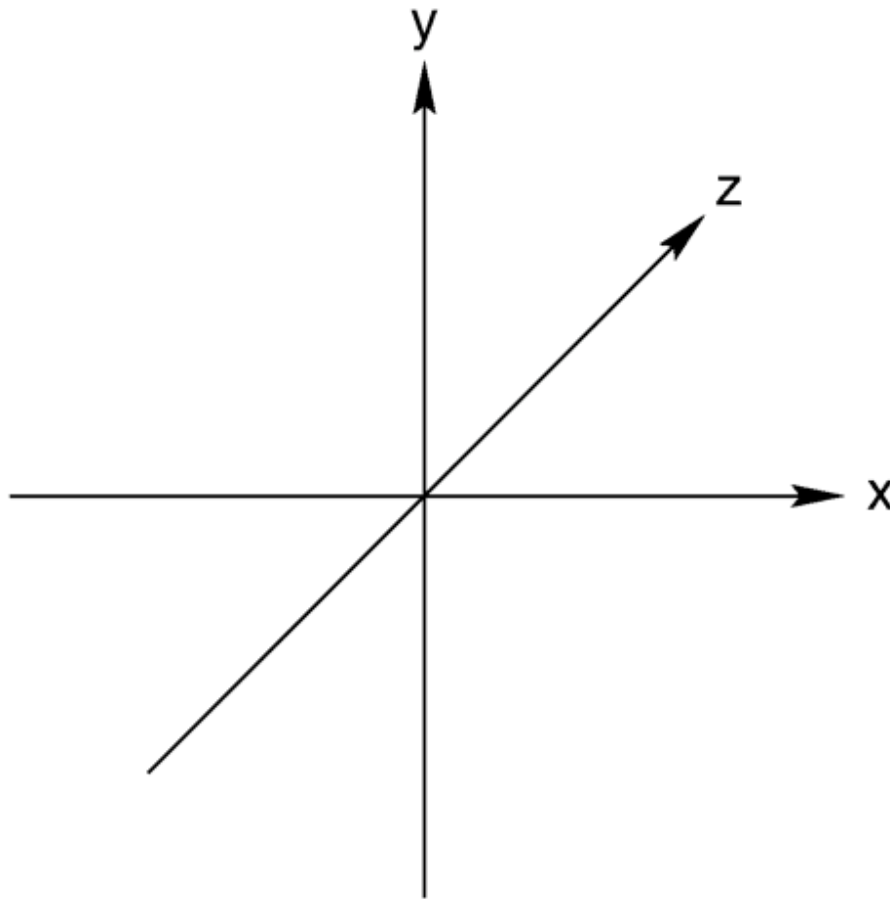


## POV-Ray Mini-Manual (excerpt from POVray documentation)

### 2.2.1.1 Understanding POV-Ray's Coordinate System

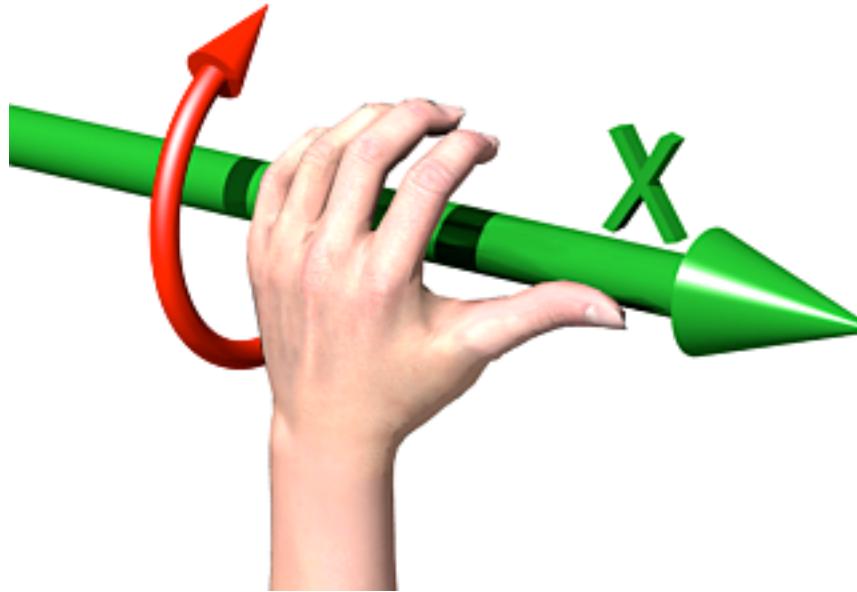
First, we have to tell POV-Ray where our camera is and where it is looking. To do this, we use 3D coordinates. The usual coordinate system for POV-Ray has the positive y-axis pointing up, the positive x-axis pointing to the right, and the positive z-axis pointing into the screen as follows:



This kind of coordinate system is called a left-handed coordinate system. If we use our left hand's fingers we can easily see why it is called left-handed. We just point our thumb in the direction of the positive x-axis (to the right), the index finger in the direction of the positive y-axis (straight up) and the middle finger in the positive z-axis direction (forward). We can only do this with our left hand. If we had used our right hand we would not have been able to point the middle finger in the correct direction.

The left hand can also be used to determine rotation directions. To do this we must perform the famous "*Computer Graphics Aerobics*" exercise. We hold our left hand

and point our thumb in the positive direction of the axis of rotation. Our fingers will curl in the positive direction of rotation. Similarly if we point our thumb in the negative direction of the axis our fingers will curl in the negative direction of rotation.



