INVESTIGATION OF POV-RAY DATA STRUCTURES AND ALGORITHMS

Fariha Sultana
Student ID: 04201001

Department of Computer Science and Engineering
April 2009

BRAC University, Dhaka, Bangladesh

DECLARATION

I hereby declare that this thesis is based on my own analysis of POV-Ray software. Materials of work found by other researcher are mentioned by reference. This thesis, neither in whole nor in part, has been previously submitted for any degree.

Signature of
Supervisor

Signature of
Author

ACKNOWLEDGEMENT

My special thanks to Dr. Mumit Khan, my supervisor who has guided me thoroughly to complete the thesis paper. I would also like to thank J. D. Foley, A. Van Dam, J. F. Hughes, Steven K. Feiner for writing the book *Computer Graphics: Principle and Practice* which I consulted numerous times. Thanks to the POV-Ray newsgroup writers who helped me by answering my questions about the source code and last but not the least, thanks to POV-Ray authors for making POV-Ray freeware and open source for which I was able to study it and learned a great deal.

ABSTRACT

Ray tracing is the most powerful technique for producing photo-realistic images. POV-Ray is a popular freeware and open source ray tracing software capable of producing stunningly realistic images. While the user documentation is descriptive and thorough, not much explained for the advanced users who want to study the source code. This paper hopes to provide a detailed description of the ray tracing data structures and implementation of various features and to give pointers for further study.

TABLE OF CONTENTS

LIST OF FIGURES

1.0 INTRODUCTION

Photo-realistic images[1] attempt to synthesize the field of light intensities that would be focused on the film plane of a camera aimed at the objects depicted. It has been a topic of discussion in computer graphics since 1960. To produce photo-realistic image it is necessary to properly simulate the various phenomena of light with objects and texture them to give a realistic look. Photo-realism is necessary in a variety of applications. For example flight simulators, games and movies, design of 3D objects such as automobiles, airplanes and buildings etc. In scanline rendering and rasterization there are several algorithms of shading polygon meshes to model the interaction of light with objects. Scanline rendering technique is well suited for real time engines since it is fast and all current graphics cards use it. However, those images do not look very realistic. Images produced by scanline rendering techniques cannot simulate reflection, refraction, shadow, motion blur etc properly to give the impression necessary for photo-realistic images. The reason is the algorithms used in scanline rendering and rasterization are not based on real world lighting models.

Ray tracing is a powerful technique to render realistic images. Ray tracing determines the visibility of surface by tracing imaginary rays from the viewer's eyes to objects in the scene. The idea of ray tracing came from physics. To design lenses physicians usually plotted the path of light rays starting from the light source. This was called ray tracing. Computer graphic designers thought

that it would be a good idea to use the simulation of light physics to produce realistic images. Ray tracing was first developed by Appel[2] and by Goldstein and Nagel[3,4]. Appel used a sparse grid of rays used to determine shading, including whether a point was in shadow. Goldstein and Nagel pioneered the use of ray tracing to evaluate Boolean set operation. Whitted[5] and Kay[6] extended ray tracing to handle specular reflection and refraction.

Ray tracing is capable of simulating the real world reflection, refraction and shadow effects but very slow for real time rendering. A lot of research is going on with ray tracing and how it can be made faster. There are many ray tracers around which are used for commercial purpose and some are freeware. Some of the popular and well known are POV-ray, Radiance, BRL-CAD, V-Ray, Flamingo, YafRay etc. I started to use POV-Ray because it is a very powerful multi platform, freeware and open source ray tracing program. There is extensive user documentation for the POV-Ray scene developers. However, the source code is hard to follow, as there is not enough documentation. The goal of this thesis paper is to explain in detail the data structures and algorithms used in POV-Ray so that, enthusiastic learners can use it as a basis to understand the various features implemented by POV-Ray.

I used POV-Ray in windows and the version I worked on is 3.6.0. This version introduced major changes from 3.5 and provides a framework for future re-implementation. The most recent official version is 3.6.1, which is a minor bug fix of 3.6.0. Currently beta versions of 3.7 are being released which are subjected to frequent changes. The next official version will be 4. So, I think it is a good idea to study the source code of 3.6.0 to learn about the feature implementations of POV-Ray that will also help to understand the later implementations.

2.0 HISTORY OF POV-RAY

In 1986 David Buck found the C code of a raytracer for Unix, which he modified to run on his Amiga. Being impressed by the raytracer he decided to build his own. In that year he released his own raytracer named DKBTrace and released it for free in the internet. The raytracer supported quadric objects, CSG and procedural texture. The raytracer attracted a lot of interest and he continued to develop new versions of it. In around 1987 Aron Collins contacted David Buck and together they developed DKBTrace 2.12, which was portable among different platforms. The program became very popular and they could not keep up with the request for new features. They found about a user group in the CompuServe who were very excited about the program and wanted to build a new raytracer from scratch, which will have the features they wanted. David Buck proposed the idea of building a new raytracer to them, which will have new features and will be free, portable and support for the program will be undertaken and a new name. Initial suggestion for name was STAR (Software Taskforce on Animation and Rendering) but later they settled for Persistence of vision or POV-Ray. It worked in three ways. It was the result of a persistent vision of the developers, it was a reference to the Salvador Dali work which depicted a distorted but realistic world, and the term "persistance of vision" in biology referred to the ability to see an image that was presented briefly - almost an after image.

In 1989 the POV-Ray team began to work under Drew Wells to produce POV-Ray 0.5, an enhanced version of DKBTrace2.12 to meet the demand of the

user community. Later they launched POV-Ray1, which had many new features with a new grammar. As Drew Wells dropped out of the project Chris Young took over as the project leader. Later David Buck dropped out of the team as well and POV-Ray is continuing with Chris Cason as the project leader. The latest official version is POV-Ray 3.6 which was released in 2004.

The full history is available in [7].

3.0 BACKGROUND RESEARCH

Before going through the source code of POV-Ray researching is necessary to understand the main ray tracing concepts implemented in POV-Ray.

POV-ray is a recursive raytracer. This means additional rays are spawned conditionally after ray hits an object. Whitted described the recursive ray-tracing algorithm. Whitted[5] and Kay[6] fundamentally extended ray tracing to include specular reflective and ray tracing transparency. In the algorithm a shadow ray is traced back to the lights on the scene from the ray object intersection point to determine the amount of light recived from the light sources. If the surface is reflective then a reflection ray is spawned and if it is refractive and or transparent a reflection or transmitted ray is spawned. The ray from the eye to object is called the primary ray and the shadow, reflection and refraction rays are called secondary rays. Each of these secondary rays may in tern recursively spawn additional secondary rays. Thus a tree of rays is formed. In Whitted's algorithm a branch is terminated if the reflected or refracted ray fails to intersect an object or if some user specified maximum depth is reached or if the system runs out of storage. The tree is evaluated bottom up and each node's intensity is computed as its children's intensities.

Most of the ray object intersections in POV-Ray are found by replacing the ray parametric equation in the implicit equations of the objects. For the 2D objects solutions are found by first checking intersection with the plane on which

they reside and checking if the point is inside the primitive. There are many algorithms for checking if a point is inside a triangle or polygon. For 3D primitives the implicit object equation is replaced by the ray parametric equation. For some primitives a simpler and faster solution exists. To intersect a ray with a box Cyrus Beck algorithm is used to clip the ray against the six sides of the box. Ray sphere intersection geometric solution is faster than the algebraic counterpart. Surveys of theses algorithms were performed by Haines[8] and Hanrahan[9].

POV-Ray uses solid texturing techniques instead of just wrapping 2D texture around an object. Peachy[10] and Perlin[11] have extended traditional texture mapping by using solid textures. A texture value is assigned to each point in a 3D space. To each point of the object to be textured, there is associated some point in the texture space; the value of the texture at that point is also associated with the surface point. Texture patterns are actually functions that give a unique value for a point in the 3D space. A color is specified for different range of values and it is assigned to that point.

Real world surfaces are not very smooth, so it is necessary to simulate the roughness of the surface. Giving the surface a bumpy appearance by perturbing the surface normal instead of the surface itself was first discussed by Blinn[12]. POV-Ray supports normal perturbation for this purpose. The normal is modified in several ways to simulate bumps, dents, wrinkles etc. the ideas were taken from [10] and [13].

Speed-ups are necessary to make the ray tracing process faster and POV-Ray employs bounding volume hierarchy (BVH) and vista buffer and light buffer technique for this purpose. BVH is a hierarchy of nested bounding boxes. This system compartmentalizes all finite objects in a scene into invisible rectangular boxes that are arranged in a tree-like hierarchy. Before testing the objects within the bounding boxes the tree is descended and only those objects are tested whose bounds are hit by a ray. This can greatly improve rendering

speed. The vista buffer is created by projecting the bounding box hierarchy onto the screen and determining the rectangular areas that are covered by each of the elements in the hierarchy. Only those objects whose rectangles enclose a given pixel are tested by the primary viewing ray. A detailed discussion on BVH can be found in [14]. Light buffer is used to speed up the point in shadow testing process. The light buffer is created by enclosing each light source in an imaginary box and projecting the bounding box hierarchy onto each of its six sides. The light buffer concept was given by Haines and Greenberg [15].

4.0 POV-RAY

The Persistence of Vision Ray-Tracer creates three-dimensional, photo-realistic images using a rendering technique called ray-tracing. It reads in a text file containing information describing the objects and lighting in a scene and generates an image of that scene from the view point of a camera also described in the text file. Ray-tracing is not a fast process by any means, but it produces very high quality images with realistic reflections, shading, perspective and other effects. Details of POV-Ray features can be found here[16]

The ray tracing process begins from the camera. Rays are traced backwards from the camera to the scene. Detail discussions about POV-Ray ray tracing process follows from the next chapter.

5.0 CAMERA

The camera definition describes the position, projection type and properties of the camera viewing the scene. Its syntax is:

CAMERA:
  camera{ [CAMERA_ITEMS...] }
CAMERA_ITEM:
  CAMERA_TYPE | CAMERA_VECTOR | CAMERA_MODIFIER |
  CAMERA_IDENTIFIER
CAMERA_TYPE:
  perspective | orthographic | fisheye | ultra_wide_angle |
  omnimax | panoramic | cylinder CylinderType | spherical
CAMERA_VECTOR:
  location <Location> | right <Right> | up <Up> |
  direction <Direction> | sky <Sky>
CAMERA_MODIFIER:
  angle HORIZONTAL [VERTICAL] | look_at <Look_At> |
  blur_samples Num_of_Samples | aperture Size |
  focal_point <Point> | confidence Blur_Confidence |
  variance Blur_Variance | NORMAL | TRANSFORMATION

Camera default values:

```
DEFAULT CAMERA:
 camera {
   perspective
   location <0,0,0>
   direction <0,0,1>
   right 1.33*x
   up y
   sky <0,1,0>
 }

CAMERA TYPE: perspective
angle      : ~67.380 ( direction_length=0.5*
                       right_length/tan(angle/2) )
confidence : 0.9 (90%)
direction  : <0,0,1>
focal_point: <0,0,0>
location   : <0,0,0>
look_at    : z
right      : 1.33*x
sky        : <0,1,0>
up         : y
variance   : 1/128
```

5.1 Placing the Camera

The POV-Ray camera has ten different models, each of which uses a different projection method to project the scene onto the screen. Regardless of the projection type all cameras use the location, right, up, direction, and keywords to

determine the location and orientation of the camera. The type keywords and these four vectors fully define the camera. All other camera modifiers adjust how the camera does its job. The meaning of these vectors and other modifiers differ with the projection type used.

5.2 Perspective projection

The perspective keyword specifies the default perspective camera which simulates the classic pinhole camera. The (horizontal) viewing angle is either determined by the ratio between the length of the direction vector and the length of the right vector or by the optional keyword angle, which is the preferred way. The viewing angle has to be larger than 0 degrees and smaller than 180 degrees.

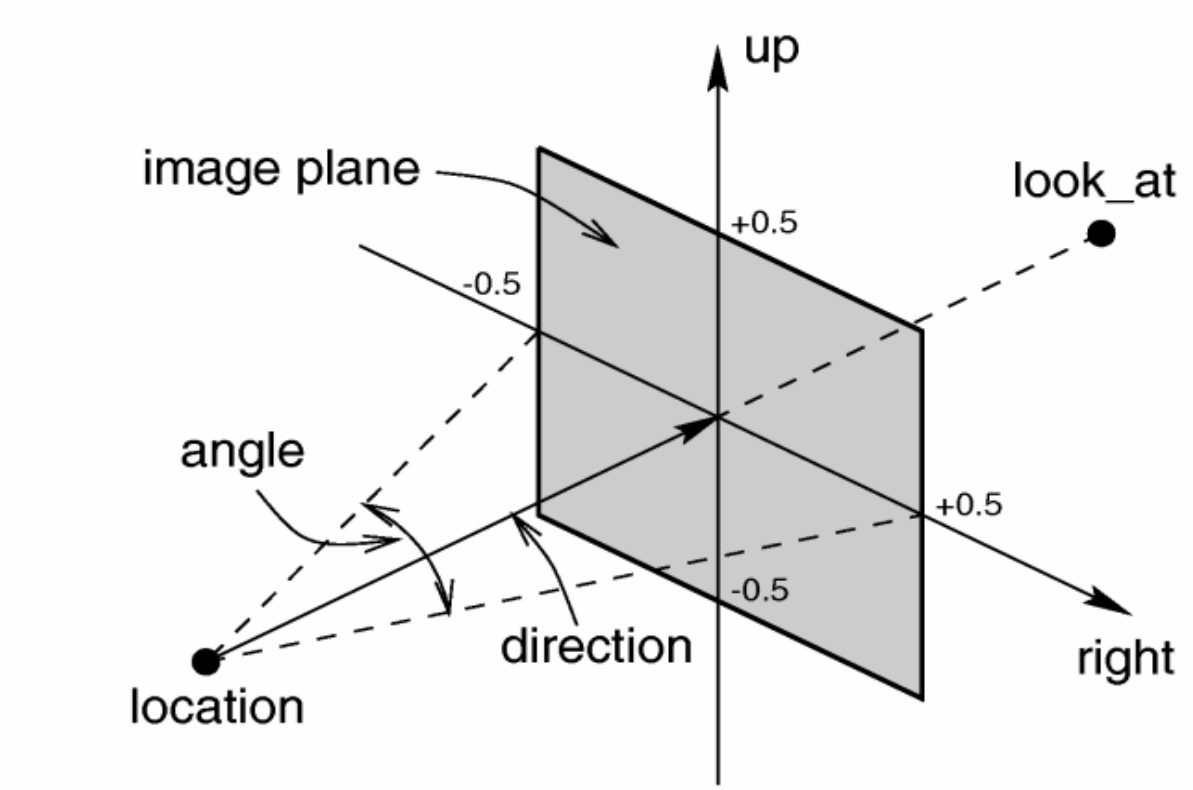


Fig. 5.1 : Perspective

6.0 RAY TRACING

6.1 RAY CREATION

To start ray tracing first a ray needs to be created,

The ray tracing starts from

x=0
y=0

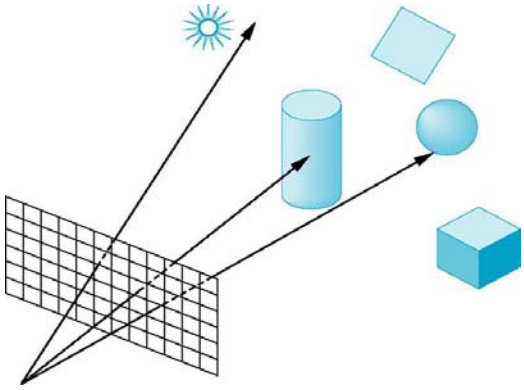


Fig. 6.1 : Shooting a Ray Through Viewport

P(x,y) is an arbitray pixel.

Moving to the center of the pixel,

x += 0.5
y -= 0.5

Converting the x and y coordinate to be between -0.5 to 0.5.

x0 = x / Screen_Width - 0.5;

y0 =(Screen_Height - 1) - y) / (DBL)Frame.Screen_Height - 0.5;

Projecting x0 on the right vector and y0 on the up vector and moving the ray from its previous location to pass through the center of pixel(x,y),

Ray->Direction=Dot(Ray->Direction,1.0)+ Dot(Right, x0), Dot (Up, y0)

6.2 Equation of a Ray

A ray is defined by an origin or eye point, , and an offset vector, $\mathbf{D} = (x_D, y_D, z_D)$.

