Tachyon User's Guide
UNDER DEVELOPMENT

John E. Stone
E-Mail john.stone@gmail.com

Abstract

This document contains information on using Tachyon to create ray traced images, and animations. Information on the parallel raytracing engine and its use as an external rendering library is contained in other documents. Corrections and suggestions should be mailed to the author at *john.stone@gmail.com*

Contents

1 Introduction

Tachyon is designed to be a very fast renderer, based on ray tracing, and employing parallel processing to achieve high performance.

At the present time, Tachyon and its scene description language are fairly primitive, this will be remedied as time passes. For now I'm going to skip the "intro to ray tracing" and related things that should probably go here, they are better addressed by the numerous books on the subject written by others. This document is designed to serve the needs of sophisticated users that are already experienced with ray tracing and basic graphics concepts, rather than catering to beginners. If you have suggestions for improving this manual, I'll be glad to address them as time permits.

Until this document is finished and all-inclusive, the best way to learn how Tachyon works is to examine some of the sample scenes that I've included in the Tachyon distribution. Although they are all very simple, each of the scenes tries to show something slightly different Tachyon can do. Since Tachyon is rapidly changing to accommodate new rendering primitives and speed optimizations, the scene description language is likely to change to some degree as well.

1.1 Tachyon Feature List

Although Tachyon is a relatively simple renderer, it does have enough features that they bear some discussion.

- Parallel execution using MPI.
- Parallel execution using POSIX or Unix-International threads libraries.
- Automatic grid-based spatial decomposition scheme for greatly increased rendering speeds.
- Simple antialiasing based on psuedo-random supersampling.
- Linear, exponential, and exponential-squared fog.
- Perspective, orthographic, and depth-of-field camera projection modes, with eye-space frustum controls.
- Positional, directional, and spot lights, with optional attenuation.
- Provides many useful geometric objects including Spheres, Planes, Triangles, Cylinders, Quadrics, and Rings

- Texture mapping, with automatic MIP-map generation
- Supports rendering of volumetric data sets

2 Compiling Tachyon From Source Code

In order to use Tachyon you may need to compile it from source code, since it is normally distributed in source code form. Building Tachyon binaries is a fairly straightforward process. Download the Tachyon distribution from the web/ftp server. Once you have downloaded the distribution, unpack the distribution using gunzip and tar. Once the distribution is unpacked, cd into the 'tachyon' directory, and then into the 'unix' directory. Once in the 'unix' directory, type 'make' to see the list of configurations that are currently supported.

```
johns:/disk5/users/johns/graphics % gunzip tachyon.tar.gz
johns:/disk5/users/johns/graphics % tar -xvf tachyon.tar.gz
johns:/disk5/users/johns/graphics % cd tachyon
johns:/disk5/users/johns/graphics/tachyon % cd unix
johns:/disk5/users/johns/graphics/tachyon/unix % make
  Choose one of the architectures specified below.
-------------------------------------------------------------
              Parallel Versions

paragon-thr-mpi - Intel Paragon    (MPI + Threads + Thread I/O)
 paragon-mp-mpi - Intel Paragon    (MPI + Threads + Reg I/O)
    paragon-mpi - Intel Paragon    (MPI)
    ipsc860-mpi - Intel iPSC/860   (MPI)
        sp2-mpi - IBM SP2          (MPI)
    solaris-mpi - Sun Solaris 2.x  (MPI)
      irix5-mpi - SGI Irix 5.x     (MPI)
    solaris-thr - Sun Solaris 2.x  Threads
 solaris-c4-thr - Sun Solaris 2.x  Threads (Sun C 4.x)
-------------------------------------------------------------
              Sequential Versions

   solaris-v9 - Sun Solaris 2.5      (Sun C 4.x)
   solaris-c4 - Sun Solaris 2.[345]  (Sun C 4.x)
   solaris-c3 - Sun Solaris 2.[345]  (Sun C 3.x)
       sunos4 - SunOS 4.1.x
```

```
        irix5 - SGI Irix 5.x (32 bit, R4000)
        irix6 - SGI Irix 6.x (64 bit, R8000)
          aix - IBM AIX 3.x  (Generic RS/6000)
      aix-ppc - IBM AIX 3.x  (PPC 601)
         hpux - HP/UX 9.x and 10.x
        linux - Linux (on a little endian machine
          bsd - BSD   (on a little endian machine

        clean - Remove .o .a and executables
----------------------------------------------------------------
Type: 'make arch' to build for an architecture listed above.

johns:/disk5/users/johns/graphics/ray/unix % make solaris-thr

[lots of make output ommitted]
```

