# DKBTrace Ray-Tracer, Version 2.12

David K. Buck
22C Sonnet Cres.
Nepean, Ontario
Canada, K2H 8W7

"It's free, and it's well worth the price"

June 14, 1991

# Contents


**1 Copyrights and Disclaimers** **1**

**2 System Overview** **3**

**3 Getting Started** **4**

**4 Command Line Options** **6**

**5 A Tutorial Walkthrough** **11**
5.1 The First Image . . . . . 11
5.2 Phong Highlights . . . . . 14
5.3 Textures . . . . . 14
5.4 Other Shapes . . . . . 15
5.5 Constructive Solid Geometry . . . . . 16
5.6 Advanced Textures . . . . . 19
5.7 Walk-through Wrap-up . . . . . 22

**6 The Scene Description Language** **23**
6.1 The Basic Data Types . . . . . 23
6.1.1 Floats . . . . . 23
6.1.2 Vectors . . . . . 24
6.1.3 Transformations – `TRANSLATE`, `ROTATE`, and `SCALE` . . 24
6.1.4 Colours . . . . . 25
6.1.5 Colour Maps . . . . . 25
6.2 More Complex Data Types – `DECLARE` . . . . . 26
6.3 Viewpoint . . . . . 28
6.4 Fog . . . . . 30
6.5 Shapes . . . . . 30
6.5.1 Quadric Shapes . . . . . 31

6.5.2 Quadric surfaces the easy way . . . 31
6.5.3 Spheres . . . 32
6.5.4 Planes . . . 33
6.5.5 Triangles . . . 33
6.5.6 Quartic Surfaces . . . 34
6.5.7 Constructive Solid Geometry . . . 37
6.6 Objects . . . 38
6.6.1 COLOUR . . . 39
6.6.2 TRANSLATE, ROTATE, and SCALE . . . 39
6.6.3 LIGHT_SOURCE . . . 40
6.7 Texture . . . 41
6.7.1 Object Surface Lighting Characteristics . . . 42
6.7.1.1 AMBIENT . . . 42
6.7.1.2 DIFFUSE . . . 42
6.7.1.3 BRILLIANCE . . . 42
6.7.1.4 REFLECTION . . . 43
6.7.1.5 REFRACTION . . . 43
6.7.1.6 IOR . . . 43
6.7.1.7 PHONG . . . 44
6.7.1.8 PHONGSIZE . . . 44
6.7.1.9 SPECULAR . . . 44
6.7.1.10 ROUGHNESS . . . 44
6.7.1.11 METALLIC . . . 45
6.7.2 Surface Colouring Textures . . . 45
6.7.2.1 CHECKER . . . 45
6.7.2.2 CHECKER_TEXTURE . . . 45
6.7.2.3 BOZO . . . 46
6.7.2.4 SPOTTED . . . 47
6.7.2.5 AGATE . . . 48
6.7.2.6 GRADIENT . . . 48
6.7.2.7 GRANITE . . . 49
6.7.3 Surface Perturbation Textures . . . 49
6.7.3.1 RIPPLES . . . 49
6.7.3.2 WAVES . . . 49
6.7.3.3 PHASE . . . 50
6.7.3.4 FREQUENCY . . . 50
6.7.3.5 BUMPS . . . 50
6.7.3.6 DENTS . . . 50
6.7.3.7 WRINKLES . . . 50

6.7.4 IMAGEMAP . . . . . 51
6.8 Composite Objects . . . . . 52
6.9 Bounding Shapes . . . . . 53

**7 Displaying the Images** **55**
7.1 Amiga Systems . . . . . 55
7.2 IBM Systems . . . . . 56
7.3 Unix Systems . . . . . 58

**8 DKBTrace Utilities** **59**
8.1 Data File Creation Utilities . . . . . 59
8.1.1 SA2DKB . . . . . 59
8.1.2 DXF2DKB . . . . . 60
8.1.3 ShellGen . . . . . 60
8.1.4 Twister . . . . . 60
8.1.5 Chem2DKB . . . . . 60
8.1.6 Lissajou . . . . . 60
8.2 Output File Manipulation Utilities . . . . . 61
8.2.1 dump2i24 . . . . . 61
8.2.2 catdump . . . . . 61
8.2.3 combdump . . . . . 61
8.2.4 dump2mtv . . . . . 61
8.2.5 dump2raw . . . . . 62
8.2.6 halftga . . . . . 62
8.2.7 gluetga . . . . . 62
8.2.8 tga2dump . . . . . 62
8.3 Animation Utilities . . . . . 62
8.3.1 RayScene . . . . . 62

**9 How it All Works (or How to Get What You Want)** **64**
9.1 Viewpoints . . . . . 64
9.2 Ray-Object Intersections . . . . . 65
9.3 Transparency and Refraction . . . . . 66
9.4 Textures, Noise, and Turbulence . . . . . 67
9.5 Layered Textures . . . . . 70
9.6 Image Mapping . . . . . 71
9.7 Output File Formats . . . . . 72

**10 Common Questions and Answers** **75**

11 Converting Data Files from Versions Prior to 2.10 82

12 Handy Hints 85

13 Compiling the Code 87

14 Porting to Different Platforms 88

15 Program History and Information 90

16 References 94

17 Concluding Remarks 96

18 About the Authors 97

# List of Figures


5.1 DKBTrace Coordinate Axes . . . . . 11
5.2 Sample 2D figures . . . . . 17
5.3 `UNION` of the Sample Figures . . . . . 18
5.4 `INTERSECTION` of the Sample Figures . . . . . 18
5.5 `DIFFERENCE` of the Sample Figures . . . . . 18
5.6 `INVERSE-DIFFERENCE` of the Sample Figures . . . . . 19

6.1 Left-hand Coordinate System . . . . . 25
6.2 The Difference between `SKY` and `UP` . . . . . 29
6.3 Radii of a torus . . . . . 36

9.1 Viewpoint definition vectors . . . . . 65
9.2 Layered color calculation . . . . . 67
9.3 Marble texture . . . . . 68
9.4 Wood texture . . . . . 68
9.5 Effect of turbulence . . . . . 69
9.6 Object parsing, step 1 . . . . . 70
9.7 Object parsing, step 2 . . . . . 71
9.8 Object parsing, step 3 . . . . . 71
9.9 Parallel projection . . . . . 72
9.10 Image mapping with vector `<1 -1 0>` . . . . . 73
9.11 Image mapping with vector `<-1 1 0>` . . . . . 73

10.1 The "sides" of a plane . . . . . 78
10.2 Distortion with short `DIRECTION` vector . . . . . 79

# Chapter 1

# Copyrights and Disclaimers

DKBTrace has been made freely distributable. The author retains the copyright to the program but authorizes free distribution by BBS'es, networks or by magnetic media. The distributor may choose to charge for the cost of the disk but must not sell the software for profit. Non-profit organizations such as clubs may charge for the software so long as the price is reasonable (less than $5.00 more than the cost of the disk) and so long as the buyers are informed that the program is freely distributable.

The images and data files generated by the raytracer are the property of the user of the software and may be used for any purpose without restriction.

The author makes no guarantees or warranties with this program and claims no responsibility for any damage or loss of time caused by this program. Bug reports may be sent to the author but the author is under no obligation to provide bug fixes, features, or any support for this software.

I would also like to place the following conditions on the use of this program:

1. that it should not be used as part of any commercial package without my explicit written consent.

2. if you make any neat and interesting pictures, please send them to me.

3. If you make any changes to the source code, please let me know. I'd like to see what you've done.

4. This text file should accompany the program.

The GIF file reader was written by Steven A. Bennett. Here's his copyright notice:

```
* DECODER.C - An LZW decoder for GIF
* Copyright (C) 1987, by Steven A. Bennett
*
* Permission is given by the author to freely redistribute and include
* this code in any program as long as this credit is given where due.
*
* In accordance with the above, I want to credit Steve Wilhite, who
* wrote the code which this is heavily inspired by...
*
* GIF and 'Graphics Interchange Format' are trademarks (tm) of
* Compuserve, Incorporated, an H&R Block Company.
```

The `Noise` and `DNoise` functions (used for texturing) were written by Robert Skinner and used here with his permission.

# Chapter 2

# System Overview

This program is a ray tracer written completely in C. It supports arbitrary quadric surfaces (spheres, ellipsoids, cones, cylinders, planes, etc.), constructive solid geometry, and various shading models (reflection, refraction, marble, wood, and many others). It also has special-case code to handle spheres, planes, triangles, and smooth triangles. By using these special primitives, the rendering can be done much more quickly than by using the more general quadrics. In order to create pictures with this program, you must describe the objects in the world. This description is a text file called *filename*`.data`, and *filename* defaults to `object` if not specified. Normally, such files are difficult to write and to read. In order to make this task easier, the program contains a parser to read the data file. It allows the user to easily create complex worlds from simple components. Since the parser allows include files, the user may put the object descriptions into different files and combine them all into one final image.

This document is organized as follows. The first chapters describe how to run DKBTrace, the various options you can give to alter its behaviour, and provide a "walk-through" tutorial to get you familiar with its concepts and features. The largest chapter then specifies the scene description language in detail, and serves as a reference manual for the system. Subsequent chapters discuss issues related to DKBTrace, such as common questions (with answers), compiling and porting the system, utilities for use with DKBTrace, and system history and compatibility notes. Finally, a short bibliography of ray-tracing references and an index are provided.

# Chapter 3

# Getting Started

This chapter describes what you need to do to jump right in and start generating images. The next chapter describes the command-line options in more detail. Here's what you need to do:

1. Put the file `sunset.dat` into your current directory.

2. Make sure the files `colors.dat`, `shapes.dat`, and `textures.dat` are present.

3. Set up the default parameters. This can be done by creating a file called `trace.def` or by setting the `DKBOPT` environment variable. In either case, use the following line (you may change these defaults to suit your system):

   ```
   -w320 -h400 -v +f +d +p +x -a
   ```

   Meaning:

   `-v` Don't be verbose, i.e. don't show line numbers during trace.

   `+f` Write an output file in "dump" or "Targa" format (machine specific; IBM default is "Targa", Amiga and Unix default is "dump").

   `+d` Display the image while rendering (on some systems, an additional character may follow this option to specify the graphics mode to use for the display).

   `-w320` Make the image 320 pixels wide.

   `-h400` Make the image 400 pixels high.

`+p` Prompt before exitting to let you look at the picture.

`+x` Allow exitting with a key hit before the trace is finished.

`-a` Don't Antialias (Antialiasing smooths out jagged edges).

4. To render a scene, type:

   `dkb`*xxx* `-isunset.dat -osunset.dis`

   Meaning:

   `dkb`*xxx* On different systems, the name of the executable may vary. Check to see what it is on your system. For the Amiga, for example, two versions are supplied: `dkb881` and `dkbieee` for systems with and without a floating-point accelerator. For the IBM, `DKB.EXE` is the 20286 optimized/uses 80287 math coprocessor version, and `DKBN087.EXE` is the plain 8086/no 8087 math coprocessor version.

   `-isunset.dat` Read the input file `sunset.dat`.

   `-osunset.dis` Call the output file `sunset.dis`. This is the usual file name extension for "dump" format. "Targa" format files (default for IBM's) generally use the extension `.tga`.

5. Once the image has been rendered, you must use a post-processor to create the final viewable image file (i.e. an IFF or GIF file), unless you possess 24-bit display hardware and can view the generated output files directly. The post-processor used depends on your system. See the section "Displaying the Images" for more details on post-processing the image.

# Chapter 4

# Command Line Options

This program is designed to be run from a command line. The command-line options may be specified in any order. Repeated options overwrite the previous values. Options may also be specified in a file called `trace.def` or by the environment variable `DKBOPT`.

`-w`*width* Width of the picture in pixels. (On the Amiga, use 320 for full-sized pictures.)

`-h`*height* Height of the picture in pixels. (On the Amiga, use 400 for full-sized pictures.)

`+v` Verbose option.
`-v` Disable verbose option.

In verbose mode, the scan line number is printed as each line is traced.

`+f`*x* Produce an output file.
`-f` Don't produce an output file.

If the `+f` option is used, the ray tracer will produce an output file of the picture. This output file describes each pixel with 24 bits. Currently, three formats of output files are supported:

`+fd` Default – Dump format (QRT-style).
`+fr` Raw format – three files for R, G and B.
`+ft` Uncompressed Targa-24 format.

Normally, a post-processor is required to create the final finished image from the data file. See the section on "Displaying the Images" for details.

`+d`*x* Display the picture while tracing.
`-d` Don't display the picture while tracing.

If the `+d` option is used, then the picture will be displayed while the program performs the ray tracing. On most systems, this picture is not as good as the one created by the post-processor because it does not try to make optimum choices for the colour registers.

Depending on the system, a letter may follow the `+d` option to specify the graphics mode to use.

**All systems:**

`+d` Default Format (same as `+d0`)

**Amiga:**

`+d0` Ham format
`+dE` Ham-E format

**IBM:**

`+d0` Autodetect (S)VGA type
`+d1` Standard VGA 320x200
`+d2` Simulated SVGA 360x480
`+d3` Tseng Labs 3000 SVGA 640x480
`+d4` Tseng Labs 4000 SVGA 640x480
`+d5` AT&T VDC600 SVGA 640x400
`+d6` Oak Technologies SVGA 640x480
`+d7` Video 7 SVGA 640x480
`+d8` Video 7 Vega (Cirrus) VGA 360x480
`+d9` Paradise SVGA 640x480
`+dA` Ahead Systems Ver. A SVGA 640x480
`+dB` Ahead Systems Ver. B SVGA 640x480
`+dC` Chips & Technologies SVGA 640x480
`+dD` ATI SGVA 640x480
`+dE` Everex SVGA 640x480
`+dF` Trident SVGA 640x480
`+dG` VESA Standard SVGA Adapter 640x480

`+p` Wait for prompt (IBM: beep and pause) before quitting.

`-p` Finish without waiting.

The `+p` option makes the program wait for a carriage return before exitting (and closing the graphics screen). This gives you time to admire the final picture before destroying it.

`-i`*filename* Set the input filename.
`-o`*filename* Set the output filename.

If your input file is not `Object.dat`, then you can use `-i` to set the filename. The default output filename will be `data.dis` for dump mode, `data.red`, `data.grn` or `data.blu` for raw mode, and `data.tga` for Targa mode. If you want a different output file name, use the `-o` option. (On IBM's, the default extensions for raw mode are `.r8`, `.g8`, and `.b8` to conform to PICLAB's "raw" format.)

`+a`*level* Anti-alias – *level* is an optional tolerance level (default 0.3).
`-a` Don't anti-alias.

The `+a` option enables adaptive anti-aliasing. The number after the `+a` option determines the threshold for the anti-aliasing. If the colour 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 supersampled. If the threshold is 1.0, then no anti-aliasing is done. 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.

`+x` Allow early exit by hitting any key. [IBM only]
`-x` Lock in trace until finished. [IBM only]

On the IBM, the `-x` option disables the ability to abort the trace by hitting a key. If you are unusually clumsy or have cats that stomp on your keyboard, you may want to use it. If you are writing a file, the system will recognize `Ctrl-C` at the end of line if the system `BREAK` is `ON`. If you aren't writing a file, you won't be able to abort the trace until it's done.

This option was meant for big, long late-nite traces that take all night (or longer!), and you don't want them interrupted by anything less important than a natural disaster such as hurricane, fire, flood, famine, etc.

`-b`*size* Use an output file buffer of *size* kilobytes.

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 (especially for floppies). If this parameter is zero (the default), 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 on disk.

`+c` Continue Rendering.

If, for some reason, you abort a raytrace while it's in progress or if you used the `-e` option (below) to end the raytrace prematurely, you can use the `+c` option to continue the raytrace 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.[1]

`-s`*line* Start tracing at line number *line*.
`-e`*line* End tracing at line number *line*.

The `-s` option allows you to start rendering an image starting from 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. Alternatively, you can render groups of scanlines on different systems and concatenate them later. WARNING: 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. There is an example of a standard ANSI-C random number generator in the file `IBM.C`. Cut it out and paste it into your machine-specific `.c` file if you plan to try "distributed processing" and are not sure if you need this standardization.

