A Quick Introduction to POV-Ray

POV-Ray, the "Persistence of Vision" ray tracer, is open source software available from www.povray.org. This is good, highly functional software that you might enjoy playing with. The *User's Guide* is 150 pages and the *Reference Manual* another 350 pages. This is just a quick introduction to get you started.

POV-Ray includes a text editor for creating a scene file. The production cycle requires you to create a scene, then click on the Run button to render the scene. The scene description language is similar to a programming language, with constructs to create objects, lights, cameras, and so forth. You can either create these objects directly or store them in variables and instantiate them when you need them. In the following I will do enough to show you some of the options.

Coordinate Systems. Before we can look at POV-Ray scences, you need to understand the POV-Ray coordinate system. POV-Ray uses a left-handed object coordinate system, with the x-axis horizontal and the y-axis vertical. If you think of a room, the x-axis is where the floor meets the front wall, the y-axis is where the front and left-side walls meet, and the *negative* z-axis is where the left side wall meets the floor. The positive z-axis extends into the next room.

First Example: Here is a simple POV-Ray scene. This has a yellow sphere of radius 3 at position (0, 1, 5). The camera is at (0, 3, -3) looking at the center of the sphere. There is one light in the scence, which is behind, above, and to the right of the viewer.

```
#include "colors.inc"

background { color Cyan }

camera {
        location <0,3,-3>
        look_at <0, 1, 5>
}

sphere {
        <0, 1, 5>, 3
        texture {
          pigment {color Yellow }
        }
}

light_source { <4, 10, -20> color White}
```

Cameras. The most important parameters for the camera are the camera location and lookat point. If you want to tilt the camera sideways, you can declare a *sky* vector, which is essentially what we have called the up-vector. This defaults to <0, 1, 0>. If you think of a vertical vector coming out of the top of the camera, the camera will be rotated until this vector lines up with the sky vector. For example, the following will rotate the camera by 45 degrees around an axis through its lens:

```
camera {
        location <0,3,-3>
        look_at <0, 1, 5>
        sky <1, 1, 0>
}
```

There are three parameters that are used to determine the size of the viewport. The up and right vectors set the aspect ratio by creating a rectangular box. The default values of these are <0,1,0> for *up* and <1.33,0,0> for *right*. The *angle* parameter, controls the horizontal aperture of vision; this is exactly what we called Theta in our view pipeline. This *angle* should be specified after *up* and *right*. For example, for a square window we might use

```
camera {
        location <0,3,-3>
        look_at <0, 1, 5>
        up <0,1,0>
        right <1,0,0>
        angle 60
```

There are a number of other camera parameters, but the only ones you are likely to want to use control the depth of field. There are three parameters to set for this. The *focal_point* is the point that is most in focus. The camera *aperture* controls the region of focus – the smaller the aperture the larger the area of focus of the lens. Finally, the blur_samples parameter controls the quality of the focal blur imaging. The higher this is, the more accurate the rendering will be and the longer it will take. For a quick view of the scene you might set blur_samples to 4 or 5; for a high quality production view you might increase this to 20 or more, but be aware that this significantly increases rendering time. Altogether, camera declaration with focal blur might be

```
camera {
        location <0,3,-3>
        look_at <0, 1, 5>
        focal_point <0, 1, 2>
        aperture 0.4
        blur_samples 20
```

Lights. Point lights have only color and location:

```
light_source {
        <4, 10, -20>
        color <1,1,1>  // or White
}
```

Spotlights have some additional parameters. First, to create a spotlight you need to add the *spotlight* directive to a light declaration. The *point_at* parameter controls the direction of the spotlight; it is like the look_at point for a camera. The *radius* parameter is the aperture of the fully illuminated portion of the cone. The *falloff* parameter is the angle of the umbra of the cone – the region of partial illumination and partial shadow. You get sharp shadows by setting falloff to 0, soft shadows by giving falloff a positive value. A typical spotlight declaration is

```
light_source {
        <4, 10, -20>
        color White
        spotlight
        radius 15
        falloff 20
        point_at <0,0,5>
```