$\mathbf{E} = (x_E, y_E, z_E)$

The equation for the ray is:

$$\mathbf{P}(t) = \mathbf{E} + t\mathbf{D}, t \geq 0$$

This is equivalent to the three equations:

$$\left.\begin{aligned} x(t) &= x_E + tx_D \\ y(t) &= y_E + ty_D \\ z(t) &= z_E + tz_D \end{aligned}\right\} t \geq 0$$

7.0 POV-RAY OBJECTS

POV-ray defines most objects mathematically. Meshes of triangles can also be constructed.

7.1 Finite Solid Object:

BLOB | BOX | CONE | CYLINDER | HEIGHT_FIELD | JULIA_FRACTAL | LATHE | PRISM | SPHERE | SPHERE_SWEEP | SUPERELLIPSOID | SOR | TEXT | TORUS

7.2 Infinite Solid Object:

PLANE | POLY | CUBIC | QUARTIC | QUADRIC

7.3 Finite Patch Object:

BICUBIC_PATCH | DISC | MESH | MESH2 | POLYGON | TRIANGLE | SMOOTH_TRIANGLE

Describe a totally thin, finite shape.

7.4 Isosurface

Describe a surface via a mathematical function.

7.5 Parametric

Describe a surface using functions to locate points on the surface.

7.6 CSG

Describe one complex shape from multiple shapes.

UNION | INTERSECTION | DIFFERENCE | MERGE

8.0 RAY TRACING OBJECTS

Most primitives are ray traced assuming they are centered in the origin. This is because of the implicit equations are defined such that they are centered in the origin. Ray tracing those objects simply means replacing the explicit equation of the ray in the implicit equation of the object. However, an object may go through transformations or may not be in the ideal position. POV-ray assumes a separate co-ordinate system for each of its objects. In that co-ordinate system the objects are in their ideal position. So, we can say if an object is not centered in the origin, it was transformed from the origin to its current location. The ray is defined in another co-ordinate system. To ray trace the object the ray is transformed to the objects space.

To do this the inverse of the transformation is multiplied with the ray. Except sphere and box all other primitive 3D objects goes through a scaling, rotation and translation to be in its current co-ordinate system. This transformation is defined while parsing the object so that, it can be used to inverse transform the ray to objects space. To find the normal on the point, the point is transformed to object space and normal is computed and then it is transformed back to the world space.

8.1 Sphere

POV-Ray Syntax:

The syntax of the sphere object is:

```
SPHERE:
  sphere
  {
    <Center>, Radius
    [OBJECT_MODIFIERS...]
  }
```

Where *<Center>* is a vector specifying the x, y, z coordinates of the center of the sphere and *Radius* is a float value specifying the radius. Spheres may be scaled unevenly giving an ellipsoid shape.

Because spheres are highly optimized they make good bounding shapes (if manual bounding seems to be necessary).

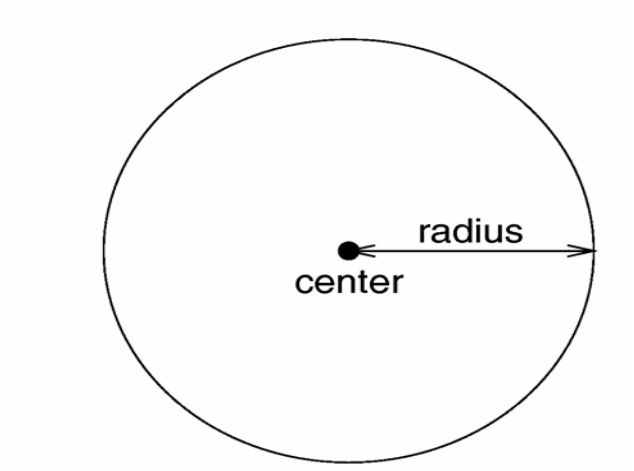


Fig. 8.1 : A Sphere

Sphere Data Structure:

```
struct Sphere_Struct
{
  OBJECT_FIELDS
  VECTOR Center;
  DBL    Radius;
}
```

Ray Sphere Intersection

Ray sphere intersection is very simple. It can be computed in both by algebra and geometry. In the algebraic solution the ray is intersected with the sphere centered in the origin. The ray equation is substituted in the implicit sphere equation

$x^2+y^2+z^2=1$……

$(x_E+tx_D)^2+(y_E+ty_D)^2+(z_E+tz_D)^2=1$

t is found by solving the quadratic equation.

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Where $a=x_D^2+y_D^2+z_D^2$, $b=2x_Ex_D+2y_Ey_D+2z_Ez_D$, and $c=x_E^2+y_E^2+z_E^2-1$

This gives zero, one, or two real values for *t*. If there are zero real values then there is no intersection between the ray and the sphere.

The geometric solution is considered better than the algebraic solution. For algebraic solution the sphere needs to be centered in origin, which is not required for geometric solution. So, a transformation needs to be defined and to solve the equation is computationally more expensive then solving it

geometrically. In some machines division operation is expensive. Pov-ray tries to avoid division operations as much as possible. These are the reasons for preferring geometric solutions over algebraic.

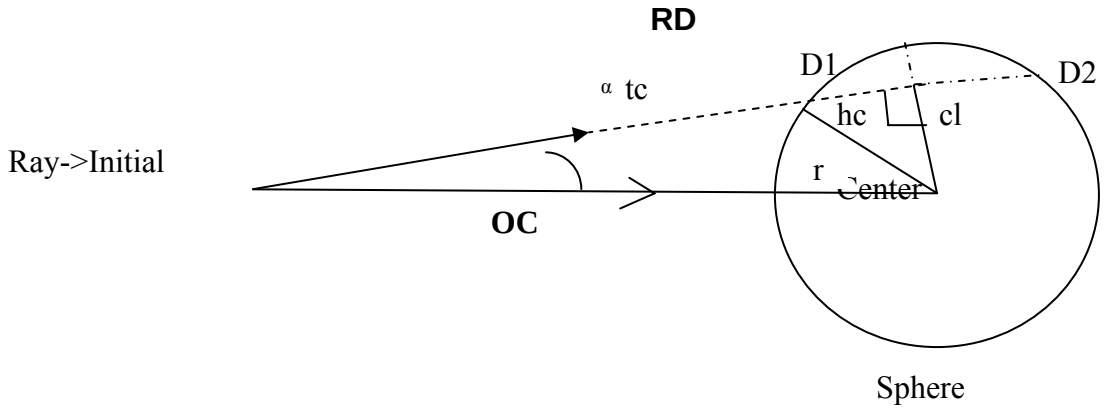


Fig. 8.2 : Ray-sphere intersection calculations

The Ray->Initial is where the camera is placed and the ray is coming from. RD is the ray that intersects the sphere. hc is the half of the chord created by intersection of the ray. The spheres radius is perpendicular on it. D1 and D2 are the value of the depths of the spheres intersections.

We can find the depth of the intersection if we know the distance of the ray origin to the closest point to the center of sphere. The closest point to the center on the line the ray is part of is cl. Let, the distance from ray origin to cl is tc.
So, if we know tc and hc then the depth values are
D1=tc-hc
D2=tc+hc

Dot(**OC,OC)=**length(OC)

=>Dot**(OC,RD)=**length(**OC**)*length(**RD**)*cos(α)

=>Dot**(OC,RD)**=length(OC)*tc/length(**OC**) [Since length(**RD**)=1]

=> Dot**(OC,RD)**=tc

If the ray origin is outside then length(OC) must be greater than R and therefore tc must be a positive value since α<90. So, if tc is negative we can say the ray will not intersect the sphere. If the ray origin is inside the sphere then α can be greater than 90. So, a negative value of tc is accepted.

In both of the case,

$hc^2 = r^2 - (length(OC))^2 + tc^2$

So,

D1=RD-HC

D2=RD+HC

The normal on the sphere is the direction from the center to intersection point.

Inside sphere test

To check if a point is inside a sphere or not, a line is drawn from the center of the sphere to the point. If the squarred length of the line is less than or equal to the radius then the point is inside the sphere. If inverted otherwise.

8.2 Box

A simple box can be defined by listing two corners of the box using the following syntax for a box statement:

POV-Ray Syntax:

```
BOX:
   box
   {
      <Corner_1>, <Corner_2>
      [OBJECT_MODIFIERS...]
   }
```

Where *<Corner_1>* and *<Corner_2>* are vectors defining the x, y, z coordinates of the opposite corners of the box.

Box Structure

```
struct Box_Struct
{
  OBJECT_FIELDS
  VECTOR bounds[2];
}
```

Ray Box Intersection

The process of intersecting a ray with a cube is essentially the Cyrus_beck algorithm. This is the method used everywhere for ray box intersection with slight

varification. The basic idea is that each plane of the cube defines an inside half space and an outside half space and that a point on ray lies inside the cube if and only if it lies on the inside of every half space of the cube. So intersecting a ray with a cube is a matter of finding the interval of time in which the ray lies inside all of the palnes on the cube.

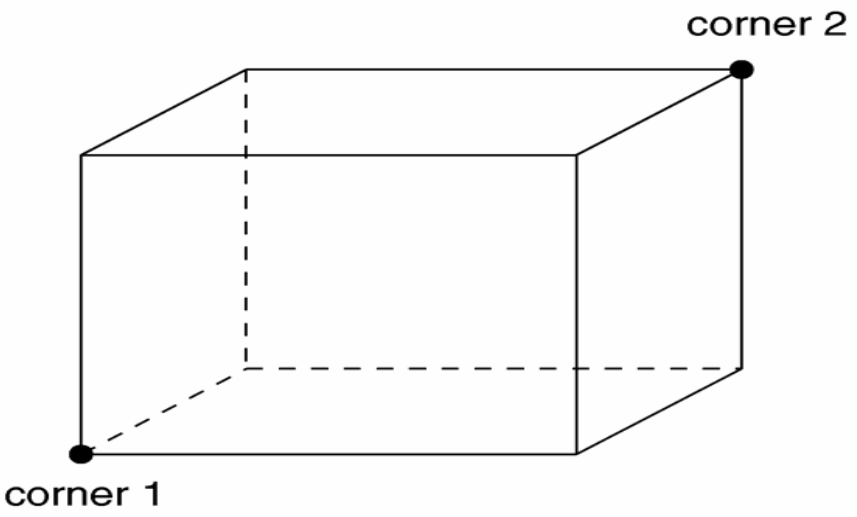
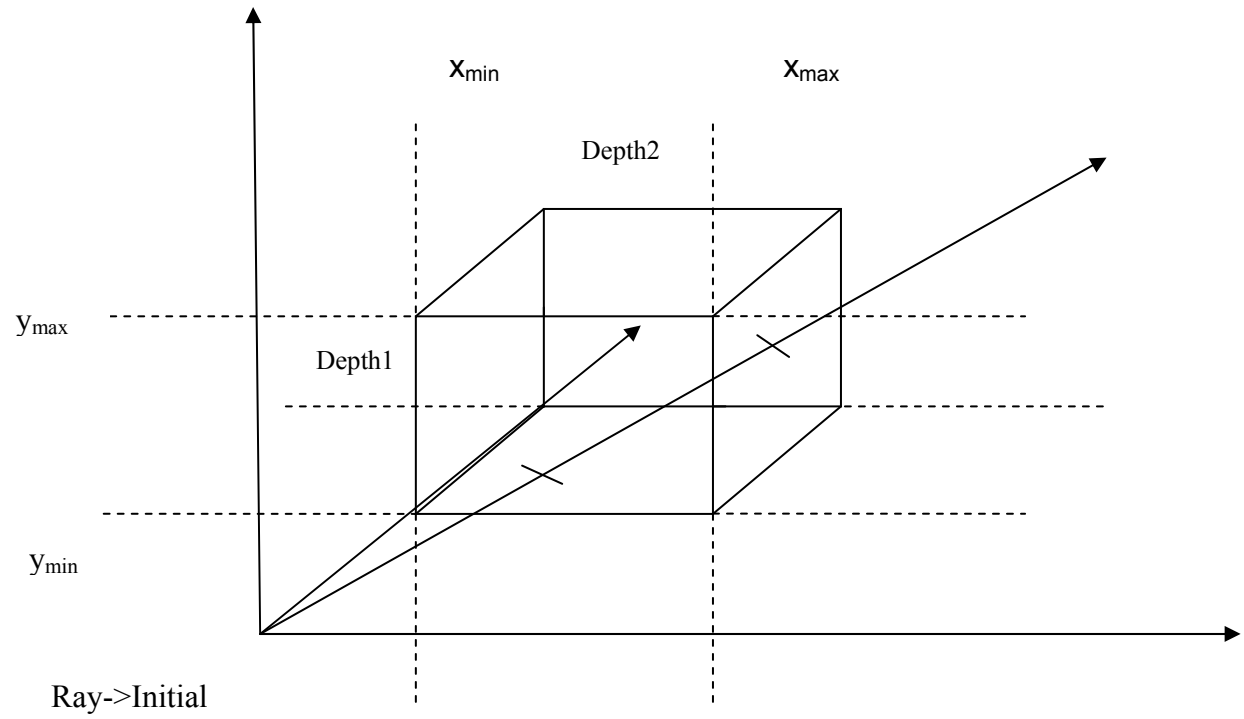


Fig. 8.3 : A Box

The ray is transformed to the boxes space. Each of the boxes faces are parallel to the co-ordinate axes in which they reside. So, for testing intersection with the faces of the boxes they only use the co-ordinate value of the axes that is perpendicular to the faces. The left and right sides are parallel to y-z plane, top and bottom to z-x plane and front and back to x-y plane and intersection calculations is performed with x, y and z co-ordinates respectively.



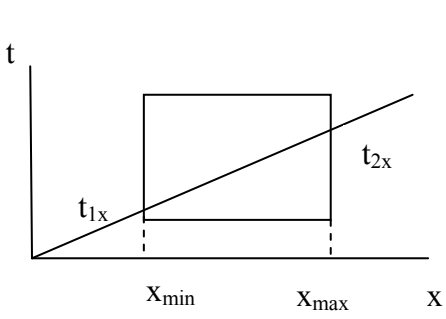


Figure:A

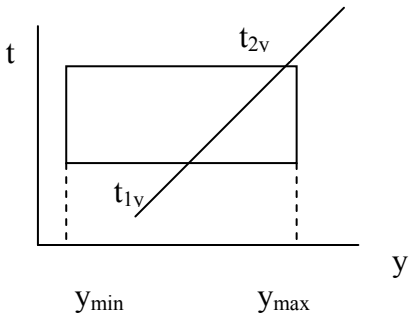


Figure:B

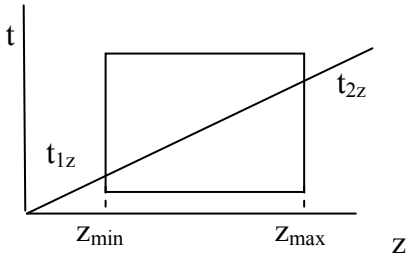


Figure:C

Fig. 8.4 : Ray box intersection

In figure “A” the sides of the box are checked for intersection, in figure “B” top and bottom of the box are checked and in figure “C” front and back of the box are checked.

Depth1 = max(t_{1x},t_{1y},t_{1z})

Depth2 = min(t_{2x},t_{2y},t_{2z})

General equation

t1 = (lowest bound-P)/D.

t2= (highest bound-P)/D

Where

Lowest bound= Smallest co-ordinates of one corner of the box.

Highest bound= Largest co-ordinates of one corner of the box.

Before performing plane intersection tests the sign of the ray co-ordinate is checked to find the direction of the ray. If positive sign then

tmin = (lowest bound-P)/D

tmax= (highest bound-P)/D

If negative

tmax = (lowest bound-P)/D

tmin= (highest bound-P)/D

It means if the ray co-ordinate value is positive it will first intersect either the left, front or bottom first. If it is negative then it will intersect the right, back or top first.

In this process if tmax<tmin then there was no intersection.

Normal

It is important to keep track of the sides that associated with tmin and tmax. It is used later to find the normal on the box surface. The normals on the surface according to the side hit are,

Face	
Left	-1.0, 0.0, 0.0
Right	1.0, 0.0, 0.0
Bottom	0.0, -1.0, 0.0
Top	0.0, 1.0, 0.0
Front	0.0, 0.0, -1.0
Back	0.0, 0.0, 1.0

Inside box test

Transform point into box space. Check if the point is inside the minmum and maximum values of x, y and z co-ordinate.