[1] If you want to impress your friends with the speed of your computer, take an image you've already rendered and use `+c` in the command-line. It renders *real* fast that way!

The `-s` option has no effect when continuing a raytrace using the `+c` option. The renderer will figure out where to restart.

`-q`*level* Rendering quality.

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

0,1: Just show colours. Ambient lighting only.
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* The `-l` option may be used to specify a "library" pathname to look into for data files to include or for images. 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.

`+z` The `+z` option is an undocumented feature. You will not see any references to it in this or any other documentation file for DKBTrace. In fact, no other section of the document will even admit that it was mentioned here. If you really want to know what it does, then you will have to look into the source code (`trace.c`) and read the comment just above the `+z` option that says "Turn on debugging print statements." The full purpose of this option will, therefore, be left as an exercise for the reader, but believe me – it's nothing terribly exciting.[2]

You may specify the default parameters by modifying the file `trace.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 also set the environment variable `DKBOPT` to the desired set of command-line parameters.

[2]For those people who run the raytracer on super-fast systems and want to slow it down, you may try this option.

# Chapter 5

# A Tutorial Walkthrough

This section, is designed to get you up and running designing your own pictures without all the nit-picky details. Once you've made a few of your own data files, you'll probably want to advance to the next section to fill in the gaps.

## 5.1 The First Image

Let's get right to the meat of the matter and create the data file for a simple picture. Since raytracers thrive on spheres, that's what we'll render first.

First we have to create a viewpoint to tell the computer where our camera is and where it's looking. To do this, we use 3D coordinates. The usual coordinate system for DKBTrace has Y pointing up, X pointing to the right, and Z pointing into the screen as shown in Figure 5.1.

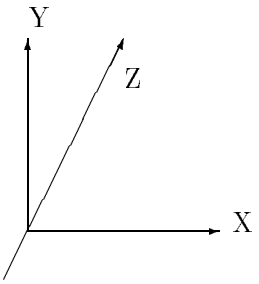


Figure 5.1: DKBTrace Coordinate Axes

Using your personal favorite text editor (i.e., user interface), create a file

called `picture1.dat`. Now, type in the following:[1]

```
INCLUDE "colors.dat"
INCLUDE "shapes.dat"
INCLUDE "textures.dat"

VIEW_POINT
   LOCATION  <0 0 0>
   DIRECTION <0 0 1>
   UP        <0 1 0>
   RIGHT     <1.33333 0 0>
END_VIEW_POINT
```

The first `INCLUDE` statement reads in definitions for various useful colours. (Being a proud Canadian, I spell colour the proper way with a "u". To avoid confusing the rest of the world, however, I've set up the raytracer to allow either spelling of the word.)

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. The total pre-defined number of `INCLUDE`'d files (nested or not) per scene is 10.

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, `shapes.dat`, `colors.dat`, and `textures.dat` in an "include" subdirectory, and giving an `-l` option on the command line to where your library of include files are.

This viewpoint declaration puts our camera at the center of the universe (`LOCATION <0 0 0>`) pointing into the Z direction (`DIRECTION <0 0 1>`) and with the camera being held upright (`UP <0 1 0>`). The final term compensates for the aspect ratio of the screen (`RIGHT <1.33333 0 0>`). If your computer has square pixels, you may want to change this to "`RIGHT <1.0 0 0>`". For details on exactly how the camera works, see the section "How it All Works".

Now, let's place a red sphere into the world:

```
OBJECT
```

[1] The input is case sensitive, so be sure to get capital and lowercase letters correct.

```
   SPHERE <0 0 3> 1 END_SPHERE
   TEXTURE
      COLOUR Red
   END_TEXTURE
END_OBJECT
```

This sphere is 3 units away from the camera and has a radius of 1. 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-END_TEXTURE` block. In this case, we are just setting the colour.

One more detail – we need a light source:

```
OBJECT
   SPHERE <0 0 0> 1 END_SPHERE
   TEXTURE
      COLOUR White
   END_TEXTURE
                        { This is 2 units to our right, }
   TRANSLATE <2 4 -3>   { 4 units above, and 3 units    }
                        { behind our camera.            }
   LIGHT_SOURCE
   COLOUR White
END_OBJECT
```

Note: For light sources, *always* declare them to be centered at the origin `<0 0 0>`, then use `TRANSLATE` to put them where you want. If you don't do this, the light source won't work right. We must also specify the colour of the light source *outside* the `TEXTURE` block because the renderer doesn't want to work out the whole surface colour just to get the colour of the light it emits.

That's it! Close the file and render a small picture of it:[2]

```
trace -w80 -h100 -f -ipicture1.dat
```

On the IBM, the command line would be:[3]

```
dkb -w80 -h50 -f -ipicture1.dat
```

[2]The program name may vary on different systems.

[3]The program name would be `dkbno87` if you have an 8086/8088 system or no math co-processor.

## 5.2 Phong Highlights

You've now rendered your first picture. I know you want to run out and show all your friends how amazing your computer is to be able to generate such an incredible picture, but just wait a few minutes – you ain't seen nothin' yet. (For those people who complained that the picture took too long to draw, just wait – you ain't seen nothin' yet, either...)

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 0 3> 1 END_SPHERE
   TEXTURE
      COLOUR Red
      PHONG 1.0
   END_TEXTURE
END_OBJECT
```

Now render this. In all seriousness, the `PHONG` highlight does add a lot of credibility to the picture. You'll probably want to use it in many of your pictures.

## 5.3 Textures

One of the really nice features of this raytracer is its sophisticated textures. Change the definition of our sphere to the following and then re-render it:

```
OBJECT
   SPHERE <0 0 3> 1 END_SPHERE
   TEXTURE
      Dark_Wood
      SCALE <0.2 0.2 0.2>
      PHONG 1.0
   END_TEXTURE
END_OBJECT
```

The textures are set up by default to give you one "feature" across a sphere of radius 1.0. A "feature" is roughly defined as a colour transition. For example, a wood texture would have one band on a sphere of radius 1.0.

By scaling the wood by `<0.2 0.2 0.2>`, we shrink the texture to give us about five bands. Please note that this 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. Don't start reporting problems if you get three bands instead of five. This rule of thumb just puts you in the ballpark.

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 in the left-right direction. The second term controls the up-down direction and the third term controls the front-back direction.

I encourage you to look through the `textures.dat` file to see what textures are defined there and try them out. Some of them are quite spectacular.

## 5.4 Other Shapes

So far, we've just defined spheres. There are several other kinds of shapes that can be rendered by DKBTrace. Let's try one out with a computer graphics standard – a checkered floor. Add the following object to your `.dat` file:

```
OBJECT
   PLANE <0.0 1.0 0.0> -1.0 END_PLANE
   TEXTURE
      CHECKER
         COLOUR RED 1.0
         COLOUR BLUE 1.0
   END_TEXTURE
END_OBJECT
```

The object defined here is an infinite plane. The vector `<0.0 1.0 0.0>` is the surface normal of the plane (i.e., if you where standing on the surface, the normal points straight up.) The number afterward is the distance that the plane is displaced along the normal – in this case, we move the floor down one unit so that the sphere (radius 1) is resting on it. The checker texture specifies the two colours to use in the checker pattern.

Looking at the floor, you'll notice that the wooden ball casts a shadow on the floor. Shadows are calculated accurately (well, almost – more later) by the raytracer.

Another kind of shape you can use is called a quadric surface. To be totally honest, the shapes you've been using so far have been quadrics. Spheres

and planes are types of quadric surfaces. There are many other quadric surfaces however. 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 as well as the spheres and planes we've used so far.

All quadrics except for ellipsoids and spheres 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.

Enough talk – let's render one of these "quadrics"...While we're at it, we'll add a few features to the surface. Add the following definition to your `.dat` file:

```
OBJECT
   QUADRIC Cylinder_Y END_QUADRIC
   TEXTURE
      COLOUR GREEN 0.5
      REFLECTION 0.5
   END_TEXTURE
   SCALE <0.4  0.4  0.4>
   TRANSLATE <2 0 5>
END_OBJECT
```

This object is a cylinder along the Y (up-down) axis. It's green in colour and has a mirrored surface (hence the reflection of 0.5) this means that half the light coming from the sphere 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. (Don't use 0, though. That will definitely cause a fatal crash!) Finally, the cylinder has been moved back and to the right so you can see it more clearly.

## 5.5 Constructive Solid Geometry

The shapes we've talked about 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, 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).

Before getting into CSG, however, let me talk about inside and outside. Every primitive (except triangles – I'll talk about this later) divides the world into two regions. One region is inside the surface and one is outside. So, 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 usually 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 unimportant, 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. Consider the situation in Figure 5.2.[4]

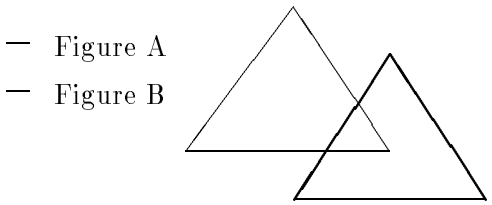


Figure 5.2: Sample 2D figures

There are three CSG operations you can use:

`UNION A B END_UNION` 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, as shown in Figure 5.3.

`INTERSECTION A B END_INTERSECTION` 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., as shown in Figure 5.4.

[4]The diagrams shown here demonstrate the concepts in 2D and are indended only as an analogy to the 3D case. Note that the triangles supported by DKBTrace cannot be used in CSG (except for unions) since they have no inside and outside.

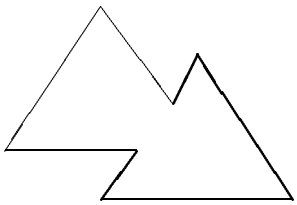

Figure 5.3: `UNION` of the Sample Figures

Figure 5.4: `INTERSECTION` of the Sample Figures

`DIFFERENCE A B END_DIFFERENCE` A point is inside the difference if it's inside A but not inside B. The result is a "subtraction" of the 2nd shape from the first shape, shown in Figure 5.5.

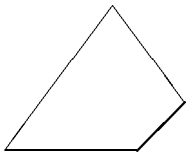

Figure 5.5: `DIFFERENCE` of the Sample Figures

Let's give a concrete example by drilling a yellow hole through our sphere. Go to the definition of the sphere and change it to read the following:

```
OBJECT
   DIFFERENCE
      SPHERE <0 0 3> 1 END_SPHERE
      QUADRIC
         Cylinder_Z
         SCALE <0.2 0.2 0.2>
         COLOUR Yellow
      END_QUADRIC
   END_DIFFERENCE
   TEXTURE
      Dark_Wood
      SCALE <0.2 0.2 0.2>
```

```
      PHONG 1.0
    END_TEXTURE
END_OBJECT
```

One more point about CSG operations. 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, the result of

```
INTERSECTION B A-INVERSE END_INTERSECTION
```

yields the shape shown in Figure 5.6. Note that a `DIFFERENCE` is really just

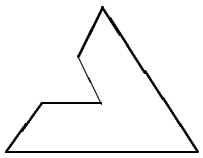

Figure 5.6: `INVERSE-DIFFERENCE` of the Sample Figures

an `INTERSECTION` of one shape with the `INVERSE` of another. This happens to be how `DIFFERENCE` is actually implemented in the code.

## 5.6 Advanced Textures

The textures available in DKBTrace are 3D solid textures. This means that the texture defines a colour for any 3D point in space. Just like a real block of marble or wood, there is colour 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 colours in the texture – you only see the colours 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.

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
      Dark_Wood
      TURBULENCE 0.0
      SCALE <0.2 0.2 0.2>
      PHONG 1.0
   END_TEXTURE
```

When you compare this with the original image, you'll see that the pattern is much more boring.

Finally, many textures use colour maps. A colour map translates a number between 0.0 and 1.0 into a colour. The number typically represents the distance into a vein of colour – the further into the vein you get, the more the colour changes. Here's a typical colour map. Try this out on the sphere defined above by changing the definition to this:

```
OBJECT
   SPHERE <0 0 3> 1 END_SPHERE
   TEXTURE
      WOOD
      SCALE <0.2 0.2 0.2>
      COLOUR_MAP
          [0.0 0.3  COLOUR Red   COLOUR Green]
          [0.3 0.6  COLOUR Green COLOUR Blue]
          [0.6 1.01 COLOUR Blue  COLOUR Red]
      END_COLOUR_MAP
      PHONG 1.0
   END_TEXTURE
END_OBJECT
```

This means that as the texture enters into a vein of wood, it changes colour 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 colour should show up quite well.)

You can get more "bang for your buck" from textures by using `ALPHA` properties of colour. Every colour you define in DKBTrace is a combination of red, green, blue and alpha. The red, green and blue are simple enough. The alpha determines how transparent that colour is. A colour with an alpha of 1.0 is totally transparent. A colour with an alpha of 0.0 is totally opaque. Here's a neat texture to try:

```
TEXTURE
   TURBULENCE 0.5
   BOZO
   COLOUR_MAP
      { transparent to transparent }
      [0.0 0.6 COLOUR RED 1.0 GREEN 1.0 BLUE 1.0 ALPHA 1.0
               COLOUR RED 1.0 GREEN 1.0 BLUE 1.0 ALPHA 1.0]
      { transparent to white }
      [0.6 0.8 COLOUR RED 1.0 GREEN 1.0 BLUE 1.0 ALPHA 1.0
               COLOUR RED 1.0 GREEN 1.0 BLUE 1.0]
      { white to grey }
      [0.8 1.001 COLOUR RED 1.0 GREEN 1.0 BLUE 1.0
                 COLOUR RED 0.8 GREEN 0.8 BLUE 0.8]
   END_COLOUR_MAP
   SCALE <0.4  0.08  0.4>
END_TEXTURE
```

This is my (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.

