

Some hints, tips and code snippets - POV-Ray workshop

Brian R. Pauw*
DTU, Forskningscenter Risø
(Dated: March 23, 2007)

Contents

Exercise 1, The interface	1
Exercise 2, Setting the scene	1
Exercise 3, Basic objects	2
Exercise 4, Scale, rotate and translate	2
Exercise 5, Constructive Solid Geometry Operator	3
Exercise 6, Texture, interior and animation	4
some useful macro's	7
Dashed line	7
cube ribs	8
Axes	8
References	9

This program is one of the most well-documented open-source programs on the planet. This documentation can be found at <http://povray.org/documentation/>

EXERCISE 1, THE INTERFACE

The windows version places its templates in an "Insert" pulldown menu. When a template is selected, the template is placed in the scene (programming block) at the location of the cursor. Just like any text editor, you can cut, copy and paste

When selecting an item, be mindful of the position of the cursor in your programming block. You might inadvertently end up with a "cylinder" template inside another "cylinder" template, resulting in dysfunctional code.

In the top left corner of the main window, there is a list of preset resolutions you can choose from. Any loaded scene will render in this resolution. A resolution followed by an "AA" tag, will render the scene with antialiasing switched on.

The "messages" tab in the main window will show rendering information and statistics. It also shows the (sometimes cryptic, but often helpful) errors that point to programming errors in the scene description/program.

The "Run" button renders the scene description in the active tab.

EXERCISE 2, SETTING THE SCENE

Comments on a line can be inserted by placing two slashes ("/") followed by your comment. Anything on a line after these slashes does not get interpreted by POV-Ray.

Co-ordinates can be input either by using $\langle x, y, z \rangle$ notation or by multiplication of the "x", "y" and "z" vertices. The latter requires more insight into the co-ordinate system and vector multiplications. Therefore, the $\langle x, y, z \rangle$ -notation is advised to be used in the beginning.

A camera definition as:

```
camera{
location <0,0,-5> //the location of the camera
look_at <0,0,0> //the direction in which it is pointing
angle 60 //the opening angle of the camera
}
```

will give you a camera for a standard image size (i.e. one with a 4/3 aspect ratio, such as 640x480 or 1024x768). The positive x direction will be to the right of the image, positive y will be to the top of the image, and the positive z direction will be through (back out of) your image. This is the so-called cartographer's view.

It is easiest to place a standard (point) light source above the origin (from the camera viewpoint). When you select the template for a point light source, the POV-Ray templates show a peculiar way of defining the light position. Remove the "0*x"-line and the "translate"-line, and instead, place a set of coordinates directly after the opening bracket of the light source:

```
light_source{
<0,100,0> // the light location.
color rgb <1,1,1> //let's give it a color
}
```

A background is set by typing in the programming block (outside of any other definitions):

```
background{color rgb<0.7,0.7,0.7>}
```

This color is defined by giving the "red, green and blue" (RGB) values of the color, e.g. $\langle 0, 0, 0 \rangle$ is black, $\langle 1, 1, 1 \rangle$ is white and $\langle 1, 1, 0 \rangle$ is yellow.

A plane is the first object you will place. For our standard camera (described above), if we want a "ground"-plane, we type:

```
plane{
<0,1,0>, -1 //the normal vector and the distance to the origin
texture{pigment{color rgb <0.8,0.8,0.8>}} //let's give it a color
}
```

The color definition here is a little more complex, surrounded by a "texture" and "pigment" statement. This will be elaborated on later.

EXERCISE 3, BASIC OBJECTS

You can use your previously defined camera, light-source and ground plane here.

Remember that the plane effectively cuts off any object surface below $y = -1$. Also make note that the centre of any object outside of the sphere, is not defined as this could be any definition of centre (e.g. geometrical centre, weight centre, etc.). Placing a cylinder at $\langle 0, 0, 0 \rangle$ means that the endpoints should be placed around that point.

Objects can be given colors in a similar manner as done for the plane. An example input for the cylinder would be:

```
cylinder{
<-0.5,-0.5,0>, <0.5,0.5,0>, 0.2 //<END1>, <END2>, radius
texture{pigment{color rgbt<0.2,0.1,0.4,0.5>}}
}
```

The color is here defined with an additional variable "t", symbolising the "transmission" factor. 1 is fully transparent, 0 is fully opaque.

