
Ray Tracing and Mathematical Graphics

■ What is Ray Tracing?

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

Some of the programs available: AVS, geomview (very, very expensive) but allow (among others) real time interpolation.

POV-Ray (Persistence of vision) is a free program that can be obtained on the web (www.povray.org).

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

- * Easy to use scene description language
- * Large library of stunning example scene files
- * Standard include files that pre-define many shapes, colors and textures
- * Very high quality output image files (24-bit color)
- * Create landscapes using smoothed height fields
- * Spotlights for sophisticated lighting
- * Phong and specular highlighting for more realistic-looking surfaces
- * Several image file output formats including Targa, PNG and PPM
- * Basic shape primitives such as ... sphere, box, quadric, cylinder, cone, triangle and plane
- * Advanced shape primitives such as ... torus (donut), hyperboloid, paraboloid, bezier patch, height fields (mountains), blobs, quartics, smooth triangles, text, fractals, superquadrics, surfaces of revolution
- * Shapes can easily be combined to create new complex shapes. This feature is called Constructive Solid Geometry (CSG). POV-Ray supports unions, merges, intersections and differences in CSG.
- * Objects are assigned materials called textures. (A texture describes the coloring and surface properties of a shape.)
- * Built-in color patterns: Agate, Bozo, Checker, Granite, Gradient, Leopard, Mandel, Marble, Onion, Spotted, Radial, Wood and image file mapping.
- * Built-in surface bump patterns: Bumps, Dents, Ripples, Waves, Wrinkles and mapping.

-
- * Users can create their own textures or use pre-defined textures such as... Mirror, Metals like Chrome, Brass, Gold and Silver, Bright Blue Sky with Clouds, Sunset with Clouds, Sapphire Agate, Jade, Shiny, Brown Agate, Apocalypse, Blood Marble, Glass, Brown Onion, Pine Wood, Cherry Wood
 - * Combine textures using layering of semi-transparent textures or tile or material map files.
 - * Display preview of image while computing (not available on all computers)
 - * Animation

■ An example of a POV -Ray file

■ Step 1: Make a file test.pov

The simplest file must contain: the camera location, an object and the light source

■ Step 2: Place the camera

The usual coordinate system for POV-Ray has the positive Y axis pointing up, the positive X axis pointing to the right, and the positive Z axis pointing into the screen as follows:

```
 ^+Y
 | /+Z
 | /
 |/
 |/   +X
 |----->
```

The syntax is straightforward and will be learned in the process:

```
camera {
  location <0, 2, -3>      #the location of the camera
  look_at <0, 1, 2>      #the point in the center of the screen
}
```

■ Step 3: Define an object

We place a sphere around <0,1,2> of radius 2 with a yellow color

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

Yellow is pre-defined. Any color can be specified by the command

`color rgb<r,g,b>` where r,g,b are numbers between 0 and 1.

■ Step 4: Define a light source

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

These are the minimum requirements to start a program: the camera, the light source and the object. The complete file looks like: (test1.pov)

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

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

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

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

Let us try now. (file test1.pict) It works but is a little boring. Notice the difference in reflection already. Let us try some variations:

■ Step 5: Making a picture nicer

The famous checkered floor. Among the objects we can add to the scene, we can add a plane with a checkered floor:

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

We can make the surface look nicer by adding some finish. The `phong` keyword adds a highlight the same color of the light shining on the object (test2.pov)

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

```
    }  
  }
```

The surface of the sphere is too smooth, maybe some bumpiness would help (test3.pov):

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

If you prefer, you can use pre-defined textures (first: #include "textures.inc").

Here I added a sky with some clouds. the sphere is Red marble and the plane sandalwood. (test4.pov)

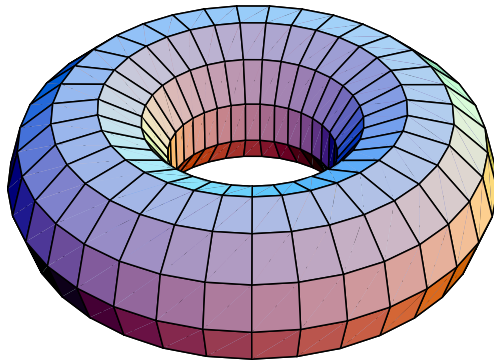
```
sphere {  
  <0, 1, 2>, 2  
  texture {  
    pigment {Red_Marble}  
    finish { Shiny } // This finish defined in textures.inc  
  }  
}  
plane {  
  <0, 1, 0>, -1  
  texture {Sandalwood}  
}  
sky_sphere {  
  pigment {  
    gradient y  
    color_map {  
      [0.000 0.002 color rgb <1.0, 0.2, 0.0>  
        color rgb <1.0, 0.2, 0.0>]  
      [0.002 0.200 color rgb <0.8, 0.1, 0.0>  
        color rgb <0.2, 0.2, 0.3>]  
    }  
    scale 2  
    translate -1  
  }  
  pigment {  
    bozo  
    turbulence 0.65  
    octaves 6  
    omega 0.7  
    lambda 2  
    color_map {
```

```
    [0.0 0.1 color rgb <0.85, 0.85, 0.85>
      color rgb <0.75, 0.75, 0.75>]
    [0.1 0.5 color rgb <0.75, 0.75, 0.75>
      color rgbt <1, 1, 1, 1>]
    [0.5 1.0 color rgbt <1, 1, 1, 1>
      color rgbt <1, 1, 1, 1>]
  }
  scale <0.2, 0.5, 0.2>
}
rotate -135*x
}
```