Now for one more feature which is new for 2.10 (hold onto your hats!) You can now layer textures one on top of another to create more sophisticated textures. For example, suppose I want a wood-coloured cloudy texture. What I do is put the wood texture down first followed by my 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 0 3> 1 END_SPHERE
   TEXTURE                { This is the wood texture }
      Dark_Wood           { we used earlier.         }
      TURBULENCE 0.0
      SCALE <0.2 0.2 0.2>
      PHONG 1.0
   END_TEXTURE
   TEXTURE                { This is the cloud texture }
     TURBULENCE 0.5       { we just defined.          }
     BOZO
     COLOUR_MAP
       { transparent to transparent }
```

```
        [0.0 0.6 COLOUR RED 1.0 GREEN 1.0 BLUE 1.0 ALPHA 1.0
                 COLOUR RED 1.0 GREEN 1.0 BLUE 1.0 ALPHA 1.0]
        { transparent to white }
        [0.6 0.8 COLOUR RED 1.0 GREEN 1.0 BLUE 1.0 ALPHA 1.0
                 COLOUR RED 1.0 GREEN 1.0 BLUE 1.0]
        { white to grey }
        [0.8 1.001 COLOUR RED 1.0 GREEN 1.0 BLUE 1.0
                   COLOUR RED 0.8 GREEN 0.8 BLUE 0.8]
      END_COLOUR_MAP
      SCALE <0.4  0.08  0.4>
    END_TEXTURE
  END_OBJECT
```

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.

## 5.7 Walk-through Wrap-up

In this walk-through, I've only tried to show the highlights of the raytracer without getting into all possible options and features. To get all of those, you'll have to read through the following section on the Scene Description Language. Hopefully it will be fairly straight forward now that you have a feel for the language and how it works. Hopefully you'll find that the textual interface provided by DKBTrace isn't quite as scary as people think at first.

# Chapter 6

# The Scene Description Language

The Scene Description Language allows the user to describe the world in a readable and convenient way. This chapter is effectively a reference manual for DKBTrace.

The language delimits comments by the left and right braces ("{" and "}"). Nested comments are allowed, but no sane person uses them anyway, right? The language allows include files to be specified by placing the line:

```
INCLUDE "filename"
```

at any point in the input file (include files may be nested). The filename must be enclosed in double quotes and may be up to 40 characters long, including the two double-quote (") characters. You may have at most 10 `INCLUDE`'d files per scene trace, whether nested or not.

## 6.1 The Basic Data Types

There are several basic types of data.

### 6.1.1 Floats

Floats are represented by an optional sign (`-`), some digits, an optional decimal point, and more digits. Version 2.10 now supports the "`e`" notation for exponents. The following are valid floats:

```
1.0  -2.0  -4  34  3.4e6 2e-5
```

### 6.1.2 Vectors

Vectors are arrays of three 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 typically 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. "Of what?" you ask? Well, usually it's relative to the center of the "universe" at `<0 0 0>`.

### 6.1.3 Transformations – TRANSLATE, ROTATE, and SCALE

In a few places here and there, vectors are used as a convenient notation but don't represent a point in space. This is the case for the transformations `TRANSLATE`, `ROTATE`, and `SCALE`:

`TRANSLATE` <*x y z*> Move the object *x* units to the right, *y* units up, and *z* units away from us.

`SCALE` <*xs ys zs*> Scale the object by *xs* units in the left/right direction, *ys* units in the up/down direction and *zs* units in the front/back direction.

`ROTATE` <*xr yr zr*> Rotate the object *xr* degrees about the X axis, then *yr* degrees about the Y axis, then *zr* degres about the Z axis (note that the order matters).

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", illustrated in Figure 6.1. If you want to use a right-hand system, as some CAD systems do, the `RIGHT` vector in the `VIEW_POINT` needs to be changed. See the detailed description of the `VIEW_POINT`, and use your right hand for the "Aerobics".

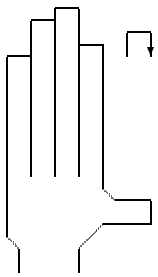

Figure 6.1: Left-hand Coordinate System

### 6.1.4 Colours

A colour consists of a red component, a green component, a blue component, and possibly an alpha component. All four components are floats in the range 0.0 to 1.0. The syntax for Colours is the word `COLOUR` followed by any or all of the `RED`, `GREEN`, `BLUE` or `ALPHA` components in any order. For example:

```
COLOUR  RED 1.0  GREEN 1.0  BLUE 1.0
COLOUR  BLUE 0.56
COLOUR  GREEN 0.45 RED 0.3 BLUE 0.1 ALPHA 0.3
```

`ALPHA` is a transparency indicator. If an object's colour 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.

For those people who spell "Colour" the American way as "Color", the program will also accept `COLOR` as equivalent to `COLOUR` in all instances.

### 6.1.5 Colour Maps

For wood, marble, spotted, agate, granite, and gradient texturing, the user may specify arbitrary colours to use for the texture. This is done by a colour map or "colour spline". When the object is being textured, a number between 0.0 and 1.0 is generated which is then used to form the colour of the point. A Colour map specifies the mapping used to change these numbers into colours. The syntax is as follows:

```
COLOUR_MAP
      [start-value end-value colour1 colour2]
      [start-value end-value colour1 colour2]
```

```
   ...
END_COLOUR_MAP
```

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

## 6.2 More Complex Data Types – DECLARE

The data types used to describe the objects in the world are a bit more difficult to describe. To make this task easier, the program allows users to describe these types in two ways. The first way is to define it from first principles specifying all of the required parameters. The second way allows the user to define an object as a modification of another object (the other object is usually defined from first principles but is much simpler). Here's how it works.

You can use the term DECLARE to declare a type of object with a certain description. The object is not included in the world but is made known to the program that it can be used as a "prototype" for defining other objects by the name given, as this basic example shows:

```
DECLARE Sphere = QUADRIC  { First principles definition }
     <1.0 1.0 1.0>        {   of a sphere               }
     <0.0 0.0 0.0>
     <0.0 0.0 0.0>
     -1.0
END_QUADRIC
```

To then reference the declaration elsewhere in your source file or in another included one, and to actually include the object in the world, you would define the object using the DECLARE'd object's name, like this:

```
OBJECT
     QUADRIC Sphere
          SCALE <20.0 20.0 20.0>
     END_QUADRIC
     TEXTURE
        COLOUR White
        AMBIENT 0.2
        DIFFUSE 0.8
```

```
     END_TEXTURE
END_OBJECT
```

The real power of declaration becomes apparent when you declare several primitive types of objects and then define an object with several component shapes, using either `COMPOSITE` methods or the CSG methods `INTERSECTION`, `UNION`, or `DIFFERENCE`. Also, using the `DECLARE` keyword to pre-define textures can make several objects share a texture without the need for each object to store a duplicate copy of the same texture, for more efficient memory usage. For example:

```
DECLARE Dull TEXTURE  { A Basic, Boring Texture }
     AMBIENT 0.3
     DIFFUSE 0.7
END_TEXTURE
OBJECT         { A Hot dog in a Hamburger Bun (?) }
     UNION
          QUADRIC Sphere
               SCALE <20.0, 10.0, 20.0>
       TEXTURE Dull END_TEXTURE
          END_QUADRIC
          QUADRIC Cylinder_X
               SCALE <40.0, 20.0, 20.0>
       TEXTURE Dull END_TEXTURE
          END_QUADRIC
     END_UNION
END_OBJECT
```

Layered textures, new to version 2.10, may also be `DECLARE`'d. The `DECLARE` mechanism keeps looking for successive `TEXTURE` definitions and will layer them onto the same texture until another `DECLARE` (or any other statement except another `TEXTURE`) is encountered in the input data file. For example:

```
DECLARE Cloud_Wood TEXTURE  { This is the cloudy wood }
  Dark_Wood                 {   texture used earlier. }
  TURBULENCE 0.0
  SCALE <0.2 0.2 0.2>
  PHONG 1.0
END_TEXTURE
```

```
TEXTURE            { This is layered onto the wood }
  TURBULENCE 0.5   {   texture just defined.       }
  BOZO
  COLOUR_MAP
    { transparent to transparent }
    [0.0 0.6 COLOUR RED 1.0 GREEN 1.0 BLUE 1.0 ALPHA 1.0
             COLOUR RED 1.0 GREEN 1.0 BLUE 1.0 ALPHA 1.0]
    { transparent to white }
    [0.6 0.8 COLOUR RED 1.0 GREEN 1.0 BLUE 1.0 ALPHA 1.0
             COLOUR RED 1.0 GREEN 1.0 BLUE 1.0]
    { white to grey }
    [0.8 1.001 COLOUR RED 1.0 GREEN 1.0 BLUE 1.0
               COLOUR RED 0.8 GREEN 0.8 BLUE 0.8]
  END_COLOUR_MAP
  SCALE <0.4  0.08  0.4>
END_TEXTURE

DECLARE (etc.) { Ends the layered definition. }
```

## 6.3 Viewpoint

The viewpoint tells the ray tracer the location and orientation of the camera. The viewpoint is described by four vectors – `LOCATION`, `DIRECTION`, `UP`, and `RIGHT`. Location determines where the camera is located. Direction determines the direction that the camera is pointed. Up determines the "up" direction of the camera. Right determines the direction to the right of the camera.

A first principle's declaration of a viewpoint would look like this:

```
VIEWPOINT
     LOCATION < 0.0  0.0  0.0>
     DIRECTION < 0.0  0.0  1.0>
     UP < 0.0  1.0  0.0 >
     RIGHT < 1.0  0.0  0.0>
END_VIEWPOINT
```

This format becomes cumbersome, however, because the vectors must be hand calculated. This is especially difficult when the vectors are not lined up with the X, Y, and Z axes as they are in the above example. To make

it easier to define the viewpoint, you can define one viewpoint, then modify the description. For example,

```
VIEWPOINT
     LOCATION < 0.0  0.0  0.0>
     DIRECTION < 0.0  0.0  1.0>
     UP < 0.0  1.0  0.0 >
     RIGHT < 1.0  0.0  0.0 >
     TRANSLATE < 5.0  3.0  4.0 >
     ROTATE < 30.0  60.0  30.0 >
END_VIEWPOINT
```

In this example, the viewpoint 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.

Unfortunately, even this is somewhat cumbersome. So, in version 2.0 and above, you can now specify two more parameters:

`SKY` *vector*
`LOOK_AT` *vector*

The `SKY` keyword tells the viewpoint where the sky is. It tries to keep the camera's `UP` direction aligned as closely as possible to the sky. The `LOOK_AT` keyword tells the camera to look at a specific point. The camera is rotated as required to point directly at that point. By changing the `SKY` vector, you can twist the camera while still looking at the same point.

One subtle point: the `SKY` direction is not necessarily the same as the `UP` direction. For example, consider the situation shown in Figure 6.2. If

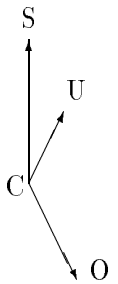


Figure 6.2: The Difference between `SKY` and `UP`

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. UP's like putting an antenna on your camera. If you point the camera downwards, the antenna will no longer point straight up.

The RIGHT vector, as was mentioned previously, controls the aspect ratio of the screen display. It also determines the "handedness" of the coordinate system in use. A normal (left-handed) RIGHT statement would be:

```
RIGHT < 1.3333 0.0 0.0 >
```

To use a right-handed coordinate system, as is popular in some CAD programs and some other ray-tracers, such as Sculpt for the Amiga, use a RIGHT like:

```
RIGHT < -1.3333 0.0 0.0 >
```

Some CAD systems also have the peculiar conception 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 0.0 1.0 >
```

Note that a pinhole camera model is used, so no focus or depth-of-field effects are supported at this time. For more detailed information on the camera model, see the section "How it All Works".

## 6.4 Fog

Version 2.0 of the raytracer includes the ability to render fog. To add fog to a scene, place the following declaration outside of any object definitions:

```
FOG
     COLOUR colour { the fog colour }
     200.0 { the fog distance }
END_FOG
```

The fog to colour ratio is calculated as $1 - e^{-depth/distance}$, so at depth 0, the colour is pure (1.0) with no fog (0.0). At the fog distance, you'll get 63from the object's colour and 37

## 6.5 Shapes

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

### 6.5.1 Quadric Shapes

The most general shape used by this system is called a Quadric Surface. Quadric Surfaces can produce shapes like spheres, cones, cylinders, paraboloids (dish shapes), and hyperboloids (saddle or hourglass shapes).

The easiest way to define these shapes is to include the standard file `shapes.dat` into your program and to transform these shapes (using `TRANSLATE`, `ROTATE`, and `SCALE`) into the ones you want. To be complete, however, I will describe 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
END_QUADRIC
```

defines a surface in three dimensions which satisfies the following equation

$$\mathtt{A}x^2 + \mathtt{B}y^2 + \mathtt{C}z^2 + \mathtt{D}xy + \mathtt{E}xz + \mathtt{F}yz + \mathtt{G}x + \mathtt{H}y + \mathtt{I}z + \mathtt{J} = 0$$

Did you really want to know that? I didn't think so.

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:

$$\begin{aligned} x^2 + y^2 + z^2 - 1 &= 0 - \text{Sphere} \\ x^2 + y^2 - 1 &= 0 - \text{Cylinder along Z axis} \\ x^2 + y^2 - z^2 &= 0 - \text{Cone along Z axis} \end{aligned}$$

### 6.5.2 Quadric surfaces the easy way

Now that doesn't sound so hard, does it? Well, actually, it does. Only the hard-core graphics fanatic would define his objects using the `QUADRIC` definition directly. Even I don't do it that way and I know how it works.[1]

[1] Well, at least I worked it out once or twice.

Fortunately, there is an easier way. The file `shapes.dat` already includes the definitions of many quadric surfaces. They are centered about the origin `< 0 0 0 >` and have a radius of 1. To use them, you can define shapes simply as follows:

```
INCLUDE "colors.dat"
INCLUDE "shapes.dat"   { important for this example }
INCLUDE "textures.dat"

QUADRIC
     Cylinder_X
     SCALE < 50.0  50.0  50.0 >
     ROTATE < 30.0  10.0  45.0 >
     TRANSLATE < 2.0  5.0  6.0 >
END_QUADRIC
```

You may have as many transformation lines (scale, rotate, and translate) as you like in any order. Usually, however, it's easiest 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).

### 6.5.3 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 END_SPHERE
```

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 END_SPHERE
     TEXTURE
        COLOR Blue
        AMBIENT 0.3
        DIFFUSE 0.7
```

```
     END_TEXTURE
END_OBJECT

OBJECT
     SPHERE  < 0.0 0.0 0.0 > 1.0
          TRANSLATE <0.0  25.0  0.0>
          SCALE <10.0  10.0  10.0>
     END_SPHERE
     TEXTURE
        COLOR Blue
        AMBIENT 0.3
        DIFFUSE 0.7
     END_TEXTURE
END_OBJECT
```

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 scaled quadric "Sphere" shape instead, as it can be arbitrarily scaled in any dimension.

### 6.5.4 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:

```
{ plane in the X-Z axes 10 units below the origin. }
PLANE <0.0  1.0  0.0> -10.0 END_PLANE
{ plane in the X-Z axes 10 units above the origin. }
PLANE <0.0  1.0  0.0>  10.0 END_PLANE
{ plane in the X-Y axes 10 units behind the origin.}
PLANE <0.0  0.0  1.0>  -10.0 END_PLANE
```

### 6.5.5 Triangles

In order to make more complex objects than the class of quadrics will permit, a new primitive shape for triangles has been added. 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  < 0.0   20.0  0.0>
          < 20.0  0.0   0.0>
          <-20.0  0.0   0.0>
END_TRIANGLE
```

The smooth shaded triangles use 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
     {      points              surface normals     }
     <  0.0  30.0  0.0 >    <0.0   0.7071   -0.7071>
     < 40.0 -20.0  0.0 >    <0.0   -0.8664  -0.5   >
     <-50.0 -30.0  0.0 >    <0.0   -0.5     -0.8664>
END_SMOOTH_TRIANGLE
```

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.

**Note 2:** To be honest, I don't expect mere mortals to work out the surface normals of triangles. Ideally, you'd have another program generate them for you. See the section on "Common Questions and Answers" for details on how this might be done.

### 6.5.6 Quartic Surfaces

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):

$$\begin{array}{ccccccccccc}
a_{00}x^4 & + & a_{01}x^3y & + & a_{02}x^3z & + & a_{03}x^3 & + & a_{04}x^2y^2 & + \\
a_{05}x^2yz & + & a_{06}x^2y & + & a_{07}x^2z^2 & + & a_{08}x^2z & + & a_{09}x^2 & + \\
a_{10}xy^3 & + & a_{11}xy^2z & + & a_{12}xy^2 & + & a_{13}xyz^2 & + & a_{14}xyz & + \\
a_{15}xy & + & a_{16}xz^3 & + & a_{17}xz^2 & + & a_{18}xz & + & a_{19}x & + \\
a_{20}y^4 & + & a_{21}y^{3z} & + & a_{22}y^3 & + & a_{23}y^2z^2 & + & a_{24}y^2z & + \\
a_{25}y^2 & + & a_{26}yz^3 & + & a_{27}yz^2 & + & a_{28}yz & + & a_{29}y & + \\
a_{30}z^4 & + & a_{31}z^3 & + & a_{32}z^2 & + & a_{33}z & + & a_{34} & = 0
\end{array}$$

To declare a quartic surface requires that each of the coefficients ($a_{00} \ldots a_{34}$) 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 with 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 >
 END_QUARTIC
 BOUNDED_BY
  SPHERE <0 0 0> 10 END_SPHERE
 END_BOUND
END_OBJECT
```

The code to calculate Ray-Surface intersections for quartics is somewhat slower than that for intersections with quadric surfaces. Benchmarks on a stock 8Mhz 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, I know for a fact that there are bad combinations of surface

equations and lighting orientations that cause math errors (crash). If this happens to you, as the joke goes, "then don't do that."

