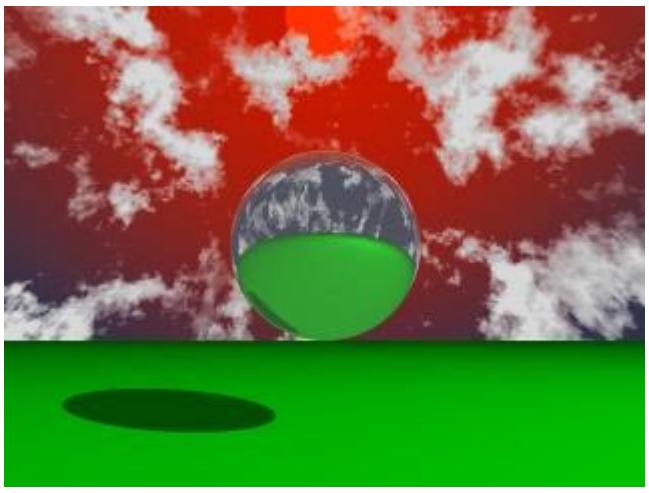
A red sun descends into the night.

Looking at the resulting image you'll see what impressive effects you can achieve with the sky sphere.

2.9.2.3 Adding Some Clouds

To further improve our image we want to add some clouds by adding a second pigment. This new pigment uses the bozo pattern to create some nice clouds. Since it lays on top of the other pigment it needs some transparent colors in the color map (look at entries 0.5 to 1.0).

```
sky_sphere {
  pigment {
    gradient y
    color_map {
      [0.000 0.002 color rgb <1.0, 0.2, 0.0>
                   color rgb <1.0, 0.2, 0.0>]
      [0.002 0.200 color rgb <0.8, 0.1, 0.0>
                   color rgb <0.2, 0.2, 0.3>]
    }
    scale 2
    translate -1
  }
  pigment {
    bozo
    turbulence 0.65
    octaves 6
    omega 0.7
    lambda 2
    color_map {
        [0.0 0.1 color rgb <0.85, 0.85, 0.85>
                 color rgb <0.75, 0.75, 0.75>]
        [0.1 0.5 color rgb <0.75, 0.75, 0.75>
                 color rgbt <1, 1, 1, 1>]
        [0.5 1.0 color rgbt <1, 1, 1, 1>
                 color rgbt <1, 1, 1, 1>]
    }
    scale <0.2, 0.5, 0.2>
  }
  rotate -135*x
}
```



A cloudy sky with a setting sun.

The sky sphere has one drawback as you might notice when looking at the final image (`skysph3.pov`). The sun doesn't emit any light and the clouds will not cast any shadows. If you want to have clouds that cast shadows you'll have to use a real, large sphere with an appropriate texture and a light source somewhere outside the sphere.

2.9.3 The Fog

You can use the **`fog`** feature to add fog of two different types to your scene: constant fog and ground fog. The constant fog has a constant density everywhere while the ground fog's density decreases as you move upwards.

The usage of both fog types will be described in the next sections in detail.

2.9.3.1 A Constant Fog

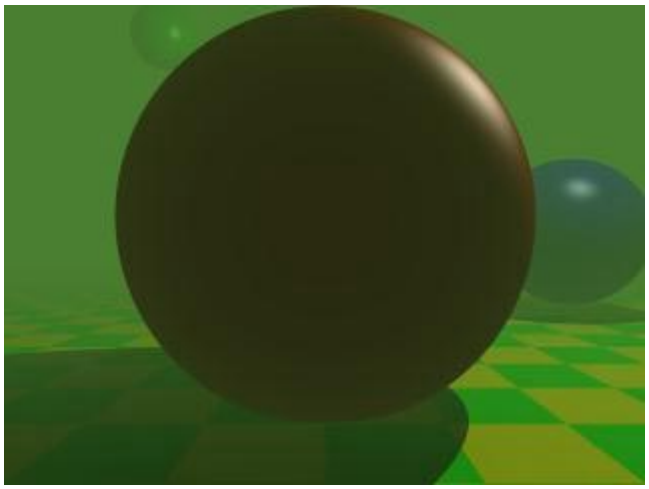
The simplest fog type is the constant fog that has a constant density in all locations. It is specified by a **`distance`** keyword which actually describes the fog's density and a fog **`color`**.

The distance value determines the distance at which 36.8% of the background are still visible (for a more detailed explanation of how the fog is calculated read the reference section "Fog").

The fog color can be used to create anything from a pure white to a red, blood-colored fog. You can also use a black fog to simulate the effect of a limited range of vision.

The following example will show you how to add fog to a simple scene (`fog1.pov`).

```
#include "colors.inc"
camera {
  location  <0, 20, -100>
}
background { color SkyBlue }
plane { y, -10
  pigment {
    checker color Yellow color Green
    scale 20
  }
}
sphere { <0, 25, 0>, 40
  pigment { Red }
  finish { phong 1.0 phong_size 20 }
}
sphere { <-100, 150, 200>,  20
  pigment { Green }
  finish { phong 1.0 phong_size 20 }
}
sphere { <100, 25, 100>, 30
  pigment { Blue }
  finish { phong 1.0 phong_size 20 }
}
light_source { <100, 120, 40> color White}
fog {
  distance 150
  color rgb<0.3, 0.5, 0.2>
}
```



A foggy scene.

According to their distance the spheres in this scene more or less vanish in the greenish fog we used, as does the checkerboard plane.

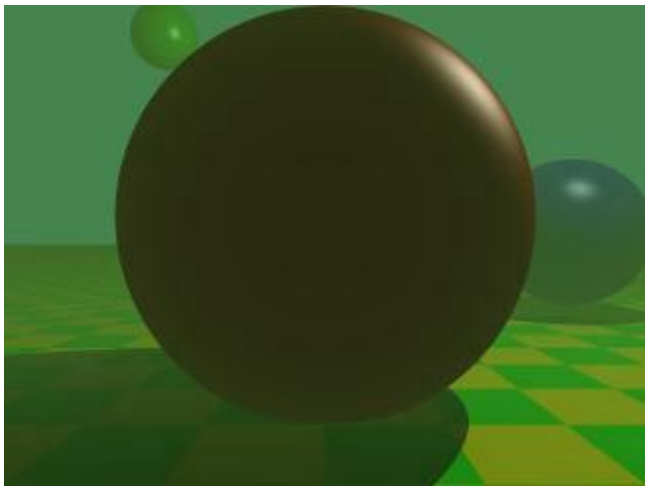
2.9.3.2 Setting a Minimum Translucency

If you want to make sure that the background does not completely vanish in the fog you can set the transmittance channel of the fog's color to the amount of background you always want to be visible.

Using as transmittance value of 0.2 as in

```
fog {
  distance 150
  color rgbt<0.3, 0.5, 0.2, 0.2>
}
```

the fog's translucency never drops below 20% as you can see in the resulting image (`fog2.pov`).



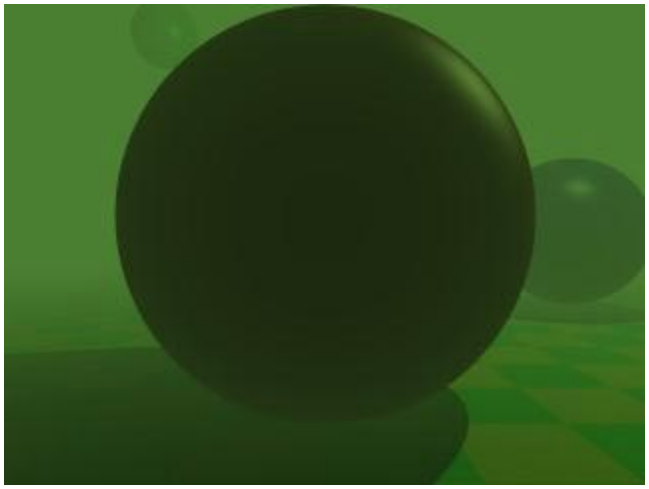
Adding a translucency threshold you make sure that the background does not vanish.

2.9.3.3 Creating a Filtering Fog

The greenish fog we have used so far doesn't filter the light passing through it. All it does is to diminish the light's intensity. We can change this by using a non-zero filter channel in the fog's color (`fog3.pov`).

```
fog {
  distance 150
  color rgbf<0.3, 0.5, 0.2, 1.0>
}
```

The filter value determines the amount of light that is filtered by the fog. In our example 100% of the light passing through the fog will be filtered by the fog. If we had used a value of 0.7 only 70% of the light would have been filtered. The remaining 30% would have passed unfiltered.



A filtering fog.

You'll notice that the intensity of the objects in the fog is not only diminished due to the fog's color but that the colors are actually influenced by the fog. The red and especially the blue sphere got a green hue.

2.9.3.4 Adding Some Turbulence to the Fog

In order to make our somewhat boring fog a little bit more interesting we can add some turbulence, making it look like it had a non-constant density (`fog4.pov`).

```
fog {
  distance 150
  color rgbf<0.3, 0.5, 0.2, 1.0>
  turbulence 0.2
  turb_depth 0.3
}
```

Adding some turbulence makes the fog more interesting.

The **turbulence** keyword is used to specify the amount of turbulence used while the **turb_depth** value is used to move the point at which the turbulence value is calculated along the viewing ray. Values near zero move the point to the viewer while values near one move it to the intersection point (the default value is 0.5). This parameter can be used to avoid noise that may appear in the fog due to the turbulence (this normally happens at very far away intersection points, especially if no intersection occurs, i. e. the background is hit). If this happens just lower the **turb_depth** value until the noise vanishes.

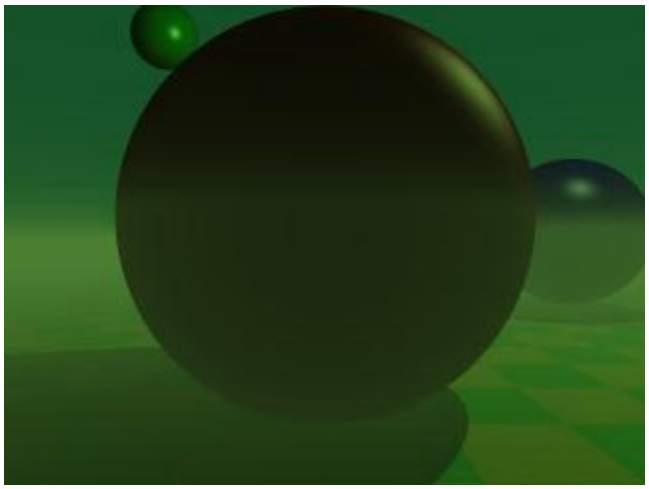
You should keep in mind that the actual density of the fog does not change. Only the distance-based attenuation value of the fog is modified by the turbulence value at a point along the viewing ray.

2.9.3.5 Using Ground Fog

The much more interesting and flexible fog type is the ground fog, which is selected with the **fog_type** statement. It's appearance is described with the **fog_offset** and **fog_alt** keywords. The **fog_offset** specifies the height, i. e. y value, below which the fog has a constant density of one. The **fog_alt** keyword determines how fast the density of the fog will approach zero as one moves along the y axis. At a height of fog_offset+fog_alt the fog will have a density of 25%.

The following example (fog5.pov) uses a ground fog which has a constant density below y=25 (the center of the red sphere) and quickly falls off for increasing altitudes.

```
fog {
  distance 150
  color rgbf<0.3, 0.5, 0.2, 1.0>
  fog_type 2
  fog_offset 25
  fog_alt 1
}
```



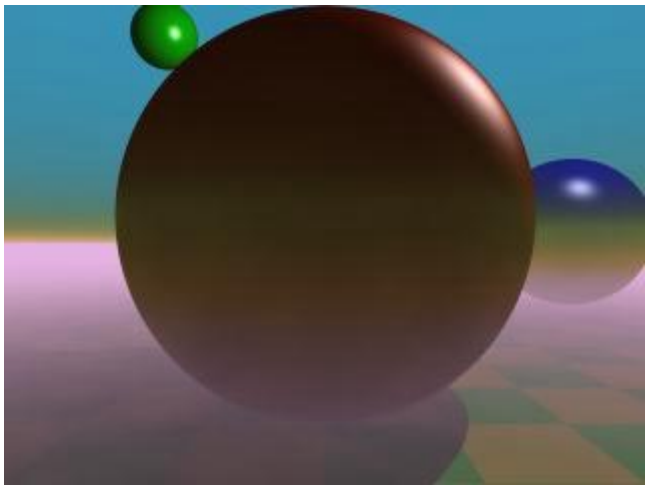
The ground fog only covers the lower parts of the world.'

2.9.3.6 Using Multiple Layers of Fog

It is possible to use several layers of fog by using more than one fog statement in your scene file. This is quite useful if you want to get nice effects using turbulent ground fogs. You could add up several, differently colored fogs to create an eerie scene for example.

Just try the following example (`fog6.pov`).

```
fog {
  distance 150
  color rgb<0.3, 0.5, 0.2>
  fog_type 2
  fog_offset 25
  fog_alt 1
  turbulence 0.1
  turb_depth 0.2
}
fog {
  distance 150
  color rgb<0.5, 0.1, 0.1>
  fog_type 2
  fog_offset 15
  fog_alt 4
  turbulence 0.2
  turb_depth 0.2
}
fog {
  distance 150
  color rgb<0.1, 0.1, 0.6>
  fog_type 2
  fog_offset 10
  fog_alt 2
}
```



Quite nice results can be achieved using multiple layers of fog.

You can combine constant density fogs, ground fogs, filtering fogs, non-filtering fogs, fogs with a translucency threshold, etc.

2.9.3.7 Fog and Hollow Objects

Whenever you use the fog feature and the camera is inside a non-hollow object you won't get any fog effects. For a detailed explanation why this happens see "Empty and Solid Objects".

In order to avoid this problem you have to make all those objects hollow by either making sure the camera is outside these objects (using the **`inverse`** keyword) or by adding the **`hollow`** to them (which is much easier).

2.9.4 The Rainbow

The **`rainbow`** feature can be used to create rainbows and maybe other more strange effects. The rainbow is a fog like effect that is restricted to a cone-like volume.

2.9.4.1 Starting With a Simple Rainbow

The rainbow is specified with a lot of parameters: the angle under which it is visible, the width of the color band, the direction of the incoming light, the fog-like distance based particle density and last but not least the color map to be used.

The size and shape of the rainbow are determined by the **`angle`** and **`width`** keywords. The **`direction`** keyword is used to set the direction of the incoming light, thus setting the rainbow's position. The rainbow is visible when the angle between the direction vector and the incident light direction is larger than angle-width/2 and smaller than angle+width/2.

The incoming light is the virtual light source that is responsible for the rainbow. There needn't be a real light source to create the rainbow effect.

The rainbow is a fog-like effect, i.e. the rainbow's color is mixed with the background color based on the distance to the intersection point. If you choose small distance values the rainbow will be visible on objects, not just in the background. You can avoid this by using a very large distance value.

The color map is the crucial part of the rainbow since it contains all the colors that normally can be seen in a rainbow. The color of the innermost color band is taken from the color map entry 0 while the outermost band is take

from entry 1. You should note that due to the limited color range any monitor can display it is impossible to create a real rainbow. There are just some colors that you cannot display.

The filter channel of the rainbow's color map is used in the same way as with fogs. It determines how much of the light passing through the rainbow is filtered by the color.

The following example shows a simple scene with a ground plane, three spheres and a somewhat exaggerated rainbow (`rainbow1.pov`).

```
#include "colors.inc"
camera {
  location <0, 20, -100>
  look_at <0, 25, 0>
  angle 80
}
background { color SkyBlue }
plane { y, -10 pigment { color Green } }
light_source {<100, 120, 40> color White}
// declare rainbow's colors
#declare r_violet1 = color rgbf<1.0, 0.5, 1.0, 1.0>;
#declare r_violet2 = color rgbf<1.0, 0.5, 1.0, 0.8>;
#declare r_indigo  = color rgbf<0.5, 0.5, 1.0, 0.8>;
#declare r_blue    = color rgbf<0.2, 0.2, 1.0, 0.8>;
#declare r_cyan    = color rgbf<0.2, 1.0, 1.0, 0.8>;
#declare r_green   = color rgbf<0.2, 1.0, 0.2, 0.8>;
#declare r_yellow  = color rgbf<1.0, 1.0, 0.2, 0.8>;
#declare r_orange  = color rgbf<1.0, 0.5, 0.2, 0.8>;
#declare r_red1    = color rgbf<1.0, 0.2, 0.2, 0.8>;
#declare r_red2    = color rgbf<1.0, 0.2, 0.2, 1.0>;
// create the rainbow
rainbow {
  angle 42.5
  width 5
  distance 1.0e7
  direction <-0.2, -0.2, 1>
  jitter 0.01
  color_map {
    [0.000  color r_violet1]
    [0.100  color r_violet2]
    [0.214  color r_indigo]
    [0.328  color r_blue]
    [0.442  color r_cyan]
    [0.556  color r_green]
    [0.670  color r_yellow]
    [0.784  color r_orange]
    [0.900  color r_red1]
  }
}
```

Some irregularity is added to the color bands using the **`jitter`** keyword.

A colorful rainbow.

The rainbow in our sample is much too bright. You'll never see a rainbow like this in reality. You can decrease the rainbow's colors by decreasing the RGB values in the color map.

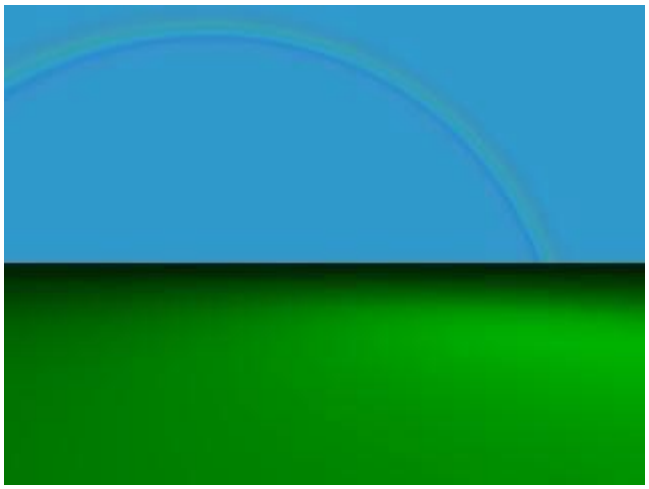
2.9.4.2 Increasing the Rainbow's Translucency

The result we have so far looks much too bright. Just reducing the rainbow's color helps but it's much better to increase the translucency of the rainbow because it is more realistic if the background is visible through the rainbow.

We can use the transmittance channel of the colors in the color map to specify a minimum translucency, just like we did with the fog. To get realistic results we have to use very large transmittance values as you can see in the following example (`rainbow2.pov`).

```
rainbow {
  angle 42.5
  width 5
  distance 1.0e7
  direction <-0.2, -0.2, 1>
  jitter 0.01
  color_map {
    [0.000  color r_violet1 transmit 0.98]
    [0.100  color r_violet2 transmit 0.96]
    [0.214  color r_indigo  transmit 0.94]
    [0.328  color r_blue    transmit 0.92]
    [0.442  color r_cyan    transmit 0.90]
    [0.556  color r_green   transmit 0.92]
    [0.670  color r_yellow  transmit 0.94]
    [0.784  color r_orange  transmit 0.96]
    [0.900  color r_red1    transmit 0.98]
  }
}
```

The transmittance values increase at the outer bands of the rainbow to make it softly blend into the background.



A much more realistic rainbow.

The resulting image looks much more realistic than our first rainbow.

2.9.4.3 Using a Rainbow Arc

Currently our rainbow has a circular shape, even though most of it is hidden below the ground plane. You can easily create a rainbow arc by using the **arc_angle** keyword with an angle below 360 degrees.

If you use **arc_angle 120** for example you'll get a rainbow arc that abruptly vanishes at the arc's ends. This does not look good. To avoid this the **falloff_angle** keyword can be used to specify a region where the arc smoothly blends into the background.

As explained in the rainbow's reference section (see "Rainbow") the arc extends from -arc_angle/2 to arc_angle/2 while the blending takes place from -arc_angle/2 to -falloff_angle/2 and falloff_angle/2 to arc_angle/2. This is the reason why the **falloff_angle** has to be smaller or equal to the **arc_angle**.

In the following examples we use an 120 degrees arc with a 45 degree falloff region on both sides of the arc (rainbow3.pov).

```
rainbow {
  angle 42.5
  width 5
  arc_angle 120
  falloff_angle 30
  distance 1.0e7
  direction <-0.2, -0.2, 1>
  jitter 0.01
  color_map {
    [0.000  color r_violet1 transmit 0.98]
    [0.100  color r_violet2 transmit 0.96]
    [0.214  color r_indigo  transmit 0.94]
    [0.328  color r_blue    transmit 0.92]
    [0.442  color r_cyan    transmit 0.90]
    [0.556  color r_green   transmit 0.92]
    [0.670  color r_yellow  transmit 0.94]
    [0.784  color r_orange  transmit 0.96]
    [0.900  color r_red1    transmit 0.98]
  }
}
```

The arc angles are measured against the rainbows up direction which can be specified using the **up** keyword. By default the up direction is the y-axis.

A rainbow arc.

We finally have a realistic looking rainbow arc.

2.9.5 Animation

There are a number of programs available that will take a series of still image files (such as POV-Ray outputs) and assemble them into animations. Such programs can produce AVI, MPEG, FLI/FLC, QuickTime, or even animated GIF files (for use on the World Wide Web). The trick, therefore, is how to produce the frames. That, of course, is where POV-Ray comes in. In earlier versions producing an animation series was no joy, as everything had to be done manually. We had to set the clock variable, and handle producing unique file names for each individual frame by hand. We could achieve some degree of automation by using batch files or similar scripting devices, but still, We had to set it all up by hand, and that was a lot of work (not to mention frustration... imagine forgetting to set the individual file names and coming back 24 hours later to find each frame had overwritten the last).

Now, at last, with POV-Ray 3, there is a better way. We no longer need a separate batch script or external sequencing programs, because a few simple settings in our INI file (or on the command line) will activate an internal animation sequence which will cause POV-Ray to automatically handle the animation loop details for us.

Actually, there are two halves to animation support: those settings we put in the INI file (or on the command line), and those code modifications we work into our scene description file. If we've already worked with animation in previous versions of POV-Ray, we can probably skip ahead to the section "INI File Settings" below. Otherwise, let's start with basics. Before we get to how to activate the internal animation loop, let's look at a couple examples of how a couple of keywords can set up our code to describe the motions of objects over time.

2.9.5.1 The Clock Variable: Key To It All

POV-Ray supports an automatically declared floating point variable identified as **clock** (all lower case). This is the key to making image files that can be automated. In command line operations, the clock variable is set using the **+k** switch. For example, **+k3.4** from the command line would set the value of clock to 3.4. The same could be accomplished from the INI file using **Clock=3.4** in an INI file.

If we don't set clock for anything, and the animation loop is not used (as will be described a little later) the clock variable is still there - it's just set for the default value of 0.0, so it is possible to set up some POV code for the purpose of animation, and still render it as a still picture during the object/world creation stage of our project.

The simplest example of using this to our advantage would be having an object which is travelling at a constant rate, say, along the x-axis. We would have the statement

```
translate <clock, 0, 0>
```

in our object's declaration, and then have the animation loop assign progressively higher values to clock. And that's fine, as long as only one element or aspect of our scene is changing, but what happens when we want to control multiple changes in the same scene simultaneously?

The secret here is to use normalized clock values, and then make other variables in your scene proportional to clock. That is, when we set up our clock, (we're getting to that, patience!) have it run from 0.0 to 1.0, and then use that as a multiplier to some other values. That way, the other values can be whatever we need them to be, and clock can be the same 0 to 1 value for every application. Let's look at a (relatively) simple example

```
#include "colors.inc"
camera {
  location <0, 3, -6>
  look_at <0, 0, 0>
}
light_source { <20, 20, -20> color White }
plane { y, 0
  pigment { checker color White color Black }
}
sphere { <0, 0, 0> , 1
  pigment {
    gradient x
    color_map {
      [0.0 Blue  ]
      [0.5 Blue  ]
      [0.5 White ]
      [1.0 White ]
    }
    scale .25
  }
  rotate <0, 0, -clock*360>
  translate <-pi, 1, 0>
  translate <2*pi*clock, 0, 0>
}
```

Assuming that a series of frames is run with the clock progressively going from 0.0 to 1.0, the above code will produce a striped ball which rolls from left to right across the screen. We have two goals here:

1. Translate the ball from point A to point B, and,
2. Rotate the ball in exactly the right proportion to its linear movement to imply that it is rolling -- not gliding -- to its final position.

Taking the second goal first, we start with the sphere at the origin, because anywhere else and rotation will cause it to orbit the origin instead of rotating. Throughout the course of the animation, the ball will turn one complete 360 degree turn. Therefore, we used the formula, **`360*clock`** to determine the rotation in each frame. Since clock runs 0 to 1, the rotation of the sphere runs from 0 degrees through 360.

Then we used the first translation to put the sphere at its initial starting point. Remember, we couldn't have just declared it there, or it would have orbited the origin, so before we can meet our other goal (translation), we have to compensate by putting the sphere back where it would have been at the start. After that, we re-translate the sphere by a clock relative distance, causing it to move relative to the starting point. We've chosen the formula of 2*pi* r*clock (the widest circumference of the sphere times current clock value) so that it will appear to move a distance equal to the circumference of the sphere in the same time that it rotates a complete 360 degrees. In this way, we've synchronized the rotation of the sphere to its translation, making it appear to be smoothly rolling along the plane.

Besides allowing us to coordinate multiple aspects of change over time more cleanly, mathematically speaking, the other good reason for using normalized clock values is that it will not matter whether we are doing a ten frame animated GIF, or a three hundred frame AVI. Values of the clock are proportioned to the number of frames, so that same POV code will work without regard to how long the frame sequence is. Our rolling ball will still travel the exact same amount no matter how many frames our animation ends up with.