In the above illustration, the left hand is curling around the x-axis. The thumb points in the positive x direction and the fingers curl over in the positive rotation direction.

### 2.2.1.3 Adding a Camera

The camera statement describes where and how the camera sees the scene. It gives x-, y- and z-coordinates to indicate the position of the camera and what part of the scene it is pointing at. We describe the coordinates using a three-part vector. A vector is specified by putting three numeric values between a pair of angle brackets and separating the values with commas. We add the following camera statement to the scene.

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

Briefly, location  $\langle 0, 2, -3 \rangle$  places the camera up two units and back three units from the center of the ray-tracing universe which is at  $\langle 0, 0, 0 \rangle$ . By default +z is into the screen and -z is back out of the screen.

Also look\_at  $\langle 0, 1, 2 \rangle$  rotates the camera to point at the coordinates  $\langle 0, 1, 2 \rangle$ . A point 1 unit up from the origin and 2 units away from the origin. This makes it 5 units in front of

and 1 unit lower than the camera. The look\_at point should be the center of attention of our image.

### 2.2.1.4 Describing an Object

Now that the camera is set up to record the scene, let's place a yellow sphere into the scene. We add the following to our scene file:

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

The first vector specifies the center of the sphere. In this example the x coordinate is zero so it is centered left and right. It is also at y=1 or one unit up from the origin. The z coordinate is 2 which is five units in front of the camera, which is at z=-3. After the center vector is a comma followed by the radius which in this case is two units. Since the radius is half the width of a sphere, the sphere is four units wide.

### 2.2.1.5 Adding Texture to an Object

After we have defined the location and size of the sphere, we need to describe the appearance of the surface. The texture statement specifies these parameters. Texture blocks describe the color, bumpiness and finish properties of an object. In this example we will specify the color only. This is the minimum we must do. All other texture options except color will use default values.

The color we define is the way we want an object to look if fully illuminated. If we were painting a picture of a sphere we would use dark shades of a color to indicate the shadowed side and bright shades on the illuminated side. However ray-tracing takes care of that for you. We only need to pick the basic color inherent in the object and POV-Ray brightens or darkens it depending on the lighting in the scene. Because we are defining the basic color the object actually **has** rather than how it **looks** the parameter is called pigment.

Many types of color patterns are available for use in a pigment statement. The keyword color specifies that the whole object is to be one solid color rather than some pattern of colors. We can use one of the color identifiers previously defined in the standard include file colors.inc.

### 2.2.1.6 Defining a Light Source

One more detail is needed for our scene. We need a light source. Until we create one, there is no light in this virtual world. Thus we add the line

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

to the scene file to get our first complete POV-Ray scene file as shown below.

```
#include "colors.inc"
background { color Cyan }
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 }
```

The vector in the `light_source` statement specifies the location of the light as two units to our right, four units above the origin and three units back from the origin. The light source is an invisible tiny point that emits light. It has no physical shape, so no texture is needed.

That's it! We close the file and render a small picture of it using whatever methods you used for your particular platform. If you specified a preview display it will appear on your screen. If you specified an output file (the default is `file output on`), then POV-Ray also created a file.

## 2.2.2 Basic Shapes

So far we have just used the sphere shape. There are many other types of shapes that can be rendered by POV-Ray. The following sections will describe how to use some of the more simple objects as a replacement for the sphere used above.

### 2.2.2.1 Box Object

The box is one of the most common objects used. We try this example in place of the sphere:

```
box {
  <-1, 0, -1>, // Near lower left corner
  < 1, 0.5, 3> // Far upper right corner
  texture {
    T_Stone25 // Pre-defined from stones.inc
    scale 4 // Scale by the same amount in all
             // directions
  }
  rotate y*20 // Equivalent to "rotate <0,20,0>"
}
```

In the example we can see that a box is defined by specifying the 3D coordinates of its opposite corners. The first vector is generally the minimum x-, y- and z-coordinates and the 2nd vector should be the maximum x-, y- and z-values however any two opposite corners may be used. Box objects can only be defined parallel to the axes of the world coordinate system. We can later rotate them to any angle.

**Note:** we can perform simple math on values and vectors. In the rotate parameter we multiplied the vector identifier y by 20. This is the same as  $\langle 0,1,0 \rangle * 20$  or  $\langle 0,20,0 \rangle$ .

### 2.2.2.2 Cone Object

Here's another example showing how to use a cone:

```
cone {
  <0, 1, 0>, 0.3 // Center and radius of one end
  <1, 2, 3>, 1.0 // Center and radius of other end
  texture { T_Stone25 scale 4 }
}
```

The cone shape is defined by the center and radius of each end. In this example one end is at location  $\langle 0,1,0 \rangle$  and has a radius of 0.3 while the other end is centered at  $\langle 1,2,3 \rangle$  with a radius of 1. If we want the cone to come to a sharp point we must use radius=0. The solid end caps are parallel to each other and perpendicular to the cone axis. If we want an open cone with no end caps we have to add the keyword open after the 2nd radius like this:

```
cone {
  <0, 1, 0>, 0.3 // Center and radius of one end
  <1, 2, 3>, 1.0 // Center and radius of other end
```

```
open          // Removes end caps
texture { T_Stone25 scale 4 }
}
```

### 2.2.2.3 Cylinder Object

We may also define a cylinder like this:

```
cylinder {
  <0, 1, 0>, // Center of one end
  <1, 2, 3>, // Center of other end
  0.5       // Radius
  open      // Remove end caps
  texture { T_Stone25 scale 4 }
}
```

### 2.2.2.4 Plane Object

Let's try out a computer graphics standard "*The Checkered Floor*". We add the following object to the first version of the demo.pov file, the one including the sphere.