Here are some surfaces to get you started. A Torus can be represented by the equation:

$$x^4 + y^4 + z^4 + 2x^2y^2 + 2x^2z^2 + 2y^2z^2 - 2(r_0^2 + r_1^2)x^2 + 2(r_0^2 - r_1^2)y^2 - 2(r_0^2 - r_1^2)z^2 + (r_0^2 - r_1^2)^2 = 0$$

Where $r_0$ is the "major" radius of the torus – the distance from the hole of the donut to the middle of the ring of the donut, and $r_1$ is the "minor" radius of the torus – the distance from the middle of the ring of the donut to the outer surface (see Figure 6.3). Described another way, a torus is a circle

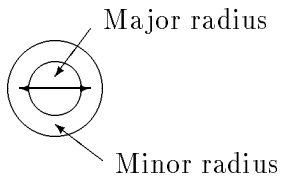


Figure 6.3: Radii of a torus

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. 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:

$$\begin{array}{lllllll} a_{00} & = & a_{20} & = & a_{30} & = & 1 \\ a_{04} & = & a_{07} & = & a_{23} & = & 2 \\ a_{09} & = & a_{32} & & & = & -2(r_0^2 + r_1^2) \\ a_{25} & & & & & = & 2(r_0^2 - r_1^2) \\ a_{34} & & & & & = & 2(r_0^2 - r_1^2)^2 \end{array}$$

The torus can then be rotated and translated into position once its shape has been defined. See the file `TORUS.DAT` for examples.

A generalization of the lemniscate of Gerono can be represented by the equation:

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

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

$$\begin{array}{lllllll} a_{00} & = & a_{25} & = & a_{32} & = & 1 \\ a_{09} & & & & & = & -1 \end{array}$$

See the example file `LEMNISCA.DAT` for a more complete example.

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

$$-bc^2x^4 - ac^2x^3 + y^2 + z^2 = 0$$

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

$$\begin{array}{lllll} a_{00} & & & = & 4 \\ a_{03} & & & = & -4 \\ a_{25} & = & a_{32} & = & 1 \end{array}$$

See the file `PIRIFORM.DAT` 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 inclied, 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.

### 6.5.7 Constructive Solid Geometry

This ray tracer supports Constructive Solid Geometry (CSG) in order to make the object definition abilities more powerful. Constructive Solid Geometry allows you to define shapes which are the union, intersection, or difference of other shapes. Unions superimpose the two shapes. This has the same effect as defining two separate objects, but is simpler to create and/or manipulate. Intersections define the space where the two 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
     INTERSECTION
          QUADRIC
               ...
          END_QUADRIC
```

```
            QUADRIC
                 ...
            END_QUADRIC

            QUADRIC
                 ...
            END_QUADRIC
        END_INTERSECTION
        ...
    END_OBJECT
```

`UNION` or `DIFFERENCE` may be used instead of `INTERSECTION`. 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 (in reverse order from version 1.2 `DIFFERENCE`'s).

Constructive solid geometry shapes may be translated, rotated, or scaled in the same way as a Quadric surface. The quadric surfaces making up the CSG object may be individually translated, rotated, and scaled as well.

When using CSG, it is often useful to invert an shape so that it's inside-out. The `INVERSE` keyword can be used to do this for any `SPHERE`, `PLANE`, or `QUADRIC`. When `INVERSE` is used, the "inside" of the object is flipped to become the "outside". For Planes, "inside" is defined to be in the opposite direction from the "normal" or "up" direction.

Note that performing an `INTERSECTION` between an 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.

## 6.6 Objects

There is more to defining an object than just its shape. You must tell the ray tracer about the properties of the object like surface colour, alpha, reflectivity, refractivity, the index of refraction, and so on. This may be done by specifying it in the shape or the object. Generally, an `OBJECT` definition will contain two pieces of information about an object: First, what shape it is, and second, what it's looks like (on the surface and throughout). A typical object definition looks something like this:

```
    OBJECT
```

```
        QUADRIC Sphere
             TRANSLATE < 40.0 40.0 60.0 >
        END_QUADRIC

        TEXTURE
           AMBIENT  0.3
           DIFFUSE   0.7
           REFLECTION  0.3
           REFRACTION  0.3
           IOR 1.05
           COLOUR RED 1.0 GREEN 1.0 BLUE 1.0 ALPHA 0.5
        END_TEXTURE
    END_OBJECT
```

The following keywords may be used when defining objects:

### 6.6.1 COLOUR

When used on an `OBJECT` (i.e., not inside a `TEXTURE` block), the `COLOUR` keyword determines what colour to use for a low quality rendering when `TEXTURE`'s are ignored. See the `+q` option for details on setting the low-quality option. Examples:

```
COLOUR RED 1.0  BLUE 0.4
```

or

```
DECLARE Yellow = COLOUR RED 1.0 GREEN 1.0
     ...
COLOUR Yellow
```

### 6.6.2 TRANSLATE, ROTATE, and SCALE

Objects 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` before a `TEXTURE`, the texture will *not* be transformed. If the `SCALE`, `TRANSLATE`, or `ROTATE` is after the `TEXTURE`, the texture will be transformed.

### 6.6.3 LIGHT_SOURCE

If the LIGHT_SOURCE keyword is used in the definition of an object, then the object is included in the list of light sources. It can light objects and produce shadows. (You should also specify the COLOUR of the light source, it will usually be "White"). Light sources have a peculiar restriction: The light source *must* be TRANSLATE'd to its final position in the scene, so the normal way to define a light source is a sphere or quadric centered about the origin, then TRANSLATE'd to where desired in the final scene. For example:

```
OBJECT
     { we use a sphere, could use a quartic, too }
     SPHERE <0.0  0.0  0.0> 2.0 END_SPHERE
     TRANSLATE <100.0  120.0  40.0>

     LIGHT_SOURCE
     COLOUR White     { colour of the light emitted }
     TEXTURE
        COLOUR White  { surface colour of the sphere }
        AMBIENT 1.0
        DIFFUSE 0.0
     END_TEXTURE
END_OBJECT
```

**Note:** You *must* specify the colour of the light outside the TEXTURE block. This allows the renderer to quickly determine the colour of the light source without having to plow through the textures. Any colour information inside the TEXTURE block is used to render the light source object itself if it is visible in the scene. The subtle difference between the actual sphere object and the light ray emanation point (the center of the sphere) is why LIGHT_SOURCE's must be defined at <0,0,0> then TRANSLATE'd to where you want them. It ties together and TRANSLATE's both the object itself and the light ray source point to the specified point in the scene. Usually, light sources have an AMBIENT value of 1.0 and a DIFFUSE of 0.0, but this is not a hard and fast rule.

## 6.7 Texture

The texture feature is an experiment into functionally based modelling. This feature allows you to assign colouring schemes to objects. Many procedural surface textures are provided, and by using different colour maps with them, nearly infinite permutations are possible. For example, you can make some object look like wood or marble, etc. In DKBTrace, any parameter that changes the appearance of the surface *must* be put into a `TEXTURE` block.

The basic `TEXTURE` syntax is as follows:

```
TEXTURE
     0.05
     WOOD
     TURBULENCE 0.2
     TRANSLATE < 1.0 2.0 3.0 >
     ROTATE < 0.0 10.0 40.0 >
     SCALE < 10.0 10.0 10.0 >
END_TEXTURE
```

Transformations are optional. They allow you to transform the texture independent of the object itself. If you are doing animation, then the transformations should be the same as the object transformations so that the texture follows the object through 3-D space.

The floating-point value given immediately following the texture keyword is an optional "texture randomness" value, which causes a minor random scattering of calculated colour values and produces a sort of "dithered" appearance. Note this is **bad, bad, bad** for animations!! This is the *only* "truly random" thing in all of DKB, and will produce a most annoying flicker of flying pixels on any textures animated with a "randomness" value used.

Instead of using `WOOD`, you may use `MARBLE`, `BOZO`, `CHECKER`, or a handful of other interesting textures. The `WOOD` and `MARBLE` textures are perturbed by a turbulence function. This makes them look more random and irregular than they would normally appear. The amount of turbulence can be changed by the `TURBULENCE` keyword followed by a number. Values from 0.1 to 0.3 seem to give the best results. The default is 0.0, or no turbulence.

Note some of the textures given are coloration textures, such as `MARBLE`, `WOOD`, `CHECKER`, `GRANITE`, and `AGATE`. These work with colour maps, and have default "colour maps" that they resort to if none are given. The rest of the textures available are "surface perturbation" textures, and do not

directly affect the colour of the object, but rather the surface's apparent orientation in space. Examples of these are `WAVES`, `RIPPLES`, `DENTS`, `BUMPS`, and `WRINKLES`. Note that any given texture may include up to two actual textures, one coloration and one surface perturbation choice per texture. This would allow rippled wood, or dented granite combinations, etc., but, keep in mind that any texture transformations applied to one texture (i.e. `TRANSLATE` or `SCALE`) will also transform the other one in the same fashion.

As of version 2.10, it is possible to create layered textures. If you use more that one texture block, the raytracer will compute the colour of the last texture and if there's any transparency in the colour (i.e., any alpha), it will mix in some of the colour from the underlying textures.

**Important Note:** As of version 2.10, the keywords in this following section *cannot* be used outside of a `TEXTURE`-`END_TEXTURE` structure. This is a change in the input language from prior versions.

### 6.7.1 Object Surface Lighting Characteristics

The following object surface lighting characteristics are available. Each keyword should be followed by a value in the specified range.

#### 6.7.1.1 `AMBIENT`

Ambient light is light that is scattered everywhere in the room. An object lit only by ambient light will appear to have the same brightness over the entire surface. The default value is very little ambient light (0.3). The value can range from 0.0 to 1.0.

#### 6.7.1.2 `DIFFUSE`

Diffuse light is light coming from a light source that is scattered in all directions. An object lit only by diffuse light looks like a rubber ball with a spot light shining on it. The value can range from 0.0 to 1.0. By default, there is mostly diffuse lighting (0.7).

#### 6.7.1.3 `BRILLIANCE`

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. This is best used in concert with either the `SPECULAR` or `PHONG` highlighting.

#### 6.7.1.4 `REFLECTION`

By setting the reflection value to be non-zero, you can give the object a mirrored finish. It will reflect all other objects in the room. The value can range from 0.0 to 1.0. By default there is no reflection.

#### 6.7.1.5 `REFRACTION`

By setting the refraction value to be non-zero, the object is made transparent. Light will be refracted through the object like a lens. The value can be set between 0.0 and 1.0. There is no refraction by default.

**Note 1:** New for 2.10: In order to refraction to work properly, you must have some `ALPHA` component in the surface colour. In the places where the `ALPHA` is high, the refracted light can get through. In places where the `ALPHA` is low, the refracted light is suppressed. This is a change in the input language from prior versions.

**Note 2:** The refracted light is filtered by (takes on) the surface colour.

**Note 3:** In layered textures, the `REFRACTION` and `IOR` keywords *must* be in the last texture, otherwise they will not take effect.

**Note 4:** If a texture has an `ALPHA` component and no value for `REFRACTION` was supplied, the renderer will simply transmit the ray through the surface with no bending.

#### 6.7.1.6 `IOR`

If the object is refracting light, then the `IOR` or "index of refraction" should be set. This determines how dense the object is. 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.

#### 6.7.1.7 PHONG

Controls the amount of Phong Specular Reflection highlighting on the object. Causes bright shiny spots on the object, the colour of the light source that is being reflected. The size of the spot is defined by the value given for PHONGSIZE below. PHONG's value is typically from 0.0 to 1.0, where 1.0 causes complete saturation of the object's colour to the light source's colour at the brightest area (center) of the highlight. There is no PHONG highlighting given by default.

#### 6.7.1.8 PHONGSIZE

Controls the size of the PHONG Highlight on the object, sort of an arbitrary "glossiness" factor. Values range from 1.0 (Very Dull) to 100 (Highly Polished). Default PHONGSIZE is 40 (plastic?) if not specified. This 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.

#### 6.7.1.9 SPECULAR

Very similar to PHONG Specular Highlighting, but a better model is used for determining light ray/object intersection, so a more credible spreading of the highlights occur near the object horizons, supposedly. PHONG is thus included for mostly academic reasons, but try them both and you decide which you like better. This effect is most obvious for light sources behind objects. The size of the spot is defined by the value given for ROUGHNESS below. Like PHONG, SPECULAR values are typically from 0.0 to 1.0 for full saturation. Default is no SPECULAR highlighting.

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.

#### 6.7.1.10 ROUGHNESS

Controls the size of the SPECULAR Highlight on the object, relative to the object's "roughness". Values range from 1.0 (Very Rough) to 0.001 (Very

Smooth). The default value, if not specified, is 0.05 (Plastic?). The roughness or average directional distribution of the microfacets facing in the same direction as the perpendicular surface "normal" cause the most notable reflection of the highlight to the observer.

#### 6.7.1.11 METALLIC

This keyword indicates that the colour of the specular and phong hightlights will be the surface colour instead of the lightsource colour. This creates a metallic appearance. When using this feature, you should set AMBIENT to about 0.5 and set DIFFUSE to 0.0. This keyword does not take a value.

### 6.7.2 Surface Colouring Textures

The first type of advanced procedural textures provided by DKBTrace are surface coloring textures. These affect the colour of an object.

#### 6.7.2.1 CHECKER

CHECKER texturing gives a checker-board appearance. This option works best on planes. When using the CHECKER texturing, you must specify two colours immediately following the word CHECKER. These colours are the colours of alternate squares in the checker pattern. The default orientation of the CHECKER texture is on an X-Z plane (good for ground work, etc.) but to use it on an object which has mostly X-Y orientation (such as a sphere, for instance), you must ROTATE the texture. To rotate the CHECKER texture onto an X-Y plane:

```
TEXTURE
     CHECKER COLOUR White COLOUR Red
     SCALE <10.0 10.0 10.0>
     ROTATE <-90.0 0.0 0.0>   { now in the X-Y plane }
END_TEXTURE
```

#### 6.7.2.2 CHECKER_TEXTURE

I've had many requests for a checker pattern to allow you to alternate between wood and marble or any other two textures. So, in version 2.10, I've added a texture called CHECKER_TEXTURE that takes two textures instead of two colours.

In order to support layered textures, I've made the syntax a bit more verbose than it would be otherwise. The syntax is:

```
TEXTURE
   CHECKER_TEXTURE
      TEXTURE ... texture ... END_TEXTURE
      TEXTURE ... optional layers ... END_TEXTURE
   TILE2
      TEXTURE ... second texture ... END_TEXTURE
      TEXTURE ... optional layers ... END_TEXTURE
   END_CHECKER_TEXTURE
   AMBIENT ...
   DIFFUSE ...
END_TEXTURE
```

Note that the textures in `CHECKER_TEXTURE` only use the surface colouring texture information. Information about `AMBIENT`, `DIFFUSE`, `REFLECTION`, etc. and surface normal information (`WAVES`, `RIPPLES`) are ignored inside the `CHECKER_TEXTURE`. (Hey, what do you want for a 10 minute change?)

#### 6.7.2.3 BOZO

`BOZO` texture 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.

As mentioned above, for coloration textures such as `WOOD`, `MARBLE`, and `BOZO`, etc., you may change the colouring scheme by using a colour map. This map allows you to convert numbers from 0.0 to 1.0 (which are generated by the ray tracer) into ranges of colours. For example, the default `BOZO` colouring can be specified by:

```
TEXTURE
     BOZO
     COLOUR_MAP
          [0.0 0.4 COLOUR White COLOUR White]
          [0.4 0.6 COLOUR Green COLOUR Green]
          [0.6 0.8 COLOUR Blue COLOUR Blue]
          [0.8 1.0 COLOUR Red COLOUR Red]
```

```
     END_COLOUR_MAP
END_TEXTURE
```

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

```
TEXTURE
     BOZO
     TURBULENCE 1.0    { A blustery day.  For a calmer }
     COLOUR_MAP        {   one, try 0.2.               }
          { blue to blue }
          [0.0 0.5  COLOUR RED 0.5 GREEN 0.5 BLUE 1.0
                    COLOUR RED 0.5 GREEN 0.5 BLUE 1.0]
          { blue to white }
          [0.5 0.6  COLOUR RED 0.5 GREEN 0.5 BLUE 1.0
                    COLOUR RED 1.0 GREEN 1.0 BLUE 1.0]
          { white to grey }
          [0.6 1.001 COLOUR RED 1.0 GREEN 1.0 BLUE 1.0
                     COLOUR RED 0.5 GREEN 0.5 BLUE 0.5]
     END_COLOUR_MAP
     SCALE <800.0 800.0 800.0>
     TRANSLATE <200.0 400.0 100.0>
END_TEXTURE
```

The color map above indicates that for small values of texture, 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). Check out `sunset.dat` for a really neat (but slow) sky pattern using `ALPHA`.

#### 6.7.2.4 SPOTTED

Spotted texture is a sort of swirled random spotting of the colour 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. Should work with colour maps. With small scaling values, looks like masonry or concrete.

#### 6.7.2.5 AGATE

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

#### 6.7.2.6 GRADIENT

This is a specialized texture that uses approximate local coordinates of an object to control colour map gradients. This texture *only* works with colour maps (one *must* be given!) and has a special <X,Y,Z> triple 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 colour map", along the Y axis. Values given other than 0.0 are taken as 1.0.

For smooth repeating gradients, you should use a "circular" colour map, that is, one in which the first colour value (0.0) is the same as the last one (1.001) 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 that 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 >
     COLOUR_MAP
          [0.00 0.25  COLOUR RED 1.0 GREEN 0.0 BLUE 0.0
                      COLOUR RED 1.0 GREEN 0.0 BLUE 0.0]
          [0.25 0.75  COLOUR RED 1.0 GREEN 1.0 BLUE 1.0
                      COLOUR RED 1.0 GREEN 1.0 BLUE 1.0]
          [0.75 1.001 COLOUR RED 1.0 GREEN 0.0 BLUE 0.0
                      COLOUR RED 1.0 GREEN 0.0 BLUE 0.0]
     END_COLOUR_MAP
     SCALE <30.0 30.0 30.0>
     TRANSLATE <30.0 -30.0 0.0>
END_TEXTURE
```

You may also specify a TURBULENCE value with the gradient to give a more irregular colour gradient. This may help to do neat things like fire or corona effects.