EXERCISE 4, SCALE, ROTATE AND TRANSLATE

The location of the transformation operators is the end of the object:

```

box{
    [object location]
    [object color, etc]
    scale <1,2,3>
    rotate <4,5,6>
    translate <7,8,9>
}

```

1. Choose object, place the center coordinates at origin
2. scale $\langle \lambda_x, \lambda_y, \lambda_z \rangle$
3. rotate $\langle \alpha, \beta, \gamma \rangle$
4. translate $\langle \Delta_x, \Delta_y, \Delta_z \rangle$

Reversing the order (especially the translation), might end up with you losing your object from the field-of-view. More elaborate transformations can be achieved with the “matrix” operator, this can apply a matrix transformation to the object, useful for shearing the object.

EXERCISE 5, CONSTRUCTIVE SOLID GEOMETRY OPERATOR

An intermediate render of overlapping objects will help make sure you do have objects with coinciding volumes.

A Constructive Solid Geometry operator should be placed around two (or more) objects. Make sure to place the pigment outside the objects, at the end of the CSG operator (but within the final closing curly bracket).

- Union creates a single object out of the “unionized” objects. Allows for texture definition outside each object, and can be useful for many objects.
- Merge merges the objects, so that the outer surface of the merged objects marks the confining space. Useful when filling a composite object with an atmosphere (described later), or when working with semitransparent objects.
- Intersection makes a new object out of the coincident volume of all objects.
- Difference subtracts all shapes from the first mentioned shape.

These CSG operations can be combined. You can “difference” an object with the result of an “intersect”. for example:

```

difference{
box{<-2,-0.5,-2>,<2,0,2>}
intersection{
box{<0.5,0.51,0.5>,<-0.5,-0.51,-0.5>}
sphere{<0,0,0>,1
scale <0.3,0.4,1>
rotate <10,0,0>
}
}
texture{pigment{color rgbt<1,1,0,0.5>}}
}

```

As the example shows, the individual objects can be transformed just like it was done before. It is useful to first render the object that is to be subtracted, to verify that it is in the right place with the right shape. Note the placement of the texture.

EXERCISE 6, TEXTURE, INTERIOR AND ANIMATION

A texture statement usually consists of the following elements

```
texture {
  pigment { color rgbt<r,g,b,t> }
  normal {
bumps 0.1 //bumps, dents, ripples, waves or wrinkles
  }
  rotate <px,py,pz> //affects pigment+normal
  scale <x,y,z>
  translate <x,y,z>
  finish {
    ambient 0.1 //self-glow
  diffuse 0.5 //scattering of other light-sources, creates shadows
  reflection 0.1 //shiny
  }
}
```

There are many more options available, all of which are described in the POV-Ray online documentation. This is where things get realistic and unrealistic.

An interior statement can look like this:

```
box<<1,1,1>,<-1,-1,-1>
hollow on
texture [...]
interior{
  absorption <r,g,b>
  scattering {[type], <r,g,b>} //type is 1,2,3,4 or 5
}
```

note the use of the “hollow on” option. If the object is not hollow, it will not be filled with the interior. The walls should also have a certain transparency to them in order to see the interior.

Animation can be achieved by using the “clock” variable. Substitution of any number (be it indicating a coordinate, color or other property), will make the clock variable, and thereby the object property, change with time.

POV-Ray will generate values for “clock” based on the number of frames to generate, and the beginning and endvalues of clock.

For example, you can move your camera forwards and down in two consecutive steps.

```
#switch (clock) //this tells the program to consider the variable clock
#range (0,1) //if clock is between 0 and 1, do the following:
camera{
  location <0,0,-5-(clock*5)>
  look_at <0,0-clock,0> //this is changed to prevent weird things.
}
#break //end the range statement
#range (1,2)
camera{
  location <0,0-5*clock,0>
  look_at <0,1-5*clock,0>
}
#break
#end //closing the switch statement
```

In order to tell POV-Ray how and how much to change the “clock” variable, the following lines need to be added to the “povray.ini” file:

```

;clock=1
Initial_Frame=1
Final_Frame=20
Initial_Clock=0
Final_Clock=2

```

The “clock” line is commented out in this example (i.e. it will not be read and interpreted by POV-Ray. Uncommenting it (removing the “;”) allows the user to render a single frame with a specific value for clock.