8.3 Cylinder

The cylinder statement creates a finite length cylinder with parallel end caps The syntax is:

```
CYLINDER:
  cylinder
  {
    <Base_Point>, <Cap_Point>, Radius
    [ open ][OBJECT_MODIFIERS...]
  }
```

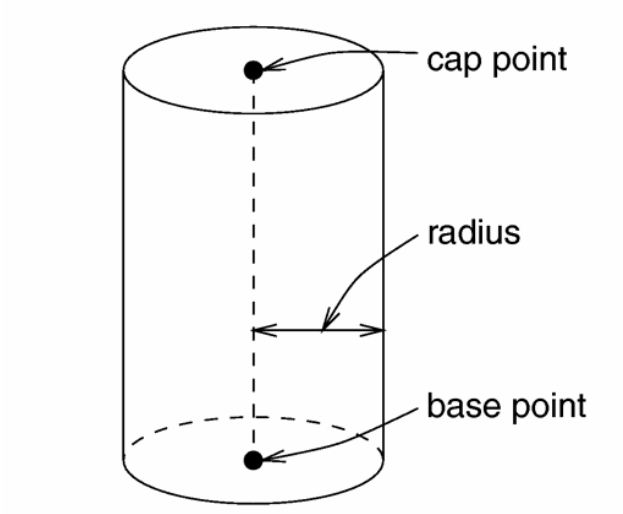


Fig. 8.5 : A Cylinder

Cylinder Structure:

```
struct Cone_Struct
{
 OBJECT_FIELDS
 VECTOR apex;       /* Center of the top of the cone */
 VECTOR base;       /* Center of the bottom of the cone */
 DBL apex_radius;   /* Radius of the cone at the top */
 DBL base_radius;   /* Radius of the cone at the bottom */
 DBL dist           /* Distance from the origin to the base point of the
cone in canonical co-ordiante*/
```

The ideal cylinder is

Base(0,0,0)

Apex(0,0,1)

Apex_radius=base_radius=1

Dist=0

Transformation:

The axis of cone is

Axis=Apex-base

Axis is normalized.

Length=length(axis)

However, the cylinder might not be defined in that position. Pov-ray assumes that the cylinder was in its ideal position before being transformed to the original position. So, the inverse of the transformation would bring the cylinder from its current location to ideal location.

Let us consider the transformation of the cylinder from the ideal location to the current location. Each time the cylinder is moving from to a new co-ordinate system.

First the cylinders base is translated to the origin.

T (base)

The cylinder is rotated so that it is aligned with its current axis moving from the z axis. The rotation is performed with respect to the perpendicular vector of the axis and the angle is, Cos^{-1}(axis[z]).

If absolute value axis[z]<1-epsilon, epsilon is a very small value positive value, then it means the other co-ordinate value is negligible. Which means the cylinder is already aligned with z axis.

axis <0,0,1> if axis[z]>0

axis <0,0,-1> if axis[z]<0

Rotation is not needed for this cylinder. But, since a general case is considered, the rotation is performed with respect to x axis by angle $Cos^{-1}(axis[z])= 0$.

Otherwise the axis vector is projected to xy plane and it becomes axis <x,y,0>. The perpendicular of this vector is <-y,x,0>.

So, the rotation is,

$R\ Cos^{-1}(axis[z])\ axis^{\perp}$

Finally, the cylinder is scaled by radius in x and y axis and by length in z axis to achieve the current shape.

S (radius, radius, length)

So,

Transformation =SRT

Inverse transformation=$S^{-1}R^{-1}T^{-1}$

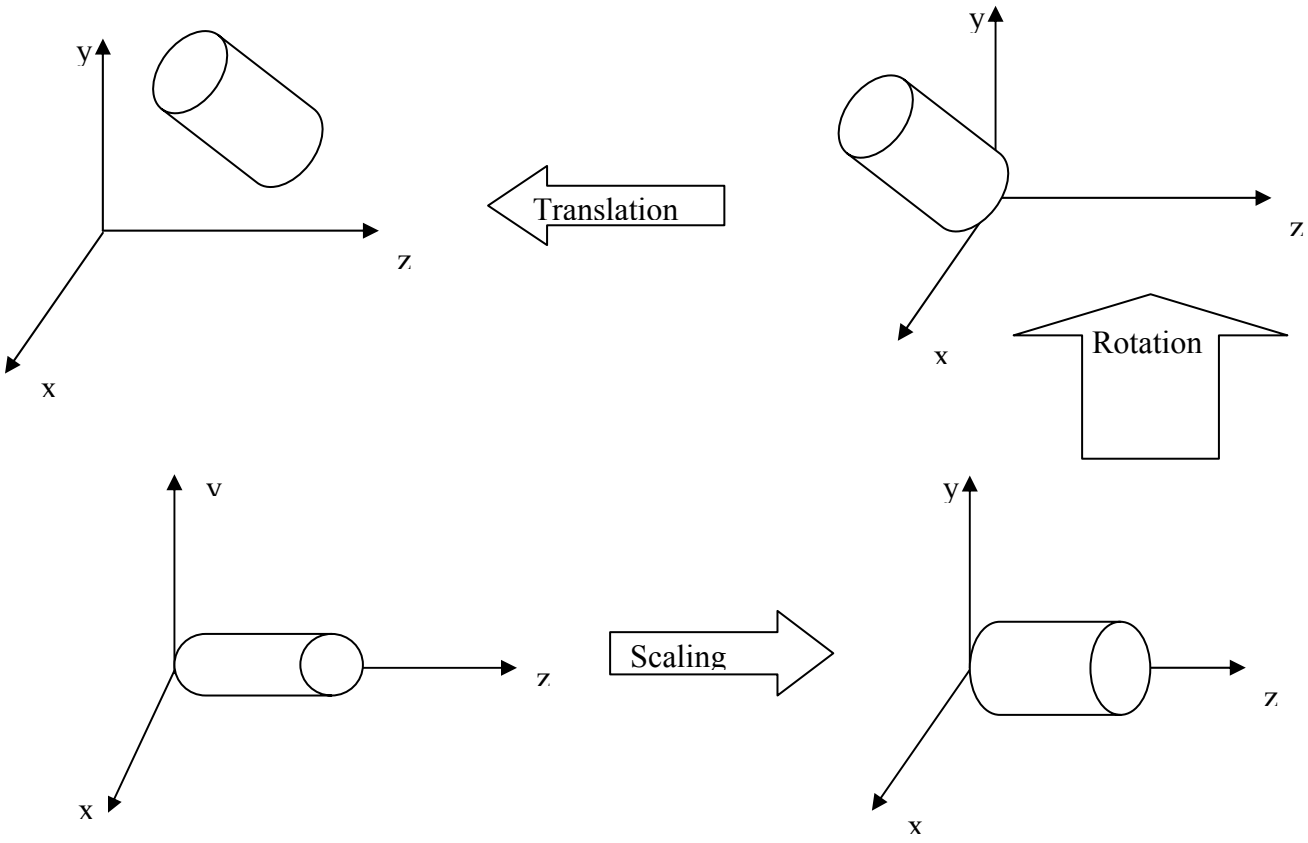


Fig. 8.6 : Cylinder Transformation From Ideal Position to Current Location

Ray Tracing Cylinder

Side hit:

To compute the intersection of the ray with the cylinder equation is replaced by the ray equation,

$(x_E+tx_D)^2+(y_E+ty_D)^2=1$

t1and t2 can be computed by inserting

$a = x_D*x_D + y_D*y_D$

$b = x_E*x_D + y_E*y_D$

$c = x_E*x_E + y_E*y_E-1$

in this two eqautions,

$t1=-b+\sqrt{(b^2-a*c)}/a$

$t2=-b-\sqrt{(b^2-a*c)}/a$

this is the reduced form of quadratic root finder equation,

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$z=z_E + t1*z_D$

$z = z_E + t2*z_D$

Base or cap hit:

The cylinder by default is closed by discs at its ends.

Two compute the base and cap hit,

$z_E +t *z_D = z_{min} /z_{max}$

For base hit

$t = (z_{min} - z_E) / z_D;$

For cap hit

$t = (z_{max} - z_E) / z_D;$

$z_{min}=0$

$z_{max}=1$

For both base and cap hit it is checked if $a^2+b^2<=1$. If the condition is fulfilled than the ray intersects the cap.

Normal

Normal on a point p(x,y,z) is,

```
case SIDE_HIT:
   normal(x,y,0)

case BASE_HIT:
   normal(0.0, 0.0, -1.0)

case CAP_HIT:
   normal(0.0, 0.0, 1.0)
```

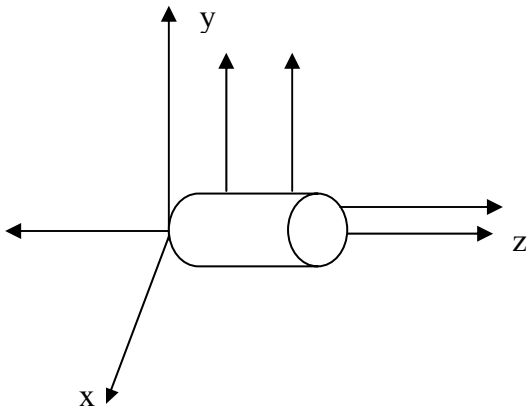


Fig 8.7 : Normals on the cylinder

Inside Cylinder

A point p(x,y,z) will be inside the cylinder if,

$x^2+y^2<=1$

and 0<z<1

8.4 Cone

POV-Ray Syntax:

The cone statement creates a finite length cone or a *frustum* (a cone with the point cut off). The syntax is:

```
CONE:
   cone
   {
      <Base_Point>, Base_Radius, <Cap_Point>, Cap_Radius
      [ open ][OBJECT_MODIFIERS...]
   }
```

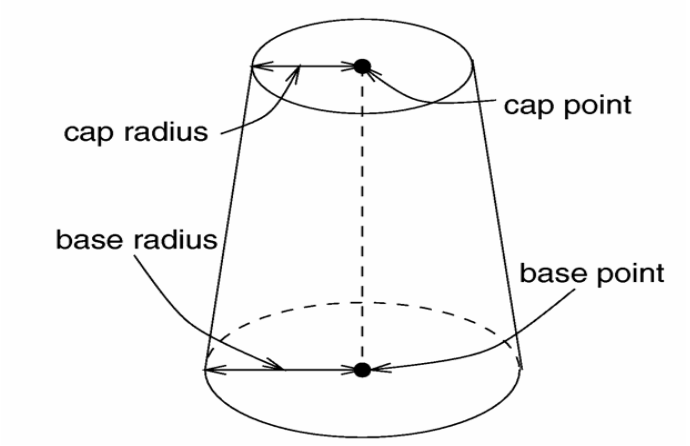


Fig.8.8 : A cone

Cone Structure:

The same structure of the cylinder is used.

The ideal cone has the following values,

Base(0,0,0)

Apex(0,0,1)

Apex_radius=1

Base_radius=0

Dist=0

Transformation:

The cones defined in POV-Ray has two radius values. The base radius may not be 0. So, to make the base radius 0, the length of the cone is extended so that, the cone converges to base radius 0.

axis=Cone->apex, Cone->base

Axis is normalized.

Determining alignment,

tmpf = Cone->base_radius * length(axis) / (Cone->apex_radius - Cone->base_radius);

Origin=axis*tmpf

The new origin of the cone where base radius is 0,

Origin=Cone->base-origin

The extended length of the cone

tlen = tmpf + len

The distance of the actual base point from the origin.

Cone->dist = tmpf / tlen

The transformations defined are same as that of the cylinder.

Ray Tracing Cone

The *infinite* double cone aligned along the *z*-axis is defined as:

$$x^2+y^2=z^2$$

To intersect a ray with the cone, cone equation is replace by the ray equation,

$$(x_E+tx_D)^2+(y_E+ty_D)^2=(z_E+tz_D)^2$$

Side hit

Placing

$a = x_D*x_D + y_D*y_D - z_D*z_D$

$b = x_E*x_D + y_E*y_D - z_E*z_D$

$c = x_E*x_E + y_E*y_E - z_E*z_E$

in,

if a=0

t = -0.5 * c / b

and,

t1 = (-b - d) / a

t2 = (-b + d) / a

The above equations can be deduced from

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

$z=z_E + t1*z_D$

$z = z_E + t2*z_D$

If the z values are within the range of cone->dist to 1 than we obtain the intersection depth by

Depth= t1 / len(D)

Base or cap hit:

The cone by default is closed by discs at its ends.

Two compute the base and cap hit,

$z_E + t * z_D = z_{min} / z_{max}$

For base hit

$t = (z_{min} - z_E) / z_D;$

For cap hit

$t = (z_{max} - z_E) / z_D;$

z_{min}=cone->dist

z_{max}=1

$a=x_E + t * x_D$

$b = y_E + t * y_D$

For cap hit checked if $a^2+b^2<=1$, for base hit checked if $a^2+b^2<=$cone->dist. If the condition is fulfilled than the ray intersects the cap.

Inside Cone

Inside the cone if

$x^2+y^2<=z^2$

cone->dist<=z<=1

Normal

Normal on point p(x,y,z) is,

```
case SIDE_HIT
    normal(x,y,-z)
case BASE_HIT
    normal(0,0,-1)
```

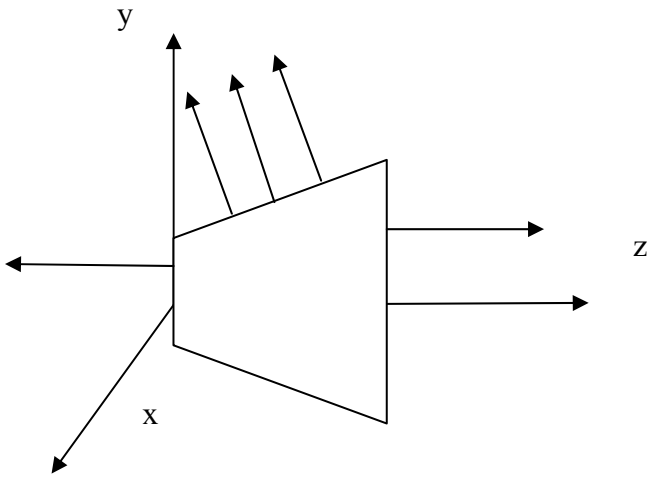


Fig. 8.9 :Cone normals on different points

```
case CAP_HIT

    normal(0,0,1)
```

8.5 Plane

POV-Ray Syntax:

The plane primitive is a simple way to define an infinite flat surface. The plane is not a thin boundary or can be compared to a sheet of paper. A plane is a solid object of infinite size that divides POV-space in two parts, inside and outside the plane. The plane is specified as follows:

```
PLANE:
  plane
  {
    <Normal>, Distance
    [OBJECT_MODIFIERS...]
  }
```

The *<Normal>* vector defines the surface normal of the plane. A surface normal is a vector which points up from the surface at a 90 degree angle. This is followed by a float value that gives the distance along the normal that the plane is from the origin (that is only true if the normal vector has unit length; see below).

Plane Structure

```
struct Plane_Struct
{
  OBJECT_FIELDS
  VECTOR Normal_Vector;
```

```
  DBL    Distance;
};
```

Ray Tracing Plane

The distance is always multiplied by -1.

Intersection

If the distance along the normal vector on the surface of the plane is d and P(t) is a point in the plane then we can write,

N.P+d=0……..(1)

Let the ray intersects point P.

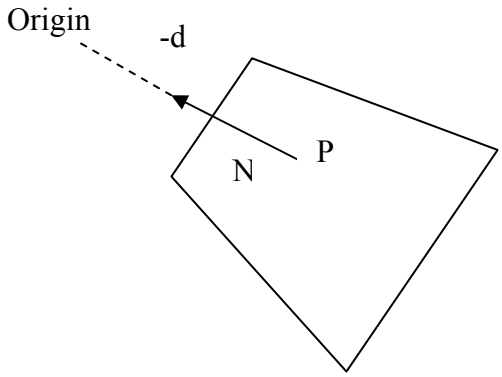


Fig. 8.10: A plane

So,

P=E+Dt

A ray will intersect the plane if,

N.(E+Dt)+d=0

N.D= Dot(Plane normal , Ray->direction)

N.E= Dot(Plane normal, Ray->initial)=projection of ray->initial on normal vector

Depth=-(N.E+ d)/ N.D

The normal vector points to the outside of a plane.

Inside Plane:

The normal points outward of the plane. The point is projected on the planes normal, which gives the distance from the oeigin. So, if the addition of the distance of the plane and distance of the point is<=0, it means the point is inside the plane.

8.6 Disc

Disc is a flat, finite object available with POV-Ray. The disc is infinitely thin, it has no thickness. If you want a disc with true thickness you should use a very short cylinder.

POV-Ray Syntax:

A disc shape may be defined by:

```
disc
{
   <Center>, <Normal>, Radius [, Hole_Radius]
   [OBJECT_MODIFIERS...]
```

The hole radius is optional.