2.9.5.2 Clock Dependant Variables And Multi-Stage Animations

Okay, what if we wanted the ball to roll left to right for the first half of the animation, then change direction 135 degrees and roll right to left, and toward the back of the scene. We would need to make use of POV's new conditional rendering directives, and test the clock value to determine when we reach the halfway point, then start rendering a different clock dependant sequence. But our goal, as above, it to be working in each stage with a variable in the range of 0 to 1 (normalized) because this makes the math so much cleaner to work with when we have to control multiple aspects during animation. So let's assume we keep the same camera, light, and plane, and let the clock run from 0 to 2! Now, replace the single sphere declaration with the following...

```
#if ( clock <= 1 )
  sphere { <0, 0, 0> , 1
    pigment {
      gradient x
      color_map {
        [0.0 Blue  ]
        [0.5 Blue  ]
        [0.5 White ]
        [1.0 White ]
      }
      scale .25
    }
    rotate <0, 0, -clock*360>
    translate <-pi, 1, 0>
    translate <2*pi*clock, 0, 0>
  }
#else
  // (if clock is > 1, we're on the second phase)
  // we still want to work with  a value from 0 - 1
  #declare ElseClock = clock - 1;
  sphere { <0, 0, 0> , 1
    pigment {
      gradient x
      color_map {
        [0.0 Blue  ]
        [0.5 Blue  ]
        [0.5 White ]
        [1.0 White ]
      }
      scale .25
    }
    rotate <0, 0, ElseClock*360>
    translate <-2*pi*ElseClock, 0, 0>
    rotate <0, 45, 0>
    translate <pi, 1, 0>
  }
#end
```

If we spotted the fact that this will cause the ball to do an unrealistic *snap turn* when changing direction, bonus points for us - we're a born animator. However, for the simplicity of the example, let's ignore that for now. It will be easy enough to fix in the real world, once we examine how the existing code works.

All we did differently was assume that the clock would run 0 to 2, and that we wanted to be working with a normalized value instead. So when the clock goes over 1.0, POV assumes the second phase of the journey has begun, and we declare a new variable **`Elseclock`** which we make relative to the original built in clock, in such a way that while clock is going 1 to 2, Elseclock is going 0 to 1. So, even though there is only one **`clock`**, there can be as many additional variables as we care to declare (and have memory for), so even in fairly complex scenes, the single clock variable can be made the common coordinating factor which orchestrates all other motions.

2.9.5.3 The Phase Keyword

There is another keyword we should know for purposes of animations: the **`phase`** keyword. The phase keyword can be used on many texture elements, especially those that can take a color, pigment, normal or texture map. Remember the form that these maps take. For example:

```
color_map {
  [0.00 White ]
  [0.25 Blue ]
  [0.76 Green ]
  [1.00 Red ]
}
```

The floating point value to the left inside each set of brackets helps POV-Ray to map the color values to various areas of the object being textured. Notice that the map runs cleanly from 0.0 to 1.0?

Phase causes the color values to become shifted along the map by a floating point value which follows the keyword **`phase`**. Now, if we are using a normalized clock value already anyhow, we can make the variable clock the floating point value associated with phase, and the pattern will smoothly shift over the course of the animation. Let's look at a common example using a gradient normal pattern

```
#include "colors.inc"
#include "textures.inc"
#background { rgb<0.8, 0.8, 0.8> }
camera {
  location <1.5, 1, -30>
  look_at <0, 1, 0>
  angle 10
}
light_source { <-100, 20, -100> color White }
// flag
polygon { 5, <0, 0>, <0, 1>, <1, 1>, <1, 0>, <0, 0>
  pigment { Blue }
  normal {
    gradient x
    phase clock
    scale <0.2, 1, 1>
    sine_wave
  }
  scale <3, 2, 1>
  translate <-1.5, 0, 0>
}
// flagpole
cylinder { <-1.5, -4, 0>, <-1.5, 2.25, 0>, 0.05
  texture { Silver_Metal }
}
// polecap
sphere { <-1.5, 2.25, 0>, 0.1
  texture { Silver_Metal }
}
```

Now, here we've created a simple blue flag with a gradient normal pattern on it. We've forced the gradient to use a sine-wave type wave so that it looks like the flag is rolling back and forth as though flapping in a breeze. But the real magic here is that phase keyword. It's been set to take the clock variable as a floating point value which, as the clock increments slowly toward 1.0, will cause the crests and troughs of the flag's wave to shift along the x-axis. Effectively, when we animate the frames created by this code, it will look like the flag is actually rippling in the wind.

This is only one, simple example of how a clock dependant phase shift can create interesting animation effects. Trying phase will all sorts of texture patterns, and it is amazing the range of animation effects we can create simply by phase alone, without ever actually moving the object.

2.9.5.4 Do Not Use Jitter Or Crand

One last piece of basic information to save frustration. Because jitter is an element of anti-aliasing, we could just as easily have mentioned this under the INI file settings section, but there are also forms of anti-aliasing used in area lights, and the new atmospheric effects of POV-Ray, so now is as good a time as any.

Jitter is a very small amount of random ray perturbation designed to diffuse tiny aliasing errors that might not otherwise totally disappear, even with intense anti-aliasing. By randomizing the placement of erroneous pixels, the error becomes less noticeable to the human eye, because the eye and mind are naturally inclined to look for regular patterns rather than random distortions.

This concept, which works fantastically for still pictures, can become a nightmare in animations. Because it is random in nature, it will be different for each frame we render, and this becomes even more severe if we dither the final results down to, say 256 color animations (such as FLC's). The result is jumping pixels all over the scene, but especially concentrated any place where aliasing would normally be a problem (e.g., where an infinite plane disappears into the distance).

For this reason, we should always set jitter to **`off`** in area lights and anti-aliasing options when preparing a scene for an animation. The (relatively) small extra measure quality due to the use of jitter will be offset by the ocean of jumpies that results. This general rule also applies to any truly random texture elements, such as **`crand`**.

2.9.5.5 INI File Settings

Okay, so we have a grasp of how to code our file for animation. We know about the clock variable, user declared clock-relative variables, and the phase keyword. We know not to jitter or crand when we render a scene, and we're all set build some animations. Alright, let's have at it.

The first concept we'll need to know is the INI file settings, **`Initial_Frame`** and **`Final_Frame`**. These are very handy settings that will allow us to render a particular number of frames and each with its own unique frame number, in a completely hands free way. It is of course, so blindingly simple that it barely needs explanation, but here's an example anyway. We just add the following lines to our favorite INI file settings

```
Initial_Frame = 1
Final_Frame = 20
```

and we'll initiate an automated loop that will generate 20 unique frames. The settings themselves will automatically append a frame number onto the end of whatever we have set the output file name for, thus giving each frame an unique file number without having to think about it. Secondly, by default, it will cycle the clock variable up from 0 to 1 in increments proportional to the number of frames. This is very convenient, since, no matter whether we are making a five frame animated GIF or a 300 frame MPEG sequence, we will have a clock value which smoothly cycles from exactly the same start to exactly the same finish.

Next, about that clock. In our example with the rolling ball code, we saw that sometimes we want the clock to cycle through values other than the default of 0.0 to 1.0. Well, when that's the case, there are setting for that too. The format is also quite simple. To make the clock run, as in our example, from 0.0 to 2.0, we would just add to your INI file the lines

```
Initial_Clock = 0.0
Final_Clock = 2.0
```

Now, suppose we were developing a sequence of 100 frames, and we detected a visual glitch somewhere in frames, say 51 to 75. We go back over our code and we think we've fixed it. We'd like to render just those 25 frames instead of redoing the whole sequence from the beginning. What do we change?

If we said make **`Initial_Frame = 51`**, and **`Final_Frame = 75`**, we're wrong. Even though this would re-render files named with numbers 51 through 75, they will not properly fit into our sequence, because the clock will begin at its initial value starting with frame 51, and cycle to final value ending with frame 75. The only time **`Initial_Frame`** and **`Final_Frame`** should change is if we are doing an essentially new sequence that will be appended onto existing material.

If we wanted to look at just 51 through 75 of the original animation, we need two new INI settings

```
Subset_Start_Frame = 51
Subset_End_Frame = 75
```

Added to settings from before, the clock will still cycle through its values proportioned from frames 1 to 100, but we will only be rendering that part of the sequence from the 51st to the 75th frames.

This should give us a basic idea of how to use animation. Although, this introductory tutorial doesn't cover all the angles. For example, the last two settings we just saw, subset animation, can take fractional values, like 0.5 to 0.75, so that the number of actual frames will not change what portion of the animation is being rendered. There is also support for efficient odd-even field rendering as would be useful for animations prepared for display in interlaced playback such as television (see the reference section for full details).

With POV-Ray 3 now fully supporting a complete host of animation options, a whole fourth dimension is added to the raytracing experience. Whether we are making a FLIC, AVI, MPEG, or simply an animated GIF for our web site, animation support takes a lot of the tedium out of the process. And don't forget that phase and clock can be used to explore the range of numerous texture elements, as well as some of the more difficult to master objects (hint: the julia fractal for example). So even if we are completely content with making still scenes, adding animation to our repertoire can greatly enhance our understanding of what POV-Ray is capable of. Adventure awaits!

3 POV-Ray 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.

POV-Ray was originally created as a command-line program for operating systems without graphical interfaces, dialog boxes and pull-down menus. Most versions of POV-Ray still use command-line switches to tell it what to do. This documentation assumes you are using the command-line version. If you are using Macintosh, MS-Windows or other GUI versions, there will be dialog boxes or menus which do the same thing. There is system-specific documentation for each system describing the specific commands.

3.1 Setting POV-Ray Options

There are two distinct ways of setting POV-Ray options: command line switches and INI file keywords. Both are explained in detail in the following sections.

3.1.1 Command Line Switches

Command line switches consist of a **+** (plus) or **-** (minus) sign, followed by one or more alphabetic characters and possibly a numeric value. Here is a typical command line with switches.

```
POVRAY +Isimple.pov +V +W80 +H60
```

`povray` is the name of the program and it is followed by several switches. Each switch begins with a plus or minus sign. The **+I** switch with the filename tells POV-Ray what scene file it should use as input and **+V** tells the program to output its status to the text screen as it's working. The **+W** and **+H** switches set the width and height of the image in pixels. This image will be 80 pixels wide by 60 pixels high.

In switches which toggle a feature, the plus turns it on and minus turns it off. For example **+P** turns on the *pause for keypress when finished* option while **-P** turns it off. Other switches are used to specify values and do not toggle a feature. Either plus or minus may be used in that instance. For example **+W320** sets the width to 320 pixels. You could also use **-W320** and get the same results.

Switches may be specified in upper or lower case. They are read left to right but in general may be specified in any order. If you specify a switch more than once, the previous value is generally overwritten with the last specification. The only exception is the **+L** switch for setting library paths. Up to ten unique paths may be specified.

Almost all **+** or **-** switches have an equivalent option which can be used in an INI file which is described in the next section. A detailed description of each switch is given in the option reference section.

3.1.2 Using INI Files

Because it is difficult to set more than a few options on a command line, you have the ability to put multiple options in one or more text files. These initialization files or INI files have .ini as their default extension. Previous versions of POV-Ray called them default files or DEF files. You may still use existing DEF files with this version of POV-Ray.

The majority of options you use will be stored in INI files. The command line switches are recommended for options which you will turn off or on frequently as you perform test renderings of a scene you are developing. The file `povray.ini` is automatically read if present. You may specify additional INI files on the command-line by simply typing the file name on the command line. For example:

```
POVRAY MYOPTS.INI
```

If no extension is given, then `.ini` is assumed. POV-Ray knows this is not a switch because it is not preceded by a plus or minus. In fact a common error among new users is that they forget to put the **+I** switch before the input file name. Without the switch, POV-Ray thinks that the scene file `simple.pov` is an INI file. Don't forget! If no plus or minus precedes a command line switch, it is assumed to be an INI file name.

You may have multiple INI files on the command line along with switches. For example:

```
POVRAY MYOPTS +V OTHER
```

This reads options from `myopts.ini`, then sets the **+V** switch, then reads options from `other.ini`.

An INI file is a plain ASCII text file with options of the form...

```
Option_keyword=VALUE ; Text after semicolon is a comment
```

For example the INI equivalent of the switch **+Isimple.pov** is...

```
Input_File_Name=simple.pov
```

Options are read top to bottom in the file but in general may be specified in any order. If you specify an option more than once, the previous values are generally overwritten with the last specification. The only exception is the **Library_Path=path** options. Up to ten unique paths may be specified.

Almost all INI-style options have equivalent **+** or **-** switches. The option reference section gives a detailed description of all POV-Ray options. It includes both the INI-style settings and the **+**/**-** switches.

The INI keywords are not case sensitive. Only one INI option is permitted per line of text. You may also include switches in your INI file if they are easier for you. You may have multiple switches per line but you should not mix switches and INI options on the same line. You may nest INI files by simply putting the file name on a line by itself with no equals sign after it. Nesting may occur up to ten levels deep.

For example:

```
; This is a sample INI file. This entire line is a comment.
; Blank lines are permitted.
Input_File_Name=simple.pov ;This sets the input file name
+W80 +H60 ; Traditional +/- switches are permitted too
MOREOPT   ; Read MOREOPT.INI and continue with next line
+V        ; Another switch
; That's all folks!
```

INI files may have labeled sections so that more than one set of options may be stored in a single file. Each section begins with a label in [] brackets. For example:

```
; RES.INI
; This sample INI file is used to set resolution.
+W120 +H100  ; This section has no label.
             ; Select it with "RES"
[Low]
+W80 +H60    ; This section has a label.
             ; Select it with "RES[Low]"
[Med]
+W320 +H200  ; This section has a label.
             ; Select it with "RES[Med]"
[High]
+W640 +H480  ; Labels are not case sensitive.
             ; "RES[high]" works
[Really High]
+W800 +H600  ; Labels may contain blanks
```

When you specify the INI file you should follow it with the section label in brackets. For example...

```
POVRAY RES[Med] +Imyfile.pov
```

POV-Ray reads `res.ini` and skips all options until it finds the label **`Med`**. It processes options after that label until it finds another label and then it skips. If no label is specified on the command line then only the unlabeled area at the top of the file is read. If a label is specified, the unlabeled area is ignored.

3.1.3 Using the POVINI Environment Variable

The environment variable POVINI is used to specify the location and name of a default INI file that is read every time POV-Ray is executed. If POVINI is not specified, or if your computer platform does not use environment variables, a default INI file may be read. If the specified file does not exist, a warning message is printed.

To set the environment variable under MS-DOS you might put the following line in your `autoexec.bat` file...

```
set POVINI=c:\povray3\default.ini
```

On most operating systems the sequence of reading options is as follows:

1. Read options from default INI file specified by the POVINI environment variable or platform specific INI file.
2. Read switches from command line (this includes reading any specified INI/DEF files).

The POVRAYOPT environment variable supported by previous POV-Ray versions is no longer available.

3.2 Options Reference

As explained in the previous section, 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)

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

3.2.1 Animation Options

POV-Ray 3.0 greatly improved its animation capability with the addition of an 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. The internal animation loop is simple yet flexible. You may still use external programs or batch files to create animations without the internal loop as you may have done in POV-Ray 2.

3.2.1.1 External Animation Loop

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

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.

3.2.1.2 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*
+KFI*n*	Same as **Initial_Frame=***n*
+KFF*n*	Same as **Final_Frame=***n*
+KI*n.n*	Same as **Initial_Clock=***n.n*
+KF*n.n*	Same as **Final_Clock=***n.n*

The internal animation loop new to POV-Ray 3.0 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 that 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 **Frame_Initial** or **Frame_Final**. 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 that 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".

3.2.1.3 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
+SF*n* or **+SF***0.n*	Same as **Subset_Start_Frame**
+EF*n* or **+EF***0.n*	Same as **Subset_End_Frame**

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.

3.2.1.4 Cyclic Animation

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

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.

3.2.1.5 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

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.

3.2.2 Output Options

3.2.2.1 General Output Options

3.2.2.1.1 Height and Width of Output

Height=*n*	Sets screen height to *n* pixels
Width=*n*	Sets screen width to *n* pixels
+H*n*	Same as **Height=***n (when n > 8)*
+W*n*	Same as **Width=***n*

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.

3.2.2.1.2 Partial Output Options

Start_Column=*n*	Set first column to *n* pixels
Start_Column=*0.n*	Set first column to *n* percent of width
+SC*n* or **+SC***0.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

+SR*n* or **+S***n*	Same as **Start_Row=***n*
+SR*0.n* or **+S***0.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
+EC*n* or **+EC***0.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
+ER*n* or **+E***n*	Same as **End_Row=***n*
+ER*0.n* or **+E***0.n*	Same as **End_Row=***0.n*

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

3.2.2.1.3 Interrupting Options

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
+X*n*	Set to test for abort every *n* pixels on
-X*n*	Set to test for abort off (in future test every *n* pixels)

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 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.

3.2.2.1.4 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`
+GI*file*	Same as **Create_Ini=***file*

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.

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.

3.2.2.2 Display Output Options

3.2.2.2.1 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
+D*x*	Set display on; Set mode to *x*
-D*x*	Set display off; but for future use mode *x*
Palette=*y*	Set display palette to *y*; does not affect on/off
+D*xy*	Set display on; Set mode *x*; Set palette *y*
-D*xy*	Set display off; use mode *x*, palette *y* in future
Display_Gamma=*n.n*	Sets the display gamma to *n.n*

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.

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 new with POV-Ray 3.0, and 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 that 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. Some high-end PC graphics cards can do hardware gamma correction and should use the current Display_Gamma setting, usually 1.0. A gamma test image is also available to help users to set their **Display_Gamma** accurately.

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.

3.2.2.2.2 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

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.

3.2.2.2.3 Mosaic Preview

Preview_Start_Size=*n*	Set mosaic preview start size to n
+SP*n*	Same as Preview_Start_Size=n
Preview_End_Size=*n*	Set mosaic preview end size to n
+EP*n*	Same as Preview_End_Size=n

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.

Future versions of POV-Ray will include some system of temporary files or buffers which will eliminate these inefficiencies and limitations. Mosaic preview is still a very useful feature for test renderings.

3.2.2.3 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

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**.

3.2.2.3.1 Output File Type

Output_File_Type=*x*	Sets file output format to *x*
+F*xn*	Sets file output on; sets format *x*, depth *n*
-F*xn*	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*

The default type of image file depends on which platform you are using. MS-DOS and most others default to 24-bit uncompressed Targa. 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...

+FC	Compressed Targa-24 format (RLE, run length encoded)
+FN	New PNG (portable network graphics) format
+FP	Unix PPM format
+FS	System-specific such as Mac Pict or Windows BMP
+FT	Uncompressed Targa-24 format

Note that the obsolete **+FD** dump format and **+FR** raw format have been dropped from POV-Ray 3.0 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 a new 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 ray-tracing. 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 allows 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 format files by setting **Bits_Per_Color=***n* or by specifying **+FN***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**. See section "Using the Alpha Channel" for further details on transparency.

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.

3.2.2.3.2 Output File Name

Output_File_Name=*file*	Sets output file to *file*
+O*file*	Same as **Output_File_Name=***file*

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 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.

3.2.2.3.3 Output File Buffer

`Buffer_Output=`*bool*	Turn output buffering on/off
`+B`	Turn output buffering on
`-B`	Turn output buffering off
`Buffer_Size=`*n*	Set output buffer size to *n* kilobytes. If *n* is zero, no buffering. If *n* < system default, the system default is used.
`+B`*n*	Turn buffer on, set size *n*
`-B`*n*	Turn buffer off, but for future set size *n*

The **`Buffer_Output`** and **`Buffer_Size`** options and the **`+B`** switch allows you to assign large buffers to the output file. This reduces the amount of time spent writing to the disk. If this parameter is not specified, then as each row of pixels is finished, the line is written to the file and the file is flushed. On most systems, this operation ensures that the file is written to the disk so that in the event of a system crash or other catastrophic event, at least a part of the picture has been stored properly and retrievable on disk. The default is not to use any buffer.

3.2.2.4 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.

3.2.2.4.1 File Type

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

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`	New 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

Note that **+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.

3.2.2.4.2 File Name

Histogram_Name=*file*	Set histogram name to *file*
+HN*file*	Same as **Histogram_Name=***file*

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 `histgram.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.

3.2.2.4.3 Grid Size

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

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 +HNhistogrm.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 that 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.

3.2.3 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.

3.2.3.1 Input File Name

Input_File_Name=*file*	Sets input file name to *file*
+I*file*	Same as **Input_File_Name=***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.