An area light source is rectangular. To make an area light you specify a location for its center, axis vectors for its horizontal and vertical dimensions, and the number of rows and columns of lights within it. For example, the following places 25 small lights within a 1x1 rectangle parallel to the x-y plane

```
light_source {
        <4, 10, -20>
        color White
        area_light
        <1,0,1>,<0,1,0>,5,5
```

There are two additional flags for area lights. The *adaptive* n parameter tells the ray tracer that it may use "adaptive sampling" and send fewer than 1 ray per small light when illuminating from this source; n is the minimum number of rays to use for the entire area. The *jitter* flag tells the ray tracer to randomize the placement of the small lights within this area rather than placing them in a rectangular grid; this helps with aliasing artifacts. Altogether, a typical area light declaration is

```
light_source {
        <4, 10, -20>
        color White
        area_light
        <1,0,1>,<0,1,0>,5,5
        adaptive 1
        jitter
}
```

Objects. Object declarations generally have a shape specification followed by a texture specification. The sphere declaration in our first example was typical of this:

```
sphere {
        <0, 1, 5>, 3
        texture {
          pigment {color Yellow }
        }
}
```

All of our objects have such a texture declation. In a subsequent section we will look in more detail at the possible texture portions of this; for now we will just use simple diffuse texture like this.

Here are the basic shapes:

- Spheres: These need a point for the center, and a value for the radius.
- Boxes: These need two points for corners across a diagonal (such as near lower left corner and far upper right corner). For example

```
box {
        <1, 2, 3>,
        <4, 5, 6>
        texture {...}
}
```

- Cones: These need 4 attributes: the center and radius of one end of the cone, and the center and radius of the other end. Use 0.0 as a radius if you want to include the vertex of the cone. For example

```
cone {
        <0,0,0>, 1
        <4, 0, 0>, 3
        texture {...}
}
```

- Cylinders: These need 3 attributes: the center of each end and the radius. By default cylinders include their endcaps; if you don't want the endcaps add the keyword *open*, as in

```
cylinder {
        <0,0,0>,
        <4,0,0>,
        1.0
        open
        texture {....}
}
```

- Planes: The plane ax+by+cz=d is specified by the normal <a, b, c> and the constant d. For example,

```
plane {
        <0, 1, 0>, -1
        texture {...}
}
```

is the horizontal plane y=-1.

- Polygons: A polygon is specified by a vertex count followed by a list of vertices. For example

```
polygon {
        4,
        <0,0,0>, <0,1,0>,<1,1,0>,<1,0,0>
        texture {...}
}
```

- Torus: A torus centered at the origin needs only two parameters: the major and minor radii, as in

```
torus {
        4, 1
        texture { ... }
}
```

Positioning the torus away from the origin requires using transformations, which we discuss below.

Textures and Colors. POV-Ray has an extensive set of options for surface materials. We will only scratch the surface (so to speak) here. The base color or color patter on an object is set in a field called "pigment". The simplest texture is just a solid pigment:

```
texture { pigment {color <1,1,0>} }
```

makes a yellow surface. There are several predefined options for patterns:

```
pigment {checker color1, color2}
```

makes a checkerboard pattern on the surface, using the (x,y,z) coordinates of the surface

```
pigment {brick color1, color2}
```

makes a brick pattern, with color 1 as the outline or mortar color, color2 as the rectangular "brick" color.

```
pigment {hexagon color1, color2, color3}
```

tiles the surface with hexagons in the three colors.