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

The object defined here is an infinite plane. The vector  $\langle 0, 1, 0 \rangle$  is the surface normal of the plane (i.e. if we were standing on the surface, the normal points straight up). The number afterward is the distance that the plane is displaced along the normal from the origin -- in this case, the floor is placed at  $y = -1$  so that the sphere at  $y = 1$ , radius = 2, is resting on it.

**Note:** even though there is no texture statement there is an implied texture here. We might find that continually typing statements that are nested like `texture {pigment}` can get to be tiresome so POV-Ray lets us leave out the `texture` statement under many circumstances. In general we only need the texture block surrounding a texture identifier (like the `T_Stone25` example above), or when creating layered textures (which are covered later).

This pigment uses the checker color pattern and specifies that the two colors red and blue should be used.

Because the vectors  $\langle 1,0,0 \rangle$ ,  $\langle 0,1,0 \rangle$  and  $\langle 0,0,1 \rangle$  are used frequently, POV-Ray has three built-in vector identifiers `x`, `y` and `z` respectively that can be used as a shorthand. Thus the plane could be defined as:

```
plane { y, -1
  pigment { ... }
}
```

**Note:** that we do not use angle brackets around vector identifiers.

Looking at the floor, we notice that the ball casts a shadow on the floor. Shadows are calculated very accurately by the ray-tracer, which creates precise, sharp shadows. In the real world, penumbral or "soft" shadows are often seen. Later we will learn how to use [extended light sources](#) to soften the shadows.

### 2.2.2.5 Torus Object

A torus can be thought of as a donut or an inner-tube. It is a shape that is vastly useful in many kinds of CSG so POV-Ray has adopted this 4th order quartic polynomial as a primitive shape. The syntax for a torus is so simple that it makes it a very easy shape to work with once we learn what the two float values mean. Instead of a lecture on the subject, let's create one and do some experiments with it.

We create a file called `tordemo.pov` and edit it as follows:

```
#include "colors.inc"
camera {
  location <0, .1, -25>
  look_at 0
  angle 30
}
background { color Gray50 } // to make the torus easy to see
light_source { <300, 300, -1000> White }
torus {
  4, 1 // major and minor radius
  rotate -90*x // so we can see it from the top
  pigment { Green }
}
```

We trace the scene. Well, it's a donut alright. Let's try changing the major and minor radius values and see what happens. We change them as follows:

```
torus { 5, .25 // major and minor radius
```

That looks more like a hula-hoop! Let's try this:

```
torus { 3.5, 2.5 // major and minor radius
```

Whoa! A donut with a serious weight problem!

With such a simple syntax, there isn't much else we can do to a torus besides change its texture... or is there? Let's see...

Tori are very useful objects in CSG. Let's try a little experiment. We make a difference of a torus and a box:

```
difference {
  torus {
    4, 1
    rotate x*-90 // so we can see it from the top
  }
  box { <-5, -5, -1>, <5, 0, 1> }
  pigment { Green }
}
```

Interesting... a half-torus. Now we add another one flipped the other way. Only, let's declare the original half-torus and the necessary transformations so we can use them again:

```
#declare Half_Torus = difference {
  torus {
    4, 1
    rotate -90*x // so we can see it from the top
  }
  box { <-5, -5, -1>, <5, 0, 1> }
  pigment { Green }
}
#declare Flip_It_Over = 180*x;
#declare Torus_Translate = 8; // twice the major radius
```

Now we create a union of two Half\_Torus objects:

```
union {
  object { Half_Torus }
  object { Half_Torus
    rotate Flip_It_Over
```

```

    translate Torus_Translate*x
  }
}

```

This makes an S-shaped object, but we can't see the whole thing from our present camera. Let's add a few more links, three in each direction, move the object along the +z-direction and rotate it about the +y-axis so we can see more of it. We also notice that there appears to be a small gap where the half Tori meet. This is due to the fact that we are viewing this scene from directly on the x-z-plane. We will change the camera's y-coordinate from 0 to 0.1 to eliminate this.

```

union {
  object { Half_Torus }
  object { Half_Torus
    rotate Flip_It_Over
    translate x*Torus_Translate
  }
  object { Half_Torus
    translate x*Torus_Translate*2
  }
  object { Half_Torus
    rotate Flip_It_Over
    translate x*Torus_Translate*3
  }
  object { Half_Torus
    rotate Flip_It_Over
    translate -x*Torus_Translate
  }
  object { Half_Torus
    translate -x*Torus_Translate*2
  }
  object { Half_Torus
    rotate Flip_It_Over
    translate -x*Torus_Translate*3
  }
  object { Half_Torus
    translate -x*Torus_Translate*4
  }
  rotate y*45
  translate z*20
}

```

Rendering this we see a cool, undulating, snake-like something-or-other. Neato. But we want to model something useful, something that we might see in real life. How about a chain?