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-
```

3.2.3.2 Library Paths

Library_Path=*path*	Add *path* to list of library paths
+L*path*	Same as **Library_Path=***path*

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 counts once. The current directory will be searched first followed by the indicated library directories in the order in which you specified them.

3.2.3.3 Language Version

Version=*n.n*	Set initial language compatibility to version *n.n*
+MV*n.n*	Same as **Version=***n.n*

As POV-Ray as evolved from version 1.0 through 3.1 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 setting. 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=3.1**.

NOTE: Some obsolete or re-designed features *are totally unavailable in POV-Ray 3.1 REGARDLES OF THE VERSION SETTING.* Details on these features are noted throughout this documentation.

3.2.4 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

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.

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.

3.2.4.1 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.

3.2.4.2 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) Go back to (4a) until all frames are done.
5) Execute **Post_Scene_Command** if any.
6) Exit POV-Ray.

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 that 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 also 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.

3.2.4.3 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

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

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.

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.

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.

On return from a **Pre_Frame_Command** if there is an action triggered...

...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).

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).

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.

3.2.5 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. New to POV-Ray 3.0, 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.

3.2.5.1 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.

3.2.5.2 Console Text Output

`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**

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.

3.2.5.3 Directing Text Streams to Files

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*
+GD*file*	Both **Debug_Console=On**, **Debug_File=***file*
-GD*file*	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*
+GF*file*	Both **Fatal_Console=On**, **Fatal_File=***file*
-GF*file*	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*
+GR*file*	Both **Render_Console=On**, **Render_File=***file*
-GR*file*	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*
+GS*file*	Both **Statistic_Console=On**, **Statistic_File=***file*
-GS*file*	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*
+GW*file*	Both **Warning_Console=On**, **Warning_File=***file*
-GW*file*	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*
+GA*file*	Both **All_Console=On**, **All_File=***file*

-GA*file*	Both **All_Console=Off, All_File=***file*

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.

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.

3.2.5.4 Help Screen Switches

+H or **+?**	Show help screen 0 if this is the only switch
+H0 to **+H8**	Show help screen 0 to 8 if this is the only switch
+?0 to **+?8**	Same as **+H0** to **+H8**

Note that there are no INI style equivalents to these options.

Graphical interface versions of POV-Ray such as Mac or Windows have extensive online help. Other versions of POV-Ray have only a few quick-reference help screens. The **+?** switch, optionally followed by a single digit from 0 to 8, will display these help screens to the banner text stream. 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 however that 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 and if the value after the switch is less than or equal to 8.

3.2.6 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.

3.2.6.1 Quality Settings

Quality=*n*	Set quality value to *n (0 <= n <= 11)*
+Q*n*	Same as **Quality=***n*

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 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.

8	Compute reflected, refracted, and transmitted rays.
9	Compute media.
10	Compute radiosity but no media
11	Compute radiosity and media

The default is 9 if not specified.

3.2.6.2 Radiosity Setting

Radiosity=*bool*	Turns radiosity on/off
+QR	Turns radiosity on
-QR	Turns radiosity on

Radiosity is an additional calculation which computes diffuse inter-reflection. It is an extremely slow calculation that is somewhat experimental. By default, radiosity is off. The parameters which control how radiosity calculations are performed are specified in the **radiosity** section of the **global_settings** statement. See section "Radiosity" for further details.

3.2.6.3 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*
+MB*n*	Turn bounding on; bound threshold to *n*
-MB*n*	Turn bounding off; for 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

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 " an interior for these objects.

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.

3.2.6.4 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

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**.

3.2.6.5 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)
+AM*n*	Same as **Sampling_Method=***n*
Antialias_Threshold=*n.n*	Sets anti-aliasing threshold
+A*n.n*	Sets aa on with aa-threshold at *n.n*
-A*n.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
+J*n.n*	Sets aa-jitter on; jitter amount to *n.n*. If *n.n <= 0* aa-jitter is set off
-J*n.n*	Sets aa-jitter off (jitter amount *n.n* in future)
Antialias_Depth=*n*	Sets aa-depth *(1 <= n <= 9)*
+R*n*	Same as **Antialias_Depth=***n*

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 negative 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 more than a 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

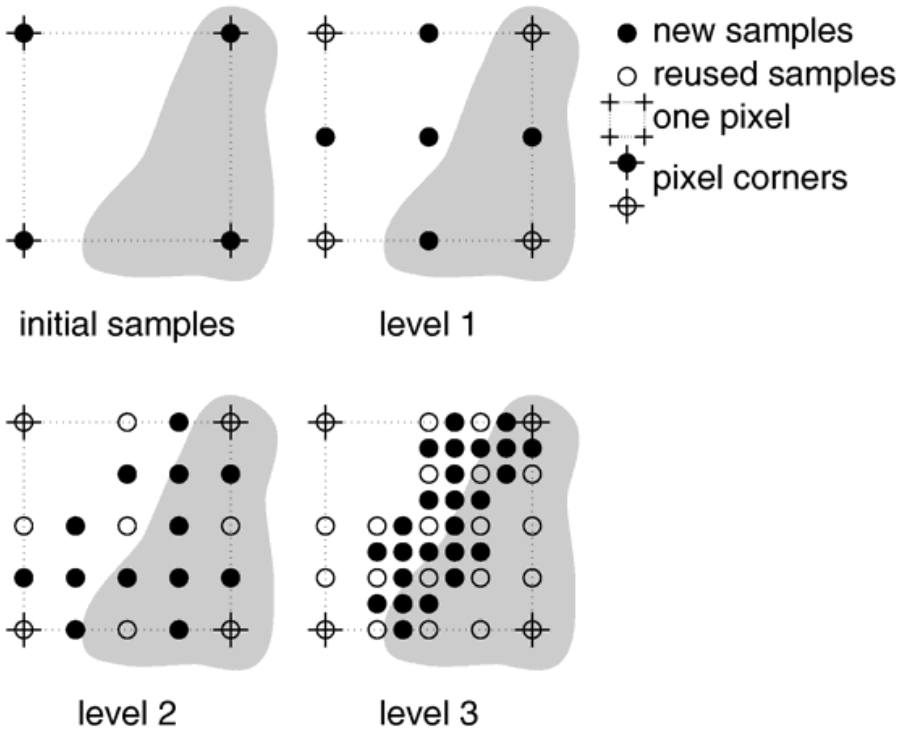
```
diff = abs(r1-r2) + abs(g1-g2) + abs(b1-b2)
```

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).



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 were 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.

`+R`*n*	Number of 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

You should note that 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 artifacts 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 that 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.

4 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 " 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.

4.1 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.

4.1.1 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 or 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.

abs	direction	metallic	slope_map
absorption	disc	min	smooth
acos	distance	minimum_reuse	smooth_triangle
acosh	distance_maximum	mod	sor
adaptive	div	mortar	specular
adc_bailout	eccentricity	nearest_count	sphere
agate	else	no	spherical
agate_turb	emission	normal	spiral1
all	end	normal_map	spiral2
alpha	error	no_shadow	spotlight

ambient	error_bound	number_of_waves	spotted
ambient_light	exp	object	sqr
angle	extinction	octaves	sqrt
aperture	fade_distance	off	statistics
append	fade_power	offset	str
arc_angle	falloff	omega	strcmp
area_light	falloff_angle	omnimax	strength
array	false	on	strlen
asc	fclose	once	strlwr
asin	file_exists	onion	strupr
asinh	filter	open	sturm
assumed_gamma	finish	orthographic	substr
atan	fisheye	panoramic	superellipsoid
atan2	flatness	perspective	switch
atanh	flip	pgm	sys
average	floor	phase	t
background	focal_point	phong	tan
bezier_spline	fog	phong_size	tanh
bicubic_patch	fog_alt	pi	text
black_hole	fog_offset	pigment	texture
blob	fog_type	pigment_map	texture_map
blue	fopen	planar	tga
blur_samples	frequency	plane	thickness
bounded_by	gif	png	threshold
box	global_settings	point_at	tightness
boxed	gradient	poly	tile2
bozo	granite	polygon	tiles
break	gray_threshold	poly_wave	torus
brick	green	pot	track
brick_size	height_field	pow	transform
brightness	hexagon	ppm	translate
brilliance	hf_gray_16	precision	transmit
bumps	hierarchy	prism	triangle
bump_map	hollow	pwr	triangle_wave
bump_size	hypercomplex	quadratic_spline	true
camera	if	quadric	ttf
case	ifdef	quartic	turbulence
caustics	iff	quaternion	turb_depth
ceil	ifndef	quick_color	type
checker	image_map	quick_colour	u
chr	include	quilted	ultra_wide_angle
clipped_by	int	radial	undef
clock	interior	radians	union
clock_delta	interpolate	radiosity	up
color	intersection	radius	use_color
color_map	intervals	rainbow	use_colour
colour	inverse	ramp_wave	use_index
colour_map	ior	rand	u_steps
component	irid	range	v
composite	irid_wavelength	ratio	val
concat	jitter	read	variance
cone	julia_fractal	reciprocal	vaxis_rotate
confidence	lambda	recursion_limit	vcross
conic_sweep	lathe	red	vdot
control0	leopard	reflection	version
control1	light_source	reflection_exponent	vlength

cos	linear_spline	refraction	vnormalize
cosh	linear_sweep	render	vrotate
count	local	repeat	v_steps
crackle	location	rgb	warning
crand	log	rgbf	warp
cube	looks_like	rgbft	water_level
cubic	look_at	rgbt	waves
cubic_spline	low_error_factor	right	while
cubic_wave	macro	ripples	width
cylinder	mandel	rotate	wood
cylindrical	map_type	roughness	wrinkles
debug	marble	samples	write
declare	material_map	scale	x
default	matrix	scallop_wave	y
degrees	max	scattering	yes
density	max_intersections	seed	z
density_file	max_iteration	shadowless	
density_map	max_trace_level	sin	
dents	media	sine_wave	
difference	media_attenuation	sinh	
diffuse	media_interaction	sky	
dimensions	merge	sky_sphere	
dimension_size	mesh	slice	

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.

4.1.2 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.

4.1.3 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**

FLOAT_FUNCTION:
abs(*FLOAT* **)** | **acos(** *FLOAT* **)** | **val(** *STRING* **)** | **asc(** *STRING* **)** |
asin(*FLOAT* **)** | **atan2(** *FLOAT* **,** *FLOAT* **)** | **ceil(** *FLOAT* **)** | **cos(** *FLOAT* **)** |
degrees(*FLOAT* **)** | **div(** *FLOAT* **,** *FLOAT* **)** | **exp(** *FLOAT* **)** |
file_exists(*STRING* **)** | **floor(** *FLOAT* **)** | **int(** *FLOAT* **)** | **log(** *FLOAT* **)** |
max(*FLOAT* **,** *FLOAT* **)** | **min(** *FLOAT* **,** *FLOAT* **)** | **mod(** *FLOAT* **,** *FLOAT* **)** |
pow(*FLOAT* **,** *FLOAT* **)** | **radians(** *FLOAT* **)** | **sin(** *FLOAT* **)** | **sqrt(** *FLOAT* **)** |

strcmp(*STRING* **,** *STRING* **)** | **strlen(** *STRING* **)** | **tan(** *FLOAT* **)** |
vdot(*VECTOR* **,** *VECTOR* **)** | **vlength(** *VECTOR* **)** | **seed(** *FLOAT* **)** | **rand(** *FLOAT* **)** |
dimensions(*ARRAY_IDENTIFIER* **)** | **dimension_size(** *ARRAY_IDENTIFIER* **,** *FLOAT* **)**

FLOAT_BUILT-IN_IDENT:
clock | **pi** | **version** | **true** | **yes** | **on** | **false** | **no** |
off | **clock_delta**

FULL_EXPRESSION:
*LOGICAL_EXPRESSION [***?** *FULL_EXPRESSION* **:** *FULL_EXPRESSION]*

LOGICAL_EXPRESSION:
REL_TERM [LOGICAL_OPERATOR REL_TERM]

LOGICAL_OPERATOR:
& | **|** *(note this means an ampersand or a vertical bar is a logical operator)*

REL_TERM:
FLOAT [REL_OPERATOR FLOAT]

REL_OPERATOR:
< | **<=** | **=** | **>=** | **>** | **!=**

INT:
FLOAT (note any syntax which requires a integer INT will accept a FLOAT and it will be truncated to an integer internally by POV-Ray).

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".

4.1.3.1 Float 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

4.1.3.2 Float 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 that there should be a semi-colon after the expression in a float declaration. This semi-colon is new with POV-Ray version 3.1. 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.

4.1.3.3 Float 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

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.

(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

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 that these are not bit-wise operations, they are logical.

(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

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.

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

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.

4.1.3.4 Built-in Float Identifiers

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

FLOAT_BUILT-IN_IDENT:

clock | **pi** | **version** | **true** | **yes** | **on** | **false** | **no** | **off** | **clock_delta**

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

```
#declare pi = 3.14159265358979323846;
#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 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.

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.

The built-in float identifier **version** contains the current setting of the version compatibility option. Although this value defaults to 3.1 which is 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 that there should be a semi-colon after the float expression in a **#version** directive. This semi-colon is new with POV-Ray version 3.1. If omitted, it generates a warning and some macros may not work properly.

4.1.3.5 Boolean Keywords

The built-in float identifiers **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 default on 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
```

4.1.3.6 Float 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.

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**.

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".

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

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.

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.

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

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

max(A,B) Maximum of **A** and **B**. Returns **A** if **A** larger than **B**. Otherwise returns **B**.

min(A,B) Minimum of **A** and **B**. 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**.

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(A) 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.

sin(A) Sine of **A**. Returns the sine of the angle **A**, where **A** is measured in radians.

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.

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**. 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 new **#macro** directive. See the section "User Defined Macros" for more details.

4.1.4 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:
vaxis_rotate(*VECTOR* **,** *VECTOR* **,** *FLOAT* **)** |
vcross(*VECTOR* **,** *VECTOR* **)** |
vrotate(*VECTOR* **,** *VECTOR* **)** |
vnormalize(*VECTOR* **)**

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

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

4.1.4.1 Vector Literals

Vectors 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.

4.1.4.2 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 that there should be a semi-colon after the expression in a vector declaration. This semi-colon is new with POV-Ray version 3.1. 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 that 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.

4.1.4.3 Vector 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 its 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.

4.1.4.4 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>**.

4.1.4.5 Built-in Vector Identifiers

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.

4.1.4.6 Vector 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.

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 proportional to the angle between them. 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/vlength(A)*.

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.

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 new **#macro** directive. See the section "User Defined Macros" for more details.

4.1.5 Specifying Colors

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 is added to the light passing through so this is called additive transparency.

Note that 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 below, 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:
    COLOR_BODY |
    color *COLOR_BODY* | *(this means the keyword* **color** or **colour** *may*
    colour *COLOR_BODY* *optionally precede any color specification)*

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.

4.1.5.1 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.

4.1.5.2 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.

4.1.5.3 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 that there should be a semi-colon at the end of the declaration. This semi-colon is new with POV-Ray version 3.1. 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.

4.1.5.4 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.

4.1.5.5 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.

One final issue, 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.

4.1.6 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* **)**

4.1.6.1 String Literals

String literals begin with a double quote mark "" which is followed by up to 256 printable ASCII characters and are terminated by another double quote mark. 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. For 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, most of the time you need do nothing special. However if the string ends in a backslash, you will have to specify two. For example:

"This is a backslash \ and so is this\\"

Is converted to:

This is a backslash \ and so is this\

The substitution of **\"** with a **"** occurs in all POV-Ray string literals regardless usage however other formatting codes such as **\n** for new line are only supported in user message streams. See "Text Formatting" for details.

4.1.6.2 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 that 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.

4.1.6.3 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 ASCII value is **B**. Returns a single character string. The ASCII value of the character is specified by an integer **B** which must be in the range 0 to 255. For example **chr(70)** is the string "F". When rendering text objects you should be aware that the characters rendered for values of *B > 127* are dependent on the (TTF) font being used. Many (TTF) fonts use the Latin-1 (ISO 8859-1) character set, but not all do.

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 "EF". If *P+L>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!".

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 new **#macro** directive. See the section "User Defined Macros" for more details.

4.1.7 Array Identifiers

New in POV-Ray 3.1 you may now declare arrays of identifiers of up to five dimensions. Any item that can be declared as an identifier can be declared in an array.

4.1.7.1 Declaring Arrays

The syntax for declaring an array is as follows:

STRING_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 Initalizers". Here is an example of a one-dimensional, uninitalized array.

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

This declares an uninitalized 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 uninitalized 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: uninitalized 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
```

Large uninitalized 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 that 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.

4.1.7.2 Array Initalizers

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.

4.2 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	**#case**	**#debug**	**#declare**
#default	**#else**	**#end**	**#fclose**
#fopen	**#local**	**#macro**	**#read**
#render	**#statistics**	**#switch**	**#undef**
#version	**#warning**	**#write**	

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.

4.2.1 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.

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".

4.2.2 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.

4.2.2.1 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 new with POV-Ray version 3.1. If omitted, it generates a warning and some macros may not work properly.

Declarations, like most language directives, can appear 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. However if you attempt to re-declare an identifier as anything other than its original type, it will generate a warning message.

Note that 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 for 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.

4.2.2.2 #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 ".

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 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".

4.2.2.3 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 multiply-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)**. Note 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. Note that 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.

4.2.2.4 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".

4.2.3 File I/O Directives

New in POV-Ray 3.1 you may now 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.

4.2.3.1 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.

4.2.3.2 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".

4.2.3.3 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 global/local 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.

4.2.3.4 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_ITEM*, *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 that 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 delimeted 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.

4.2.4 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}
```

```
   }
 }
```

means objects will default to red bumps and slightly high ambient finish. Note 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 that 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.

4.2.5 The #version Directive

As POV-Ray as evolved from version 1.0 through 3.1 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 that there should be a semi-colon after the float expression in a **`#version`** directive. This semi-colon is new with 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 3.1`**.

NOTE: Some obsolete or re-designed features *are totally unavailable in POV-Ray 3.1 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 3.1 syntax as much as possible.

4.2.6 Conditional Directives

POV-Ray 3.0 allows a variety of new 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.

4.2.6.1 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 that 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.

4.2.6.2 The #ifdef and #ifndef Directives

The **#ifdef** and **#ifndef** directive are similar to the **#if** directive however they is 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.

4.2.6.3 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 Switch* 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
```

4.2.6.4 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 that 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.

4.2.7 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.

4.2.7.1 Text Message Streams

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

TEXT_STREAM_DIRECTIVE:
#debug *STRING* |
#error *STRING* |
#render *STRING* |
#statistics *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)
  #render "Clock in 0 to 180 range\n"
 #break
 #range (180,360)
  #render "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 five 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.

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.

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.

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.

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 banner and status streams can not be accessed by the user.

4.2.7.2 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
`"\v"`	Vertical tab	0x0B
`"\0"`	Null	0x00
`"\\"`	Backslash	0x5C
`"\'"`	Single quote	0x27
`"\""`	Double quote	0x22

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.

Note that most of these control characters only apply in text message directives and **#write** directives which write strings They are not implemented for other string usage in POV-Ray such as text objects or file names.

4.2.8 User Defined Macros

New in POV-Ray 3.1 are user defined macros with parameters. This new feature, along with the ability to declare **#local** variables, turns the POV-Ray Language into a fully functional programming language. It should now be possible to write scene generation utilities that previously required external utilities.

4.2.8.1 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 my the **#end** directive. Note however that 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. Also you may not nest macro declarations.

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".

4.2.8.2 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.

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,Hoiz,-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.

4.2.8.3 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.

4.2.8.4 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. This is especially true because the POV-Ray language has no user-defined functions or procedures. 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. Note 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.

4.2.8.5 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 verses 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.

4.3 POV-Ray Coordinate System

Objects, lights and the camera are positioned using a typical 3D coordinate system. The usual coordinate system for POV-Ray has the positive y-axis pointing up, the positive x-axis pointing to the right and the positive z-axis pointing into the screen. The negative values of the axes point the other direction as shown in the images in section "Understanding POV-Ray's Coordinate System".

Locations within that coordinate system are usually specified by a three component vector. The three values correspond to the x, y and z directions respectively. For example, the vector **<1,2,3>** means the point that's one unit to the right, two units up and three units in front of the center of the universe at **<0,0,0>**.

Vectors are not always points though. They can also refer to an amount to size, move or rotate a scene element or to modify the texture pattern applied to an object.

The size, location, orientation, and deformation of items within the coordinate system is controlled by modifiers called *transformations*. The follow sub-sections describe the transformations and their usage.

4.3.1 Transformations

The supported transformations are **rotate**, **scale**, and **translate**. They are used to turn, size and move an object or texture. A transformation matrix may also be used to specify complex transformations directly. Groups of transformations may be merged together and stored in a transformation identifier. The syntax for transformations is as follows.

TRANSFORMATION:
rotate <*Rotate_Amt*> |
scale <*Scale_Amt*> |
translate <*Translate_Amt*> |
transform *TRANSFORM_IDENTIFIER* |
matrix <*Val00***,** *Val01***,** *Val02***,**
*Val10***,** *Val11***,** *Val12***,**
*Val20***,** *Val21***,** *Val22***,**
*Val30***,** *Val31***,** *Val32***>**

TRANSFORM_DECLARATION:
#declare *IDENTIFIER* **= transform{** *TRANSFORMATION...* **}** |
#local *IDENTIFIER* **= transform{** *TRANSFORMATION...* **}**

4.3.1.1 Translate

Items may be moved by adding a **translate** modifier. It consists of the keyword **translate** followed by a vector expression. The three terms of the vector specify the number of units to move in each of the x, y and z directions. Translate moves the element relative to it's current position. For example

```
sphere { <10, 10, 10>, 1
 pigment { Green }
 translate <-5, 2, 1>
}
```

will move the sphere from the location **<10,10,10>** to **<5,12,11>**. It does not move it to the absolute location **<-5,2,1>**. Translations are always relative to the item's location before the move. Translating by zero will leave the element unchanged on that axis. For example:

```
sphere { <10, 10, 10>, 1
 pigment { Green }
 translate 3*x // evaluates to <3,0,0> so move 3 units
        // in the x direction and none along y or z
}
```

4.3.1.2 Scale

You may change the size of an object or texture pattern by adding a **scale** modifier. It consists of the keyword **scale** followed by a vector expression. The three terms of the vector specify the amount of scaling in each of the x, y and z directions.

Uneven scaling is used to *stretch* or *squish* an element. Values larger than one stretch the element on that axis while values smaller than one are used to squish it. Scale is relative to the current element size. If the element has been previously re-sized using scale then scale will size relative to the new size. Multiple scale values may used.

For example

```
sphere { <0,0,0>, 1
 scale <2,1,0.5>
}
```

will stretch and smash the sphere into an ellipsoid shape that is twice the original size along the x-direction, remains the same size in the y-direction and is half the original size in the z-direction.

If a lone float expression is specified it is promoted to a three component vector whose terms are all the same. Thus the item is uniformly scaled by the same amount in all directions. For example:

```
object {
 MyObject
 scale 5 // Evaluates as <5,5,5> so uniformly scale
     // by 5 in every direction.
}
```

4.3.1.3 Rotate

You may change the orientation of an object or texture pattern by adding a **`rotate`** modifier. It consists of the keyword **`rotate`** followed by a vector expression. The three terms of the vector specify the number of degrees to rotate about each of the x-, y- and z-axes.

Note that the order of the rotations does matter. Rotations occur about the x-axis first, then the y-axis, then the z-axis. If you are not sure if this is what you want then you should only rotate on one axis at a time using multiple rotation statements to get a correct rotation. As in

```
rotate <0, 30, 0>  // 30 degrees around Y axis then,
rotate <-20, 0, 0> // -20 degrees around X axis then,
rotate <0, 0, 10>  // 10 degrees around Z axis.
```

Rotation is always performed relative to the axis. Thus if an object is some distance from the axis of rotation it will not only rotate but it will *orbit* about the axis as though it was swinging around on an invisible string.

POV-Ray uses a left-handed rotation system. Using the famous "*Computer Graphics Aerobics*" exercise, you hold up your left hand and point your thumb in the positive direction of the axis of rotation. Your fingers will curl in the positive direction of rotation. Similarly if you point your thumb in the negative direction of the axis your fingers will curl in the negative direction of rotation. See "Understanding POV-Ray's Coordinate System" for a illustration.

4.3.1.4 Matrix Keyword

The **`matrix`** keyword can be used to explicitly specify the transformation matrix to be used for objects or textures. Its syntax is:

MATRIX:
`matrix <`*Val00*`,` *Val01*`,` *Val02*`,`
Val10`,` *Val11*`,` *Val12*`,`
Val20`,` *Val21*`,` *Val22*`,`
Val30`,` *Val31*`,` *Val32***`>`**

Where *Val00* through *Val32* are float expressions enclosed in angle brackets and separated by commas. Note this is not a vector. It is a set of 12 float expressions. These floats specify the elements of a 4 by 4 matrix with the fourth column implicitly set to **`<0,0,0,1>`**. At any given point *P*, *P*=<*px*, *py*, *pz*>, is transformed into the point *Q*, *Q*=<*qx*, *qy*, *qz*> by

*qx = Val00 * px + Val10 * py + Val20 * pz + Val30*
*qy = Val01 * px + Val11 * py + Val21 * pz + Val31*
*qz = Val02 * px + Val12 * py + Val22 * pz + Val32*

Normally you won't use the matrix keyword because it's less descriptive than the transformation commands and harder to visualize. However the matrix command allows more general transformation effects like *shearing*. The following matrix causes an object to be sheared along the y-axis.

```
object {
 MyObject
 matrix < 1, 1, 0,
      0, 1, 0,
      0, 0, 1,
      0, 0, 0 >
}
```

4.3.2 Transformation Order

Because rotations are always relative to the axis and scaling is relative to the origin, you will generally want to create an object at the origin and scale and rotate it first. Then you may translate it into its proper position. It is a common mistake to carefully position an object and then to decide to rotate it. However because a rotation of an object causes it to orbit about the axis, the position of the object may change so much that it orbits out of the field of view of the camera!

Similarly scaling after translation also moves an object unexpectedly. If you scale after you translate the scale will multiply the translate amount. For example

```
translate <5, 6, 7>
scale 4
```

will translate to **<20,24,28>** instead of **<5,6,7>**. Be careful when transforming to get the order correct for your purposes.

4.3.3 Transform Identifiers

At times it is useful to combine together several transformations and apply them in multiple places. A transform identifier may be used for this purpose. Transform identifiers are declared as follows:

TRANSFORM_DECLARATION:
#declare *IDENTIFIER* **= transform{** *TRANSFORMATION...* **}** |
#local *IDENTIFIER* **= transform{** *TRANSFORMATION...* **}**

Where *IDENTIFIER* is the name of the identifier up to 40 characters long and *TRANSFORMATION* is any valid transformation modifier. See "#declare vs. #local" for information on identifier scope. Here is an example...

```
#declare MyTrans = transform {
  rotate ThisWay
  scale SoMuch
  rotate -ThisWay
  scale Bigger
  translate OverThere
  rotate WayAround
 }
```

A transform identifier is invoked by the **transform** keyword without any brackets as shown here:

```
object {
 MyObject      // Get a copy of MyObject
 transform MyTrans // Apply the transformation
 translate -x*5   // Then move it 5 units left
```

```
}
object {
 MyObject      // Get another copy of MyObject
 transform MyTrans // Apply the same transformation
 translate x*5   // Then move this one 5 units right
}
```

On extremely complex CSG objects with lots of components it may speed up parsing if you apply a declared transformation rather than the individual **`translate`**, **`rotate`**, **`scale`**, or **`matrix`** modifiers. The **`transform`** is attached just once to each component. Applying each individual **`translate`**, **`rotate`**, **`scale`**, or **`matrix`** modifiers takes longer. This only affects parsing - rendering works the same either way.

4.3.4 Transforming Textures and Objects

When an object is transformed all textures attached to the object *at that time* are transformed as well. This means that if you have a **`translate`**, **`rotate`**, **`scale`**, or **`matrix`** modifier in an object *before* a texture, then the texture will not be transformed. If the transformation is *after* the texture then the texture will be transformed with the object. If the transformation is *inside* the **`texture`** statement then *only the texture* is affected. The shape remains the same. For example:

```
sphere { 0, 1
 texture { Jade } // texture identifier from TEXTURES.INC
 scale 3      // this scale affects both the
          // shape and texture
}
sphere { 0, 1
 scale 3      // this scale affects the shape only
 texture { Jade }
}
sphere { 0, 1
 texture {
  Jade
  scale 3     // this scale affects the texture only
 }
}
```

Transformations may also be independently applied to pigment patterns and surface normal patterns. Note that scaling a normal pattern affects only the width and spacing. It does not affect the apparent height or depth of the bumps. For example:

```
box { <0, 0, 0>, <1, 1, 1>
 texture {
  pigment {
   checker Red, White
   scale 0.25 // This affects only the color pattern
  }
  normal {
   bumps 0.3 // This specifies apparent height of bumps
   scale 0.2 // Scales diameter and space between bumps
         // but not the height. Has no effect on
         // color pattern.
  }
  rotate y*45 // This affects the entire texture but
 }       // not the object.
}
```

4.4 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*

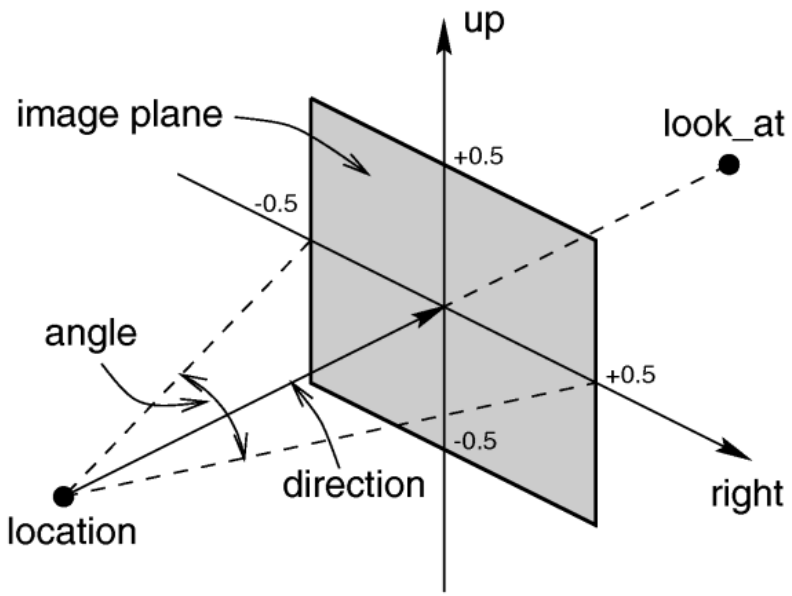
CAMERA_VECTOR:
location <*Location*> | **right** <*Right*> | **up** <*Up*> | **direction** <*Direction*> |
sky <*Sky*>

CAMERA_MODIFIER:
angle *Degrees* | **look_at** <*Look_At*> |
blur_samples *Num_of_Samples* | **aperture** *Size* | **focal_point** <*Point*> |
confidence *Blur_Confidence* | **varience** *Blur_Varience* |
NORMAL |
TRANSFORMATION

Depending on the projection type some of the parameters are required, some are optional and some aren't used. If no projection type is given the perspective camera will be used (pinhole camera). If no camera is specified a default camera is used. *CAMERA_ITEM*s may legally appear in any order but the order of some items is critical to the proper functioning of the camera. Follow the guidelines in this document closely because POV-Ray will not stop you from making mistakes.

4.4.1 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.



The perspective camera.

4.4.1.1 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, 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.

4.4.1.2 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. Note that the **sky** vector must appear before the **look_at** vector.

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>**.

4.4.1.3 Angle

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.

4.4.1.4 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.

4.4.1.5 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 **up** and **right** so you should always specify them before **look_at**. Also the **angle** calculation depends on the **right** vector so **right** should precede it.

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. If you specify the 3 vectors as initially perpendicular and do not explicitly re-specify the after any **look_at** or **rotate** vectors, the everything will work fine.

4.4.1.5.1 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 that 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.

4.4.1.5.2 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.

4.4.1.6 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.

4.4.2 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. In general the *CAMERA_TYPE* should be the *first* item in a **camera** statement. If none is specified, the **perspective** camera is the default.

Perspective projection: The **perspective** 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: This projection uses parallel camera rays to create an image of the scene. The size of the image is determined by the lengths of the **right** and **up** vectors.

