Persistence of Vision Ray Tracer (POV-Ray)
Version 1.0
July 18, 1992

Introduction.....5
Program Description -- What is Ray Tracing?.....5
Which Version of POV-Ray Should You Use?.....7
    IBM-PC and compatibles.....7
    Apple Macintosh.....7
    Commodore Amiga.....7
    UNIX and other systems.....8
    All Versions.....8
Where to find POV-Ray files.....8
    Computer Art Forum on CompuServe.....9
    PC Graphics area on America On-Line.....9
    You Can Call Me Ray BBS in Chicago.....9
    The Graphics Alternative BBS in Oakland.....9
    Internet.....9
Quick Start.....9
Installing POV-Ray.....9
Using Sample Scenes.....10
    Command Line Parameters.....12
Beginning Tutorial.....19
Your First Image.....19
Phong Highlights.....22
Simple Textures.....23
More Shapes.....24
Quadric Shapes.....24
Scene Description Language Reference.....25
    Comments.....25
    Include Files.....26
    Float.....27
    Vector.....27
    Declare.....27
    What can be declared?.....30
    Camera.....32
        Telephoto and Wide-Angle.....33
        Sky and Look_at.....34
        Aspect Ratio.....35
        Translating and Rotating the Camera.....36
    light_source.....36
    spotlight light_source.....37
    no_shadow.....38
Objects.....40

Quick Render Color....41
Transformations....41
Scale....41
Rotate....41
Translate....41
Transformation order....43
"Handedness" -- Rotation Direction....44
Shapes....44
Spheres....44
Planes....45
Boxes....46
Quadric Shapes....46
Quadric surfaces the easy way....47
Triangles....48
Quartic Surfaces....49
Blob....52
How do blobs work?....53
Threshold:....53
Strength:....53
Radius:....54
Center:....54
The Formula....55
Height Fields....55
Bezier/Bicubic Patches....59
Constructive Solid Geometry (CSG)....59
Inside and outside....61
union { A B }....63
intersection { A B }....63
difference { A B }....63
Inverse....64
Composite Objects....65
Bounding Shapes....67
Clipping Shapes....67
Textures....69
Color Pattern....69
Bump Patterns....70
Surface Properties....70
Texture Syntax....71
Textures and Animation....71
Random Dither....72
Layered Textures....73
More about Textures....73
Turbulence or Noise....73
Color Maps....74
Alpha....74
Layering Textures....75
Texture Surface Properties....77

Color....77
ambient value....77
diffuse value....78
brilliance value....78
reflection value....78
refraction value....78
ior value....79
phong value....79
phong_size value....80
specular value....80
roughness value....80
metallic....81
Color Pattern Texture Types....82
Color Maps....82
checker - Color Pattern....83
bozo - Color Pattern....83
spotted - Color Pattern....84
marble - Color Pattern....84
wood - Color Pattern....84
agate - Color Pattern....84
gradient - Color Pattern....84
granite - Color Pattern....85
onion - Color Pattern....86
leopard - Color Pattern....86
tiles - A texture pattern....86
Bump Patterns....88
ripples - Bump Pattern....88
waves - Bump Pattern....88
bumps - Bump Pattern....89
dents - Bump Pattern....89
wrinkles - Bump Pattern....89
Mapping Textures....90
Mapping Types:....90
Mapping Method:....90
image_map - Color Pattern....90
bump_map - A Bump Pattern....93
material_map - A texture pattern....94
Interpolation....95
Misc Features....96
Fog....96
Default Texture....97
Max trace level....98
Advanced Lessons....99
Camera....100
Ray-Object Intersections....101
Textures, Noise, and Turbulence....103
Octaves....105

Layered Textures.....105
Parallel Image Mapping.....108
Common Questions and Answers.....109
Tips and Hints.....113
Suggested Reading.....115
Legal Information.....118
Contacting the Authors.....118

Introduction

This document details the use of the Persistence of Vision Ray Tracer (POV-Ray) and is broken down into several sections.

The first section describes the program POV-Ray, explains what ray tracing is and also describes where to find the latest version of the POV-Ray software.

The next section is a quick start that helps you quickly begin to use the software.

After the quick start is a more in-depth tutorial for beginning POV-Ray users.

Following the beginning tutorial is a scene description language reference that describes the language used with POV-Ray to create an image.

The last sections include some tips and hints, suggested reading, and legal information.

POV-Ray is based on DKBTrace 2.12 by David Buck & Aaron A. Collins

Program Description -- What is Ray Tracing?

The Persistence of Vision Ray Tracer (POV-Ray) is a copyrighted freeware program that allows a user to easily create fantastic, three dimensional, photo-realistic images on just about any computer. POV-Ray reads standard ASCII text files that describe the shapes, colors, textures and lighting in a scene and mathematically simulates the rays of light moving through the scene to produce a photo-realistic image!

No traditional artistic or programming skills are required to use POV-Ray. First, you describe a picture in POV-Ray's scene description language, then POV-Ray takes your description and automatically creates an image of it with perfect shading, perspective, reflections and lighting.

The standard POV-Ray package also includes a large collection of sample scene files that have been created by other artists using the program. These scenes can be

rendered and enjoyed even before learning the scene

description language. They can also be modified to create new scenes.

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

-Easy to use scene description language
-Image size up to 4096 x 4096 pixels and larger
-Large library of stunning example scene files
-Standard include files that pre-define many shapes, colors and textures
-Very high quality output image files (24-bit color.)
-16 bit color display on IBM-PC's using the Sierra HiColor chip
-Create fractal landscapes using height fields
-Spotlights for sophisticated lighting
-Phong and specular highlighting for more realistic-looking surfaces.
-Several image file output formats including Targa, dump and raw
-Wide range of shapes:
-Basic Shape Primitives such as...
Sphere, Box, Ellipsoid, Cylinder, Cone, Triangle and Plane
-Advanced Shape Primitives such as...
Torus (Donut), Hyperboloid, Paraboloid, Bezier Patch, Height Fields (Mountains), Blobs, Quartics, Smooth Triangles (Phong shaded)
-Shapes can easily be combined to create new complex shapes. This feature is called Constructive Solid Geometry (CSG). POV-Ray supports unions, intersections and differences in CSG.
-Multiple objects can be combined into one object using the composite feature
-Objects and shapes are assigned materials, these materials are called textures.
(A texture describes the coloring and surface properties of a shape, its material.)

-Many built-in textures such as...
Marble, Checkerboard, Wood, Bumpy, Agate, Clouds, Granite, Ripples, Waves, Leopard, Wrinkled,
-Users can create their own textures or use pre-defined textures such as...
Mirror, Metals like Chrome, Brass, Gold and Silver, Bright Blue Sky with Clouds, Sunset with Clouds, Sapphire Agate, Jade, Shiny, Brown Agate, Apocalypse, Blood Marble, Glass, Brown Onion , Pine Wood, Cherry Wood
-Combine textures for sophisticated or novel effects

-Display image while computing (not available on all computers)
-Halt rendering when part way through
-Continue rendering a halted partial scene later

Which Version of POV-Ray Should You Use?

There are specific versions of POV-Ray available for three different computers, the IBM-PC, the Apple Macintosh, and the Commodore Amiga.

IBM-PC and compatibles

The IBM-PC version is called POVRAY.EXE and is found in the archive POVIBM.ZIP. It can be run on any IBM-PC with a 386 or 486 CPU and 2 megabytes of memory. A math co-processor is not required, but it is recommended. This version of POV-Ray may be run under DOS, OS\2, and Windows. It will not run under Desqview at this time. A version that runs on IBM-PC's using the 286 CPU will be available soon.

Apple Macintosh

The Apple Macintosh version can be found in the file POVMAC.SIT. It requires a 68020 or better processor, 32 bit Color Quickdraw, and a math co-processor or software emulation of a math co-processor to run.

The math co-processor emulator is needed on the Mac LC and Powerbook 140. The documentation included with the executable describes where to find a math co-processor emulator.

32 bit Color Quickdraw is part of System 7's automatic installation on color Macs. Owners of Mac SE/30 and other non-color Macs will need to use custom installation options to add 32 bit Color Quickdraw.

The installation of QuickTime will enhance POV-Ray's ability to save pictures by allowing it to save in QuickTime compressed formats.

Commodore Amiga

The Commodore Amiga version of POV-Ray can be found in the file POVAMI.LZH. Two executables are supplied, one for computers with a math co-processor, and one for computers without a math co-processor. This version will run on Amiga 500, 1000, 2000, and 3000's and should work under AmigaDOS 1.3 or 2.xx. The Amiga version supports HAM mode as well as

HAM-E and the Firecracker.

UNIX and other systems

POV-Ray is written in highly portable C source code and it can be compiled and run on many different computers. There is specific source code in the source archive for UNIX, X-Windows, VAX, and generic computers. If you have one of these, you can use the C compiler included with your operating system to compile a POV-Ray executable for your own use. This executable may not be distributed. See the source documentation for more details. Users on high powered computers like Suns, SGI, RS-6000's, Crays, and so on use this method to run POV-Ray.

All Versions

All versions of the program share the same ray tracing features like shapes, lighting and textures. In other words, an IBM-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. 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.

Where to find POV-Ray files

POV-Ray is a complex piece of software made up of many files. At the time of this writing, the POV-Ray package was made up of several archives including executables, documentation, and example scene files.

The average user will need an executable for their computer, the example scene files and the documentation. The example scenes are invaluable for learning about POV-Ray, and they include some exciting artwork.

Advanced users, developers, or the curious may want to download the C source code as well.

There are also many different utilities for POV-Ray that generate scenes, convert scene information from one format to another, create new materials, and so on. You can find these files from the same sources as the other POV-Ray files. No comprehensive list of these utilities is available at the time of this writing.

The latest versions of the POV-Ray software are available

from these sources:

Computer Art Forum on CompuServe
POV-Ray headquarters are on CompuServe Comart forum Raytracing section 16. We meet there to share info and graphics and discuss raytracing, fractals and other kinds of computer art. Comart is also the home of the Stone Soup Group, developers of Fractint, a popular IBM-PC fractal program. Everyone is welcome to join in on the action on CIS Comart. Hope to see you there! You can get information on joining CompuServe by calling (800)848-8990. CompuServe access is also available in Japan, Europe and many other countries.

PC Graphics area on America On-Line
There's an area now on America On-Line dedicated to POV-Ray support and information. You can find it in the PC Graphics section of AOL. Jump keyword "PCGRAPHICS". This area includes the Apple Macintosh executables also.

You Can Call Me Ray BBS in Chicago
There is a ray-trace specific BBS in the (708) Area Code (Chicago suburbia, United States) for all you Traceaholics out there. The phone number of this BBS is (708) 358-5611.Bill Minus is the sysop and Aaron Collins is co-sysop of that board, and it's filled with interesting stuff.

The Graphics Alternative BBS in Oakland
For those on the West coast, you may want to find the POV-Ray files on The Graphics Alternative BBS at (510) 524-2780. It's a great graphics BBS run by Adam Schiffman.

Internet
The POV-Ray files are also available over Internet by anonymous FTP from alfred.ccs.carleton.ca (134.117.1.1).

Quick Start

The next section describes how to quickly install POV-Ray and render a sample scene on your computer.

Installing POV-Ray

Specific installation instructions are included with the executable program for your computer. In general, there are two ways to install POV-Ray.

[Note that the generic word "directory" is used

throughout.
Your operating system may use another word (subdirectory, folder, etc.)]

1-- The messy way: Create a directory called POVRAY and copy all POV-Ray files into it. Edit and run all files and

programs from this directory. This method works, but is not recommended.

Or the preferred way:
2-- Create a directory called POVRAY and several subdirectories called
INCLUDE, SAMPLES, SCENES, UTIL.
The file tree for this should look something like this:

```
\--
  |
  +POVRAY --
           |
           +INCLUDE
           |
           +SAMPLES
           |
           +SCENES
           |
           +UTIL
```

Copy the executable file and docs into the directory POVRAY. Copy the standard include files into the subdirectory INCLUDE. Copy the sample scene files into the subdirectory SAMPLES. And copy any POV-Ray related utility programs and their related files into the subdirectory UTIL. Your own scene files will go into the SCENES subdirectory. Also, you'll need to add the directories \POVRAY and \POVRAY\UTIL to your "search path" so the executable programs can be run from any directory.

*Note that some operating systems don't
*have an equivalent to the
*multi-path search command.

The second method is a bit more difficult to set-up, but is preferred. There are many files associated with POV-Ray and they are far easier to deal with when separated into several directories.

Using Sample Scenes

This section describes how to render a sample scene file. You can use these steps to render any of the sample scene files included in the sample scenes archive.

A scene file is a standard ASCII text file that contains a description in the POV-Ray language of a three dimensional scene. The scene file text describes objects and lights in the scene, and a camera to view the scene. Scene files have

the file extension .POV and can be created by any word processor or editor that can save in standard ASCII text format.

Quite a few example scenes are provided with this distribution in the example scenes archive. These have been organized into separate archives within the main archive and are grouped by complexity and category.

These scene files range from a simple red ball on a tiled floor, to a jade panther in a marble temple. They have been created by users of POV-Ray all over the world, and were picked to give examples of the variety of features in POV-Ray. Many of them are stunning in their own right.

The scenes were graciously donated by the artists because they wanted to share what they had created with other users. Feel free to use these scenes for any purpose. You can just marvel at them as-is, you can study the scene files to learn the artists techniques, or you can use them as a starting point to create new scenes of your own.

Here's how to make these sample scenes into images you can view on your computer. We'll use SIMPLE.POV as an example, just substitute another filename to render a different image.

The file SIMPLE.POV was included with this document and should now be in the POVRAY directory.

> Note: The sequence of commands is not the same for every version of POV-Ray. There should be a document with the executable describing the specific commands to render a file.

Make POVRAY the active directory, and then at the command line, type:

```
POVRAY +iSIMPLE.POV +v +w80 +h60<ENTER>
```

POVRAY is the name of your executable, +i<filename.pov> 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. +w and +h set the width and height of the image in pixels. This image will be 80 pixels wide by 60 pixels high.

POV-Ray will read in the text file SIMPLE.POV and begin working to render the image. It will write the image to a file called DATA.TGA. The file DATA.TGA contains a 24 bit image of the scene file SIMPLE.POV. Because many computers can't display a 24 bit image, you will probably have to convert DATA.TGA to an 8 bit format before you can view it on your computer. The docs included with your executable lists the specific steps required to convert a 24 bit file to an 8 bit file.

Command Line Parameters
The following section gives a detailed description of the command-line options.

The command-line parameters may be specified in any order. Repeated parameters overwrite the previous values. Default parameters may also be specified in a file called "povray.def" or by the environment variable "POVRAYOPT". Some examples follow this table.

Table 1-1 Command Line Parameters

Parameter.....	.....Range.....	...Description.....
-?		Turn on/off feature off
-?(val)		Turn feature on with parameter val.
+?		Turn feature on.
+i<filename>	Varies w/ sys	Input scene file name, generally ends in .pov.
+o<filename>	Varies w/ sys	Output image filename.
+w#####	1-32,768	Width of image in pixels.
+h#####	1-32,768	Height of image in pixels.
+v(#)	Varies w/sys	Display image stats while rendering.
+d(??)	Varies w/sys	Display image graphically while rendering (Not avail on all vers).
+x		Allow abort with keypress.(IBM-PC).
+f(?)	t, d or r	Output file format: Targa, dump or raw.
-f		Disable file output.
+s#####	1-32,768	Start line for tracing a portion of a scene.
+e#####	1-32,768	End line for tracing a portion of a scene.
+c		Continue an aborted partial image.
+p		Pause and wait for keypress after tracing image.
+b#####	Varies w/ sys	Output file buffer size.
+a#####	0.0 to 1.0	Render picture with anti-aliasing,or "smoothing", on. Lower values cause more smoothing.
+q#	0 to 9	Image quality: 9 highest(default) to 0 lowest.
+l<path>	Varies w/ sys	Library path: POV-Ray will search for files in the directory listed here. Multiple lib paths may be specified.

Normally, you don't need to specify the +f? option. The default setting will create the correct format image file for your computer. The docs included with the executable specify which format is used.

You can disable image file output by using the command line option -f. This is only useful if your computer has display options and should be used in conjunction with the +p option. If you disable file output using -f, there will be no record kept of the image file generated. This option is not normally used.

Unless file output is disabled (-f) POV-Ray will create an image file of the picture. This output file describes each pixel with 24 bits of color information. Currently, three output file formats are directly supported:

+ft - Uncompressed Targa-24 format (IBM-PC Default)
+fd - Dump format (QRT-style)
+fr - Raw format - one file each for Red, Green and Blue.

If the +d option is used and your computer supports a graphic display, then the image will be displayed while the program performs the ray tracing. On most systems, the picture displayed is not as good as the one created by the post-processor because it does not try to make optimum choices for the color registers.

The +d parameters are listed in the executable documentation.

-ifilename Set the input filename
-ofilename Set output filename

The default input filename is "object.pov". The default output filename is "data" and the suffix for your default file type.

IBM-PC default file type is Targa, so the file is "data.tga".

Amiga uses dump format and the default outfile name is "data.dis".

Raw mode writes three files, "data.red", "data.grn" and "data.blu". On IBM-PC's, the default extensions for raw mode are ".r8", ".g8", and ".b8" to conform to Piclab's "raw" format. Piclab is a widely used free-ware image processing program. Normally, Targa files are used with Piclab, not raw files.

+a[xxx] Anti-alias - xxx is an optional tolerance level

(default 0.3)
-a Don't anti-alias

Anti-aliasing is a technique used to make the ray traced image look smoother. Often the color difference between two objects creates a "jaggy" appearance. When anti-aliasing is turned on, POV-Ray attempts to "smooth" the jaggies by shooting more rays into the scene and averaging the results. This technique can really improve the appearance of the final image. Be forewarned though, anti-aliasing drastically slows the time required to render a scene since it has do many more calculations to "smooth" the image. Lower numbers mean more anti-aliasing and also more time. Use anti-aliasing for your final version of a picture, not the rough draft.

The +a option enables adaptive anti-aliasing. The number after the +a option determines the threshold for the anti-aliasing.

If the color of a pixel differs from its neighbor (to the left or above) by more than the threshold, then the pixel is subdivided and super-sampled.

If the anti-aliasing threshold is 0.0, then every pixel is super-sampled. If the threshold is 1.0, then no anti-aliasing is done.

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.

The super-samples are jittered to introduce noise and make the pictures look better. Note that the jittering "noise" is non-random and repeatable in nature, based on an object's

3-D orientation in space. Thus, it's okay to use anti-aliasing for animation sequences, as the anti-aliased pixels won't vary and flicker annoyingly from frame to frame.

+bxxx Use an output file buffer of xxx kilobytes. (if 0, flush and write the output file on every line - this is the default)

The +b option allows you to assign large buffers to the output file. This reduces the amount of time spent writing to the disk and prevents unnecessary wear. If this parameter is zero, then as each scanline is finished, the line is written to the file and the file is flushed. On most systems, this operation insures that the file is written to the disk so that in the event of a system crash or other catastrophic event, at least part of the picture has been stored properly and retrievably on disk.(see also the +c option below.)
+b30 is a good value to use to speed up small renderings.

+c Continue Rendering

If you abort a render while it's in progress or if you used the -exxx option (below) to end the render prematurely, you can use the +c option to continue the render when you get back to it. This option reads in the previously generated output file, displays the image to date on the screen, then proceeds with the raytracing. In many cases, this feature can save you a lot of rendering time when things go wrong.

+sxxx Start tracing at line number xxx.
+exxx End tracing at line number xxx.

The +s option allows you to begin the rendering of an image at a specific scan line. This is useful for rendering part of a scene to see what it looks like without having to render the entire scene from the top. You can also use this option to render groups of scanlines on different systems and concatenate them later.

WARNING: Image files created on with different executables on the same or different computers may not look exactly the same due to different random number generators used in some textures. If you need to merge image files from different

computers contact the authors for more info.

If you are merging output files from different systems, make sure that the random number generators are the same. If not, the textures from one will not blend in with the textures from the other.

+q# Rendering quality

The +q option allows you to specify the image rendering quality, for quickly rendering images for testing. The parameter can range from 0 to 9. The values correspond to the following quality levels:

```
0,1  Just show quick colors. Ambient lighting only.
     Quick colors are specified outside the texture block
     and affect only the low quality renders.
2,3  Show Diffuse and Ambient light
4,5  Render shadows
6,7  Create surface textures
8,9  Compute reflected, refracted, and transmitted rays.
```

The default is +q9 (maximum quality) if not specified.

+l<path> - Library search path
The +l option may be used to specify a "library" pathname to look in for include, parameter and image files. Up to 10 +l options may be used to specify a search path. The home (current) directory will be searched first followed by the indicated library directories in order.

Default Parameter File and Environment Variable
You may specify the default parameters by modifying the file "povray.def" which contains the parameters in the above format. This filename contains a complete command line as though you had typed it in, and is processed before any options supplied on the command line are recognized. You may put commands on more than one line in the "povray.def" file.

