AM 205 POV-Ray tutorial

September 30, 2021

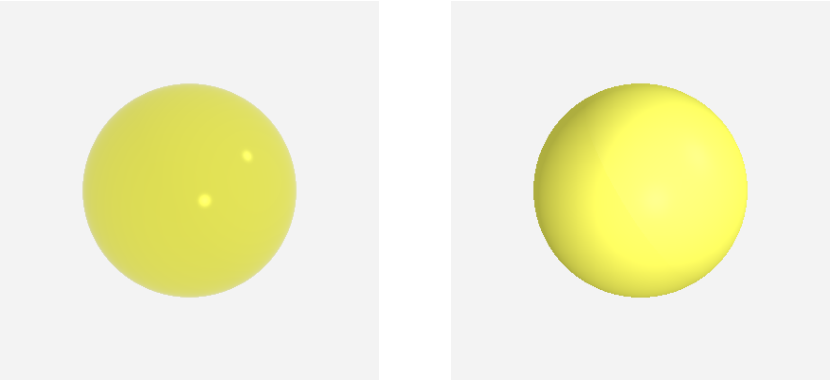
1 Introduction

1.0.0.1 POV-Ray POV-Ray is a free software for producing high-quality computer graphics. It uses the ray-tracing rendering technique. POV-Ray reads in a text file that contains objects, lighting and camera viewpoint of a scene, and generates a high quality image with realistic reflections, shading, perspective and other effects.

Figure 1: Realistic rendering of glasses using POV-Ray. Source: `http://www.lilysoft.org/CGI/SR/Spectral%20Render.htm`

1.0.0.2 Ray-tracing Ray-tracing is an image rendering technique. It simulates how rays of light travel in the real world, how the light rays hit on objects, and how the light rays reflect or refract on the objects' surfaces depending on the material properties of the objects.

Intuitively, how light rays travel results in how we see the world. We can perceive an object and tell the material of the object because of the light bouncing off its surface. For example, a metallic ball reflects a lot of light, and when we see the shiny surface we could tell that it is made of metal. In comparison, a rubbery ball does not reflect light so much and it is much dimmer. And we can easily tell the two materials apart. How we perceive objects and their properties depends heavily on the visual effects they reflect or refract light. And ray-tracing simulates the process.



(a) (b)

Figure 2: (a) a metallic texture ball; (b) a rubbery texture ball.

In POV-Ray, we can define a "camera", which is like our "eyes", our viewpoint. We can also define light sources, which are the sources emitting lights in the scene. POV-Ray then originate light rays from the camera and trace the rays "backwards" into the scene. When the light rays hit an object, it will work out the color of the surface at the hit points, depending on the material properties of the object and other scene setting. It will work out the amount of light coming from each light source. In the end, colors of all the pixels in the image are calculated, and you have a highly realistic scene in the image.

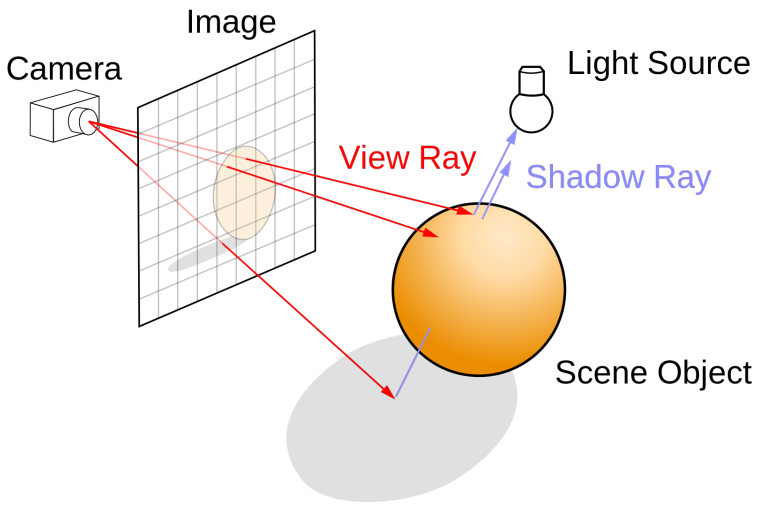


Figure 3: Schematic illustration of backward ray-tracing. Source: `https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg`

2 Basics

2.1 Setting the scene

Here, we need to define our viewpoint with "camera", the lights in the scene with "light source", and the background color with "background".

2.1.0.1 coordinate system To start with, we need to know POV-Ray's coordinate system. It uses a "left-handed" coordinate system, where the positive y-axis points up, the positive x-axis points to the right, and the positive x-axis points into the screen.

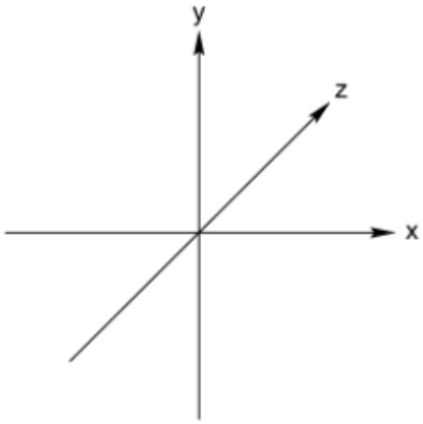


Figure 4: POV-Ray's left-handed coordinate system. Source: `https://www.povray.org/documentation/3.7.0/t2_2.html`

2.1.0.2 camera Camera sets our viewpoint, view angle and the direction we are looking at. You may refer to the illustration below in understanding the different settings that go into the syntax.

```
// basic syntax: usually, this is all we need to position the camera
camera {
    location <x0,y0,z0>   //where we are looking from
    look_at <x1,y1,z1>    //where we are looking at
}

//advanced options and modifiers
camera {
    Location <x0,y0,z0>
    look_at <x1,y1,z1>
    //sky: you can think of this as an antenna pointing out of the top of
   the camera, it tells us how the camera is tilted
```