Thinking about it for a moment, we realize that a single link of a chain can be easily modeled using two half tori and two cylinders. We create a new file. We can use the same camera, background, light source and declared objects and transformations as we used in `tordemo.pov`:

```
#include "colors.inc"
camera {
  location <0, .1, -25>
  look_at 0
  angle 30
}
background { color Gray50 }
light_source{ <300, 300, -1000> White }
#declare Half_Torus = difference {
  torus {
    4,1
    sturm
    rotate x*-90 // so we can see it from the top
  }
  box { <-5, -5, -1>, <5, 0, 1> }
  pigment { Green }
}
#declare Flip_It_Over = x*180;
#declare Torus_Translate = 8;
```

Now, we make a complete torus of two half tori:

```
union {
  object { Half_Torus }
  object { Half_Torus rotate Flip_It_Over }
}
```

This may seem like a wasteful way to make a complete torus, but we are really going to move each half apart to make room for the cylinders. First, we add the declared cylinder before the union:

```
#declare Chain_Segment = cylinder {
  <0, 4, 0>, <0, -4, 0>, 1
```

```
pigment { Green }
}
```

We then add two Chain\_Segments to the union and translate them so that they line up with the minor radius of the torus on each side:

```
union {
  object { Half_Torus }
  object { Half_Torus rotate Flip_It_Over }
  object { Chain_Segment translate x*Torus_Translate/2 }
  object { Chain_Segment translate -x*Torus_Translate/2 }
}
```

Now we translate the two half tori +y and -y so that the clipped ends meet the ends of the cylinders. This distance is equal to half of the previously declared Torus\_Translate:

```
union {
  object {
    Half_Torus
    translate y*Torus_Translate/2
  }
  object {
    Half_Torus
    rotate Flip_It_Over
    translate -y*Torus_Translate/2
  }
  object {
    Chain_Segment
    translate x*Torus_Translate/2
  }
  object {
    Chain_Segment
    translate -x*Torus_Translate/2
  }
}
```

We render this and voila! A single link of a chain. But we aren't done yet! Whoever heard of a green chain? We would rather use a nice metallic color instead. First, we remove any pigment blocks in the declared tori and cylinders. Then we add a declaration for a golden texture just before the union that creates the link. Finally, we add the texture to the union and declare it as a single link:

```
#declare Half_Torus = difference {
```

```
torus {
  4,1
  sturm
  rotate x*-90 // so we can see it from the top
}
box { <-5, -5, -1>, <5, 0, 1> }
}
```

```
#declare Chain_Segment = cylinder {
  <0, 4, 0>, <0, -4, 0>, 1
}
```

```
#declare Chain_Gold = texture {
  pigment { BrightGold }
  finish {
    ambient .1
    diffuse .4
    reflection .25
    specular 1
    metallic
  }
}
```

```
#declare Link = union {
  object {
    Half_Torus
    translate y*Torus_Translate/2
  }
  object {
    Half_Torus
    rotate Flip_It_Over
    translate -y*Torus_Translate/2
  }
  object {
    Chain_Segment
    translate x*Torus_Translate/2
  }
  object {
    Chain_Segment
    translate -x*Torus_Translate/2
  } texture { Chain_Gold }
}
```

Now we make a union of two links. The second one will have to be translated +y so that its inner wall just meets the inner wall of the other link, just like the links of a chain. This distance turns out to be double the previously declared Torus\_Translate minus 2 (twice the minor radius). This can be described by the expression:

```
Torus_Translate*2-2*y
```

We declare this expression as follows:

```
#declare Link_Translate = Torus_Translate*2-2*y;
```

In the object block, we will use this declared value so that we can multiply it to create other links. Now, we rotate the second link 90\*y so that it is perpendicular to the first, just like links of a chain. Finally, we scale the union by 1/4 so that we can see the whole thing:

```
union {  
  object { Link }  
  object { Link translate y*Link_Translate rotate y*90 }  
  scale .25  
}
```

We render this and we will see a very realistic pair of links. If we want to make an entire chain, we must declare the above union and then create another union of this declared object. We must be sure to remove the scaling from the declared object:

```
#declare Link_Pair =  
union {  
  object { Link }  
  object { Link translate y*Link_Translate rotate y*90 }  
}
```

Now we declare our chain:

```
#declare Chain = union {  
  object { Link_Pair }  
  object { Link_Pair translate y*Link_Translate*2 }  
  object { Link_Pair translate y*Link_Translate*4 }  
  object { Link_Pair translate y*Link_Translate*6 }  
  object { Link_Pair translate -y*Link_Translate*2 }  
  object { Link_Pair translate -y*Link_Translate*4 }  
  object { Link_Pair translate -y*Link_Translate*6 }  
}
```

}

And finally we create our chain with a couple of transformations to make it easier to see. These include scaling it down by a factor of 1/10, and rotating it so that we can clearly see each link:

```
object { Chain scale .1 rotate <0, 45, -45> }
```



We render this and we should see a very realistic gold chain stretched diagonally across the screen.

## .2.3 CSG Objects

*You know you have been raytracing too long when ...  
... Your friends are used to the fact that you will suddenly stop walking in order to look at objects and figure out how to do them as CSGs.  
-- Jeff Lee*

Constructive Solid Geometry, or CSG, is a powerful tool to combine primitive objects to create more complex objects as shown in the following sections.

### 2.2.3.1 What is CSG?

CSG stands for *Constructive Solid Geometry*. POV-Ray allows us to construct complex solids by combining primitive shapes in four different ways. In the [union](#) statement, two or more shapes are added together. With the [intersection](#) statement, two or more shapes are combined to make a new shape that consists of the area common to both shapes. The [difference](#) statement, an initial shape has all subsequent shapes subtracted from it.