Disc Structure:

```
struct Disc_Struct
{
  OBJECT_FIELDS
  VECTOR center;    /* Center of the disc */
```

```
  VECTOR normal;    /* Direction perpendicular to the disc (plane normal) */
  DBL d;            /* The constant part of the plane equation */
  DBL iradius2;     /* Distance from center to inner circle of the disc squarred*/
  DBL oradius2;     /* Distance from center to outer circle of the disc */
};
```

The ideal disc has the following values,

Center (0.0, 0.0, 0.0)

Normal (0.0, 0.0, 1.0);

iradius2 = 0.0

oradius2 = 1.0

d = 0.0

Transformation:

A transformation is defined for the disc. An ideal disc should be centered in the origin, lying in xy plane, facing the positive z axis with outer radius 1. So, if the disc is not in the ideal position, it is assumed to have gone through a transformation.

The discs was translated to its current location from the origin,

T(center)

The scaling is not needed for disc. However, for general case,

S(1,1,1)

The disc was rotated by $\cos^{-1}$(normal[z]) from its original location to be on the current location,

R $\cos^{-1}$(normal[z]) normal$^{\perp}$

So,

Transformation =SRT

Inverse transformation=$S^{-1}R^{-1}T^{-1}$

Ray Disc Intersection:

Intersection of the ray with the disc plane is found by,

$t=-y_E/y_D$

Now, to check if the point lies on the disc,

$u = x_E + t * x_D$

$v = y_E + t * y_D$

r^2 = Sqr(u) + Sqr(v);

if (inner radius)^2<r^2<(outer radius)^2 then the depth is

d=t/length(Ray->direction)

The normal to the intersection point is the assigned normal of the disc.

Inside Disc

The discs inside is defined the same way as the plane. Every point that is on the opposite of the discs normal is inside the disc and the others are outside the disc.

Normal

The disc surface normal is the normal on the intersection point.

8.7 Polygon

POV-Ray Syntax:

The polygon object is useful for creating rectangles, squares and other planar shapes with more than three edges. Their syntax is:

POLYGON:

```
polygon
{
   Number_Of_Points, <Point_1> <Point_2>... <Point_n>
   [OBJECT_MODIFIER...]
}
```

The float *Number_Of_Points* tells how many points are used to define the polygon. The points *<Point_1>* through *<Point_n>* describe the polygon or polygons. A polygon can contain any number of sub-polygons, either overlapping or not. In places where an even number of polygons overlaps a hole appears. When the first point of a sub-polygon is repeated, it closes it and starts a new sub-polygon's point sequence. This means that all points of a sub-polygon are different.

If the last sub-polygon is not closed a warning is issued and the program automatically closes the polygon. This is useful because polygons imported from other programs may not be closed, i.e. their first and last points are not the same.

All points of a polygon are three-dimensional vectors that have to lay on the same plane. If this is not the case an error occurs. It is common to use two-dimensional vectors to describe the polygon. POV-Ray assumes that the z value is zero in this case.

Polygon Structure:

```
Polygon_Data_Struct
{
  int References;
  int Number;
  UV_VECT *Points;
};

Polygon_Struct
{
  OBJECT_FIELDS
```

```
  VECTOR S_Normal;
  POLYGON_DATA *Data;
};
```

Transformation:

While parsing the polygon is closed if not already closed.

In ray tracing and other applications the original polygon is defined in three dimensions. To simplify computation it is worthwhile to project the polygon and test point into two dimensions. One way to do this is to simply ignore one component.

Make two vectors by connecting the first point o with two other points of the polygon. Find three vectors **u, v, w** such that they are perpendicular to each other and the origin is the first point.

These two matrices are defined for the polygon

Matrix a				Matrix b			
1	0	0	0	u[x]	v[x]	w[x]	0
0	1	0	0	u[y]	v[y]	w[y]	0
0	0	1	0	u[z]	v[z]	w[z]	0
-o[x]	-o[y]	-o[z]	1	0	0	0	1

Poly-trans->inverse=a*b

The polygon goes through this transformation to be put in its own co-ordinate system where the unit vectors are u, v and w and the origin is o. Matrix b puts the polygon into u, v and w co-ordinate system and matrix a scales it

accordingly. This will be used in ray polygon intersection test to transform the ray to polygon axis.

This is the transformation defined for the polygon to bring it to the x, y and z co-ordinate system from its own co-ordinate system.

Poly-trans= $(a*b)^{-1}$

Transforming the polygon into the u v vector space by doing this

```
Polyg->Data->Points[i][X] = x * u[X] + y * u[Y] + z * u[Z];

Polyg->Data->Points[i][Y] = x * v[X] + y * v[Y] + z * v[Z];
```

Ray Polygon Intersection

The ray is transformed to polygons space.

Find the depth of the intersection with ray and polygon plane,

```
t = -p[Z] / d[Z];
```

Find the intersection points x and u co-ordinate

```
x = p[X] + t * d[X];
y = p[Y] + t* d[Y];
```

Now, what remains is to check whether the point is inside the polygon or not Point Inside Polygon Strategies**.** [17]

There are two different types of polygons, convex and non-convex. A polygon is convex if a line between any two points inside the polygon also lies entirely inside the polygon.

One definition of whether a point is inside a region is the *Jordan Curve Theorem*. Essentially, it says that a point is inside a polygon if, for any ray from this point, there is an odd number of crossings of the ray with the polygon's edges. This definition means that some areas which are enclosed by a polygon are not considered inside. The center pentagonal area inside a star is classified as outside because any ray from this area will always intersect an even number of edges.

If the entire area enclosed by the polygon is to be considered inside, then the *winding number* is used for testing. This value is the number of times the polygon goes around the point. For example, a point in the center pentagonal area formed by a star has a winding number of two, since the outline goes around it twice. If the point is outside, the polygon does not wind around it and so the winding number is zero. Winding numbers also have a sign, which corresponds to the direction the edges wrap around the point.

Complex polygons can be formed using either polygon definition. A complex polygon is one that has separate outlines (which can overlap). For example, the letter "R" can be defined by two polygons, one consisting of the exterior outline, the other the inside hole in the letter. Most point in polygon algorithms can be easily extended to test multiple outline polygons by simply running each polygon outline through separately and keeping track of the total parity.

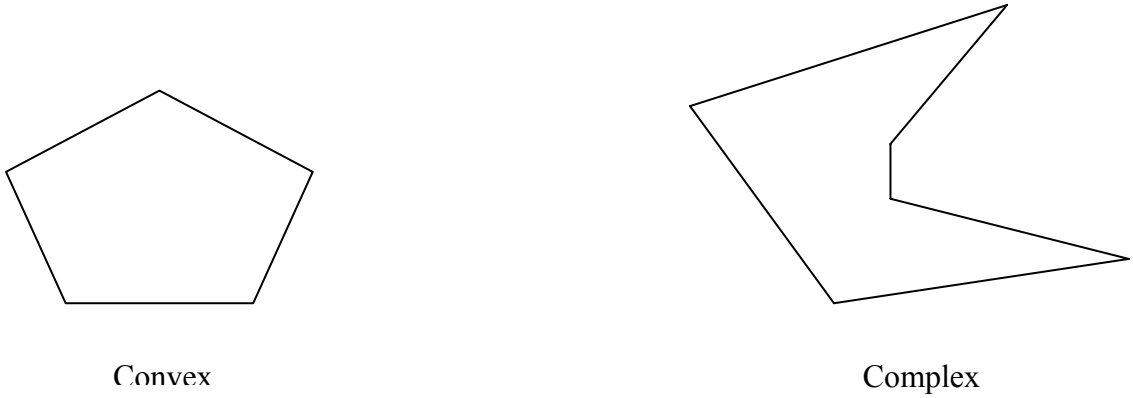


Fig. 8.11 : Convex and complex polygon

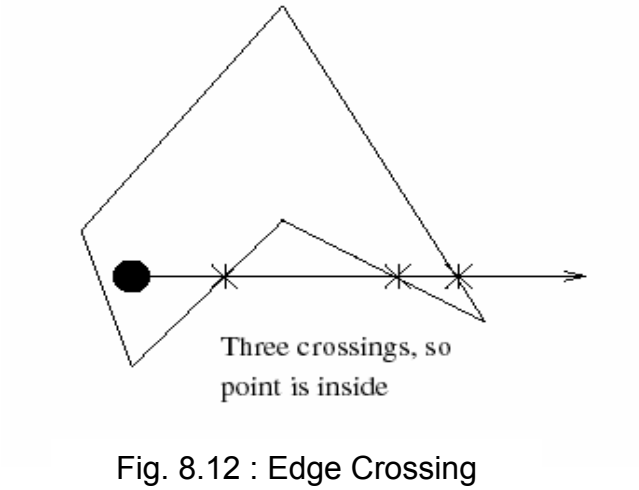


Fig. 8.12 : Edge Crossing

1. Angle Summation Test

The worst algorithm in the world for testing points is the angle summation method. It's simple to describe: sum the signed angles formed at the point by each edge's endpoints. If the sum is near zero, the point is outside; if not, it's inside. The problem with this scheme is that it involves a square root, arc-cosine, division, dot and cross product for each edge tested.

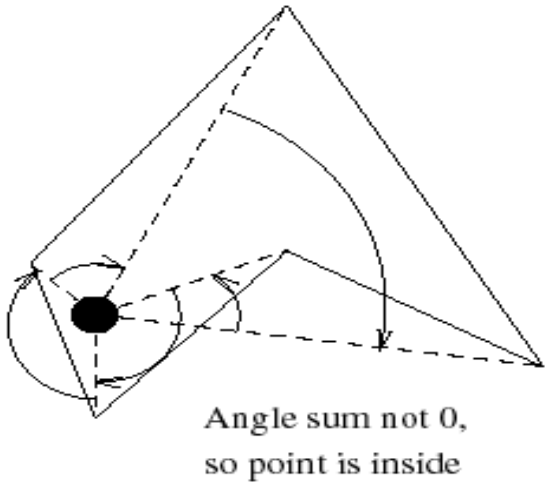


Fig. 8.13 : Angle summation test

2. Triangle Tests

In *Graphics Gems*, Didier Badouel presented a method of testing points against convex polygons. The polygon is treated as a fan of triangles emanating from one vertex and the point is tested against each triangle by computing its barycentric coordinates. As Berlin pointed out, this test can also be used for non-convex polygons by keeping a count of the number of triangles which overlap the point; if odd, the point is inside the polygon. Unlike the convex test, where an

intersection means that the test is done, all the triangles must be tested against the point for the non-convex test. Also, for the non-convex test there may be multiple barycentric coordinates for a given point, since triangles can overlap.

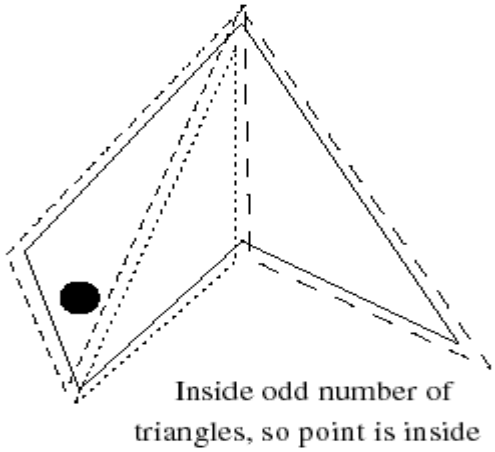


Fig. 8.14 : Triangle Fan Test

A faster triangle fan tester proposed by Green is to store a set of half-plane equations for each triangle and test each in turn. If the point is outside any of the three edges, it is outside the triangle. The half-plane test is an old idea, but storing the half-planes instead of deriving them on the fly from the vertices gives this scheme its speed at the cost of some additional storage space. For triangles this scheme is the fastest of all of the algorithms discussed so far.

Both the half-plane and barycentric triangle testers can be sped up further by sorting the order of the edge tests. Worley and Haines note that the half-plane triangle test is more efficient if the longer edges are tested first. Larger edges

tend to cut off more exterior area of the polygon's bounding box, and so can result in earlier exit from testing a given triangle. Sorting in this way makes the test up to 1.6 times faster, rising quickly with the number of edges in the polygon. However, polygons with a large number of edges tend to bog down the sorted edge triangle algorithm.

For the general test a better ordering for each triangle's edges is to sort by the area of the polygon's bounding box outside the edge, since we are trying to maximize the amount of area discarded by each edge test. This ordering provides up to another 10% savings in testing time. Unfortunately, for the convex test below, this ordering actually loses about 10% for regular polygons due to a subtle quirk. As such, this ordering is not presented in the statistics section or the code.

3. Crossings Test

One algorithm for checking a point in any polygon is the crossings test. The earliest presentation of this algorithm is Shimrat, though it has a bug in it. A ray is shot from the test point along an axis (+X is commonly used) and the number of crossings is computed. Either the Jordan Curve or winding number test can be used to classify the point. What happens when the test ray intersects one or more vertices of the polygon? This problem can be ignored by considering the test ray to be a half-plane divider, with one of the half-planes including the ray's points. In other words, whenever the ray would intersect a vertex, the vertex is always classified as being infinitesimally above the ray. In this way, no vertices are intersected and the code is both simpler and speedier.

One way to think about this algorithm is to consider the test point to be at the origin and to check the edges against this point. If the Y components of a polygon edge differ in sign, then the edge can cross the test ray. In this case, if

both X components are positive, the edge and ray must intersect and a crossing is recorded. Else, if the X signs differ, then the X intersection of the edge and the ray is computed and if positive a crossing is recorded.

MacMartin pointed out that for polygons with a large number of edges there are generally runs of edges which have Y components with the same sign. For example, a polygon representing Brazil might have a thousand edges, but only a few of these will straddle any given latitude line and there are long runs of contiguous edges on one side of the line. So a faster strategy is to loop through just the Y components as fast as possible; when they differ then retrieve and check the X components. Compared to the basic crossings test the MacMartin test was up to 1.8 times faster for polygons up to 100 sides, with performance particularly enhanced for polygons with many edges.

4. Crossing Multiply algorithm

This algorithm is proposed by Eric Haines. By turning the division for testing the X axis crossing into a tricky multiplication test this part of the test became faster, which had the additional effect of making the test for "both to left or both to right" a bit slower for triangles than simply computing the intersection each time. The main increase is in triangle testing speed, which was about 15% faster; all other polygon complexities were pretty much the same as before. On machines where division is very expensive this test should be much faster overall. Mileage may (in fact, will) vary, depending on the machine and the test data, but in general this code is both shorter and faster. This test was inspired by unpublished Graphics Gems submitted by Joseph Samosky and Mark Haigh-Hutchinson. Related work by Samosky is in: Samosky, Joseph, "SectionView: A system for interactively specifying and visualizing sections through three-dimensional medical image data", M.S. Thesis, Department of Electrical

Engineering and Computer Science, Massachusetts Institute of Technology, 1993.

Shoot a test ray along +X axis. The strategy is to compare vertex Y values to the testing point's Y and quickly discard edges which are entirely to one side of the test ray.

POV-ray implements this algorithm.

An X-ray is shot from the point (tx,ty) and the intersection is calculated with each of the polygon edges.

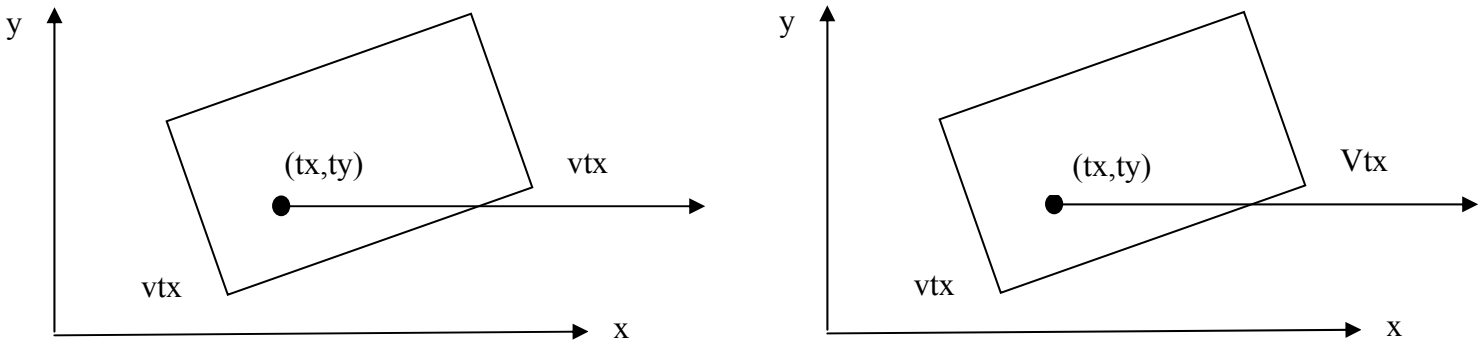
To intersect a ray with polygon edge through point vtx0 and vtx1,

First checks if the end points of the edges have different y values.

```
yflag0 = (vtx0[Y] >= ty);

yflag1 = (vtx1[Y] >= ty);
```

If they don’t have different y values than the vertex is not checked for intersection. So it means the edge is horizontal to x axis. One might think points on the edges might be skipped but it does not happen. Because this point if inside the polygon will be found crossing the next edge.