There are many other possible options for pigments. The only one we will discuss here is a "texture map". This has a series of textures that can be applied to changes in coordinates along any of the three axes. The axis is specified by a *gradient* declaration: `gradient x`, `gradient y`, or `gradient z`. The textures are specified by entries `[alpha tex]`, where `alpha` is a value between 0 and 1 and tex is anything that can be specified in a texture declaration. If we put these in order of increasing alpha, the first texture is displayed for values of the gradient between 0 and the first alpha. The last texture is displayed for values of the gradient between the last alpha and 1. In between the textures are taken in pairs and the blended. For example, the following will ramp between lines of white and black:

```
texture {
    gradient x
    texture_map {
        [0.0 pigment {color <0,0,0> }]
        [0.5 pigment {color <1,1,1>}]
        [1.0 pigment {color <0,0,0>}]
    }
}
```

The following will make alternating red and blue stripes with no blending:

```
texture {
    gradient x
    texture_map {
        [0.5 pigment {color <1,0,0> }]
        [0.5 pigment {color <0,0,1>}]
    }
}
```

Here we have 4 stripes in colors red, yellow, blue and green:

```
texture {
        gradient x
        texture_map {
                [0.25 pigment {color <1,0,0> }]
                [0.25 pigment {color <1,1,0> }]
                [0.5  pigment {color <1,1,0> }]
                [0.5  pigment {color <0,0,1> }]
                [0.75  pigment {color <0,0,1> }]
                [0.75 pigment {color <0,1,0> }]
        }
}
```

Pigments give the basic color of the surface. The light reflection model used for an object is determinted by the *finish* of the object's texture. Here is a typical finish declaration as part of an overall texture:

```
texture {
        pigment {color <1,1,0>}
        finish {
                ambient 0.1
                diffuse 0.8
                phong 0.3
                phong _size 30  // phong exponent
        }
}
```

The amounts given after each of the reflection types are the amounts of light reflected in this way; the phong_size parameter is what we called the "phong exponent" in class.

There are a few other options. There is a *brilliance* parameter that can make surfaces appear more metallic. The default value of brilliance is 1.0; adding the line brilliance amount to a finish declaration will alter the brilliance to the given amount. For metallic surfaces use a brilliance factor between 5 and 10. An alternative to the phong model is a more realistic model called *specular* in POV-Ray. This sets a specular amount and also a value of an attribute called *roughness* that affects the size of the specular highlight. Using this we might have finish declaration

```
        finish {
                diffuse 0.7
                specular 0.3
                roughness 0.03
        }
```

Roughness values should be between 1.0 very rough, giving a large, dim highlight to 0.0005 (very smooth with a small, bright highlight).
If you look closely at a shiny metallic surface in a bright light, you may notice that the hightlights are colored rather than white. With either the phong or the specular models for specular reflection you can add the directive *metallic*. This causes the POV-Ray shader to alter the colors of the highlights to those more resembling reflections from real metallic surfaces.

Both the Phong and specular portions of the finish refer only to highlights. If you want mirror reflections, add the parameter reflection to the model in the form

reflection amount

The amount is the portion of the light coming into the surface that is reflected in the form of a specular mirror. In theory the ambient, diffuse, specular, and reflection amounts for a surface sum to 1, though there is no requirement that this be the case.

If you want to make transparent items there are two steps to take. First, you need to create a transparent color. Colors in POV-Ray are actually 5-tuples: <r,g,b,f,t>. The first three dimensions are the usual RGB color scale, with all three values numbers between 0 and 1. The fourth dimension, f, is for *filtered* transparency – the amount of light reaching the object that passes through and is altered or filtered by the object's color. The t-dimension is for *transmitted* transparency – light that reaches the object and does not pick up the object's color. For example, a light green glass object might have color <0.9,1,0.9,0.9,0> -- light green, and filtering almost all the light reaching it.

The other step to creating transparent items is to set the index of refraction. This is an *interior* property in POV-Ray; there are a number of properties that can be set for the interior of an object, just as the texture sets properties for the object's exterior. The index of refraction property is ior. The index of refraction of glass is 1.5. Thus, the following two lines set the properties for a greenish glass:

```
texture {pigment {color <0.9, 1, 0.9, 0.9, 0>}}
interior {ior 1.5}
```

CSG. You may make intersections, unions, and differences of objects. In most situations you will want to give texture properties to the combined object rather than the individual components. For example

```
intersection {
        sphere {<0, 1, 5>, 3}
        sphere {<3, 1, 5>, 3}
        texture { .... }
}
```

The keyword *intersection* may be replaced by *union* or *difference*. You can intersect or union any number of objects; differences need to be between two objects.