Hopefully once you've run 'make' to build the ray tracer for your machine, everything went well and you now have a binary to run. If you are building an MPI version, you may need to edit the make-config file to edit the locations of libraries and header files as they are listed there. If you have trouble, for now the best way to go is to send me email, at *john.stone@gmail.com* As I have time I'll improve this document and give more detailed instructions on building.

3 Running Tachyon

Since Tachyon runs on a wide variety of platforms, the exact commands required to run it vary substantially. The easiest way to get started using Tachyon is to try running one of the non-parallel, uniprocessor versions first. Tachyon includes a built-in help page describing all available command line options with very brief text, this help page is displayed when Tachyon is run with the `-help` option.

3.1 General command line options

Several command line options are available to tune Tachyon performance, display built-in help text, and set output verbosity.

- `-nobounding`: disable automatic generation of hierarchical grid-based acceleration data structures

- `-boundthresh` *object_count*: override default threshold for subdividing grid cells with a new grid
- `-numthreads` *thread_count*: command line override for the number of threads to spawn during the ray tracing process. When this options is not specified, Tachyon determines the number of threads to spawn based on the number of CPUs available on a given node.
- `+V`: enable verbose status messages, including reporting of overall node and processor count.
- `-nosave`: disable saving of output images to disk files. This feature is normally only used when benchmarking, or when using one of the OpenGL-enabled Tachyon configurations which provide runtime display of rendered images.
- `-camfile` *filename*: run a fly-through animation from the named camera file.

3.2 Command line shading controls

Tachyon supports a number of command line parameters which affect the quality and algorithms used to render scene files. The parameters select one of several quality levels, which implement various compromises between rendering speed and quality. Along with the overall shading quality controls, several specific options provide control over individual rendering algorithms within Tachyon.

- `-fullshade`: enables the highest quality rendering mode
- `-mediumshade`: disables computation of shadows, ambient occlusion
- `-lowshade`: minmalistic shading, using texture colors only
- `-lowestshade`: solid colors only
- `-aasamples` *sample_count*: command line override for the number of antialiasing supersamples computed for each pixel. A value of zero disables antialiasing. If this option is not used, the number of antialiasing samples is determined by the contents of the scen file.
- `-rescale_lights` *scalefactor*: rescale all light intensity values by the specified factor. (performed before other lighting overrides take effect)

- `-auto_skylight` *aofactor*: force the use of ambient occlusion lighting, automatically rescaling all other light sources to compensate for the additional illumination from the ambient occlusion lighting.
- `-add_skylight` *aofactor*: force the use of ambient occlusion lighting, existing lights must be rescaled manually using the `-rescale_lights` flag.
- `-skylight_samples` *samplecount*: number of samples to use for ambient occlusion lighting shadow tests.
- `-shade_phong`: use traditional phong shading for specular highlights.
- `-shade_blinn`: use Blinn's equation for specular highlights.
- `-shade_blinn_fast`: use a fast approximation to Blinn-style specular highlights.
- `-shade_nullphong`: entirely disables computation of specular highlights by registering a no-op function pointer.
- `-trans_orig`: use original Tachyon transparency mode.
- `-trans_vmd`: a special transparency mode designed for use with VMD. The resulting color is multiplied by opacity, giving results similar to what one would see with screen-door transparency in OpenGL.