And last but not least [merge](#), which is like a union where the surfaces inside the union are removed (useful in transparent CSG objects). We will deal with each of these in detail in the next few sections.

CSG objects can be extremely complex. They can be deeply nested. In other words there can be unions of differences or intersections of merges or differences of intersections or even unions of intersections of differences of merges... ad infinitum. CSG objects are (almost always) finite objects and thus respond to auto-bounding and can be transformed like any other POV primitive shape.

### 2.2.3.2 CSG Union

Let's try making a simple union. Create a file called `csgdemo.pov` and edit it as follows:

```
#include "colors.inc"
camera {
  location <0, 1, -10>
  look_at 0
  angle 36
}
light_source { <500, 500, -1000> White }
plane { y, -1.5
  pigment { checker Green White }
}
```

Let's add two spheres each translated 0.5 units along the x-axis in each direction. We color one blue and the other red.

```
sphere { <0, 0, 0>, 1
  pigment { Blue }
  translate -0.5*x
}
sphere { <0, 0, 0>, 1
  pigment { Red }
```

```
    translate 0.5*x
}
```

We trace this file and note the results. Now we place a union block around the two spheres. This will create a single CSG union out of the two objects.

```
union{
  sphere { <0, 0, 0>, 1
    pigment { Blue }
    translate -0.5*x
  }
  sphere { <0, 0, 0>, 1
    pigment { Red }
    translate 0.5*x
  }
}
```

We trace the file again. The union will appear no different from what each sphere looked like on its own, but now we can give the entire union a single texture and transform it as a whole. Let's do that now.

```
union{
  sphere { <0, 0, 0>, 1
    translate -0.5*x
  }
  sphere { <0, 0, 0>, 1
    translate 0.5*x
  }
  pigment { Red }
  scale <1, .25, 1>
  rotate <30, 0, 45>
}
```

We trace the file again. As we can see, the object has changed dramatically. We experiment with different values of scale and rotate and try some different textures.

There are many advantages of assigning only one texture to a CSG object instead of assigning the texture to each individual component. First, it is much easier to use one texture if our CSG object has a lot of components because changing the objects appearance involves changing only one single texture. Second, the file parses faster because the texture has to be parsed only once. This may be a great

factor when doing large scenes or animations. Third, using only one texture saves memory because the texture is only stored once and referenced by all components of the CSG object. Assigning the texture to all n components means that it is stored n times.

### 2.2.3.3 CSG Intersection

Now let's use these same spheres to illustrate the next kind of CSG object, the intersection. We change the word union to intersection and delete the scale and rotate statements:

```
intersection {
  sphere { <0, 0, 0>, 1
    translate -0.5*x
  }
  sphere { <0, 0, 0>, 1
    translate 0.5*x
  }
  pigment { Red }
}
```

We trace the file and will see a lens-shaped object instead of the two spheres. This is because an intersection consists of the area shared by both shapes, in this case the lens-shaped area where the two spheres overlap. We like this lens-shaped object so we will use it to demonstrate differences.

### 2.2.3.4 CSG Difference

We rotate the lens-shaped intersection about the y-axis so that the broad side is facing the camera.

```
intersection{
  sphere { <0, 0, 0>, 1
    translate -0.5*x
  }
  sphere { <0, 0, 0>, 1
    translate 0.5*x
  }
  pigment { Red }
  rotate 90*y
}
```

Let's create a cylinder and stick it right in the middle of the lens.

```
cylinder { <0, 0, -1> <0, 0, 1>, .35
  pigment { Blue }
}
```

We render the scene to see the position of the cylinder. We will place a difference block around both the lens-shaped intersection and the cylinder like this:

```
difference {
  intersection {
    sphere { <0, 0, 0>, 1
      translate -0.5*x
    }
    sphere { <0, 0, 0>, 1
      translate 0.5*x
    }
    pigment { Red }
    rotate 90*y
  }
  cylinder { <0, 0, -1> <0, 0, 1>, .35
    pigment { Blue }
  }
}
```

We render the file again and see the lens-shaped intersection with a neat hole in the middle of it where the cylinder was. The cylinder has been subtracted from the intersection. Note that the pigment of the cylinder causes the surface of the hole to be colored blue. If we eliminate this pigment the surface of the hole will be black, as this is the default color if no color is specified.

OK, let's get a little wilder now. Let's declare our perforated lens object to give it a name. Let's also eliminate all textures in the declared object because we will want them to be in the final union instead.

```
#declare Lens_With_Hole = difference {
  intersection {
    sphere { <0, 0, 0>, 1
      translate -0.5*x
    }
  }
  sphere { <0, 0, 0>, 1
    translate 0.5*x
  }
}
```

```

    }
    rotate 90*y
  }
  cylinder { <0, 0, -1> <0, 0, 1>, .35 }
}

```

Let's use a union to build a complex shape composed of copies of this object.

```

union {
  object { Lens_With_Hole translate <-.65, .65, 0> }
  object { Lens_With_Hole translate <.65, .65, 0> }
  object { Lens_With_Hole translate <-.65, -.65, 0> }
  object { Lens_With_Hole translate <.65, -.65, 0> }
  pigment { Red }
}

```

We render the scene. An interesting object to be sure. But let's try something more. Let's make it a partially-transparent object by adding some filter to the pigment block.

```

union {
  object { Lens_With_Hole translate <-.65, .65, 0> }
  object { Lens_With_Hole translate <.65, .65, 0> }
  object { Lens_With_Hole translate <-.65, -.65, 0> }
  object { Lens_With_Hole translate <.65, -.65, 0> }
  pigment { Red filter .5 }
}

```

We render the file again. This looks pretty good... only... we can see parts of each of the lens objects inside the union! This is not good.