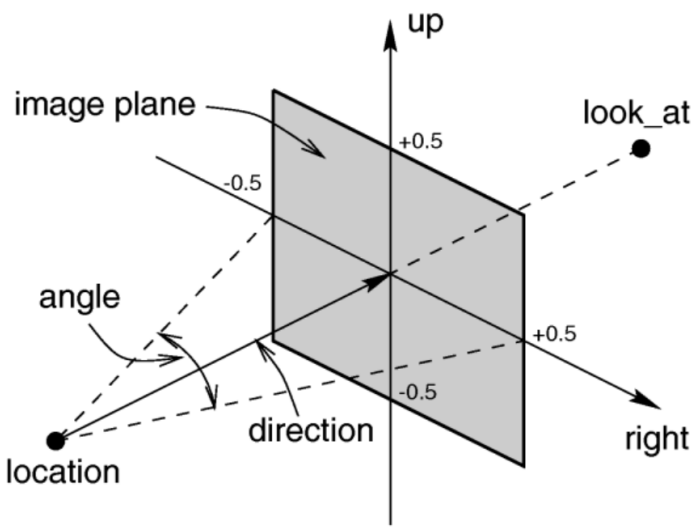
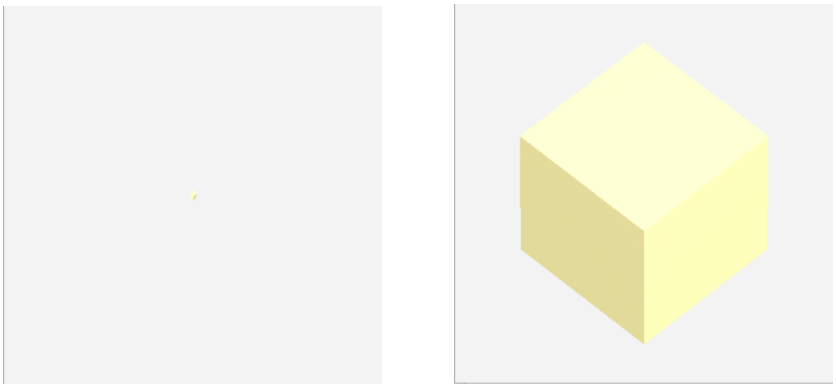


Figure 5: Schematic illustration of camera. Source: `http://www.povray.org/documentation/view/3.7.0/246/`

```
    sky <x2,y2,z2>
    //right and up: they let us specify the relative height and width of the
    view screen. We can use the two options to "zoom in" to the image.
    right 0.03*x
    up 0.03*z*image_height/image_width
    //rotate: rotate the camera with d1 degree about the x-axis, d2 degree
   about the y-axis, and d3 degree about the z-axis.
    rotate <d1,d2,d3>

}
// example: a camera at location <-30,-30,50>, align the top with the z-axis
    (0,0,1), looking into the direction towards point (0,0,0), and zoom
   into the view with "right" and "up" options.
camera {
    location <-30,-30,50>
    sky z
    right 0.033*x
    up 0.033*z*image_height/image_width
    look_at <0,0,0>
}
```

2.1.0.3 light source A point or area that emits light in a scene. We just need to specify the location of the light source, and the color of the light that it emits.



(a) (b)

Figure 6: (a) looking at an object from far away; (b) looking at the same object with added "right" and "up" options to zoom into the view.

Here is a nice chart to pick your favorite color's RGB value: `https://tug.org/pracjourn/2007-4/walden/color.pdf`

```
// basic syntax
light_source {
    <Location>, COLOR
}
// example: a light at location <-8,-20,30>, with RGB color <0.97, 0.55,
   0.8>
light_source{<-8,-20,30>, color rgb <0.97,0.55,0.8> }
```

You can also try adding in a few more light sources at different locations and with different colors, to light up the scene from different angles!

2.1.0.4 background This sets the background color. In another word, it assigns a color to all rays that do not hit any object.

```
// basic syntax
background {
    COLOR
}
// example: background with RGB color <0.4, 0.9, 0.9>
background{rgb <0.4,0.9,0.9>}
```

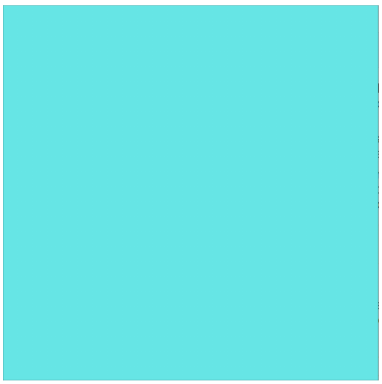


Figure 7: Background with RGB <0.4,0.9,0.9> gives a beautiful sky blue canvas!

2.2 Some basic objects and object modifiers

2.2.0.1 box This gives you a 3D rectangular prism. To construct it, you just need to define the two diagonal corners of the shape.

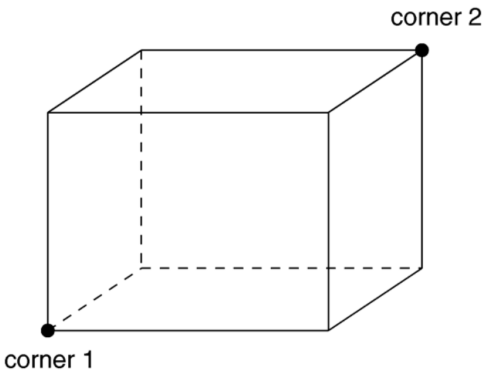
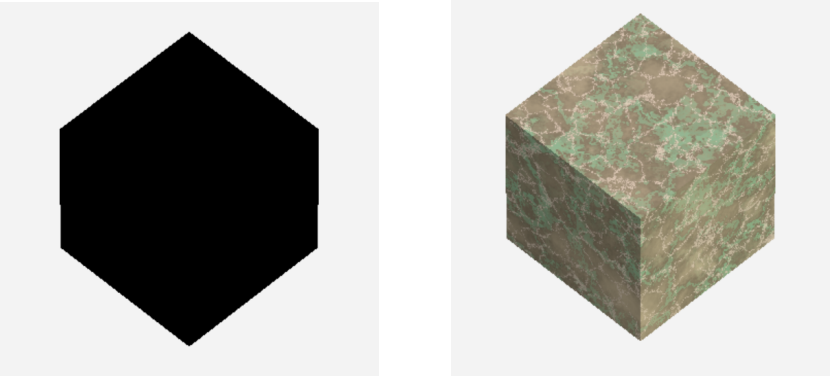


Figure 8: The box object. Source: `https://www.povray.org/documentation/view/3.6.1/276/`

```
//basic syntax
box {
    <Corner_1>, <Corner_2>
    [OBJECT_MODIFIERS...]
}
//basic example:
box { <-0.5,-0.5,-0.5>,<0.5,0.5,0.5> }

//prettier example: We will talk about object modifier later!
#include "stones.inc"
box {
    <-0.5,-0.5,-0.5>,<0.5,0.5,0.5>
    texture {
```

```
      T_Stone25     // Pre-defined from stones.inc library
    }
}
```



(a) (b)

Figure 9: (a) box from basic example; (b) prettier example box with stone texture.