3.3 Command line image format options

Tachyon optionally supports several image file formats for output. The output format is specified by the `-format` *formatname* command line parameter. Several of these formats are only available if Tachyon has been compiled with optional features enabled.

- `-res` *Xresolution Yresolution*: override the scene-defined output image resolution parameters.
- `TARGA`: uncompressed 24-bit Targa file
- `BMP`: uncompressed 24-bit Windows bitmap
- `PPM`: uncompressed 24-bit NetPBM portable pixmap (PPM) file
- `RGB`: uncompressed 24-bit Silicon Graphics RGB file

- `JPEG`: compressed 24-bit JPEG file
- `PNG`: uncompressed 24-bit PNG file

3.4 Tips for running MPI versions

Tachyon support the use of MPI for distributed memory rendering of complex scenes. Most commercial supercomputers and cluster vendors provide their own custom-tuned implementations of MPI which perform optimally on their hardware. Homegrown clusters typically use either the LAM or MPICH implementation of MPI. While Tachyon will work with any comformant implementation of MPI, some implementations perform much better than others. In the author's experience, the LAM implementation of MPI gives the best performance when used with Tachyon.

3.5 Interactive ray tracing

Tachyon is fast enough to support ray tracing at interactive rates when run on a large enough parallel computer, or with a simple enough scene. To this end, Tachyon can be optionally compiled with support for the Spaceball 6DOF motion control device. Using the Spaceball, one can fly around in an otherwise static scene. This is accomplished with the `-spaceball` *serial_port_device* command line parameters.

4 Scene Description Files

At the present time, scene description files are very simple. The parser can't handle multiple file scene descriptions, although they may be added in the future. Most of the objects and their scene description are closely related to the Tachyon API *(See the API docs for additional info.)*

4.1 Basic Scene Requirements

Unlike some other ray tracers out there, Tachyon requires that you specify most of the scene parameters in the scene description file itself. If users would rather specify some of these parameters at the command line, then I may add that feature in the future. A scene description file contains keywords, and values associated or grouped with a keyword. All keywords can be in caps, lower case, or mixed case for the convenience of the user. File names and texture names are normally case-sensitive, although the behavior for file names is operating system-dependent. All values are either

character strings, or floating point numbers. In some cases, the presence of one keyword will require additional keyword / value pairs.

At the moment there are several keywords with values, that must appear in every scene description file. Every scene description file must begin with the **BEGIN_SCENE** keyword, and end with the **END_SCENE** keyword. All definitions and declarations of any kind must be inside the **BEGIN_SCENE**, **END_SCENE** pair. The **RESOLUTION** keyword is followed by an x resolution and a y resolution in terms of pixels on each axis. There are currently no limits placed on the resolution of an output image other than the computer's available memory and reasonable execution time. An example of a simple scene description skeleton is show below:

```
BEGIN_SCENE
  RESOLUTION 1024 1024
...
...  Camera definition..
...
...  Other objects, etc..
...

END_SCENE
```

4.2 Camera and viewing parameters

One of the most important parts of any scene, is the camera position and orientation. Having a good angle on a scene can make the difference between an average looking scene and a strikingly interesting one. There may be multiple camera definitions in a scene file, but the last camera definition overrides all previous definitions. There are several parameters that control the camera in Tachyon, **PROJECTION**, **ZOOM**, **ASPECTRATIO**, **ANTIALIASING**, **CENTER**, **RAYDEPTH**, **VIEWDIR**, and **UPDIR**.

The first and last keywords required in the definition of a camera are the **CAMERA** and **END_CAMERA** keywords. The **PROJECTION** keyword is optional, the remaining camera keywords are required, and must be written in the sequence they are listed in the examples in this section.