Orientation anticlockwise Orientation clockwise

Fig. 8.15 : Point Inside Polygon Test

The intersection point between the X+ray and edge (vtx0,vtx1) can be computed by, Intersection= vtx1[X] - (vtx1[Y]-ty)*(vtx0[X]-vtx1[X])/(vtx0[Y]-vtx1[Y])

This equation was formed by simplifying the equation of intersection between two line segments.

The intersection value must be greater than tx for the ray to intersecct the edg. So, it can be written as,

(vtx1[X] - (vtx1[Y]-ty)*(vtx0[X]-vtx1[X])/(vtx0[Y]-vtx1[Y])) >= tx

By cross multiplying the equation the division operation can be avoided,

(vtx1[Y]-ty) * (vtx0[X]-vtx1[X]) >= (vtx1[X]-tx) * (vtx0[Y]-vtx1[Y])

POV-Ray does not change the orientation of the polygon. So, we need to consider yflag1.

So, the point crosses the edge,

if ((vtx1[Y]-ty) * (vtx0[X]-vtx1[X]) >= (vtx1[X]-tx) * (vtx0[Y]-vtx1[Y])= =yflag1)

Every time the point crosses an edge a counter value is increased. If the value is odd then the point is inside the polygon otherwise point is outside polygon.

Normal

The polygon normal is defined to be 0,0,1. This is so because the polygon is projected onto uv plane and w axis is perpendicular to it. The normal is multiplied by the transformation matrix to get it out of the polygons space.

8.8 Triangle

POV-Ray Syntax:

The triangle primitive is available in order to make more complex objects than the built-in shapes will permit. Triangles are usually not created by hand but are converted from other files or generated by utilities. A triangle is defined by

```
TRIANGLE:
  triangle
  {
    <Corner_1>, <Corner_2>, <Corner_3>
    [OBJECT_MODIFIER...]
  }
```

where *<Corner_n>* is a vector defining the x, y, z coordinates of each corner of the triangle.

Triangle Structure:

```
struct Triangle_Struct
{
  OBJECT_FIELDS
  VECTOR  Normal_Vector;
  DBL     Distance;
  unsigned int  Dominant_Axis:2;
  unsigned int  vAxis:2;  /* used only for smooth triangles */
  VECTOR  P1, P2, P3;
};
```

Transformation:

Find two vectors such that

V1= Triangle->P1-Triangle->P2

V2=Triangle->P3-Triangle->P2

Find normal vector on the triangle with the cross product of v1 and v2.

Find the triangle distance or projection of p1 on normal vector.

Triangle->Distance=Dot(Triangle->Normal_Vector, Triangle->P1)

Triangle->Distance *= -1.0

The dominant axis of the triangle is the one where the difference between the points of the triangles is maximum.

The tringle is projected on the plane perpendicular to the dominant axis.

P2

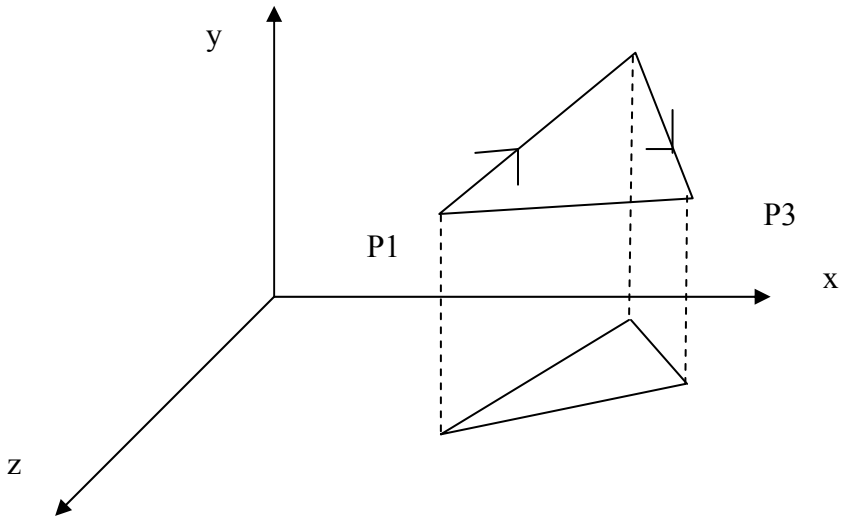


Fig. 8.16 : Projection of triangle from 3D to 2D

The orientation of the triangle must be anticlockwise. It is checked if the defined points by the user actually orient anticlockwise.

On the figure we can see a triangle tha has clockwise rotation. The triangle was projected on xz plane. A triangle that has clockwise orientation, will always have the next point after defining the vector between the first two points on its left side.

Since the dominant axis is y pov-ray uses this condition

```
case Y:

  if ((Triangle->P2[X] - Triangle->P3[X])*(Triangle->P2[Z] - Triangle->P1[Z]) <
     (Triangle->P2[Z] - Triangle->P3[Z])*(Triangle->P2[X] - Triangle->P1[X]))
  {
    swap = true;
```

}

The equation in the condition can be written as

```
(Triangle->P3[X] - Triangle->P2[X])*(Triangle->P1[Z] - Triangle->P2[Z]) <
(Triangle->P3[Z] - Triangle->P2[Z])*(Triangle->P1[X] - Triangle->P2[X])
```

Let,

A[X]= Triangle->P3[X] - Triangle->P2[X]

B[Z]= Triangle->P1[Z] - Triangle->P2[Z]

A[X]= Triangle->P3[Z] - Triangle->P2[Z]

B[Z]= Triangle->P1[X] - Triangle->P2[X]

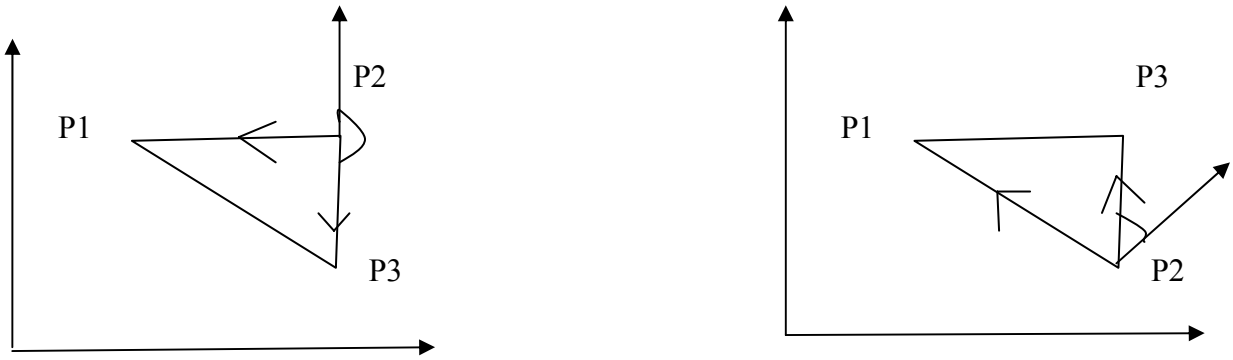
(A[Y] * B[Z] < A[Z] * B[Y])

=>(A[Y] * B[Z] - A[Z] * B[Y])<0

=>(A[Y] * B[Z] + A[Z] *(- B[Y])<0

Which happens to be the dot product of A and a vector perpendicular to B.

Interpreted this way, the test boils down to checking whether the angle between A, and that particular perpendicular to B, is greater or smaller than 90 degrees. If the condition is true then p1 and p2 are swapped, which makes the triangle orient clockwise.



Anticlockwise **Clockwise**

Fig. 8.17 : Anticlockwise and clockwise orientation of a triangle

From the figure the test becomes clear. The test is same for the other dominant axis except the co-ordinates are different.

Ray Triangle Intersection

To find the intersection of a triangle with the ray first the intersection of the ray with the triangle plane is calculated and then it is checked whether the point is inside the triangle or not.

Intersection depth with the triangle plane is,
Depth=-(N.E+ d)/ N.D

The intersection point of the co-ordinate values are calculated and projected on the triangles plane. For each of the triangles edges it is checked whether the point is on their left side. If it is then the point is inside the triangle.

This check is performed the same way as the orientation test.

For dominant y axis,

```
case Y:

    s = Ray->Initial[X] + Depth * Ray->Direction[X];
    t = Ray->Initial[Z] + Depth * Ray->Direction[Z];

    if ((Triangle->P2[X] - s) * (Triangle->P2[Z] - Triangle->P1[Z]) <
       (Triangle->P2[Z] - t) * (Triangle->P2[X] - Triangle->P1[X]))
    {
     return(false);
    }

    if ((Triangle->P3[X] - s) * (Triangle->P3[Z] - Triangle->P2[Z]) <
       (Triangle->P3[Z] - t) * (Triangle->P3[X] - Triangle->P2[X]))
    {
     return(false);
    }

    if ((Triangle->P1[X] - s) * (Triangle->P1[Z] - Triangle->P3[Z]) <
       (Triangle->P1[Z] - t) * (Triangle->P1[X] - Triangle->P3[X]))
    {
     return(false);
    }
```

So, for each of the edges it is checked whether the triangle formed by the point (s,t) and other two points orient anticlockwise or not. If the orientation is anticlockwise then the point is inside the triangle.

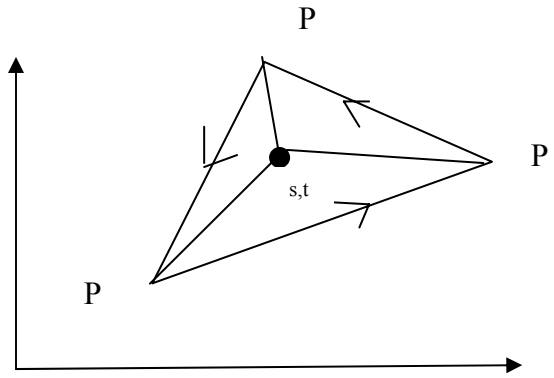


Fig. 8.18 : Point inside a triangle

Normal

The normal vector of the triangle is the perpendicular vector on which the triangle was projected.

8.9 Torus

A torus is a 4th order quartic polynomial shape that looks like a donut or inner tube.

POV-Ray Syntax

TORUS:

```
torus
{
   Major, Minor
   [TORUS_MODIFIER...]
```

}

TORUS_MODIFIER:

sturm | OBJECT_MODIFIER

Torus default values:

sturm : off

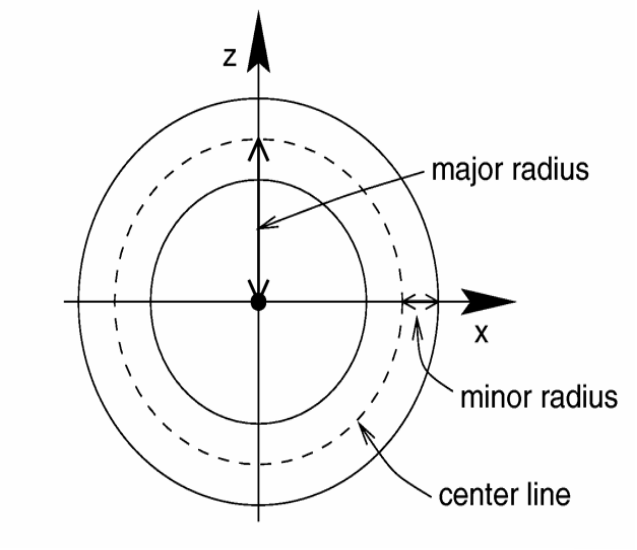


Fig. 8.19 : A Torus

Where *Major* is a float value giving the major radius and *Minor* is a float specifying the minor radius. The major radius extends from the center of the hole to the mid-line of the rim while the minor radius is the radius of the cross-section of the rim. The torus is centered at the origin and lies in the x-z-plane with the y-axis sticking through the hole.

The keyword sturm to uses a slower-yet-more-accurate Sturmian root solver. It may be used if the torus is not displayed properly.

Data Structure

struct Torus_Struct

```
{
  OBJECT_FIELDS
  DBL R, r;
};
```

Ray tracing Torus

Implicit definition of the torus which lies in xz plane is:

$(\sqrt{(x^2+z^2)}-R)^2+y^2=r^2$

This torus lies in the xz plane.

First it checked if the ray intersects a thick cylinder around the torus. The cylinder has inner radius R-r, outer radius R+r and extends from -r to r in y axis. This is done to check the possibility of the ray intersecting the torus. The ray torus intersection equation is a 4^{th} degree polynomial. By checking the intersection with a qudratic object first the computationally expensive root solver can be avoided for rays that passes through the ring of the torus.

A bounding sphere is assumed around the torus with radius R+2r. If the squared distance of the ray initial from is bigger than the sphere radius then the ray initial point is moved to be on the surface of the sphere. This is done for a more precise root calculation.

Closer=Distance(P)-Sphere Radius

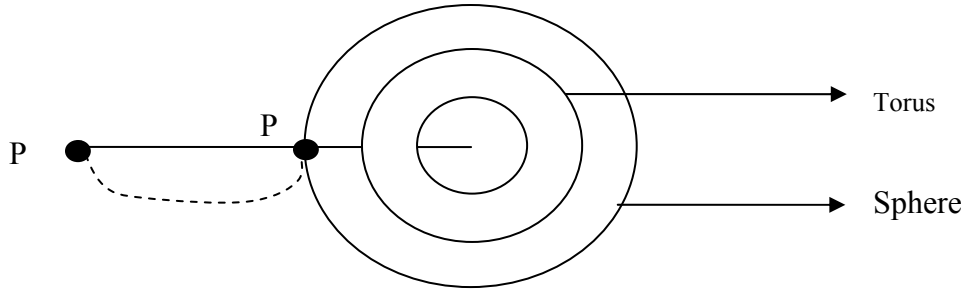


Fig. 8.20 : Moving the ray initial point P on the sphere surface moving closer to torus

Inserting these values in equation (1) we will get a forth degree polynomial equation that is solved to find the intersection of the torus. A maximum of four intersections can be found since the torus is quatric shape.

After inserting $\left.\begin{aligned} x(t) &= x_E + tx_D \\ y(t) &= y_E + ty_D \\ z(t) &= z_E + tz_D \end{aligned}\right\} t \geq 0$

In the torus equation and normalizing it the equation obtained is,

c[0]*t4+c[1]*t3+c[2]*t2+c[3]*t+c[4]=0

Where,

c[0]=1

c[1]= $4*(x_Ex_D+y_Ey_D+z_Ez_D)$

c[2]= $2\{x_E^2 +y_E^2+ z_E^2-R^2-r^2+2*(x_Ex_D+z_Ez_D+y_Ey_D)^2+R^2y_D^2\}$

c[3]= $(x_Ex_D+y_Ey_D +z_Ez_D)(x_E^2+y_E^2+z_E^2 -R^2-r^2)+2R^2y_Ey_D$

c[4]= $(x_E^2+y_E^2+z_E^2 -R^2-r^2)^2+4R^2(y_E^2-r^2)$

The first coefficient c[0] is actually $(x_D^2+y_D^2+z_D^2)^2$. length(**D**)=1. So, c[0]=1

This is a fourth degree polynomial that is solved to find the roots of t.
But the root is not the actual depth of the intersection since the ray initial point was moved.
Depth=(root+closer)/length(**D**)

Inside Torus test

Translate point into torus space. For a point to be inside the torus it has to satisfy the condition,

$(\sqrt{(x^2+z^2)}-R)^2+y^2<=r^2$

Normal

The normal on the torus is computed from derivatives. A vector M is chosen such that,
$M=R*x_P/Distance(P), 0, R*z_P/Distance(P)$
The y component is 0 because the derivative is computed with repset to x and z, since the torus lies on xz plane.
So, normal is,
N=P-M
Nomral is moved out of the torus space by multiplying the transformation matrix to it and value is normalized.

9.0 Constructive Solid Geometry

In addition to all of the primitive shapes POV-Ray supports, it can also combine multiple simple shapes into complex shapes using *Constructive Solid Geometry* (CSG). There are four basic types of CSG operations: union, intersection, difference, and merge. CSG objects can be composed of primitives or other CSG objects to create more, and more complex shapes.