Examples:

```
  povray +ibox.pov +obox.tga +v +x +w320 +h200

     +ibox.pov = Use the scene file box.pov for input
     +obox.tga = Output the image as a
                 Targa file to box.tga
     +v = Show line numbers while rendering.
```

```
     +x = Allow key press to abort render.
     +w320 = Set image width to 320 pixels
     +h200 = Set image height to 200 pixels
```

Some of these parameters could have been put in the POVRAYOPT environment variable to save typing:

```
  SET POVRAYOPT = +v +x +w320 +h200
```

Then you could just type:

```
  povray +ibox.pov +obox.tga
```

Or, you could create a file called POVRAY.DEF in the same directory as the scene file box.pov that contains the command line options.

POVRAY.DEF contains "+v +x +w320 +h200"

Then you could also type:

```
  povray +ibox.pov +obox.tga
```

With the same results. You could also create an option file with a different name and specify it on the command line:

For example, if QUICK.DEF contains "+v +x +w80 +h60"

Then you could also type:

```
  povray +ibox.pov +obox.tga QUICK.DEF
```

When POV-Ray sees QUICK.DEF, it will read it in just as if you typed it on the command line.

The order that the options are read in for the IBM-PC version are as follows:

```
  POVRAYOPT environment variable

  POVRAY.DEF in current directory or,
             if not found, in library path

  Command line and command line option files
```

For example, +v in POVRAY.DEF would override -v in

POVRAYOPT. +x on the command line would override -x in POVRAY.DEF and so on.

Other computer's versions read in the POVRAY.DEF file before the POVRAYOPT environment variable.

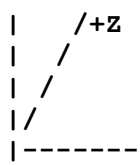
Beginning Tutorial

This section describes how to create a scene using POV-Ray's scene description language and how to render this scene.

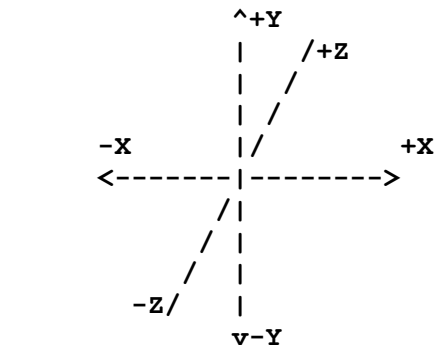
Your First Image

Let's create the scene file for a simple picture. Since raytracers thrive on spheres, that's what we'll render first.

First, we have to tell POV-Ray where our camera is and where it's 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:



he the negative values of the axes point the other direction, as follows:



Using your personal favorite text editor, create a file

called "picture1.pov". Now, type in the following (note: The

input is case sensitive, so be sure to get capital and lowercase letters correct):

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

camera {
  location  <0 1 -3>
  direction <0 0 1.5>
  up        <0 1 0>
  right     <1.33 0 0>
  look_at   <0 1 2>
}
```

The first include statement reads in definitions for various useful colors. The second and third include statements read in some useful shapes and textures respectively. When you get a chance, have a look through them to see but a few of the many possible shapes and textures available.

Include files may be nested, if you like but only 10 files may be active at once.
You may have as many include files as needed in a scene file. Include files may themselves contain include files, but you are limited to declaring includes only 10 "deep".

Filenames specified in the include 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" (library path) options active. This would facilitate keeping all your "include" (.inc) files such as shapes.inc, colors.inc, and textures.inc in an "include" subdirectory, and giving an "-l" option on the command line to where your library of include files are.

The camera declaration describes where and how the camera sees the scene. It gives X, Y, Z coordinates to indicate the position of the camera and what part of the scene it is pointing at. And the camera definition has direction vectors to describe the properties of the camera like the aspect ratio and lens length. Details can be found in the section on cameras.

Briefly, "location <0 1 -3>" places the camera three units back from the center of the ray-tracing universe which is at

<0 0 0>. Remember that by default +Z is into the screen and -Z is back out of the screen.

The line "direction <0 0 1.5>" says we're looking in the +Z direction with a "lens length" of 1.5 units. 1.5 is used instead of the default 1.0 to avoid "wide-angle" distortion that occurs when the lens is too short. In other words, the value 1.5 gives a slight "tele-photo" effect which reduces perspective distortion. The default value of 1 is used for historical reasons. The direction vector should be higher in most cases to avoid perspective distortion.

The parameter "up <0 1 0>" tells POV-Ray that "up" or the top of the image is in the positive Y axis direction.

The parameter "right <1.33 0 0>" indicates that "right" or the right side of the image is in the direction of the positive X axis.

The size of the up and right values also indicate the aspect ratio of the scene. Most scenes are 1.33 times wider than they are high. This corresponds to the popular computer graphics image size of 640 pixels wide by 480 pixels high.

Finally, "look_at <0 1 2>" rotates the camera to point at X, Y, Z coordinates <0 1 2>. A point 6 units in front of and 1 unit higher than the camera. The look_at point should be the center of attention of your image.

Now that the camera is set up to record the scene, let's place a red sphere into the scene:

```
object {
  sphere { <0 1 2> 1 }
  texture {color Red } // Red is pre-defined in COLORS.INC
                       // You could also use "color red 1"
}
```

The center of this sphere at Z=2 is approx. 5 units in front of the camera at Z=-3. It has a radius of 1 unit. Since the radius is 1/2 the width of a sphere, the sphere 2 units wide.

Note that any parameter that changes the appearance of the surface (as opposed to the shape of the surface) is called a texture parameter and must be placed into a texture { ... }

block. In this case, we are just setting the color of the sphere using a pre-defined color from COLORS.INC. We could manually set this color by specifying the amount of red, green, and blue in the color like this:

```
    color red # green # blue #
```

Values for red, green, and blue are between 0-1. If one is not specified it is assumed to be 0. Colors are explained in more detail later.

One more detail, we need a light source. Until you create one, there is no light in this virtual world:

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

This white light is 2 units to our right, and 4 units above the camera. The light_source is invisible, it only casts light, so no texture is needed.

That's it! Close the file and render a small picture of it using this command:

```
povray -w160 -h120 +p +x +d1 -v -ipicture1.pov
```

If your computer does not use the command line, see the executable docs for the correct command to render a scene.

You may set any other command line options you like, also. The scene is output to the image file DATA.TGA (or some suffix other than TGA if your computer uses a different file format). You can convert DATA.TGA to a GIF image using the commands listed in the docs included with your executable.

Phong Highlights

You've now rendered your first picture. Let's add a nice little specular highlight (shiny spot) to the sphere. It gives it that neat "computer graphics" look. Change the definition of the sphere to this:

```
object {
  sphere{ <0 1 2> 1 }
   texture {
    color Red
    phong 1
   }
}
```

Now render this the same way you did before. 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. Phong can be between 0 and 1.

Simple Textures

POV-Ray has some very sophisticated textures built-in and ready to use. Change the definition of our sphere to the following and then re-render it:

```
object {
   sphere { <0 1 3> 1 }
   texture {
     DMFWood1 // Pre-defined texture from textures.inc
     scale <.2 .2 1> // Shrink the wood along the
                     // X & Y axes and leave
                     // the Z axis unchanged.
     phong 1
   }
}
```

The texture patterns are set up by default to give you 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. By scaling the wood by <.2 .2 1>, we shrink the texture to on the X and Y axes to give us about five bands. The Z axis is scaled by 1, which leaves it unchanged.

Please note that "one feature across a unit sphere" is not a hard and fast rule. It's only meant to give you a rough idea for the scale to use for a texture.

One note about the scale operation. You can magnify or shrink along each direction separately. The first term tells how much to magnify or shrink on the X axis or in the left-right direction. The second term controls the Y axis or up-down direction and the third term controls the Z axis or front-back direction. Scale values larger than 1 will stretch an element. Scale values smaller than one will squish an element. And scale value 1 will leave and element unchanged.

Look through the TEXTURES.DOC file to see what textures are defined in TEXTURES.INC and try them out. Just insert the

name of the new texture where DMFWood1 is now and re-render your file.

More Shapes

So far, we've just used the sphere shape. There are many other types of shapes that can be rendered by POV-Ray. Let's try out a computer graphics standard - "The Checkered Floor." Add the following object to your .pov file:

```
object {
   plane {<0 1 0> 0 }
   texture {
    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 you 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=0 so that the sphere at Y=1, radius= 1, is resting on it. The texture uses the checker color pattern and specifies that the two colors red and blue be used in the checker pattern.

Looking at the floor, you'll notice that the wooden ball casts a shadow on the floor. Shadows are calculated very accurately by the ray tracer and creates precise, sharp shadows. In the real world, penumbral or "soft" shadows are often seen. POV-Ray does not simulate penumbral shadows, but they can be "faked." See the advanced section for tips if you need soft shadows.

Quadric Shapes

Another kind of shape you can use is called a quadric surface. There are many types of quadric surfaces. These are all described by a certain kind of mathematical formula (see the section on Quadrics in the next chapter). They include cylinders, cones, paraboloids (like a satellite dish), hyperboloids (saddle-shaped) and ellipsoids.

All quadrics except for ellipsoids are infinite in at least one direction. For example, a cylinder has no top or bottom

- it goes to infinity at each end. Quadrics all have one

common feature, if you draw any straight line through a quadric, it will hit the surface at most twice. A torus (donut), for example, is not a quadric since a line can hit the surface up to four times going through.

Let's render a quadric. While we're at it, we'll add a few features to the surface. Add the following definition to your scene file:

```
object {
  quadric { Cylinder_Y }
  texture {
    color green .5 // This color is 50% green
    reflection  .5
  }
  scale <.4  .4  .4>
  translate <2 0 5>
}
```

This object is a cylinder along the Y (up-down) axis. It's green in color and has a mirrored surface, reflection 0.5, this means that half the light coming from the cylinder is reflected from other objects in the room. A reflection of 1.0 is a perfect mirror.

The object has been shrunk by scaling it by <0.4 0.4 0.4>. Note that since the cylinder is infinite along the Y axis, the middle term is kind of pointless. One four tenths of infinity is still infinity.

Note: Scene elements can't be scaled by zero, that will cause a divide by zero floating point error. Use a scale value of 1 if you wish to leave a component unchanged.

Finally, the cylinder has been moved back and to the right by the translate command so you can see it more clearly.

Scene Description Language Reference

The Scene Description Language allows the user to describe the world in a readable and convenient way.

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 humans to read.

There are two types of comments in POV-Ray:

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

Anything on a line after a double slash // is ignored by the ray tracer. You can have scene file information on the line in front of the comment, as in:

```
object { // this is an object
```

The other type of comment is used for multiple lines.

```
/* These lines
    Are ignored
    By the
    Raytracer */
```

This type of comment starts with /* and ends with */ everything in-between is ignored. This can be useful if you want to temporarily remove elements from a scene file. Just comment them out, comments and all.

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

/*...*/ comments can "comment out" lines containing the 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 */
 */
```

Include Files

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

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

at any point in the input file . The filename must be enclosed in double quotes and 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 the same as actually

cutting and pasting the entire contents of this file into your scene.

Include files may be nested. You may have at most 10 include'd files per scene trace, whether nested or not.

Generally, include files have data for scenes, but are not scenes in themselves. Scene files should end in .pov.

Float
A float is a floating point number, or a number with a decimal point.

Floats are represented by an optional sign (-), some digits, an optional decimal point, and more digits. This version supports scientific notation for exponents. The following are all valid floats:
 1.0 -2.0 -4 34 3.4e6 2e-5 .3 0.6

Vector
Vectors are arrays of three or more floats. They are bracketed by angle brackets (< and >), and the three terms usually represent x, y, and z. For example:

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

Vectors often refer to relative points. For example, the vector:

```
<1 2 3>
```

means the point that's 1 unit to the right, 2 units up, and 3 units in front 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.

Declare
The parameters used to describe the scene elements can be tedious to use at times. Some parameters are often repeated and it seems wasteful to have to type them over and over again. To make this task easier, the program allows users to create synonyms for a pre-defined set of parameters and use them anywhere the parameters would normally be used. For example, the color white is defined in the POV-Ray language as:

```
   color red 1 green 1 blue 1
```

This can be pre-defined in the scene as:

```
#declare White = color red 1 green 1 blue 1
```

and then substituted for the full description in the scene file, for example:

```
object {
   sphere { <0 0 0> 1 }
   texture { color red 1 green 1 blue 1 }
}
```

becomes:

```
#declare White = color red 1 green 1 blue 1

object {
   sphere { <0 0 0> 1 }
   texture { color White }
}
```

This is much easier to type and to read. The pre-defined element may be used many times in a scene.

You use the keyword "declare" to pre-define a scene element and give it a one-word synonym. This pre-defined scene element is not used in the scene until you invoke its synonym. Shapes, textures, objects, colors, numbers and more can be predefined.

Pre-defined elements may be modified when they are used, for example:

```
#declare Mickey = // Pre-define a union called Mickey
   union {
     sphere { <0 0 0> 2 }
     sphere { <-2 2 0> 1}
     sphere { <2 2 0> 1}
   }

// Use Mickey
object{
   union {
      Mickey
      scale <3 3 3>
      rotate <0 20 0>
      translate <0 8 10>
   }
   texture {
     color red 1
```

```
     phong .7
   }
}
```

This scene will only have one "Mickey", the Mickey that is described doesn't appear in the scene. Notice that Mickey is scaled, rotated, translated, and a texture is added to it. The Mickey synonym could be used many times in a scene file, and each could have a different size, position, orientation, and texture.

Declare is especially powerful when used to create a complex shape or object. Each part of the shape or object is defined separately using declare. These parts can be tested, rotated, sized, positioned, and textured separately then combined in one shape or object for the final sizing, positioning, etc. For example, you could define all the parts of a car like this:

```
#declare Wheel = shape_type {...}
#declare Seat = shape_type {...}
#declare Body = shape_type {...}
#declare Engine = shape_type {...}
#declare Steering_Wheel = shape_type {...}

#declare Car =
  union {
     shape_type { Wheel translate <1 1 2>}
     shape_type { Wheel translate <-1 1 2>}
     shape_type { Wheel translate <1 1 -2>}
     shape_type { Wheel translate <-1 1 -2>}
     shape_type { Seat translate <.5 1.4 1>}
     shape_type { Seat translate <-.5 1.4 1>}
     shape_type { Steering_Wheel translate <-.5 1.6 1.3>}
     shape_type { Body texture { Brushed_Steel } }
     shape_type { Engine translate <0 1.5 1.5>
   }
```

and then it like this:

```
// Here is a car
object {
   union { Car }
   translate <4 0 23>
}
```

Notice that the Wheel and Seat are used more than once. A

declared element can be used as many times as you need.
Declared elements may be placed in "include" files so they can be used with more than one scene.
Declare can be used anywhere in a scene or include file.

There are several files included with POV-Ray that use declare to pre-define many shapes, colors, and textures. See the archive INCLUDE for more info.

NOTE: Declare is not the same as the C language's define. Declare creates an internal object of the type specified that POV-Ray can recognize when it is prefaced by the object type, while define substitutes like a macro. The only declared object type that doesn't need to be prefaced with a keyword is numeric, which can be used wherever a number is expected.

What can be declared?
Here's a list of what can be declared, how to declare the element, and how to use the declaration. See the reference section for element syntax.

Objects:

```
  #declare Tree = object {...}

  object {
    Tree
    (object_modifiers)
  }
```

Composite Objects:

```
  #declare Duck = composite {...}

  composite {
    Duck
    (object_modifiers)
  }
```

Shapes:

```
   (Any shape type may be declared, sphere, box,
height_field, blob,
    etc.)
  #declare Disk = shape_type {...}

  object {
    shape_type { Disk (shape_modifiers) }
    (object_modifiers)
  }
```

```
 Specific Shape Example:
  #declare BigBox = box {<0 0 0> <10 10 10>}

  object {
    box { BigBox (shape_modifiers) }
    (object_modifiers)
  }


CSG Shapes:
  Declare and use these just like a normal shape type. The
CSG shape types are union, intersection, and difference.

  #declare Moose = csg_shape_type {...}

  object {
    shape_type { Moose (shape_modifiers) }
    (object_modifiers)
  }

Textures:
  #declare Fred = texture {...}

  object {
    sphere { <0 0 0> 1 }
    texture {
      Fred
      (texture_modifiers)
    }
  }

Layered textures:
  #declare Fred =
     texture {...}
     texture {...}
     texture {...} (etc.)

  object {
    sphere { <0 0 0> 1 }
    texture {
      Fred
      (texture_modifiers)
    }
  }
```

```
Colors:
  #declare Fred = color red # green # blue # alpha #

  object {
    sphere { <0 0 0> 1 }
    texture {
       color Fred
    }
  }

Numeric Values:
  #declare Fred  = 3.45
  #declare Fred2 = .02
  #declare Fred3 = .5

object { // Use the numeric value synonym
         // anywhere a number would go
   sphere { <-Fred 2 Fred> Fred }
   texture {
      color red 1
      phong Fred3
   }
}

Camera:
  #declare Fred = camera {..}

  camera { Fred }

Vectors:
   #declare Fred = <9 3 2>
   #declare Fred2 = <4 4 4>

   object {
     sphere { Fred 1 }
     scale Fred2
   }

Light sources:
   #declare Light1 = light_source {...}

   object {
     light_source { Light1 (modifiers) }
   }


Camera
Every scene in POV-Ray must have one and only one camera
```

definition. The camera definition describes the position, angle and properties of the camera viewing the scene. POV-Ray uses this definition to do a simulation of the camera in the ray-tracing universe and "take a picture" of your scene.

The camera simulated in POV-Ray is a pinhole camera. Pinhole cameras have a fixed focus so all elements of the scene will always be perfectly in focus. The pinhole camera is not able to do soft focus or depth of field effects.

The camera block tells the ray tracer the location and orientation of the camera or viewpoint. The camera is described by several vectors -
Location, Direction, Up, and Right, Sky, and Look_at.

Location specifies the position of the camera in X, Y, Z coordinates. The camera can be positioned at any point in the ray-tracing universe.

  Telephoto and Wide-Angle
Direction serves two purposes. It is a vector describing what direction the camera is pointing. The direction vector is usually set to point straight ahead. The look_at vector described below is used to point the camera. It is also used as a kind of lens length setting. Shorter is more wide angle, longer is more telephoto.

Be careful with short direction vector lengths like 1.0 and less. You'll end up with distortion on the edges of your images. Objects will appear to be shaped strangely. If this happens, move the location back and make the direction vector longer.

The Up vector describes the "up" direction of the camera to POV-Ray. It tells POV-Ray where the top of your screen is.

The Right vector 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 also determines the "handedness" of the coordinate system in use. A normal (left-handed) right statement would be:

right <1.33 0 0>

To use a right-handed coordinate system, as is popular in some CAD programs and some other ray-tracers use a right

like:

right <-1.33 0 0>

Some CAD systems, like AutoCAD, also have the assumption that the Z axis is the "elevation" and is the "up" direction instead of the Y axis. If this is the case you will want to change your "up" statement to:

up <0 0 1>

Together the Up and Right vectors define the aspect ratio of the image being rendered. These vectors are described in more detail later in this document.

Here's a basic declaration of a camera:

```
camera {
  location  <0 0 0>
  direction <0 0 1>
  up        <0 1 0>
  right     <1.33 0 0>
}
```

The above vectors are set to their default values. If you do not specify a value then these defaults are used. For instance, if your camera definition didn't have the line "up <0 1 0>", the up vector would still be set to "up <0 1 0>" because POV-Ray does that automatically, by default.

These four vectors may be specified in any order. They should be specified before the modifiers sky and look_at which are described below.

  Sky and Look_at

If the camera always pointed straight along an X, Y or Z axis this definition format would be easy enough. In almost all cases however, the camera angle is much more interesting. This format becomes cumbersome then because its difficult to compute the vectors by hand.

Fortunately, POV-Ray doesn't require you to do any vector computations. The vectors Sky and Look_at make it easy to point the camera just the direction you'd like.

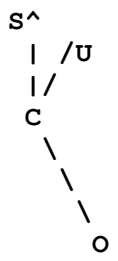
The look_at keyword defines the specific point the camera should be pointing at. The camera is rotated as required to

point the "center of the lens" directly at that point.

Normally, you will use the look_at keyword, not the direction vector to point the camera. The look_at command is the best way to rotate the camera to point at a scene element. Also, it should always be last in the camera definition. Rotate and translate may be used in the camera definition, but they should be used before look_at or the camera will probably not remain looking at the point specified.

The sky keyword tells the camera where the sky in the ray-tracing universe is. POV-Ray uses the sky vector to keep the camera's up direction aligned as closely as possible to the sky.

One subtle point: the sky direction is not necessarily the same as the up direction. For example, consider the following situation:



If you said that the camera C has a sky direction S and is looking at O, the up direction will not point to the sky. Imagine hanging your camera from a string attached at the top of the camera. The string points to the sky. The up keyword is like putting an antenna on your camera. If you point the camera downwards using look_at, the antenna will no longer point straight up.

Note that the default value for sky is <0 1 0>. Sky affects the other vectors when you use look_at.

Sky must be defined before the look_at point.

Aspect Ratio

As mentioned before, the length of the up and right vectors controls the aspect ratio of the image. Typically computer

screens have an aspect ratio (width to height ratio) of 4:3. Most of the sample images distributed with POV-Ray were designed to fill the "landscape" orientation of a typical screen and they generally define "up <0 1 0>" and "right <1.33 0 0>". Thus the right-to-up ratio is 1.33:1 or about 4:3. A tall, slim picture might use "up <0 5 0> right <1 0 0>".

Note that this is the ratio of the overall image and is independent of pixel size or shape. If you assume square pixels then the command line values "+w" and "+h" should also have this same ratio. If you have non-square pixels you should adjust the "+w" & "+h" values... not the up & right values.

Suppose we have "up <0 1 0> right <1.33 0 0>" giving a 4:3 ratio. On an IBM VGA 640x480 mode the pixels are square so you would specify "+w640 +h480" on the command line. Note that 640:480 = 4:3 = 1.33:1. However there is a mode on Amiga that has wide, flat pixels which fills the screen when 320 by 400 pixels are used. You would still keep the same up & right vectors because the image itself should still look 4:3 wide. But you would specify "+w320 +h400" on the command line to compensate for non-square pixels.

Translating and Rotating the Camera

Also, the "translate" and "rotate" commands 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>
    translate <5  3  4>
    rotate   <30 60 30>
  }
```

In this example, the camera is created, then translated to another point in space and rotated by 30 degrees about the X axis, 60 degrees about the Y axis, and 30 degrees about the Z axis.

light_source

Syntax: light_source { color ...}

Light sources are treated like shapes, but they are invisible points from which light rays stream out. They light objects and create shadows and highlights. Because of the way ray tracing works, lights do not reflect. You can use many light sources in a scene, but each light source used will increase rendering time. The brightness of a light is determined by its color. A bright color is a bright light, a dark color, a dark one. White is the brightest possible light , Black is completely dark and Grey is somewhere in the middle.
The syntax for a light source is:

```
  light_source { <X Y Z> color red # green # blue #}
```

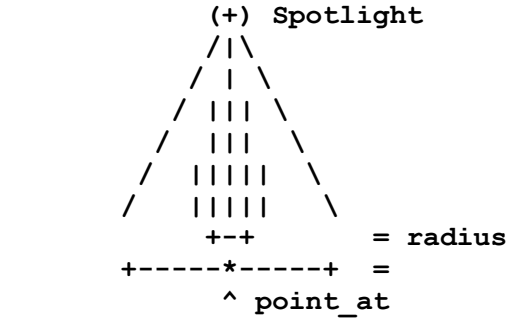
Where X, Y and Z are the coordinates of the location and color , is any color or color identifier. For example,

```
  light_source { <3 5 -6> color Grey}
```

Is a Grey light at X=3, Y=5, Z=-6.

```
spotlight light_source
Syntax:   light_source {  <center> color red # green # blue
#
      spotlight
      point_at <point>
      radius #
      falloff #
      tightness #
     }
```

A spotlight is a point light source where the rays of light are constrained by a cone. The light is bright in the center of the spotlight and falls off/darkens to soft shadows at the edges of the circle.