4.2.1 Camera projection modes

The **PROJECTION** keyword must be followed by one of the supported camera projection mode identifiers **PERSPECTIVE**, **PERSPECTIVE_DOF**,

ORTHOGRAPHIC, or **FISHEYE**. The **FISHEYE** projection mode requires two extra parameters **FOCALLENGTH** and **APERTURE** which precede the regular camera options.

```
Camera
  projection perspective\_dof
  focallength 0.75
  aperture 0.02
  Zoom 0.666667
  Aspectratio 1.000000
  Antialiasing 128
  Raydepth 30
  Center  0.000000 0.000000 -2.000000
  Viewdir -0.000000 -0.000000 2.000000
  Updir   0.000000 1.000000 -0.000000
End\_Camera
```

4.2.2 Common camera parameters

The **ZOOM** parameter controls the camera in a way similar to a telephoto lens on a 35mm camera. A zoom value of 1.0 is standard, with a 90 degree field of view. By changing the zoom factor to 2.0, the relative size of any feature in the frame is twice as big, while the field of view is decreased slightly. The zoom effect is implemented as a scaling factor on the height and width of the image plane relative to the world.

The **ASPECRATIO** parameter controls the aspect ratio of the resulting image. By using the aspect ratio parameter, one can produce images which look correct on any screen. Aspect ratio alters the relative width of the image plane, while keeping the height of the image plane constant. In general, most workstation displays have an aspect ratio of 1.0. To see what aspect ratio your display has, you can render a simple sphere, at a resolution of 512x512 and measure the ratio of its width to its height.

The **ANTIALIASING** parameter controls the maximum level of supersampling used to obtain higher image quality. The parameter given sets the number of additional rays to trace per-pixel to attain higher image quality.

The **RAYDEPTH** parameter tells Tachyon what the maximum level of reflections, refraction, or in general the maximum recursion depth to trace rays to. A value between 4 and 12 is usually good. A value of 1 will disable rendering of reflective or transmissive objects (they'll be black).

The remaining three camera parameters are the most important, because they define the coordinate system of the camera, and its position in the scene. The **CENTER** parameter is an X, Y, Z coordinate defining the center of the camera *(also known as the Center of Projection)*. Once you have determined where the camera will be placed in the scene, you need to tell Tachyon what the camera should be looking at. The **VIEWDIR** parameter is a vector indicating the direction the camera is facing. It may be useful for me to add a "Look At" type keyword in the future to make camera aiming easier. If people want or need the "Look At" style camera, let me know. The last parameter needed to completely define a camera is the "up" direction. The **UPDIR** parameter is a vector which points in the direction of the "sky". I wrote the camera so that **VIEWDIR** and **UPDIR** don't have to be perpendicular, and there shouldn't be a need for a "right" vector although some other ray tracers require it. Here's a snippet of a camera definition:

```
CAMERA
  ZOOM 1.0
  ASPECTRATIO 1.0
  ANTIALIASING 0
  RAYDEPTH 12
  CENTER 0.0 0.0 2.0
  VIEWDIR 0 0 -1
  UPDIR 0 1 0
END\_CAMERA
```

4.2.3 Viewing frustum

An optional **FRUSTUM** parameter provides a means for rendering sub-images in a larger frame, and correct stereoscopic images. The **FRUSTUM** keyword must be followed by four floating parameters, which indicate the top, bottom, left and right coordinates of the image plane in eye coordinates. When the projection mode is set to **FISHEYE**, the frustum parameters correspond to spherical coordinates specified in radians.

```
CAMERA
  ZOOM  1.0
  ASPECTRATIO 1.0
  ANTIALIASING 0
  RAYDEPTH  4
  CENTER    0.0 0.0 -6.0
```

```
  VIEWDIR   0.0 0.0 1.0
  UPDIR     0.0 1.0 0.0
  FRUSTUM   -0.5 0.5 -0.5 0.5
END\_CAMERA
```

4.3 Including Files

The **INCLUDE** keyword is used anywhere after the camera description, and is immediately followed by a valid filename, for a file containing additional scene description information. The included file is opened, and processing continues as if it were part of the current file, until the end of the included file is reached. Parsing of the current file continues from where it left off prior to the included file.