■ Mathematica Graphics

The problem is to transfer the graphics 3D data from *Mathematica* to a format compatible with POVray. Consider:

```
G=ParametricPlot3D[{Cos[phi](2+Cos[psi]),  
Sin[phi](2+Cos[psi]),  
Sin[psi]},{phi,0,2 Pi},  
{psi,0,2 Pi},PlotPoints->{36,12},  
Axes->None,Boxed->False]
```



-Graphics3D-

Short [G[[1]],2]

```
{Polygon[{{3., 0., 0.}, {2.95179, 0.535671, 0.}, <<1>>,  
{2.84125, 0., 0.540641}}], <<384>>}
```

The graphics is made out of little polygons. In order to produce a smooth surface one has to interpolate the color within the different triangles. Therefore one has to create a connectivity matrix to remember how the different polygons are patched together.

Needs ["SurfaceGraphics3D`"]

Needs ["POVray`"]

The **SurfaceGraphics3D.m** package overrides the definition of **ParametricPlot3D** and create a new format which include information on the connectivity. This information can be passed on to the program **POVray** which creates two files **file.inc** and **file.pov**.

The program **SurfaceGraphics3D.m** overrides the definition of **ParametricPlot3D** and transform the format to the format **SurfaceGraphics3D** which takes into account the connectivity of the polygons.

```
G=ParametricPlot3D[{Cos[phi](2+Cos[psi]),
                  Sin[phi](2+Cos[psi]),
                  Sin[psi]}, {phi, 0, 2 Pi},
                  {psi, 0, 2 Pi}, PlotPoints->{36, 12},
                  Axes->None, Boxed->False]
```

```
-SurfaceGraphics3D-
```

```
Short[InputForm[G], 3]
```

```
SurfaceGraphics3D[{{<<12>>},
                  {{{-3., 0, 0}, <<34>>, {-3., 0., 0}}, <<11>>},
                  {PlotPoints -> {36, 12}, Axes -> None, Boxed -> False}]
```

This format is compatible with **POVray** and can be modified by the procedure **POVray**. The program **POVray** creates two files: **torus.pov** and **torus.inc**.

```
POVray["Macintosh HD:POV:swigpov:torus", G,
       POVInformation->True]
```

```
Macintosh HD:POV:swigpov:torus.pov
```

The first file contains the *Mathematica* informations on the position of the light sources (3 light, one red, one green one blue) and the camera (as you would see the object in mathematica).

```
// Persistence Of Vision raytracer version 3.0 sample file.
#version 3.0
global_settings { assumed_gamma 2.2 }
#include "colors.inc"
#include "shapes.inc"
#include "textures.inc"
#include "glass.inc"

/* Texture declarations for object 'mysurface' */
#declare mytexture =
  texture { pigment {color rgb <1,1,1>}
          finish {phong 0.8 ambient 0.4} }

camera {
  location <4.0928, -7.5524, 2.0786>
  direction <0, 3.3838, 0>
```

```

up <0, 0, 3.>
right <3., 0, 0>
sky <0., 0., 1.>
look_at <0.0060386, 0., 0.>
}
light_source {
  <3.1497, 0., 1.0393>
  color rgb <1, 0, 0>
}
light_source {
  <3.1497, 3.1468, 1.0393>
  color rgb <0, 1, 0>
}
light_source {
  <0.0060386, 3.1468, 1.0393>
  color rgb <0, 0, 1>
}

```

```
#include "Macintosh HD:POV:swigpov:torus.inc"
```

The file **torus.inc** contains the definition of the object as a set of smooth triangles with texture (*mytexture*). This texture can be changed by editing the file **torus.pov**

```

/* Object 'MYSFCE' */
  mesh {
smooth_triangle {
  <2.8413, 0, 0.54064>
  , <-2.3902, 0, -1.5361>
  , <2.7956, 0.50733, 0.54064>
  , <-2.3518, -0.42679, -1.5361>
  , <3., 0, 0>
  , <-3., 0, 0>
}
....
smooth_triangle {
  <2.8413, 0, -0.54064>
  , <-2.3902, 0, 1.5361>
  , <2.7956, -0.50733, -0.54064>
  , <-2.3518, 0.42679, 1.5361>
  , <3., 0, 0>
  , <-3., 0, 0>
}
texture { mytexture }}

```

The camera is well placed but a lot of little details can be readily changed to make the picture better. In the picture *torus.pict* I changed, the position of the camera, I included some plane under the torus.