2.2.0.2 sphere A sphere object. In the most basic syntax, you just need to specify the center (point coordinate) and radius (number) of the sphere.

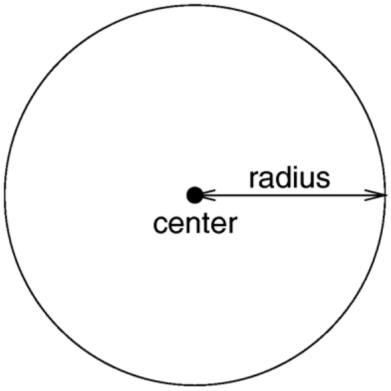


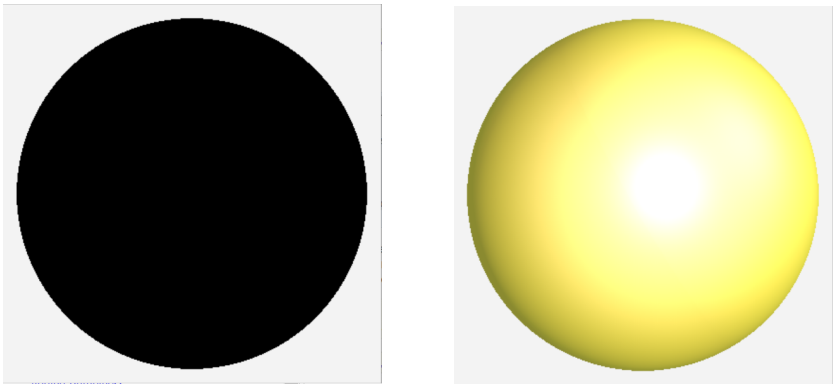
Figure 10: The sphere object. Source: `http://www.povray.org/documentation/view/3.6.1/283/`

```
//basic syntax
sphere {
    <Center>, Radius
    [OBJECT_MODIFIERS...]
```

```
}

//basic example: A sphere centered at (0,0,0) with radius 1.
sphere { <0,0,0>, 1 }

//prettier example: We will introduce object modifiers later!
#include "colors.inc"
sphere {
     <0,0,0>, 1
     texture{
         pigment { BrightGold }
         finish {
            ambient 0.3
            diffuse 0.7
            specular 0.5
            roughness .2
            phong .75
            phong_size 10
         }
     }
}
```



(a) (b)

Figure 11: (a) sphere from basic example; (b) prettier example sphere with bright gold color and surface effects.

2.2.0.3 plane This is a simple way to define an infinite flat surface in the scene. For example, it can be the "ground" or the "table surface" in your scene. Here, we need to specify

<Normal>, which is a vector normal to the plane. Then, we need to specify "Distance", which gives the distance along the normal that the plane is from the origin.

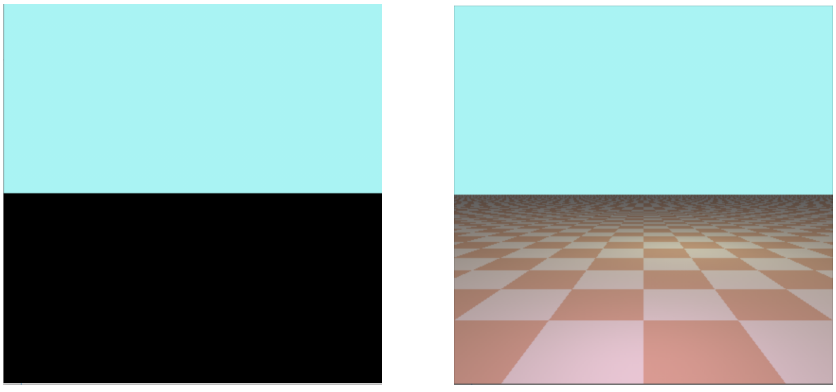
A plane divides a space in two parts, any point that is "under" the plane is inside, and any point that is "above" the plane is outside.

```
//basic syntax
plane {
    <Normal>, Distance
    [OBJECT_MODIFIERS...]
}

//basic example: a plane with normal vector (0,1,0), that's 4 distance away
   from the origin.
plane { <0, 1, 0>, 4 }

//The above is equivalent to the following, using the x, y or z built-in
   vector identifiers
plane { y, 4 }

//Prettier example: checkerboard patterns. We will introduce adding object
   modifiers later.
#include "colors.inc"
plane {
    y, -1
    texture{ pigment {checker White Tan} }
}
```



(a) (b)

Figure 12: (a) plane from basic example; (b) prettier example plane with checkerboard patterns.

2.2.0.4 **torus** Donut shape! A torus is a 4^{th} order quartic polynomial shape that looks like a donut or inner tube. Here, as shown in the picture below, we need to define the major radius and minor radius.