4.4 Scene File Comments

The # keyword is used anywhere after the camera description, and will cause Tachyon to ignore all characters from the # to the end of the input line. The # character must be surrounded by whitespace in order to be recognized. A sequence such as ### will not be recognized as a comment.

4.5 Lights

The most frequently used type of lights provided by Tachyon are positional point light sources. The lights are actually small spheres, which are visible. A point light is composed of three pieces of information, a center, a radius (since its a sphere), and a color. To define a light, simply write the **LIGHT** keyword, followed by its **CENTER** (a X, Y, Z coordinate), its **RAD** (radius, a scalar), and its **COLOR** (a Red Green Blue triple). The radius parameter will accept any value of 0.0 or greater. Lights of radius 0.0 will not be directly visible in the rendered scene, but contribute light to the scene normally. For a light, the color values range from 0.0 to 1.0, any values outside this range may yield unpredictable results. A simple light definition looks like this:

```
  LIGHT CENTER 4.0 3.0 2.0
        RAD    0.2
        COLOR  0.5 0.5 0.5
```

This light would be gray colored if seen directly, and would be 50% intensity in each RGB color component.

Tachyon supports simple directional lighting, commonly used in CAD and scientific visualization programs for its performance advantages over positional lights. Directional lights cannot be seen directly in scenes rendered by Tachyon, only their illumination contributes to the final image.

```
DIRECTIONAL_LIGHT
  DIRECTION 0.0 -1.0 0.0
  COLOR   1.0 0.0  0.0
```

Tachyon supports spotlights, which are described very similarly to a point light, but they are attenuated by angle from the direction vector, based on a "falloff start" angle and "falloff end"angle. Between the starting and ending angles, the illumination is attenuated linearly. The syntax for a spotlight description in a scene file is as follows.

```
SPOTLIGHT
  CENTER  0.0 3.0  17.0
  RAD     0.2
  DIRECTION 0.0 -1.0 0.0
    FALLOFF_START 20.0
    FALLOFF_END   45.0
  COLOR   1.0 0.0  0.0
```

The lighting system implemented by Tachyon provides various levels of distance-based lighting attenuation. By default, a light is not attenuated by distance. If the *attenuation* keywords is present immediately prior to the light's color, Tachyon will accept coefficients which are used to calculate distance-based attenuation, which is applied the light by multiplying with the resulting value. The attenuation factor is calculated from the equation

$$\frac{1}{K_c + K_l d + k_q d^2} \tag{1}$$

This attenuation equation should be familiar to some as it is the same lighting attenuation equation used by OpenGL. The constant, linear, and quadratic terms are specified in a scene file as shown in the following example.

```
LIGHT
  CENTER  -5.0 0.0 10.0
  RAD     1.0
  ATTENUATION CONSTANT 1.0 LINEAR 0.2 QUADRATIC 0.05
  COLOR   1.0 0.0 0.0
```

4.6 Atmospheric effects

Tachyon currently only implements one atmospheric effect, simple distance-based fog.

4.6.1 Fog

Tachyon provides a simple distance-based fog effect intended to provide functionality similar to that found in OpenGL, for compatibility with software that requires an OpenGL-like fog implementation. Much like OpenGL, Tachyon provides linear, exponential, and exponential-squared fog.

```
FOG
  LINEAR START 0.0  END 50.0  DENSITY 1.0  COLOR 1.0 1.0 1.0

FOG
  EXP START 0.0  END 50.0  DENSITY 1.0  COLOR 1.0 1.0 1.0

FOG
  EXP2 START 0.0  END 50.0  DENSITY 1.0  COLOR 1.0 1.0 1.0
```

4.7 Objects

4.7.1 Spheres

Spheres are the simplest object supported by Tachyon and they are also the fastest object to render. Spheres are defined as one would expect, with a **CENTER**, **RAD** (radius), and a texture. The texture may be defined along with the object as discussed earlier, or it may be declared and assigned a name. Here's a sphere definition using a previously defined "NitrogenAtom" texture:

```
SPHERE  CENTER 26.4 27.4 -2.4   RAD 1.0   NitrogenAtom
```

A sphere with an inline texture definition is declared like this:

```
Sphere center 1.0 0.0 10.0
          Rad 1.0
       Texture  Ambient 0.2  Diffuse 0.8  Specular 0.0  Opacity 1.0
                Color   1.0 0.0 0.5
                TexFunc 0
```

Notice that in this example I used mixed case for the keywords, this is allowable... Review the section on textures if the texture definitions are confusing.