If you add the **orthographic** keyword after all other parameters of a perspective camera you'll get an orthographic view with the same image area, i.e. the size of the image is the same. In this case you needn't specify the lengths of the **right** and **up** vector because they'll be calculated automatically. 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.

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

You should note that the vista buffer can only be used with the perspective and orthographic camera.

4.4.3 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 that, 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. Objects close to this point are in focus and those farther from that point 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).

Larger **confidence** values will lead to more samples, slower traces and better images. The same holds for smaller **variance** thresholds.

4.4.4 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.

4.4.5 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
  angle 3
 }
#declare Short_Lens =
 camera {
  location -z*50
  angle 15
 }
camera {
 Long_Lens  // edit this line to change lenses
 look_at Here
```

```
}
```

4.5 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 follow, we describe "Finite Solid Primitives", "Finite Patch Primitives", "Infinite Solid Primitives", and "Light Sources". These primitive shapes may be combined into complex shapes using "Constructive Solid Geometry" or 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, pigment, normal, finish, interior, bounding, clipping or transformations. Specifically the syntax is:

OBJECT:
FINITE_SOLID_OBJECT | *FINITE_PATCH_OBJECT* |
INFINITE_SOLID_OBJECT | *CSG_OBJECT* | *LIGHT_SOURCE* |
object { *OBJECT_IDENTIFIER [OBJECT_MODIFIERS...]* **}**

FINITE_SOLID_OBJECT:
BLOB | *BOX* | *CONE* | *CYLINDER* | *HEIGHT_FIELD* | *JULIA_FRACTAL* |
LATHE | *PRISM* | *SPHERE* | *SUPERELLIPSOID* | *SOR* | *TEXT* | *TORUS*

FINITE_PATCH_OBJECT:
BICUBIC_PATCH | *DISC* | *MESH* | *POLYGON* | *TRIANGLE* | *SMOOTH_TRIANGLE*

INFINITE_SOLID_OBJECT:
PLANE | *POLY* | *CUBIC* | *QUARTIC* | *QUADRIC*

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. Note that 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. 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 new feature 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.

4.5.1 Finite Solid Primitives

There are thirteen different solid finite primitive shapes: blob, box, cone, cylinder, height field, Julia fractal, lathe, prisms, sphere, 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.

4.5.1.1 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 like that.

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

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 that 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. What happens is that 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. By using the optional keyword **hierarchy** followed by an optional boolean float value to 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 it 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 {
 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:

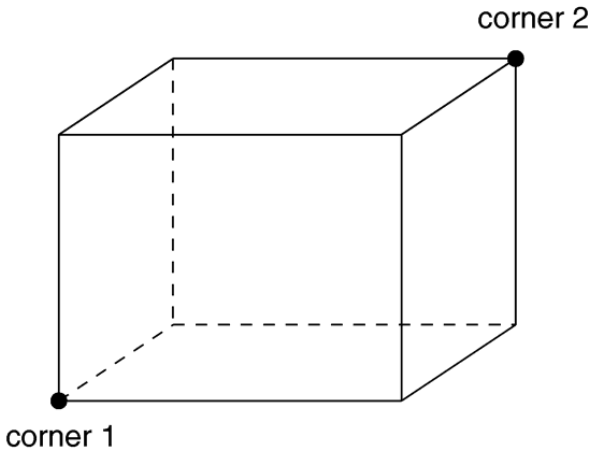
$$density = strength * \left(1 - \left(\frac{distance}{radius}\right)^2\right)^2$$

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.

4.5.1.2 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...]***}**



The geometry of a box.

Where *<Corner_1>* and *<Corner_2>* are vectors defining the x, y, z coordinates of the opposite corners of the box.

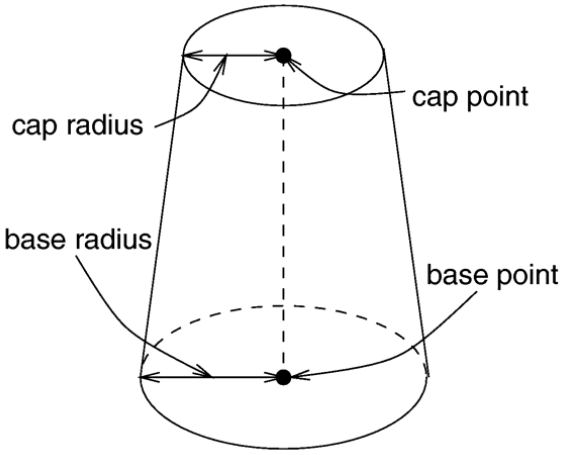
Note that all boxes are defined with their faces parallel to the coordinate axes. They may later be rotated to any orientation using the **rotate** keyword.

Boxes are calculated efficiently and make good bounding shapes (if manually bounding seems to be necessary).

4.5.1.3 Cone

The **cone** statement creates a finite length cone or a *frustum* (a cone with the point cut off). The syntax is:

CONE:
cone { *<Base_Point>*, *Base_Radius*, *<Cap_Point>*, *Cap_Radius*
[**open** *][OBJECT_MODIFIERS...]*
}



The geometry of a cone.

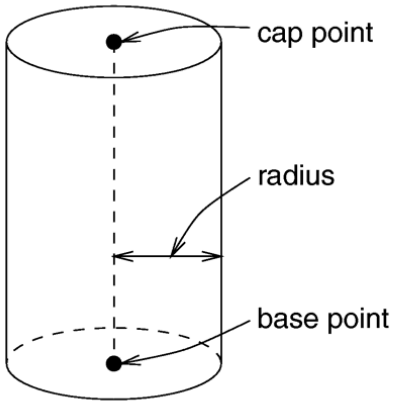
Where <*Base_Point*> and < *Cap_Point*> are vectors defining the x, y, z coordinates of the center of the cone's base and cap and *Base_Radius* and *Cap_Radius* are float values for the corresponding radii.

Normally the ends of a cone are closed by flat planes which 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.

4.5.1.4 Cylinder

The **`cylinder`** statement creates finite length cylinder with parallel end caps The syntax is:

CYLINDER:
`cylinder{` <*Base_Point*>**`,`** <*Cap_Point*>**`,`** *Radius*
[**`open`** *][OBJECT_MODIFIERS...]*
`}`



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 planes which 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.

4.5.1.5 Height Field

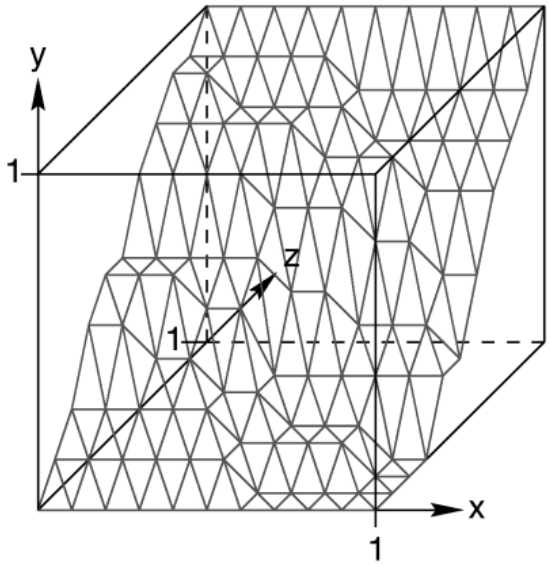
Height fields are fast, efficient objects that are generally used to create mountains or other raised surfaces out of hundreds of triangles in a mesh. The **height_field** statement syntax is:

HEIGHT_FIELD:
height_field{ *HF_TYPE "filename" [HF_MODIFIER...]* **}**

HF_TYPE:
gif | **tga** | **pot** | **png** | **pgm** | **ppm** | **sys**

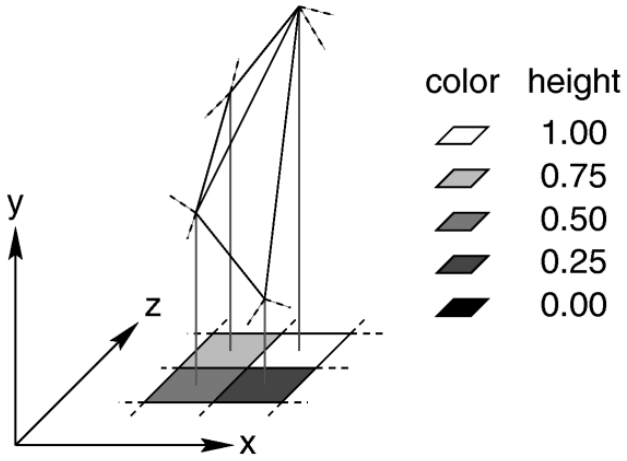
HF_MODIFIER:
hierarchy *[Boolean]* |
smooth *[Boolean]* |
water_level *Level* |
OBJECT_MODIFIER

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.



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.



Four pixels of an image and the resulting heights and triangles in the 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 six or possibly seven 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`**, and possibly **`sys`** which is a system specific (e. g. Windows BMP or Macintosh Pict) format file. The GIF, PNG, PGM, 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 like GIF that uses a color palette 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 that 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. You may optionally use a boolean value such as **`smooth on`** or **`smooth off`**.

In order to speed up the intersection tests an 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`**.

4.5.1.6 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:

sqr | **cube** | **exp** | **reciprocal** | **sin** | **asin** |
sinh | **asinh** | **cos** | **acos** | **cosh** | **acosh** |
tan | **atan** | **tanh** | **atanh** | **log** | **pwr(** *X_Val*, *Y_Val* **)**

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 4D 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.

Quaternion basis vector multiplication rules:		
ij = k	**jk = i**	**ki = j**
ji = -k	**kj = -i**	**ik = -j**
ii = jj = kk = -1	**ijk = -1**	

Hypercomplex basis vector multiplication rules:		
ij = k	**jk = -i**	**ki = -j**
ji = k	**kj = -i**	**ik = -j**
ii = jj = kk = -1	**ijk = 1**	

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:
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*
log	natural logarithm of *h*
pwr(x,y)	*h* raised to the complex power *x*+**i***y*

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
}
```

The first renderings of julia fractals using quaternions were done by Alan Norton and later by John Hart in the '80's. This new 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 $z^2 + c$ formula. With an extra two dimensions and eighteen functions to work with, intrepid explorers should be able to locate some new fractal beasties in hyperspace, so have at it!

4.5.1.7 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*

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 by your own if you want a closed curve. The curve thus defined is rotated about the y-axis to form the lathe object which is 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 that 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 that 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 location 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 overcome the transition problem of quadratic splines because they also take the 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, 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.

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.

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 with the quadratic spline lathe 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. In case of cubic or bezier splines, the Sturmian root solver is always used because a 6th order polynomial has to be solved.

4.5.1.8 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
```

The first items specify the spline type and sweep type. The defaults if none is specified is **linear_spline** and **conic_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 **quadric_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>, <6,8>, <0,8>, <0,0>,  //outer rim
     //Control is <1,-1> and <0,0> is first & last point
 <2,0>, <1,1>, <5,1>, <5,7>, <1,7>, <1,1>   //inner rim
     //Control is <2,0> and <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>, <6,8>, <0,8>, <0,0>, <-1,1>, //outer rim
 //First control is <1,-1> and <0,0> is first & last point
 // Last control of first spline is <-1,1>
 <2,0>, <1,1>, <5,1>, <5,7>, <1,7>, <1,1>, <0,2> //inner rim
 //First control is <2,0> and <1,1> is first & last point
 // 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.

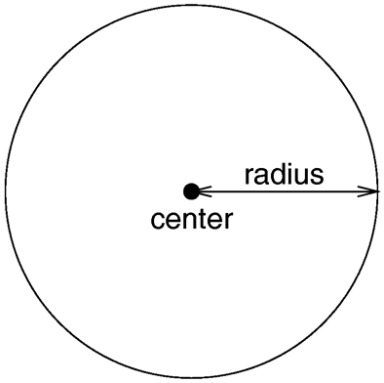
For an explanation of the spline concept read the description of the "Lathe" object. Also see the tutorials on "Lathe Object" and "Prism Object".

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.

4.5.1.9 Sphere

The syntax of the **sphere** object is:

SPHERE:
sphere { *<Center>, Radius [OBJECT_MODIFIERS...]* **}**



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).

4.5.1.10 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:

$$f(x,y,z)=\left(\left|x\right|^{\left(\frac{2}{e}\right)}+\left|y\right|^{\left(\frac{2}{e}\right)}\right)^{\left(\frac{e}{n}\right)}+\left|z\right|^{\left(\frac{2}{n}\right)}-1=0$$

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).

4.5.1.11 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*

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.

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 anymore 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 of 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

$$r^2 = f(h) = A \cdot h^3 + B \cdot h^2 + C \cdot h + D$$

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

$$b = M \cdot x, with:$$

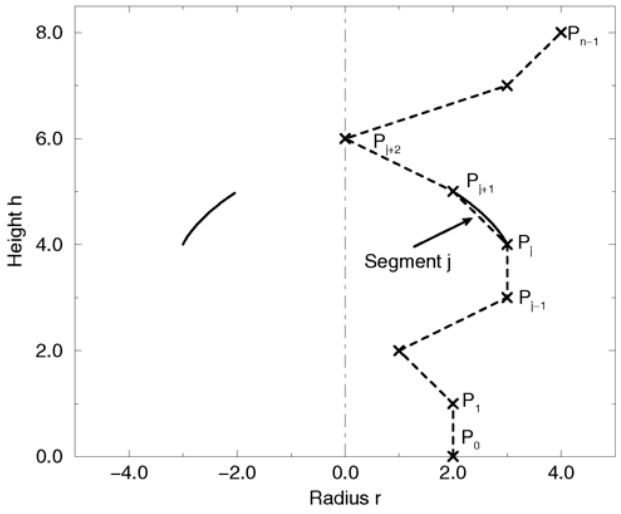
$$b = \begin{bmatrix} r(j)^2 \\ r(j+1)^2 \\ \dfrac{2 \cdot r(j) \cdot (r(j+1) - r(j-1))}{h(j+1) - h(j-1)} \\ \dfrac{2 \cdot r(j+1) \cdot (r(j+2) - r(j))}{h(j+2) - h(j)} \end{bmatrix}$$

$$M = \begin{bmatrix} h(j)^3 & h(j)^2 & h(j) & 1 \\ h(j+1)^3 & h(j+1)^2 & h(j+1) & 1 \\ 3 \cdot h(j)^2 & 2 \cdot h(j) & 1 & 0 \\ 3 \cdot h(j+1)^2 & 2 \cdot h(j+1) & 1 & 0 \end{bmatrix}$$

$$x = \begin{bmatrix} A(j) \\ B(j) \\ C(j) \\ D(j) \end{bmatrix}$$

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.



Segment j of n-3 segments in a point configuration of n points.

The points describe the curve of a surface of revolution.'

4.5.1.12 Text

A **`text`** object creates 3-D text as an extruded block letter. Currently only TrueType fonts are supported but the syntax allows for other font types to be added in the future. The syntax is:

TEXT_OBECT:
`text { ttf` *"fontname.ttf" "String_of_Text" Thickness*`,` *<Offset> [OBJECT_MODIFIERS...]* **`}`**

Where `fontname.ttf` 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 decenders 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 3.1" 1, 0
  pigment { Red }
}
```

Only printable characters are allowed in text objects. Characters such as return, line feed, tabs, backspace etc. are not supported.

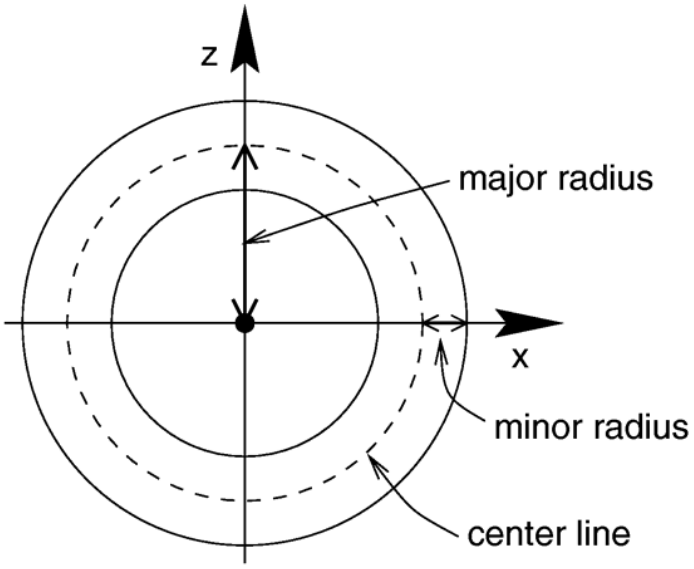
4.5.1.13 Torus

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*

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.



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.

4.5.2 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. They may be combined in CSG union but cannot be use 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.

4.5.2.1 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*

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. All but **flatness** are 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. 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.

4.5.2.2 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...]* **}**

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.

4.5.2.3 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> [MESH_TEXTURE]* **}** |
smooth_triangle {
<Corner_1>, <Normal_1>,
<Corner_2>, <Normal_2>,
<Corner_3>, <Normal_3>
[MESH_TEXTURE]
}

MESH_TEXTURE:
texture { *TEXTURE_IDENTIFIER* **}**

MESH_MODIFIER:
hierarchy *[Boolean]* | *OBJECT_MODIFIER*

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'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 hundred copies of an 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.

4.5.2.4 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 imagemap 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
}
```

4.5.2.5 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.

4.5.3 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.

4.5.3.1 Plane

The **plane** primitive is a simple way to define an infinite flat surface. 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. 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.

4.5.3.2 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*

where *Order* is an integer number from 2 to 7 inclusively that specifies the order of the equation. *A1, A2, ... An* are float values for the coefficients of the equation. There are *m* 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. Remember **cubic** is actually a 3rd order polynomial and **quartic** is 4th order.

	2nd	3rd	4th	5th	6th	7th
A_1	x^2	x^3	x^4	x^5	x^6	x^7
A_2	xy	x^2y	x^3y	x^4y	x^5y	x^6y
A_3	xz	x^2z	x^3z	x^4z	x^5z	x^5z
A_4	x	x^2	x^3	x^4	x^5	x^5
A_5	y^2	xy^2	x^2y^2	x^3y^2	x^4y^2	x^5y^2
A_6	yz	xyz	x^2yz	x^3yz	x^4yz	x^5yz
A_7	y	xy	x^2y	x^3y	x^4y	x^5y
A_8	z^2	xz^2	x^2z^2	x^3z^2	x^4z^2	x^5z^2
A_9	z	xz	x^2z	x^3z	x^4z	x^5z
A_{10}	1	x	x^2	x^3	x^4	x^5
A_{11}		y^3	xy^3	x^2y^3	x^3y^3	x^4y^3
A_{12}		y^2z	xy^2z	x^2y^2z	x^3y^2z	x^4y^2z
A_{13}		y^2	xy^2	x^2y^2	x^3y^2	x^4y^2
A_{14}		yz^2	xyz^2	x^2yz^2	x^3yz^2	x^4yz^2
A_{15}		yz	xyz	x^2yz	x^3yz	x^4yz
A_{16}		y	xy	x^2y	x^3y	x^4y
A_{17}		z^3	xz^3	x^2z^3	x^3z^3	x^4z^3
A_{18}		z^2	xz^2	x^2z^2	x^3z^2	x^4z^2
A_{19}		z	xz	x^2z	x^3z	x^4z
A_{20}		1	x	x^2	x^3	x^4
A_{21}			y^4	xy^4	x^2y^4	x^3y^4
A_{22}			y^3z	xy^3z	x^2y^3z	x^3y^3z
A_{23}			y^3	xy^3	x^2y^3	x^3y^3
A_{24}			y^2z^2	xy^2z^2	$x^2y^2z^2$	$x^3y^2z^2$
A_{25}			y^2z	xy^2z	x^2y^2z	x^3y^2z
A_{26}			y^2	xy^2	x^2y^2	x^3y^2
A_{27}			yz^3	xyz^3	x^2yz^3	x^3yz^3
A_{28}			yz^2	xyz^2	x^2yz^2	x^3yz^2
A_{29}			yz	xyz	x^2yz	x^3yz
A_{30}			y	xy	x^2y	x^3y
A_{31}			z^4	xz^4	x^2z^4	x^3z^4
A_{32}			z^3	xz^3	x^2z^3	x^3z^3
A_{33}			z^2	xz^2	x^2z^2	x^3z^2
A_{34}			z	xz	x^2z	x^3z
A_{35}			1	x	x^2	x^3

	5th	6th	7th
A_{41}	y^3	xy^3	x^2y^3
A_{42}	y^2z^3	xy^2z^3	$x^2y^2z^3$
A_{43}	y^2z^2	xy^2z^2	$x^2y^2z^2$
A_{44}	y^2z	xy^2z	x^2y^2z
A_{45}	y^2	xy^2	x^2y^2
A_{46}	yz^4	xyz^4	x^2yz^4
A_{47}	yz^3	xyz^3	x^2yz^3
A_{48}	yz^2	xyz^2	x^2yz^2
A_{49}	yz	xyz	x^2yz
A_{50}	y	xy	x^2y
A_{51}	z^5	xz^5	x^2z^5
A_{52}	z^4	xz^4	x^2z^4
A_{53}	z^3	xz^3	x^2z^3
A_{54}	z^2	xz^2	x^2z^2
A_{55}	z	xz	x^2z
A_{56}	1	x	x^2
A_{57}		y^6	xy^6
A_{58}		y^5z	xy^5z
A_{59}		y^5	xy^5
A_{60}		y^4z^2	xy^4z^2
A_{61}		y^4z	xy^4z
A_{62}		y^4	xy^4
A_{63}		y^3z^3	xy^3z^3
A_{64}		y^3z^2	xy^3z^2
A_{65}		y^3z	xy^3z
A_{66}		y^3	xy^3
A_{67}		y^2z^4	xy^2z^4
A_{68}		y^2z^3	xy^2z^3
A_{69}		y^2z^2	xy^2z^2
A_{70}		y^2z	xy^2z
A_{71}		y^2	xy^2
A_{72}		yz^5	xyz^5
A_{73}		yz^4	xyz^4
A_{74}		yz^3	xyz^3
A_{75}		yz^2	xyz^2

	6th	7th
A_{81}	z^3	xz^3
A_{82}	z^2	xz^2
A_{83}	z	xz
A_{84}	1	x
A_{85}		y^7
A_{86}		y^6z
A_{87}		y^6
A_{88}		y^5z^2
A_{89}		y^5z
A_{90}		y^5
A_{91}		y^4z^3
A_{92}		y^4z^2
A_{93}		y^4z
A_{94}		y^4
A_{95}		y^3z^4
A_{96}		y^3z^3
A_{97}		y^3z^2
A_{98}		y^3z
A_{99}		y^3
A_{100}		y^2z^5
A_{101}		y^2z^4
A_{102}		y^2z^3
A_{103}		y^2z^2
A_{104}		y^2z
A_{105}		y^2
A_{106}		yz^6
A_{107}		yz^5
A_{108}		yz^4
A_{109}		yz^3
A_{110}		yz^2
A_{111}		yz
A_{112}		y
A_{113}		z^7
A_{114}		z^6
A_{115}		z^5

A_{36}	y^5	xy^5	x^2y^5
A_{37}	y^4z	xy^4z	x^2y^4z
A_{38}	y^4	xy^4	x^2y^4
A_{39}	y^3z^2	xy^3z^2	$x^2y^3z^2$
A_{40}	y^3z	xy^3z	x^2y^3z

A_{76}	yz	xyz
A_{77}	y	xy
A_{78}	z^6	xz^6
A_{79}	z^5	xz^5
A_{80}	z^4	xz^4

A_{116}	z^4
A_{117}	z^3
A_{118}	z^2
A_{119}	z
A_{120}	1

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:

$x^4 + y^4 + z^4 + 2\,x^2\,y^2 + 2\,x^2\,z^2 + 2\,y^2\,z^2 -$
$2\,(r\_0^2 + r\_1^2)\,x^2 + 2\,(r\_0^2 - r\_1^2)\,y^2 -$
$2\,(r\_0^2 + r\_1^2)\,z^2 + (r\_0^2 - r\_1^2)^2 = 0$

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
}
```

Poly, cubic and quartics are just like quadrics in that you don't have to understand what one is 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. If you are interested and mathematically inclined, an excellent reference book for curves and surfaces where you'll find more polynomial shape formulas is:

"The CRC Handbook of Mathematical Curves and Surfaces"
David von Seggern
CRC Press, 1990

4.5.3.3 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 that you 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

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`.

4.5.4 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.

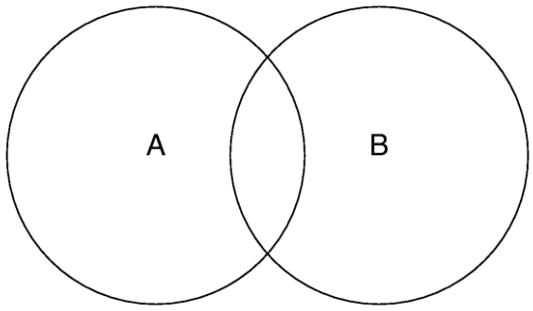
4.5.4.1 Inside and Outside

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 and triangle-based shapes cannot be used as solid objects in CSG since they have no well defined inside and outside.

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**.



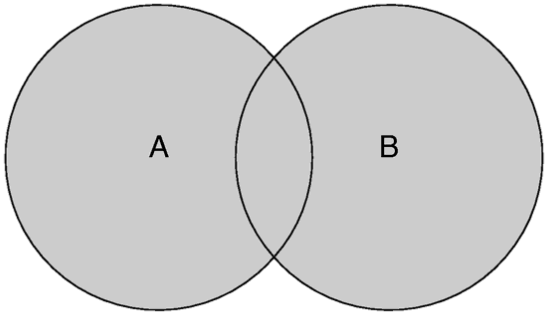
Two overlapping objects.

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".

4.5.4.2 Union



The union of two objects.

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 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.