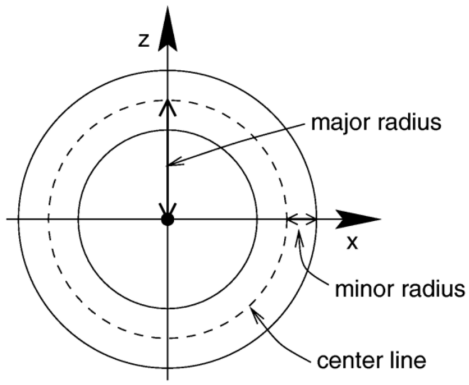


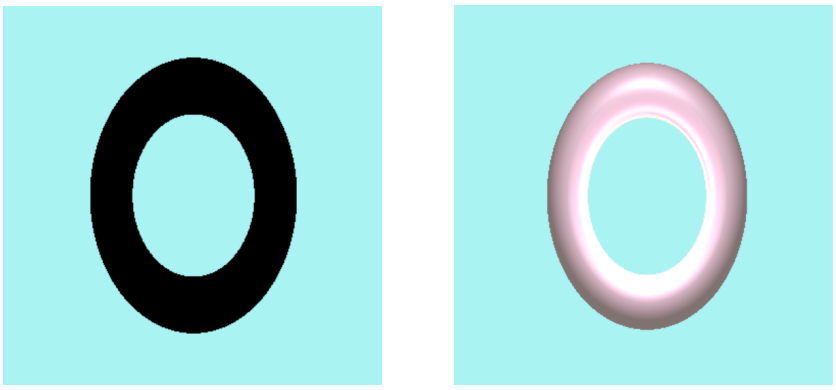
Figure 13: The torus object. Source: `https://www.povray.org/documentation/view/3.6.1/288/`

```
//basic syntax
torus {
    Major, Minor
    [OBJECT_MODIFIERS...]
}

//basic example: A torus with major radius 4 and minor radius 1.
torus { 4, 1 }

//prettier example: A pink donut with shiny texture
#include "colors.inc"
torus {
    4, 1
    rotate -90*x      // rotate -90 degrees around x axis
    texture{
        pigment { Pink }
        finish {
            ambient 0.3
            diffuse 0.7
            specular 0.3
            roughness .2
            phong .75
            phong_size 10
        }
    }
```

```
}
```



(a) (b)

Figure 14: (a) donut from basic example; (b) prettier example donut with pink color and shiny surface (icing? :)).

2.2.0.5 **Exercise 1** Try setting your own scene with camera, light source(s), and background. And put (an) object(s) in the scene! Adjust your settings till you are satisfied with the scene! For now, if you don't add in any object modifiers, your object would just look like a solid black block. But that's okay! In what follows, we will introduce some object modifiers, which will add surface properties and textures to your object(s).

2.3 Object modifiers

Adding object modifiers to your object gives it a more realistic appearance, visualized by the different colors, textures and finishes of the surface!

2.3.0.1 **texture** We can use the "texture" statement to simulate an object's surface. The following description taken from POV-Ray documentation (`http://wiki.povray.org/content/Reference:Texture`) makes it clear what "texture" does:

- Textures are combinations of pigments, normals, and finishes.
- Pigment is the color or pattern of colors inherent in the material.
- Normal is a method of simulating various patterns of bumps, dents, ripples or waves by modifying the surface normal vector.
- Finish describes the reflective properties of a material.

```
//Example syntax
texture {
  pigment { MyPigment }
  normal { MyNormal }
  finish { MyFinish }
}
```

2.3.0.2 pigment The specifies colors of objects.

1. Using pre-defined colors in the library, *color.inc*: Check out the list of colors here: `http://povray.tashcorp.net/library/colors.inc/`

```
//Example:
#include "color.inc"
sphere{
    <0,0,0>, 1
    texture{ pigment { SlateBlue } }
}
```

2. Specify RGB colors: use *color rgb* $<COLOR>$ syntax. "color" is optional and can be omitted in the syntax.

```
//Example:
sphere{
    <0,0,0>, 1
    texture{ pigment { color rgb <0.9,0.5,0.4> } }
}
```

3. Specifying "filter" and "transmit" with RGBF or RGBFT: (Source:`http://www.povray.org/documentation/view/3.7.1/230/`)

"F" stands for filter, it specifies the amount of filtered transparency of the material. This value is useful to specify in materials like stained glass. Default is 0.

"T" stands for transmit, which is the amount of non-filtered light transmitted through the surface. Some real world examples are thin see-through cloth, or dust on a surface. Default is 0. A value between 0 and 1 also gives transparency effect of the material.

```
//Example 1: rgbf follows a 4-term vector. The following creates 60%
   transparency of the object.
sphere{
    <0,0,0>, 1
```

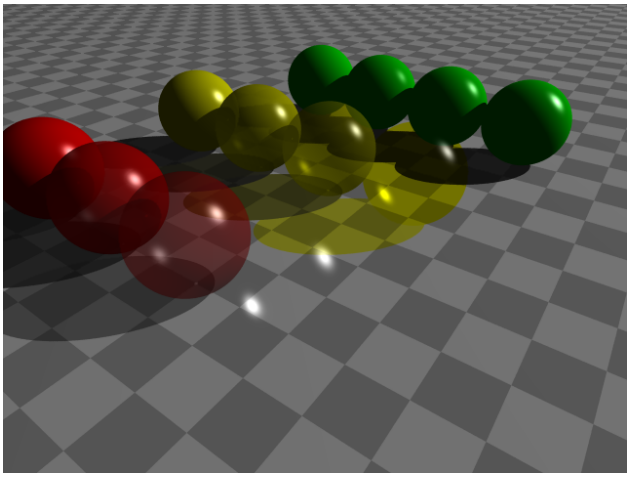


Figure 15: "The yellow balls have progressive “filter” values. The red balls have progressive “transmit” values. Note that if the transmit value is 1, then the object becomes invisible.Both “filter” and “transmit” let light pass through. Their difference is that filter will tint the light with the object’s surface texture, while transmit will not. (look at their shadows)."-excerpt and photo from source: `http://xahlee.info/3d/povray-glassy.html`

```
    texture{ pigment { color rgbf <0.9,0.5,0.4, 0.6> } }
}
//Example 2: rgbft follows a 5-term vector. The following creates 0.6
   transparency and 0.7 contrast in colors.
sphere{
    <0,0,0>, 1
    texture{ pigment { color rgbft <0.9,0.5,0.4, 0.6, 0.7> } }
}
//Example 3: alternatively, we can use the transmit and filter keywords
#include "color.inc"
sphere{
    <0,0,0>, 1
    texture{ pigment { Pink filter 0.5 transmit 0.3 } }
}
```