9.1 Inside and Outside

Most shape primitives, like spheres, boxes and blobs divide the world into two regions. One region is inside the object and one is outside. Given any point in space you can say it's either inside or outside any particular primitive object. CSG uses the concepts of inside and outside to combine shapes together. The inside/outside distinction is not important for a union, but is important for intersection, difference, and merge. Therefore any objects may be combined using union but only solid objects, i.e. objects that have a well-defined interior can be used in the other kinds of CSG. Polygon and triangle cannot be used in csg.

9.2 CSG Operations

9.2.1 Inverse

“Inverse” operation inverts an object so that it'll be inside-out. The appearance of the object is not changed, just the way that POV-Ray perceives it.

When the inverse keyword is used the *inside* of the shape is flipped to become the *outside* and vice versa.

9.2.2 Union

The simplest kind of CSG is the union. The syntax is:

```
UNION:
  union
  {
    OBJECTS...
    [OBJECT_MODIFIERS...]
  }
```

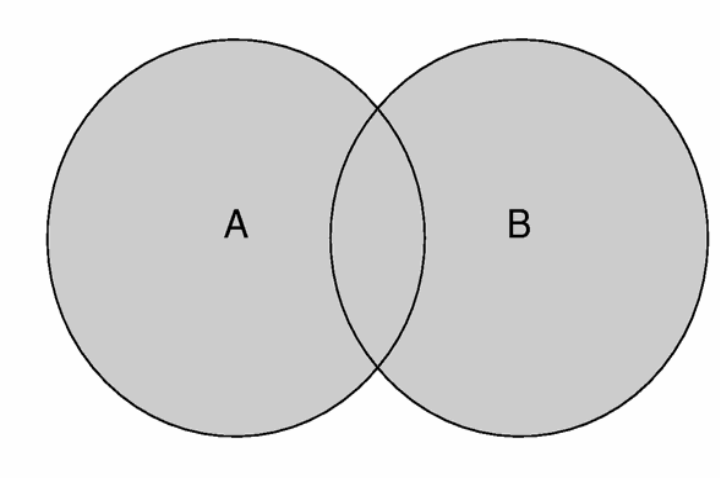


Fig. 9.1 : Union

Unions are simply glue used to bind two or more shapes into a single entity that can be manipulated as a single object. The image above shows the union of A and B. The new object created by the union operation can be scaled, translated and rotated as a single shape. The entire union can share a single

texture but each object contained in the union may also have its own texture, which will override any texture statements in the parent object. The surface inside the union is not removed.

Ray Tracing Union

Ray tracing a union is simply tracing intersection of ray with the two objects separately.

9.2.3 Intersection

The intersection object creates a shape containing only those areas where all components overlap. A point is part of an intersection if it is inside both objects, A and B, as show in the figure below.

The syntax is:

```
INTERSECTION:
   intersection
   {
      SOLID_OBJECTS...
      [OBJECT_MODIFIERS...]
   }
```

The component objects must have well defined inside/outside properties. Patch objects are not allowed. if all components do not overlap, the intersection object disappears.

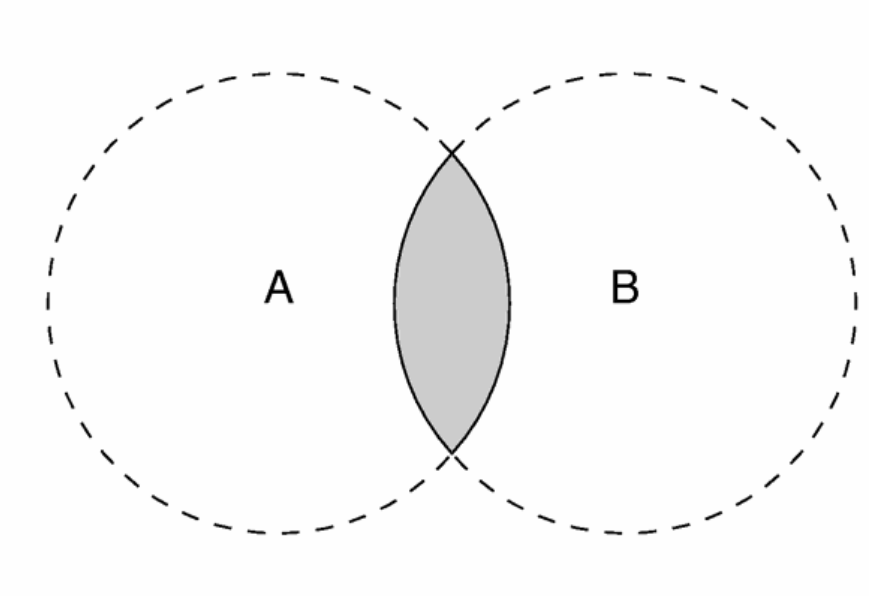


Fig. 9.2 : An Intersection Operation

Ray Tracing Intersection

One of the objects defined in the intersection operation is ray traced first and it is checked if the intersection point lies inside the other object. If it does then the point is displayed otherwise it is discarded.

9.2.4 Difference

The CSG difference operation takes the intersection between the first object and the inverse of all subsequent objects. Thus only points inside object A and outside object B belong to the difference of both objects.

The syntax is:

DIFFERENCE:
  difference

```
{
   SOLID_OBJECTS...
   [OBJECT_MODIFIERS...]
}
```

The component objects must have well defined inside/outside properties. Patch objects are not allowed.

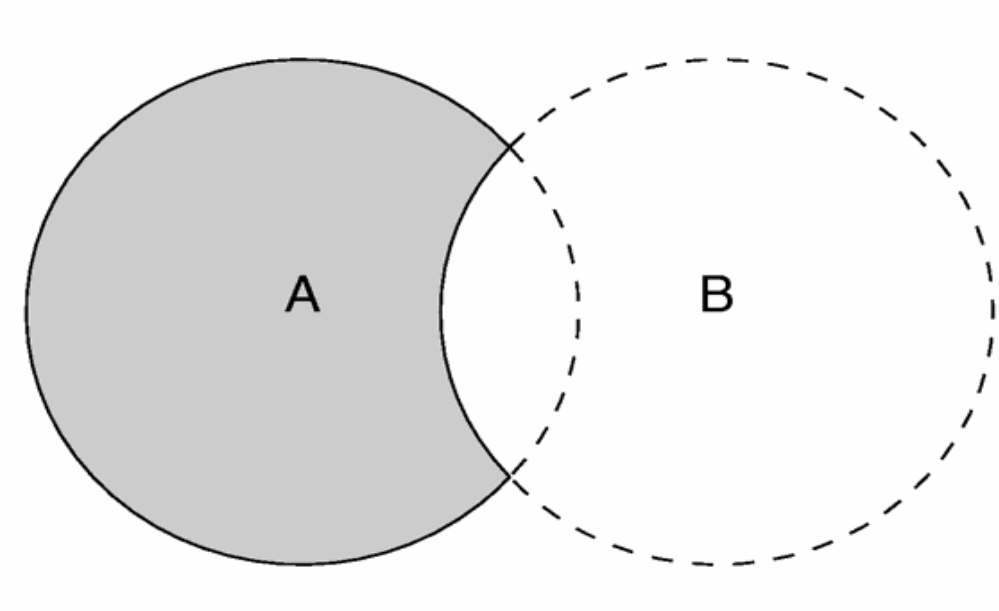


Fig. 9.3 : A Difference Operation

Ray Tracing

Pov-ray inverts the second object defined in a difference operation. When ray tracing intersection with the first object is computed first and then it is checked if it lies outside of the other object. Since, the object was outside of the object is considered inside.

9.2.5 Merge

The union operation just glues objects together; it does not remove the objects' surfaces inside the union. Under most circumstances this doesn't matter. However if a transparent union is used, those interior surfaces will be visible. The merge operations can be used to avoid this problem. It works just like union but it eliminates the inner surfaces like shown in the figure below.

The syntax is:

```
MERGE:
  merge
  {
    SOLID_OBJECTS...
    [OBJECT_MODIFIERS...]
  }
```

Ray Tracing Merge

For merge if the ray intersects a point, it is checked if the point is inside the other object or not. If yes then the point is discarded otherwise not.

10.0 COLORS AND TEXTURES

10.1 Colors

There is no color statement in POV-Ray. Colors are defined by their names or rgb values. POV-ray colors have five components[POV-Ray 3.6.0 Documentation Authors, 3.1.1.5 Specifying Colors*, POV-Ray Help,* POV-Ray(Version 3.6),POV-Ray Pty. Ltd. (2004)].The first three are red, green and blue, the fourth is filter and the fifth is transmit.

The 4th component, called filter, specifies the amount of filtered transparency of a substance. Examples of filtered transparency are stained glass windows or tinted cellophane. The light passing through such objects is tinted by the appropriate color as the material selectively absorbs some frequencies of light while allowing others to pass through. The color of the object is subtracted from the light passing through so this is called subtractive transparency.

The 5th component, called transmit, specifies the amount of non-filtered light that is transmitted through a surface. Some real-world examples of non-filtered transparency are thin see-through cloth, fine mesh netting and dust on a surface. In these examples, all frequencies of light are allowed to pass through tiny holes in the surface. Although the amount of light passing through is diminished, the color of the light passing through is unchanged.

The color of the object and the color transmitted through the object together contribute 100% of the final color. So if transmit is set to 0.9, the transmitted color contributes 90% and the color of the object contributes only 10%. This is also true outside of the 0-1 range, so for example if transmit is set to 1.7, the transmitted color contributes with 170% and the color of the object contributes with minus 70%. Using transmit values outside of the 0-1 range can be used to create interesting special effects, but does not correspond to any phenomena seen in the real world.

POV-Ray Syntax

```
COLOR:
   COLOR_BODY           |
   color COLOR_BODY     | (this means the keyword color or
   colour COLOR_BODY       colour may optionally precede
                        any color specification)
COLOR_BODY:
   COLOR_VECTOR         |
   COLOR_KEYWORD_GROUP  |
   COLOR_IDENTIFIER
COLOR_VECTOR:
   rgb <3_Term_Vector>  |
   rgbf <4_Term_Vector> |
   rgbt <4_Term_Vector> |
   [ rgbft ] <5_Term_Vector>
COLOR_KEYWORD_GROUP:
   [ COLOR_KEYWORD_ITEM ]...
COLOR_KEYWORD_ITEM:
   COLOR_IDENTIFIER     |
   red Red_Amount       |
   blue Blue_Amount     |
   green Green_Amount   |
   filter Filter_Amount |
   transmit Transmit_Amount
```

10.2 Textures

In real life most of the objects have more than one color. Many of them have a complex mix of colors and it would very tedious to explicitly define one color per pixel to model those objects. The simple solution is textures. Textures can add more details to a surface. Mapping an image onto a surface is known as texture mapping[18]. The image is called the texture map and the individual elements are called texels. A simple approach of mapping a texture is by mapping the four corners of the pixel from a surface(s,t) co-ordinate space to textures (u,v) co-ordinate space. After being mapped the shape of the pixel may not be a perfect square if the surface is curved. The four (u,v) points on the texture space defines a quadrilateral. The value for the pixel is computed by summing all the texels lying within the quadrilateral and weighting each by the fraction of the texel that lies within the quadrilateral. For flat surfaces the texture map co-ordinates are directly assigned to its vertices.

This type of mapping can also be called 2D mapping. Although effective in many situations it does not always produce the desirable results. The 2D origin of the textures changes if they are mapped on a curved surface. The texture also looks as if it was painted on the surface, which is an undesirable effect for 3D surfaces.

POV-Ray uses solid textures[19], which does not have the limitations of the 2D textures. POV-Ray textures are actually three-dimensional. The textures in POV-Ray are actually patterns that are defined as functions.

10.2.1 Patterns[20]

Patterns are functions that produce a unique value for every point in the space. Patterns do not wrap around a surface like putting wallpaper on an object. The patterns exist in 3d and the objects are carved from them like carving an object from a solid block of wood or stone. The textured objects would reveal the pattern inside if they were cut through.

Patterns can be generated using a random number system and a noise function. In each case, the x, y, z coordinate of a point on a surface is used to compute some mathematical function that returns a float value. POV-Ray has many predefined patterns and patterns can be created by defining functions. These patterns are used with pigments and normals which will be discussed in the later sections.

Data Structure

The items inside the TPATTERN_FIELDS will be explained later. The ones not explained is not the scope of this report.

```
struct Pattern_Struct

{
  TPATTERN_FIELDS
};


#define TPATTERN_FIELDS       \
 unsigned short Type, Wave_Type, Flags; \
 int References;            \
 SNGL Frequency, Phase;     \
 SNGL Exponent;             \
 WARP *Warps;               \
```

```
TPATTERN *Next;            \
BLEND_MAP *Blend_Map;      \
union {                    \
  DENSITY_FILE *Density_File; \
  IMAGE *Image;            \
  VECTOR Gradient;         \
  SNGL Agate_Turb_Scale;   \
  short Num_of_Waves;      \
  short Iterations;        \
  short Arms;              \
  struct { SNGL Mortar; VECTOR Size; } Brick; \
  struct { SNGL Control0, Control1; } Quilted;  \
  struct { DBL Size, UseCoords; VECTOR Metric; } Facets; \
  struct { VECTOR Form; VECTOR Metric; DBL Offset; DBL Dim; \
          short IsSolid; VECTOR *cv; int lastseed; \
          VECTOR lastcenter; }  Crackle;  \
  struct { VECTOR Slope_Vector, Altit_Vector; \
          short Slope_Base, Altit_Base; DBL Slope_Len, \
          Altit_Len; UV_VECT Slope_Mod, Altit_Mod; } Slope; \
  struct { UV_VECT Coord; short Iterations, interior_type, \
          exterior_type; DBL efactor, ifactor; \
          int Exponent; } Fractal; \
  struct { void *Fn; void *Data; } Function;\
  PIGMENT *Pigment; \
  OBJECT *Object;\
} Vals;
```

10.2.1.1 Pattern types

10.2.1.1.1 Block patterns

The block patterns are not actually functions. They are list of colors with distinct edges that are patterned in blocks. There are four block patterns checker, hexagon, brick and object. They can also be called color lists.

POV-Ray Syntax

```
COLOR_LIST_PIGMENT:
  pigment {brick [COLOR_1, [COLOR_2]] [PIGMENT_MODIFIERS...] }|
  pigment {checker [COLOR_1, [COLOR_2]] [PIGMENT_MODIFIERS...]}|
  pigment {
    hexagon [COLOR_1, [COLOR_2, [COLOR_3]]] [PIGMENT_MODIFIERS...]
  }|
  pigment {object OBJECT_IDENTIFIER | OBJECT {} [COLOR_1, COLOR_2]}
```

Checker is an example of a block pattern.

Checker

The checker pattern produces a checkered pattern consisting of alternating squares of two colors. The syntax is:

```
  pigment { checker [COLOR_1 [, COLOR_2]] [PATTERN_MODIFIERS...] }
```

If no colors are specified then default blue and green colors are used.

The checker pattern is actually a series of cubes that are one unit in size. The cubes are arranged in an alternating check pattern and stacking them in

layer after layer so that the colors still alternate in every direction. Eventually a larger cube is formed. The pattern of checks on each side is what the POV-Ray checker pattern produces when applied to a box object. Cutting away at the cube until it is carved into a smooth sphere or any other shape will reveal what it looks like when applied to that object. Since colors of the checker pattern are one unit cube, it is not mapped very well on round surfaces like sphere, cylinder and torus.

10.2.1.1.2 Blending Functions

Blending function patterns produces a series of values from 0 to 1 provided the co-ordinate values of a point.

There are many built in patterns defined by POV-Ray. An example of a blending function patterns is cylindrical.

Cylindrical Pattern

The cylindrical pattern creates a one unit radius cylinder along the Y axis. It is computed by: *value = 1.0-min(1, sqrt(X^2 + Z^2))* It starts at 1.0 at the origin and decreases to a minimum value of 0.0 as it approaches a distance of 1 unit from the Y axis. It remains at 0.0 for all areas beyond that distance.

10.2.2 POV-Ray Texture

The textures of POV-Ray can be of three basic kinds of textures: plain, patterned, and layered. A plain texture consists of a single pigment, an optional normal, and a single finish. A patterned texture combines two or more textures using a block pattern or blending function pattern. Patterned textures may be made quite complex by nesting patterns within patterns. At the innermost levels however, they are made up from plain textures. A layered texture consists of two or more semi-transparent textures layered on top of one another.

