Tachyon User's Guide
UNDER DEVELOPMENT

John E. Stone
E-Mail johns@megapixel.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 *johns@megapixel.com*

Contents

1 Introduction

At the present time, Tachyon and its scene description grammar are extremely primitive, this will be remedied as time passes. For now I'm going to skip the "Intro to Raytracing" and related things that should probably go here, I'll put them in later. This document is currently designed to serve the needs of sophisticated users. If you have suggestions, I'll be glad to get to them.

Until this document is finished, 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 Tachyoncan do. Since Tachyon is rapidly changing to accomodate new rendering primitives and speed optimizations, the scene description language is likely to change to some degree as well. The scene desciption language used is extremely simple.

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
- Provides many useful geometric objects including Spheres, Planes, Triangles, Cylinders, Quadrics, and Rings
- Texture mapping
- 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 *johns@megapixel.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.

3.1 Command line parameters

- -fullshade this option enables the highest quality rendering mode

3.2 Tips for running MPI versions

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 raytracing engine's API *(See the API docs for additional info).*

4.1 Basic Scene Requirements

Unlike some other raytracers out there, Tachyonrequires 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. 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 next keyword required after **BEGIN_SCENE** is **OUTFILE**. The **OUTFILE** keyword precedes the name of the output file. The only output file format currently implemented is *Targa 24*. Targa files can be read by most image viewers and converters. Following the **OUTFILE** keyword and filename is the output file resolution. 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. The last keyword in the head of a scene description is the **VERBOSE** keyword. The **VERBOSE** keyword is followed by an integer value indicating how much information should be printed while the raytracer is rendering the scene. At the present time, the **VERBOSE** is either zero or nonzero, with zero meaning no extra message, and nonzero printing full diagnostic messages. An example of a simple scene description skeleton is show below:

```
BEGIN_SCENE
  OUTFILE myimage.tga
  RESOLUTION 1024 1024
  VERBOSE 0
...
...  Camera definition..
...
...  Other objects, etc..
...

END_SCENE
```

4.2 The Camera

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 are seven parameters that control the camera in this raytracer, **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 remaining camera keywords are all required, and must be written in the sequence they are listed below, and as shown in the example.

The **ZOOM** parameter controls the camera in a way similar to a telephoto lens on a standard 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 the raytracer what the maximum level of reflections, refractions, 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 the raytracer 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 raytracers 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
```

4.3 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

In the initial versions of Tachyon the only type of lights available are point lights. The lights are actually small spheres, which are visible. If users would like more advanced lighting such as spot lights with falloff and other features, let me know. A 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) 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.

4.6 Objects

4.6.1 Spheres

Spheres are the simplest object supported by the raytracer, 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.

4.6.2 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
```

4.6.3 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
```

4.6.4 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
```

4.6.5 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
```

4.6.6 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
```

4.6.7 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.

4.6.8 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
```

4.6.9 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.

4.6.10 Arbitrary Quadric Surfaces

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

4.6.11 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
```

Index