4. Declared your own color variables to use in your program. We can define our own colors, store in a variable, and use throughout the program. This uses the "#declare" keyword in defining the variable.

```
#declare White = rgb <1,1,1>;
#declare LightGray = White*0.8;
sphere{
    <0,0,0>, 1
    texture{ pigment { LightGray } }
```

```
}
```

Something fun: Let's make a beautiful pink transparent donut!

```
torus {
    4, 1
    rotate -90*x
    texture {
        pigment { Pink filter 0.5 transmit 0.3}
        finish {
            ambient 0.3
            diffuse 0.7
            specular 0.3
            roughness .2
            phong .75
            phong_size 10
         }
     }
  }
```

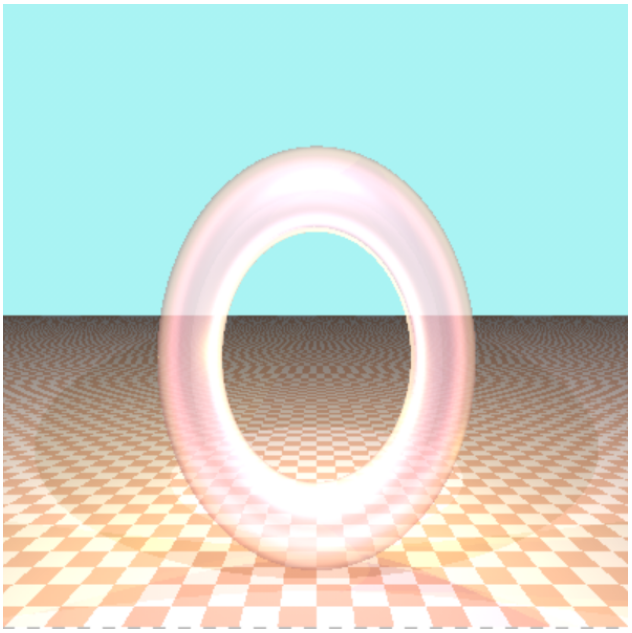


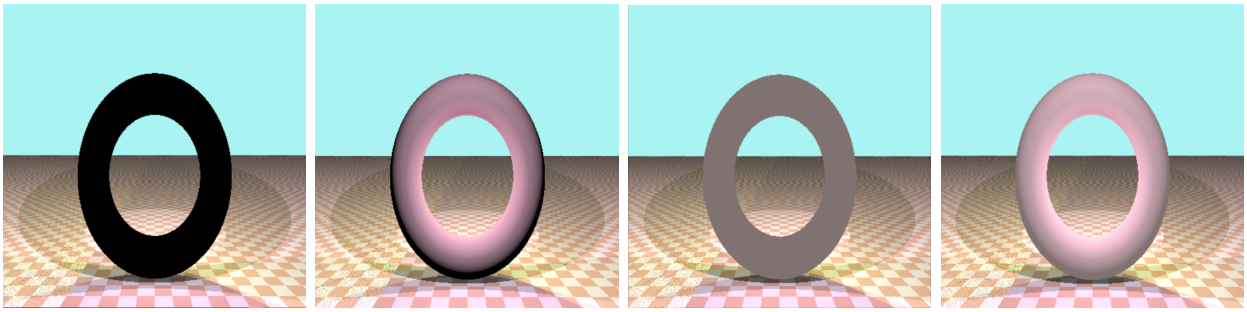
Figure 16: A beautiful pink transparent donut.

2.3.0.3 **finish** This specifies the reflective properties of a material. The content of this section can be referred to in the source POV-Ray documentation: `http://www.povray.org/documentation/view/3.6.0/79/`

1. ambient and diffuse These two specifies where the light that illuminates the object comes from.

Keyword "ambient" simulates the amount of light scattered around the object that does not come directly from a light source.

Keyword "diffuse" simulates the amount of light directly from a light source.



(a) (b) (c) (d)

Figure 17: (a) ambient 0, diffuse 0; (b) diffuse 0.7; (c) ambient 0.3; (d) ambient 0.3, diffuse 0.7.

Usually, we want the light source to contribute the majority of lighting in the scene. Therefore, usually the diffuse value is larger than the ambient value. In most cases, we can use an ambient value of 0.1 ... 0.2 and a diffuse value of 0.5 ... 0.7.

2. phong and phong_size

The two keywords generate effects of highlights of a hard and shiny surface.

The float that follows "phong" determines the brightness of the highlight. The float following "phong_size" determines its size. The larger the "phong_size" value, the harder and shinier surface, and the smaller the highlight is. Figure 18 shows some examples with different values of the two parameters.

3. specular and roughness

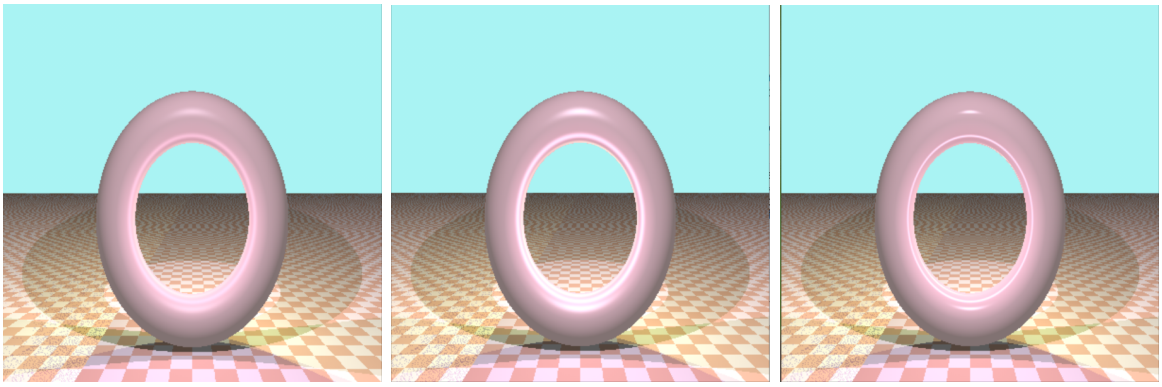
Another way to create highlighting effects is to use "specular" and "roughtness" keywords.