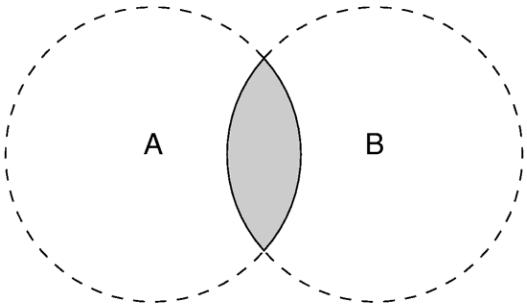
The following union will contain a box and a sphere.

```
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.

4.5.4.3 Intersection

The **intersection** object creates a shape containing only those areas where all components overlap. A point is part an intersection if it is inside both objects, **A** and **B**, as show in the figure below.



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 that if all components do not overlap, the intersection object disappears.

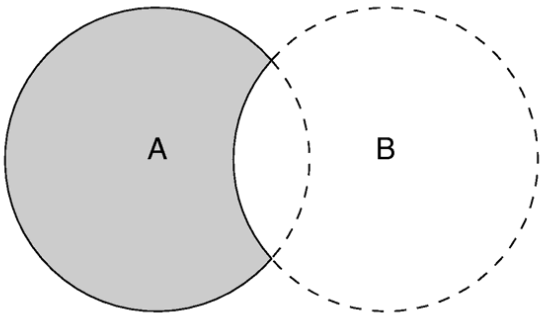
Here is an example that overlaps:

```
intersection {
  box { <-1.5, -1, -1>, <0.5, 1, 1> }
  cylinder { <0.5, 0, -1>, <0.5, 0, 1>, 1 }
}
```

4.5.4.4 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 results is a subtraction of the 2nd shape from the first shape as shown in the figure below.



The difference between two objects.

The syntax is:

DIFFERENCE:
difference { SOLID_*OBJECTS... [OBJECT_MODIFIERS...]* **}**

The component objects must have well defined inside/outside properties. Patch objects are not allowed. Note that if the first object is entirely inside the subtracted objects, the difference object disappears.

Here is an example of a properly formed difference:

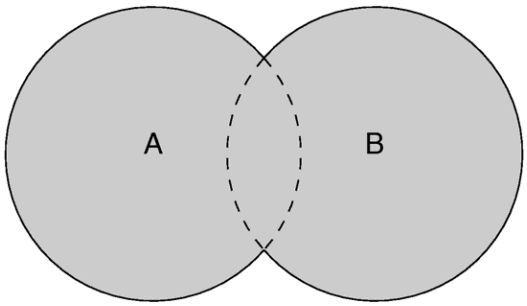
```
difference {
  box { <-1.5, -1, -1>, <0.5, 1, 1> }
  cylinder { <0.5, 0, -1>, <0.5, 0, 1>, 1 }
}
```

Note that 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 }
}
```

4.5.4.5 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.



Merge removes inner surfaces.

The syntax is:

MERGE:
merge { SOLID_*OBJECTS... [OBJECT_MODIFIERS...]* **}**

The component objects must have well defined inside/outside properties. Patch objects are not allowed. Note that **merge** is slower rendering than **union** so it should only be used when it is really necessary.

4.5.5 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**

SPOTLIGHT_ITEM:
radius *Radius* | **falloff** *Falloff* | **tightness** *Tightness* | **point_at** *<Spot>*

AREA_LIGHT_ITEM:
area_light *<Axis_1>, <Axis_2>, Size_1, Size_2* |
adaptive *Adaptive* | **jitter** *Jitter*

GENERAL_LIGHT_MODIFIERS:
looks_like { OBJECT **}** | *TRANSFORMATION*
fade_distance *Fade_Distance* | **fade_power** *Fade_Power* |
media_attenuation *[Bool]* | **media_interaction** *[Bool]*

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 that 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 four mutually exclusive light types. If no *LIGHT_TYPE* is specified it is a point light. The other choices are **spotlight**, **shadowless**, and **cylinder**.

4.5.5.1 Point Lights

The simplest kid 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
}
```

4.5.5.2 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.

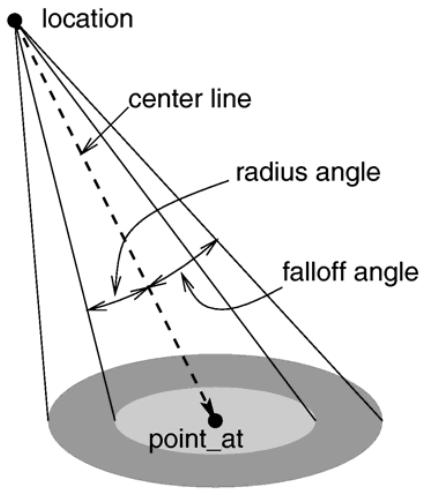
Although the cone of light fades to soft edges, objects illuminated by spotlights still cast hard shadows. The syntax is:

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>*

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.



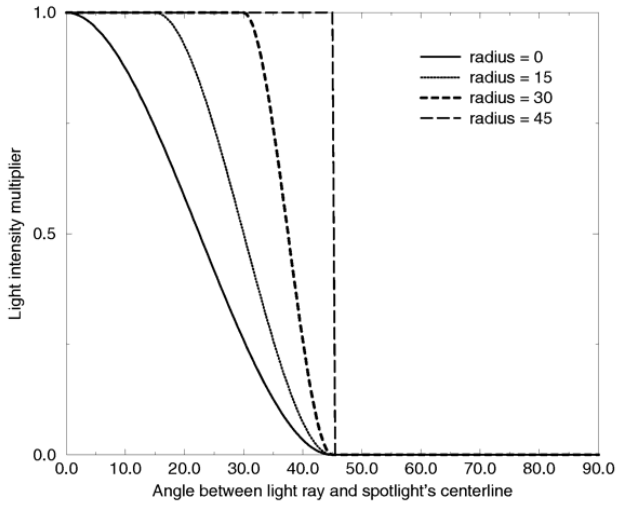
The geometry of a spotlight.

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.

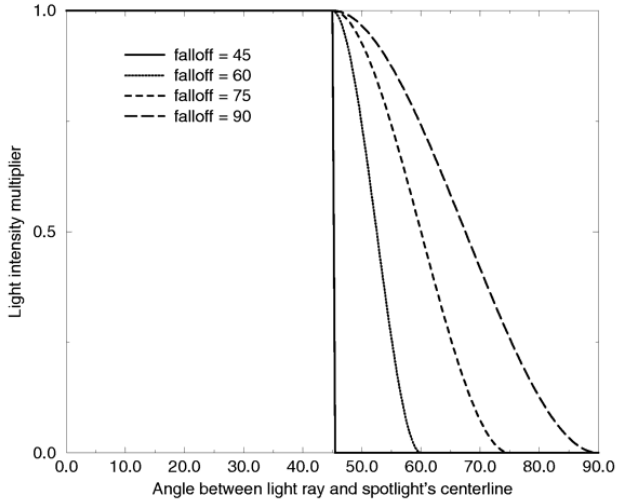
The **falloff** keyword specifies the overall size of the cone of light. This is the point where the light falls off to zero intensity. The float value you specify is the angle, in degrees, between the edge of the cone and center line. The **radius** keyword specifies the size of the "hot-spot" at the 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 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 that if the radius and falloff values are close or equal the light intensity drops rapidly and the spotlight has a sharp edge. The default values for both **radius** and **falloff** is 70.

The values for these two 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).

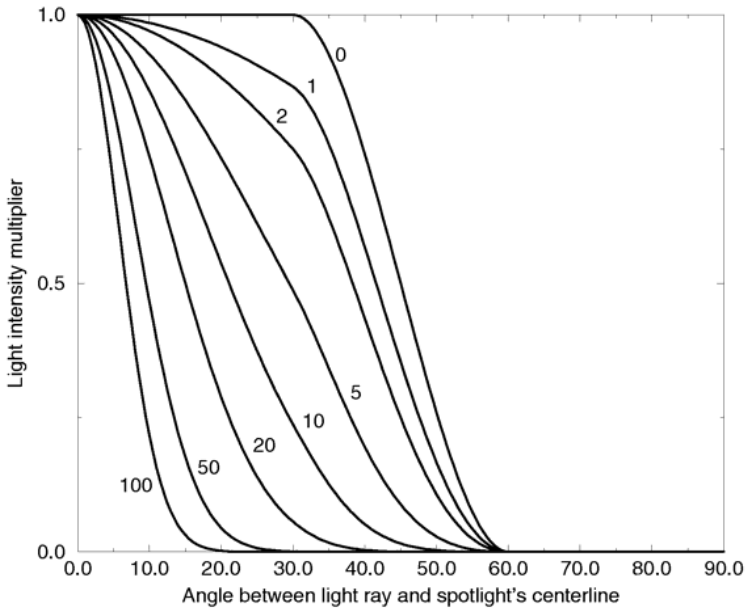


Intensity multiplier curve with a fixed falloff angle of 45 degrees.



Intensity multiplier curve with a fixed radius angle of 45 degrees.

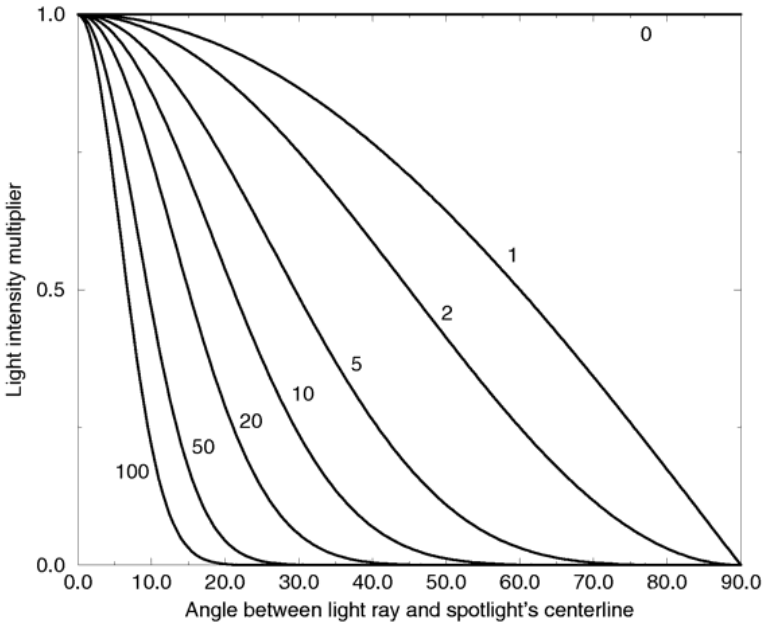
The **`tightness`** keyword is used to specify an *additional* exponential softening of the edges. The intensity of light at an angle from the center line is given by: $intensity * cos(angle)^{tightness}$. The default value for tightness is 10. Lower tightness values will make the spotlight brighter, making the spot wider and the edges sharper. Higher values will dim the spotlight, making the spot tighter and the edges softer. Values from 1 to 100 are acceptable.



Intensity multiplier curve with fixed angle and falloff angles

of 30 and 60 degrees respectively and different tightness values.

You should note from the figures that the radius and falloff angles interact with the tightness parameter. Only negative radius angles will give the tightness value full control over the spotlight's appearance 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.



Intensity multiplier curve with a negative radius angle and different tightness values.

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.

4.5.5.3 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>*

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.

4.5.5.4 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 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>*, *COLOR* **area_light** *<Axis_1>*, *<Axis_2>*, *Size_1*, *Size_2*
*[***adaptive** *Adaptive] [* **jitter** *Jitter]*
[LIGHT_MODIFIERS...]
}

The light's location and color are specified in the same way as a for a regular light source. Any type of light source may be an area light.

The **area_light** command defines the size and orientation of the area light as well as the number of lights in the light source array. The vectors *<Axis_1>* and *<Axis_2>* 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.

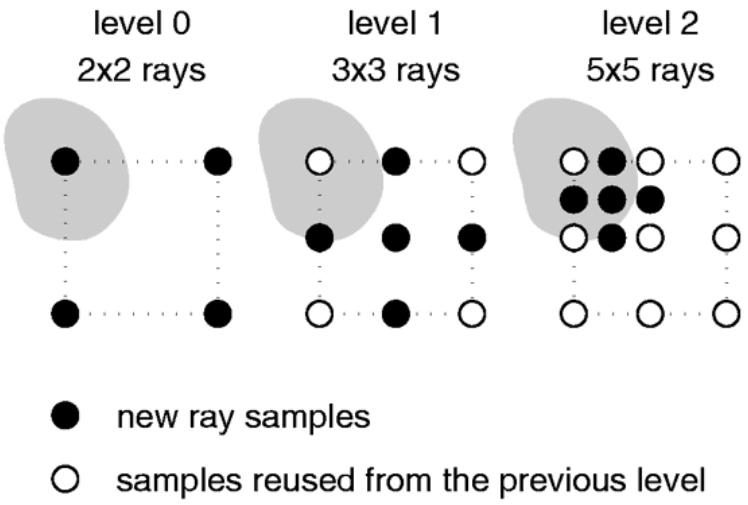
Note that it is possible to specify spotlight parameters along with the area light parameters to create area spotlights. Using area spotlights is a good way to speed up scenes that use area lights since you can confine the lengthy soft shadow calculations to only the parts of your scene that need them.

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 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.



Area light adaptive samples.

While the adaptive sampling method is fast (relatively speaking) it can sometimes produces 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.

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.

4.5.5.5 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 **area_light** but not **spotlight** or **cylinder**.

4.5.5.6 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.

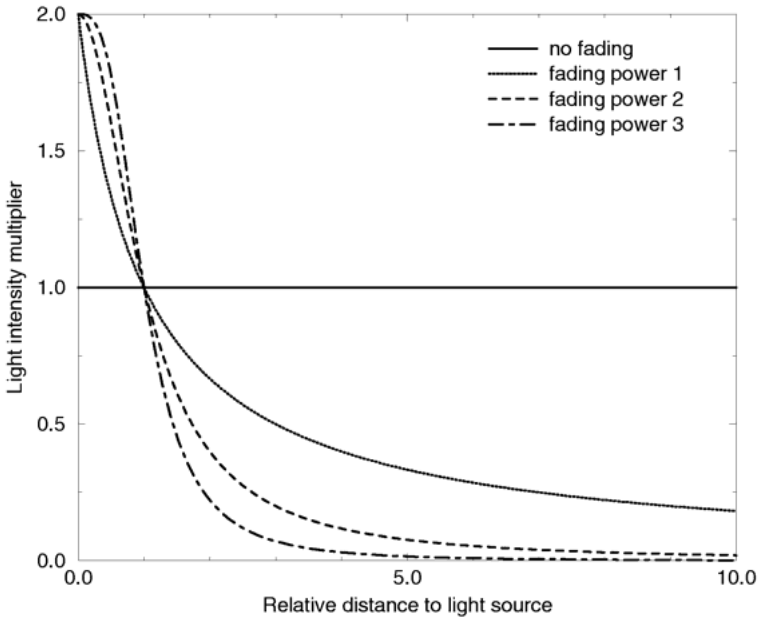
4.5.5.7 Light Fading

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** *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

$$attenuation = \frac{2}{1 + \left(\frac{d}{fade\_distance} \right)^{fade\_power}}$$

with *d* being the distance the light has traveled.



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.

4.5.5.8 Atmospheric Media Interaction

By default light sources will interact with an atmosphere added to the scene. This behaviour can be switched off by using **`media_interaction off`** keyword inside the light source statement. Note in POV-Ray 3.0 this feature was turned off and on with the **`atmosphere`** keyword.

4.5.5.9 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.

4.5.6 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` | **`inverse`** | **`sturm`** *[Bool]* | **`hierarchy`** *[Bool]* |
`interior {` *INTERIOR_ITEMS...* **`}`** |
`texture {` *TEXTURE_BODY* **`}`** | **`pigment {`** *PIGMENT_BODY* **`}`** |
`normal {` *NORMAL_BODY* **`}`** | **`finish {`** *FINISH_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**, **no_shadow**, **hollow**, **inverse**, **sturm**, and **hierarchy**. 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.

4.5.6.1 Clipped_By

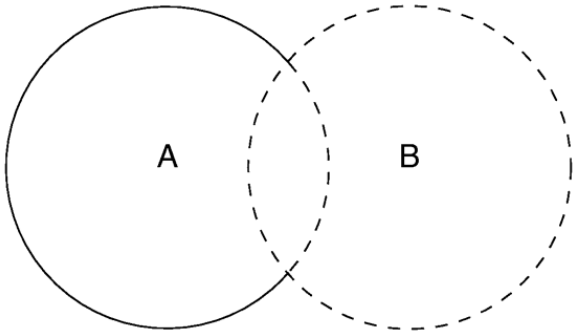
The **clipped_by** statement is technically an object modifier but it provides a type of CSG similar to CSG intersection. The syntax is:

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**.



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 bounds.

4.5.6.2 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 large a 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 may be best to hand bound a merge, especially if it is very complex.

Note that 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 bounds that was used as a clip.

4.5.6.3 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.

Finally the **internal_reflections** and **internal_highlights** keywords depend upon the inside/outside status of an object.

4.5.6.4 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. 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**. The default no **hollow** is specified is **off**.

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 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
}
```

4.5.6.5 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
}
```

4.5.6.6 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`**. The default no **`sturm`** is specified is **`off`**.

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

4.6 Interior

New with 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* |
`fade_distance` *Distance* | **`fade_power`** *Power*
MEDIA...

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. 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 that 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.

4.6.1 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.

4.6.2 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.

4.6.3 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 also that 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 original specified in **finish** but are now properly specified in **interior**. They are accepted in **finish** for backward compatibility and generate a warning message.

4.6.4 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_Power* and **fade_distance** *Fade_Distance* keywords 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. For realistic effects a fade power of 1 to 2 should be used. Default values for both keywords is 0.0 which turns this feature off.

The attenuation is calculated by a formula similar to that used for light source attenuation.

$$attenuation = \frac{1}{1 + \left(\frac{d}{fade\_distance} \right)^{fade\_power}}$$

The **fade_power** and **fade_distance** keywords were original specified in **finish** but are now properly specified in **interior**. They are accepted in **finish** for backward compatibility and generate a warning message.

4.6.5 Faked 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) 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.

4.6.6 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.

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. For example:

```
object{MyObject pigment{rgbt 1.0} interior{media{MyMedia}}} //no shadows
object{MyObject pigment{rgbf 1.0} interior{media{MyMedia}}} //shadows
```

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.

4.7 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 that 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 the 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 that the **tiles** and **material_map** versions of patterned textures are obsolete and are only supported for backwards compatibility.

4.7.1 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}}
```

Note however that 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.

4.7.1.1 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 option 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.

4.7.1.2 Color List Pigments

There are three color list patterns: **checker**, **hexagon** and **brick**. 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...]* **}**

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.

4.7.1.3 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 that 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. Note that 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`** 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}
  }
}
```

4.7.1.4 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.

4.7.1.5 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.

4.7.1.5.1 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 | sys
```
IMAGE_MAP_MOD:
```
   map_type Type | once | interpolate Type |
   filter Palette, Amount | filter all Amount |
   transmit Palette, Amount | transmit all Amount
```

After the required *BITMAP_TYPE* keyword is a string expression containing the name of a bitmapped image file of the specified type. Several optional modifiers may follow the file specification. The modifiers are described below. Note that 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"

4.7.1.5.2 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 that 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.

4.7.1.5.3 Using the Alpha Channel

Another way to specify non-filtered transmit transparency in an image map is by using the *alpha channel*. 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**.

4.7.1.6 Quick Color

When developing POV-Ray scenes its 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.

Note that 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.

4.7.2 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*

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**, 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 a 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}}
```

Note however that 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.

4.7.2.1 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...

The syntax for **slope_map** 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 that 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 vectors 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`**, **`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 }
    }
  }
```

4.7.2.2 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 are 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 that 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.

4.7.2.3 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).

4.7.2.3.1 Specifying a Bump Map

The syntax for an **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** | **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 that 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"

4.7.2.3.2 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
}
```

4.7.2.3.3 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.

4.7.3 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* | **reflection_exponent** *Amount* |
irid { *Irid_Amount [IRID_ITEMS...]* **}** | **crand** *Amount*

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 that 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}}
```

Note however that 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.

4.7.3.1 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.

4.7.3.2 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.

POV-Ray and most other ray-tracers can only simulate directly light which comes directly from actual light sources. Light coming from other objects such as mirrors via specular reflection (such as shining a flashlight onto a mirror for example) cannot be simulated. Neither can we simulate light coming from other objects via diffuse reflections. For example look at some dark area under a desk or in a corner: even though a lamp may not directly illuminate that spot, you can still see a little bit because light comes from diffuse reflection off of nearby objects.

4.7.3.2.1 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**.

4.7.3.2.2 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.

4.7.3.2.3 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** 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 that 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.

4.7.3.3 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 that 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.

4.7.3.3.1 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.

4.7.3.3.2 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 reflects perfectly of a smooth 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.

4.7.3.3.3 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 of the surface.

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.

4.7.3.4 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** keyword in a **finish** statement. For example:

```
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.

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.

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 **reflection_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. We are researching ways to deal with this issue in a more scientific model. The **reflection_exponent** has no effect unless **reflection** is used.

Note that although such reflection is called specular it is not controlled by the **specular** keyword. That keyword controls a specular highlight.

4.7.3.5 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).

4.7.4 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**.

4.7.5 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.

4.7.5.1 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 that 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.

4.7.5.2 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.

4.7.5.3 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.

4.7.5.3.1 Specifying a Material Map

The syntax for an **material_map** is:

MATERIAL_MAP:
```
texture {
   material_map{
      BITMAP_TYPE "bitmap.ext" [BITMAP_MODS...] TEXTURE... [TRANSFORMATIONS...]
   }
}
```
BITMAP_TYPE:
gif | **tga** | **iff** | **ppm** | **pgm** | **png** | **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 that 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 or 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 that 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. Note that 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.

4.7.6 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 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.
  }
}
```

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.

4.7.7 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**.

4.7.7.1 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}
}
```

4.7.7.2 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 "**Error! Reference source not found.**" for more information.

4.7.7.3 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 increases 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.

4.7.7.4 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!

4.7.7.5 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.

4.7.7.6 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 peal 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.

4.7.7.7 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} }
```

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.

4.7.7.8 Crackle

The **crackle** pattern is a set of random tiled polygons. 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.

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).

4.7.7.9 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 increases 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.

4.7.7.10 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 **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*). This is followed by x*y*z unsigned integer bytes of data. The data in the range of 0 to 255 is scaled into a float value in the range 0.0 to 1.0. It remains at 0.0 for all areas beyond the unit cube. The pattern occupies the unit cube regardless of the dimensions in voxels.

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.

See the sample scenes for data file `include\spiral.df3`,and the scenes which use it: `scenes\textures\surfaces\densfile.pov`, `scenes\interior\media\galaxy.pov` for examples.

4.7.7.11 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.

4.7.7.12 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. The pattern reverses for negative values of x. 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 that the comma is required especially if *Bump_Size* is negative.

4.7.7.13 Granite

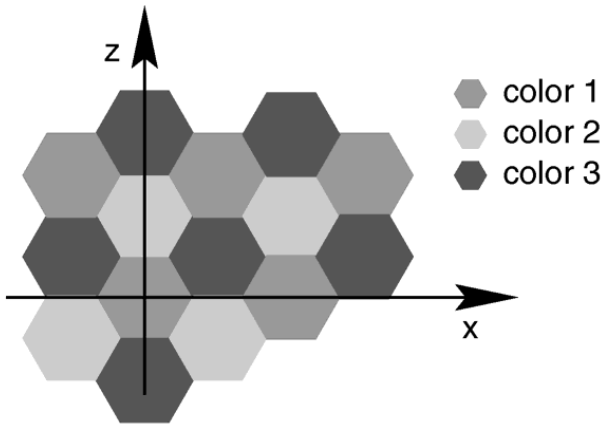
The **granite** pattern uses a simple 1/f fractal noise function to give a good granite pattern. This pattern is used with creative color maps in `stones.inc` to create some gorgeous layered stone textures.

As a normal pattern it creates an extremely bumpy surface that looks like a gravel driveway or rough stone.