Variables. Scene descriptions can become very elaborate and difficult to follow. Comments help, but it also helps to work some English into the scene description itself. One way to do this is to use variables to attach names to parts of objects. The declaration of a variable is

```
#declare <name> = <description>;
```

For example,

```
#declare Ball=sphere{ <1,2,3>, 4};
```

Note that the declaration is terminated by a semicolon; don't leave this off. At the top level you can use variables as replacements for their declarations without any adornment. In CSG constructions you need to unpack the declaration with an *object* phrase, as in the following:

```
#declare Ball = sphere {<0, 1, 5>, 3};
#declare Ball2 = sphere {<3, 1, 5>,3};
union {
        object {Ball}
        object {Ball2}
        texture {…}
}
```

Transformations. POV-Ray has the usual affine transformations: translation, rotation, and scaling. All three take vector arguments:

```
translate <a, b, c>
```

moves the current object by vector <a,b,c>

```
scale <a, b, c>
```

scales by a on the x-axis, b on the y-axis and c on the z-axis.

```
rotate <a,b,c>
```

rotates by a degrees around the x-axis, b degrees around the y-axis, and c degrees around z. POV-Ray defines the vectors x, y and z as <1,0,0>, <0,1,0> and <0,0,1>, so we could rotate by 30 degrees around the x-axis with either

```
rotate <30,0,0>
```

or

```
rotate 30*x
```

These transformations generally go inside object definitions. They apply to everything in the object definition prior to the transformation. If you have a texture definition that is sensitive to location, you probably want to put a translation or rotation statement after the texture declaration because you usually want the transformation to apply to the texture as well.

```
#declare Ball = sphere {<0, 1, 5>, 3};
object {Ball texture {pigment {color Red}}}
object {Ball texture {pigment {color Blue}}
        translate <0,-1,-5>
        scale<1,1.2,1.5>
         translate <3,0,5>
}
```

Examples. We finish with three examples. The first shows a room with a table on top of which is a ball and a silver cone. The only light is coming in from an unseen window (i.e, and area light) to the left of the viewer.

```
#include "colors.inc"
#include "textures.inc"


background { color <0.6,0.6,0> }

//light_source { <10, 7.8, 2> color <0.5,0.5,0.5>}
light_source { <0.1, 5, 1> color <1,1,1> area_light <0,0,1> <0,1,0> 10, 10}

camera {
      location <5, 6,1 >
      look_at <6, 3, 10>
     }




#declare Backwall =  polygon {4 <0,0,10>, <0,8,10>, <15,8,10>,<15,0,10>};
#declare Frontwall =  polygon {4 <0,0,0>, <0,8,0>, <15,8,0>,<15,0,0>};
#declare Ceiling = polygon {4 <0,8,10>, <15,8,10>, <15,8,0>, <0,8,0>};
#declare Floor = polygon { 4 <0,0,10>, <15,0,10>,<15,0,0>, <0,0,0>};
#declare Tabletop = box {<4,4,4>,<7,4.1,6>};
#declare Tableleg = box {<-0.1, 0,-0.1>,<0.1,4,0.1>};
#declare Mirror = polygon{4 <3,4,9.9>,<3,7,9.9>,<6,7,9.9>,<6,4,9.9>};

union {
      object {Backwall}
      object {Frontwall}
      texture {pigment {color Yellow}}
      }

object {Ceiling
      texture {pigment {color White}
      finish {ambient 0.3 diffuse 0.9}}
      }


object {Floor
      texture {pigment {color Green}}
     }
```

```
object {Mirror
      texture {pigment {color Silver}}
      finish { diffuse 0.2 specular 0.4 reflection {1.0}} }

object {Tabletop texture {DMFDarkOak scale <0.3,0.3,0.3> rotate 90*y}}

union {
      object {Tableleg translate <4.1,0,4.1>}
      object {Tableleg translate <6.9,0,4.1>}
      object {Tableleg translate <4.1,0,5.9>}
      object {Tableleg translate <6.9,0,5.9>}
      texture {DMFDarkOak scale <0.3,0.3,0.3>rotate 90*x}
     }

sphere {<4.6,4.3,4.5>,0.2 texture {pigment {color Red} finish {diffuse 0.5 specular 0.8}}}
cone {<5.2,4.1,5.2>, 0.3, <5.2,4.6,5.2> 0 texture {pigment {color Silver}
      finish {diffuse 0.5 specular 0.8 roughness 0.001 metallic}} }
```