The "specular" value is typically in between 0.0 and 1.0. Value 0.0 means no highlight. Value 1.0 means complete saturation to the color of the light source at the brightest spots of the highlight.

Keyword "roughness" specifies the size of the highlight. Typical values range from 1.0 (very rough - large highlight) to 0.0005 (very smooth - small highlight). The default value, if roughness is not specified, is 0.05 (plastic). Figure 19 shows some examples with different values of the two parameters.

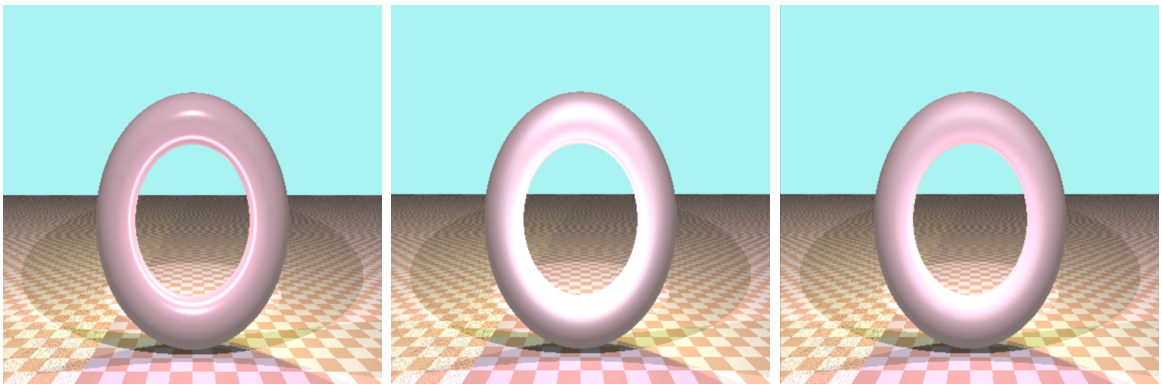
Generally, the highlighting effects generated by "specular" is more realistic than "phong". But you can play with both when designing the texture.

4. reflection



(a) (b) (c)

Figure 18: (a) phong 0.3, phong_size 25; (b) phong 0.8, phong_size 25; (c) phong 0.8, phong_size 150.



(a) (b) (c)

Figure 19: (a) specular 1.0, roughness 0.005; (b) specular 1.0, roughness 0.1; (c) specular 0.4, roughness 0.1.

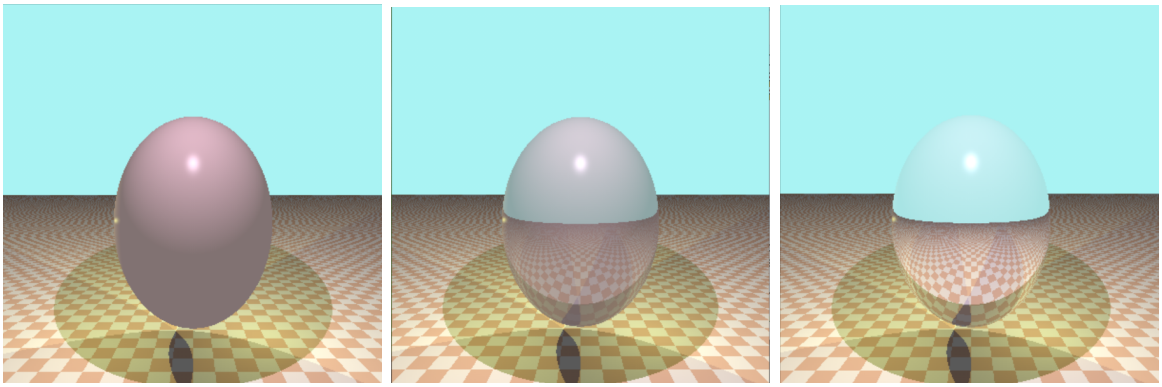
We can specify a value from 0.0 to 1.0 to specify how reflective the surface is. 0.0 would be no reflection, and 1.0 would be perfect reflection, like a mirror.

Generally, to get a more realistic look, you may find that the higher "reflection" is, the lower "diffuse" and "ambient" should be. Figure 21 shows some examples of different reflection values. Notice how the ambient and diffuse values change too.

2.3.0.4 Exercise 2 Try adding object modifiers to the objects in your scene. Try adjusting the settings till it meets your artistic standards!

Example syntax for object modifiers:

```
sphere{
    <0,0,0>, 4
```



(a) (b) (c)

Figure 20: (a) ambient 0.3, diffuse 0.8, specular 0.8, roughness 0.006, reflection 0; (b) ambient 0.2, diffuse 0.6, specular 0.8, roughness 0.006, reflection 0.3; (c) ambient 0.1, diffuse 0.3, specular 0.8, roughness 0.006, reflection 0.8

```
    texture {
        pigment { Pink  filter 0.5 transmit 0.3}
        finish {
            ambient 0.1
            diffuse 0.3
            specular 0.8
            roughness .006
            //phong 0.8
            //phong_size 150
            reflection {0.8}
         }
     }
}
```

2.4 Operations on the object

2.4.0.1 **Transformations** We can define transformation of objects by rotate, scale, translate and transformation matrix.

1. translate The keyword "translate" just move an object in space. It follows a vector, $< a, b, c >$, meaning move the object in the x, y, z directions for a, b, c distance, respectively.

```
//Example: translating a sphere to be centered at <1-7,2+4,3+3>=<-6,6,6>
sphere {
    <1, 2, 3>, 1
    translate <-7, 4, 3>
```

```
}
```

2. scale We can also scale the size of an object. It follows a vector, $< a, b, c >$, meaning the amount of scaling in each of the x, y, z directions.

```
//Example: stretch the sphere in the x-direction to twice the original
   length, and squish in the z-direction to half the original length. We
   get an ellipsoid.
sphere {
    <0,0,0>, 1
    scale <2,1,0.5>
}
```

3. rotate

We can rotate an object too. It follows a vector, $< a, b, c >$, which are the number of degrees to rotate about the $x-, y-, z-$axes.

```
//Example: rotate the ellipsoid around the x-axis for 30 degrees, around the
    y-axis for 20 degrees and around the z-axis for -60 degrees.
sphere {
    <0,0,0>, 1
    scale <2,1,0.5>
    rotate <30,20,-60>
}
```