4.7.7.14 Hexagon

The **hexagon** pattern is a block pattern that generates a repeating pattern of hexagons in the x-y-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...]* **}**



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 }
}
```

This is a block pattern which cannot use wave types, **`color_map`**, or **`slope_map`** modifiers.

4.7.7.15 Leopard

Leopard creates regular geometric pattern of circular spots. The formula used is:

value = Sqr((sin(x)+sin(y)+sin(z))/3)

4.7.7.16 Mandel

The **`mandel`** pattern computes the standard Mandelbrot fractal pattern and projects it onto the x-y-plane. It uses the x and y coordinates to compute the Mandelbrot set.

It is specified as

`pigment {mandel` *Max_Iteration [PIGMENT_MODIFIERS...]* **`}`**

The pattern is specified with the keyword **`mandel`** followed by an integer number. This number is the maximum number of iterations to be used to compute the set. Typical values range from 10 up to 256 but any positive integer may be used. For example:

```
pigment {
  mandel 25
  color_map {
    [0.0  color Cyan]
    [0.3  color Yellow]
    [0.6  color Magenta]
    [1.0  color Cyan]
  }
  scale .5
}
```

The value passed to the color map is computed by the formula:

value = number_of_iterations / max_iterations

When used as a normal pattern, the syntax is...

normal{mandel *Max_Iteration [, Bump_Size] [NORMAL_MODIFIERS...]* **}**

where the integer *Max_Iteration* is a required parameter but the float *Bump_Size* which follows is optional. Note that the comma is required especially if *Bump_Size* is negative.

4.7.7.17 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.

4.7.7.18 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.

4.7.7.19 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 increases 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.

4.7.7.20 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.

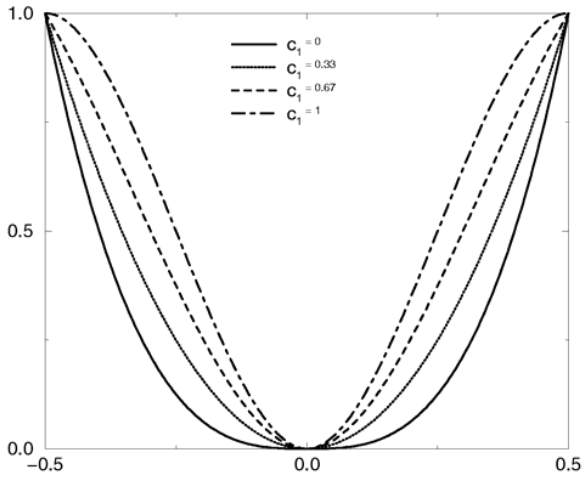
The two parameters **control0** and **control1** are used to adjust the curvature of the *seam* or *gouge* area between the **quilts**. The syntax is:

It is specified as

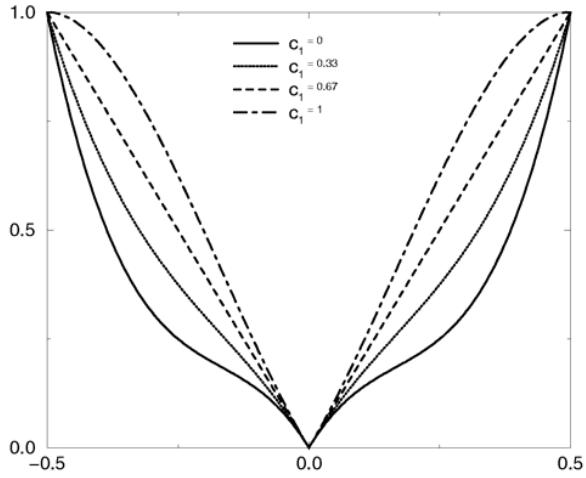
pigment {quilted *[QUILTED_MODIFIERS...]* **}**

QUILTED_MODIFIERS:
control0 *Value_0* | **control** *Value_1* | *PIGMENT_MODIFIERS*

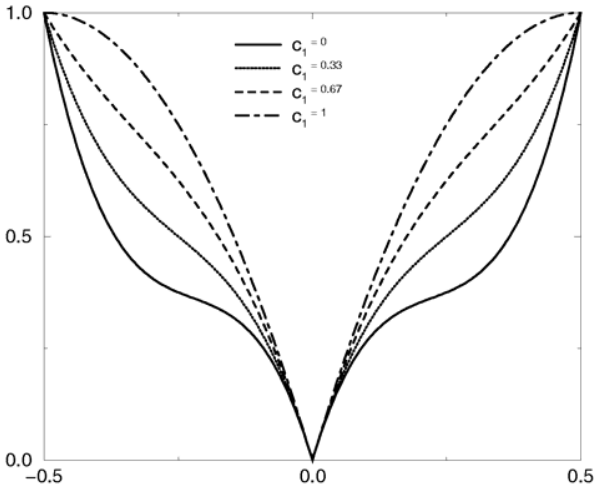
The values should generally be kept to around the 0.0 to 1.0 range. The default value is 1.0 if none is specified. Think of this gouge between the tiles in cross-section as a sloped line.



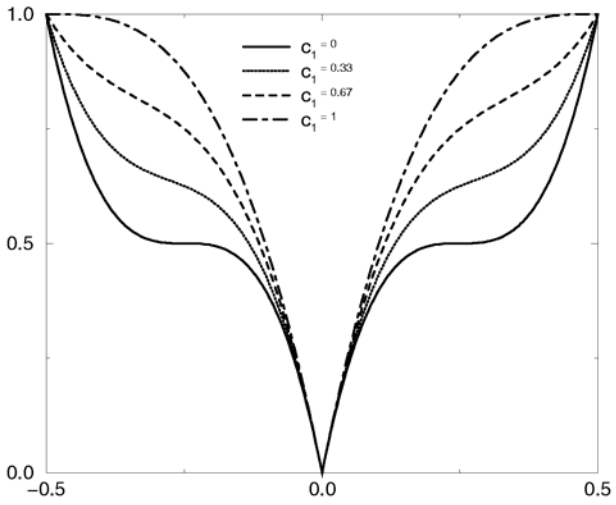
Quilted pattern with c0=0 and different values for c1.



Quilted pattern with c0=0.33 and different values for c1.



Quilted pattern with c0=0.67 and different values for c1.



Quilted pattern with c0=1 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.

4.7.7.21 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:

```
pigement {
  radial color_map{[0.5 Black][0.5 White]}
  frequency 10
}
```

creates 10 white bands and 10 black bands radiating from the y axis.

4.7.7.22 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.

4.7.7.23 Spherical

The **spherical** pattern creates a one unit radius sphere along the Y axis. 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 increases to a max 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.

4.7.7.24 Spiral1

The **spiral1** pattern creates a spiral that winds around the y-axis similar to a screw. When viewed sliced in the X-Z 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 y-axis.

As a normal pattern, the syntax is

normal {spiral1 *Number_of_Arms [, Bump_Size] [NORMAL_MODIFIERS...]* **}**

where the vector <*Orientation*> is a required parameter but the float *Bump_Size* which follows is optional. Note that 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.

4.7.7.25 Spiral2

The **spiral2** pattern creates a double spiral that winds around the y-axis similar **spiral1** except it is two overlapping spirals the twist in opposite directions. The results 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 y-axis.

As a normal pattern, the syntax is

normal {spiral2 *Number_of_Arms [, Bump_Size] [NORMAL_MODIFIERS...]* **}**

where the vector <*Orientation*> is a required parameter but the float *Bump_Size* which follows is optional. Note that 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.

4.7.7.26 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.

4.7.7.27 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.

4.7.7.28 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.

4.7.7.29 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.

4.7.8 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* |
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*

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*

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* **}**

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.

4.7.8.1 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.

4.7.8.2 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.

4.7.8.3 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 grove 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**, 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**.

4.7.8.4 Turbulence

The keyword **turbulence** followed by a float or vector may be used to stir up any **pigment**, **normal**, **texture**, **irid** 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.

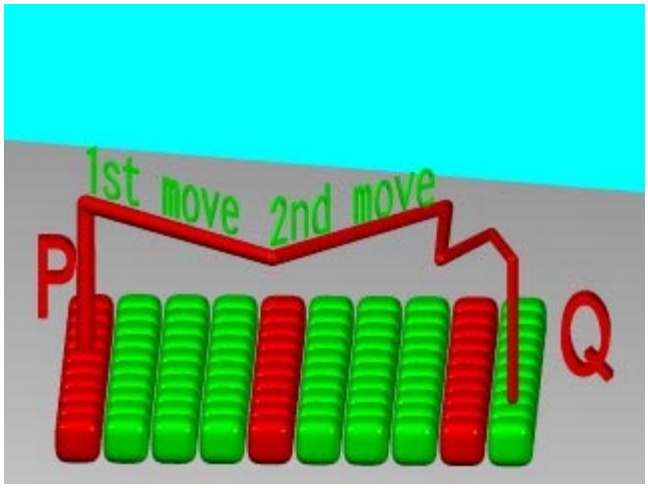
Turbulence uses a random noise function called *DNoise*. This is similar to the noise used in the **bozo** pattern except that instead of giving a single value it gives a direction. You can think of it as the direction that the wind is blowing at that spot. Points close together generate almost the same value but points far apart are randomly different.

In general the order of turbulence parameters relative to other pattern modifiers such as transformations, color maps 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. See section "" for a way to work around this behavior.

In general, the order of turbulence parameters relative to each other and to other pattern modifiers such as transformations or **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. You can also specify turbulence inside **warp** and in this way you can force turbulence to be applied after transformations. See "Warps" for details.

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:



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.

4.7.8.5 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.

4.7.8.6 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.

4.7.8.7 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.

4.7.8.8 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 three types of warps but the syntax was designed to allow future expansion. The first two, the **repeat** warp and the **black_hole** warp are new features for POV-Ray that modify the pattern in geometric ways. The other warp provides an alternative way to specify turbulence.

The syntax for using a **warp** statement is:

WARP:
warp { *WARP_ITEM* **}**

WARP_ITEM:
repeat <*Directiont*> *[REPEAT_ITEMS...]* |
black_hole <*Location*>**,** *Radius [BLACK_HOLE_ITEMS...]* |
turbulence <*Amount*> *[TURB_ITEMS...]*

REPEAT_ITEMS:
offset <*Amount*> | **flip** <*Axis*>

BLACK_HOLE_ITEMS:
strength *Strength* | **falloff** *Amount* | **inverse** | **type** *Type* |
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.

4.7.8.8.1 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 woodgrain. 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 woodgrain 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 woodgrain, it would look to the viewer as if there were a knothole in the wood. In this text we use a woodgrain 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 **inverted** keyword is specified then 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 back 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 co-ordinates. 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
  }
```

4.7.8.8.2 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** <*Directiont*> *[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.

4.7.8.8.3 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 suddenly 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.

4.7.8.9 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.

4.7.8.9.1 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
}
```

4.7.8.9.2 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
    }
  }
}
```

4.7.8.9.3 The interpolate Option

Adding the **interpolate** keyword can smooth the jagged look of a bitmap. When POV-Ray asks an image map color or a bump amount for a bump map, it often asks for a point that is not directly on top of one pixel but sort of between several differently colored pixels. Interpolations returns 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.

4.8 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. 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:
intervals *Number* | **samples** *Min*, *Max* |
confidence *Value* | **variance** *Value* | **ratio** *Value* |
absorption *COLOR* | **emission** *COLOR* |
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* **}**

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 "**Error! Reference source not found.**" 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** statements 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 that 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 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. For example:

```
object{MyObject pigment{rgbt 1.0} interior{media{MyMedia}}} //no shadows
object{MyObject pigment{rgbf 1.0} interior{media{MyMedia}}} //shadows
```