This example renders 20 images, each with a slightly different value for clock. These images can then be compiled into a movie

If a more smooth transition is requested, the clock variable can be surrounded by some mathematics, for example:

$$\frac{\sin\left(-\frac{\pi}{2} + \pi \times \text{clock}\right) + 1}{2} \quad (1)$$

substituting any variable for this expression will evoke a smooth transition from 0 to 1 (see example below).

As a complete example, here is the code for the movie shown at the beginning of the workshop:

```

// Persistence of Vision Ray Tracer Scene Description File
// File: final_animation.pov
// Vers: 3.6
// Desc: A rendering to show off the capabilities of POV-Ray
// Date: 20-3-07
// Auth: Brian R. Pauw

//camera definitions follow later, they change per section of clock

box {
  <-0.5, -0.5, 0>, <0.5, 0.5, 0>
  texture { pigment { rgb <1, 0, 1> } }
}

// create a regular point light source
light_source {
  0*x // light's position (translated below)
  color rgb <1,1,1> // light's color
  translate <-20, 40, -20>
}

// create a point "spotlight" (conical directed) light source
light_source {
  0*x // light's position (translated below)
  color rgb <1,1,1> // light's color
  spotlight // this kind of light source
  translate <40, 80, -40> // <x y z> position of light
  point_at <0, 0, 0> // direction of spotlight
  radius .1 // hotspot (inner, in degrees)
  tightness 50 // tightness of falloff (1..100) lower is softer, higher is tighter
  falloff 1 // intensity falloff radius (outer, in degrees)
}

background { color rgb <0.8, 0.8, 0.8> }

// An infinite planar surface
// plane {<A, B, C>, D } where: A*x + B*y + C*z = D
plane {
  y, // <X Y Z> unit surface normal, vector points "away from surface"

```

```

-1.0 // distance from the origin in the direction of the surface normal
hollow on // has an inside pigment?
texture {
  pigment {
    color rgbt<1,1,1,0>
  }
}
}

difference{
box {
  <-3, -0.5, -3> // one corner position <X1 Y1 Z1>
  < 3, 0.5, 3> // other corner position <X2 Y2 Z2>
}
union{
cylinder{<0,-0.51,0>, <0,0.51,0>, 0.51 translate 0*<1,0,1>}
cylinder{<0,-0.51,0>, <0,0.51,0>, 0.51 translate 1*<1,0,1>}
cylinder{<0,-0.51,0>, <0,0.51,0>, 0.51 translate -1*<1,0,1>}
cylinder{<0,-0.51,0>, <0,0.51,0>, 0.51 translate 1*<-1,0,1>}
cylinder{<0,-0.51,0>, <0,0.51,0>, 0.51 translate -1*<-1,0,1>}
cylinder{<0,-0.51,0>, <0,0.51,0>, 0.51 translate 2*<1,0,1>}
cylinder{<0,-0.51,0>, <0,0.51,0>, 0.51 translate -2*<1,0,1>}
cylinder{<0,-0.51,0>, <0,0.51,0>, 0.51 translate 2*<-1,0,1>}
cylinder{<0,-0.51,0>, <0,0.51,0>, 0.51 translate -2*<-1,0,1>}
cylinder{<0,-0.51,0>, <0,0.51,0>, 0.51 translate 1*<2,0,0>}
cylinder{<0,-0.51,0>, <0,0.51,0>, 0.51 translate 1*<0,0,2>}
cylinder{<0,-0.51,0>, <0,0.51,0>, 0.51 translate 1*<-2,0,0>}
cylinder{<0,-0.51,0>, <0,0.51,0>, 0.51 translate 1*<0,0,2>}
}

  hollow on
texture{pigment{color rgbt<0,1,1,0.5>}
finish{
reflection 0.5
}}
interior{ media{absorption <0.1,0.1,0.1>
scattering {1,<1,1,1>}}}

}

//-----
// here beginneth the clock-dependent items

#switch (clock)
#range (0,1)
sphere{<0,2-clock,-3+3*((sin(-0.5*pi+pi*clock)+1)/2)>,0.5
texture{pigment{rgb<0.8,0.8,0.7>}
finish{
ambient 0.1
diffuse 0.1
reflection 0.9
}
}
}
}
}

```

```

camera {
  location <2.0-2*clock, 2.0, -5.0+5*clock>
  look_at <0.0, 0.0-clock, 0.0>
  angle 60
}

#break

#range (1,2)
sphere{<0,1-2*(clock-1),0>,0.5
texture{pigment{rgb<0.8,0.8,0.7>}
finish{
ambient 0.1
diffuse 0.1
reflection 0.9
}
}
}

camera {
  location <0, 2.0-((sin(-0.5*pi+pi*(clock-1))+1)/2), 0>
  look_at <0.0, -1-2*(clock-1), 0.0>
  angle 60
}

#break
#end