4. matrix Using matrix syntax is useful when you have more complicated transformations. Its syntax is:

```
matrix <Val00, Val01, Val02,
        Val10, Val11, Val12,
        Val20, Val21, Val22,
        Val30, Val31, Val32>
```

Val00 to Val32 are floating numbers. For a point, $P =< px, py, pz >$, the matrix transform it into $Q =< qx, qy, qz >$ by:

$$qx = Val00 * px + Val10 * py + Val20 * pz + Val30$$

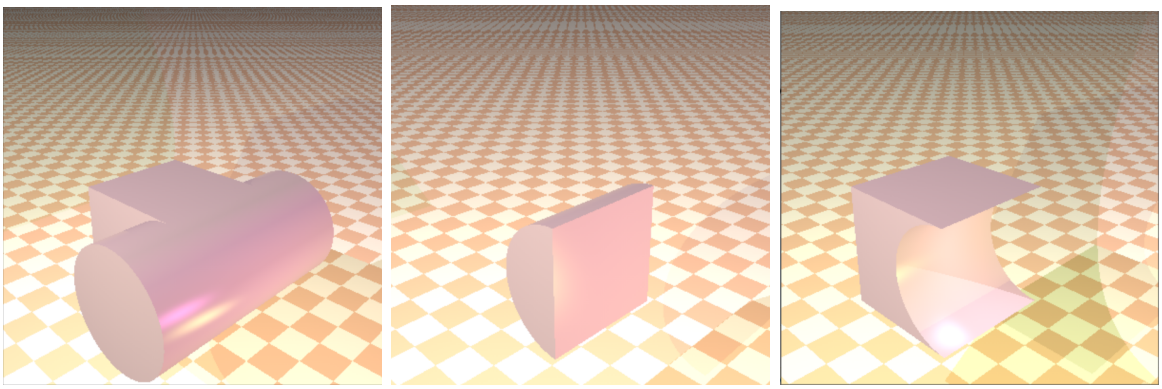
$$qy = Val01 * px + Val11 * py + Val21 * pz + Val31$$

$$qz = Val02 * px + Val12 * py + Val22 * pz + Val32$$

2.4.0.2 Set operations 1. union; 2. intersection; 3. difference

```
//Example: difference of a box with a cylinder. We can then add object
   modifiers to this new object generated.
difference {
    box { <-1.5, -1, -1>, <0.5, 1, 1> }
    cylinder { <0.5, 0, -2>, <0.5, 0, 2>, 1 }

    texture{
        pigment { Pink }
        finish {
            ambient 0.3
            diffuse 0.7
            specular 0.8
            roughness .006
         }
    }
}
//the "difference" keyword above can be changed to "union" or "intersection"
    to get different objects. The results are shown in the figure below.
```



(a) (b) (c)

Figure 21: (a) union; (b) intersection; (c) difference

2.4.0.3 Exercise 3 Construct and visualize the interesting shape that is the intersection of the three cylinders $x^2 + y^2 < 1, x^2 + z^2 < 1, y^2 + z^2 < 1$.

2.5 Define shapes using mathematical expressions

The section is perhaps very useful for our purposes - as we are mathematically oriented users! It allows us to use mathematical functions and expressions to define the shape of an object.

2.5.0.1 isosurface "isosurface" generates objects whose shapes are defined by mathematical expressions. Its syntax is:

```
//Example syntax of a sphere
isosurface {
  function { sqrt(pow(x,2) + pow(y,2) + pow(z,2)) - 1 }
  contained_by { box { -10, 10 } } //here, -10 is short for -10*<1,1,1>
  [threshold FLOAT_VALUE]
  [open]
  [OBJECT_MODIFIERS...]
}
```

In "function", we would enter the function that define the shape. Here, for a sphere defined by $\sqrt{x^2+y^2+z^2}-1=0$, we would put the left hand side of the equation in.

There is an optional argument, "threshold", which defaults to 0. Here, the syntax means to generate a shape defined by "function"="threshold". Therefore, if we input "threshold 3", the sphere would become $\sqrt{x^2+y^2+z^2}-1=3$, which would have radius 2, instead of 1.

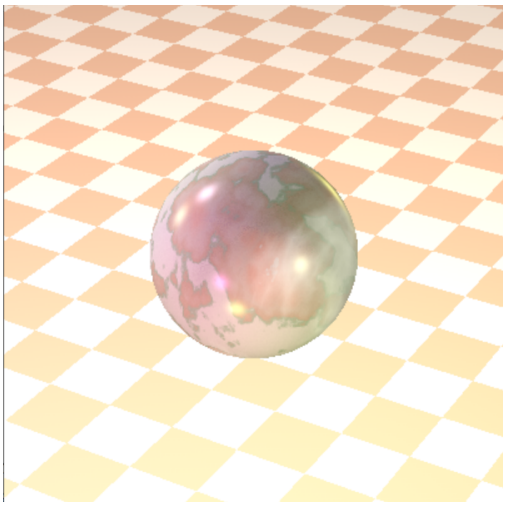
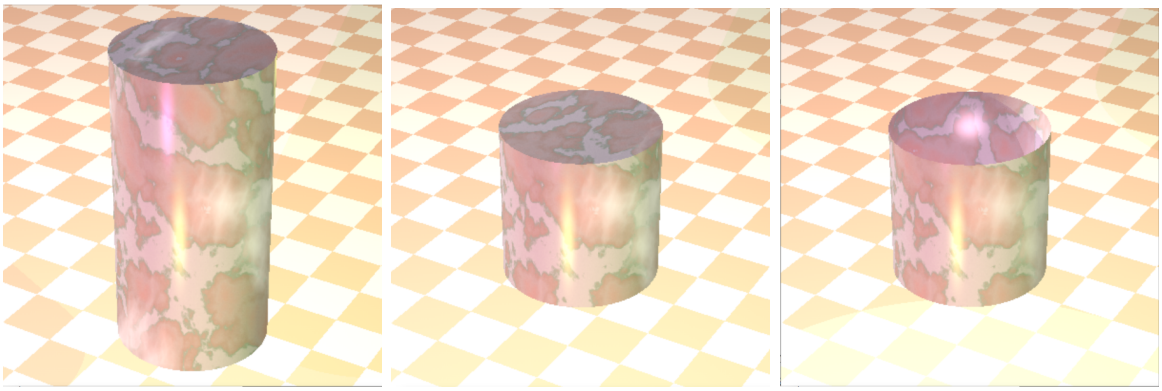


Figure 22: A sphere generated by isosurface.