I gave the surface a bright red color and added two planes on which the torus reflects itself *ad infinitum* .

The image is 624x832 pixels and took 20 min to render on the PowerPC9500/132

■ Examples

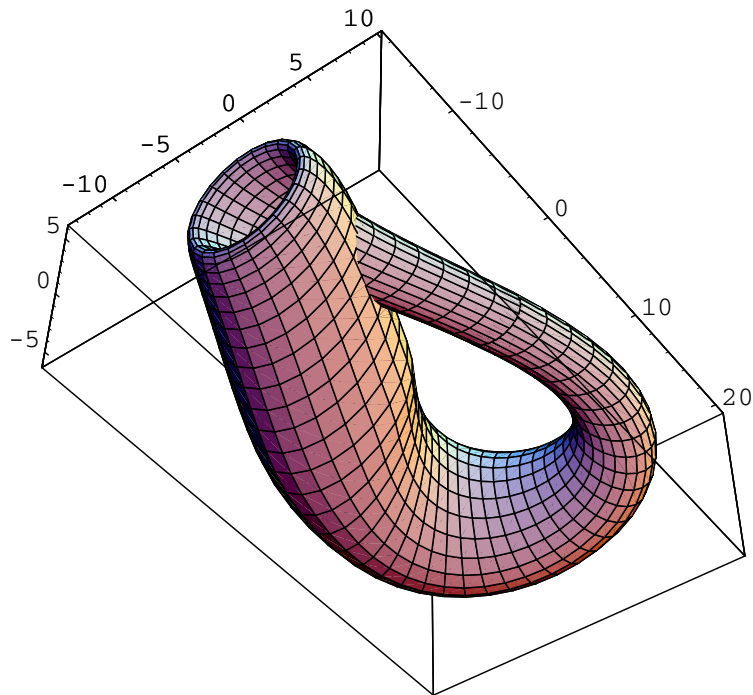
■ The Klein Bottle

The next one is the famous Klein bottle

```
bx = 6 Cos[u] (1 + Sin[u]);
by = 16 Sin[u];
rad = 4 (1 - Cos[u] / 2);
X = If[Pi < u <= 2 Pi, bx + rad Cos[v + Pi],
      bx + rad Cos[u] Cos[v]];
Y = If[Pi < u <= 2 Pi, by, by + rad Sin[u] Cos[v]];
Z = rad Sin[v];
Klein=ParametricPlot3D[{X, Y, Z}, {u, 0, 2 Pi},
                      {v, 0, 2 Pi+2 Pi/12},
                      PlotPoints -> {60,24}, Axes -> False,
                      Boxed -> False,
                      ViewPoint-> {1.4,-1.9,-2.4}]
```

-SurfaceGraphics3D-

```
Show[Klein, Boxed->True, Axes->True]
```



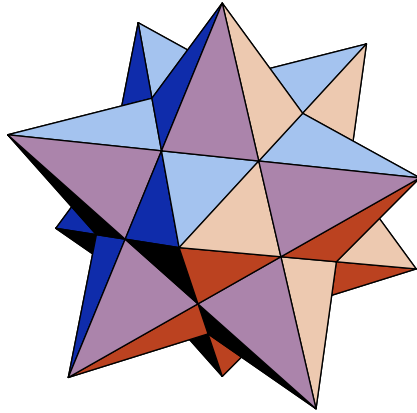
```
POVray["Macintosh HD:POV:swigpov:klein", Klein,  
POVInformation->True]
```

```
Macintosh HD:POV:swigpov:klein.pov
```

■ Some Polyhedra

```
Needs["Graphics`Polyhedra`"]  
dodec = Polyhedron[SmallStellatedDodecahedron]  
-Graphics3D-
```

```
Show[Polyhedron[SmallStellatedDodecahedron],Boxed -> False];
```



```
POVray["Macintosh HD:POV:swigpov:dodec",dodec,  
POVInformation->True]
```

```
Macintosh HD:POV:swigpov:dodec.pov
```

```
dodec = Polyhedron[Dodecahedron]  
POVray["Macintosh HD:POV:swigpov:dodec2",dodec,  
POVInformation->True]
```

■ A knot

□ Implementation

□ 5 crossings

```
x=32 Cos[s]-51 Sin[s]-104 Cos[2s]-34 Sin[2s]+104 Cos[3s]-91Sin[3s];  
y=94 Cos[s]+41 Sin[s]+113 Cos[2s]-68 Cos[3s]-124Sin[3s];  
z=16 Cos[s]+73 Sin[s]-211 Cos[2s]-39 Sin[2s]-99 Cos[3s]-21Sin[3s];
```