```

SOME USEFUL MACRO'S

Dashed line

“Stippellijn” is a macro to make a dashed line with spheres as endcaps. Its syntax is:

```
Stippellijn(<Begin>,<End>,[number of pieces],[thickness])
```

And the code is:

```

#macro Stippellijn(Begin,Eind,stukjes,dikte)

#declare BallCount = 0;
#declare lengte=vlength( Eind-Begin ) ;//length of vector
#declare legestukjes=stukjes-1; //number of empty parts always end with non-empty parts
#declare stuklengte=lengte/(stukjes+legestukjes); //length of the parts
#declare stukje = vnormalize(Eind-Begin)*stuklengte;

#while (BallCount < stukjes)
  union{
    cylinder {<0,0,0>,stukje, dikte }
    translate (Begin+stukje*BallCount*2)
  }
  #declare BallCount = BallCount+1; // increment our counter
#end

// create a sphere shape
sphere {Begin,dikte} // capping cylinders
sphere {Eind,dikte} // capping cylinders

```

```
#end
```

cube ribs

The ribs of a cubic lattice can be described using the macro “Cubeydubey”, which places the ribs of a cube between the points described by x,y,z and 0. It works in conjunction with “Stippellijn”, in case one is in need of cube ribs consisting of dashed “lines”:

```
Cubeydubey([x],[y],[z],[number of pieces per rib],[thickness of rib])
```

The code is:

```
#macro Cubeydubey(scalex,scaley,scalez,stukjes, thick)
union {
Stippellijn( <0, 0, 0>,<scalex 0 0>,stukjes, thick )
Stippellijn( <0, 0, 0>,<0 scaley 0>,stukjes, thick )
Stippellijn( <0, 0, 0>,<0 0 scalez>,stukjes, thick )
Stippellijn( <scalex,scaley,scalez>,<0 scaley scalez>,stukjes, thick )
Stippellijn( <scalex,scaley,scalez>,<scalex 0 scalez>,stukjes, thick )
Stippellijn( <scalex,scaley,scalez>,<scalex scaley 0>,stukjes, thick )
Stippellijn( <scalex, 0, 0>,<scalex scaley 0>,stukjes, thick )
Stippellijn( <scalex, 0, 0>,<scalex 0 scalez>,stukjes, thick )
Stippellijn( <0, scaley, 0>,<scalex scaley 0>,stukjes, thick )
Stippellijn( <0, scaley, 0>,<0 scaley scalez>,stukjes, thick )
Stippellijn( <0, 0, scalez>,<scalex 0 scalez>,stukjes, thick )
Stippellijn( <0, 0, scalez>,<0 scaley scalez>,stukjes, thick )
}
#end
```

Axes

In case one loses sight of the direction of the axes in the scene, a very common case, here is a coordinate system indicator. The font (Chalkboard.ttf) might have to be placed in a location where POV-Ray can find it.

```
#macro Axes()
sphere{<0,0,0>,0.1} //neatify the ends of the cylinders
cylinder{<0,0,0>,<1,0,0>,0.05} //x
cone{<1,0,0>,0.15,<1.5,0,0>,0}
cylinder{<0,0,0>,<0,1,0>,0.05} //y
cone{<0,1,0>,0.15,<0,1.5,0>,0}
cylinder{<0,0,0>,<0,0,1>,0.05} //z
cone{<0,0,1>,0.15,<0,0,1.5>,0}
// create a TrueType text shape
union{
text {
ttf // font type (only TrueType format for now)
"Chalkboard.ttf"
"x", // the string to create
0.1, // the extrusion depth
0 // inter-character spacing
}
translate<1,0.2,0>
}
union{
```

```
text {
  ttf          // font type (only TrueType format for now)
  "Chalkboard.ttf"
  "y",        // the string to create
  0.1,        // the extrusion depth
  0           // inter-character spacing
}
translate<0.2,1.2,0>
}
union{
text {
  ttf          // font type (only TrueType format for now)
  "Chalkboard.ttf"
  "z",        // the string to create
  0.1,        // the extrusion depth
  0           // inter-character spacing
}
translate<0,0.2,1>
}
#end
```

* Electronic address: brian@stack.nl