#### 6.7.2.7 GRANITE

This texture uses a simple $1/f$ fractal noise function to give a pretty darn good grey granite texture. Typically used with small scaling values (2.0 to 5.0). Also looks good with a little dithering (texture randomness). Should work with colour maps, so try your hand at pink granite or alabaster!

### 6.7.3 Surface Perturbation Textures

The other type of advanced procedural texture provided by DKBTrace are surface perturbation textures. These affect the 3D appearance of an object, rather than its colour directly. A surface perturbation texture can be used in conjunction with the above coloration textures.

#### 6.7.3.1 RIPPLES

RIPPLES makes 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
     TRANSLATE < 1.0 2.0 3.0 >
     ROTATE < 0.0 10.0 40.0 >
     SCALE < 10.0 10.0 10.0 >
END_TEXTURE
```

In this case, the WOOD, MARBLE, or BOZO, etc. keywords are optional. If a different colouring is specified (WOOD, MARBLE, or BOZO), then the COLOUR parameter is ignored (except for light sources where it gives the light colour or when rendering with a low -q option).

#### 6.7.3.2 WAVES

Another option that you may want to experiment with is called WAVES. 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. (I haven't done much testing on WAVES, so I can't guarantee that it works very well).

#### 6.7.3.3 PHASE

Both WAVES and RIPPLES respond to a texturing option 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.

#### 6.7.3.4 FREQUENCY

The WAVES and RIPPLES textures also respond to a keyword called FREQUENCY. If you increase the frequency of the waves, they get closer together. If you decrease it, they get farther apart.

#### 6.7.3.5 BUMPS

Approximately the same turbulence function as SPOTTED, but uses the derived value to perturb the surface normal. 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 (Extremely Bumpy). Values beyond 1.0 may do weird things.

#### 6.7.3.6 DENTS

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. Scale the texture to make the pitting more or less frequent.

#### 6.7.3.7 WRINKLES

This is sort of a 3-D (normal perturbing) GRANITE. It uses a similar $1/f$ fractal noise function to perturb the surface normal in 3-D space. With ALPHA values of greater than 0.0, 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.

### 6.7.4 IMAGEMAP

This is a very special coloration texture that allows you to import a bit-mapped file in "dump" format (the format output by the ray-tracer), IFF format or Compuserve GIF format and map that bitmap onto an object. In the texture of an object, you must give the `IMAGEMAP` keyword, the format, and a file name. The syntax is:

`IMAGEMAP` *optional-gradient* `DUMP "`*filename*`"` *optional*-`ONCE`
`IMAGEMAP` *optional-gradient* `IFF "`*filename*`"` *optional*-`ONCE`
`IMAGEMAP` *optional-gradient* `GIF "`*filename*`"` *optional*-`ONCE`

The texture will then be mapped onto the object as a repeating pattern. The optional `ONCE` keyword places only one image onto the object instead of an infinitely repeating tiled pattern. When `ONCE` is used, the colour outside the mapped texture is set to transparent. You can use the layered textures to place other textures or colours below the image.

In version 2.10 and up, you can specify the `ALPHA` values for the colour registers of IFF or GIF pictures (at least for the modes that use colourmaps). You can do this by putting the keyword `ALPHA` immediately following the filename followed by the register value and transparency. If the `ALL` keyword is used instead of a register number, then all colours in that colourmap get that alpha value. For example:

```
IMAGEMAP <1.0 -1.0 0.0> IFF "mypic.iff"
    ALPHA ALL 0.0
    ONCE
```

or

```
IMAGEMAP <1.0 -1.0 0.0> IFF "mypic.iff"
    ALPHA 0   0.5
    ALPHA 1   1.0
    ALPHA 2   0.3
    ONCE
```

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 `IMAGEMAP`. 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:

```
INCLUDE "shapes.data"
OBJECT
     QUADRIC Plane_XY END_QUADRIC
     TRANSLATE <0.0  -20.0  0.0>
     TEXTURE
          { make this texture use the x and z axes
            for the mapping.                      }
          IMAGEMAP <1.0  0.0  -1.0> GIF "image.gif"
          SCALE <40.0 40.0 40.0>
     END_TEXTURE
END_OBJECT
```

Filenames specified in the `IMAGEMAP` 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 imagemaps (`.dis`, `.gif` or `.iff`) files in a "textures" subdirectory, and giving an `-l` option on the command line to where your library of imagemaps are.

When I was bored with nothing to do, I decided that it would be neat to have turbulent texture maps. So, I said – "What the hell!" You can specify `TURBULENCE` with texture maps and it will perturb the image. It may give some bizarre results. Is this useful? I dunno. It was easy to do so I did it. Try it out and see what you get.

## 6.8 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
```

```
        ...
    END_OBJECT
    OBJECT
        ...
    END_OBJECT
    ...
    SCALE < 2.0 2.0 2.0 >
    ROTATE < 30.0 45.0 160.0 >
    TRANSLATE < 100.0 20.0 40.0 >
END_COMPOSITE
```

Composite objects can contain other composite objects as well as regular objects. Composite objects cannot be light sources (although any number of their components can). This is because it is nearly impossible to determine the true "center" of the composite object, and our light sources are pinpoint ray sources from the center of the light source object, wherever that may be.

## 6.9 Bounding Shapes

Let's face it. This program is no speed demon. You can save yourself a lot of time, however, if you use bounding shapes around any complex objects. Bounding shapes tell the ray tracer that the object is totally enclosed by a simple shape. When tracing rays, the ray is first tested against the simple bounding shape. If it strikes the bounding shape, then the ray is further tested against the more complicated object inside.

**Note:** Don't use bounding shapes instead of CSG to clip objects. You will not get the result you want. For the raytracer to work properly, you must have the entire object inside the bounding shape.

To use bounding shapes, you simply include the following lines into the declaration of your `OBJECT` or `COMPOSITE`:

```
BOUNDED_BY
    shape
END_BOUND
```

An example of a Bounding Shape is:

```
OBJECT
     INTERSECTION
          SPHERE <0.0 0.0 0.0> 2.0 END_SPHERE
          PLANE <0.0 1.0 0.0> 0.0 END_PLANE
          PLANE <1.0 0.0 0.0> 0.0 END_PLANE
     END_INTERSECTION
     BOUNDED_BY
          SPHERE <0.0 0.0 0.0> 2.0 END_SPHERE
     END_BOUND
END_OBJECT
```

The best bounding shape is a `SPHERE` since this shape is highly optimized. Any shape may be used, however, if more convenient.

# Chapter 7

# Displaying the Images

When the ray tracer draws the picture on the screen, it doesn't make good choices for the colour registers. Without knowing all the needed colours ahead of time, only approximate guesses can be made. Usually, a post-processor is really needed to view the final image correctly.

## 7.1 Amiga Systems

A post-processor has been provided for the Amiga which scans the picture and chooses the best possible colour registers. It then redisplays the picture. For the Amiga, `DumpToIFF` can optionally save this picture in IFF format. The syntax for the `DumpToIFF` post-processor is:

```
DumpToIFF filename
```

where the filename is the one given in the `-o` parameter of the ray tracer. If you didn't specify the `-o` option, then use:

```
DumpToIFF data.dis
```

If you want to save to an IFF file, then put the name of the IFF file after the name of the data file:

```
DumpToIFF data.dis picture
```

This will create a file called `picture` which contains the IFF image.

An alternative approach is to buy the commercial package called *The Art Department* from ASDG. You can then use the `+fr` option of the raytracer to produce raw files which can be read in to *TAD* using Sculpt mode. You

can also render using `+fd` to produce a dump format file, and use `d2iff` to convert this to a 24-bit IFF image to load into *TAD*.

## 7.2 IBM Systems

For the IBM, you will probably want to use the `+ft` option (default if `+f` is given) and write the image out in Targa-24 format. If you have a Targa or compatible display adapter, you may view the picture in the full 16 million colors (that's why they still cost the big bucks, but Hercules and Everex, notably, are introducing their lower-priced SVGA-compatible 24-bit color display systems for the IBM PC and compatibles). If you don't have one of these, there are several different post-processing programs available to convert the TARGA true-color image into a more suitable color-mapped image (such as `.GIF`). If you have a VGA/MCGA or SVGA adapter capable of 320x200 by 256 colors or better, then you may use the +d option which will display the image as it generates using only approximate screen colors. The `+d` option will Autodetect the type of display adapter card you have and briefly say what kind it found before displaying the picture. If you say `+d`*x* where *x* is one of the predefined IBM (S)VGA display adapter types, no hardware test is performed, so if you don't have that type of (S)VGA card, *don't* use that particular `+d`*x* option!

When displaying the image to screen, a HSV conversion method is used (hue, saturation, value). This is a convenient way of translating colors from a "true color" format (16 million) down a "colour mapped" format of something reasonable (like 256), while still approximating the color as closely as the available display hardware permits. As mentioned previously, the tracer has no way of knowing which colors will be finally used in the image, nor can it deal properly with all of the colors which will be generated (up to 16M), so only 4 shades each of 64 possible hues are mapped into the palette DAC, as well as black, white, and two grey levels. The advantage a post-processor has in choosing mapped colors is that it can throw away all the unused colors in the palette map, and thereby free up some space for making better gradient shades of the colors that are actually used.

There are several available image processing programs that can do this, a public domain one that is very good is `PICLAB`, by the Stone Soup Group (the folks who brought you `FRACTINT`). The procedure is to load the TARGA file, then use the `MAKEPAL` command to generate a 256 color map which is the histogram-weighted average of the most-used colors in the image (You

could also PLOAD a palette file from FRACTINT or one you previously had saved using PSAVE). You then MAP the palette onto the image one of two ways:

1. If the DITHER variable is OFF, a nearest-match color-fit is used, which can sometimes produce unwanted "banding" of colored regions (called false contours).

2. If the DITHER variable is ON, then a standard dither is used to determine final color values. This is much better at blending the color bands, but can produce noise in reflections and make mirrors appear dirty or imperfect.

Then you would typically SHOW the image or GSAVE it into GIF format. While the picture is still in the unmapped form (TARGA, etc.) you can perform a variety of advanced image processing transformations and conversions, so save the .TGA or .RAW files you make (in case you ever get a TARGA card, or give them to a friend who has one!).

A commercial product that also does a good job of nearest-match color-fit is the CONVERT utility of *AutoDesk Animator*. However, the dither effect is not as good as that of PICLAB. To convert the file in *AA*'s CONVERT, you LOAD TARGA, then in the CONVERT menu, go to the SCALE function and just hit RENDER. Click on the DITHER (lights up with an asterisk when on) if you want it to use it's dithering. CONVERT will scale (if asked to) and then do a histogram of total used colors like PICLAB, but then makes 7 passes on the color map and image to determine shading thresholds of each hue. This nearly eliminates the color banding (false contours) in a lot of cases without resorting to good 'ol dithering. By now you must get the feeling DITHER is a 4-letter word. If you have a low-resolution display, it is. If you have too few colors, however, it can be a saving grace. At resolutions of 640x400 or higher the "spray paint" effect of dithering and anti-aliasing is much less noticeable, and effects a much smoother blending appearance.

A new package to show up in the public domain/shareware circles for the IBM is something called *Image Alchemy*, by Handmade Software. It will convert Targa format to GIF files and do a decent job of palette selection and dithering. To use it simply say

```
ALCHEMY file.tga file.gif -g -8 -c256
```

It also features a quick-and-dirty display mode where it uses a standardized palette in much the same way DKB's +d option does, but offers dithering of

the image while using the pre-defined palette, for a somewhat better quick display.

## 7.3 Unix Systems

I don't have many details on Unix systems, but I hear that the FBM utilities work well to convert the Dump format files into various formats of images. For people with access to anonymous FTP over USENET, the FBM utilities are available from `nl.cs.emu.edu` (128.2.222.56) in directory `/usr/mlm/ftp`.

# Chapter 8

# DKBTrace Utilities

In many cases, creating data files for DKBTrace is difficult and tedious. To help remedy this problem, I and various other people have developed some utilities to create data files. These utilities are described below.

As well, there are some utilities that perform operations on the image files created by DKBTrace. These utilities convert between various formats and allow you to modify or merge output files together.

I'd like to thank all the people who wrote these utilities and sent them to me. If anybody else comes up with other utilities, please let me know and I'll include them in the distribution.

Some of these utilities are written in BASIC in IBM systems. As such, they are not easily portable from system to system. If anyone wants to convert them to C, let me know and I'll post the C versions.

## 8.1 Data File Creation Utilities

The data creation utilities fall into two categories: Those that convert from some other format into DKB format, and those that generate DKB files using algorithmic techniques. These utilities are described below.

### 8.1.1 SA2DKB

This program converts Sculpt-Animate 3D and 4D data files into DKB format. It currently only supports the basic triangles and textures. It doesn't support smooth triangles (it treats them like normal triangles), light sources, cameras, or floors. (This utility was formerly called `Sculpt2DKB` but the IBM

systems out there kept calling it `SCULPT2D`, then couldn't figure out what a 2D program had to do with raytracing or what the nonexistent Amiga program called `Sculpt-2D` was.

### 8.1.2 DXF2DKB

This utility converts AutoCAD DXF (Drawing eXchange Format) files into DKBTrace format scene description files. It was written by Aaron Collins. It does not support all of the DXF primitives, but will suffice for simple objects and scenes after `EXPLODE`'ing and `DXFOUT`'ing then in *AutoCAD*.

### 8.1.3 ShellGen

`ShellGen` is a BASIC program written by Dan Farmer. It's based on a short code fragment from Clifford Pickover's book *Computers, Pattern, Chaos, and Beauty* (St. Martin's Press). This code fragment was reprinted in *Ray Tracing News*, issue 3.3.

As far as I know, the BASIC program only works on IBM's. It does, however, allow you to change the parameters and see a quick outline of what the result will look like. For those people without IBM's, I've changed the original code fragment to at least output a DKB-format file. No user interface has been provided, however.

### 8.1.4 Twister

`Twister` is a C program written by Drew Wells (CIS `73767,1244`). It creates data files for twisted shapes. The program uses a text interface and prompts the user with a question/answer format.