POV-Ray Syntax

```
TEXTURE:
   PLAIN_TEXTURE | PATTERNED_TEXTURE | LAYERED_TEXTURE
PLAIN_TEXTURE:
   texture
   {
      [TEXTURE_IDENTIFIER]
      [PNF_IDENTIFIER...]
      [PNF_ITEMS...]
   }
PNF_IDENTIFIER:
   PIGMENT_IDENTIFIER | NORMAL_IDENTIFIER | FINISH_IDENTIFIER
PNF_ITEMS:
   PIGMENT | NORMAL | FINISH | TRANSFORMATION
LAYERED_TEXTURE:
   NON_PATTERNED_TEXTURE...
PATTERNED_TEXTURE:
   texture
   {
[PATTERNED_TEXTURE_ID]
```

```
 [TRANSFORMATIONS...]
   } |
   texture
   {
      PATTERN_TYPE
      [TEXTURE_PATTERN_MODIFIERS...]
   } |
   texture
   {
      tiles TEXTURE tile2 TEXTURE
      [TRANSFORMATIONS...]
   } |
   texture
   {
      material_map
      {
         BITMAP_TYPE "bitmap.ext"
         [MATERIAL_MODS...] TEXTURE... [TRANSFORMATIONS...]
      }
   }
TEXTURE_PATTERN_MODIFIER:
   PATTERN_MODIFIER | TEXTURE_LIST |
   texture_map { TEXTURE_MAP_BODY }
```

Data Structure:

The texture data structure holds the values that describe the material properties of an object.

```
struct Texture_Struct
{
  TEXTURE_FIELDS
  PIGMENT *Pigment;
  TNORMAL *Tnormal;
  FINISH *Finish;
  TEXTURE *Materials;
  int Num_Of_Mats;

};
```

10.2.2.1 Pigments[21]

The color or pattern of colors for an object is defined by a pigment statement. All plain textures must have a pigment. A default pigment is used if one is not specified. The color you define is the way you want the object to look if fully illuminated. POV-Ray brightens or darkens it depending on the lighting in the scene. The parameter is called pigment because it defines the basic color the object actually is rather than how it looks.

The syntax for pigment is:

```
PIGMENT:
   pigment {
      [PIGMENT_IDENTIFIER]
      [PIGMENT_TYPE]
      [PIGMENT_MODIFIER...]
   }
PIGMENT_TYPE:
   PATTERN_TYPE | COLOR |
```

```
   image_map {
      BITMAP_TYPE "bitmap.ext" [IMAGE_MAP_MODS...]
   }
PIGMENT_MODIFIER:
   PATTERN_MODIFIER | COLOR_LIST | PIGMENT_LIST |
   color_map { COLOR_MAP_BODY } | colour_map { COLOR_MAP_BODY } |
   pigment_map { PIGMENT_MAP_BODY } | quick_color COLOR |
   quick_colour COLOR
```

Data Structure:

```
struct Pigment_Struct
{
  TPATTERN_FIELDS
  COLOUR Colour;
};
```

The default value of the color is black if not specified. The TPATTERN_FIELDS is included in the structure because a pigment may contain a patterns type rather than a color.

10.2.2.1.1 Patterns as Pigments

Patterns can be used in pigments with or without a color map. A color map is a list of colors that are defined to be applied for different values returned by the pattern. If a color map is not present with the patterns the default color values are used. The block patterns cannot have a color map since they do not return series of values from 0 to 1. Patterns can also be used with pigment maps, which define different pigments to be used with different values generated by the pattern function.

POV-Ray color map syntax

COLOR_MAP:
color_map { COLOR_MAP_BODY } | colour_map { COLOR_MAP_BODY }
COLOR_MAP_BODY:
COLOR_MAP_IDENTIFIER | COLOR_MAP_ENTRY...
COLOR_MAP_ENTRY:
[Value COLOR] |
[Value_1, Value_2 color COLOR_1 color COLOR_2]

POV-Ray pigment map syntax

PIGMENT_MAP:
pigment_map { PIGMENT_MAP_BODY }
PIGMENT_MAP_BODY:
PIGMENT_MAP_IDENTIFIER | PIGMENT_MAP_ENTRY...
PIGMENT_MAP_ENTRY:
[Value PIGMENT_BODY]

These maps are called blend maps. If the value returned by the function is lower than the first entry in the blend map, then the color specified with the first entry is applied. If greater than the last entry in the blend map, then the color associated with that value is applied. Otherwise if the returned value is in between two defined values then linear interpolation is used between the colors.

10.2.2.1.2 Image maps

An image map can be used to wrap a 2-D bit-mapped image around your 3-D objects. Image maps can define transmit and transparency for different color values contained in it.

The syntax for an image_map is:

IMAGE_MAP:
pigment

```
  {
    image_map
    {
      [BITMAP_TYPE] "bitmap[.ext]"
      [IMAGE_MAP_MODS...]
    }
    [PIGMENT_MODFIERS...]
  }
BITMAP_TYPE:
  gif | tga | iff | ppm | pgm | png | jpeg | tiff | sys
IMAGE_MAP_MOD:
  map_type Type | once | interpolate Type |
  filter Palette, Amount | filter all Amount |
  transmit Palette, Amount | transmit all Amount
```

10.2.2.2 Normals[22]

Ray-tracing is known for the dramatic way it depicts reflection, refraction and lighting effects. Much of our perception depends on the reflective properties of an object. Ray tracing can exploit this by playing tricks on our perception to make us see complex details that aren't really there.

It would be very difficult to mathematically model bumps on an object. However, altering the way light reflects off of the surface bumps can be simulated. Reflection calculations depend on a vector called a *surface normal* vector. This is a vector which points away from the surface and is perpendicular to it. By artificially modifying (or perturbing) this normal vector you can simulate bumps. This is done by adding an optional normal statement. Attaching a normal pattern does not really modify the surface. It only affects the way light reflects or refracts at the surface so that it looks bumpy.

The syntax is:

```
NORMAL:
  normal { [NORMAL_IDENTIFIER] [NORMAL_TYPE] [NORMAL_MODIFIER...] }
NORMAL_TYPE:
  PATTERN_TYPE Amount |
  bump_map { BITMAP_TYPE "bitmap.ext" [BUMP_MAP_MODS...]}
NORMAL_MODIFIER:
  PATTERN_MODIFIER | NORMAL_LIST | normal_map { NORMAL_MAP_BODY } |
  slope_map{ SLOPE_MAP_BODY } | bump_size Amount |
  no_bump_scale Bool | accuracy Float
```

Patterns as normals

Patterns can be used as normals as well. A pattern can be defined with a slope map or a normal map.

10.2.2.2.1 Slope Map

A slope_map is a normal pattern modifier which gives the user a great deal of control over the exact shape of the bumpy features. Each of the various pattern types available is in fact a mathematical function that takes any x, y, z location and turns it into a number between 0.0 and 1.0 inclusive. That number is used to specify where the various high and low spots are. The slope_map lets further shape the contours.

The syntax is as follows...

```
SLOPE_MAP:
   slope_map { SLOPE_MAP_BODY }
SLOPE_MAP_BODY:
   SLOPE_MAP_IDENTIFIER | SLOPE_MAP_ENTRY...
SLOPE_MAP_ENTRY:
   [ Value, <Height, Slope> ]
```

The [] brackets are part of the actual *SLOPE_MAP_ENTRY*. They are not notational symbols denoting optional parts. The brackets surround each entry in the slope map.

There may be from 2 to 256 entries in the map.

Each *Value* is a float value between 0.0 and 1.0 inclusive and each <*Height*, *Slope*> is a 2 component vector such as <0,1> where the first value represents the apparent height of the wave and the second value represents the slope of the wave at that point. The height should range between 0.0 and 1.0 but any value could be used.

The slope value is the change in height per unit of distance. For example a slope of zero means flat, a slope of 1.0 means slope upwards at a 45 degree angle and a slope of -1 means slope down at 45 degrees. Theoretically a slope straight up would have infinite slope. In practice, slope values should be kept in the range -3.0 to +3.0. Keep in mind that this is only the visually apparent slope. A normal does not actually change the surface.

10.2.2.2.2 Nomral map

Normal maps lets the user define different normals for different parts of the objects surface.

The syntax for normal_map is as follows:

```
NORMAL_MAP:
   normal_map { NORMAL_MAP_BODY }
NORMAL_MAP_BODY:
   NORMAL_MAP_IDENTIFIER | NORMAL_MAP_ENTRY...
NORMAL_MAP_ENTRY:
   [ Value NORMAL_BODY ]
```

Where *Value* is a float value between 0.0 and 1.0 inclusive and each *NORMAL_BODY* is anything which can be inside a normal{...} statement. The normal keyword and {} braces need not be specified.

There may be from 2 to 256 entries in the map. The normals can be nested to any levels of complexity.

Surface normals that use patterns that were not designed for use with normals (anything other than bumps, dents, waves, ripples, and wrinkles) uses a slope_map. To create a perturbed normal from a pattern, POV-Ray samples the pattern at four points in a pyramid surrounding the desired point to determine the gradient of the pattern at the center of the pyramid. The distance that these points are from the center point determines the accuracy of the approximation. Using points too close together causes floating-point inaccuracies. However, using points too far apart can lead to artefacts as well as smoothing out features that should not be smooth. These effects can be controlled by delta or accuracy which is not the scope of this report.

10.2.2.2.3 Bump Map

A bump_map can be used to wrap a 2-D bit-mapped bump pattern around your 3-D objects.

Instead of placing the color of the image on the shape like an image_map a bump_map perturbs the surface normal based on the color of the image at that point. The result looks like the image has been embossed into the surface. By default, a bump map uses the brightness of the actual color of the pixel. Colors are converted to gray scale internally before calculating height. Black is a low spot, white is a high spot.

Specifying a Bump Map

The syntax for a bump_map is:

```
BUMP_MAP:
   normal
   {
      bump_map
      {
         BITMAP_TYPE "bitmap.ext"
         [BUMP_MAP_MODS...]
      }
      [NORMAL_MODFIERS...]
   }
BITMAP_TYPE:
   gif | tga | iff | ppm | pgm | png | jpeg | tiff | sys
BUMP_MAP_MOD:
   map_type Type | once | interpolate Type | use_color |
   use_colour | bump_size Value
```

10.2.2.3 Patterns as textures

Patterned textures are complex textures made up of multiple textures. The component textures may be plain textures or may be made up of patterned textures. A plain texture has just one pigment, normal and finish statement. Even a pigment with a pigment map is still one pigment and thus considered a plain texture as are normals with normal map statements.

Patterned textures use either a texture_map statement to specify a blend or pattern of textures or they use block textures such as checker with a texture list or

a bitmap similar to an image map called a *material map* specified with a material_map statement.

The syntax is...

```
PATTERNED_TEXTURE:
  texture
  {
    [PATTERNED_TEXTURE_ID]
    [TRANSFORMATIONS...]
  } |
  texture
  {
    PATTERN_TYPE
    [TEXTURE_PATTERN_MODIFIERS...]
  } |
  texture
  {
    tiles TEXTURE tile2 TEXTURE
    [TRANSFORMATIONS...]
  } |
  texture
  {
    material_map
    {
      BITMAP_TYPE "bitmap.ext"
      [BITMAP_MODS...] TEXTURE... [TRANSFORMATIONS...]
    }
  }
TEXTURE_PATTERN_MODIFIER:
  PATTERN_MODIFIER | TEXTURE_LIST |
  texture_map { TEXTURE_MAP_BODY }
```

The syntax for texture_map is as follows:

```
TEXTURE_MAP:
   texture_map { TEXTURE_MAP_BODY }
TEXTURE_MAP_BODY:
  TEXTURE_MAP_IDENTIFIER | TEXTURE_MAP_ENTRY...
TEXTURE_MAP_ENTRY:
   [ Value TEXTURE_BODY ]
```

Texture maps are similar to color and pigment map except there is texture defined instead of colors or pigments.

Layered Textures[23]

It is possible to create a variety of special effects using layered textures. A layered texture consists of several textures that are partially transparent and are laid one on top of the other to create a more complex texture. The different texture layers show through the transparent portions to create the appearance of one texture that is a combination of several textures.

A layered textures can be created by listing two or more textures one right after the other. The last texture listed will be the top layer, the first one listed will be the bottom layer. All textures in a layered texture other than the bottom layer should have some transparency. For example:

```
 object {
   My_Object
   texture {T1}  // the bottom layer
   texture {T2}  // a semi-transparent layer
   texture {T3}  // the top semi-transparent layer
 }
```

In this example T2 shows only where T3 is transparent and T1 shows only where T2 and T3 are transparent.

The color of underlying layers is filtered by upper layers but the results do not look exactly like a series of transparent surfaces. If you had a stack of surfaces with the textures applied to each, the light would be filtered twice: once on the way in as the lower layers are illuminated by filtered light and once on the way out. Layered textures do not filter the illumination on the way in. Other parts of the lighting calculations work differently as well. The results look great and allow for fantastic looking textures but they are simply different from multiple surfaces.

11.0 LIGHT SOURCES

In any ray-traced scene, the light needed to illuminate the objects and their surfaces must come from a light source. There are many kinds of light sources[24] available in POV-Ray and careful use of the correct kind can give very impressive results. Pov-ray light sources are not objects. They do not have any visible shape. They are only parsed as an object to allow union between an object and a light source.

Light Source Data Structure

The items specified on the data structure will be explained through the chapter. The ones not explained are not in the scope of this document.

```
struct Light_Source_Struct
{
  COMPOUND_FIELDS
  COLOUR Colour;
  VECTOR Direction, Center, Points_At, Axis1, Axis2;
  DBL Coeff, Radius, Falloff;
  DBL Fade_Distance, Fade_Power;
  LIGHT_SOURCE *Next_Light_Source;
  unsigned char Light_Type, Area_Light, Jitter;
  bool Orient;
  bool Circular;
  unsigned char Track, Parallel;
  unsigned char Photon_Area_Light;  /* these bytes could be compressed
```

```
                                to a single flag byte */
  int Area_Size1, Area_Size2;
  int Adaptive_Level;
  int Media_Attenuation;
  int Media_Interaction;
  COLOUR **Light_Grid;
  OBJECT *Shadow_Cached_Object,*Projected_Through_Object;
  BLEND_MAP *blend_map;                              /* for dispersion */
  PROJECT_TREE_NODE      *Light_Buffer[6];     /*     Light     buffers     for     the     six
                                                  general directions in space*/

};
```

11.1 Point light sources

Lights from point light sources come from a single point. All types of point light sources cast hard shadows.

11.1.1 The Default Pointlight

The simplest kind of light is a point light. A point light source sends light of the specified color uniformly in all directions. The default light type is a point source. The *<Location>* and *COLOR* is all that is required to define a point light source.

POV-Ray Syntax

```
light_source
  {
    <Location>, COLOR
  }
```

11.1.2 Parallel Lights

Syntax:

```
light_source {
  LOCATION_VECTOR, COLOR
  [LIGHT_SOURCE_ITEMS...]
  parallel
  point_at VECTOR
}
```

The parallel keyword can be used with any type of light source.

Parallel lights are useful for simulating very distant light sources, such as sunlight. As the name suggests, it makes the light rays parallel.

Technically this is done by shooting rays from the closest point on a plane to the object intersection point. The plane is determined by a perpendicular defined by the light location and the point_at vector.

Parallel ray

The figure below shows a parallel light source. The center of the light source is c and the light points at p. The light direction vector is **L**. An intersection point on the plane is i.

To find the parallel ray first the original shadow ray is computed which is,

O=c-i

Both **L** and **O** are normalized.

The angle between them is ɵ and,

Dot**(L,O)=**cos ɵ

The projection of **O** on **L** is

k= length(**O)***cos ɵ

The depth of the light source is k, which is equal to the length of the **S,** the parallel shadow ray.