The center and color of the spotlight is specified the same way as a normal light_source.

Point_at <point> is the location that the cone of light is aiming at.

The radius # is the radius in degrees of the bright circular hotspot at the center of the spotlight's area of affect.

The falloff # is the falloff radius of the spotlight area in degrees. This is the value where the light "falls off" to zero brightness. Falloff should be larger than the radius. Both values should be between 1 and 180.

The tightness value is how fast the light falls off at the edges. Low values will make the spot have very soft edges. High values will make the edges sharper, the spot "tighter".

Spotlights may used anyplace that a normal light source is used. Like normal light sources, they are invisible points. They are treated as shapes and may be included in CSG shapes.

Example:

```
// This is the spotlight.
   light_source {
      <10 10 0>
      color red 1 green 1 blue 0.5
      spotlight
      point_at <0 1 0>
      tightness 50
      radius 11
      falloff 25
     }
```

no_shadow

Syntax: object { shape {...} texture {...} no_shadow }

You may specify the no_shadow keyword in object and that object will be transparent to all light sources. It will not cast a shadow. This is useful for special effects and for creating the illusion that a light source actually is visible.
Here are two examples of "visible" light sources using no_shadow:

```
// This a light masquerading as the sun.
// Note: The texture does not affect the light,
//      only the sphere
 object {
   union { // the union is a CSG shape type,
           // see CSG for more info
   sphere { <0 50 200> 100 }
   light_source { <0  50 200>  color red 1 green .5  }
  }
   texture {
      color red 1 green .5
      ambient 1
      diffuse 0
   }
 no_shadow
 }
// This is the spotlight trying to be a small sphere.
 object {
   union { // the union is a CSG shape type,
           // see CSG for more info
   sphere { <10 10 0> 0.5 }
   light_source {
      <10 10 0>
      color red 1 green 1 blue 0.5
      spotlight
      point_at <0 1 0>
      tightness 50
      radius 11
      falloff 25
   }
  }
 texture { color White ambient 1 diffuse 0 }
 no_shadow
 }
```

If no_shadow were not used here, the light inside the spheres would not escape the sphere and would not show up in the scene. The sphere would, in effect, cast a shadow over everything.

Note that it does not mean 'this light_source casts no shadow'. No_shadow affects the entire object and means 'this object casts no shadow'." Although the no_shadow keyword is most often used with light_sources, it can be used in any object.

Objects

An object is made out of three basic elements: a shape, a texture, and transformations.

The shape is the form of the object, like a sphere, a cone or a cylinder. The shape can be any standard shape type or it can be a CSG shape type, like union, or intersection. The CSG shape types create one shape that is made up of multiple shapes. Any shape may have its own texture.

The texture describes what a shape or object looks like, ie. its material.

The transformations tell the ray tracer how to move, size or rotate the shape and/or the texture in the scene.

Here's the basic layout for an object:

```
object {
  shape   {
    (shape parameters)
    [shape transformations]
  }
  [shape only transformations]

  texture {
     (texture parameters)
     [texture transformations]
  }
  [object,shape and texture, transformations]
}
```

For example:

```
object {
  sphere { <1 2 3> 2 }
  texture { marble scale <2 2 2> }
  scale <5 5 5>  // This transformation scales
                 // both shape and texture
}
```

CSG shapes are the same:

```
object {
  intersection { Disk_Y }
  scale < .2 1 .2 > // This scales shape only
  texture {
```

```
     marble
     scale <2 2 2> // This scales texture only
  }
  scale <2 1 3>  // This scales both shape and texture
}
```

Quick Render Color

When used outside of the texture block in an object, the color keyword determines what color to use for a low quality rendering when textures are ignored. When the quality command option is set to +q0, +q1, ..., +q5, the program will use the quick color for objects instead of the texture. Normally, +q is set to +q9 and the quick color won't affect the object.

Example:

```
object {
   shape { ... }
   texture { ... }
   color red # green # blue #   // This is the quick
                                // render color
}
```

Transformations

Often, vectors are used as a convenient notation but don't represent a point in space. This is the case for the transformations scale, rotate, and translate. Scale sizes a shape, texture or object. Translate moves a shape, texture or object. And rotate turns a shape, texture or object.

Note: Transformations are relative to the current transformation of the scene element. That is, they are additive. For example, scale <2 0 0> scale <2 0 0> actually scales by 4 in X direction.

Most scene elements like objects, shapes, and lights can be translated, rotated, and scaled just like surfaces.

If 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, or scale in an object before a texture, the texture will not be transformed. If the scale, translate, or rotate is after the texture, the texture will be transformed.

For example,

```
   object {
     sphere { <0 0 0> 1}
```

```
texture { marble }
```

```
    scale <3 3 3> // This scale affects the
                  // shape and texture
  }
  object {
    sphere { <0 0 0> 1}
    scale <3 3 3> // This scale affects the
                  // shape only
    texture { marble }
  }
```

Scale
Syntax:
scale <xs ys zs>

Scale the element by xs units in the left/right direction, ys units in the up/down direction and zs units in the front/back direction. Scale is used to "stretch" or "squish" an element. Values larger than 1 stretch the element on that axis. Values smaller than one are used to squish the element on that axis. 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.

Rotate
Syntax:
rotate <xr yr zr>

Rotate the element xr degrees about the X axis, then yr degrees about the Y axis, then zr degrees about the Z axis. Rotate turns the object relative to its current position. Multiple rotate values may used.

Note that the order of the rotations does matter and you should use multiple rotation statements to get a correct multiple rotation. You should only rotate on one axis at a time. 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.
```

Note also that rotate always rotates around the point <0 0 0> so if you translate the object before rotating, it will be rotated "in orbit" around <0 0 0>.

Translate
Syntax:
translate <x y z>

Move the element x units to the right, y units up, and z units away from us. Translate moves the element relative to it's current position. For example,

```
  object {
    sphere { <10 10 10> 1 }
    texture { color Green }
    translate <-5 2 1>
  }
```

Will move the sphere from <10 10 10> to <5 12 11> not <5 2 1>. Translating by zero will leave the element unchanged on that axis. For example,

```
  object {
    sphere { <10 10 10> 1 }
    texture { color Green }
    translate <0 0 0>
  }
```

Will not move the sphere at all.

Multiple translates may be used.

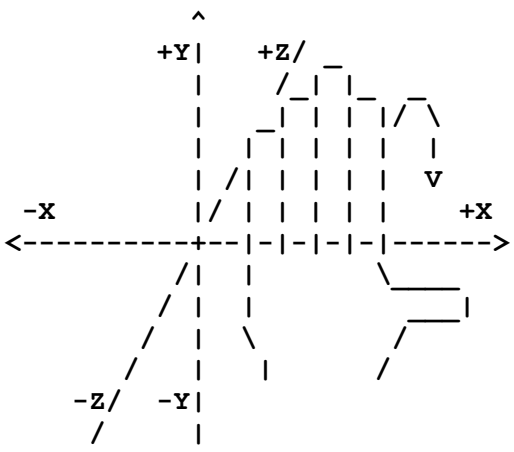
Transformation order

Generally, this is the order that you should perform these operations in. If you scale after you translate, the scale will multiply the translate amount. For example:

```
  translate < 5 6 7>
  scale <4 4 4>
```

Will translate to 20, 24, 28 instead of 5, 6, 7. Be careful when transforming to get the order correct for your purposes.

"Handedness" -- Rotation Direction
To work out the rotation directions, you must perform the famous "Computer Graphics Aerobics" exercise. Hold up your left hand. point your thumb in the positive direction of the axis you want to rotate about. Your fingers will curl in the positive direction of rotation. This is the famous "left-hand coordinate system".



In this 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 you want to use a right hand system, as some CAD systems such as AutoCAD do, the right vector in the camera needs to be changed. See the detailed description of the camera, and use your right hand for the "Aerobics".

Shapes

Shapes describe the shape of an object without mentioning any surface characteristics like color, surface lighting and reflectivity.

Spheres
Since spheres are so common in ray traced graphics, A sphere primitive has been added to the system. This primitive will render much more quickly than the corresponding quadric shape. The syntax is:

```
sphere { <center> radius }
```

You can also add translations, rotations, and scaling to the sphere. For example, the following two objects are

identical:

```
object{
  sphere { < 0.0 25.0 0.0 > 10.0 }
    texture {
      color Blue
      phong .5
    }
}

object {
  sphere { < 0.0 0.0 0.0 > 1.0
     scale <10.0  10.0  10.0>
     translate <0.0  25.0  0.0>
  }
  texture {
     color Blue
      phong .5
  }
}
```

Note that Spheres may only be scaled uniformly. You cannot use:

```
     scale <10.0 5.0 2.0>
```

on a sphere. If you need oblate (flattened) spheroids, use a quadric "Ellipsoid" shape instead, as it can be arbitrarily scaled in any dimension.

Planes

Another primitive provided to speed up the raytracing is the plane. This is a flat infinite plane. To declare a plane, you specify the direction of the surface normal to the plane (the up direction) and the distance from the origin of the plane to the world's origin. As with spheres, you can translate, rotate, and scale planes.

Examples:

A plane in the X-Z axes 10 units below the world origin. Like a floor or ceiling.

```
plane { <0.0  1.0  0.0> -10.0 }
```

A plane in the X-Z axes 10 units above the world origin. Like a wall in front of the viewer.

```
plane { <0.0  1.0  0.0>  10.0 }
```

A plane in the X-Y axes 10 units behind the world origin.
Like a wall to the side of the viewer.

```
plane <0.0  0.0  1.0>  -10.0 }
```

Boxes
A simple box can be defined by listing two corners of the box like this:

```
  box { <corner1> <corner2> }
  box { <0 0 0> <1 1 1> }
```

Corner1 should be smaller than corner2. That is, each element of Corner1 should be less than the corresponding element in Corner2. If any elements of Corner1 are larger than Corner2, the box will not appear in the scene.

Corner1 and corner2 are XYZ points. Boxes are calculated efficiently and make good bounding shapes. As with all shapes, they can be translated, rotated and scaled.

Quadric Shapes
Quadric Surfaces can produce shapes like ellipsoids (spheres), cones, cylinders, paraboloids (dish shapes), and hyperboloids (saddle or hourglass shapes).

The easiest way to use these shapes is to include the standard file "SHAPES.INC" into your program. It contains several pre-defined quadrics and you can transform these pre-defined shapes (using translate, rotate, and scale) into the ones you want.

To be complete, here are the mathematical principles behind quadric surfaces. Those who are not interested in the mathematical details can skip to the next section.
The quadric:

```
      quadric {
          <A B C>
          <D E F>
          <G H I>
          J
      }
```

defines a surface in three dimensions which satisfies the following equation:

```
       A y^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. So, 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        Cylinder along the Z axis

     X**2 + Y**2 - Z**2 = 0     Cone along the Z axis
```

Quadric surfaces the easy way
The file "shapes.inc" uses the declare command to pre-define many quadric surfaces. You can invoke them by using the syntax,

```
  quadric { Quadric_Name }
```

The pre-defined quadrics are centered about the origin <0 0 0> and have a radius of 1. Don't confuse radius with width. The radius is half the diameter or width making the standard quadrics 2 units wide.

Some of the pre-defined quadrics are,

```
 Ellipsoid
 Cylinder_X, Cylinder_Y, Cylinder_Z
 QCone_X, QCone_Y, QCone_Z
 Paraboloid_X, Paraboloid_Y, Paraboloid_Z
```

For a complete list, see the file SHAPES.INC.

To use the pre-defined quadrics, you describe shapes simply as follows:

```
#include "colors.inc"
#include "shapes.inc" // The shape definitions are
                      // in this file
#include "textures.inc"

quadric {
  Cylinder_X  // This is the predefined quadric name
  scale     <50 50 50>
  rotate    <0 45 0>
  rotate    <30 0 0>
  translate <2 5 6>
```

```
  }
```

You may have as many transformation commands (scale, rotate, and translate) as you like in any order. Usually, however, it's best to do a scale first, one or more rotations, then finally a translation. Otherwise, the results may not be what you expect.

The transformations always transform the object about the origin. If you have a sphere at the origin and you translate it then rotate it, the rotation will spin the sphere around the origin like planets about the sun.

Triangles

The triangle shape 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.

There are two different types of triangles: flat shaded triangles and smooth shaded (Phong) triangles.

Flat shaded triangles are defined by listing the three vertices of the triangle. For example:

```
        triangle { <0 20 0> <20 0 0> <-20 0 0> }
```

The smooth shaded triangles use a formula called Phong normal interpolation to calculate the surface normal for the triangle. This makes the triangle appear to be a smooth curved surface. In order to define a smooth triangle, however, you must supply not only the vertices,but also the surface normals at those vertices. For example:

```
     smooth_triangle {
        <  0  30 0 >    <0  .7071 -.7071>
        < 40 -20 0 >    <0 -.8664 -.5   >
        <-50 -30 0 >    <0 -.5    -.8664>
     }
```

As with the other shapes, triangles can be translated, rotated, and scaled.

NOTE 1: Triangles cannot be used in CSG intersection or difference types (described next) since triangles have no clear "inside". The CSG union type works acceptably, but with no real difference from a composite made up of TRIANGLE primitives.

Quartic Surfaces
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. The most common one is the torus or donut. The syntax for using a pre-defined quartic is:

```
quartic { Quartic_Name (sturm) }
```

They may be scaled, rotated, and translated like all other shape types.

Quartics use highly complex computations and will not always render perfectly. If the quartic 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. See the archive MATH for examples of quartics in scenes.

Here's a more mathematical description of quartics for those who are interested.
Quartic surfaces are 4th order surfaces, and can be used to describe a large class of shapes including the torus, the lemniscate,
etc. The general equation for a quartic equation in three variables is (hold onto your hat):

```
a00 x^4 + a01 x^3 y + a02 x^3 z+ a03 x^3 + a04 x^2 y^2+

a05 x^2 y z+ a06 x^2 y + a07 x^2 z^2+a08 x^2 z+a09 x^2+
a10 x y^3+a11 x y^2 z+ a12 x y^2+a13 x y z^2+a14 x y z+
a15 x y + a16 x z^3 + a17 x z^2 + a18 x z + a19 x+
a20 y^4 + a21 y^3 z + a22 y^3+ a23 y^2 z^2 +a24 y^2 z+
a25 y^2 + a26 y z^3 + a27 y z^2 + a28 y z + a29 y+
a30 z^4 + a31 z^3 + a32 z^2 + a33 z + a34
```

To declare a quartic surface requires that each of the coefficients (a0 -> a34) be placed in order into a single long vector of 35 terms.

As an example, the following object declaration is for a torus having major radius 6.3 minor radius 3.5 (Making the maximum width just under 10).

```
//Torus having major radius sqrt(40), minor radius sqrt(12)
```

```
object {
   quartic {
      < 1.0   0.0   0.0   0.0   2.0   0.0   0.0   2.0   0.0

     -104.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0   0.0

        0.0   0.0   1.0   0.0   0.0   2.0   0.0  56.0   0.0

        0.0   0.0   0.0   1.0   0.0 -104.0  0.0 784.0 >
      }
      bounded_by { // bounded_by speeds up the render,
                   // see bounded_by
                   // explanation later
                   // in docs for more info.
       sphere { <0 0 0> 10 }
      }
     }
```

The code to calculate Ray-Surface intersections for quartics is somewhat slower than that for intersections with quadric surfaces. Benchmarks on a stock 8Mhz IBM-PC AT (with FPU) give results of around 1400 solutions per second for 2nd order equations (quadrics) vs 444 per second for 3rd order equations and 123 per second for 4th order equations (quartics). So clever use of bounding shapes can make a big difference in processing time.

While a great deal of time has gone into debugging the code for handling quartic surfaces, we know there are possible bad combinations of surface equations and lighting orientations that could cause math errors. If this happens to you, as the joke goes, "then don't DO that."
Here are some quartic surfaces to get you started.

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 (r0^2 + r1^2) x^2 + 2 (r0^2 - r1^2) y^2
-2 (r0^2 - r1^2) z^2 + (r0^2 - r1^2)^2 = 0
```

Where r0 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 r1 is the "minor" radius of the torus - the distance from the middle of the ring of the donut to the outer surface. Described another way, a torus is a circle that is revolved around an axis. The major radius is the distance from the axis to the center of the circle, the

minor radius is the radius of the circle.

Note that scaling surfaces like a torus scales everything uniformly. In order to change the relationship between the size of the hole and the size of the ring, it is necessary to enter new coefficients for the torus.

The only coefficients of the 35 that need to be filled in are:

```
a0  = a20 = a30 = 1,
a4  = a7  = a23 = 2,
a9  = a32       = -2 (r0^2 + r1^2)
a25             = 2 (r0^2 - r1^2)
a34             = 2 (r0^2 - r1^2)^2
```

the torus can then be rotated and translated into position once its shape has been defined. (See 'TORUS.POV')
A generalization of the lemniscate of Gerono can be represented by the equation:

```
c^2 x^4 - a^2 c^2 x^2 + y^2 + z^2 = 0
```

where good start values are a = 1.0 and c = 1.0, giving the coefficients:

```
a0 = a25 = a32 = 1,
a9 = -1
```

(See the example file "LEMNISCA.POV" for a more complete example)

A generalization of a piriform can be represented by the equation:

```
-b c^2 x^4 - a c^2 x^3 + y^2 + z^2 = 0
```

where good start values are a = 4, b = -4, c = 1, giving the coefficients

```
a0 = 4,
a3 = -4,
a25 = a32 = 1
```

(See the file "PIRIFORM.POV" for more on this...)

There are really so many different quartic 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 quartic shape formulas is:

"The CRC Handbook of Mathematical Curves and Surfaces"
David von Seggern
CRC Press
1990

Blob
Syntax:

```
blob {
   threshold #
   component (strength_val) (radius_val) <component center>

   component (strength_val) (radius_val) <component center>

   [any number of components]
   [sturm]
}
```

Blobs are an interesting shape type. Their components are "flexible" spheres that attract or repel each other creating a "blobby" organic looking shape. The spheres' surfaces actually stretch out smoothly and connect, as if coated in silly putty (honey? glop?) and pulled apart.

Blobs can be used in CSG shapes and they can be scaled, rotated and translated. They may contain any number of components. Because the calculations for blobs need to be highly accurate, sometimes they will not trace correctly. Try using the "sturm" keyword for slower, more accurate calculations if this happens.

Component strength may be positive or negative. A positive value will make that component attract other components. Negative strength will make that component repel other components. Components in different, separate blob shapes do not affect each other.