### 8.1.5 Chem2DKB

`Chem2DKB` is an IBM BASIC program written by Dan Farmer. It takes models generated by the `CHEM.EXE` program written by Larry Puhl.

### 8.1.6 Lissajou

This is an IBM BASIC program written by Dan Farmer. It creates data files for lissajous figures. The basic algorithms were from Clifford Pickover. See *Scientific American*, January 1991 and *Omni*, February 1990 for examples.

## 8.2 Output File Manipulation Utilities

These utilities perform some useful manipulations on the dump format and Targa format output files from DKBTrace. I'd like to thank the people who wrote these utilities and provided them for general distribution.

### 8.2.1 `dump2i24`

Also known as `DumpToIFF24`, this program was written by Helge E. Rasmussen (`her@compel.dk`). It converts the dump format files produced by DKBTrace into 24-bit IFF format files. These files can then be read by a variety of programs including *The Art Department* by ASDG.

### 8.2.2 `catdump`

This utility was written by Ville Saari (`vsaari@niksula.hut.fi`, and copyright by the Ferry Island Pixelboys.) It takes two or more partially rendered files in DKBTrace's dump format and merges them into one file. This is useful for all sorts of things like rendering different parts on different computers and combining the results.

**NOTE:** Be careful if you combine pictures produced on different systems. If the random number generator works differently between the two systems, the textures may look completely different from one another. So long as you use the same executable, you should be fine.

### 8.2.3 `combdump`

This utility was also written by Ville Saari. It takes two images generated with DKBTrace with slightly different viewpoints, and creates one dump-format image file to be viewed with Red-Blue or Red-Green 3D glasses. The program allows you to compensate for the exact filtering characteristics of your glasses to get the best possible result.

### 8.2.4 `dump2mtv`

This is yet another utility written by Ville Saari. This one converts DKBTrace dump format files onto MTV format used by the MTV and RayShade raytracers.

### 8.2.5 dump2raw

The `dump2raw` utility was written by Aaron Collins to convert the dump format output of DKBTrace into three separate files for red, green, and blue. On the IBM, the extensions for these files are `r8`, `g8`, and `b8`. On the other systems, they are `red`, `grn` and `blu`.

Version 2.10 of the raytrace allows you to use the `+fr` option to output raw format files directly without the need for a conversion program like this.

### 8.2.6 halftga

The `halftga` utility (written by Aaron Collins) shrinks a Targa file to exactly half its original size. This file can then be converted into a GIF image and used in an `IMAGE_MAP` statement. For systems with little memory available for imagemaps, this command can be a life-saver.

### 8.2.7 gluetga

This utility (by Aaron Collins) is similar to `catdump` but works for Targa format files. It takes several partially-rendered Targa files and glues them together into one image.

### 8.2.8 tga2dump

This utility was written by Aaron Collins. It converts Targa format 16, 24, and 32 bit images into DKB's dump format for use in image-mapping.

## 8.3 Animation Utilities

One of the most frequent questions I'm asked is whether or not DKBTrace supports animation. The answer is no, not directly. However, I have made some changes to the program to provide frame-to-frame consistency so you can use it for animation if you want to. The problem, then, is creating the data files for each individual frame. That's what this section is all about.

### 8.3.1 RayScene

Although RayScene is not being distributed with this raytracer, I thought I'd at least mention it and tell you where you can get it. RayScene is a program

that creates data files for DKBTrace based on a high-level (higher-level?) description of the motion of the camera and the objects. It was written by Jari Kûhkünen (`hole@tolsun.oulu.fi`) and Panu Hassi (`oldfox@tolsun.oulu.fi`) and is available by anonymous FTP from `tolsun.oulu.fi` (128.214.5.6) in the directory `/pub/rayscene` or from `iear.arts.rpi.edu` in the directory `/pub/graphics/ray/rayscene`. This explanation of RayScene was sent to me by Panu Hassi:

> I've tried animation with DBW before DKBTrace2.0 was released. The procedure was this: First I wrote the first scene file, copied it for `NUMBER_OF_FRAMES` times and then edited some parts of those files to create movement etc. If something went wrong (I accidentally edited wrong value etc), I had to edit all those scene files again to make the changes. Not so nice if there are 100 files to edit...
>
> So a friend of mine, Jari Kûhkünen, and I decided to write RayScene to make that process even a little easier. With RayScene the process goes like this: you create a scene file and mark the places that should be changed with a variable, like:
>
> ```
> BOUNDED_BY
>      SPHERE <0.0 0.0 0.0> #sphere_size# END_SPHERE
> END_BOUND
> ```
>
> Then you create another file where the values for these variables are listed. Rayscene simply creates N scene files inserting current value of each variable to proper place. That's all.
>
> We have included couple of simple utilities that help with creating those variable values, but the original scene files are still created "manually". Still, the results have been really nice. There are several animations for Amiga and PC in `tolsun.oulu.fi`.

# Chapter 9

# How it All Works (or How to Get What You Want)

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

## 9.1 Viewpoints

Viewpoints can be completely defined by four vectors. The `LOCATION` is easy. That's where the camera is. The `DIRECTION` is a vector that starts at the `LOCATION` and points to the center of a window. The `UP` vector starts at the center of the window and points to the center of the top edge. The `RIGHT` vector starts at the center of the window and points to the center of the right edge. These vectors are illustrated in Figure 9.1.

The window is then divided up according to the resolution you specified and rays are fired through the pixels out into the world. For an eye ray, therefore, the equation of the ray is:

$$\texttt{LOCATION} + t(\texttt{DIRECTION} + ((height - y) \cdot \texttt{UP}) + (x \cdot \texttt{RIGHT}))$$

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.

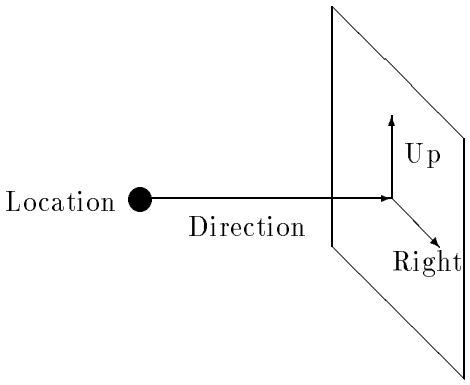


Figure 9.1: Viewpoint definition vectors

This viewpoint 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.

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 me 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.

## 9.2 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 (well, almost), 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.

Fortunately, all ray-object intersections for all shapes in DKBTrace can be solved by a simple quadratic equation. This is why `QUADRIC`s are used in DKBTrace. Solving for things like B-Splines, Bezier Splines, NURBS, Tori, etc. is a lot more complicated. That's why I haven't implemented primitives for these shapes.

## 9.3 Transparency and Refraction

This section gets really complicated because of the way transparency and refraction are implemented. If you don't really care, skip to the next section. If you don't mind slogging through this and getting confused, then read on – you've been warned.

The way that transparency and refraction work has changed slightly from previous versions. Now, transparency and refraction work together instead of separately.

First, let me distinguish between reflected light and filtered light. Suppose you painted a table with patches of colour. You then took some red sand and sprinkled it on top of the various colours. The red sand will tint the colours red, but you will still see some of the original colour. If, instead, you took a sheet of red plexiglass and put it on top of the table, all you could see would be shades of red. That's because the plexiglass filter *only* allows the red colour to show through.

In DKBTrace, the layered textures work like the red sand – the colours on top mix with the underlying colours depending on the density (`ALPHA`) of the distribution.

Refraction, however, works like a filter. The surface colour determines that colours of light are allowed to pass from the inside of the object to the outside, and vice versa. Here are some filter colours:

```
{ a red filter }
RED 1.0  GREEN 0.0  BLUE 0.0   ALPHA 1.0
{ a clear glass }
```

```
RED 1.0  GREEN 1.0  BLUE 1.0   ALPHA 1.0
{ a dark filter - this will appear black }
RED 0.0  GREEN 0.0  BLUE 0.0   ALPHA 1.0
```

Now, consider the following layered textures:

```
TEXTURE COLOUR Green ALPHA 0.6 END_TEXTURE
TEXTURE COLOUR Yellow ALPHA 0.3 REFRACTION 0.5 END_TEXTURE
```

Keep in mind that the last texture is on top.[1] The colour you get is calculated as shown in Figure 9.2. The top texture layer supplies some fraction

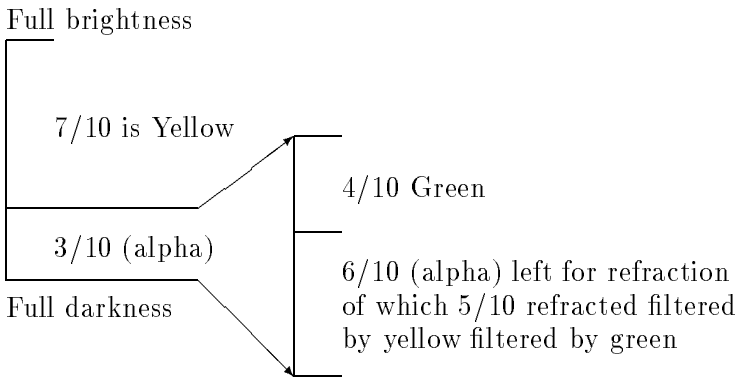


Figure 9.2: Layered color calculation

of the light that reflects off the surface of the object. If the `ALPHA` was nonzero, it allows the lower textures to supply the remainder. If, after all the textures are processed, there's still some fraction left over, it is applied to the light that's refracted through the object.

This algorithm probably doesn't coincide with reality, but neither does the rest of the raytracer, so I'm not terribly concerned about it.

## 9.4 Textures, Noise, and Turbulence

If there's one thing that DKBTrace is known for, it's textures. Here's how they really work. If you want some good reading material, check out "An Image Synthesizer" by Ken Perlin in the SIGGRAPH '84 Conference Proceedings.

[1] In layered textures, only the `REFRACTION` component of the *last* entry has any effect.

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

For example, take the simple block of marble shown in Figure 9.3. If you

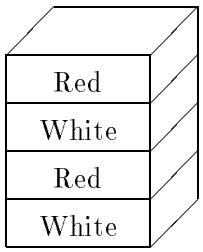


Figure 9.3: Marble texture

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 colours, as shown in Figure 9.4. Cutting a shape out of a piece of wood will tend to

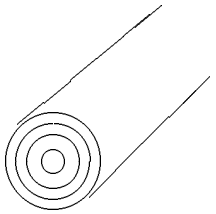

Figure 9.4: Wood texture

give you rings of colour.

Now, this is fine, but the textures are still a bit boring. For the next step, we blend the colours 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 colour 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 procedure is roughly illustrated in Figure 9.5. This is what we use to create

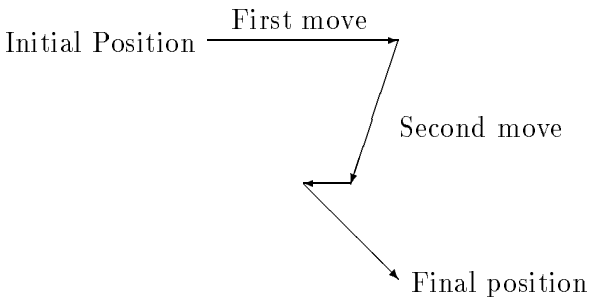


Figure 9.5: Effect of turbulence

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

## 9.5 Layered Textures

As of version 2.10, DKBTrace supports layered textures. Here's how that works. 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 `DECLARE`'d 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 `DECLARE`'d 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 END_SPHERE
   TEXTURE WOOD END_TEXTURE
   TEXTURE MARBLE END_TEXTURE
END_OBJECT
```

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, as shown in Figure 9.6. When the parser sees the first

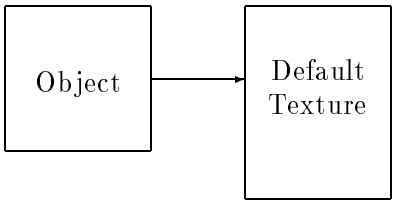


Figure 9.6: Object parsing, step 1

`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, as shown in Figure 9.7. On the next texture, it sees that the texture isn't the default one, so it adds the second texture into the linked list, as shown in Figure 9.8.

Now for a problem. If you want to specify the `REFRACTION` of the texture, the raytracer must first calculate the surface colour. It does this by marching

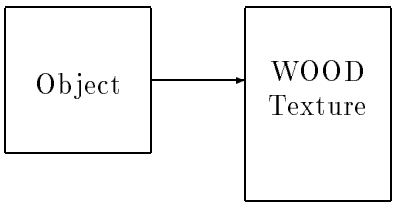


Figure 9.7: Object parsing, step 2

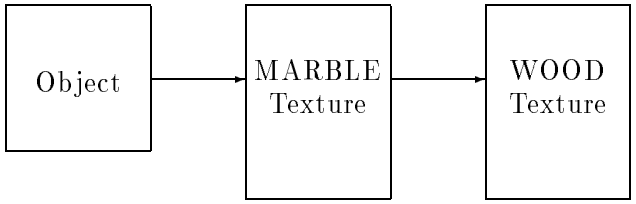


Figure 9.8: Object parsing, step 3

through the texture list and mixing all the colours. When it's finished, it checks the `ALPHA` value of the surface colour 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. I don't see any reason to have refraction values for any other textures in the layer as it applies to the whole object.

## 9.6 Image Mapping

One major problem people have when designing data files for DKBTrace is how to position images onto the desired surfaces. With version 2.10, this problem becomes slightly easier since the image can be mapped onto the object in the object's natural coordinate system. Thereafter, when the object is translated, rotated, or scaled, the image map will follow it.

The image mapping that DKBTrace currently supports is called a "parallel projection" mapping. This technique is simple (that's why I implemented it), 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. Consider the cross section shown in Figure 9.9, which shows an image with colours A, B, C, D, and E being mapped onto three

objects. The raytracer performs a similar operation to map a 2D picture

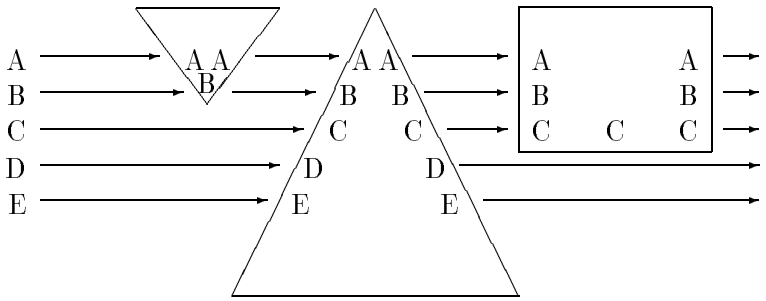


Figure 9.9: Parallel projection

onto a 3D object. Note that objects cannot shadow each other 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 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:

```
IMAGEMAP <1 -1 0> GIF "filename"
```

will map the GIF picture onto the square from `<0 0 0>` to `<1 1 0>` as shown in Figure 9.10. If we reversed the vector, the picture would be transposed:

```
IMAGEMAP <-1 1 0> GIF "filename"
```

produces the result shown in Figure ??. Once the image orientation has been determined, it can be translated, rotated, and scaled as desired to map properly onto the object.

## 9.7 Output File Formats

People always ask me to describe the output file formats of DKBTrace. I received so many requests for this that I decided to put it into the document.

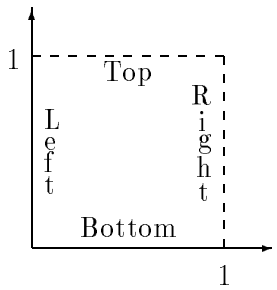


Figure 9.10: Image mapping with vector `<1 -1 0>`

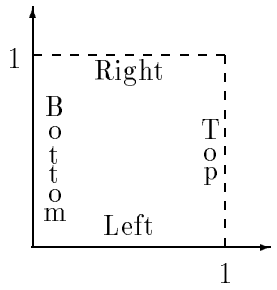


Figure 9.11: Image mapping with vector `<-1 1 0>`

The normal "default" output format is "Dump" format. This is based on QRT format and goes like this, where each character is a hex digit:

| | | |
|---|---|---|
| Header: | | |
| | `wwww hhhh` | Width, height (16 bits each, LSB first) |
| For each data line: | | |
| | `llll` | line number (16 bits, LSB first, 0 to LINES-1) |
| | `rr rr rr rr rr ...` | the red components for that line (8 bits each - 0=dark, 255=bright) |
| | `gg gg gg gg gg ...` | the green components for that line (8 bits each - 0=dark, 255=bright) |
| | `bb bb bb bb bb ...` | the blue components for that line (8 bits each - 0=dark, 255=bright) |

Note that this format is slightly different from QRT's.

The `+fr` option of the raytracer produces "raw" files. These are simply three files with no header information and no line number information – just the raw data.

The `+ft` option writes out Targa format. Specifically, the fields are:

| | | |
|---|---|---|
| Header: | | |
| | `00 00 02 00 00` | Fixed header information for... |
| | `00 00 00` | ...uncompressed type 2 image |
| | `0000` | Horizontal offset always is at 0000 |
| | `llll` | Vertical offset (1st line number, 16 bits, LSB first) |
| | `wwww hhhh` | width, height of image (16 bits each, LSB first) |
| | `18 20` | 24 bits per pixel, Top-down raster |
| For each data line: | | |
| | `bb gg rr bb gg rr ...` | blue, green, and red data, 8 bits for each pixel in that line. |

# Chapter 10

# Common Questions and Answers

I often get asked the same questions again and again. I usually take this to mean that the documentation is not complete or not sufficiently clear. In order to correct this problem, I've added some sections to the document describing the features in more detail. I've also collected some of the more popular questions and answered them here.

Q: Will you be implementing radiosity?

A: I don't expect so. The techniques for radiosity are quite involved and time consuming (although they are getting faster). The amount of effort required to implement radiosity is beyond my current plans.

Q: Do you intend make DKBTrace RenderMan compatible?

A: Probably not. RenderMan is a specification that requires much more functionality than DKBTrace currently provides. The camera models, modelling primitives, and shading language of RenderMan are all very involved and difficult to implement. As well, RenderMan is not well suited to a raytracing approach. Don't expect to see DKBTrace RenderMan compatible in the near (or distant) future. Note, though that several of the DKBTrace textures are similar to those obtainable by RenderMan.

Q: I defined a light source but the shadows and lighting are all wrong.

A: Light sources must be defined at the origin (0,0,0) and translated to the proper place. The reason for this is to allow the diffuse lighting calculations to quickly determine where the center of the light source is. It's a very difficult task to calculate the center of an object (in general), so I simply take the place that the object was translated to as the center of the light source.

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. 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`s instead of `COMPOSITE` objects to put pieces together. Previous versions of the raytracer didn't allow this because the texture applied to the entire object. Version 2.10 and up allow you to change the texture on a per-shape basis.
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 colour table instead of 24 bits per pixel.

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

A: Oh no! Not another one! The raytracer (obviously) performs *many* floating point operations when tracing a scene. If I had to check each one for overflow or underflow, the program would be much longer and I would be much closer to going insane trying to locate all possible cases. If you get this problem, I'd suggest that you first look through your data file to make sure you're not doing something stupid like:

- Scaling something by 0.0 in *any* dimension;
- Making the `LOOK_AT` point the same as the `LOCATION`;

– Defining triangles with two points the same (or nearly the same);
– Using the zero vector for normals.

If it doesn't seem to be one of these problems, let me know, but there may not be a lot I can do because overflows can occur almost anywhere. Sorry. If you do have such troubles, you can try to isolate the problem in the input data 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 capped cylinder, you must use CSG:

```
INTERSECTION
    QUADRIC Cylinder_Y END_QUADRIC
    PLANE <0.0 1.0 0.0> 1.0 END_PLANE
    PLANE <0.0 -1.0 0.0> 1.0 END_PLANE
END_INTERSECTION
```

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: Why don't you define a primitive for a 6-sided box?

A: Because you can do it so easily with CSG:

```
INTERSECTION
   PLANE < 1.0  0.0  0.0> 1.0 END_PLANE
   PLANE <-1.0  0.0  0.0> 1.0 END_PLANE
   PLANE < 0.0  1.0  0.0> 1.0 END_PLANE
   PLANE < 0.0 -1.0  0.0> 1.0 END_PLANE
   PLANE < 0.0  0.0  1.0> 1.0 END_PLANE
   PLANE < 0.0  0.0 -1.0> 1.0 END_PLANE
END_INTERSECTION
```

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 (see Figure 10.1). In other words,

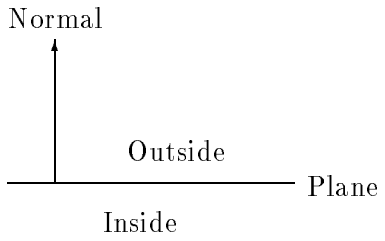


Figure 10.1: The "sides" of a plane

planes are 3D objects that are infinitely thick.

Q: Can DKBTrace render soft shadows?

A: No. This would require a lot more programming work and a lot more calculation time. You can place an unturbulated `WOOD` texture "filter" over a lamp in a can, and use a colour map going from dark (`ALPHA` 0.0) around the edges to clear (`ALPHA` 1.0) at the center, and get a soft-shadow-like effect for a "spotlight".

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 5-10loss 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 viewpoint is not very long, you may get distortion at the edges of the screen. The reason for this is that the viewpoint's screen is flat, as illustrated in Figure 10.2. The object labelled A appears smaller than the object labelled B, despite their actually being the size size.

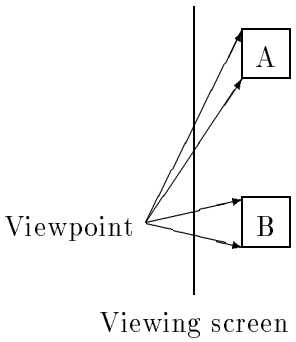


Figure 10.2: Distortion with short `DIRECTION` vector

Q: How do you position 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 colour that determines how much light can pass through that colour. 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: When I implemented smooth triangles, I never really expected anyone to manually calculate the surface normals. There are two ways of getting another program to calculate them for you:

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 DKB-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.

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. DKBTrace 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 `COLOUR` declared inside a `TEXTURE` and one that's in a shape or an object and not in a texture?

A: The colour in the texture specifies the colour to use for qualities 5 and up. The colour on the shape and object are used for faster rendering in qualities 4 and lower and for the colour 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), then 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.

Q: Will you be writing a Graphical User Interface for DKB?

A: No, but several other people have expressed an interest in writing one. I'd like to form a mailing list to get all these people in touch with each other, so if you have an interest in this, please let me know.

Q: When will the next version be available?

A: If I told you, I'd be lying because I don't know. I'm finding that releasing an official version of a program like this is a major effort. It requires not only the changes to the code, but also changes to the documentation. Sometimes (as with this release), I have to change the older data files to conform to the new syntax. Usually, I have to spend a lot of time re-rendering scenes that used to work to make sure they still do. That combined with sending the files to beta testers, getting feedback, making fixes, and re-issuing the changes adds up to a lot of work. I don't expect I'll be doing it terribly often.

Bottom line – If I say "next week", don't believe me. I'm probably wrong.

# Chapter 11

# Converting Data Files from Versions Prior to 2.10

Unfortunately, version 2.10 changed the format of the data files. These changes were mostly to avoid confusion – especially in the case of layered textures. There were also some changes that were made to allow more flexible library objects. This section details the changes in the data file format and why it was necessary to make those changes. I'm sorry that you have to go back through all your old data files and edit them all, but I think you'll agree (I hope you'll agree) that the changes are for the better.

1. The following keywords are no longer accepted outside of a `TEXTURE` block:

   `AMBIENT`, `DIFFUSE`, `BRILLIANCE`, `REFLECTION`, `REFRACTION`, `IOR`, `PHONG`, `PHONGSIZE`, `SPECULAR`, `ROUGHNESS`

   **Reason:** For layered textures, it's not clear which texture these keywords apply to if they're not in a `TEXTURE` block. I was really getting messed up in the implementation trying to locate the proper references for these keywords. It's much cleaner to only allow them inside a `TEXTURE` block.

2. A new texture called `COLOUR` (or `COLOR`) has been added. This texture simply specifies a simple colour to use.

   **Reason:** It's useful to be able to declare a `TEXTURE` with the colour embedded inside it. Besides, I had to provide some way of specifying a simple colour after I made the following change.

3. The keyword `COLOUR` or `COLOR` when used outside of a `TEXTURE` block is only used to provide a colour for low-quality renderings (ones where the `-q` value is 5 or below.

   **Reason:** This is the same reason as for change 1. For layered textures, it wasn't clear what the object or shape colour meant when the texture itself contained colour information. Rather than have a convoluted searching scheme (which I tried at first, then abandoned due to difficulties in explaining it), I decided to keep it simple. If you want a simple colour, put it in a `TEXTURE` block.

4. When an object or shape is transformed, the textures attached to it are transformed as well.

   **Reason:** Previously, you couldn't `DECLARE` objects with textures, then create identical copies of them in different places. This was especially annoying when you create an object that has an image mapped onto it. As soon as you moved the object, the image was left behind. Unfortunately, this means that the texture transformations inside `TEXTURE` blocks in old data files would double transform the texture, and must be removed.

5. The interaction between `ALPHA` colours and `REFRACTION` has changed. Previously, there was no interaction at all. If you had a surface that contained an `ALPHA` colour, you could see through the surface. If the surface had refraction, you could also see through. With the new release, `ALPHA` and `REFRACTION` in combination tell you how much light is passed through from the inside of the object.

   To make things a bit easier, if an object has an `ALPHA` component but the `REFRACTION` is 0.0 (or unspecified), the renderer will simply transmit the ray through the object without any refractive bending.

   **Reason:** It seemed to make sense. The code for `ALPHA` was doing the same work as the code for `REFRACTION` except for actually bending the ray. Now, you can create objects that are partially opaque and partially transparent. Where they are transparent, the light passing through the object is bent by the index of refraction. Makes sense, no?

6. When you use an `IMAGEMAP` with the `ONCE` option, the colour outside the mapped image is transparent (`RED 1.0 GREEN 1.0 BLUE 1.0`

`ALPHA 1.0)`.

**Reason:** This allows you to see the underlying textures.

7. I've removed `BasicShapes.data` and replaced it with `shapes.dat`, `colors.dat`, and `textures.dat`.

   **Reason:** The old name didn't apply any more (since it contained more than just shapes). Also, I was tired of the IBM people using the name `BASICSHA.DAT`.

8. The textures in `textures.dat` (formerly `BasicShapes.data`) were previously scaled to `<10.0 10.0 10.0>`. This scaling has been removed.

   **Reason:** The factor of 10.0 seemed to be a totally arbitrary scale factor that someone used. I'd like to keep the scaling factors up to the users.

# Chapter 12

# Handy Hints

This chapter provides a list of helpful hints.

- To see a quick version of your picture, use `-w64 -h80` as command line parameters on the Amiga. For the IBM, try `-w80 -h50`. This displays the picture in a small rectangle so that you can see how it will look.
- Try using the sample default files for different usages – `QUICK.DEF` shows a fast postage-stamp rendering (80x50, as above) to the screen only, `LOCKED.DEF` will display the picture with anti-aliasing on (takes forever) with no abort (do this before you go to bed...). The normal default options file `trace.def` is read and you can supersede this with another `.def` file by specifying it on the command line, for example:

```
trace -iworld.dat -oworld.out quick.def
```

- When translating light sources, translate the `OBJECT`, not the `QUADRIC` surface. The light source uses the center of the object as the origin of the light.
- 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 in version 2.10 if the transformations are placed *after* the `TEXTURE` block.
- 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.

- The amount of ambient light plus diffuse light should be less than or equal to 1.0. The program accepts any values, but may produce strange results.

- When using `RIPPLES`, don't make the ripples too deep or you may get strange results (the dreaded "digital zits"!).

- `WOOD` textures usually look better when they are scaled to different values in $x$, $y$, and $z$, and then rotated to a different angle.

- You can sort of dither a colour by placing a floating point number into the texture record:

```
TEXTURE
     0.05
END_TEXTURE
```

  This adds a small random value to the intensity of the diffuse light on the object. Don't make the number too big or you may get strange results. You generally won't want to use this if you are rendering frames for animations, as this "dithering" is random. Better results can be obtained, however, by doing proper dithering in a post-processor.

- You can compensate for non-square aspect ratios on the monitors by making the `RIGHT` vector in the `VIEWPOINT` longer or shorter. A good value for the Amiga is about 1.333. This seems okay for IBM's too at 320x200 resolution. If your spheres and circles aren't round, try varying it.

- 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 right. All you have to do is negate the terms in the `RIGHT` vector of the viewpoint to flip the camera left-to-right (use the "right-hand" coordinate system).

- By making the `DIRECTION` vector in the `VIEWPOINT` longer, you can achieve the effect of a tele-photo lens.

- When rendering on the Amiga, use a resolution of 320 by 400 to create a full sized HAM picture.

# Chapter 13

# Compiling the Code

If you want to compile the source code on a supported platform, do the following:

1. Copy or rename the file called *system*`conf.h` to `config.h`, where *system* corresponds to the platform you wish to compile for. For example:

   ```
   rename amigaconf.h config.h
   ```

2. Copy or rename the appropriate make file to `makefile`. For example:

   ```
   rename amigamake makefile
   ```

3. Edit the makefile to make sure all compiler options are set up properly for your system.

4. Type `make`.

# Chapter 14

# Porting to Different Platforms

I've taken great pains to make DKBTrace as portable as possible. So far, it's working out fairly well. For the most part, the core of the raytracer will compile with any decent C compiler.

If you want to port the raytracer to another system, please try to modify the core of the raytracer as little as possible. Each system will have its own makefile, config file, and C file. The config file is included by all of the raytrace modules and can be used to perform special `#defines` for that system. The C file should contain all system-specific code. It must implement the following functions:

```
void display_init()      /* Open the graphics device and initialize it */
void display_close()     /* Close the graphics device                  */
void display_finished()  /* Perform any operations required after      */
                         /* finishing the rendering but before quitting*/
void display_plot (x, y, Red, Green, Blue)
                         /* Display the specified colour at point x,y  */
```

In your config file, you may customize the following things:

```
#define FILE_NAME_LENGTH        /* default 150                        */
#define DBL_FORMAT_STRING       /* the string to use to read a double */
#define DEFAULT_OUTPUT_FORMAT   /* 'd', 'r' or 't' - default output   */
                                /*  file format                       */
#define TEST_ABORT              /* the operation to perform while     */
                                /* tracing to see if we should abort  */
                                /* the trace.                         */
#define RED_RAW_FILE_EXTENSION  /* the default extensions for +fr     */
```