4.7.2 Triangles

Triangles are also fairly simple objects, constructed by listing the three vertices of the triangle, and its texture. The order of the vertices isn't important, the triangle object is "double sided", so the surface normal is always pointing back in the direction of the incident ray. The triangle vertices are listed as **V1**, **V2**, and **V3** each one is an X, Y, Z coordinate. An example of a triangle is shown below:

```
TRI
  V0  0.0 -4.0 12.0
  V1  4.0 -4.0 8.0
  V2 -4.0 -4.0 8.0
  TEXTURE
    AMBIENT  0.1 DIFFUSE  0.2 SPECULAR 0.7 OPACITY 1.0
    COLOR 1.0 1.0 1.0
    TEXFUNC 0
```

4.7.3 Smoothed Triangles

Smoothed triangles are just like regular triangles, except that the surface normal for each of the three vertexes is used to determine the surface normal across the triangle by linear interpolation. Smoothed triangles yield curved looking objects and have nice reflections.

```
STRI
  V0 1.4   0.0   2.4
  V1 1.35 -0.37  2.4
  V2 1.36 -0.32  2.45
  N0 -0.9 -0.0  -0.4
  N1 -0.8  0.23 -0.4
  N2 -0.9  0.27 -0.15
  TEXTURE
    AMBIENT  0.1 DIFFUSE  0.2 SPECULAR 0.7 OPACITY 1.0
    COLOR 1.0 1.0 1.0
    TEXFUNC 0
```

4.7.4 Infinite Planes

Useful for things like desert floors, backgrounds, skies etc, the infinite plane is pretty easy to use. An infinite plane only consists of two pieces of information, the **CENTER** of the plane, and a **NORMAL** to the plane. The

center of the plane is just any point on the plane such that the point combined with the surface normal define the equation for the plane. As with triangles, planes are double sided. Here is an example of an infinite plane:

```
PLANE
  CENTER 0.0 -5.0 0.0
  NORMAL 0.0  1.0 0.0
  TEXTURE
    AMBIENT  0.1 DIFFUSE  0.9 SPECULAR 0.0  OPACITY 1.0
    COLOR  1.0 1.0 1.0
    TEXFUNC  1
      CENTER 0.0 -5.0 0.0
      ROTATE 0. 0.0 0.0
      SCALE  1.0 1.0 1.0
```

4.7.5 Rings

Rings are a simple object, they are really a not-so-infinite plane. Rings are simply an infinite plane cut into a washer shaped ring, infinitely thing just like a plane. A ring only requires two more pieces of information than an infinite plane does, an inner and outer radius. Here's an example of a ring:

```
  Ring
    Center 1.0 1.0 1.0
    Normal 0.0 1.0 0.0
    Inner  1.0
    Outer  5.0
    MyNewRedTexture
```

4.7.6 Infinite Cylinders

Infinite cylinders are quite simple. They are defined by a center, an axis, and a radius. An example of an infinite cylinder is:

```
  Cylinder
    Center 0.0 0.0 0.0
    Axis   0.0 1.0 0.0
    Rad    1.0
    SomeRandomTexture
```

4.7.7 Finite Cylinders

Finite cylinders are almost the same as infinite ones, but the center and length of the axis determine the extents of the cylinder. The finite cylinder is also really a shell, it doesn't have any caps. If you need to close off the ends of the cylinder, use two ring objects, with the inner radius set to 0.0 and the normal set to be the axis of the cylinder. Finite cylinders are built this way to enhance speed.

```
FCylinder
  Center 0.0 0.0 0.0
  Axis   0.0 9.0 0.0
  Rad    1.0
  SomeRandomTexture
```