Blob example:

```
 blob {
      threshold 0.6
      component 1.0 1.0 <.75 0 0>
      component 1.0 1.0 <-.375  .64952 0>
      component 1.0 1.0 <-.375 -.64952 0>
      scale <2 2 2>
      sturm
```

```
}
```

How do blobs work?
Picture each blob component as a point floating in space, each point has a field around it that starts very strong at the point and drops off to zero at some radius. POV-Ray looks for the places that the strength of the field is exactly the same as the "threshold" value that was specified.

If you have a single blob component then the surface you see will look just like a sphere, with the radius of the surface being somewhere inside the "radius" value you specified for the component. The exact radius of this sphere-like surface 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 modelling).

If you have a number of blob components, then their fields add together at every point in space - this means that if the blob components are close together the resulting surface will smoothly flow around the components.

The various numbers that you specify in the blob declaration interact in
several ways. The meaning of each can be roughly stated as:

Threshold:
This is the total density value that POV-Ray is looking for. 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 density is equal to the "threshold" value.

1) "threshold" must be greater than 0. POV-Ray only looks for positive densities.
2) If "threshold" is greater than the strength of a component, then the component will disappear.
3) As "threshold" gets larger the surface you see gets closer to the centers of the components.
4) As "threshold" gets smaller, the surface you see gets closer to the spheres at a distance of "radius" from the centers of the components.

Strength:

Each component has a strength value - this defines the density of the component at the center of the component. Changing this value will usually have only a subtle effect.

1) "strength" may be positive or negative. Zero is a bad value, as the net result is that no density was added - you might just as well have not used this component.

2) If "strength" is positive, then POV-Ray will add its density to the space around the center of the component. If this adds enough density to be greater than "threshold", you will see a surface.

3) If "strength" is negative, then POV-Ray will subtract its density 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.

Radius:

Each component has a radius of influence. The component can only affect space within "radius" of its center. This means that if all of the components are farther than "radius" from each other, you will only see a bunch of spheres. If a component is within the radius of another component, then the two components start to affect each other. At first there is only a small bulge outwards on each of the two components, as they get closer they bulge more and more until they attach along a smooth neck. If the components are very close (i.e. their centers are on top of each other), then you will only see a sphere (this is just like having a component of more strength. bigger than the size of each of the component radii)

1) "radius" must be bigger than 0.
2) As "radius" increases the apparent size of the component will increase.

Center:

This is simply a point in space. It defines the center of a blob component. By changing the x/y/z values of the center you move the component around.

The Formula

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 formula used for a single blob component is:

```
density = strength * (1 - radius^2)^2
```

This formula has the nice property that it is exactly equal to strength" at the center of the component and drops off to exactly 0 at a distance of "radius" from the center of the component. The density formula for more than one blob component is just the sum of the individual component densities:

```
density = density1 + density2 + ...
```

Height Fields

Syntax:

```
height_field { image_type "filename" [water_level #] }
```

Examples:

```
height_field { gif "povmap.gif" water_level .1 }
height_field { tga "sine.tga" }
height_field { pot "fract003.pot" water_level .5 }
```

Height fields are a very exciting feature that you may find a little hard to understand at first. Read the explanation below, and look in the example scene files for height field examples. Once you begin to use them, they are much easier to understand. Among other things, they allow you to create fractal landscapes that rival the best computer graphics you've ever seen.

A height field is essentially a 1 unit wide by 1 unit long box made up of many small triangles. The height of each triangle making up the box is taken from the color number (or palette index) of the pixels in a graphic image file.

The mesh of triangles corresponds directly to the pixels in the image file. In fact, there are two small triangles for every pixel in the image file. The Y (height) component of the triangles is determined by the palette index number stored at each location in the image file. The higher the number, the higher the triangle. The maximum height of an

un-scaled height field is 1 unit.

The higher the resolution of the image file used to create the height field, the smoother the height field will look. A 640 X 480 GIF will create a smoother height field than a 320 x 200 GIF.

The optional "water_level" parameter tells the program not to look for the height field below that value. Default value is 0, and legal values are between 0 and 1. For example, "water_level .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. 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 .001"

The image file used to create a height field can be a GIF, TGA or POT format file. The GIF format is the only one that can be created using a standard paint program.

The size/resolution of the image does not affect the size of the height field. The un-scaled height field size will always be 1x1. Higher resolution image files will create smaller triangles, not larger height fields.

In a GIF file, the color number is the palette index at a given point. Use a paint program to look at the palette of a GIF image. The first color is palette index zero, the second is index 1, the third is index 2, and so on. The last palette entry is index 255. Portions of the image that use low palette entries will be lower on the height field. Portions of the image that use higher palette entries will be higher on the height field. For example, an image that was completely made up of entry 0 would be a flat 1x1 square. An image that was completely made up of entry 255 would be a 1x1x1 cube.

The maximum number of colors in a GIF are 256, so a GIF height field can have any number of triangles, but they will only 256 different height values.

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". If you have access to an IBM-PC, you can 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 greater than 32,000. This means a POT height field can have over 32,000 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 IBM-PC program called Fractint. POT files generated with this fractal program create fantastic landscapes. If you have access to an IBM-PC, you can get Fractint from most of the same sources as POV-Ray.

The TGA file format is used as a storage device for numbers rather than an image file. That is, you can't use a paint program to create TGA height field files, and a scanned TGA image will not make a nice smooth height field.

The TGA format uses the red and green bytes of each pixel to store the high and low bytes of a height value. TGA files are as smooth as POT files, but they must be generated with special custom-made programs. Currently, this format is for programmers exclusively, though you should see TGA height field generator programs arriving soon.
There is example C source code included with the POV-Ray source archive to create a TGA file for use with a height field.

Height fields may be rotated translated and scaled like any

other shape. Height fields cannot be used as true CSG shapes, though they are allowed in CSG shapes. They can be included in a CSG shape, but they will not be treated like solid objects.

Here are a few notes on height fields from their creator, Doug Muir:

The height field is mapped to the x-z plane, with its lower left corner sitting at the origin. It extends to 1 in the positive x direction and to 1 in the positive z direction. It is maximum 1 unit high in the y direction. You can translate it, scale it, and rotate it to your heart's content.

When deciding on what water_level to use, remember, this applies to the un-transformed height field. If you are a Fractint user, the water_level should be used just like the water_level parameter for 3d projections in Fractint.

Here's a detailed explanation of how the ray-tracer creates the height field. You can skip this if you aren't interested in the technical side of ray-tracing. This information is not needed to create or use height fields.

To find an intersection with the height field, the raytracer first checks to see if the ray intersects the box which surrounds the height field. Before any transformations, this box's two opposite vertexes are at (0, water_level, 0) and (1, 1, 1). If the box is intersected, the raytracer figures out where, and then follows the line from where the ray enters the box to where it leaves the box, checking each pixel it crosses for an intersection.

It checks the pixel by dividing it up into two triangles. The height vertex of the triangle is determined by the color index at the corresponding position in the GIF, POT, or TGA file.

If your file has a uses the color map randomly, your height field is going to look pretty chaotic, with tall, thin spikes shooting up all over the place. Not every GIF will make a good height field.

If you want to get an idea of what your height field will look like, I recommend using the IBM-PC program Fractint's

3d projection features to do a sort of preview. If it doesn't look good there, the raytracer isn't going to fix it. For those of you who can't use Fractint, convert the image palette to a gray scale from black at entry 0 to white at entry 255 with smooth steps of gray in-between. The dark parts will lower than the brighter parts, so you can get a feel for how the image will look as a height field.

Bezier/Bicubic Patches
Syntax:

```
bicubic_patch { patch_type_# [flatness_#] u_ steps v_steps
      < Control point1>  < CP2>  <CP3>  < CP4>
      <CP5>  <CP6>   <CP7>   <CP8>
      <CP9>  <CP10>  <CP11>  <CP12>
      <CP13> <CP14>  <CP15>  <CP16>
}
```

Like triangles, the bicubic patch is not meant to be generated by hand. These shapes should be created by a special utility. You should be able to acquire utilities to generate these shapes from the same source that you received POV-Ray from.

A bezier or bicubic path is a 3D curved surface created from a mesh of triangles. The number of triangles is u_steps * v_steps. Since it's made from triangles it cannot be used as a solid object in a CSG shape. It may be included in a CSG shape, though. It can be scaled, rotated, translated like any other shape. The 16 control points determine the shape of the curve by "pulling" the curve in the direction of the points.

Example:

```
// Patch type 1 with 8 u steps and 8 v steps
  bicubic_patch { 1 8 8
     < 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>
}
```

Constructive Solid Geometry (CSG)

This ray tracer supports Constructive Solid Geometry (also called Boolean operations) in order to make the shape definition abilities more powerful.

The simple shapes used so far are nice, but not terribly useful on their own for making realistic scenes. It's hard to make interesting objects when you're limited to spheres,

boxes, infinite cylinders, infinite planes, and so forth.

Constructive Solid Geometry (CSG) is a technique for taking these simple building blocks and combining them together. You can use a cylinder to bore a hole through a sphere. You can use planes to cap cylinders and turn them into flat circular disks that are no longer infinite.

Constructive Solid Geometry allows you to define shapes which are the union, intersection, or difference of other shapes.

Unions superimpose two or more shapes. This has the same effect as defining two or more separate objects, but is simpler to create and/or manipulate.

Intersections define the space where the two or more surfaces meet.

Differences allow you to cut one object out of another.

CSG intersections, unions, and differences can consist of two or more shapes. They are defined as follows:

```
object {
   csg_shape_type {
      shape {...}
      shape {...}
      shape {...}
      (...)
   }
   texture { ... }
}
```

CSG shapes may be used in CSG shapes. In fact, CSG shapes may be used anyplace that a standard shape is used.

The order of the shapes doesn't matter except for the difference shapes. For CSG differences, the first shape is visible and the remaining shapes are cut out of the first.

Constructive solid geometry shapes may be translated, rotated, or scaled in the same way as any shape. The shapes making up the CSG shape may be individually translated, rotated, and scaled as well.

When using CSG, it is often useful to invert a shape so that

it's inside-out. The appearance of the shape is not changed, just the way that POV-Ray perceives it. The inverse keyword can be used to do this for any shape. When inverse is used, the "inside" of the shape is flipped to become the "outside". For planes, "inside" is defined to be "in the opposite direction to the "normal" or "up" direction.

Note that performing an intersection between a shape and some other inverse shapes is the same as performing a difference. In fact, the difference is actually implemented in this way in the code.

Inside and outside
Most shape primitives, like spheres, boxes, and blobs, divide the world into two regions. One region is inside the surface and one is outside. (The exceptions to this rule are triangles, height fields and bezier patches - we'll talk about this later.)

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 numerical inaccuracies will put it to one side or the other).

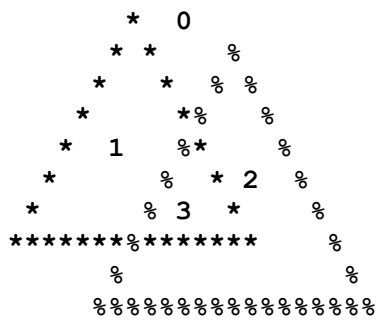
Even planes have an inside and an outside. By definition, the surface normal of the plane points towards the outside of the plane. (For a simple floor, for example, the space above the floor is "outside" and the space below the floor is "inside". For simple floors this in un-important, but for planes as parts of CSG's it becomes much more important).

CSG uses the concepts of inside and outside to combine shapes together. Take the following situation:

Note: The diagrams shown here demonstrate the concepts in 2D and are intended only as an analogy to the 3D case.

Note that the triangles and triangle-based shapes cannot be used as solid objects in CSG since they have no clear inside and outside.

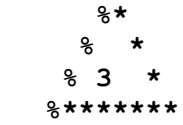
* = Shape A
% = Shape B



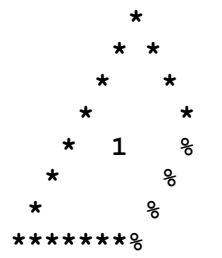
There are three CSG operations you can use, union, intersection, and difference:

union { A B }
A point is inside the union if it is inside A OR it's inside B (or both). This gives an "additive" effect to the component objects:

intersection { A B }
A point is inside the intersection if it's inside both A AND B. This "logical AND's" the shapes and gets the common part, most useful for "clipping" infinite shapes off, etc:



difference { A B }
A point is inside the difference if it's inside A but not inside B. The results is a "subtraction" of the 2nd shape from the first shape:



Let's give a concrete example by drilling a yellow hole

through our sphere. Go to the definition of the sphere in PICTURE1.POV and change it to read the following:

```
object {
  difference {
    sphere { <0 1 3> 1 }
    quadric {
      Cylinder_Z
      scale <.2 .2 1>
      translate <0 1 0>
      texture { color Yellow }
    }
  }
     texture {
      DMFWood1
      scale <2 2 1>
      phong 1
    }
}
```

Inverse

You can flip a shape "inside-out" by putting the keyword inverse into the shape's definition. This keyword will not change the appearance of the shape unless you're using CSG. In the case of CSG, it gives you more flexibility. For example:

```
   intersection { B {A inverse}  }
```

Note: A difference is really just an intersection of one shape with the inverse of another. This happens to be how difference is actually implemented in the code.

Composite Objects

Often it's useful to combine several objects together to act as a whole. A car, for example, consists of wheels, doors, a roof, etc. A composite object allows you to combine all of these pieces into one object. This has two advantages. It makes it easier to move the object as a whole and it allows you to speed up the ray tracing by defining bounding shapes that contain the objects. (Rays are first tested to see if they intersect the bounding shape. If not, the entire composite object is ignored). Composite objects are defined as follows:

```
composite {
  object {...}
  object {...}
  object {...}
  (...)
  [transformations]
}
```

Composite objects can contain other composite objects as well as regular objects.

NOTE: Each object in a composite object may have its own texture, but the composite object may not have a texture.

NOTE: Composite objects may only be used in other composite objects, not in normal objects or shapes.

NOTE: Composite objects can not be used in CSG shapes. CSG shapes must be in an object to be used in a composite.

For example,

```
composite { // This is legal
   object { union  {...} }
   object { sphere {...} }
   object { intersection  {...} }
}

composite { // This is legal
   object { union  {...} texture {...} }
   object { sphere {...} texture {...} }
   object { intersection  {...} texture {...} }
}
```

```
composite { // This is NOT legal
   object { union  {...}}
   object { sphere {...}}
   object { intersection  {...}}
   texture {...} // Texture is NOT legal in composite
}

composite { // This is NOT legal
    union {...} // Shapes not in objects are
                // not legal in composites
    sphere {...}
}

object { // This is NOT legal
   composite { ... } // Composites are not
                     // allowed in objects
}

object { // This is NOT legal
   union {
     composite { ... } // Composites are not
                       // allowed in CSG shape types
   }
}


Example:
#declare Abstract =
composite {
   object { sphere { <0 0 0> 1 } texture {color Red} }
   object { sphere { <0 1 0> 1 } texture {color Yellow} }
   composite { // This is a composite in a composite
       object {
          box { <2 2 2> <3 3 3> }
          texture {color Green}
       }
       object {
          box { <-3 -3 -3> <-2 -2 -2> }
          texture {color Blue}
       }
   }
   object {
      quadric { Ellipsoid }
      texture { PinkAlabaster }
      scale <3 1 3>
      translate <0 4 0>
    }
```

```
}
composite { Abstract translate <0 5 10> }
```

Bounding Shapes

NOTE: Efficient use of bounding shapes are the best way to speed up the rendering of your scenes.

If you use bounding shapes around any complex objects you can speed up the rendering. 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, then the ray is further tested against the more complicated object inside. Otherwise the entire complex shape is skipped, which greatly speeds rendering.

To use bounding shapes, simply include the following lines in the declaration of your object or composite_ object:

```
     bounded_by {
          shape { ... }
     }
```

An example of a Bounding Shape:

```
      object {
          intersection {
               sphere <0.0 0.0 0.0> 2.0 }
               plane <0.0 1.0 0.0> 0.0 }
               plane <1.0 0.0 0.0> 0.0 }
          }
          bounded_by {
               sphere <0.0 0.0 0.0> 2.0 }
          }
     }
```

The best bounding shape is a sphere or a box since these shapes are highly optimized, although, any shape may be used, including CSG shapes.

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 Clipping Shapes below.

Clipping Shapes

A clipping shape is used to "cut off" part of an object.

For example:

```
object {
     sphere {<0 1 3> 1}
     clipped_by {
          sphere { <.5 1 2> 1 }
     }
}
```

You can use any kind of shape including CSG shapes to clip an object.

Textures

Textures are the materials that the shapes and objects in POV-Ray are made out of. They specifically describe the surface coloring, shading, and properties like transparency and reflection.

You can create your own textures using the parameters described below, or you can use the many pre-defined high quality textures that have been provided in the files TEXTURES.INC and STONES.INC. See TEXTURES.DOC for more info on using these predefined materials.

The textures, or materials, in POV-Ray are generally made up of three portions, a color pattern, a bump pattern, and surface properties.

Color Pattern

A color pattern is where two or more colors are used to create a three-dimensional pattern or design. There are many built-in color patterns in POV-Ray ranging from checker, to marble, to wood, to agate.

For example, a checker pattern is just alternating color blocks stacked on and next to each other. The marble color pattern is vertical bands of colors placed next to each other. The wood color pattern is cylindrical bands of color like the rings of a tree, and so on.

These patterns are all generated from mathematical formulas and are very regular unless they are "stirred up" using the turbulence parameter which is described below.

The individual colors in a color pattern can be made partially or completely transparent by specifying the alpha color component. See the color description for more info.

The names of the patterns are only suggestions for their use. Don't let those names limit you, by using different colors and turbulence values, nearly infinite permutations of these patterns are possible.

The colors in a color pattern are usually defined in a "color_map". Color maps are described below.

A shape may also have a solid color instead of a color pattern. This is specified by describing one color instead of a color pattern type. This color may also be made

transparent by specifying the alpha component of the color.

Transformations like scale, rotate, and translate will affect the color pattern and bump pattern in a texture in the same way. It is not currently possible to specify different transformation values for the color and bump patterns in the same texture.

Bump Patterns
A bump pattern makes a surface look "bumpy" without actually changing the form of the shape. The bump pattern modifies the way the surface reflects light so that it is shaded and highlighted just like it were bumpy, but the ray-tracer saves time by doing most calculations as if it were a smooth shape.

A true "bumpy" shape is very complex and time consuming to render. By using bump patterns you can create the appearance of dents, bumps, and ripples without the penalty of rendering a highly complex shape.

Some bump patterns are ripples, dents, bumps, wrinkles and waves. These patterns can produce very realistic looking surface variations. They can be combined with color patterns for more variation. If no bump pattern is specified then the texture is treated as being smooth.

Normally, the bump amount is kept small to enhance realism. Since the bump pattern only changes the shading, the silhouette of a shape is left unchanged. A bumpy sphere will still have a perfectly round edge, even though its surface looks very convincingly bumpy.

The bump pattern never changes the colors on a shape, only the shading. Bump patterns, like color patterns, are three dimensional. Turbulence does not affect bump patterns.

Transformations like scale, rotate, and translate will affect the color pattern and bump pattern in a texture in the same way. It is not currently possible to specify different transformation values for the color and bump patterns in the same texture.

Surface Properties
Surface properties are the miscellaneous parameters which describe whether a shape is reflective, shiny, refractive, and so on.

```
Some examples of surface properties are:
  phong #       -- Make surface shiny by adding highlights
  phong_size #  -- Change the size of the highlight on
                   surface
  reflection #  -- Make surface reflect the scene around it


Texture Syntax
The basic texture syntax is as follows:

 texture {
    (random dither #)
    (color_pattern_type) or (color red # green # blue #)
    (bump_pattern_type (bump_amount))
    [texture modifiers]
    [texture transformations]
 }

For example:

 texture {
    0.05 // Random dither is not recommended
    wood
    turbulence 0.2
    phong 1
    scale < 10.0 10.0 10.0 >
    translate < 1.0 2.0 3.0 >
    rotate < 0.0 10.0 40.0 >
}

Transformations, like translate, rotate, and scale, are
optional. They allow you to transform the texture
independent of the shape's or object's transformations. IE.
The texture may be in a different "location" than the shape,
or be a different size than the shape.

Any parameter that changes the appearance of the surface
must be put into a texture block. Outside of the texture
block, the texture parameters will cause an error message to
be generated.


Textures and Animation
If you are doing animation, you should specify all object
transformations after the texture transformations so the
texture "follows" or "sticks to" the object through 3-D
space. For example,

object {
```

```
   shape {}   // Shape description
   texture {} // Texture description
   scale     // transform after shape & texture
   rotate    // transform after shape & texture
   translate // transform after shape & texture
}
```

As in:

```
object{
   box {<-1 -1 -1> <1 1 1>} // Box shape centered on zero
   texture {
      agate
      phong 1
      scale <1.3 2 1.3> // scale texture to the
                        // untransformed shape
      rotate <30 0 0>   // rotate texture to the
                        // untransformed shape
   }
   scale <3 3 3> // scale *both* sphere & texture
   rotate <0 40 0> // rotate *both* sphere & texture
   translate <1 5 5> // Move box & texture to final
                     // position
}
```

If the transformations are done after the shape and before the texture, the texture will not "stick to" the shape. In an animation, this will look like the texture is constantly moving on the moving shape. What is actually happening is that the shape is moving and the texture is not, so the texture on the shape changes whenever the shape is scaled, rotated, or translated. This can be an interesting effect, but is not desirable for most animations.