```
#define GREEN_RAW_FILE_EXTENSION
#define BLUE_RAW_FILE_EXTENSION
#define STARTUP_DKB_TRACE       /* the code to call immediately after */
                                /* starting the main program. Useful  */
                                /* if you don't have a command-line   */
                                /* interface.                         */
#define PARAMS(x)               /* "(x)" if you have prototypes, "()" */
                                /* otherwise.                         */
```

# Chapter 15

# Program History and Information

## Version 1.2

Initial Amiga release.

## Version 2.0

- First release Version 2.0 Conversion to the IBM done by Aaron A. Collins.
- New textures, Specular and Phong highlighting added by Aaron A. Collins.
- Triangle, Smooth Triangle, Sphere, Plane support added by David Buck.
- RAW, IFF and GIF image mapping added by David Buck and Aaron Collins.
- Transparency and Fog added by David Buck.
- GIF format file reader by Steve Bennett (used with permission).
- New Noise and DNoise functions by Robert Skinner (used with permission).
- TARGA format output file capability added by Aaron A. Collins.

- ANSI-C function prototyping for ALL functions.
- Reversal of the order of writing screen data from the original DKB/QRT "RAW" file format.
- For IBM's, it has a crude VGA 320x200 by 256 color display rendering ability.

Version 2.0 compiles under Turbo-C 2.0 on the IBM P.C. and Lattice C 5.05 on the Amiga. The only file which contains the ANSI extensions is `dkbproto.h`, so for non-ANSI compilers, you only need to remove the declaration of the parameters in the `config.h` file and the whole thing should compile. There are several example `config.h` files for Amiga, IBM, and Unix. The appropriate one should be copied over `CONFIG.H`, and the `MAKEFILE` should be edited for your particular system and compiler configuration before compilation.

Version 2.0 has a significant difference from prior releases: Speed! The new primitives of `SPHERE`, `PLANE`, `TRIANGLE`, etc. greatly speed up tracing. Another significant speed-up is that world X-Y-Z values beyond 10 Million or so are ignored, and ray tracing beyond that distance will cease. This produces 2 minor peculiarities:

1. A black stripe at the horizon point of Pre-2.0 scene description `.data` files that have "ground" and "sky" planes defined. The planes were traced out to a much greater "infinity" so this effect was unnoticeable, prior to version 2.0.
2. Tiny black pixels in the texture, or "Surface Acne".

This is usually caused by rays being refracted or reflected such that the ray does not happen to hit any object, and eventually becomes black in color as it gets too far away and gets clipped. This effect can be minimized by enclosing the scene with distant "walls", "floors", or placing "ocean floors" beneath water, etc. So far, no scenes have required placing such planes behind the camera, unless an "environment map" of sorts is desired. See `SKYTEST.DAT` for several examples of spurious distant planes. If your "acne" still doesn't go away, it may be due to a large pixel sample area and it's accidentally picking a point which is just inside the primitive being hit. This is a more tricky problem to solve, and anti-aliasing the image will definitely help if this sort of thing occurs.

## Version 2.10

A few unofficial releases were made between 2.01 and 2.10. The following points capture the major changes:

- Less memory is required for image mapped GIF and IFF files.
- The output format command-line option was changed to `+f`$x$ where $x$ denotes the output format.
- The display option `+d` now takes an optional extra character `+d`$x$ where $x$ is system-dependent. This allows you to specify the graphics mode by a command-line switch.
- The tokenizing pass has been removed. It's now called directly by the parser.
- The environment variable `DKBOPT` is used in addition to the `trace.def` file and the command-line options.
- The numbers in the data file can now use full scientific notation, eg. 10.23e-4.
- The `+c` option was added to continue an aborted trace.
- You can now colour or texture each component of the CSG's separately.
- Layered textures implemented (see the section on textures).
- When using GIF and IFF images for image mapping, you can now indicate that specified registers are partially or completely transparent.
- Textures are now transformed whenever the object or shape they are attached to are transformed.
- The texture `CHECKER_TEXTURE` has been added.
- All keywords relating to the appearance of the surface have been made illegal in an object definition unless they are inside a `TEXTURE` block.
- The "basicshapes.data" file has been split up into `shapes.dat`, `colors.dat`, and `textures.dat`. These files have also been expanded with more useful declarations.
- The `-l` command-line option has been added to support library directories.

## Version 2.11

- Quartic surfaces (4th order) added by Alexander Enzmann.
- Parser now accepts `Ctrl-Z` as a whitespace.
- Keyword `END_SMOOTH_TRIANGLE` added (previously, `END_TRIANGLE` was used.)

## Version 2.12

- Bug in smooth triangles fixed to allow them to be scaled and translated.
- `METALLIC` texture added.

# Chapter 16

# References

I'm always asked about good books on raytracing and graphics in general. This section address that issue by listing several good books or periodicals that you should be able to locate in your local computer book store or your local university library.


- J. D. Foley and A. Van Dam. *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, 1983.
- J. D. Foley, A. van Dam, J. F. Hughes. *Computer Graphics: Principles and Practice*, (2nd Ed.). Addison-Wesley, 1990.
- Andrew S. Glassner (ed.). *An Introduction to Raytracing.* Academic Press, 1989.
- Andrew S. Glassner (ed.) *Graphics Gems.* Academic Press, 1990.
- Nadia Magnenat-Thalman and Daniel Thalmann. *Image Synthesis: Theory and Practice.* Springer-Verlag, 1987.
- Clifford Pickover. *Computers, Pattern, Chaos, and Beauty.* St. Martin's Press.
- David von Seggern *The CRC Handbook of Mathematical Curves and Surfaces* CRC Press, 1990.
- Steve Upstill. *The RenderMan Companion.* Addison Wesley, 1989.
- *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 Alimitos, CA 90720
- *The CRC Handbook of Standard Mathematical Tables* CRC Press.

# Chapter 17

# Concluding Remarks

It seems that in any project like this, as soon as you fix some bugs, some more appear. I expect that the new features I provided in this release will cause some problems somewhere. There's only so much you can do to get all the bugs out. For the next little while, I intend to keep the program stable feature-wise and just tackle major bugs. Hopefully things will stablize this way (and I can take a much needed breather).

Thanks go out to Alexander Enzmann for generously providing the code for the `QUARTIC` surface algorithms, and for his assistance in implementing it and for helping squash a couple of other bugs.

I would like to thank all the people who used versions 2.01 and 2.04 and provided useful comments and helpful suggestions. The program has been much improved with your help.

I would also like to thank my beta testers for all the help, bug reports, suggestions, comments, and time spent. I'd like to say a special thank-you to Aaron Collins who has fielded many of the questions on the raytracer and has been invaluable in debugging, testing, and generally improving the program. Thanks guys.

David Buck

# Chapter 18

# About the Authors

David Buck can be reached on the following BBS'es:

| | |
|---|---|
| OMX | (613) 731-3419 |
| Mystic | (613) 731-0088 or (613) 731-6698 |
| FidoNet | `1:163/109.9` |
| Internet | `dbuck@ccs.carleton.ca` |
| CIS | `70521,1371` |
| "You can call me Ray" | (708) 358-5611 |

Aaron Collins (IBM port) can be reached on the following BBS'es:

| | |
|---|---|
| "You can call me Ray" | (708) 358-5611 |

As of July of 1990, there is a Ray-Trace specific BBS in the (708) Area Code (Chicago suburbia) for all you Traceaholics out there. The phone number of this new BBS is (708) 358-5611. Aaron Collins is Co-Sysop of that board, and David Buck is a frequent caller. There is now also a Ray-Trace and Computer-Generated Art specific SIG on Compuserve, `GO COMART`.

The LaTeX version of the manual was produced by George Ferguson, (`ferguson@cs.rochester.edu`).

# Index


`a` option, 5, 8
`AGATE` keyword, 48
`ALL` keyword, 51
`ALPHA` keyword, 20, 43, 47, 50, 51, 66, 71, 78, 79, 83
`AMBIENT` keyword, 42, 82
animation, *see* textures, animation of
anti-aliasing, *see* `a` option
aspect ratio, 86
*AutoCad*, 60
*Autodesk Animator*, 57

`b` option, 9
Bennett, Steve, 1, 90
bounding shapes, 53, 80
box, 77
`BOZO` keyword, 41, 46, 69
`BRILLIANCE` keyword, 42, 82
Buck, David, 90, 96, 97
buffer size, *see* `b` option
`BUMPS` keyword, 50

`c` option, 9, 92
camera model, 30
`catdump`, 61
`CHECKER` keyword, 41, 45
`CHECKER_TEXTURE` keyword, 45, 92
`Chem2DKB`, 60
Collins, Aaron, 60, 62, 90, 96, 97
`COLOUR` keyword, 39, 49, 80, 82
colour
  and color, 25
  and `TEXTURE`, 80
  data type, 25
colour map data type, 25
command-line options, 6
  `a` option, 5, 8
  `b` option, 9
  `c` option, 9, 92
  `d` option, 4, 7, 92
  `e` option, 9
  `f` option, 4, 6, 55, 56, 62, 74, 92
  `h` option, 4, 6, 85
  `i` option, 5, 8
  `l` option, 10, 12, 52, 92
  `o` option, 5, 55
  `p` option, 5, 7
  `q` option, 10, 39, 49, 80, 83
  `s` option, 9
  `v` option, 4, 6
  `w` option, 4, 6, 85
  `x` option, 5, 8
  `z` option, 10
comments, 23
`compdump`, 61
compiling, 87
`COMPOSITE` keyword, 52, 76
constructive solid geometry, 37
  and bounding shapes, 53
  definition of box, 77

difference, 37
intersection, 37
transforming, 38
union, 37
continue rendering, *see* `c` option
coordinate system, 11
inverted Y, 64
left-handed, 24, 30
right-handed, 30
CSG, *see* constructive solid geometry
cube, 77

`d` option, 4, 7, 92
`d2iff`, 56
data types
colour, 25
colour map, 25
complex, 26
float, 23
vector, 24
debugging, *see* `z` option
declarations, 26
composite, 27
first principles, 26
layered textures, 27
textures, 27
`DECLARE` keyword, 26, 70, 76, 83
`DENTS` keyword, 50
`DIFFERENCE` keyword, 18, 37
`DIFFUSE` keyword, 42, 82
`DIRECTION` keyword, 28, 64, 78, 86
displaying
after tracing, 55
while tracing, *see* `d` option
distortion, 78
dithering, simulated, 86
`DKBOPT` environment variable, 4, 6, 10, 92
dump format, 74
`dump2i24`, 61
`dump2mtv`, 61
`dump2raw`, 62
`DumpToIFF`, 55
`DXF2DKB`, 60

`e` option, 9
elevation, and `UP` vector, 30
`END_SMOOTH_TRIANGLE` keyword, 93
ending scan line, *see* `e` option
Enzmann, Alexander, 93, 96
errors, 36, 76
exiting early, *see* `x` option

`f` option, 4, 6, 55, 56, 62, 74, 92
Farmer, Dan, 60
FBM utilities, 58
Ferguson, George, 97
filename
input, *see* `i` option
output, *see* `o` option
float data type, 23
fog, 30
`FREQUENCY` keyword, 50

GIF, 1, 5, 51, 57, 62, 72, 76
`gluetga`, 62
`GRADIENT` keyword, 48
gradient, *see* image mapping
`GRANITE` keyword, 49, 50

`h` option, 4, 6, 85
`halftga`, 62
Hassi, Paul, 62
height of picture, *see* `h` option

`i` option, 5, 8
IFF, 5, 51, 55, 61, 76
IFF format, 55

*Image Alchemy*, 57
image mapping, 51, 71
    and ALL, 51
    and ALPHA, 51
    and ONCE, 51, 83
    and TURBULENCE, 52
    filenames, 52
    gradient, 51, 72
    positioning, 79
    transformation, 51
IMAGE_MAP keyword, 62
IMAGEMAP keyword, 51, 83
INCLUDE keyword, 12, 23, 85
input filename, *see* i option
inside, of objects, 17
INTERSECTION keyword, 17, 37, 77
INVERSE keyword, 19, 38
IOR keyword, 43, 82

Kûhkünen, Jari, 62

l option, 10, 12, 52, 92
left-handed coordinate system, 24, 30
lemniscate curve, 36
library path, *see* l option
light
    reflected vs. filtered, 66
light sources
    and TEXTURE, 13
    and TRANSLATE, 13
    colour, 40
    composite objects, 53
    position restriction, 40, 75
    translating, 85
LIGHT_SOURCE keyword, 40
lighting
    total amount of, 86
Lissajou, 60
LOCATION keyword, 28, 64, 76
LOOK_AT keyword, 29, 65, 76

mapping, images, *see* image mapping
MARBLE keyword, 41
memory usage, 76
METALLIC keyword, 45, 50, 93

noise, 19, 68

o option, 5, 8, 55
OBJECT keyword, 38, 85
objects, 38
    as light sources, *see* light sources
    colour of, 39
    composite, 52
      as light sources, 53
    inside and outside, 17
    surface lighting
      ambient, 42
      brilliance, 42
      diffuse, 42
      index of refraction, 43
      metallic, 45
      phong highlighting, 44
      reflection, 43
      refraction, 43
      refraction, and ALPHA, 43
      refraction, and IOR, 43
      specular reflection, 44
    transforming, 39
      and textures, 39
ONCE keyword, 51, 83
options,command-line, *see* command-line options
output
    buffer size, *see* b option
    file generation, *see* f option
    filename, *see* o option

formats, 72
outside, of objects, 17

p option, 5, 7
parallel projection, 71
perturbation, 19
PHASE keyword, 50
PHONG keyword, 43, 44, 82
PHONGSIZE keyword, 44, 82
Pickover, Clifford, 60
PICLAB, 56
pinhole camera model, 30
piriform curve, 37
PLANE keyword, 33, 91
plane
inside and outside, 38, 78
primitive, 33
porting, 88
post-processing, 55
processing speed, 35, 54, 66, 78, 91
projection, *see* parallel projection
Puhl, Larry, 60

q option, 10, 39, 49, 80, 83
QUADRIC keyword, 85
quadric surfaces
definition of, 31
transformation of, 32
quadrics
scaling, 77
quality, *see* q option
QUARTIC keyword, 35
quartic surfaces, 34, 93
first principles declaration of, 35
math errors in, 36
questions, 75

radiosity, 75
Rasmussen, Helge E., 61
raw format, 74
RayScene, 62
REFLECTION keyword, 43, 82
REFRACTION keyword, 43, 67, 79, 82, 83
refraction, 66
rendering quality, *see* q option
RenderMan, 75
RIGHT keyword, 28, 64, 86
right-handed coordinate system, 30
RIPPLES keyword, 49, 86
ROTATE keyword, 24, 39
ROUGHNESS keyword, 44, 82

s option, 9
SA2DKB, 59
Saari, Ville, 61
SCALE keyword, 15, 24, 39
scaling qaudrics, 77
scan line
ending, *see* e option
starting, *see* s option
von Seggern, David, 37
shadows, 78
shapes.dat definitions file, 31
ShellGen, 60
Skinner, Robert, 2, 90
SKY keyword, 29, 65
SMOOTH_TRIANGLE keyword, 34
SPECULAR keyword, 43, 44, 82
specular highlight, 14
speed, *see* processing speed
SPHERE keyword, 32, 54, 91
sphere
primitive, 32
scaling of, 33
SPOTTED keyword, 47, 50
starting scan line, *see* s option
surfaces

quartic, 34

Targa, 4, 56, 57, 61, 62
Targe format, 74
tele-photo lens effect, 86
`TEXTURE` keyword, 41, 80, 82
textures, 41, 67
and `REFRACTION`, 70
and `TURBULENCE`, 19
animation of, 19, 85, 86
coloring
candy cane, 48
checkered, 45
gradient, 48
granite, 49
marbled, 41, 48
noisy, 46
spotted, 47
wood, 41
colour maps, 20, 41, 46
combining, 42
famous cloud, 21
image mapped, *see* image mapping
layered, 21, 42, 66, 70, 82, 92
marbled, 68
noisy, 69
parsing of, 70
perturbation
`COLOUR` ignored in, 49
bumpy, 50
dented, 50
frequency, 50
phase, 50
rippled, 49
wavy, 49
wrinkled, 50
randomness, 41, 80, 86
transforming, 41, 86
turbulence, 41
types, 41
wood, 68
`tga2dump`, 62
*The Art Department*, 55
torus, 36
scaling, 36
`trace.def` startup file, 4, 6, 10, 85, 92
transformations, 24
order of, 32
`TRANSLATE` keyword, 24, 39
`TRIANGLE` keyword, 33, 91
triangle
and CSG, 34
inside and outside, 17
primitive, 33
smooth shaded primitive, 34
surface normals, 34, 79
triangles
math errors in, 77
`TURBULENCE` keyword, 41, 48, 52
turbulence, 19, 69
`Twister`, 60

`UNION` keyword, 17, 37, 76
`UP` keyword, 28, 64
utilities, 59

`v` option, 4, 6
vector data type, 24
verbosity, *see* `v` option
`VIEWPOINT` keyword, 86
viewpoint, 28, 64
aspect ratio, 86
direction, 28
first principles declaration of, 28
location, 28

right, 28
transformation of, 29
up, 28

`w` option, 4, 6, 85
wait for prompt, *see* `p` option
`WAVES` keyword, 49
Wells, Drew, 60
width of picture, *see* `w` option
`WOOD` keyword, 41, 78, 86
`WRINKLES` keyword, 50

`x` option, 5, 8

`z` option, 10