### 2.2.3.5 CSG Merge

This brings us to the fourth kind of CSG object, the merge. Merges are the same as unions, but the geometry of the objects in the CSG that is inside the merge is not traced. This should eliminate the problem with our object. Let's try it.

```

merge {
  object { Lens_With_Hole translate <-.65, .65, 0> }
  object { Lens_With_Hole translate <.65, .65, 0> }
  object { Lens_With_Hole translate <-.65, -.65, 0> }
}

```

```
object { Lens_With_Hole translate <.65, -.65, 0> }
pigment { Red filter .5 }
}
```

Sure enough, it does!

### 2.2.3.6 CSG Pitfalls

There is a severe pitfall in the CSG code that we have to be aware of.

#### 2.2.3.6.1 Co-incident Surfaces

POV-Ray uses inside/outside tests to determine the points at which a ray intersects a CSG object. A problem arises when the surfaces of two different shapes coincide because there is no way (due to the computer's floating-point accuracy) to tell whether a point on the coincident surface belongs to one shape or the other.

Look at the following example where a cylinder is used to cut a hole in a larger box.

```
difference {
  box { -1, 1 pigment { Red } }
  cylinder { -z, z, 0.5 pigment { Green } }
}
```

**Note:** that the vectors -1 and 1 in the box definition expand to <-1,-1,-1> and <1,1,1> respectively.

If we trace this object we see red speckles where the hole is supposed to be. This is caused by the coincident surfaces of the cylinder and the box. One time the cylinder's surface is hit first by a viewing ray, resulting in the correct rendering of the hole, and another time the box is hit first, leading to a wrong result where the hole vanishes and red speckles appear. This problem can be avoided by increasing the size of the cylinder to get rid of the coincidence surfaces. This is done by:

```
difference {
  box { -1, 1 pigment { Red } }
  cylinder { -1.001*z, 1.001*z, 0.5 pigment { Green } }
}
```

In general we have to make the subtracted object a little bit larger in a CSG difference. We just have to look for coincident surfaces and increase the subtracted object appropriately to get rid of those surfaces.

The same problem occurs in CSG intersections and is also avoided by scaling some of the involved objects.

## 2.2.4 The Light Source

*You know you have been raytracing too long when ...*

*... You take a photo course just to learn how to get the lighting right.*

*-- Christoph Rieder*

In any ray-traced scene, the light needed to illuminate our objects and their surfaces must come from a light source. There are many kinds of light sources available in POV-Ray and careful use of the correct kind can yield very impressive results. Let's take a moment to explore some of the different kinds of light sources and their various parameters.

### 2.2.4.1 The Pointlight Source

Pointlights are exactly what the name indicates. A pointlight has no size, is invisible and illuminates everything in the scene equally no matter how far away from the light source it may be (this behavior can be changed). This is the simplest and most basic light source. There are only two important parameters, location and color. Let's design a simple scene and place a pointlight source in it.

We create a new file and name it `litedemo.pov`. We edit it as follows:

```
#include "colors.inc"
#include "textures.inc"
camera {
  location <-4, 3, -9>
  look_at <0, 0, 0>
  angle 48
}
```

We add the following simple objects:

```
plane {
```

```

y, -1
texture {
  pigment {
    checker
    color rgb<0.5, 0, 0>
    color rgb<0, 0.5, 0.5>
  }
  finish {
    diffuse 0.4
    ambient 0.2
    phong 1
    phong_size 100
    reflection 0.25
  }
}
}
}
torus {
  1.5, 0.5
  texture { Brown_Agate }
  rotate <90, 160, 0>
  translate <-1, 1, 3>
}
box {
  <-1, -1, -1>, <1, 1, 1>
  texture { DMFLightOak }
  translate <2, 0, 2.3>
}
cone {
  <0,1,0>, 0, <0,0,0>, 1
  texture { PinkAlabaster }
  scale <1, 3, 1>
  translate <-2, -1, -1>
}
sphere {
  <0,0,0>,1
  texture { Sapphire_Agate }
  translate <1.5, 0, -2>
}
}

```

Now we add a pointlight:

```

light_source {
  <2, 10, -3>

```

```
    color White
}
```

We render this at 200x150 -A and see that the objects are clearly visible with sharp shadows. The sides of curved objects nearest the light source are brightest in color with the areas that are facing away from the light source being darkest. We also note that the checkered plane is illuminated evenly all the way to the horizon. This allows us to see the plane, but it is not very realistic.