"contained_by" require input of a box object or a sphere object. The object poses a limit/bound of the shape generated by the function. Here, in order for the sphere to show, the box needs to be larger than the sphere, so that it contains the sphere object. For example, in Fig 23, both (a) and (b) are generated with function input $\sqrt{x^2+z^2}-1$, which gives a cylinder surface with radius 1. However, (a) is contained by a box from $<-2,-2,-2>$ to

$< 2, 2, 2 >$, and therefore, the height of the cylinder is bounded by the box and is 4; (b) is contained by a box half the size, and o the height of the cylinder is 2.

There is also an option of "open", which would hide the object defined in "contained_by", and only show the surface defined by the mathematical function. This is shown in (c) in Fig. 23.



(a) (b) (c)

Figure 23: (a) box -2,2; (b) box 1,1; (c) box -1,1, with "open" keyword

```
//Example syntax of a cylinder
isosurface {
    function { sqrt(pow(x,2) + pow(z,2)) - 1 }
    contained_by { box { -1, 1 } }
    open

    texture{
         T_Stone19
         scale 5
         finish {
            ambient 0.3
            diffuse 0.7
         }
    }
}
```

2.5.0.2 polynomial object Keyword "poly" is used in building a polynomial object. Let's look at a simple example of how to build a sphere from scratch, using "poly". The sphere function is:

$$\sqrt{x^2 + y^2 + z^2} = r.$$

Converting this to polynomial form,

$$x^2 + y^2 + z^2 - r^2 = 0.$$

This means that we need a 2^{nd} degree polynomial, with the corresponding coefficients for each term in the equation.

```
//example syntax for creating a sphere using poly
#declare Radius=1;
poly
{ 2,
  <1,0,0,0,1,
   0,0,1,0,-Radius*Radius>
}
```

The 10 entries in the vector $< A1, A2, A3, \ldots, A10 >$ in the syntax above, corresponds to the coefficients of each term in a 2^{nd} degree polynomial.

$2^{\mathbf{nd}}$	x^2	xy	xz	x	y^2	yz	y	z^2	z	1
coefficients	A1	A2	A3	A4	A5	A6	A7	A8	A9	A10

Here are some shortcuts of different degrees of polynomials:

- 2^{nd} degree: quadric
- 3^{rd} degree: cubic
- 4^{th} degree: quartic

So, for example, the sphere can alternatively be constructed using "quadric" keyword:

```
//example syntax for creating a sphere using shortcut of 2nd degree poly
#declare Radius=1;
quadric
{
  <1,0,0,0,1,
   0,0,1,0,-Radius*Radius>
}
```

2.5.0.3 Exercise 4

Our favorite donut shape (torus) has polynomial representation:

$$x^4+2x^2y^2+2x^2z^2-2(r_1^2+r_2^2)x^2+y^4+2y^2z^2+2(r_1^2-r_2^2)y^2+z^4-2(r_1^2+r_2^2)z^2+(r_1^2-r_2^2)^2=0.$$

Could you make a donut shape using poly, or one of the shortcuts? What degree of polynomial should we use? To match the coefficients and the terms in the polynomial, you can refer to the table provided in POV-Ray documentation: `http://www.povray.org/documentation/view/3.6.1/298/`

2.5.0.4 superellipsoid

This is useful when we want to generate boxes with rounded corners. The syntax is:

```
superellipsoid
{
    <e, n>
    [OBJECT_MODIFIERS...]
}
```

where $0 < e, n \leq 1$. This gives a shape defined by the mathematical equation

$$f(x,y,z) = (|x|^{\frac{2}{e}} + |y|^{\frac{2}{e}})^{(\frac{e}{n})} + |z|^{(\frac{2}{n})} - 1 = 0.$$

If we choose $e = 1, n = 1$, we get a sphere.

2.5.0.5 Exercise 5

Recall the superellipsoid movie Chris shown in class... We can define a superellipsoid shape with the 4-vector-norm. For example,

$$(x^4 + y^4 + z^4)^{\frac{1}{4}} = R$$

As a practice for the syntax, could you construct three superellipsoid using the above three methods respectively: (a) using "isosurface"; (b) using "poly" or its shortcuts; (c) using "superellipsoid" keyword, what values of r, n should you choose? Lastly, consider assigning them different colors and textures! :)

2.6 Generating POV-Ray file from program outputs

A note on scientific visualization here! If you have outputs from some program, for example, a set of points, you can have the output written in POV-Ray syntax when printing the outputs, and save in ".pov" format. Then, the file can be rendered in POV-Ray!

Example: liss_3d.py, liss_3d.pov for a 3D Lissajous figure. `https://mathcurve.com/courbes3d.gb/lissajous3d/lissajous3d.shtml`

2.6.0.1 Exercise 6 Creativity time! Create whatever scene you like! The only requirement is that it contains an object defined by mathematical expressions. What mathematical shapes you find beautiful? Try adjusting the lights, surface properties, etc, till it meets your artistic standards! Alternatively, you can create you object(s) based on some program outputs!

Exercise 5, 6 are take home exercises. The deadline for submission is Thursday, October 7th, at 5pm, via Canvas. Please prepare a PDF with your rendered images, labeled for each exercise. Also, write down the names of members in your group on the PDF. Please also prepare the code files you use for the exercises. Lastly, please put everything in a ZIP file and submit on Canvas.

3 Reference

- `http://www.povray.org/documentation/`
- `http://www.lilysoft.org/CGI/SR/Spectral%20Render.htm`
- `https://tug.org/pracjourn/2007-4/walden/color.pdf`
- `https://commons.wikimedia.org/wiki/File:Ray_trace_diagram.svg`