This defines a finite cylinder with radius 1.0, going from 0.0 0.0 0.0, to 0.0 9.0 0.0 along the Y axis. The main difference between an infinite cylinder and a finite cylinder is in the interpretation of the **AXIS** parameter. In the case of the infinite cylinder, the length of the axis vector is ignored. In the case of the finite cylinder, the axis parameter is used to determine the length of the overall cylinder.

4.7.8 Axis Aligned Boxes

Axis aligned boxes are fast, but of limited usefulness. As such, I'm not going to waste much time explaining 'em. An axis aligned box is defined by a **MIN** point, and a **MAX** point. The volume between the min and max points is the box. Here's a simple box:

```
BOX
  MIN -1.0 -1.0 -1.0
  MAX  1.0  1.0  1.0
  Boxtexture1
```

4.7.9 Fractal Landscapes

Currently fractal landscapes are a built-in function. In the near future I'll allow the user to load an image map for use as a heightfield. Fractal landscapes are currently forced to be axis aligned. Any suggestion on how to make them more appealing to users is welcome. A fractal landscape is defined by its "resolution" which is the number of grid points along each

axis, and by its scale and center. The "scale" is how large the landscape is along the X, and Y axes in world coordinates. Here's a simple landscape:

```
SCAPE
  RES 30 30
  SCALE 80.0 80.0
  CENTER 0.0 -4.0 20.0
  TEXTURE
    AMBIENT 0.1 DIFFUSE 0.9 SPECULAR 0.0 OPACITY 1.0
    COLOR 1.0 1.0 1.0
    TEXFUNC 0
```

The landscape shown above generates a square landscape made of 1,800 triangles. When time permits, the heightfield code will be rewritten to be more general and to increase rendering speed.

4.7.10 Arbitrary Quadric Surfaces

Docs soon. I need to add these into the parser, must have forgotten before ;-)

4.7.11 Volume Rendered Scalar Voxels

These are a little trickier than the average object :-) These are likely to change substantially in the very near future so I'm not going to get too detailed yet. A volume rendered data set is described by its axis aligned bounding box, and its resolution along each axis. The final parameter is the voxel data file. If you are seriously interested in messing with these, get hold of me and I'll give you more info. Here's a quick example:

```
SCALARVOL
  MIN -1.0 -1.0 -0.4
  MAX  1.0  1.0  0.4
  DIM 256 256 100
  FILE /cfs/johns/vol/engine.256x256x110
  TEXTURE
        AMBIENT 1.0 DIFFUSE 0.0 SPECULAR 0.0 OPACITY 8.1
        COLOR 1.0 1.0 1.0
        TEXFUNC 0
```

4.8 Texture and Color

4.8.1 Simple Texture Characteristics

The surface textures applied to an object drastically alter its overall appearance, making textures and color one of the most important topics in this manual. As with many other renderers, textures can be declared and associated with a name so that they may be used over and over again in a scene definition with less typing. If a texture is only need once, or it is unique to a particular object in the scene, then it may be declared along with the object it is applied to, and does not need a name.

The simplest texture definition is a solid color with no image mapping or procedural texture mapping. A solid color texture is defined by the **AMBIENT**, **DIFFUSE**, **SPECULAR**, **OPACITY** and **COLOR** parameters. The **AMBIENT** parameter defines the ambient lighting coefficient to be used when shading the object. Similarly, the **DIFFUSE** parameter is the relative contribution of the diffuse shading to the surface appearance. The **SPECULAR** parameter is the contribution from perfectly reflected rays, as if on a mirrored surface. **OPACITY** defines how transparent a surface is. An **OPACITY** value of 0.0 renders the object completely invisible. An **OPACITY** value of 1.0 makes the object completely solid, and non-transmissive. In general, the values for the ambient, diffuse, and specular parameters should add up to 1.0, if they don't then pixels may be over or underexposed quite easily. These parameters function in a manner similar to that of other ray tracers. The **COLOR** parameter is an RGB triple with each value ranging from 0.0 to 1.0 inclusive. If the RGB values stray from 0.0 to 1.0, results are undefined. In the case of solid textures, a final parameter, **TEXFUNC** is set to zero (integer).