The second example shows a Greek temple. The floor is an octagon; on top of this there are columns supporting a dome. A stone texture is applied to the entire temple.

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



background { color Cyan }

camera {
        location <1,10, -40>
        look_at <1,12, 0>
}


light_source { <0, 15, 0> color <0.2,0.2,0.2>}
light_source { <20, 40, -10> color <1,1,1> area_light <0,2,0>, <0,0,2>,5,5}


#declare Vx = 20*cos( radians(22.5) );
#declare Vz = 20*sin( radians(22.5) );

#declare Box = box {<-Vx, 0, -Vz> <Vx, 1, Vz>};
```

```
#declare Octagon =
      union {object {Box}
            object {Box rotate 45*y}
            object {Box rotate -45*y}
            object {Box rotate 90*y}
      };


#declare Base =
      union {
            object {Octagon}
            object {Octagon scale <0.9,1,0.9> translate <0,1,0>}
            object {Octagon scale <0.81,1,0.81> translate <0,2,0>}
     };

#declare Column =
      union {
            cylinder {
                  <0,3,0>, <0,18,0>, 0.7 }
            box {<-1,3,-1> <1,4,1>}
            torus {0.7,0.3 translate <0,4,0> }
            box {<-1,17,-1> <1,18,1>}
            torus {0.7,0.3 translate <0,17,0>}
            translate <14.5,0,0> rotate 22.5*y
     };

#declare Dome =
      union {
            torus {15, 0.7}
            intersection {
                  difference {
                        sphere {<0,0,0> 15 }
                        sphere {<0,0,0> 14.5}
                  }
                  box {<-30,0,-30> <30,16,30>}
            }
            scale <1,0.7,1>
            translate <0,18.2,0>
      };

#declare Temple =
      union {
```

```
            object {Base}
            object {Column}
            object {Column rotate 45*y}
            object {Column rotate 90*y}
            object {Column rotate 135*y}
            object {Column rotate 180*y}
            object {Column rotate 225*y}
            object {Column rotate 270*y}
            object {Column rotate 315*y}
            object {Dome}
      }

object {Temple texture {T_Stone12
                  }
                  finish {ambient 0.1 diffuse 0.8 specular 0.3 roughness 0.5}
      }

object {plane {<0,1,0> 0 texture {
                           pigment {color Green}
                         }
           }
      }
```

The final example shows three cylinders on top of a yellow plane. In the background are red and blue cylinders. In the foreground is a green hollow cylinder (the difference between two cylinders) with a high degree of transparency.

```
#include "colors.inc"

camera {
       location <5,3,-8>
       look_at <5,3,0>
      }

light_source {<7, 30, -20> color White }
plane {<0,1,0> 0 texture {pigment {color Yellow} } }

cylinder {<5,0,6> <5,5,6> 1 texture {pigment {color Blue}}}
cylinder {<3,0,6> <3,5,6> 1 texture {pigment {color Red}}}
difference {
       cylinder {<4.5,0,1><4.5,3,1> 1 }
       cylinder {<4.5,0.1,1><4.5,3,1> 0.9 }
       texture {pigment {color <0.9,1,0.9,0.9,0>}} interior {ior 1.5} }
```