Random Dither

The floating-point value given immediately following the texture keyword is an optional "texture randomness" value, which causes a minor random scattering of calculated color values and produces a sort of "dithered" appearance. This is used by artists to minimize banding and to provide a more chaotic, uneven look to a texture. Values are generally kept quite small, like .05. This feature is different from turbulence in that textures using it will look slightly different each time they are rendered.

NOTE: This should not be used when rendering animations. This is the only truly random feature in POV-Ray, and will

produce an annoying flicker of flying pixels on any textures animated with a "randomness" value used.

Layered Textures

It is possible to create layered textures. A Layered texture is one where several textures that are partially transparent 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 textures 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.

More about Textures

The textures available in POV-Ray are 3D solid textures. This means that the texture defines a color for any 3D point in space. Just like a real block of marble or wood, there is color all through the block - you just can't see it until you carve away the wood or marble that's in the way. Similarly, with a 3D solid texture, you don't see all the colors in the texture - you only see the colors that happen to be visible at the surface of the object.

As you've already seen, you can scale, translate, and rotate textures. In fact, you could make an animation in which the objects stay still and the textures translate and rotate through the object. The effect would be like watching a time-lapse film of a cloudy sky - the clouds would not only move, but they would also change shape smoothly.

Turbulence or Noise

Often, textures are perturbed by noise. This "turbulence" distorts the texture so it doesn't look quite so perfect. Try changing the sphere in the above example to have the following texture:

```
texture {
  DMFWood1
  turbulence 0.0
  scale <0.2 0.2 1.0>
  phong 1.0
  }
```

When you compare this with the original image, you'll see that the grain is very regular and the pattern is much less

interesting. Turbulence can be any value, though it usually ranges from 0-1.

Color Maps

Most color patterns use color maps. A color map translates a number between 0.0 and 1.0 into a color. The number typically represents the distance into a vein of color - the further into the vein you get, the more the color changes. Color patterns actually generate patterns of numbers between 0 and 1 for each point in space. When the ray tracer request the color from the color pattern for a specific point, the color pattern calculates what number between 0-1 is at that point and then sends that number to the color map. The color map looks to see if there is a color defined for that number, if there isn't, it figures out an in-between color from the colors that were defined. It returns that number to the color pattern and that's the color that ends up on the object. Since the color map makes these "in-between" colors, you can create color maps with smooth bands of many color shades by defining only a few colors.

Here's a simple color map. Try it out on the sphere defined above by changing the definition to this:

```
object {
  sphere {<0 1 3> 1 }
  texture {
    wood  // This is the color pattern
    scale <.2 .2 1>
    color_map{ // This is where the pattern gets its colors
      [ 0  .3  color Red   color Green]
      [.3  .6  color Green color Blue]
      [.6   1  color Blue  color Red]
      }
    phong 1
    }
}
```

This means that as the texture enters into a vein of wood, it changes color smoothly from red to green, from green to blue, and from blue to red again. As it leaves the vein, the transition occurs in reverse. (Since there is no turbulence on the wood by default, the veins of color show up quite well.)

Alpha

You can get more "bang for your buck" from textures by using alpha properties of color. Every color you define in POV-Ray is a combination of red, green, blue and alpha. The red, green and blue are simple enough. The alpha determines

how transparent that color is. A color with an alpha of 1.0 is totally transparent to that color of light. It filters the light. A color with an alpha of 0.0 is totally opaque to all light.

Here's a neat texture to try:

```
texture {
  turbulence .5
  bozo
  color_map {
    // transparent to transparent
    [ 0 .6  color red 1 green 1 blue 1 alpha 1
```

```
            color red 1 green 1 blue 1 alpha 1]
    // transparent to white
    [.6 .8  color red 1 green 1 blue 1 alpha 1
            color red 1 green 1 blue 1]
    // white to grey
    [.8 1  color red 1 green 1 blue 1
           color red 0.8 green 0.8 blue 0.8]
    }
scale <.4  .08  .4>
}
```

This is David Buck's famous cloud texture. It creates white clouds with grey linings. The texture is transparent in the places where the clouds disappear so you can see through it to the objects that are behind.

Note that light is filtered through alpha colors. Red can't pass through a blue filter no matter how high the alpha value. Green can't get through a completely blue filter and so on. To make a completely clear color that all light can pass through, use color red 1 green 1 blue 1 alpha 1. This is defined in COLORS.INC as Clear.

Layering Textures

You can layer textures one on top of another to create more sophisticated textures. For example, suppose you want a wood-colored cloudy texture. What you do is put the wood texture down first followed by the cloud texture. Wherever the cloud texture is transparent, the wood texture shows through. Change your sphere to the following and you'll see.

```
object {
  sphere <0 1 3> 1 }
```

```
  texture{  //This is the wood texture we used earlier.
    DMFWood1
    scale <.2 .2 1>
    phong 1
    }
  texture {
    turbulence .5
    bozo
    color_map {
      // transparent to transparent
      [ 0 .6  color red 1 green 1 blue 1 alpha 1
              color red 1 green 1 blue 1 alpha 1]
      // transparent to white
      [.6 .8  color red 1 green 1 blue 1 alpha 1
              color red 1 green 1 blue 1]
      // white to grey
      [.8 1  color red 1 green 1 blue 1
             color red 0.8 green 0.8 blue 0.8]
      }
    scale <.4  .08  .4>
  }
}
```

Each successive texture is layered on top of the previous textures. In the places where you can see through the upper texture, you see through to the lower textures.

Texture Surface Properties

Color

A color consists of a red component, a green component, a blue component, and possibly an alpha component. All four components are numbers in the range 0.0 to 1.0.

The syntax for colors is the word "color" followed by any or all of the red, green, blue or alpha components in any order. If a parameter is not specified it is assumed to be 0. For example:

```
color  red 1.0  green 1.0  blue 1.0
color  blue 0.56
color  green 0.45 red 0.3 blue 0.1 alpha 0.3
```

Alpha is a transparency indicator. If an object's color contains some transparency, then you can see through it. If Alpha is 0.0, the object is totally opaque. If it is 1.0, it is totally transparent. Note that light is filtered, so color red 0 green 0 blue 0 alpha 1 is totally black and non-transparent. Color red 1 green 1 blue 1 alpha 1 is totally clear. Color red 1 alpha 1 filters out all but red light.

The Canadian spelling colour may be used in place of color.

ambient value

In real life, 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 default value is very little ambient light (0.1). The value can range from 0.0 to 1.0. Higher ambient values in a texture will make a shape using that texture look flat and

plastic.

diffuse value

The diffuse value specifies how much the texture will react to light directly shining on it. High values of diffuse (.7-.9) make the colors in a texture very bright, lower values (.1-.4) will make the colors in the texture seem more subdued. The default value for diffuse is .6. The value can range from 0.0 to 1.0. Diffuse does not affect the highlights or shadowed portions of the texture.

brilliance value
Objects can be made to appear more metallic by increasing their brilliance. This controls the tightness of the basic diffuse illumination on objects and minorly adjusts the appearance of surface shininess. The default value is 1.0. Higher values from 3.0 to about 10.0 can give objects a somewhat more shiny or metallic appearance. 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 either the specular or phong highlighting.

reflection value
By setting the reflection value to be non-zero, you can give the object a mirrored finish. It will reflect all other elements in the scene.

The value can range from 0.0 to 1.0. By default there is no reflection.

Reflection is one of the strongest features of ray tracing. Try making the sphere in PICTURE1.POV mirrored by adding reflection 1.0 to its texture.

Note: Adding reflection to a texture makes it take much longer to render.

refraction value
Example:

```
  texture {
    color White alpha .9
    refraction 1
    ior 1.5
    phong 1
  }
```

The refraction value attenuates the color of the refracted light through a transparent texture. Lower values of refraction will make the transparent portion less transparent. Higher values will make the transparent portion more transparent. This value is usually 1 and transparency

amounts are controlled with the alpha in a color. Legal

values for refraction are from 0-1. By default there is no refraction. If there is no alpha in a color, then it will not be transparent.

The ior value, described below, is used in conjunction with refraction and alpha to create textures that bend light passing through them. This is used to simulate glass, water, diamonds, etc.

Use "refraction 1" and an ior value with a transparent texture when you want to create a texture that bends light like water and glass do.

NOTE: In layered textures, the refraction and ior keywords MUST be in the last texture, otherwise they will not take effect.

NOTE: If a texture has an alpha component and no value for refraction and ior are supplied, the renderer will simply transmit the ray through the surface with no bending.

ior value
Example:

```
  texture {
    color red .5 green .7 blue 1 alpha .9
    refraction 1
    ior 1.33
    phong 1
  }
```

The ior or Index of Refraction determines how dense a transparent texture is or how far the light will bend as it passes through the texture.

For ior to have an affect, the texture must have some transparent colors that use alpha and the refraction value should be set to 1.

A 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. The file IOR.INC pre-defines several useful values for ior.

phong value
Controls the amount of Phong specular reflection highlighting on the texture. Causes bright shiny spots on the element that are the color of the light source being reflected.

Phong highlighting is simulating the fact that slightly reflective objects, especially metallic ones, have microscopic facets, some of which are facing in the mirror direction. The more that are facing that way, the shinier the object appears, and the tighter the specular highlights become. Phong measures the average of 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 of the object's color to the light source's color at the brightest area (center) of the highlight. There is no phong highlighting given by default.

The size of the highlight spot is defined by the phong_size value described below.

phong_size value
Controls the size of the phong Highlight on the object, sort of an arbitrary "glossiness" factor. Think of it as a "tightness" value. The larger the phong_size, the tighter, or smaller, the highlight. The smaller the phong_size, the looser, or larger, the highlight.

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.

specular value
Very similar to Phong specular highlighting, but a different model is used for determining light ray/object intersection, so a more credible spreading of the highlights occur near the object horizons, supposedly.

Specular's value is typically from 0.0 to 1.0, where 1.0 causes complete saturation of the object's color to the light source's color at the brightest area (center) of the highlight. There is no specular highlighting given by default.

The size of the spot is defined by the value given for roughness, described below.

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.

roughness value
Controls the size of the specular Highlight on the object,

relative to the object's "roughness". 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.

metallic
This keyword indicates that the color of the specular and phong highlights will be the surface color instead of the light_source color. This creates a metallic appearance. See the Metal texture in the TEXTURES.INC file for an example of metallic.

Color Pattern Texture Types

Before the color patterns are listed, here is a description of color maps which are essential to color patterns.

Color Maps

For wood, marble, spotted, agate, granite, gradient and other color patterns, you may specify a set of colors to use for the pattern. This is done by a color map or "color spline".

When the object is being textured, a number between 0.0 and 1.0 is generated which is then used to form the color of the point. A color map specifies the mapping used to change these numbers into colors. The syntax is as follows:

```
color_map {
  [start_value end_value color1 color2]
  [start_value end_value color1 color2]
  [start_value end_value color1 color2]
   ...
  }
```

For example,

```
color_map {
  [ 0 .3 color Red    color Yellow]
  [.3 .6 color Yellow color Blue]
  [.6  1 color Green  color Clear]
  }
```

The numeric value between 0.0 and 1.0 is located in the color map and the final color is calculated by a linear interpolation (a smooth blending) between the two colors in the located range.

The easiest way to see how it works is to try it. With a good choice of colors the bozo color pattern produces some of the most realistic looking cloudscapes you've ever seen indoors! Try a cloud color map such as:

```
texture {
   bozo
   turbulence 1  // A blustery day.
                 // For a calmer one, try 0.2
   color_map {
      [ 0 .5 color red .5 green .5 blue  1  // blue to blue
```

```
color red .5 green .5 blue  1]
```

```
      [.5 .6 color red .5 green .5 blue  1  // blue to
white
             color red  1 green  1 blue  1]
      [.6  1 color red  1 green  1 blue  1 // white to grey}
             color red .5 green .5 blue .5]
    }
}
```

The color map above indicates that for small values of the pattern, use a sky blue color solidly until about halfway turbulent, then fade through to white on a fairly narrow range. As the white clouds get more turbulent and solid towards the center, pull the color map toward grey to give them the appearance of holding water vapor (like typical clouds).

Don't let yourself be limited by the color pattern names, try different color schemes on them and see what you can come up with.

The following color patterns are available. They may be scaled, rotated and translated. They may be applied to any shape or object. Turbulence may be used except where noted.

checker - Color Pattern.
Syntax:
  checker color red # green # blue # color red # green # blue #
Example:
  checker color Red color Yellow

checker pattern gives a checker-board appearance. This option works best on planes. When using the checker texturing, you must specify two colors immediately following the word checker. These colors are the colors of alternate squares in the checker pattern. This is a ray tracing classic and cliche.

bozo - Color Pattern.
Syntax:
  bozo color_map {...}

The bozo color pattern basically takes a noise function and maps it onto the surface of an object. This "noise" is well-defined for every point in space. If two points are close together, they will have noise values that are close

together. If they are far apart, their noise values will be fairly random relative to each other.

spotted - Color Pattern.
Syntax:
  spotted color_map {...}

Spotted pattern is a sort of swirled random spotting of the color of the object. If you've ever seen a metal organ pipe up close you know about what it looks like (a galvanized garbage can is close...) Play with this one, it might render a decent cloudscape during a very stormy day. No extra keywords are required. With small scaling values, looks like masonry or concrete.

marble - Color Pattern.
Syntax:
  marble color_map {...}

Marble uses the color map to create parallel bands of color. Add turbulence to make it look more like marble, jade or other types of stone. By default, marble has no turbulence.

wood - Color Pattern.
Syntax:
  wood color_map {...}

Wood uses the color map to create concentric cylindrical bands of color centered on the Z axis. Add small amounts of turbulence to make it look more like real wood. By default, wood has no turbulence.

agate - Color Pattern.
Syntax:
  agate color_map {...}

This pattern is very beautiful and similar to marble, but uses a different turbulence function. The turbulence keyword has no effect, and as such it is always very turbulent.

gradient - Color Pattern.
Syntax:
  gradient <axis_vector> color_map {...}

This is a specialized pattern that uses approximate local coordinates of an object to control color map gradients. This texture does not have a default color_map, so one must be specified.

It has a special <X Y Z> triple called the axis vector given after the gradient keyword, which specifies any (or all)

axes to perform the gradient action on.

Example: a Y gradient <0.0 1.0 0.0> will give an "altitude color map", along the Y axis.

In the axis vector, values given other than 0.0 are taken as 1.0.

For smooth repeating gradients, you should use a "circular" color map, that is, one in which the first color value (0.0) is the same as the last one (1) so that it "wraps around" and will cause smooth repeating gradient patterns.

Scaling the texture is normally required to achieve the number of repeating shade cycles you want.

Transformation of the texture is useful to prevent a "mirroring" effect from either side of the central 0 axes.

Here is an example of a gradient texture which uses a sharp "circular" color mapped gradient rather than a smooth one, and uses both X and Y gradients to get a diagonally-oriented gradient. It produces a dandy candy cane texture!

```
  texture {
    gradient < 1.0 1.0 0.0 >
    color_map {
      [  0 .25  color red 1 green 0 blue 0
                color red 1 green 0 blue 0]
      [.25 .75  color red 1 green 1 blue 1
                color red 1 green 1 blue 1]
      [.75   1  color red 1 green 0 blue 0
                color red 1 green 0 blue 0]
    }
    scale <30 30 30>
    translate <30 -30 0>
 }
```

You may also specify a turbulence value with the gradient to give a more irregular color gradient. This may help to simulate things like fire or corona effects. By default, gradient has no turbulence.

granite - Color Pattern.
Syntax: granite color_map {...}

This pattern uses a simple 1/f fractal noise function to

give a pretty darn good granite pattern. Typically used with small scaling values (2.0 to 5.0). This pattern is used with creative color maps in STONES.INC to create some gorgeous layered stone textures. By default, granite has no turbulence.

onion - Color Pattern.
Syntax: onion color_map {...}

onion is a pattern of concentric spheres. By default, onion has no turbulence.

leopard - Color Pattern.
Syntax: leopard color_map {...}

leopard creates a pattern from stacked spheres. The color of the spheres and the colors in between them is taken from the color map. Turbulence works very effectively with this texture. By default, leopard has no turbulence.

tiles - A texture pattern
This isn't exactly a color pattern, but this is the best place for it at the moment.
We've had many requests for a checker pattern to allow you to alternate between wood and marble or any other two textures. So, we've added a texture called tiles that takes two textures and tiles them instead of two colors.

In order to support layered textures in the future, the syntax is a bit more verbose than it would be otherwise. The syntax is:

```
texture{
  tiles {
    texture {... put in a texture here ... }
    texture {... add optional layers on top of that one ...}
  tile2
    texture {... this is the second tile texture }
    texture  {... a texture layered on top of the 2nd tile}
  }
}
```

Example:

```
    texture{
       tiles {
          texture { Jade }
       tile2
          texture { Red_Marble }
```

```
        }
    }
```

Note that the textures in tiles only use the surface coloring texture information. Information about ambient, diffuse, reflection, etc. and surface normal information (waves, ripples) are ignored inside the tiles.

You may use layered textures with tiles, but only the top layer will show.

Bump Patterns

Bump patterns or surface normal perturbation patterns change the way a surface reflects light and make it look bumpy. The actual shape of the surface is not changed, but shading, highlights, and reflections will make it look as if it were.

ripples - Bump Pattern
Syntax:
  ripples (ripple size #) [frequency #] [phase #]
Example :
  ripples 0.5

The ripples bump pattern make a surface look like ripples of water. The ripples option requires a value to determine how deep the ripples are:

```
texture {
     wood
     ripples 0.3
     phong 1
     translate < 1.0 2.0 3.0 >
     rotate < 0.0 10.0 40.0 >
     scale < 10.0 10.0 10.0 >
}
```

The ripple size may be any number. Larger values create larger ripples. The frequency and phase parameters change the distance between ripples and the position of the ripple, respectively.

waves - Bump Pattern
Syntax:
  waves (wave size #) [frequency #] [phase #]
Example :
  waves 0.5 frequency 10 phase 0.3

This works in a similar way to ripples except that it makes waves with different frequencies. The effect is to make waves that look more like deep ocean waves.

Both waves and ripples respond to a parameter called phase. The phase option allows you to create animations in which the water seems to move. This is done by making the phase increment slowly between frames. The range from 0.0 to 1.0 gives one complete cycle of a wave.

The waves and ripples textures also respond to a parameter

called frequency. If you increase the frequency of the waves, they get closer together. If you decrease it, they get farther apart.

bumps - Bump Pattern
Syntax:
  bumps (bump size #)
Example :
  bumps .4

Approximately the same turbulence function as spotted, but uses the derived value to perturb the surface normal or, in other words, make the surface look bumpy. This gives the impression of a "bumpy" surface, random and irregular, sort of like an orange. After the bumps keyword, supply a single floating point value for the amount of surface perturbation. Values typically range from 0.0 (No Bumps) to 1.0 or greater(Extremely Bumpy).

dents - Bump Pattern
Syntax:
  dents (dent size #)
Example :
  dents .4

Interesting when used with metallic textures, it gives impressions into the metal surface that look like dents. A single value is supplied after the dents keyword to indicate the amount of denting required. Values range from 0.0 (Showroom New) to 1.0 (Insurance Wreck). Use larger values at your own risk, they will raise your rates, anyway. Scale the pattern to make the pitting more or less frequent.

wrinkles - Bump Pattern
Syntax: wrinkles (wrinkle size #)
Example : wrinkles .7

This is sort of a 3-D bumpy granite. It uses a similar 1/f fractal noise function to perturb the surface normal in 3-D space. With a transparent color pattern, could look like wrinkled cellophane. Requires a single value after the wrinkles keyword to indicate the amount of wrinkling desired. Values from 0.0 (No Wrinkles) to 1.0 (Very Wrinkled) are typical.

Mapping Textures

Mapping textures apply a picture to the surface of a shape or object. Maps are used to color objects, make them look bumpy, and to apply different textures to an object, all based on the color of the pixels in the mapped image.

There are several different types of mapping, and several different methods to apply the map.

Mapping Types:

image_map:
Applies the colors in the image to the surface of the shape.

bump_map:
Makes the surface look bumpy based on the color of the pixels in the mapped image.

material_map:
Changes the texture on the surface based on the color of the pixels in the mapped image.

Mapping Method:

0 - Planar:
Like a slide projector, it casts and image onto the surface from 0,0 to 1,1.

1 - Spherical:
Wraps the image once around a sphere with radius 1.

2 - Cylindrical:
Wraps the image once around a cylinder with radius 1, length 1.

3 - Torus:
Wraps the image around a torus (donut) with inner and outer radius of 1.

The documentation for image_map explains the basic options which are true for all mapping types.

image_map - Color Pattern.
Syntax:
image_map { [map_type #] [<gradient>] image_type "filename" [alpha # #] [once] [interpolate #] }

This is a special color pattern that allows you to import a bitmapped image file in GIF, TGA, IFF, or DUMP format and map that bitmap onto an object.