4.8.1 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>
```

4.8.1.1 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.

4.8.1.2 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 to 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.

4.8.1.3 Scattering

The syntax of a **scattering** statement is:

SCATTERING:
scattering { *Type*, *COLOR [* **eccentricity** *Value] [* **extinction** *Value]* **}**

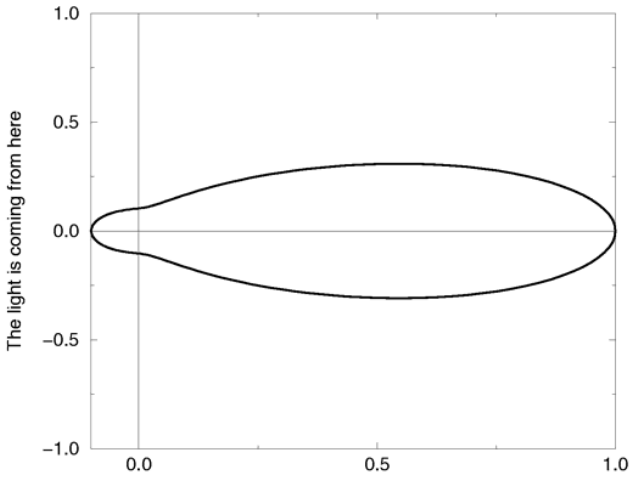
The first float value specifies the type of scattering. This is followed by the color of the scattered light. The default value if no **scattering** statement is given is **rgb<0,0,0>** which means no scattering occurs.

The scattering effect is only visible when light is shining on the media from a light source. This is similar to **difuse** 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.

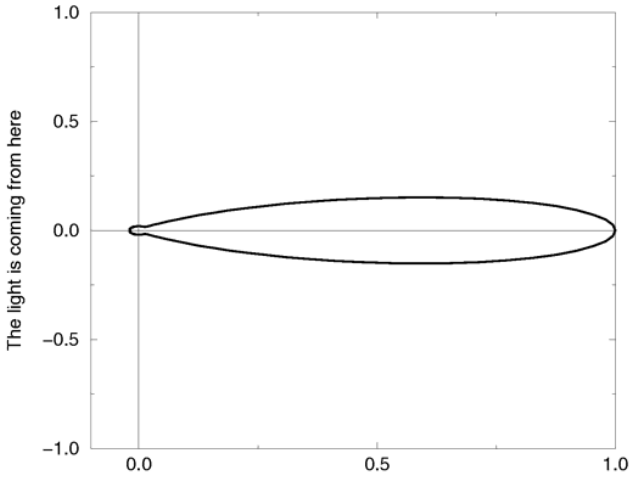
The integer value *Type* specifies one of five different scattering phase functions representing the different models: isotropic, Mie (haze and murky atmosphere), Rayleigh, and Henyey-Greenstein.

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.

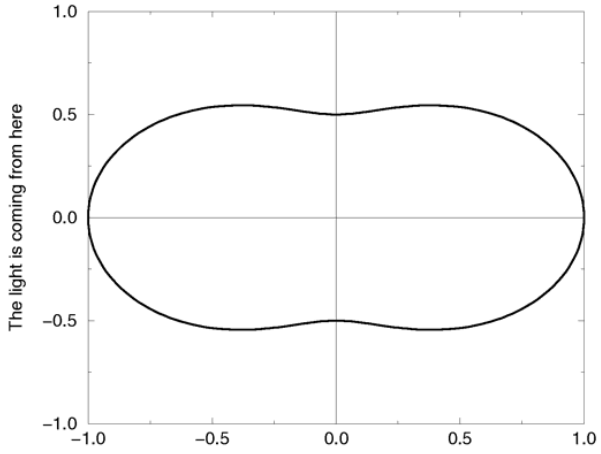


The Mie "haze" scattering function.



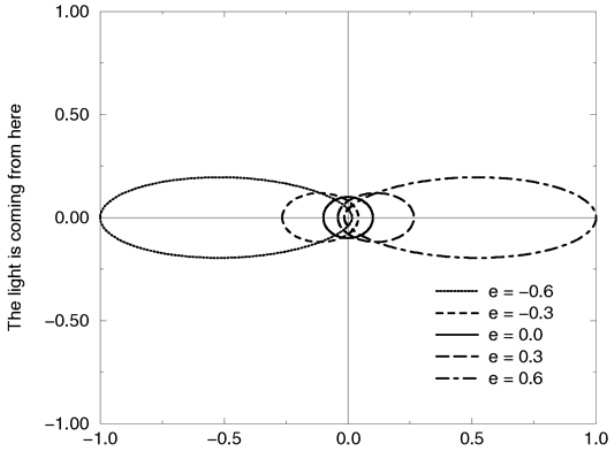
The Mie "murky" scattering function.

Type 3 *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. 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.



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.



The Heyney-Greenstein scattering function for different eccentricity values.

4.8.2 Sampling Parameters

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 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 span 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 **varience 1.0/128** and **confidence 0.9**. For slower more accurate results, decrease the variance and increase the confidence. Note however that the maximum number of samples limits the calculations even if the proper variance and confidence are never reached.

4.8.3 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 "**Error! Reference source not found.**".

The syntax for **density** is:

DENSITY:
density { *[DENSITY_IDENTIFIER] [PATTERN_TYPE] [DENSITY_MODIFIER...]* **}**

DENSITY_MODIFIER:
PATTERN_MODIFIER | *DENSITY_LIST* | *COLOR_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.

4.8.3.1 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.

4.8.3.2 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 rgb<1,0,0>, 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**.

4.8.3.3 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 that 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 that 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.

4.8.3.4 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 that 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.

4.9 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.

4.9.1 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 shaft of light beams will become visible and objects will cast shadows onto smoke or fog.

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.

You should note that 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).

4.9.2 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***}**

4.9.3 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*

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.

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_Alt* is 1.0 and for height *y* less than *Fog_Alt* 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.

You should note that 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).

4.9.4 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.

You should note that 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.

4.9.5 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*

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 colormap. 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.

4.10 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. Regardless of where you specify the statement, the feature applies to the entire scene.

Note that 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* | **max_intersections** *Number* |
max_trace_level *Number* | **number_of_waves** *Number* |
radiosity { *RADIOSITY_ITEMS...* **}**

Each item is optional and may appear in and 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.

4.10.1 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.

4.10.2 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.

4.10.3 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.

The following sections explain more thoroughly what gamma is and why it is important.

4.10.3.1 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.

4.10.3.2 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 new PNG format files generated by POV-Ray 3.0 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.

4.10.3.3 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 3.0 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.

4.10.4 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.

4.10.5 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.

4.10.6 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`** are

not restricted, so it can be set to any number as long as you have the time and memory. However, increasing its setting does not necessarily equate to increased image quality unless such depths are really needed by the scene.

4.10.7 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.

4.10.8 Number_Of_Waves

The **`waves`** and **`ripples`** pattern 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.

4.10.9 Radiosity

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 is more accurate than simplistic ambient light but it takes much longer to compute. For this reason, POV-Ray does not use radiosity by default. Radiosity is turned on using the **`Radiosity`** INI file option or the **`+QR`** command line switch.

The following sections describes how radiosity works, how to control it with various global settings and tips on trading quality vs. speed.

4.10.9.1 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'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 light 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, a traditional constant ambient term is used.

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.

4.10.9.2 Adjusting Radiosity

As described earlier, radiosity is turned on by using the **Radiosity** INI file option or the **+QR** command line switch. However radiosity has many parameters that are specified in a **radiosity** statement inside a **global_settings** statement as follows:

global_settings { radiosity { *[RADIOSITY_ITEMS...]* **}}**

RADIOSITY_ITEMS:
brightness *Float* | **count** *Integer* | **distance_maximum** *Float* |
error_bound *Float* | **gray_threshold** *Float* | **low_error_factor** *Float* |
minimum_reuse *Float* | **nearest_count** *Integer* | **recursion_limit** *Integer*

Each item is optional and may appear in and 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.

4.10.9.2.1 brightness

This **brightness** keyword specifies a float value that is the degree to which ambient values are brightened before being returned upwards to the rest of the system. If an object is red **rgb**<**1,0 0**>, with an ambient value of 0.3, in normal situations a red component of 0.3 will be added in. With radiosity on, assume it was surrounded by an object of gray color **rgb<0.6,0.6,0.6>**. The average color returned by the gathering process will be the same. This will be multiplied by the texture's ambient weight value of 0.3, returning **rgb<0.18,0.18,0.18>**. This is much darker than the 0.3 which would be added in normally. Therefore, all returned values are brightened by the inverse of the average of the calculated values, so the average ambient added in does not change. Some will be higher than specified (higher than 0.3 in this example) and some will be lower but the overall scene brightness will be unchanged. The default value is 3.3.

4.10.9.2.2 count

The integer number of rays that are sent out whenever a new radiosity value has to be calculated is given by **count**. Values of 100 to 150 make most scenes look good. Higher values might be needed for scenes with high contrast between light levels or small patches of light causing the illumination. This would be used only for a final rendering on an image because it is very compute intensive. Since most scenes calculate the ambient value at 1% to 2% of pixels, as a rough estimate, your rendering will take 1% to 2% of this number times as long. If you set it to 300 your rendering might take 3 to 6 times as long to complete (1% to 2% times 300).

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. The default value is 100.

4.10.9.2.3 distance_maximum

The **`distance_maximum`** float value is the only tuning value that depends upon the size of the objects in the scene. This one `must` be set for scenes to render properly... the rest can be ignored for a first try. It is difficult to describe the meaning simply but it sets the distance in model units from a sample at which the error is guaranteed to hit 100% (radiosity_error_bound >=1): no samples are reused at a distance larger than this from their original calculation point.

Imagine an apple at the left edge of a table. The goal is to make sure that samples on the surface of the table at the right are not used too close to the apple and definitely not underneath the apple. If you had enough rays there wouldn't be a problem since one of them would be guaranteed to hit the apple and set the reuse radius properly for you. In practice, you must limit this.

We use this technique: find the object in your scene which might have the following problem: a small object on a larger flatter surface that you want good ambient light near. Now, how far from this would you have to get to be sure that one of your rays had a good chance of hitting it? In the apple-on-the-table example, assuming I used one POV-Ray unit as one inch, I might use 30 inches. A theoretically sound way (when you are running lots of rays) is the distance at which this object's top is 5 degrees above the horizon of the sample point you are considering. This corresponds to about 11 times the height of the object. So, for a 3-inch apple, 33 inches makes some sense. For good behavior under and around a 1/3 inch pea, use 3 inches etc. Another VERY rough estimate is one third the distance from your eye position to the point you are looking at. The reasoning is that you are probably no more than 90 inches from the apple on the table, if you care about the shading underneath it. The default value is 0.

4.10.9.2.4 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. 0.5 is a little rough and ready, 0.33 is good for final renderings. Values much lower than 0.3 take *forever*. The default value is 0.4.

4.10.9.2.5 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, we only do one bounce and 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 that this does not change the lightness/darkness, only the strength of hue/grayness (in HLS terms, it changes S only). The default value is 0.5

4.10.9.2.6 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: 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 0.8.

4.10.9.2.7 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 eyepoint. 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 unitless ratio. The default value is 0.015.

4.10.9.2.8 nearest_count

The **nearest_count** integer value is the maximum number of old ambient values blended together to create a new interpolated value. 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 10, since that is the size of the array allocated. The default value is 6.

4.10.9.2.9 recursion_limit

The **recursion_limit** is an integer value which determines how many recursion levels are used to calculate the diffuse inter-reflection. Valid values are one and two. The default value is 1.

4.10.9.3 Tips on Radiosity

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** and **distance_maximum**.

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.

If your scene looks good but right near some objects you get spots of the right (usually darker) color showing on a flat surface of the wrong color (same as far away from the object), then try dropping **distance_maximum**. If that still doesn't work well increase your ray count by 100 and drop the error bound just a bit. If you still have problems, drop **nearest_count** to about 4.

5 APPENDICES

5.1 Copyright, Legal Information and License -- POVLEGAL.DOC

The following sections contain the legal information and license for the Persistence of Vision™ Ray-Tracer, also called POV-Ray™. Before you use this program you have to read the sections below.

5.1.1 General License Agreement -- POVLEGAL.DOC

Revised June 1998.

THIS NOTICE MUST ACCOMPANY ALL OFFICIAL OR CUSTOM PERSISTENCE OF VISION FILES. IT MAY NOT BE REMOVED OR MODIFIED. THIS INFORMATION PERTAINS TO ALL USE OF THE PACKAGE WORLDWIDE. THIS DOCUMENT SUPERSEDES ALL PREVIOUS GENERAL LICENSES OR DISTRIBUTION POLICIES. ANY INDIVIDUALS, COMPANIES OR GROUPS WHO HAVE BEEN GRANTED SPECIAL LICENSES MAY CONTINUE TO DISTRIBUTE VERSION 2.x BUT MUST RE-APPLY FOR VERSION 3.00 OR LATER.

This document pertains to the use and distribution of the Persistence of Vision(tm) Ray Tracer a.k.a POV-Ray(tm). It applies to all POV-Ray program source files, executable (binary) files, scene files, documentation files, help file, bitmaps and INI files contained in official POV-Ray Team(tm) archives. All of these are referred to here as "the software". The POV-Team reserves the right to revise these rules in future versions and to make additional rules to address new circumstances at any time.

All of this software is Copyright 1991,1998 by the POV-Ray Team(tm). Although it is distributed as freeware, it is NOT PUBLIC DOMAIN.

The copyrighted package may ONLY be distributed and/or modified according to the license granted herein. The spirit of the license is to promote POV-Ray as a standard ray tracer, provide the full POV-Ray package freely to as many users as possible, prevent POV-Ray users and developers from being taken advantage of, enhance the life quality of those who come in contact with POV-Ray. This license was created so these goals could be realized. You are legally bound to follow these rules, but we hope you will follow them as a matter of ethics, rather than fear of litigation.

5.1.2 Usage Provisions

Permission is granted to the user to use the software and associated files in this package to create and render images. The use of this software for the purpose of creating images is completely free. The creator of a scene file and the image created from the scene file, retains all rights to the image and scene file they created and may use them for any purpose commercial or non-commercial.

The user is also granted the right to use the scenes files, fonts, bitmaps, and include files distributed in the INCLUDE and SCENES\INCDEMO sub- directories in their own scenes. Such permission does not extend to files in the SCENES directory or its sub-directories. The SCENES files are for your enjoyment and education but may not be the basis of any derivative works.

5.1.3 General Rules For All Distribution

The permission to distribute this package under certain very specific conditions is granted in advance, provided that the following conditions are met.

These archives must not be re-archived using a different method without the explicit permission of the POV-Team. You may rename the archives only to meet the file name conventions of your system or to avoid file name

duplications but we ask that you try to keep file names as similar to the originals as possible. (For example: POVSRC.ZIP to POVSRC30.ZIP)

Ready-to-run unarchived distribution on CD-ROM is also permitted if the files are arranged in our standard directory or folder structure as though it had been properly installed on a hard disk.

You must distribute a FULL PACKAGE of files as described in the next section. No portion of this package may be separated from the package and distributed separately other than under the conditions specified in the provisions given below.

Non-commercial distribution in which no money or compensation is charged (such as a user copying the software for a personal friend or colleague) is permitted with no other restrictions.

Teachers and educational institutions may also distribute the material to students for free or they may charge minimal copying costs if the software is to be used in a course.

5.1.4 Definition Of "Full Package"

A "full package" contains an executable program, documentation, and sample scenes. For generic Unix platforms, there are no executables available so the portable C source code must be included instead of the executable program.

POV-Ray is officially distributed for MS-Dos; Windows 95/98/NT; Linux for Intel x86 series; Apple Macintosh; Apple PowerPC; SunOS; and Amiga. Other systems may be added in the future.

Distributors need not support all platforms but for each platform you support you must distribute a full package. For example a Macintosh only CD-ROM need not distribute the Windows versions.

This software may ONLY be bundled with other software packages according to the conditions specified in the provisions below.

5.1.4.1 Conditions For Shareware/Freeware Distribution Companies

Shareware and freeware distribution companies may distribute the software included in software-only compilations using media such as, but not limited to, floppy disk, CD-ROM, tape backup, optical disks, hard disks, or memory cards. This section only applies to distributors of collected programs. Anyone wishing to bundle the package with a shareware product must use the commercial bundling rules.

For floppy disk distribution, no more than five dollars U.S. ($5) can be charged per disk for the copying of this software and the media it is provided on. Space on each disk must be used as fully as possible. You may not spread the files over more disks than are necessary.

Distribution on high capacity media such as backup tape or CD-ROM is permitted if the total cost to the user is no more than $0.08 U.S. dollars per megabyte of data. For example a CD-ROM with 600 meg could cost no more than $48.00 US. Any bundling with books, magazines or other print media is permitted if the total cost of the book or magazine with CD is less than the $48.00 limit. Bundling with more expensive publications or distributions should also use the commercial rules.

5.1.5 Conditions For On-Line Services And Bbs's Including Internet

On-line services, BBS's and internet sites may distribute the POV-Ray software under the conditions in this section. Sites which allow users to run POV-Ray from remote locations must use separate provisions in the section below.

The archives must all be easily available on the service and should be grouped together in a similar on-line area.

It is strongly requested that sites remove prior versions of POV-Ray to avoid user confusion and simplify or minimize our support efforts.

The site may only charge standard usage rates for the downloading of this software. A premium may not be charged for this package. I.E. CompuServe or America On-Line may make these archives available to their users, but they may only charge regular usage rates for the time required to download.

5.1.6 Online Or Remote Execution Of POV-Ray

Some internet sites have been set up so that remote users can actually run POV-Ray software on the internet server. Other companies sell CPU time for running POV-Ray software on workstations or high-speed computers. Such use of POV-Ray software is permitted under the following conditions.

Fees or charges, if any, for such services must be for connect time, storage or processor usage ONLY. No premium charges may be assessed for use of POV-Ray beyond that charged for use of other software. Users must be clearly notified that they are being charged for use of the computer and not for use of POV-Ray software.

Users must be prominently informed that they are using POV-Ray software, that such software is free, and where they can find official POV-Ray software. Any attempt to obscure the fact that the user is running POV-Ray is expressly prohibited.

All files normally available in a full package distribution, especially a copy of this license and full documentation must be available for download or readable online so that users of an online executable have access to all of the material of a full user package.

If the POV-Ray software has been modified in any way, it must also follow the provisions for custom versions below.

5.1.7 Permitted Modification And Custom Versions

Although the full source code for POV-Ray is distributed, there are strict rules for the use of the source code. The source distribution is provided to; 1) promote the porting of POV-Ray to hardware and operating systems which the POV-Team cannot support. 2) promote experimentation and development of new features to the core code which might eventually be incorporated into the official version. 3) provide insight into the inner workings of the program for educational purposes.

These license provisions have been established to promote the growth of POV-Ray and prevent difficulties for users and developers alike. Please follow them carefully for the benefit of all concerned when creating a custom version.

The user is granted the privilege to modify and compile the source code for their own personal use in any fashion they see fit. What you do with the software in your own home is your business.

However severe restrictions are imposed if the user wishes to distribute a modified version of the software, documentation or other parts of the package (here after referred to as a "custom version"). You must follow the provisions given below. This includes any translation of the documentation into other languages or other file formats.

A "custom version" is defined as a fully functional version of POV-Ray with all existing features intact. ANY OTHER USE OF ANY POV-Ray SOURCE CODE IS EXPRESSLY PROHIBITED. The POV-Team does not license source code for any use outside POV-Ray. No portion of the POV-Ray source code may be incorporated into another program unless it is clearly a custom version of POV-Ray that includes all of the basic functions of POV-Ray.

All executables, documentation, modified files and descriptions of the same must clearly identify themselves as a modified and unofficial version of POV-Ray. Any attempt to obscure the fact that the user is running POV-Ray or to obscure that this is an unofficial version expressly prohibited.

POV-Ray may not be linked into other software either at compile-time using an object code linker nor at run-time as a DLL, ActiveX, or other system. Such linkage can tend to blur the end-user's perception of which program provides which functions and thus qualifies as an attempt to obscure what is running.

To allow POV-Ray to interface with outside programs, the official versions of POV-Ray include several "hooks" for it to call other tasks. For example: the generic part of POV-Ray provides operating system shell-out commands. The Windows version has GUI-extension hooks and the ability to replace the text editor. Modification to these hooks or other officially supported interface mechanisms to increase functionality beyond that of the official version IS EXPRESSLY PROHIBITED.

5.1.8 Conditions For Distribution Of Custom Versions

If your re-compiled version meets all requirements for custom versions listed above, the following conditions apply to its distribution:

You must provide all POV-Ray support for all users who use your custom version. You must provide information so that user may contact you for support for your custom version. The POV-Ray Team is not obligated to provide you or your users any technical support.

Include contact information in the DISTRIBUTION_MESSAGE macros in the source file OPTOUT.H and insure that the program prominently displays this information. Display all copyright notices and credit screens for the official version.

Custom versions may only be distributed as freeware. You must make all of your modifications to POV-Ray freely and publicly available with FULL SOURCE CODE to the modified portions of POV-Ray and must freely distribute full source to any new parts of the custom version. The goal is that users must be able to re-compile the program themselves using readily available compilers and run-time libraries and to be able to further improve the program with their own modifications.

You must provide documentation for any and all modifications that you have made to the program that you are distributing. Include clear and obvious information on how to obtain the official POV-Ray.

The user is encouraged to send enhancements and bug fixes to the POV-Ray Team, but the team is in no way required to utilize these enhancements or fixes. By sending material to the team, the contributor asserts that he owns the materials or has the right to distribute these materials. He authorizes the team to use the materials any way they like. The contributor still retains rights to the donated material, but by donating, grants unrestricted, irrevocable usage and distribution rights to the POV- Team. The team doesn't have to use the material, but if we do, you do not acquire any rights related to POV-Ray. The team will give you credit as the creator of new code if applicable.

Include a copy of this document (POVLEGAL.DOC).

5.1.9 Conditions For Commercial Bundling

Vendors wishing to bundle POV-Ray with commercial software (including shareware) or other distribution not already described above must first obtain explicit permission from the POV-Team. Such permission is rarely granted. The POV-Team will decide if such distribution will be allowed on a case-by-case basis and may impose certain restrictions as it sees fit. The minimum terms are given below. Other conditions may be imposed.

The product must be an existing product that has proven itself as commercially viable without POV-Ray included. The inclusion of POV-Ray should be promoted only as a free bonus and not as a feature designed to encourage customers to purchase or upgrade solely for the POV-Ray capability. Purchasers of your product must not be led to believe that they are paying for POV-Ray. Any mention of the POV-Ray bundle on the box, in advertising or in instruction manuals must be clearly marked with a disclaimer that POV-Ray is free software and can be obtained for free or nominal cost from various sources. Include clear and obvious information on how to obtain the official POV-Ray. You must provide all POV-Ray support for all users who acquired POV- Ray through your product. The POV-Team is not obligated to provide you or your customers any technical support. Include a credit page or

pages in your documentation for POV-Ray. If you modify any portion POV-Ray for use with your hardware or software, you must follow the custom version rules in addition to these rules. Include contact and support information for your product. Include a full user package as described above.

5.1.10 POV-Team Endorsement Prohibitions

On rare occasions, the POV-Team endorses distributions in which POV-Team members are compensated participants and which the POV-Team has given approval.

Without specific approval, distributors (whether free or commercial) must not claim or imply in any way that the POV-Team officially endorses or supports the distributor or the product (such as CD, book, or magazine) associated with the distribution.

You may not claim or imply that the POV-Team derives any benefit from your distribution.

If you wish to emphasize that your distribution is legal, you may use this language "This distribution of the official version of POV-Ray is permitted under the terms of the General License in the file POVLEGAL.DOC. The POV-Team does not endorse the distributor or its products. The POV-Team receives no compensation for this distribution."

5.1.11 Retail Value Of This Software

Although POV-Ray is, when distributed within the terms of this agreement, free of charge, the retail value (or price) of this program is determined as US$20.00 per copy distributed or copied. If the software is distributed or copied without authorization you are legally liable to this debt to the copyright holder or any other person or organization delegated by the copyright holder for the collection of this debt, and you agree that you are legally bound by the above and will pay this debt within 30 days of the event.

However, none of the above paragraph constitutes permission for you to distribute this software outside of the terms of this agreement. In particular, the conditions and debt mentioned above (whether paid or unpaid) do not allow you to avoid statutory damages or other legal penalties and does not constitute any agreement that would allow you to avoid such other legal remedies as are available to the copyright holder.

Put simply, POV-Ray is only free if you comply with our distribution conditions; it is not free otherwise. The copyright holder of this software chooses to give it away free under these and only these conditions.

For the purpose of copyright regulations, the retail value of this software is US$20.00 per copy.

5.1.12 Other Provisions

The team permits and encourages the creation of programs, including commercial packages, which import, export or translate files in the POV-Ray Scene Description Language. There are no restrictions on use of the language itself. We reserve the right to add or remove or change any part of the language.

"POV-Ray", "Persistence of Vision", "POV-Team" and "POV-Help" are trademarks of the POV-Team.

While we do not claim any restrictions on the letters "POV" alone, we humbly request that you not use POV in the name of your product. Such use tends to imply it is a product of the POV-Team. Existing programs which used "POV" prior to the publication of this document need not feel guilty for doing so provided that you make it clear that the program is not the work of the team nor endorsed by us.

5.1.13 Revocation Of License

VIOLATION OF THIS LICENSE IS A VIOLATION OF COPYRIGHT LAWS. IT WILL RESULT IN REVOCATION OF ALL DISTRIBUTION PRIVILEGES AND MAY RESULT IN CIVIL OR CRIMINAL PENALTY.

Such violators who are prohibited from distribution will be identified in this document.

In this regard, "PC Format", a magazine published by Future Publishing, Ltd. in the United Kingdom, distributed incomplete versions of POV-Ray 1.0 in violation the license which was effect at the time. They later attempted to distribute POV-Ray 2.2 without prior permission of the POV- Team in violation the license which was in effect at the time. There is evidence that other Future Publishing companies have also violated our terms. Therefore "PC Format", and any other magazine, book or CD-ROM publication owned by Future Publishing is expressly prohibited from any distribution of POV-Ray software until further notice.

5.1.14 Disclaimer

This software is provided as is without any guarantees or warranty. Although the authors have attempted to find and correct any bugs in the package, they are not responsible for any damage or losses of any kind caused by the use or misuse of the package. The authors are under no obligation to provide service, corrections, or upgrades to this package.

5.1.15 Technical Support

We sincerely hope you have fun with our program. If you have any problems with the program, the team would like to hear about them. Also, if you have any comments, questions or enhancements, please contact the POV-Ray Team on the CompuServe Information Service in the GO POVRAY forum or check us out on the internet at http://www.povray.org or ftp.povray.org. The USENET group comp.graphics.rendering.raytracing is a great source of information on POV-Ray and related topics.

License inquiries should be made via email and limited technical support is available via email to:

Chris Young POV-Ray Team Coordinator CIS: 76702,1655 Internet 76702.1655@compuserve.com

The following postal address is only for official license business and only if email is impossible.

We do not provide technical support via regular mail, only email. We don't care if you don't have a modem or online access. We will not mail you disks with updated versions. Do not send money.

Chris Young; 3119 Cossell Drive; Indianapolis, IN 46224 U.S.A.

The other authors' contact information may be found in the documentation.

5.2 Authors

Following is a list in alphabetic order of all people who have ever worked on the POV-Ray Team or who have made a note-worthy contribution. If you want to contact or thank the authors read the sections "Contacting the Authors" or use email addresses below.

CURRENT POV-Team MEMBERS		
NAME	PARTICIPATION	EMAIL
Dieter Bayer	Wrote sor, lathe, prism, media and many other features	104707.643@compuserve.com
Thomas Baier	New 3.1 team member, tester	thbaier@ibm.net
Chris Cason	Windows version coordinator and other	

	contributions	
Dale C. Brodin	Alpha & Beta tester, forum support	74361.1764@compuserve.com brodin@execpc.com
Andreas Dilger	Former Unix coordinator, Linux developer, PNG support	adilger@enel.ucalgary.ca www.mddsp.enel.ucalgary.ca/People/adilger/
Peter Drapich	Amiga PowerPC developer	docent@union.org.pl
Thorsten Froehlich	Mac developer	ThorstenFroehlich@compuserve.com
Alan Kong	Alpha & Beta tester, forum support	
Lutz Kretzschmar	Moray author, MS-Dos 24-bit VGA, part of the anti-aliasing code	lutz@stmuc.com
Robert A. Mickelsen	Artist, 3.0 docs contributor	71042.751@compuserve.com ram@iu.net www.websharx.com/ kahuna
Joel Newkirk	Primary Amiga developer	newkirk@snip.net
Nathan O'Brien	Artist, tester	no13@no13.net
Redaelli Paolo	Amiga developer	redaelli@inc.it
Anton Raves	Alpha & Beta tester, Mac contributor	100022.2603@compuserve.com
Eduard Schwan	Mac version coordinator, mosaic preview, docs	104706.3276@compuserve.com povraymac@compuserve.com espsw@compuserve.com http://ourworld.compuserve.com/homepages/povraymac
Erkki Sondergaard	Alpha & Beta tester, 3.0 Scene files	
Timothy Wegner	Fractal objects, PNG support	71320.675@compuserve.com twegner@phoenix.net
Chris Young	POV-Team coordinator, parser code	76702.1655@compuserve.com

PAST POV-Team MEMBERS and CONTRIBTORS		
NAME	PARTICIPATION	EMAIL
Claire Amundsen	Tutorials for the POV-Ray User Guide	
Steve Anger	POV-Ray 2.0/3.0 developer	70714.3113@compuserve.com sanger@hookup.net
Randy Antler	MS-Dos display code enhancements	
John Baily	RLE targa code	
Eric Barish	Ground fog code	
Kendall Bennett	PMODE library support	
Steve Bennett	GIF support	
David Buck	Original author of DKBTrace POV-Ray 1.0 developer	
Aaron Collins	Co-author of DKBTrace 2.12 POV-Ray 1.0 developer	
Chris Dailey	POV-Ray 3.0 developer	
Steve Demlow	POV-Ray 3.0 developer	
Joris van Drunen Littel	Mac beta tester	
Alexander Enzmann	POV-Ray 1.0/2.0/3.0 developer	70323.2461@compuserve.com xander@mitre.com
Dan Farmer	POV-Ray 1.0/2.0/3.0 developer	
Charles Fusner	Blob, lathe and prism tutorial tutorials for the POV-Ray User Guide	
David Harr	Mac balloon help and palette code	
Jimmy Hoeks	Help file for Windows user interface	

Terry Kanakis	Camera fix	
Kari Juharvi Kivisalo	Ground fog code	
Charles Marslett	MS-Dos display code	
Pascal Massimino	Fractal objects	
Jim McElhiney	POV-Ray 3.0 developer	
Mike Miller	Artist, scene files, stones.inc	
Douglas Muir	Bump maps, height fields	
Jim Nitchals	Mac version, scene files (Jim passed away in June 1998 but his contributions to POV-Ray will not be forgotten)	
Paul Novak	Texture contributions	
Dave Park	Amiga support, AGA video code	
David Payne	RLE targa code	
Bill Pulver	Time code	
Dan Richardson	3.0 Docs	
Tim Rowley	PPM and Windows-specific BMP image format support	trowley@geom.umn.edu
Robert Skinner	Noise functions	
Zsolt Szalavari	Halo code which was later turned into media	zsolt@cg.tuwien.ac.at
Scott Taylor	Leopard and onion textures	
Timothy Wegner	Fractal objects, PNG support	71320.675@compuserve.com twegner@phoenix.net
Drew Wells	POV-Ray 1.0 developer, POV-Ray 1.0 team coordinator	

5.2.1 Contacting the Authors

The POV-Team is a collection of volunteer programmers, designers, animators and artists meeting via electronic mail on CompuServe's POVRAY forum (GO POVRAY). The POV-Team's goal is to create freely distributable, high quality rendering and animation software written in C that can be easily ported to many different computers.

If you have any questions about POV-Ray, please contact

Chris Young
POV-Team Coordinator
76702.1655@compuserve.com

We love to hear about how you're using and enjoying the program. We also will do our best try to solve any problems you have with POV-Ray and incorporate good suggestions into the program.

If you have a question regarding commercial use or distribution of POV-Ray, or anything else particularly sticky, please contact Chris Young, the development team coordinator. Otherwise, spread the mail around. We all love to hear from you!

Please do not send large files to us through the e-mail without asking first. Send a query before you send the file, thanks!

5.3 What to do if you don't have POV-Ray

This documentation assumes you already have POV-Ray installed and running however the POV-Team does distribute this file by itself in various formats including online on the internet. If you don't have POV-Ray or aren't sure you have the official version or the latest version, then the following sections will tell you what to get and where to get it.

5.3.1 Which Version of POV-Ray should you use?

POV-Ray can be used under MS-Dos, Windows 3.x, Windows for Workgroups 3.11, Windows 95, Windows NT, Apple Macintosh 68k, Power PC, Amiga, Linux, UNIX and other platforms. The latest versions of the necessary files are available over CompuServe, Internet, and various CD distributions. See section "Where to Find POV-Ray Files" for more info. If your platform is not supported and you are proficient in compiling source code programs written in C then, see section "Compiling POV-Ray" for more information.

5.3.1.1 Microsoft Windows 95/98/NT

The Windows version runs under Windows 95, Windows 98, and Windows NT4 also should run under OS/2 Warp.

Windows 3.1 and Windows for Workgroups are no longer supported however the MS-Dos version will run as a dos application under those operating systems.

Required hardware and software:

Minimum - 486/100 with 16mb RAM and Windows 95.
Disk space - 15 megabytes

Required POV-Ray files:

User archive POVWIN3.EXE - a self-extracting archive containing the program, sample scenes, standard include files and documentation. This file may be split into smaller files for easier downloading. Check the directory of your download or ftp site to see if other files are needed.

Recommended:

Pentium 200 or Pentium II with 32mb and Windows 95 or NT4.
SVGA display preferably with high color or true color ability and drivers installed. (Note: accelerated graphics hardware will not improve performance.)

Optional: The source code is not needed to use POV-Ray. It is provided for the curious and adventurous.

POVWIN_S.ZIP --- The C source code for POV-Ray for Windows. Contains generic parts and Windows specific parts. It does not include sample scenes, standard include files and documentation so you should also get the executable archive as well. POV-Ray can only be compiled using C compilers that create 32-bit Windows applications. We support Watcom 10.5a or higher, Borland 4.52/5.0 compilers.

5.3.1.2 MS-Dos & Windows 3.x

The MS-Dos version runs under MS-Dos or as a DOS application under Windows 3.1 or Windows for Workgroups 3.11. Although it also runs under Windows 95/98/NT4 and OS/2 Warp, users of those systems should use the Windows version..

Required hardware and software:

A 386 or better CPU and at least 8 meg of RAM.
About 6 meg disk space to install and 2-10 meg or more beyond that for working space.
A text editor capable of editing plain ASCII text files. The EDIT program that comes with MS-Dos will work for moderate size files.
Graphic file viewer capable of viewing GIF and perhaps TGA and PNG formats.

Required POV-Ray files:

POVMSDOS.EXE - a self-extracting archive containing the program, sample scenes, standard include files and documentation in plain ASCII text format. This file may be split into smaller files for easier downloading. Check the directory of your download or ftp site to see if other files are needed.

Recommended:

Pentium 200 or Pentium II (faster the better) 32 meg or more RAM. SVGA display preferably with VESA interface and high color or true color ability. (Note: accelerated graphics hardware will not improve performance.)

Optional: The source code is not needed to use POV-Ray. It is provided for the curious and adventurous.

POVMSD_S.ZIP - The C source code for POV-Ray for MS-Dos. Contains generic parts and MS-Dos specific parts. It does not include sample scenes, standard include files and documentation so you should also get the executable archive as well. Requires a C compiler that can create 32-bit protected mode applications. We support Watcom 11, Borland 4.52 with DOS Power Pack and DJGPP 2.

5.3.1.3 Linux for Intel x86

Required hardware and software:

A 386 or better CPU and at least 8 meg of RAM.
About 6 meg disk space to install and 2-10 meg or more beyond that for working space.
A text editor capable of editing plain ASCII text files.
Graphic file viewer capable of viewing PPM, TGA or PNG formats.
Any recent (1994 onwards) Linux kernel and support for ELF format binaries. POV-Ray for Linux is not in a.out-format. ELF libraries libc.so.5, libm.so.5 and one or both of libX11.so.6 or libvga.so.1.

Required POV-Ray files:

POVLINUX.TGZ or POVLINUX.TAR.GZ - archive containing an official binary for each SVGALib and X-Windows modes. Also contains sample scenes, standard include files and documentation in plain ASCII text.

Recommended:

Pentium 200 or Pentium II (faster the better) 32 meg or more RAM. SVGA display preferably with VESA interface and high color or true color ability. If you want display, you'll need either SVGALib or X-Windows. (Note: accelerated graphics hardware will not improve performance.)

Optional: The source code is not needed to use POV-Ray. It is provided for the curious and adventurous.

POVUNI_S.TAR.GZ or POVUNI_S.TGZ - The C source code for POV-Ray for Linux. Contains generic parts and Linux specific parts. It does not include sample scenes, standard include files and documentation so you should also get the executable archive as well. Requires the GNU C compiler and (optionally) the X include files and libraries.

5.3.1.4 Apple Macintosh

The MacOS versions run under Apple's MacOS operating system version 7.1 or newer, on any 68020/030/040-based Macintosh-compatible computer (with or without a floating point coprocessor) or any of the PowerPC Macintosh-compatible computers.

Required hardware and software:

A 68020 or better CPU without a floating point unit (LC or Performa or Centris series); at least 8 MB RAM or
A 68020 or better CPU *with* a floating point unit (Mac II or Quadra series); at least 8 MB RAM or
Any PowerPC Macintosh computer and at least 8 MB RAM.
System 7.1 or newer and color QuickDraw (System 6 is no longer supported).
About 6 MB free disk space to install and an additional 2-10 MB free space for your own creations (scenes and images).
Graphic file viewer utility capable of viewing Mac PICT, GIF and perhaps TGA and PNG formats (the shareware GraphicConverter or GIFConverter applications are good.)

Required POV-Ray files:

POVMACNF.SIT or POVMACNF.SIT.HQX - a Stuffit archive containing the non-FPU 68K Macintosh application, sample scenes, standard include files and documentation (slower version for Macs without an FPU) or

POVMAC68.SIT or POVMAC68.SIT.HQX - a Stuffit archive containing the FPU 68K Macintosh application, sample scenes, standard include files and documentation (faster version for Macs WITH an FPU) or

POVPMAC.SIT or POVPMAC.SIT.HQX - a Stuffit archive containing the native Power Macintosh application, sample scenes, standard include files and documentation.

Recommended POV-Ray files:

68030/33 or faster with FPU, or any Power Macintosh
8 MB or more RAM for 68K Macintosh;
16 MB or more for Power Macintosh systems.
Color monitor preferred, 256 colors OK, but thousands or millions of colors is even better.

Optional: The source code is not needed to use POV-Ray. It is provided for the curious and adventurous. POV-Ray can be compiled using Apple's MPW 3.3, Metrowerks CodeWarrior Pro or Symantec 8.

POVMACS.SIT or POVMACS.SIT.HQX - The full C source code for POV-Ray for Macintosh. Contains generic parts and Macintosh specific parts. It does not include sample scenes, standard include files and documentation so you should also get the executable archive as well.

5.3.1.5 Amiga

The Amiga version comes in several flavors, 68000/68020 without FPU, (not recommended, very slow) 68020/68881(68882), 68030/68882 and 68040. There's two way to use Pov: Shell only and using the provided GUI interface which requires MUI 3.8. All versions run under OS3.x and up. Support exists for pensharing and window display under OS3.x with 256 color palettes, plus CybeGFX and Picasso 96 24-bit display.

Required:

At least 4 meg of RAM.
At least 2 meg of hard disk space for the necessities, 5-20 more recommended for workspace.
An ASCII text editor, GUI configurable to launch the editor of your choice.
Graphic file viewer - POV-Ray outputs to PNG, Targa (TGA), and PPM formats, converters from the PPMBIN distribution are included to convert these to IFF ILBM files.
The ixemul.libraries for certain compile of Pov. It is provided with Pov main program when needed.

Required POV-Ray files:

POVAMIx0.LHA -- a LHA archive containing executable, sample scenes, standard include files, and documentation for the 680x0 version of POV-Ray (i.e.: POVAMI60.LHA stands for 68060 version); 68020 version needs FPU.

Recommended:

8 meg or more of RAM.
68030 & 68882 or higher processor.
24bit display card (CyberGFX and Picasso 96 library supported)

Amiga PowerPC version of POV-Ray is now in beta testing. Watch the CompuServe forum or ww.povray.org for further news.

Optional: The source code is not needed to use POV-Ray. It is provided for the curious and adventurous.

POVAMI_S.LHA --- The C source code for POV-Ray for Amiga. Contains generic parts and Amiga specific parts. It does not include sample scenes, standard include files, and documentation so you should also get the executable archive as well.

5.3.1.6 SunOS

Required hardware and software:

A Sun SPARC processor and at least 4 meg of RAM.
About 6 meg disk space to install and 2-10 meg or more beyond that for working space.
Graphic file viewer capable of viewing PPM, TGA or PNG formats.
A text editor capable of editing plain ASCII text files.
SunOS 4.1.3 or other operating system capable of running such a binary (Solaris or possibly Linux for Sparc).

Required POV-Ray files:

POVSUNOS.TGZ or POVSUNOS.TAR.GZ - archive containing an official binary for each text-only and X-Windows modes. Also contains sample scenes, standard include files and documentation.

Recommended:

8 meg or more RAM.
If you want display, you'll need X-Windows or an X-Term.- preferably 24-bit TrueColor display ability, although the X display code is known to work with ANY combination of visual and color depth.
Graphic file viewer capable of viewing PPM, TGA or PNG formats.

Optional: The source code is not needed to use POV-Ray. It is provided for the curious and adventurous.

POVUNI_S.TGZ or POVUNI_S.TAR.GZ - The C source code for POV-Ray for UNIX. Contains generic parts and Unix/Linux specific parts. It does not include sample scenes, standard include files and documentation so you should also get the executable archive as well. You will need a C compiler and (optionally) the X include files and libraries and knowledge of how to use it. Although we provide source code for generic Unix systems, we do not provide technical support on how to compile the program.

5.3.1.7 Generic Unix

Because Unix runs on a wide variety of hardware and CPUs, the POV-Team cannot provide executable versions for every type of Unix. We distribute generic Unix source code in portable ANSI C source code. You will need a C compiler and (optionally) the X include files and libraries and knowledge of how to use it. Although we provide source code for generic Unix systems, we do not provide technical support on how to compile the program.

Required:

POVUNI_S.TGZ or POVUNI_S.TAR.GZ - The C source code for POV-Ray for UNIX. Contains generic parts and Unix/Linux specific parts. It does not include sample scenes, standard include files and documentation so you should also get an executable archive for another platform or get
POVUNI_D.TGZ or POVUNI_D.TAR.GZ which contains the sample scenes, standard include files and documentation.
A C compiler for your computer and KNOWLEDGE OF HOW TO USE IT.
Graphic file viewer capable of viewing PPM, TGA or PNG formats.
A text editor capable of editing plain ASCII text files.

Recommended:

Math co-processor.
8 meg or more RAM.

Optional:

X Windows if you want to be able to display as you render.
You will need the X-Windows include files as well. If you're not familiar with compiling programs for X-Windows you may need some help from someone who is knowledgeable at your installation because the X include files and libraries are not always in a standard place.

5.3.1.8 All Versions

Each executable archive includes full documentation for POV-Ray itself as well as specific instructions for using POV-Ray with your type of platform. All versions of the program share the same ray-tracing features like shapes, lighting and textures. In other words, an MS-Dos-PC can create the same pictures as a Cray supercomputer as long as it has enough memory. The user will want to get the executable that best matches their computer hardware.

In addition to the files listed above, the POV-Team also distributes the user documentation in two alternate forms. Note this is the same documentation distributed in other archives but in a different format. This may be especially useful for MS-Dos or Unix users because their documentation is plain ASCII text only.

POVUSER.PDF - Tutorial and Reference documentation in Adobe Acrobat PDF format. Requires Adobe Acrobat Reader available for Windows 3.x, Windows 95/98/NT, Mac and some Unix systems. Go to http://www.adobe.com/prodindex/acrobat/readstep.html to get the reader for free.

POVHTML.ZIP - Archive containing Tutorial and Reference documentation in html for viewing with any internet browser.

See the section "Where to Find POV-Ray Files" for where to find these files. You can contact those sources to find out what the best version is for you and your computer.

5.3.2 Where to Find POV-Ray Files

The latest versions of the POV-Ray software are available from the following sources.

5.3.2.1 POV-Ray Forum on CompuServe

The headquarters of POV-Ray are on CompuServe in the POVRAY forum, that is managed by some of the team members. We meet there to share information, useful programs and utilities and graphics created by POV-Ray. Members of our CompuServe Forum also get early access to beta versions and first hand support from POV-Team members. Everyone is welcome to join in on the action on CIS:POVRAY. Hope to see you there! You can get information on joining CompuServe by calling (800)848-8990 or visit the CompuServe home page http://www.compuserve.com. Direct CompuServe access is also available in Japan, Europe and many other countries.

5.3.2.2 World Wide Website www.povray.org

The internet home of POV-Ray is reachable on the World Wide Web via the address http://www.povray.org and via ftp as ftp.povray.org. Please stop by often for the latest files, utilities, news and images from the official POV-Ray internet site.

The comp.graphics.rendering.raytracing newsgroup has many competent POV-Ray users that are very willing to share their knowledge. They generally ask that you first browse a few files to see if someone has already answered the same question, and of course, that you follow proper "netiquette". If you have any doubts about the qualifications of the folks that frequent the group, a few minutes spend at the Ray Tracing Competition pages at www.povray.org will quickly convince you!

5.3.2.3 Books, Magazines and CD-ROMs

Unfortunately all books on POV-Ray are out of print and there are no plans to reprint them. Many popular computer magazines have been authorized to distribute POV-Ray on cover CDs. Note that such distributions of the official version of POV-Ray is permitted under the terms of the General License in the file POVLEGAL.DOC. The POV-Team does not endorse the distributor or its products. The POV-Team receives no compensation for this distribution.

The POV-Team does endorse some CD-ROMs containing POV-Ray which are prepared by team members. A portion of the proceeds from these CDs support our internet sites and other team activities. You can always find the latest information on what is available at our web site www.povray.org.

5.4 Compiling POV-Ray

The following sections will help you to compile the portable C source code into a working executable version of POV-Ray. They are only for those people who want to compile a custom version of POV-Ray or to port it to an unsupported platform or compiler.

The first question you should ask yourself before proceeding is "*Do I really need to compile POV-Ray at all?*" Official POV-Team executable versions are available for MS-Dos, Windows 95/98/NT, Mac 68k, Mac Power PC, Amiga, Linux for Intel x86, and SunOS. Other unofficial compiles may soon be available for other platforms. If you do not intend to add any custom or experimental features to the program and if an executable already exists for your platform then you need not compile this program yourself.

If you do want to proceed you should be aware that you are very nearly on your own. The following sections and other related compiling documentation assume you know what you are doing. It assumes you have an adequate C compiler installed and working. It assumes you know how to compile and link large, multi-part programs using a "make" utility or an IDE project file if your compiler supports them. Because makefiles and project files often specify drive, directory or path information, we cannot promise our makefiles or projects will work on your system. We assume you know how to make changes to makefiles and projects to specify where your system libraries and other necessary files are located.

In general you should not expect any technical support from the POV-Team on how to compile the program. Everything is provided here as is. All we can say with any certainty is that we were able to compile it on our systems. If it doesn't work for you we probably cannot tell you why.

There is no technical documentation for the source code itself except for the comments in the source files. We try our best to write clear, well- commented code but some sections are barely commented at all and some comments may be out dated. We do not provide any technical support to help you to add features. We do not explain how a particular feature works. In some instances, the person who wrote a part of the program is no longer active in the Team and we don't know exactly how it works.

When making any custom version of POV-Ray or any unofficial compile, please make sure you read and follow all provisions of our license in POVLEGAL.DOC. In general you can modify and use POV-Ray on your own however you want but if you distribute your unofficial version you must follow our rules. You may not under any circumstances use portions of POV-Ray source code in other programs.

5.4.1 Directory Structure

POV-Ray source code is distributed in archives with files arranged in a particular hierarchy of directories or folders. When extracting the archives you should do so in a way that keeps the directory structure intact. In general we suggest you create a directory called `povray3` or `povray31` and extract the files from there. The extraction will create a directory called `source` with many files and sub-directories.

In general, there are separate archives for each hardware platform and operating system but each of these archives may support more than one compiler. For example here is the directory structure for the MS-Dos archive. Other platforms use similar structures.