### 2.2.4.2 The Spotlight Source

Spotlights are a very useful type of light source. They can be used to add highlights and illuminate features much as a photographer uses spots to do the same thing. To create a spotlight simply add the `spotlight` keyword to a regular point light. There are a few more parameters with spotlights than with pointlights. These are `radius`, `falloff`, `tightness` and `point_at`. The `radius` parameter is the angle of the fully illuminated cone. The `falloff` parameter is the angle of the **umbra** cone where the light falls off to darkness. The `tightness` is a parameter that determines the rate of the light falloff. The `point_at` parameter is just what it says, the location where the spotlight is pointing to. Let's change the light in our scene as follows:

```
light_source {
    <0, 10, -3>
    color White
    spotlight
    radius 15
    falloff 20
    tightness 10
    point_at <0, 0, 0>
}
```

We render this at 200x150 -A and see that only the objects are illuminated. The rest of the plane and the outer portions of the objects are now unlit. There is a broad falloff area but the shadows are still razor sharp. Let's try fiddling with some of these parameters to see what they do. We change the falloff value to 16 (it must always be larger than the radius value) and render again. Now the falloff is very narrow and the objects are either brightly lit or in total darkness. Now we change falloff back to 20 and change the tightness value to 100 (higher is tighter) and render again. The spotlight appears to have gotten much smaller but what has

really happened is that the falloff has become so steep that the radius actually appears smaller.

We decide that a tightness value of 10 (the default) and a falloff value of 18 are best for this spotlight and we now want to put a few spots around the scene for effect. Let's place a slightly narrower blue and a red one in addition to the white one we already have:

```
light_source {
  <10, 10, -1>
  color Red
  spotlight
  radius 12
  falloff 14
  tightness 10
  point_at <2, 0, 0>
}
light_source {
  <-12, 10, -1>
  color Blue
  spotlight
  radius 12
  falloff 14
  tightness 10
  point_at <-2, 0, 0>
}
```

Rendering this we see that the scene now has a wonderfully mysterious air to it. The three spotlights all converge on the objects making them blue on one side and red on the other with enough white in the middle to provide a balance.

### 2.2.4.3 The Cylindrical Light Source

Spotlights are cone shaped, meaning that their effect will change with distance. The farther away from the spotlight an object is, the larger the apparent radius will be. But we may want the radius and falloff to be a particular size no matter how far away the spotlight is. For this reason, cylindrical light sources are needed. A cylindrical light source is just like a spotlight, except that the radius and falloff regions are the same no matter how far from the light source our object is. The shape is therefore a cylinder rather than a cone. We can specify a cylindrical light

source by replacing the spotlight keyword with the cylinder keyword. We try this now with our scene by replacing all three spotlights with cylinder lights and rendering again. We see that the scene is much dimmer. This is because the cylindrical constraints do not let the light spread out like in a spotlight. Larger radius and falloff values are needed to do the job. We try a radius of 20 and a falloff of 30 for all three lights. That's the ticket!

#### 2.2.4.4 The Area Light Source

*You know you have been raytracing too long when ...*

*... You wear fuzzy clothing to soften your shadow.*

*-- Mark Kadela*

So far all of our light sources have one thing in common. They produce sharp shadows. This is because the actual light source is a point that is infinitely small. Objects are either in direct sight of the light, in which case they are fully illuminated, or they are not, in which case they are fully shaded. In real life, this kind of stark light and shadow situation exists only in outer space where the direct light of the sun pierces the total blackness of space. But here on Earth, light bends around objects, bounces off objects, and usually the source has some dimension, meaning that it can be partially hidden from sight (shadows are not sharp anymore). They have what is known as an *umbra*, or an area of fuzziness where there is neither total light or shade. In order to simulate these *soft* shadows, a ray-tracer must give its light sources dimension. POV-Ray accomplishes this with a feature known as an area light.

Area lights have dimension in two axis'. These are specified by the first two vectors in the area light syntax. We must also specify how many lights are to be in the array. More will give us cleaner soft shadows but will take longer to render. Usually a 3\*3 or a 5\*5 array will suffice. We also have the option of specifying an adaptive value. The `adaptive` keyword tells the ray-tracer that it can adapt to the situation and send only the needed rays to determine the value of the pixel. If `adaptive` is not used, a separate ray will be sent for every light in the area light. This can really slow things down. The higher the adaptive value the cleaner the umbra will be but the longer the trace will take. Usually an adaptive value of 1 is sufficient. Finally, we probably should use the `jitter` keyword. This tells the ray-tracer to slightly move the position of each light in the area light so that the shadows appear truly soft instead of giving us an umbra consisting of closely banded shadows.

OK, let's try one. We comment out the cylinder lights and add the following:

```

light_source {
  <2, 10, -3>
  color White
  area_light <5, 0, 0>, <0, 0, 5>, 5, 5
  adaptive 1
  jitter
}

```

This is a white area light centered at <2,10,-3>. It is 5 units (along the x-axis) by 5 units (along the z-axis) in size and has 25 (5\*5) lights in it. We have specified adaptive 1 and jitter. We render this at 200x150 -A.

Right away we notice two things. The trace takes quite a bit longer than it did with a point or a spotlight and the shadows are no longer sharp! They all have nice soft umbrae around them. Wait, it gets better.

Spotlights and cylinder lights can be area lights too! Remember those sharp shadows from the spotlights in our scene? It would not make much sense to use a 5\*5 array for a spotlight, but a smaller array might do a good job of giving us just the right amount of umbra for a spotlight. Let's try it. We comment out the area light and change the cylinder lights so that they read as follows:

```

light_source {
  <2, 10, -3>
  color White
  spotlight
  radius 15
  falloff 18
  tightness 10
  area_light <1, 0, 0>, <0, 0, 1>, 2, 2
  adaptive 1
  jitter
  point_at <0, 0, 0>
}
light_source {
  <10, 10, -1>
  color Red
  spotlight
  radius 12
  falloff 14
  tightness 10
  area_light <1, 0, 0>, <0, 0, 1>, 2, 2
}

```

```

    adaptive 1
    jitter
    point_at <2, 0, 0>
}
light_source {
    <-12, 10, -1>
    color Blue
    spotlight
    radius 12
    falloff 14
    tightness 10
    area_light <1, 0, 0>, <0, 0, 1>, 2, 2
    adaptive 1
    jitter
    point_at <-2, 0, 0>
}

```

We now have three area-spotlights, one unit square consisting of an array of four (2\*2) lights, three different colors, all shining on our scene. We render this at 200x150 -A. It appears to work perfectly. All our shadows have small, tight umbrae, just the sort we would expect to find on an object under a real spotlight.