For example:

```
// Use planar (type 0) mapping to project falwell.gif
```

```
// onto the shape in a repeating pattern.
// Set interpolation to 2 (bilinear) so the mapped GIF will

// be smoothed and not look jaggy.
image_map {
  map_type 0 <1 0 -1> gif "falwell.gif" interpolate 2
}

or

// Use spherical (type 1) mapping to
// wrap earth.tga once onto a unit sphere
// No interpolation is used, so it may look
// jaggy
image_map {
  map_type 1 tga "earth.tga"
}
or
// Use cylindrical (type 2) mapping to wrap
// cambells.gif once onto a unit cylinder.
// Set interpolation to 4 (normalized distance)
// so the mapped GIF will be smoothed and not look jaggy.
//  Norm dist isn't as good as bi-linear, but it's faster.
image_map {
  map_type 2 <1 -1 0> gif "cambells.gif" interpolate 4
}
```

The texture in the first example will be mapped onto the object as a repeating pattern. The once keyword places only one image onto the object instead of an infinitely repeating tiled pattern. When once is used, the color outside the mapped texture is set to transparent. You can use the layered textures to place other textures or colors below the image.

The image map methods sphere, cylinder, and torus, wrap the image once and only once around a unit shape of the same name. The map may be scaled uniformly to apply to larger shapes. The maps may be applied to any shapes, but the results are undefined.

The planar image map method is like a slide projector and will work the same for any shape.

You can specify the alpha values for the color palette/registers of GIF or IFF pictures (at least for the modes that use palettes/colormaps). You can do this by

putting the keyword alpha immediately following the filename followed by the register/color number value and transparency. If the all keyword is used instead of a register/color number, then all colors in that colormap get that alpha value.

Eg.

```
   image_map {
     map_type 0 <1.0 -1.0 0.0> gif "mypic.gif"
     alpha all 0.2 // Make all colors 20% transparent
     ...

      or
    alpha 0   0.5 // Make color 0 50% transparent
    alpha 1   1.0 // Make color 1 100% transparent
    alpha 2   0.3 // Make color 2 30% transparent
    ...
    once
    ...
    }
```

NOTE: Alpha works as a filter, so adding alpha to the color black still leaves you with black no matter how high alpha is. If you want a color to be clear, add alpha 1 to the color white.

By default, the image is mapped onto the X-Y plane in the range (0.0, 0.0) to (1.0, 1.0). If you would like to change this default, you may use an optional gradient <x, y, z> vector after the word image_map. This vector indicates which axes are to be used as the u and v (local surface X-Y) axes. The vector should contain one positive number and one negative number to indicate the "u" and "v" axes, respectively. You may translate, rotate, and scale the texture to map it onto the object's surface as desired. Here is an example:

```
     object {
          plane { < 0 0 1 > -20 }
           // make this texture use the x and z axes for
the
           // image mapping.
          texture {
             image_map { 0 <1.0  0.0  -1.0> gif "image.gif" }
             scale <40.0 40.0 40.0>
          }
      }
```

The gradient vector and scaling will not affect a spherical image map. Nor will the "once" keyword. Spherical image maps auto-scale so that they wrap once around the sphere.

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" (library path) options active. This would facilitate keeping all your image maps (.dis, .gif or .iff) files in a "textures" subdirectory, and giving an "-l" option on the command line to where your library of image maps are. Turbulence will affect image maps.

bump_map - A Bump Pattern
Syntax:
bump_map { [map_type #] [<gradient>] image_type "filename" [bump_size #] [once] [interpolate #] [use_color] [use_index]}

Instead of placing the color of the image on the shape, bump_map perturbs the surface normal based on the color of the image at that point. By default, bump_map uses the actual color of the pixel. This can also be specifically directed by using the use_color keyword.

The height of the bump is based on how bright the pixel is. Black is not bumpy, white is very bumpy. Colors are converted to grey scale internally before calculating height.

If you specify use_index, bump_map uses the color's palette number as the height of the bump at that point. So, color #0 would not be bumpy and color #255 would be very bumpy. The actual color of the pixels doesn't matter when using the index.

The relative bump_size can be scaled using bump_size #. The bump_size number can be any number other than 0. Valid numbers are 2, .5, -33, 1000, etc.
Examples:

```
//  Use planar (type 0) mapping to project falwell.gif
// onto the shape in a repeating pattern.
//  Set interpolation to 2 (bilinear) so the mapped GIF
//  will be smoothed and not look jaggy.
// Black and white pictures make interesting bump_maps.
bump_map {
  map_type 0 <1 0 -1> gif "falwell.gif"
```

```
  interpolate 2 bump_size 5
}

or

// Use spherical (type 1) mapping to wrap
// earth.tga once onto a unit sphere
// No interpolation is used, so it may look jaggy
bump_map {
  map_type 1 tga "earth.tga" bump_size -3
}

or

// Use cylindrical (type 2) mapping to wrap cambells.gif
// once onto a unit cylinder.
//  Use color number of pixel for bump height
//  instead of actual color.
bump_map {
  map_type 2 <1 -1 0> gif "cambells.gif" use_index
}
```

General information on maps can be found above in the section on image_map.

material_map - A texture pattern
Syntax:

```
material_map { map_type #  [<gradient>] image_type
"filename"  [once] [interpolate #]
    texture{... } // First used for color 0
    texture {...} // Second texture used for color 1
    texture {...} // Third texture used for color 2
    texture {...} // Fourth texture used for color 3
                  // Color 4 will use texture 5
                  // Color 5 will use texture 6 and so on.
   }
```

Like the tiles texture pattern, material_map applies other textures to a surface. It maps the image to the surface, and then picks the texture for each point on the surface based on the color index in the mapped image.

Imagine a GIF image of blue squiggles on a black background, we'll call it SQUIG.GIF. Looking at the image's palette you can see that black is color 0 and blue is color 1. You can view the palette using a paint program. So, we make a material map for it like this:

```
material_map {
    map_type 0  gif "squig.gif"
    texture { DMFWood2 } // First texture used
                         // for color 0 Black
    texture { Gold_Metal } // Second texture used
                           // for color 1 Blue
   }
```

This map applied to a sphere would put DMFWood1 everywhere the black would be and Gold_Metal everywhere it finds blue squiggles. The effect is of gold inlaid metal on a wooden sphere! It can be very effective when used correctly.

NOTE: Layered textures don't work properly with material maps. Only the top layer of layered textures will show on them.

General information on maps can be found above in the section on image_map.

Interpolation

Interpolation smooths the jaggies on an image or bump map by using a technique similar to anti-aliasing. Basically, when POV-Ray asks for the color from a bump or image map, it often asks for a point that is not directly on top of one pixel, but sort of between several different colored pixels. Interpolations returns an "in-between" value so that the steps between the pixels in the image or bump map will look smoother.

There are currently two types of interpolation:

```
   Normalized Distance -- interpolate 4
   Bilinear            -- 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 bump or image map looks jaggy, try using interpolation instead of going to a higher resolution image. The results can be very good.

Misc Features

Fog

The ray tracer includes the ability to render fog. To add fog to a scene, place the following declaration outside of any object definitions:

```
fog {
     color White // the fog color
     200.0       // the fog distance
}
```

The fog to color ratio is calculated as:

```
1-exp(- depth/distance)
```

So at depth 0, the color is pure (1.0) with no fog (0.0). At the fog distance, you'll get 63% of the color from the object's color and 37% from the fog color.

Default Texture
When a texture is first created, POV-Ray initializes it with default values for all options. The default values are:

```
color red 0 green 0 blue 0 alpha 0
ambient .1
diffuse .6
phong 0
phong_size 40
specular 0
roughness .05
brilliance 1
metallic FALSE
reflection 0
refraction 0
ior 1
turbulence 0
octaves 6
texture randomness (dither) 0
phase 0
frequency 1
color map NONE
```

These values can be changed with the default texture feature. After the parser reads a default texture, it will initialize new textures with the new default values. The syntax of the default texture is:

```
#default {
  texture {..}
}
```

Any texture values may be used. A Common use for the default texture is to change the ambient and diffuse values through the entire scene. For example:

```
#default { texture { ambient .5 diffuse .9} }
```

Would make any textures created after this line use ambient .5 and diffuse .9. All other values will remain at their old default values. Ambient .5 diffuse .9 won't look good, but it'll brighten up the scene if you need help positioning objects.

Any textures that specifically list a value will override

the default value. Default textures may be used more than once in a scene. The new defaults will affect every texture that is created following the default definition.

Max trace level
Syntax: max_trace_level # (default = 5)

Max_trace_level sets the number of levels that POV-Ray will trace a ray. This is used when a ray is reflected or is passing through a transparent object. When a ray hits a reflective surface, it spawns another ray to see what that point reflects, that's trace level 1. If it hits another reflective surface, then another ray is spawned and it goes to trace level 2. The maximum level by default is 5.

If max trace level is reached before a non-reflecting surface is found, then the color is returned as black. Raise max_trace_level if you see black 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 the Cloud_Sky texture on them. Make ten of them in the union with radius's from 1-10 then render the Scene. The image will show the first few spheres correctly, then black. Raise max_trace_level to fix this problem.

Note: Raising max_trace_level will use more memory and time and it could cause the program to crash with a stack overflow error. 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.

Advanced Lessons

The information in this section is designed for people who are reasonably familiar with the raytracer and want more information on how things work. You probably don't need this level of detail to make interesting scene files, but if you suddenly get confused about something the program did, this section may help you figure it out.

Camera

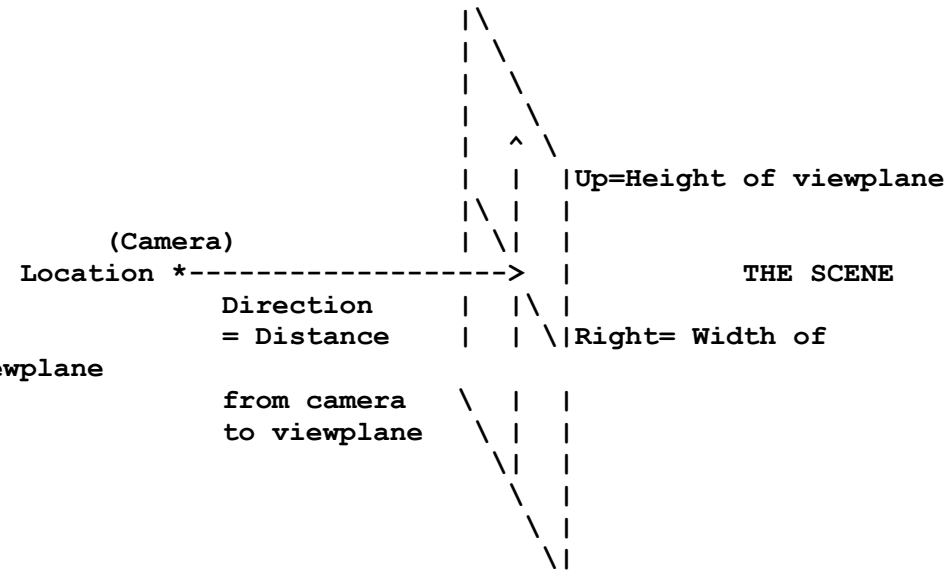
Cameras are internally defined by the four vectors: Location, Direction, Up and Right. The other keywords in a camera block are used to modify some or all of those four values.

The location is simply the X, Y, Z coordinates of the camera. The camera can be located anywhere in the ray-tracing universe.

The direction vector describes the distance from the location of the camera to the viewplane. It also specifies the angle of view. Specifically, it's a vector that starts at the location and points to the center of the viewplane.

The up vector defines the height of the viewplane.
The right vector defines the width of the viewplane.

This figure illustrates the relationship of these vectors:



The viewplane is divided up according to the image resolution specified and rays are fired through the pixels out into the scene.

This camera model is very flexible. It allows you to use left-handed or right-handed coordinate systems. It also

doesn't require that the direction, up, and right vectors be mutually orthogonal. If you want, you can distort the camera to get really bizarre results.

For an eye ray, therefore, the equation of the ray is:

```
  Location + t (Direction + ((height - y)*up) + (x*right))
```

where "t" is a parameter that determines the distance from the eye to the object being tested. The Y coordinate is inverted by subtracting it from height because most graphics systems put 0,0 in the top left corner of the screen.

Once the basic four vectors are specified, it's possible to use the sky and look_at vectors to point the camera. You must specify the sky vector first, but let's describe the look_at vector first. look_at tells the camera to rotate in such a way that the look_at point appears in the center of the screen. To do this, the camera first turns in the left-to-right direction (longitude in Earth coordinates) until it's lined up with the look_at point. It then turns in the up/down direction (latitude in Earth coordinates) until it's looking at the desired point.

Ok, now we're looking at the proper point. What else do we have to specify? If you're looking at a point, you can still turn your camera sideways and still be looking at the same spot. This it the orientation that the sky direction determines. The camera will try to position itself so that the camera's up direction lines up as closely as possible to the sky direction.

Put another way - in airplane terms, the look_at vector determines your heading (north, south, east, or west), and your pitch (climbing or descending). The sky vector determines your banking angle.

Ray-Object Intersections

For every pixel on the screen, the raytracer fires at least one ray through that pixel into the world to see what it hits. For each hit it calculates rays to each of the light sources to see if that point is shadowed from that light source. For reflecting objects, a reflected ray is traced. For refracting objects, a refracting ray is traced. That all adds up to a lot of rays.

Every ray is tested against every object in the world to see

if the ray hits that object. This is what slows down the raytracer. You can make things easier by using simple bounding shapes on your objects.

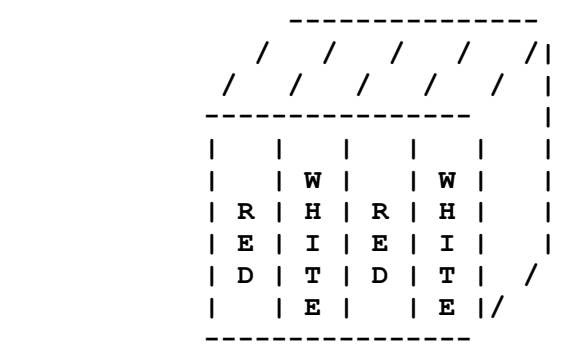
When you use the anti-aliasing option (+a) on your image, the ray tracer will send out multiple rays through each pixel and average the results. This will make the image smoother, but tracing more rays also makes it take longer to render.

Textures, Noise, and Turbulence

Here's how some of the texture features work. If you want some good reading material, check out "An Image Synthesizer" by Ken Perlin in the SIGGRAPH '84 Conference Proceedings.

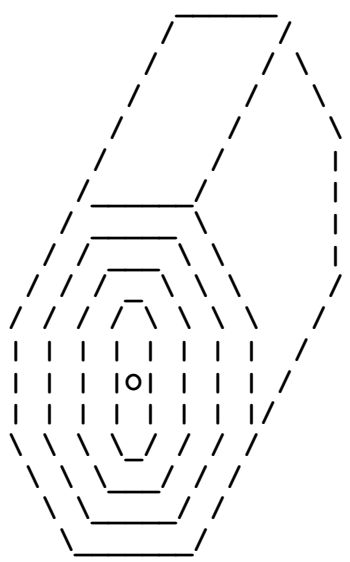
Let's start with a marble texture. Real marble is created when different colors of sediments are laid down one on top of another and compressed to form solid rock.

With no turbulence, a cube textured with marble looks like this:



If you carve a shape put of this block of marble, you will get red and white bands across it.

Now, consider wood. The rings in wood are created when the tree grows a new outer shell every year. Hence, we have concentric cylinders of colors:



Cutting a shape out of a piece of wood will tend to give you rings of color. This is fine, but the textures are still a bit boring. For the next step, we blend the colors together to create a nice smooth transition. This makes the texture look a bit better. The problem, though, is that it's too regular - real marble and wood aren't so perfect. Before we make our wood and marble look any better, let's look at how we make noise. Noise (in raytracing) is sort of like a random number generator, but it has the following properties:

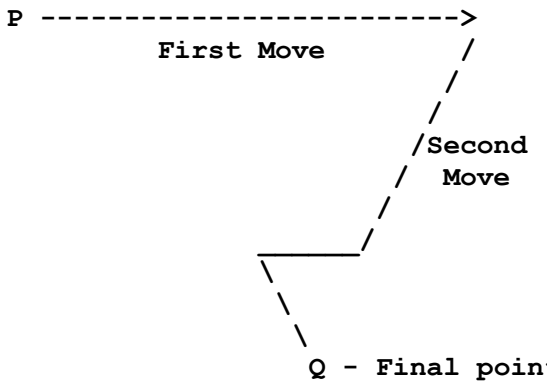
1) It's defined over 3D space i.e., it takes x, y, and z and returns the noise value there.
2) If two points are far apart, the noise values at those points are relatively random.
3) If two points are close together, the noise values at those points are close to each other.

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? bozo texture!

Another function used in texturing is called DNoise. This is sort of like noise except that instead of giving a temperature, it gives a direction. You can think of it as the direction that the wind is blowing at that spot.

Finally, we have a function called turbulence which uses DNoise to push a particle around a few times - each time going half as far as before.



This is what we use to create the "interesting" marble and wood texture. We locate the point we want to color (P), then push it around a bit using Turbulence to get to a final point (Q) then look up the color of point Q in our ordinary boring wood and marble textures. That's the color that's used for the point P.

Octaves
In conjunction with the turbulence function, is the "octaves" value. The octaves value tells the turbulence function exactly how many of these "half-moves" to make. The default value of 6 is fairly close to the upper limit; you won't see much change by setting it to a higher value, but you can achieve some very interesting effects by specifying lower values in the range of 2-5. Setting octaves higher than around 10 can cause a "stack overflow" error (crash) on some machines.

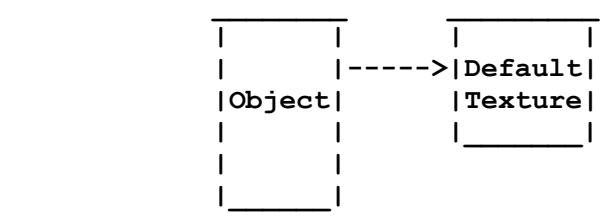
Layered Textures
POV-Ray supports layered textures. They can be used to create very sophisticated materials. Here's how they work.

Each object and each shape has a texture that may be attached to it. By default, shapes have no texture, but objects have a default texture. Internally, textures are marked as being constant or variable. A constant texture is one that was declared as a texture and is being shared by many shapes and objects. Variable textures are textures that have been declared totally within the object or have used a declared texture and modified it in a destructive way. The idea here is that we want to save on memory by sharing textures if possible.

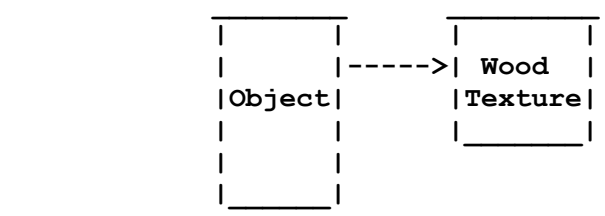
If you have several texture blocks for an object or a shape, they are placed into a linked list (First-in Last-out). For example, take the following definition:

```
 object {
   sphere <0 0 0> 1 }
   texture { Wood }
   texture { Clouds }
}
```

Here's what happens while parsing this object: Since this is an object (as opposed to a shape - sphere, plane, etc.), it starts out with the default texture attached.

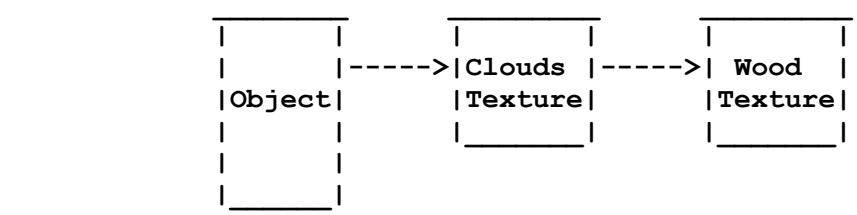


When the parser sees the first texture block, it looks to see what it has linked. Since the thing that's linked is the default texture (not a copy), it discards it and puts in the new texture.



On the next texture, it sees that the texture isn't the

default one, so it adds the second texture into the linked list.



If you want to specify the refraction of the texture, the raytracer must first calculate the surface color. It does this by marching through the texture list and mixing all the colors. When it's finished, it checks the alpha value of the surface color and decides whether it should trace a refracting ray. Where does it get the refraction value and the index of refraction? It simply takes the one in the topmost (the last one defined) texture.

Parallel Image Mapping

One form of image mapping that POV-Ray supports is called a "parallel projection" mapping. This technique is simple, but it's not perfect. It works like a slide projector casting the desired image onto the scene. The difference, however, is that the image never gets larger as you move further away from the slide projector. In fact, there is no real slide projector.

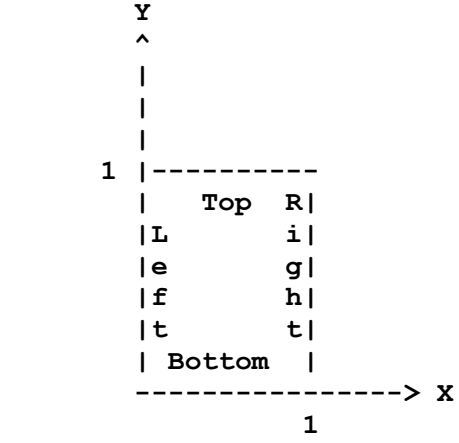
Note that an object cannot shadow itself from the image being mapped. This means that the image will also appear on the back of the object as a mirror image.

The mapping takes the original image (regardless of the size) and maps it onto the range 0,0 to 1,1 in two of the 3D coordinates. Which two coordinates to use is specified by the gradient vector provided after the image. This vector must contain one positive number, one negative number, and one zero. The positive number identifies the u axis (the left right direction in the image) and the negative number represents the v axis (the picture's up-down direction). Note that the magnitude of the number is irrelevant.

For example:

```
    image_map { <1 -1 0> gif "filename" }
```

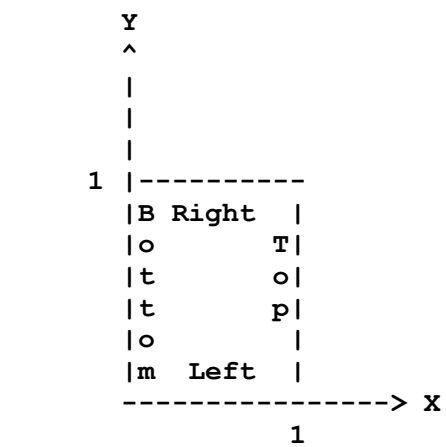
will map the gif picture onto the square from <0 0 0> to <1 1 0> like this:



If we reversed the vector, the picture would be transposed:

```
    image_map { <-1 1 0> gif "filename" }
```

produces:



Once the image orientation has been determined, it can be translated, rotated, and scaled as desired to map properly onto the object.

Common Questions and Answers

Q: I keep running out of memory. What can I do?

A: Buy more memory. But seriously, you can decrease the memory requirements for any given picture in several ways:
1) declare texture constants and use them (textures are shared).
2) Don't modify the texture that you are sharing by scaling, rotating or translating. On the first modify, the texture is copied and (therefore) takes up more space.
3) Put the object transformations before the texture structure. This prevents the texture from being transformed (and hence, copied. This may not always be desirable, though).
4) Use union instead of composite objects to put pieces together.
5) Use fewer or smaller image maps.
6) Use GIF or IFF (non-HAM) images for image maps. These are stored internally as 8 bits per pixel with a color table instead of 24 bits per pixel.

Q: I get a floating point error on certain pictures. What's wrong?

A: The raytracer performs many thousands of floating point operations when tracing a scene. If checks were added to each one for overflow or underflow, the program would be much slower. If you get this problem, first look through your scene file to make sure you're not doing something like:

- Scaling something by 0 in any dimension.
  Ex: scale <34 2 0> will generate an error.
- Making the look_at point the same as the location in the camera
- Looking straight down at the look_at point
- Defining triangles with two points the same (or nearly the same)
- Using the zero vector for normals.
- Using a roughness value of zero (0).

If it doesn't seem to be one of these problems, please let us know. If you do have such troubles, you can try to isolate the problem in the input scene file by commenting out objects or groups of objects until you narrow it down to a particular section that fails. Then try commenting out the individual characteristics of the offending object.

Q: No matter how much I scale a Cylinder, I can't make it fit on the screen. How big is it and how much do I have to scale it?

A: Cylinders (like most quadrics) are infinitely long. No matter how much you scale them, they still won't fit on the screen. To make a disk out of a cylinder, you must use CSG:

```
intersection {
   quadric { Cylinder_Y }
   plane {<0.0 1.0 0.0> 1.0 }
   plane {<0.0 -1.0 0.0> 1.0 }
}
```

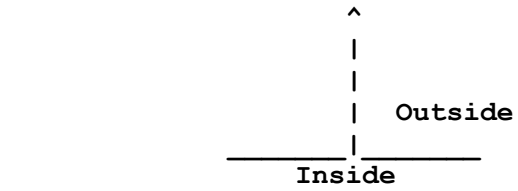
This object is defined in SHAPES.INC as a Disk. Try using a Disk instead. There are three disk types, Disk_X, Disk_Y, and Disk_Z.

Cylinders CAN be scaled in cross-section, the two vectors not in the name of the cylinder (X and Z, in our example above) can be scaled to control the width and thickness of the cylinder. Scaling the Y value (which is normally infinite) is meaningless, unless you have "capped" it as

above, then scaling the entire intersection object in the Y dimension will control the height of the cylinder.

Q: Are planes 2D objects or are they 3D but infinitely thin?

A: Neither. Planes are 3D objects that divide the world into two half-spaces. The space in the direction of the surface normal is considered outside and the other space is inside. In other words, planes are 3D objects that are infinitely thick. For the plane, plane { <0 1 0> 0 }, every point with a positive Y value is outside and every point with a negative Y value is inside.



Q: Can POV-Ray render soft shadows?

A: Not directly. You can use a spotlight to get soft edges on the spot light. You can also create multiple copies of the same scene with the light sources in a slightly different location in each copy and average them together with Piclab or DTA15b to get a soft shadow.

Q: I'd like to go through the program and hand-optimize the assembly code in places to make it faster. What should I optimize?

A: Don't bother. With hand optimization, you'd spend a lot of time to get perhaps a 5-10% speed improvement at the cost of total loss of portability. If you use a better ray-surface intersection algorithm, you should be able to get an order of magnitude or more improvement. Check out some books and papers on raytracing for useful techniques. Specifically, check out "Spatial Subdivision" and "Ray Coherence" techniques.

Q: Objects on the edges of the screen seem to be distorted. Why?

A: If the direction vector of the camera is not very long, you may get distortion at the edges of the screen. Try moving the location back and raising the value of the

direction vector.

Q: How do you position planar image maps without a lot of trial and error?

A: By default, images will be mapped onto the range 0,0 to 1,1 in the appropriate plane. You should be able to translate, rotate, and scale the image from there.

Q: What's the difference between alpha and refraction?

A: The difference is a bit subtle. Alpha is a component of a color that determines how much light can pass through that color. Refraction is a property of a surface that determines how much light can come from inside the surface. See the section above on Transparency and Refraction for more details.

Q: How do you calculate the surface normals for smooth triangles?

A: There are two ways of getting another program to calculate them for you. There are now several utilities to help with this.

1) Depending on the type of input to the program, you may be able to calculate the surface normals directly. For example, if you have a program that converts B-Spline or Bezier Spline surfaces into POV-Ray format files, you can calculate the surface normals from the surface equations.

2) If your original data was a polygon or triangle mesh, then it's not quite so simple. You have to first calculate the
surface normals of all the triangles. This is easy to do - you just use the vector cross-product of two sides (make sure you get the vectors in the right order). Then, for every vertex, you average the surface normals of the triangles that meet at that vertex. These are the normals you use for smooth triangles. Look for the utilities SANDPAPER and TXT2POV.

Q: When I render parts of a picture on different systems, the textures don't match when I put them together. Why?

A: The appearance of a texture depends on the particular

random number generator used on your system. POV-Ray seeds the random number generator with a fixed value when it starts, so the textures will be consistent from one run to another or from one frame to another so long as you use the same executables. Once you change executables, you will likely change the random number generator and, hence, the appearance of the texture. There is an example of a standard ANSI random number generator provided in IBM.C, include it in your machine-specific code if you are having consistency problems.

Q: What's the difference between a color declared inside a texture and one that's in a shape or an object and not in a texture?

A: The color in the texture specifies the color to use for qualities 5 and up. The color on the shape and object are used for faster rendering in qualities 4 and lower and for the color of light sources. See the -q option for details on the quality parameter.

Q: I created an object that passes through its bounding volume. At times, I can see the parts of the object that are
outside the bounding volume. Why does this happen?

A: Bounding volumes are not designed to change the shape of the object. They are strictly a realtime improvement feature. The raytracer trusts you when you say that the object is enclosed by a bounding volume. The way it uses bounding volumes is very simple:
If the ray hits the bounding volume (or the ray's origin is inside the bounding volume),when the object is tested against that ray. Otherwise, we ignore the object. If the object extends beyond the bounding volume, anything goes. The results are undefined. It's quite possible that you could see the object outside the bounding volume and it's also possible that it could be invisible. It all depends on the geometry of the scene. If you want this effect use a clipped_by volume instead of bounded_by.

Tips and Hints

To see a quick version of your picture, render it very small. With fewer pixels to calculate the ray tracer can finish more quickly. --w80 -h60 is a good size.

When animating objects with solid textures, the textures

must move with the object, i.e. apply the same rotate or translate functions to the texture as to the object itself. This is now done automatically if the transformations are placed _after_ the texture block. Example:

```
object {
  shape { ... }
  texture { ... }
  scale < 1 2 3 >
}
```

Will scale the shape and texture by the same amount.

```
object {
  shape { ... }
  scale < 1 2 3 >
  texture { ... }
}
```

Will scale the shape, but not the texture.

You can declare constants for most of the data types in the program including floats and vectors. By combining this with include files, you can easily separate the parameters for an animation into a separate file.

You can declare constants for most of the data types in the program including floats and vectors. By combining this with include files, you can easily separate the parameters for an animation into a separate file.

Some examples of declared constants would be:

```
   #declare Y_Rotation = 5.0
   #declare ObjectRotation = <0 Y_Rotation 0>
   #declare MySphere = sphere { <0 0 0> 1.1234 }
```

Other examples can be found scattered throughout the sample scene files.

Wood is designed like a "log", with growth rings aligned along the z axis. Generally these will look best when scaled down by about a tenth (to a unit-sized object). Start out with rather small value for the turbulence, too (around 0.05 is good for starters).

See the file "textures.doc" in the STDINC.ZIP file for more

pointers about advanced texture techniques.

You can compensate for non-square aspect ratios on the monitors by making the right vector equal to (pixel width/pixel height). On an IBM-PC VGA that is (640/480)=1.333. A good value for the Amiga & IBM-PC is about 1.333. If your spheres and circles aren't round, try varying it. Macintoshes use 1.0 for this value.

If you are importing images from other systems, you may find that the shapes are backwards (left-to-right inverted) and no rotation can make them correct. All you have to do is negate the terms in the right vector of the camera to flip the camera left-to-right (use the "right-hand" coordinate system).

By making the direction vector in the camera longer, you can achieve the effect of a tele-photo lens. Shorter direction vectors will give a kind of wide-angle affect, but you may see distortion at the edges of the image.

Suggested Reading

First, a shameless plug for two new books coming out soon that are specifically about POV-Ray:

     The Waite Group's Ray Tracing Creations
     By Drew Wells
     Waite Group Press
     1992
and
     The Waite Group's Image Lab
     By Tim Wegner
     Waite Group Press
     1992

Image Lab by Tim Wegner will be out in the summer of '92 and contains a chapter about POV-Ray. Tim is the co-author of the best selling book, Fractal Creations, also from the Waite Group.

Ray Tracing Creations by Drew Wells will be out later in '92 and is
an entire book about ray tracing with POV-Ray.

This section lists 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 Raytracing"

Andrew S. Glassner (editor)
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
Addison-Wesley
1983

"Computer Graphics: Principles and Practice (2nd Ed.)"

J. D. Foley, A. van Dam, J. F. Hughes
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
The Beginning of Time

Legal Information

For full details see POVLEGAL.DOC, the following information does not supersede the document POVLEGAL.DOC.

The gist of that document is that POV-Ray and its related files are copyrighted software that may not be modified or distributed without the express written permission of the POV-Ray development team. POV-Ray and its related files may not be used in any commercial or non-commercial program without the express written permission of the POV-Ray development team. In particular, the user may not modify and re-distribute the POV-Ray software for any reason. Nor may the user distribute an executable created by the user from the POV-Ray source code.

Scene and image files created by the user with POV-Ray are the user's property for the user to do as they see fit. The user is free to sell, display, distribute, or consume any scene files or image files created solely by the user for use with POV-Ray.

The POV-Ray development team does not offer any warranty or guarantee for the software for any use. The POV-Ray team is not responsible for any loss incurred by the use of the software.

If there is any question about the use or distribution of POV-Ray, please contact Drew Wells for clarification. See contacting the authors for info.

Contacting the Authors

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 of, distribution of, or anything particularly sticky, please contact Drew Wells, the development team leader. Otherwise, spread the mail around. We all love to hear from you!

The best method of contact is e-mail through Compuserve for most of us. America On-Line and Internet can now send mail to Compuserve, also, just use the Internet address and the mail will be sent through to Compuserve where we read our mail daily.

Please do not send large files to us through the e-mail without asking first. We pay for each minute on Compuserve and large files can get expensive. Send a query before you send the file, thanks!

Drew Wells
(Development team leader. Worked on everything.)
CIS: 73767,1244
Internet: 73767.1244@compuserve.com
AOL: Drew Wells
Prodigy: SXNX74A (Not used often)
US Mail:
  905 North Ave 66
  Los Angeles, CA
  90042
Phone: (213) 254-4041

Other authors and contributors in alphabetical order:

David Buck
(Original author of DKBTrace)
(Primary developer, quadrics, docs)
INTERNET:(preferred) dbuck@ccs.carleton.ca
CIS: 70521,1371

Aaron Collins
(Co-author of DKBTrace 2.12)
(Primary developer, IBM-PC display code,phong, docs)
CIS: 70324,3200

Alexander Enzmann
(Primary developer, Blobs, quartics, boxes, spotlights)
CIS: 70323,2461
INTERNET: xander@mitre.com

Dan Farmer
(Primary developer, docs, scene files)
CIS:70703,1632

Douglas Muir
(Bump maps and height fields)
CIS: 76207,662
Internet:dmuir@media-lab.media.mit.edu

Bill Pulver
(Time code and IBM-PC compile)
CIS: 70405,1152

Chris Young
(Primary developer, docs, scene files)
CIS: 76702,1655

Charles Marslette
(IBM-PC display code)
CIS: 75300,1636

Mike Miller
(Artist, scene files, stones.inc)
CIS: 70353,100

Jim Nitchals
(Mac version, scene files)
CIS: 73117,3020

Eduard Schwan
(Mac version, docs)
Internet: JL.MACTECH@AppleLink.Apple.COM

Randy Antler
(IBM-PC display code enhancements)
CIS: 71511,1015

David Harr
(Mac balloon help)
CIS: 72117,1704

Scott Taylor
(Leopard and Onion textures)
CIS: 72401,410

Index of Authors

Bill Allen: 116
Randy Antler: 119
David Buck: 5 74 75 118
Aaron Collins: 5 9 118
A. Van Dam: 116
Alexander Enzmann: 119
Dan Farmer: 119
J. D. Foley: 116
Andrew Glassner: 116
J. F. Hughes: 116
Charles Marslette: 119
Mike Miller: 119
Bill Minus: 9
Douglas Muir: 58 119
Jim Nitchals: 119
Ken Perlin: 103
Clifford Pickover: 116
Bill Pulver: 119
Adam Schiffman: 9
Eduard Schwan: 119
David von Seggern: 52 117
Scott Taylor: 120
Daniel Thalmann: 116
Nadia Magnenat-Thalmann: 116
Steve Upstill: 116
Tim Wegner: 115
Drew Wells: 115 117 118
Chris Young: 119

INDEX

abort: 13 16 18
abstract: 66 67
adaptive: 15
Addison Wesley: 116
advanced: 6 8 24 99 115
aerobics: 44
agate: 6 69 72 82 84
aiming: 38
air: 79
airplane: 101
alfred.ccs.carleton.ca: 9
algorithm: 111
anti-aliasing: 13 15 16 95 102
alpha: 32 69 70 74 75 76 77 78
79 90 91 92 97 107 112
altitude: 85
ambient: 17 39 77 87 97
America On-Line: 9 118
Amiga: 7 14 36 115
analogy: 62
angle: 21 27 33 34 100 101 115
animation: 16 71 72 73 78 113 114
anonymous ftp: 9
ansi: 113
antenna: 35
apocalypse: 6
Apple: 7 9 119
archive: 7 8 10 11 30 49 57
art: 9
artist: 5 11 72 105 116 119
artwork: 8
ascii: 5 10 11
aspect ratio: 20 21 34 35 36 115
assembly: 111
attention: 21
attenuates: 78
attract: 52
author: 4 17 115 117 118
autocad: 34 44
average: 8 80 102 111 112
averaging: 15
axis: 19 21 23 25 34 36 42 43
44 45 46 47 50 84 85 92
108 114
background: 56 94
backwards: 115
ball: 11 24
balloon: 119
banding: 72

bands: **23 69 74 84 103**

benchmarks: 50
bend: 79
bezier: 6 59 61 112
bicubic: 59
bigbox: 31
bilinear: 91 93 95
bitmap: 90
black: 37 57 59 77 92 93 94 95
98
blend: 17 82 104
blobs: 6 30 52 53 54 55 60 61 119
block: 17 22 33 41 69 71 73 100
103 106 114
blood: 6
blowing: 105
blu: 15
blue: 6 14 22 24 27 28 32 37 38
39 41 45 57 66 71 74 75
76 77 79 82 83 85 94 95
97
blustery: 82
board: 9 83
body: 29
book: 52 111 115
boolean: 59
bore: 60
bounces: 77
bounding: 46 50 65 67 102 113
box: 6 17 18 30 31 46 55 58 66
67 72 116
boxes: 46 60 61 119
bozo: 74 76 82 83 105
bracket: 27
brass: 6
bright: 6 23 37 38 59 78 79 80 93
brighten: 97
brightness: 37 38
brilliance: 78 97
brown: 6
brushed: 29
buffer: 13 16
building: 60
built-in: 6 23 48 69
bulge: 54
bump: 69 70 71 88 89 90 93 94
95 119
bumps: 70 89
bumpy: 6 70 88 89 90 93
bunch: 54

cad: 33 34 44
calmer: 82
cambells: 91 94
camera: 10 19 20 21 22 32 33 34
35 36 44 100 101 110 111
115
Canadian: 77
candy cane: 85
cap: 60
capped: 110
car: 29 65
carve: 73 103
catastrophic: 16
ceiling: 45
cellophane: 89
chaos: 116
chaotic: 58 72
checker: 24 69 83 86
checkerboard: 6
cherry wood: 6
Chicago: 9
chrome: 6
circle: 37 50 51 115 117
circular: 38 60 85
clip: 63 67 68 113
clouds: 6 73 75 82 83 84 98 106 107
coated: 52
code: 8 9 50 57 61 64 111 113
117 118 119
coefficients: 49 51
coherence: 111
color: 6 7 14 15 21 22 23 24 25
27 28 32 36 37 38 39 41
43 44 45 55 56 57 58 64
66 69 70 71 72 73 74 75
76 77 78 79 80 81 82 83
84 85 86 89 90 91 92 93
94 95 96 97 98 104 105
107 109 112 113
colormap: 91 92
colors: 5 6 17 20 21 22 24 28 30
32 47 57 69 70 73 74 75
77 78 79 82 83 86 90 91
92 93 103 104 107
colour: 77
COMART: 9
command: 10 11 12 13 14 17 18 19
20 22 25 35 36 41 47 48 93
comment: 25 26 110

Commodore Amiga: 7
compile: 8 119
component: 25 52 53 54 55 63 69 70
77 79 112
composite: 6 30 48 65 66 67 109
Compuserve: 9 118
concatenate: 16
concentric: 84 86 98 104
cone: 6 24 37 38 40 46 47
constant: 106
constants: 109 114
constructive: 6 59 60
contact: 8 17 117 118
contacting: 4 117 118
contrast: 15
convert: 8 12 22 59
converted: 48 93
converts: 112
coordinate: 19 33 44 100 101 115
coordinates: 19 20 21 33 37 47 84 100
101 108
copyrighted: 5 117
corner: 46 58 101
corners: 46
corona: 85
cpu: 7
crash: 16 98 105
cray: 8
crays: 8
csg: 6 31 38 39 40 48 52 58 59
60 61 62 63 64 65 66 67
68 110
cube: 56 103
curve: 59
curved: 48 59
curves: 52 117
cylinder: 6 24 25 40 47 60 64 90 91
94 110 111
cylinders: 24 46 60 104 110
cylindrical: 69 84 90 91 94
dark: 37 59
darkens: 37
debugging: 50
decimal: 27
declaration: 20 30 34 49 53 67 96
declare: 27 28 29 30 31 32 45 47
49 66 109 114
declared: 30 106 113 114
declaring: 20

default: 12 13 14 15 16 17 21 23
34 35 56 74 77 78 79 80
81 84 85 86 92 93 95 97
98 105 106 107 112
defaults: 34 98
define: 6 28 29 30 34 36 47 48 60
74
defined: 6 20 21 22 23 24 27 28 29
35 36 46 47 48 49 51 60
61 65 69 74 75 80 83 100
104 107 110
defines: 34 46 54 73 79 100
defining: 60 65 74 110
degrees: 36 38 42
dense: 79
densities: 53 55
density: 53 54 55
dent: 89
dented: 54
denting: 89
dents: 70 89
depth: 5 33 96
Desqview: 7
developer: 118 119
developers: 8 9
diameter: 47
diamond: 79
diamonds: 79
difference: 15 31 48 50 60 61 63 64
108 112 113
differences: 6 60
diffuse: 17 39 78 87 97
digits: 27
dimension: 45 110 111
dimensional: 5 10 47 69 70
dimensions: 46
directories: 10 17 20 93
directory: 9 10 11 13 17 18 20 93
dish: 24 46
disk: 16 30 40 110
disks: 60
distance: 24 45 50 53 55 74 88 91
95 96 100 101
distort: 101
distorted: 111
distortion: 21 33 111 115
distorts: 73
distribution: 11 117 118
dither: 71 72 97

dithered: 72
DKBTrace: 5 118
dmfwood: 23 24 64 73 76 95
dnoise: 105
doc: 23 69 114 117
docs: 10 12 14 22 50 118 119
document: 5 11 34 117
documentation: 7 8 14 90
donut: 6 25 49 50 90
doors: 65
dos: 7
double: 26
drilling: 63
dropouts: 49
drops: 53 55
duck: 30
dull: 80
dump: 6 13 14 90
earth: 91 94 101
edge: 70
edges: 33 37 38 111 115
elevation: 34
ellipsoid: 6 45 47 66
ellipsoids: 24 46
emulator: 7
engine: 29
equation: 46 47 49 50 51 53 101
equations: 50 112
error: 25 71 81 98 105 109 110 112
errors: 50 81
Europe: 9
example: 6 8 11 18 23 24 25 27 28 29 31 35 36 37 38 40 41 43 44 48 49 51 52 55 56 57 59 61 63 64 65 66 67 68 69 71 73 75 77 78 79 81 82 83 85 86 88 89 90 91 92 97 106 108 110 112 113 114
examples: 11 12 17 38 45 47 49 55 71 93 114
executable: 7 8 9 10 11 12 14 22 117
executables: 7 8 9 16 113
exercise: 44
experiment: 78
exponents: 27
eye: 101
facets: 80

fade: 83
faked: 24
falloff: 37 38 39
falwell: 90 91 93
famous: 44 75
fantastic: 5 57
field: 30 33 53 55 56 57 58 59
fields: 6 53 55 56 57 58 61 119
file: 6 7 10 11 12 13 14 16 17
18 19 20 22 23 24 25 26
27 28 29 30 46 47 49 51
55 56 57 58 79 81 90 110
114 118
filename: 11 13 14 17 26 55 90 92
93 94 108 109
filenames: 20 93
files: 5 6 8 9 10 11 13 15 16 17
18 20 26 27 30 48 55 56
57 69 93 99 112 114 117
118 119
film: 73
filter: 75 92
filtered: 75 77
filters: 74 77
fingers: 44
flat: 36 45 48 56 60 77
flatness: 59
flattened: 45
flicker: 16 73
flip: 64 115
flipped: 61
float: 27
floating: 25 27 53 72 89 109 110
floats: 27 114
floor: 11 24 45 61
floors: 61
flying: 73
focus: 33
fog: 96
folder: 9
fpu: 50
fract: 55
fractal: 6 9 55 56 57 85 89 115
fractals: 9
fractint: 9 57 58 59
frame: 16 113
frames: 88
freeware: 5 57
frequencies: 88

frequency: 88 89 97
front: 21 23 26 27 42 45
fundamentals: 116
future: 86
galvanized: 84
garbage: 84
geometry: 6 59 60 113
Gerono: 51
gif: 22 55 56 57 58 90 91 92
93 94 95 108 109
glass: 6 79
glop: 52
glossiness: 80
glows: 77
gold: 6 95
gradient: 82 84 85 90 92 93 94 108
gradients: 84 85
grain: 73
granite: 6 82 85 86 89
gray: 59
green: 14 22 25 27 28 32 37 38
39 41 43 57 66 71 74 75
76 77 79 82 83 85 97
grey: 37 75 76 83 93
halt: 7
halted: 7
ham: 7 109
hand: 34 44 48 59 111 115
handbook: 52 117
handed: 33 100
handedness: 33 44
hardware: 8
heading: 101
headquarters: 9
height: 6 11 13 18 30 36 55 56 57
58 59 61 93 94 100 101
111 115 119
hicolor: 6
highlight: 22 23 71 80 81
highlighted: 70
highlighting: 6 78 79 80
highlights: 6 22 37 71 78 80 81 88
hints: 4 5 113
historical: 21
hole: 50 51 60 63
home: 9 17 20 93
honey: 52
horizons: 80
hotspot: 38

hourglass: 46
humans: 25
hyperboloid: 6
hyperboloids: 24 46
ibm: 6 7 8 9 13 14 15 18 36 50
57 58 113 115 118 119
identifier: 37
iff: 90 91 93 109
image: 4 5 6 7 11 12 13 14 15 16
17 18 19 21 22 34 35 36
55 56 57 59 73 90 91 92
93 94 95 98 100 102 103
108 109 112 115 116 117
inaccuracies: 61
include: 5 6 8 10 17 20 24 26 27
30 46 47 67 113 114
included: 7 8 9 10 11 12 14 22 25
30 38 57 58 59
includes: 5 9 20 96
index: 55 56 58 79 93 94 107
indigo: 57
indoors: 82
infinite: 24 25 45 60 63 69 110
infinitely: 91 110 111
infinity: 24 25
inlaid: 95
input: 11 13 14 17 20 26 110 112
inside: 39 47 48 53 61 62 63 64
67 87 111 112 113
install: 9
installation: 7 9
installing: 9
instance: 34 98
intermediate: 57
internet: 9 118 119
interpolate: 90 91 93 94 95
interpolation: 48 82 91 93 94 95
interpolations: 95
intersect: 65
intersected: 58
intersection: 31 40 48 58 60 61 63 64
65 66 67 80 110 111
intersections: 6 50 60 101
intersects: 58
inverse: 61 64
invert: 60
inverted: 101 115
invisible: 22 37 38 113
ior: 78 79 97

islands: 56
jade: 6 11 84 86
jaggies: 15 95
jaggy: 15 91 93 94 95
japan: 9
jittered: 15
jittering: 15
key: 18
keypress: 13
keyword: 9 23 28 30 34 35 38 39 41
49 52 61 64 72 81 84 89
91 92 93
keywords: 79 84 100
kilobytes: 16
landscape: 36
landscapes: 6 55 56 57
language: 5 6 10 19 25 26 27 30
latitude: 101
layer: 73 75 87 95
layered: 4 31 73 76 79 86 87 91 95
105
layering: 75
layers: 73 86
lemnisca: 51
lemniscate: 49 51
lens: 20 21 33 35 115
leopard: 6 86 120
library: 6 13 17 18 20 93 115
light: 5 17 22 23 25 32 36 37 38
39 70 74 75 77 78 79 80
81 88 101 111 112 113
lighting: 5 6 8 17 44 50
lights: 10 37 41
location: 20 33 34 36 37 38 55 71
100 101 110 111
longitude: 101
lowercase: 20
Macintosh: 7 9 115 119
macro: 30
mactech: 119
magnenat: 116
magnify: 23
magnitude: 108 111
mail: 118
map: 58 69 74 76 82 83 84 85
86 90 91 92 93 94 95 97
108 109
mapped: 58 85 90 91 92 93 94 108
112

mapping: 4 82 90 91 92 93 94 108
maps: 69 74 82 83 86 90 91 93 94 95 108 109 112 119
marble: 6 11 40 41 42 69 73 82 84 86 103 104 105
marching cubes: 107
masonry: 84
material: 6 40 90 94 95 103
materials: 6 8 69 105
math: 7 49 50
mathematical: 24 46 49 52 69 117
mathematically: 5 51 55
metal: 81 84 89 95
metallic: 78 80 81 89 97
metals: 6
mickey: 28 29
microscopic: 80
mirror: 6 25 80 108
mirrored: 25 78
mirroring: 85
model: 80 100
modelling: 53
modes: 91
monitors: 115
moose: 31
mountains: 6
moving: 5 72 111
mypic: 92
mysphere: 114
nature: 15
negate: 115
negative: 19 47 52 54 92 108 111
neighbor: 15
nested: 20 26 27
newsletter: 116
noise: 15 73 83 84 85 89 103 104 105
normalized: 91 95
normals: 48 110 112
north: 101 118
number: 9 15 16 17 24 27 30 32 52 53 55 56 57 59 74 82 85 88 92 93 94 98 104 108 113
numbers: 15 17 28 53 57 74 77 82 93
numeric: 30 32 82
numerical: 61
object: 6 14 15 21 22 23 24 25 26

28 29 30 31 32 38 39 40
41 42 43 44 45 47 48 49
50 59 60 61 64 65 66 67
68 71 72 73 74 75 77 78
80 81 82 83 84 90 91 92
96 98 101 102 105 106 107
108 109 110 111 113 114
objectrotation: 114
objects: 6 10 15 25 28 30 33 37 40
41 44 48 58 59 60 62 63
65 66 67 69 73 75 77 78
80 90 97 98 101 102 106
109 110 111 113
oblate: 45
ocean: 88
octaves: 97 105
onion: 6 86 120
opaque: 74 77
operating: 8 9 10
optimization: 111
optimize: 111
optimized: 67
optimum: 14
option: 14 15 16 17 18 20 41 83
88 93 102 113
optional: 15 27 49 56 71 72 86 92
options: 7 12 14 17 18 20 22 90 93
97
orange: 57 89
organ: 84
organic: 52
orientation: 16 29 33 36 101 109
orientations: 50
oriented: 85
origin: 24 45 46 47 48 58 113
orthogonal: 101
overflow: 98 105 110
override: 18 19 97
overwrite: 12
paint: 56 57 94
paints: 105
palette: 55 56 57 59 91 93 94
palettes: 91
panther: 11
papers: 111
paraboloid: 6 47
paraboloids: 24 46
parameter: 13 16 17 21 56 58 69 71
77 88 101 113

parameters: 12 13 14 17 18 27 40 69
70 71 88 114
parser: 97 106
parsing: 106
particle: 105
pathname: 17
pattern: 24 69 70 71 73 74 82 83
84 85 86 88 89 90 91 93
94 116
patterns: 23 69 70 74 82 83 85 88
pause: 13
PCGRAPHICS: 9
penumbral: 24
periodicals: 115
perspective: 5 21
perturb: 89
perturbation: 88 89
perturbed: 73
perturbs: 93
phase: 88 97
phone: 9 118
phong: 6 22 23 29 32 45 48 64 71
72 73 74 76 78 79 80 81
88 97 118
Piclab: 15 111
pine: 6
pinhole: 33
pink: 57
pinkalabaster: 66
pipe: 84
piriform: 51
pitch: 101
pixel: 14 15 36 55 57 58 93 94
95 101 102 109 115
pixels: 6 11 13 16 18 21 36 49 55
73 90 93 95 100 113
planar: 90 91 93 112
plane: 6 24 45 46 58 61 67 92
106 110 111 112
planes: 45 60 61 83 111
planets: 48
plastic: 77 80 81
point: 11 19 21 25 26 27 33 34
35 36 37 38 39 41 42 44
47 53 54 56 59 61 63 72
73 74 82 83 89 93 94 95
98 101 104 105 109 110
111
points: 24 27 35 37 38 44 46 53

59 61 83 100 104 110
polished: 80
polygon: 112
portability: 111
portable: 8
pot: 55 56 57 58
povibm: 7
povlegal: 117
povmap: 55
povray: 7 9 10 11 12 17 18 19 22
povrayopt: 12 18 19
preview: 59
primitive: 44 45 61
primitives: 6 48 61
prodigy: 118
project: 90 93
projection: 59 108
projections: 58
projector: 90 91 108
publisher: 116
putty: 52
qcone: 47
QRT: 14
quadric: 24 25 44 45 46 47 50 64
66 110
quadrics: 24 46 47 49 50 110 118
qualities: 113
quality: 6 13 17 41 69 113
quartic: 49 50 51 52
quartics: 6 49 50 119
radii: 54
radius: 21 23 24 37 38 39 44 47
49 50 51 52 53 54 55 90
98
random: 15 16 17 49 71 72 84 89
104 113
randomly: 58 104
randomness: 72 73 97
range: 6 11 13 17 77 78 80 81 82
83 88 89 92 105 108 112
ranges: 74 104
ranging: 69
raw: 6 13 14 15
raytracer: 26 58 59 99 101 102 107
110 113
raytracers: 19
raytracing: 9 16 45 104 111 115
readability: 26
readable: 25

realtime: 113
red: 11 14 15 21 22 24 27 28
32 37 38 39 41 57 66 71
74 75 76 77 79 82 83 85
86 97 103
reflect: 37 71 78
reflected: 17 25 79 98 101
reflecting: 98 101
reflection: 25 69 71 78 79 87 97
reflections: 5 88
reflective: 70 80 98
reflectivity: 44
reflects: 70 88 98
refracted: 17 78
refracting: 101 107
refraction: 78 79 97 107 112
refractive: 70
register: 92
registers: 14 91
Renderman: 116
repeatable: 15
requirements: 109
requires: 7 49 88 89
resolution: 56 95 100
reverse: 74
reversed: 108
revolved: 50
right: 11 19 20 21 22 23 25 27
33 34 35 36 42 43 44 73
100 101 108 109 112 115
ring: 50 51
rings: 69 104 114
ripple: 88
ripples: 6 70 87 88
rock: 103
roof: 65
room: 25 77 104 105
rotate: 27 28 35 36 40 41 42 44
45 46 47 48 58 70 71 72
73 88 92 101 112 114
rotated: 29 34 36 41 42 46 48 49
51 52 57 59 60 72 83 109
rotates: 21 42
rotating: 36 42 49 109
rotation: 42 44 48 114 115
rotations: 42 44 48
rough: 15 23 81
roughly: 23 53
roughness: 80 81 97 110

round: 70 115
rule: 23 61
saddle: 24 46
sandpaper: 112
sapphire: 6
satellite: 24
saturation: 80
scale: 23 25 28 32 40 41 42 43
45 46 47 48 52 58 59 64
66 70 71 72 73 74 75 76
85 88 89 92 93 110 112
114
scaled: 23 25 29 41 45 46 48 49
52 56 57 59 60 72 83 91
93 109 110 114
scales: 40 41 51
scaling: 23 25 44 51 84 85 86 93
109 110 111
scan: 16
scanline: 16
scanlines: 16
scanned: 57
scene: 5 6 7 8 9 10 11 12 13 15
16 17 18 19 20 21 22 25
26 27 28 29 30 32 33 35
37 39 40 41 46 55 71 78
96 97 98 99 100 108 110
111 113 114 117 119
scenes: 5 6 8 10 11 21 27 49 59
67
screen: 11 16 19 21 33 36 101 110
111
screens: 36
SGI: 8
shade: 85
shaded: 6 48 70
shades: 74
shading: 5 69 70 88
shadow: 24 38 39 77 108 111
shadowed: 78 101
shadows: 17 24 37 111
shape: 6 21 24 29 30 31 36 38 39
40 41 42 44 45 47 48 49
51 52 56 58 59 60 61 62
63 64 65 66 67 68 69 70
71 72 73 77 83 88 90 91
93 103 104 106 113 114
shapes: 5 6 8 20 24 28 30 31 37
38 40 41 44 46 47 48 49

50 51 52 58 59 60 61 62
63 65 66 67 68 69 91 102
106 110 115
shapesq: 49
shell: 104
shinier: 80
shininess: 78
shining: 23 77 78
shiny: 6 22 23 70 71 78 79
sideways: 101
Sierra: 6
SIGGRAPH: 103 116
sign: 27 33
sine: 55
sky: 6 33 34 35 57 73 83 98
101
slash: 26
slide: 90 91 108
smooth: 6 15 23 48 49 54 57 59 70
74 81 82 85 104 112
smoothed: 91 93
smoother: 15 56 57 95 102
smoothing: 13
smoothly: 52 53 73 74 104
smooths: 95
soft: 24 33 37 38 111
software: 5 7 8 117
solid: 6 56 58 59 60 62 69 73 83
103 113
solidly: 83
south: 101
spatial: 111
specular: 6 22 78 79 80 81 97
speed: 16 45 65 67 111
speeds: 50 67
spelling: 77
sphere: 6 21 22 23 24 28 30 31 32
39 40 41 42 43 44 45 47
48 50 53 54 60 64 65 66
67 68 70 72 73 74 75 78
90 91 93 94 95 106 114
spheres: 19 39 44 45 46 52 53 54
59 61 86 98 115
spherical: 90 91 93 94
spheroids: 45
spikes: 58
spin: 48
spline: 82 112
spotlight: 37 38 39 111

spotlights: 6 38 119
spotted: 82 84 89
sqrt: 49
square: 36 56 108 115
squares: 83
squig: 94 95
squiggles: 94 95
squish: 23 42
stack: 98 105
stdinc: 114
steel: 29
steering: 29
stone: 9 84 86
stones: 69 86 119
storage: 57
stormy: 84
Sturm: 49 52
subdirectories: 10
subdirectory: 9 10 20 93
sunset: 6
supercomputer: 8
superimpose: 60
surface: 6 17 21 24 25 44 45 46 47
48 49 50 53 54 59 61 69
70 71 73 77 78 79 81 83
87 88 89 90 92 93 94 98
107 111 112
surfaces: 6 24 41 46 47 49 50 51 52
60 112 117
swirled: 84
syntax: 30 36 37 38 42 44 47 49
52 55 59 71 77 82 83 84
85 86 88 89 90 93 94 97
98
Targa: 6 13 14 15 17
telephoto: 33
temperature: 104 105
temperatures: 104
temple: 11
texture: 6 17 21 22 23 24 25 28 29
31 32 38 39 40 41 42 43
45 60 64 65 66 70 71 72
73 74 75 76 77 78 79 81
82 84 85 86 87 88 90 91
92 94 95 97 98 103 104
105 106 107 109 112 113
114 115
textured: 29 82 103
textures: 4 5 6 8 16 17 20 23 28 30

                    31 41 47 69 71 72 73 74
                    75 76 79 81 86 87 88 89
                    90 91 93 94 95 97 98 103
                    104 105 106 109 112 113
                    114 120
texturing: 83 105
thermometer: 104
thick: 111
thickness: 110
thin: 58 111
threshold: 15 52 53 54
tile: 86
tiled: 11 91
tiles: 86 87 94
tips: 4 5 24 113
tolerance: 15
torus: 6 25 49 50 51 90 91
trace: 9 27 52 98 107
traceaholics: 9
traced: 15 44 101
tracer: 5 24 25 26 33 40 56 58 59
                    67 70 74 96 102 113
tracers: 33
tracing: 5 8 13 14 16 20 33 35 37
                    58 65 67 78 83 100 102
                    110 115
transform: 46 48 71 72
transformation: 40 41 43 48 70 85
transformations: 40 41 48 58 65 70 71 72
                    109 114
transformed: 41 58 109
transforming: 43
translate: 25 28 29 35 36 41 42 43
                    45 46 47 48 58 64 66 67
                    70 71 72 73 85 88 92 112
                    114
translated: 29 36 41 46 48 49 51 52
                    57 59 60 72 83 109
translates: 43 74
translating: 36 43 49 109
translation: 48
translations: 44
transmit: 79
transmitted: 17
transparency: 69 73 77 78 92 112
transparent: 38 69 70 73 74 75 76 77
                    78 79 89 91 92 98
transposed: 108
tree: 10 30 69 104

triangle: 6 48 55 58 62 112
triangles: 6 48 55 56 57 58 59 61 62 110 112
turbulent: 83 84
tutorial: 5 19
undefined: 67 91 113
underflow: 110
union: 28 29 31 39 40 48 60 63 65 66 98 109
unions: 6 60
unit: 21 23 27 55 56 58 91 94 114
units: 20 21 22 27 42 43 45 46 47
universe: 20 27 33 35 100
unix: 8
utilities: 8 48 59 112
utility: 10 59
vapor: 83
vaqueros: 117
variable: 12 17 18 19 106
variables: 49
vax: 8
vector: 21 24 27 33 34 35 44 49 84 85 92 93 100 101 108 110 111 112 115
vectors: 20 27 32 33 34 35 36 41 100 101 110 112 114 115
vein: 74
veins: 74
verbose: 86
verlag: 116
vertex: 58 112
vertexes: 58
vertical: 69
vertices: 48
VGA: 36 115
viewplane: 100
viewpoint: 33
virtual: 22
visible: 38 60 73
vision: 5
visual: 53
visualize: 104
volume: 113
volumes: 113
Waite Group: 115
wall: 45 46
water: 55 56 58 79 83 88

wave: 88
waves: 6 70 87 88 89
west: 9 101
wheel: 29
wheels: 65
white: 22 27 28 37 39 57 59 75
76 77 78 83 92 93 96 103
wide: 6 11 21 33 36 47 55 115
width: 11 13 18 21 36 47 49 100
110 115
wind: 105
windows: 7 8
wood: 6 23 69 71 73 74 75 76 82
84 86 88 104 105 106 107
114
wooden: 24 95
wrinkle: 89
wrinkled: 6 89
wrinkles: 70 89
wrinkling: 89
yellow: 63 64 66 82 83
zero: 16 25 38 43 53 54 56 72
78 108 110
zip: 7 114