```
SOURCE
SOURCE\LPNG101
SOURCE\ZLIB
SOURCE\MSDOS
SOURCE\MSDOS\PMODE
SOURCE\MSDOS\BORLAND
SOURCE\MSDOS\DJGPP
SOURCE\MSDOS\WATCOM
```

The `source` directory contains source files for the generic parts of POV-Ray that are the same on all platforms. The `source\lpng101` contains files for compiling a library of routines used in reading and writing PNG

(Portable Network Graphics) image files. The source\zlib contains files for compiling a library of routines used by libpng to compress and uncompress data streams. All of these files are used by all platforms and compilers. They are in every version of the source archives.

The source\msdos directory contains all source files for the MS-Dos version common to all supported MS-Dos compilers. The pmode sub-directory contains source files for pmode.lib which is required by all MS-Dos versions. The borland, djgpp, and watcom sub-directories contain source, makefiles and project files for C compilers by Borland, DJGPP and Watcom.

The source\msdos directory is only in the MS-Dos archive. Similarly the Windows archive contains a source\windows directory. The Unix archive contains source/unix etc.

The source\msdos directory contains a file cmpl_msd.doc which contains compiling information specific to the MS-Dos version. Other platform specific directories contain similar cmpl_xxx.doc files and the compiler specific sub-directories also contain compiler specific cmpl_xxx.doc files. Be sure to read all pertinent cmpl_xxx.doc files for your platform and compiler.

5.4.2 Configuring POV-Ray Source

Every platform has a header file config.h that is generally in the platform specific directory but may be in the compiler specific directory. Some platforms have multiple versions of this file and you may need to copy or rename it as config.h. This file is included in every module of POV-Ray. It contains any prototypes, macros or other definitions that may be needed in the generic parts of POV-Ray but must be customized for a particular platform or compiler.

For example different operating systems use different characters as a separator between directories and file names. MS-Dos uses back slash, Unix a front slash or Mac a colon. The config.h file for MS-Dos and Windows contains the following:

```
#define FILENAME_SEPARATOR '\'
```

which tells the generic part of POV-Ray to use a back slash.

Every customization that the generic part of the code needs has a default setting in the file source\frame.h which is also included in every module after config.h. The frame.h header contains many groups of defines such as this:

```
#ifndef FILENAME_SEPARATOR
#define FILENAME_SEPARATOR '/'
#endif
```

which basically says "If we didn't define this previously in config.h then here's a default value". See frame.h to see what other values you might wish to configure.

If any definitions are used to specify platform specific functions you should also include a prototype for that function. The file source\msdos\config.h, for example, not only contains the macro:

```
#define POV_DISPLAY_INIT(w,h) MSDOS_Display_Init ((w), (h));
```

to define the name of the graphics display initialization function, it contains the prototype:

```
void MSDOS_Display_Init (int w, int h);
```

If you plan to port POV-Ray to an unsupported platform you should probably start with the simplest, non-display generic Unix version. Then add new custom pieces via the config.h file.

5.4.3 Conclusion

We understand that the above sections are only the most trivial first steps but half the fun of working on POV-Ray source is digging in and figuring it out on your own. That's how the POV-Team members got started. We've tried to make the code as clear as we can.

Be sure to read the `cmpl_xxx.doc` files in your platform specific and compiler specific directories for some more minor help if you are working on a supported platform or compiler.

Good luck!

5.5 Suggested Reading

Beside the POV-Ray material mentioned in "Books, Magazines and CD-ROMs" there are several good books or periodicals that you should be able to locate in your local computer book store or your local university library.

"An Introduction to Ray tracing" Andrew S. Glassner (editor) ISBN 0-12-286160-4; Academic Press 1989

"3D Artist" Newsletter, "The Only Newsletter about Affordable PC 3D Tools and Techniques") Publisher: Bill Allen; P.O. Box 4787; Santa Fe, NM 87502-4787; (505) 982-3532

"Image Synthesis: Theory and Practice" Nadia Magnenat-Thalman and Daniel Thalmann; Springer-Verlag; 1987

"The RenderMan Companion" Steve Upstill; Addison Wesley; 1989

"Graphics Gems" Andrew S. Glassner (editor); Academic Press; 1990

"Fundamentals of Interactive Computer Graphics" J. D. Foley and A. Van Dam; ISBN 0-201-14468-9; Addison-Wesley 1983

"Computer Graphics: Principles and Practice (2nd Ed.)" J. D. Foley, A. van Dam, J. F. Hughes; ISBN 0-201-12110-7; Addison-Wesley, 1990

"Computers, Pattern, Chaos, and Beauty" Clifford Pickover; St. Martin's Press; "SIGGRAPH Conference Proceedings"; Association for Computing Machinery Special Interest Group on Computer Graphics

"IEEE Computer Graphics and Applications"; The Computer Society; 10662, Los Vaqueros Circle; Los Alamitos, CA 90720

"The CRC Handbook of Mathematical Curves and Surfaces" David von Seggern; CRC Press; 1990

"The CRC Handbook of Standard Mathematical Tables" CRC Press

6 Index

#break, 170
#case, 170
#debug, 171
#declare, 14, 148, 153, 156, 158, 162, 163, 165, 173, 176, 179, 187
#default, 167
#else, 169, 170
#end, 169, 170, 171, 173
#error, 171, 172
#fclose, 14, 166
#fopen, 14, 166
#if, 169, 173
#ifdef, 166, 169, 173
#ifndef, 169, 173
#include, 17, 72, 161, 164
#local, 14, 148, 162, 163, 165, 169, 173, 176, 179
#macro, 14, 163, 173
#max_intersections, 161
#max_trace_level, 161
#range, 170
#read, 14, 167
#render, 171
#statistics, 171
#switch, 170
#undef, 14, 163, 164, 165, 173
#version, 14, 129, 148, 168
#warning, 171
#while, 171, 173
#write, 14, 167, 173
%o, 130
%s, 130
.blue, 157
.filter, 157
.green, 157
.red, 157
.t, 153
.transmit, 157
.u, 153
.v, 153
.x, 153
.y, 153
.z, 153
/* */, 144
//, 144
+?, 136
+A, 139
+AM, 139
+AM1, 139
+B, 127
+C, 123
+D, 123
+E, 122
+EC, 122
+EP, 125
+ER, 122
+F, 126
+FD, 126
+FN, 123, 126
+FR, 126
+GA, 135
+GD, 134, 135
+GF, 134, 135
+GI, 123
+GR, 134, 135
+GS, 135
+GW, 135
+H, 116, 136, 184
+HN, 128
+HT, 127
+HTC, 128
+I, 116, 117, 129, 130, 142
+J, 141
+K, 119, 148
+KC, 121
+KF, 120
+KFF, 119
+KFI, 119
+KI, 120
+L, 18, 116, 150, 162, 238, 245, 253
+MB, 137
+MV, 129, 148, 168
+O, 127
+P, 116, 124
+Q, 136, 239
+QR, 137, 295, 296
+R, 139, 140
+S, 122
+SC, 122, 123
+SP, 125
+SR, 122, 123
+SU, 138
+UA, 123, 126
+UD, 125
+UF, 121
+UL, 138
+UO, 121
+UV, 138
+V, 116, 117, 124, 134
+W, 116
+X, 122
abs, 149
absorption, 280, 281, 285
acos, 149
acosh, 197
adaptive, 67, 222
adc_bailout, 291, 294
agate, 74, 256, 269
agate_turb, 256, 269
all, 239
All_Console, 135
All_File, 135
alpha, 154, 239
alpha channel, 239
ambient, 68, 80, 247, 250, 281, 296
ambient light, 68
ambient_light, 247, 291
angle, 106, 183, 185, 289
animation, 110
Antialias, 139
Antialias_Depth, 139, 140
Antialias_Threshold, 139
aperture, 186
append, 166, 167
arc_angle, 109, 290
area_light, 67, 221, 223
array, 14
asc, 149
asin, 149, 197
asinh, 197
assumed_gamma, 124, 291, 293
atan, 197
atan2, 149
atanh, 197
atmosphere, 13, 224, 279, 287
atmospheric_attenuation, 224
average, 87, 237, 244, 252, 256, 286
background, 97, 287
banner stream, 134
Bezier patch, 26
bicubic_patch, 26, 206
Bits_Per_Color, 126
black_hole, 273
blob, 189
blue, 19, 154, 156
blur_samples, 186
bounded_by, 225, 226
Bounding, 137
Bounding_Threshold, 137
box, 20, 191
boxed, 13, 257, 284
bozo, 13, 74, 258, 259, 267, 272, 284

brick, 73, 85, 235, 236, 237, 241, 242, 244, 252, 256, 258, 269, 271, 285, 286
brick_size, 258, 269
brightness, 296
brilliance, 247
Buffer_Output, 127
Buffer_Size, 127
bump_map, 241, 242, 244, 245, 271, 278, 279
bump_size, 241, 245, 269
bumps, 71, 78, 241, 242, 258, 259, 271
camera, 18, 185
caustic, 13
caustics, 231, 246
ceil, 149
checker, 85, 235, 236, 237, 238, 241, 242, 244, 251, 252, 253, 256, 259, 271, 285, 286
chr, 159
clipped_by, 205, 210, 225, 226, 228
clock, 119, 148
Clock, 119, 148
clock_delta, 15, 148
color, 19, 101, 235
color_map, 72, 234, 235, 237, 256, 258, 259, 262, 269, 270, 271, 272, 273, 285, 290
colour_map, 235
comments, 144
component, 190
composite, 214
concat, 159
cone, 20, 191, 212
confidence, 187, 284
conic_sweep, 199, 200
constructive solid geometry, 21
Continue_Trace, 123
control0, 264, 269
control1, 264, 269
cos, 149, 197
cosh, 197
count, 296, 298
crackle, 259
crand, 114, 248
Create_Ini, 123
Create_INI, 131
CSG, 21
cube, 197
cubic, 211
cubic_spline, 42, 198, 200
cubic_wave, 13, 271
Cyclic_Animation, 121
cylinder, 20, 34, 67, 189, 192, 212, 217, 221, 223
cylindrical, 13, 183, 184, 186, 260, 284
cylindrical light, 67
-D, 123
debug stream, 134, 172
Debug_Console, 134
Debug_File, 135
DEF files, 116
default files, 116
degrees, 149
density, 13, 256, 270, 271, 280, 284, 285, 286
density_file, 260, 269
density_map, 256, 257, 269, 270, 271, 285
dents, 13, 78, 241, 242, 260, 271
df3, 260
difference, 23, 214, 215
diffuse, 80, 82, 247, 250
diffuse reflection, 247
difuse, 281
dimension, 149
dimension_size, 14, 149
dimensions, 14
direction, 106, 181, 183, 185, 289
disc, 207
Display, 123
Display_Gamma, 124, 126, 291, 293
distance, 101, 288, 290
distance_maximum, 297, 298
div, 149
Draw_Vistas, 125
eccentricity, 283
emission, 281, 285
End_Column, 122
End_Row, 122, 123
error_bound, 297, 298
exp, 150, 197
extinction, 281
-F, 123, 126
fade_depth, 13
fade_distance, 70, 223, 230
fade_power, 13, 70, 223, 230
falloff, 66, 218, 274
falloff_angle, 109, 290
false, 148, 149
fatal stream, 134, 172
Fatal_Console, 134
Fatal_Error_Command, 130, 131, 132
Fatal_Error_Return, 131
Fatal_File, 135
Field_Render, 121
file_exists, 150
filter, 75, 154, 156, 230, 231, 238, 280
filter all, 238, 239
Final_Clock, 120
Final_Frame, 114, 119, 120, 121
finish, 13, 71, 167, 229, 230, 231, 233, 234, 241, 245, 246, 247, 248, 249, 250
fisheye, 183, 186
flatness, 206
flip, 276
floor, 150
focal_point, 187
fog, 101, 287
fog_alt, 104, 288
fog_offset, 104, 288
fog_type, 104, 288
frequency, 74, 266, 268, 270
gif, 194, 238, 244, 253
global_settings, 137, 161, 290, 296
gradient, 74, 236, 237, 244, 252, 260, 263, 270, 271, 284
granite, 74, 261
gray_threshold, 297, 298
green, 19, 154, 156
halo, 13, 229, 231, 251, 257, 260, 263, 267, 279, 284, 287
Height, 122, 125, 184
height_field, 36, 193
hexagon, 73, 85, 235, 236, 237, 241, 242, 244, 252, 256, 261, 271, 285, 286
hf_gray_16, 36, 126, 293
hierarchy, 149, 190, 207
Histogram_Name=file, 128
Histogram_Type, 127
hollow, 106, 149, 227, 229, 232
hypercomplex, 196
iff, 238, 244, 253
image_map, 236, 237, 238, 244, 271, 278
INI files, 116
Initial_Clock, 120
Initial_Frame, 114, 119, 120
initialization files, 116
Input_File_Name, 129, 130, 142
int, 150
interior, 13, 190, 227, 228, 229, 230, 231, 232, 246, 251, 280
internal_highlights, 227
internal_reflections, 227
interpolate, 238, 245, 254, 260, 269, 279

intersection, 23, 214, 215, 225
intervals, 283
inverse, 106, 210, 214, 216, 227
inverted, 275
ior, 13, 230, 246
irid, 83, 250, 271
irid_wavelength, 294
-J, 140
jitter, 67, 107, 114, 222, 290
Jitter, 140
Jitter_Amount, 141
julia_fractal, 195
lambda, 74, 83, 272, 273, 288
lathe, 14, 38, 197
layered textures, 87, 255
leopard, 74
Library_Path, 18, 117, 150, 162, 238, 245, 253
light source, 19
Light_Buffer, 138
light_source, 19, 65, 217
linear_spline, 39, 42, 198, 199
linear_sweep, 49
location, 181, 182
log, 150, 197
look_at, 181, 182, 183, 184, 186
looks_like, 69, 227
low_error_factor, 298
mandel, 262, 270
map_type, 238, 245, 254, 278
marble, 74, 263, 271
material_map, 168, 233, 253, 256, 271, 278, 279
matrix, 177, 178, 180, 270, 277
max, 150
max_iteration, 196
max_trace_level, 291, 294
-MB, 137
media, 13, 227, 229, 231, 251, 257, 260, 263, 267, 279, 280, 285, 286, 287
media_attenuation, 149, 224, 280
media_interaction, 149, 224, 280, 287
merge, 25, 214, 216
mesh, 43, 207, 209
metallic, 82, 249
min, 150
minimum_reuse, 298
mod, 150
mortar, 258, 269
nearest_count, 298
no, 148, 149
no_shadow, 69, 223, 227
normal, 13, 78, 79, 167, 187, 231, 233, 234, 240, 241, 243, 246, 258, 259, 260, 264, 267, 268, 270, 271
normal_map, 84, 241, 243, 256, 257, 259, 260, 264, 267, 268, 269, 270, 271
number_of_waves, 295
object, 163
objects, 188
octaves, 74, 83, 272, 288
Odd_Field, 121
off, 148, 149
offset, 276
omega, 74, 83, 272, 273, 288
omnimax, 183, 186
on, 148, 149
once, 93, 238, 245, 254, 278
onion, 263, 271
open, 20, 192, 193, 200, 228
orthographic, 183, 186
Output_Alpha, 126
Output_File_Name, 119, 127
Output_File_Type, 126
Output_to_file, 123
Output_to_File, 126
-P, 116, 124
Palette, 124
panoramic, 183, 186
pattern, 256
Pause_When_Done, 124
perspective, 183, 185
pgm, 194, 238, 244, 253
phase, 113, 266, 268, 270
phong, 71, 81, 248, 249
phong_size, 81, 248
pi, 148
pigment, 13, 73, 78, 167, 231, 233, 234, 237, 241, 246, 257, 270, 271
pigment_map, 76, 83, 234, 236, 243, 256, 269, 270, 271
planar, 13, 263, 284
plane, 21, 210
png, 194, 238, 244, 253
point_at, 66, 218
pointlight, 65
poly, 210
poly_wave, 13, 271
polygon, 44, 208
Post_Frame_Command, 130, 131, 132
Post_Frame_Return, 131
Post_Scene_Command, 130, 131, 133
Post_Scene_Commands, 132
Post_Scene_Return, 131
pot, 194
pow, 150
ppm, 194, 238, 244, 253
Pre_Frame_Command, 130, 131, 132, 133
Pre_Frame_Return, 131
Pre_Scene_Command, 130, 131, 132
Pre_Scene_Return, 131
precision, 196
Preview_End_Size, 125
Preview_Start_Size, 125
prism, 14, 199
pwr, 197
quadratic_spline, 42, 198
quadric, 212
quadric_spline, 200
Quality, 136, 239
quartic, 211
quaternion, 196
quick_color, 234, 239, 269
quick_colour, 240
quilted, 264, 269, 271
radial, 266, 270
radians, 150
radiosity, 137, 296
Radiosity, 137, 295, 296
radius, 66, 218
rainbow, 106
ramp_wave, 256, 263, 271
rand, 150
ratio, 284
Ray-tracing, 11
read, 166
reciprocal, 197
recursion_limit, 298
red, 19, 154, 156
reflection, 82, 247, 249
reflection_exponent, 14, 250
refraction, 13, 230, 246
Remove_Bounds=, 138
render stream, 134, 172
Render_Console, 134
Render_File, 135
repeat, 273, 275, 276
rgb, 155
rgbf, 156
rgbft, 156
rgbt, 156
right, 17, 181, 183, 184, 185
ripples, 78, 241, 242, 266, 271, 295
rotate, 177, 178, 180, 183, 184, 185, 191, 217, 270, 277
roughness, 81, 249

samples, 284
Sampling_Method, 139
scale, 72, 153, 177, 180, 270, 277
scallop_wave, 271
scattering, 281, 285
seed, 150
shadowless, 69, 217, 223
shearing, 179
sin, 150, 197
sine_wave, 271
sinh, 197
sky, 181, 182
sky_sphere, 97
slice, 196
slope_map, 241, 256, 258, 259, 260, 262, 264, 267, 268, 269, 270, 271
smooth, 149, 195
smooth_triangle, 207, 209
sor, 39, 55, 202
specular, 81, 249, 250
specular reflection, 247, 249
sphere, 190, 201, 212
spherical, 13, 267, 284
spiral1, 267
spiral2, 267
Split_Unions, 138
spotlight, 66, 67, 217, 223
spotted, 74, 258, 259, 267
sqr, 197
sqrt, 150
Start_Column, 122
Start_Row, 122, 123
Statistic_Console, 134
Statistic_File, 135
statistics stream, 134, 172
status stream, 134
str, 159, 167
strcmp, 150
strength, 190, 274
strlen, 150
strlwr, 160
strupr, 160
sturm, 149, 190, 199, 201, 202, 205, 212
Subset_End_Frame, 115, 120
Subset_Start_Frame, 115, 120
substr, 160
superellipsoid, 52, 201
sys, 194, 238, 244, 253
t, 153
tan, 150, 197
tanh, 197
Test_Abort, 122
Test_Abort_Count, 122
text, 57, 204
texture, 13, 19, 70, 72, 78, 167, 180, 188, 228, 229, 231, 232, 234, 241, 246, 251, 252, 255, 270, 271
texture_map, 14, 85, 168, 229, 251, 256, 269, 270, 271
tga, 194, 238, 244, 253
thickness, 83, 250
threshold, 189
tightness, 66, 218
tile2, 253
tiles, 86, 168, 233, 253, 259
torus, 60, 205
transform, 177, 179, 270, 277
translate, 177, 180, 185, 217, 270, 277
transmit, 75, 154, 156, 230, 231, 238, 239, 280
transmit all, 238, 239
triangle, 207, 209
triangle_wave, 263, 267, 268, 271
true, 148, 149
ttf, 204
turb_depth, 104, 288
turbulence, 72, 74, 83, 104, 153, 250, 270, 271, 273, 275, 277, 284, 288
type, 206
u, 153
u_steps, 28, 206
-UA, 126
-UD, 125
ultra_wide_angle, 183
union, 22, 214, 216, 223
up, 110, 181, 183, 186, 288, 290
-UR, 138
use_color, 245
use_colour, 245
use_index, 245
User_Abort_Command, 130, 131, 132
User_Abort_Return, 131
v, 153
-V, 124
v_steps, 28, 206
val, 150
variance, 187, 284
vaxis_rotate, 154
vcross, 154
vdot, 151
Verbose, 124, 134
version, 148, 169
Version, 129, 148, 168
Video_Mode, 124
Vista_Buffer, 138
vlength, 151
vnormalize, 154
vrotate, 154
-W, 116
warning stream, 134, 172
Warning_Console, 135
Warning_File, 135
warp, 270, 272, 273, 284
water_level, 195
waves, 78, 241, 242, 268, 271, 284, 295
width, 106, 289
Width, 122, 125, 184
wood, 13, 72, 74, 241, 268, 271, 284
wrinkles, 78, 241, 242, 268, 271
write, 166, 167
x, 21, 153, 210
y, 21, 153, 210
yes, 148, 149
z, 21, 153, 210