#### 2.2.4.5 The Ambient Light Source

The *ambient light source* is used to simulate the effect of inter-diffuse reflection. If there wasn't inter-diffuse reflection all areas not directly lit by a light source would be completely dark. POV-Ray uses the ambient keyword to determine how much light coming from the ambient light source is reflected by a surface.

By default the ambient light source, which emits its light everywhere and in all directions, is pure white (rgb <1,1,1>). Changing its color can be used to create interesting effects. First of all the overall light level of the scene can be adjusted easily. Instead of changing all ambient values in every finish only the ambient light source is modified. By assigning different colors we can create nice effects like a moody reddish ambient lighting. For more details about the ambient light source see "Ambient Light".

Below is an example of a red ambient light source.

```

global_settings { ambient_light rgb<1, 0, 0> }

```

## 2.2.4.6 Light Source Specials

### 2.2.4.6.1 Using Shadowless Lights

Light sources can be assigned the shadowless keyword and no shadows will be cast due to its presence in a scene. Sometimes, scenes are difficult to illuminate properly using the lights we have chosen to illuminate our objects. It is impractical and unrealistic to apply a higher ambient value to the texture of every object in the scene. So instead, we would place a couple of fill lights around the scene. Fill lights are simply dimmer lights with the shadowless keyword that act to boost the illumination of other areas of the scene that may not be lit well. Let's try using one in our scene.

Remember the three colored area spotlights? We go back and un-comment them and comment out any other lights we have made. Now we add the following:

```
light_source {  
    <0, 20, 0>  
    color Gray50  
    shadowless  
}
```

This is a fairly dim light 20 units over the center of the scene. It will give a dim illumination to all objects including the plane in the background. We render it and see.

### 2.2.4.6.2 Assigning an Object to a Light Source

Light sources are invisible. They are just a location where the light appears to be coming from. They have no true size or shape. If we want our light source to be a visible shape, we can use the looks\_like keyword. We can specify that our light source can look like any object we choose. When we use looks\_like, then no\_shadow is applied to the object automatically. This is done so that the object will not block any illumination from the light source. If we want some blocking to occur (as in a lamp shade), it is better to simply use a union to do the same thing. Let's add such an object to our scene. Here is a light bulb we have made just for this purpose:

```
#declare Lightbulb = union {
```

```

merge {
  sphere { <0,0,0>,1 }
  cylinder {
    <0,0,1>, <0,0,0>, 1
    scale <0.35, 0.35, 1.0>
    translate 0.5*z
  }
  texture {
    pigment {color rgb <1, 1, 1>}
    finish {ambient .8 diffuse .6}
  }
}
cylinder {
  <0,0,1>, <0,0,0>, 1
  scale <0.4, 0.4, 0.5>
  texture { Brass_Texture }
  translate 1.5*z
}
rotate -90*x
scale .5
}

```

Now we add the light source:

```

light_source {
  <0, 2, 0>
  color White
  looks_like { Lightbulb }
}

```

Rendering this we see that a fairly believable light bulb now illuminates the scene. However, if we do not specify a high ambient value, the light bulb is not lit by the light source. On the plus side, all of the shadows fall away from the light bulb, just as they would in a real situation. The shadows are sharp, so let's make our bulb an area light:

```

light_source {
  <0, 2, 0>
  color White
  area_light <1, 0, 0>, <0, 1, 0>, 2, 2
  adaptive 1
  jitter
}

```

```
looks_like { Lightbulb }  
}
```

We note that we have placed this area light in the x-y-plane instead of the x-z-plane. We also note that the actual appearance of the light bulb is not affected in any way by the light source. The bulb must be illuminated by some other light source or by, as in this case, a high ambient value.

### 2.2.4.6.3 Using Light Fading

If it is realism we want, it is not realistic for the plane to be evenly illuminated off into the distance. In real life, light gets scattered as it travels so it diminishes its ability to illuminate objects the farther it gets from its source. To simulate this, POV-Ray allows us to use two keywords: `fade_distance`, which specifies the distance at which full illumination is achieved, and `fade_power`, an exponential value which determines the actual rate of attenuation. Let's apply these keywords to our fill light.

First, we make the fill light a little brighter by changing `Gray50` to `Gray75`. Now we change that fill light as follows:

```
light_source {  
  <0, 20, 0>  
  color Gray75  
  fade_distance 5  
  fade_power 1  
  shadowless  
}
```

This means that the full value of the fill light will be achieved at a distance of 5 units away from the light source. The fade power of 1 means that the falloff will be linear (the light falls off at a constant rate). We render this to see the result.

That definitely worked! Now let's try a fade power of 2 and a fade distance of 10. Again, this works well. The falloff is much faster with a fade power of 2 so we had to raise the fade distance to 10.