4.8.2 Texture Declaration and Aliasing

To define a simple texture for use on several objects in a scene, the **TEXDEF** keyword is used. The **TEXDEF** keyword is followed by a case sensitive texture name, which will subsequently be used while defining objects. If many objects in a scene use the same texture through texture definition, a significant amount of memory may be saved since only one copy of the texture is present in memory, and its shared by all of the objects. Here is an example of a solid texture definition:

```
TEXDEF MyNewRedTexture
   AMBIENT 0.1 DIFFUSE 0.9 SPECULAR 0.0 OPACITY 1.0
   COLOR 1.0 0.0 0.0  TEXFUNC 0
```

When this texture is used in an object definition, it is referenced only by name. Be careful not to use one of the other keywords as a defined texture, this will probably cause the parser to explode, as I don't check for use of keywords as texture names.

When a texture is declared within an object definition, it appears in an identical format to the **TEXDEF** declaration, but the **TEXTURE** keyword is used instead of **TEXDEF**. If it is useful to have several names for the same texture (when you are too lazy to actually finish defining different variations of a wood texture for example, and just want to be approximately correct for example) aliases can be constructed using the **TEXALIAS** keyword, along with the alias name, and the original name. An example of a texture alias is:

```
TEXALIAS MyNewestRedTexture  MyNewRedTexture
```

This line would alias MyNewestRedTexture to be the same thing as the previously declared MyNewRedTexture. Note that the source texture must be declared before any aliases that use it.

4.8.3 Image Maps and Procedural Textures

Image maps and procedural textures very useful in making realistic looking scenes. A good image map can do as much for the realism of a wooden table as any amount of sophisticated geometry or lighting. Image maps are made by wrapping an image on to an object in one of three ways, a spherical map, a cylindrical map, and a planar map. Procedural textures are used in a way similar to the image maps, but they are on the fly and do not use much memory compared to the image maps. The main disadvantage of the procedural maps is that they must be hard-coded into Tachyon when it is compiled.

The syntax used for all texture maps is fairly simple to learn. The biggest problem with the way that the parser is written now is that the different mappings are selected by an integer, which is not very user friendly. I expect to rewrite this section of the parser sometime in the near future to alleviate this problem. When I rewrite the parser, I may also end up altering the parameters that are used to describe a texture map, and some of them may become optional rather than required.

Texture Mapping Functions

Value for TEXFUNC	Mapping and Texture Description
0	No special texture, plain shading
1	3D checkerboard function, like a rubik's cube
2	Grit Texture, randomized surface color
3	3D marble texture, uses object's base color
4	3D wood texture, light and dark brown, not very good yet
5	3D gradient noise function (can't remember what it look like
6	Don't remember
7	Cylindrical Image Map, requires ppm filename
8	Spherical Image Map, requires ppm filename
9	Planar Image Map, requires ppm filename

Here's an example of a sphere, with a spherical image map applied to its surface:

```
SPHERE
  CENTER 2.0  0.0 5.0
  RAD 2.0
  TEXTURE
    AMBIENT  0.4 DIFFUSE  0.8 SPECULAR 0.0  OPACITY 1.0
    COLOR 1.0 1.0 1.0
    TEXFUNC 7 /cfs/johns/imaps/fire644.ppm
      CENTER 2.0 0.0 5.0
      ROTATE 0.0 0.0 0.0
      SCALE  2.0 -2.0 1.0
```

Basically, the image maps require the center, rotate and scale parameters so that you can position the image map on the object properly

Index