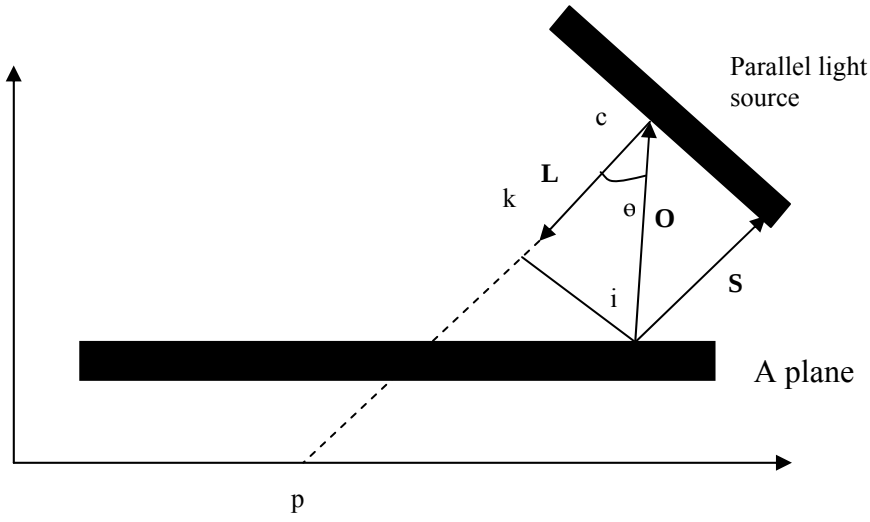


Fig. 11.1 : Parallel light source

11.1.3 Spotlights

Spotlight are a just a type of point light whose light is restricted by a cone. Normally light radiates outward equally in all directions from the source. However the spotlight keyword can be used to create a cone of light that is bright in the center and falls of to darkness in a soft fringe effect at the edge.

POV-Ray Syntax:

```
SPOTLIGHT_SOURCE:
   light_source
   {
      <Location>, COLOR spotlight
      [LIGHT_MODIFIERS...]
   }
LIGHT_MODIFIER:
   SPOTLIGHT_ITEM | AREA_LIGHT_ITEMS | GENERAL_LIGHT_MODIFIERS
SPOTLIGHT_ITEM:
   radius Radius | falloff Falloff | tightness Tightness |
   point_at <Spot>
```

Default values:

```
 radius:    30 degrees
 falloff:   45 degrees
 tightness:  0
```

The point_at keyword tells the spotlight to point at a particular 3D coordinate. A line from the location of the spotlight to the point_at coordinate forms the center line of the cone of light.

The falloff, radius, and tightness keywords control the way that light tapers off at the edges of the cone. These four keywords apply only when the spotlight or cylinder keywords are used.

The falloff keyword specifies the overall size of the cone of light. This is the point where the light falls off to zero intensity. The float value you specify is the angle, in degrees, between the edge of the cone and centerline. The radius keyword specifies the size of the "hot-spot" at the center of the cone of light. The "hot-spot" is a brighter cone of light inside the spotlight cone and has the same

centerline. The radius value specifies the angle, in degrees, between the edge of this bright, inner cone and the centerline. The light inside the inner cone is of uniform intensity. The light between the inner and outer cones tapers off to zero.

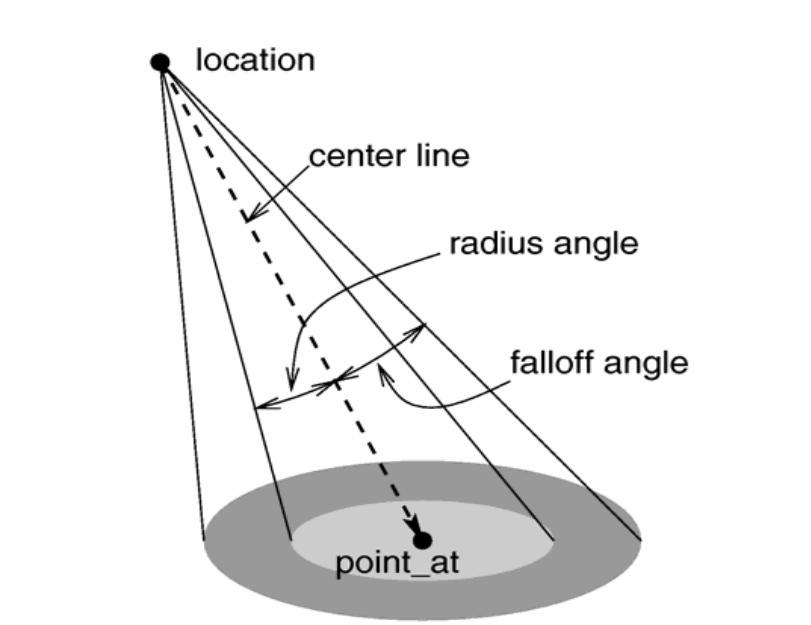


Fig.11.2 : A Spotlight

11.1.3.1 Spot light attenuation

The attenuation factor is calculated differently for spotlights. It limits the spotlights light in a cone shape and controls the brightness.

Formula is : Attenuation= $k\cos^t\theta$

Where θ is the angle between the ray direction and light direction, t is the value of the tightness and k depends on θ.

To compute k a cubic curve through cos(falloff) and cos(radius) is imagined and the position of cosθ is used to calculate k.

If cosθ<cos(falloff)
k=0

If cosθ>cos(radius)
k=1

If cos(falloff)<cosθ<cos(radius)

p= (cosθ - cos(falloff)) / (cos(radius)- cos(falloff))
k=(3 - 2 * p) * p * p

11.1.4 Cylindrical Lights

The cylinder keyword specifies a cylindrical light source that is great for simulating laser beams. Cylindrical light sources work pretty much like spotlights except that the light rays are constrained by a cylinder and not a cone.

POV-Ray Syntax

```
CYLINDER_LIGHT_SOURCE:
   light_source
   {
      <Location>, COLOR cylinder
      [LIGHT_MODIFIERS...]
   }
LIGHT_MODIFIER:
```

SPOTLIGHT_ITEM | AREA_LIGHT_ITEMS | GENERAL_LIGHT_MODIFIERS

SPOTLIGHT_ITEM:

radius Radius | falloff Falloff | tightness Tightness |

point_at <Spot>

Default values:

radius: 0.75 degrees

falloff: 1 degrees

tightness: 0

For the cylindrical light sources the radius is the value of the cylinders inner radius inside which lights full intensity is found, falloff is the outer radius of the cylinder. so, the cylindrical light source can be thought of as a thick cylinder where light intensity slowly tappers off from in between the inner and outer radius of the cylinder.

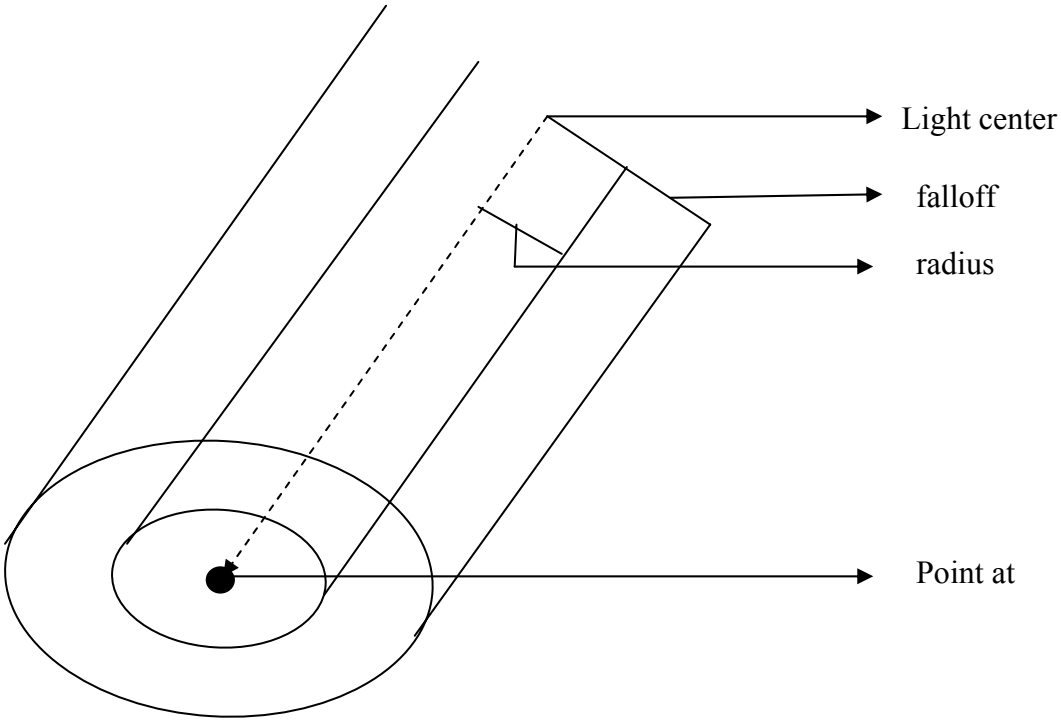


Fig. 11.3 : A cylindrical light source

11.1.4.1 Shadow Ray computation for cylindrical source

The cylinderical light source is treated as a parallel light source although it is denied in the POV-Ray documentation. The rays are only constrained by the shape of a cylinder.

The same idea used for producing the shadow ray for parallel lights.

11.1.4.2 Cylenderical light attenuation

The rays from the cylinder must be constrained by a cylindrical shape. The attenuation for cylindrical light source finds if the point is inside the cylindrical light source shape and assigns the appropriate intensity.

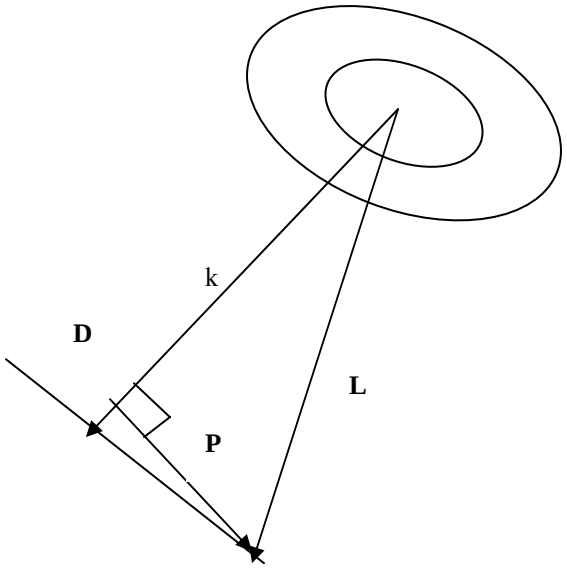


Fig. 11.4 : Cylindrical light source attenuation

In the figure **D** is the lights direction vector **L** is a vector through lights center and intersection point. K is the projection of **L** onto **D.**

A positive value for k means that the intersection point is on the right side of the light.

Now,

P=L–k***D**

l=length(P)

Len gives the distance of the intersection point from the center of the cylinder.

If l< Falloff)

d = 1.0 - l / Falloff;

Attenuation = d^t

Where t is the tightness.

If Radius > 0.0 and l > Radius

low=0

high=1.0 - Light->Radius / Light->Falloff

p= (dist - low) / (high- low)

k= (3 - 2 * p) * p * p

Attenuation *=k

11.1.5 Shadows

Ray tracing is adept at producing shadows. To compute if a point is in shadow or not a light ray is shot from the intersection point to each of the point light sources. If the ray intersects any opaque object along the way then the object casts shadow on the point if the point is not an opaque object then the shadow ray is filtered by the color of transparent object and the color contribution is added to that points color. The object casting shadow on the point is saved and for the next pixel intersection is first performed with the cached object first. This is done because an object is likely to cast shadow on an extended area of an object. If intersection with the cached object is not found then intersection is performed with the other objects.

11.2 Area Lights and soft shadows

Area lights are also known as extended light sources[25]. Area light sources occupy a finite, one- or two-dimensional area of space. They can cast soft shadows because an object can partially block their light. Point sources are either totally blocked or not blocked.

The area_light keyword in POV-Ray creates sources that are rectangular in shape, sort of like a flat panel light. Rather than performing the complex calculations that would be required to model a true area light, it is approximated as an array of point light sources spread out over the area occupied by the light. The array-effect applies to shadows only. The object's illumination is still that of a point source. The intensity of each individual point light in the array is dimmed so that the total amount

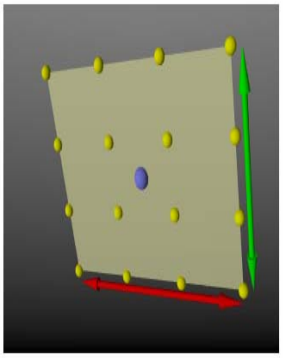


Fig. 11.5 : Area lights

of light emitted by the light is equal to the light color specified in the declaration.

POV-Ray Syntax

AREA_LIGHT_SOURCE:

```
  light_source {
  LOCATION_VECTOR, COLOR
  area_light
  AXIS_1_VECTOR, AXIS_2_VECTOR, Size_1, Size_2
  [adaptive Adaptive] [ jitter ]
  [ circular ] [ orient ]
  [ [LIGHT_MODIFIERS...]
 }
```

Any type of light source may be an area light.

The area_light command defines the location, the size and orientation of the area light as well as the number of lights in the light source array. The location vector is the centre of a rectangle defined by the two vectors *<Axis_1>* and *<Axis_2>*. These specify the lengths and directions of the edges of the light.

Since the area lights are rectangular in shape these vectors should be perpendicular to each other. The larger the size of the light the thicker the soft part of shadows will be. The integers Size_1 and Size_2 specify the number of rows and columns of point sources of the. The more lights you use the smoother your shadows will be but the longer they will take to render.

It is possible to specify spotlight parameters along with the area light parameters to create area spotlights. Using area spotlights is a good way to speed up scenes that use area lights since you can confine the lengthy soft shadow calculations to only the parts of your scene that need them.

The jitter command is optional. When used it causes the positions of the point lights in the array to be randomly jittered to eliminate any shadow banding that may occur. The jittering is completely random from render to render and should not be used when generating animations.

The adaptive command is used to enable adaptive sampling of the light source. By default POV-Ray calculates the amount of light that reaches a surface from an area light by shooting a test ray at every point light within the array. This is very slow. Adaptive sampling on the other hand attempts to approximate the same calculation by using a minimum number of test rays. The number specified after the keyword controls how much adaptive sampling is used. The higher the number the more accurate your shadows will be but the longer they will take to render. If you're not sure what value to use a good starting point is adaptive 1. The adaptive keyword only accepts integer values and cannot be set lower than 0.

When performing adaptive sampling POV-Ray starts by shooting a test ray at each of the four corners of the area light. If the amount of light received from all four corners is approximately the same then the area light is assumed to be either fully in view or fully blocked. The light intensity is then calculated as the average intensity of the light received from the four corners. However, if the light intensity from the four corners differs significantly then the area light is partially blocked. The area light is split into four quarters and each section is sampled as described above. This allows POV-Ray to rapidly approximate how much of the area light is in view without having to shoot a test ray at every light in the array. Visually the sampling goes like shown below.

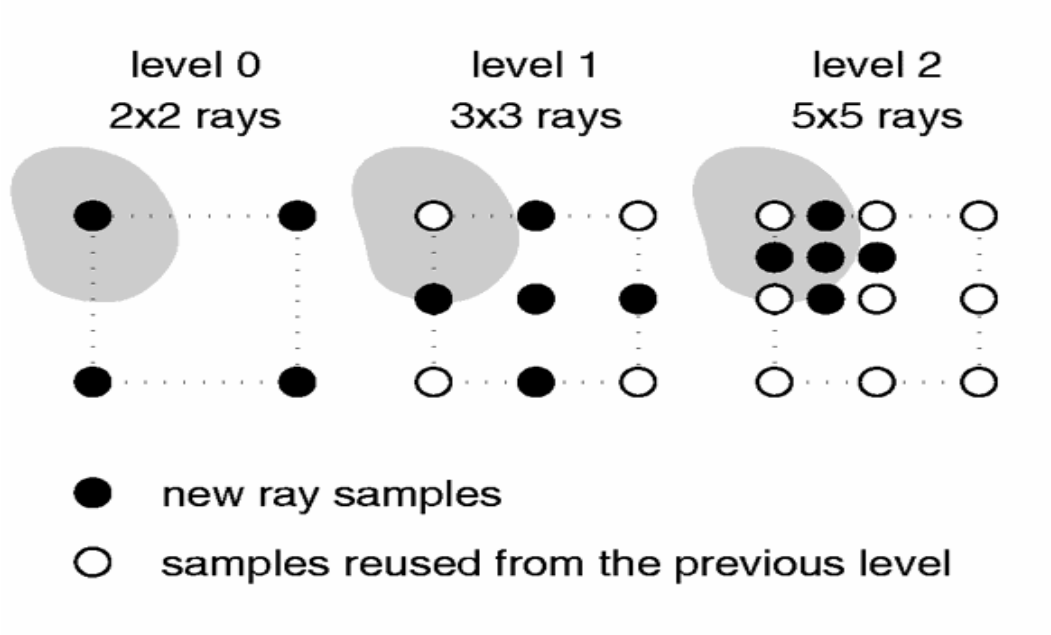


Fig. 11.6: Adaptive area light sampling

While the adaptive sampling method is faster relatively it can sometimes produce inaccurate shadows. The solution is to reduce the amount of adaptive sampling without completely turning it off. The number after the adaptive keyword adjusts the number of times that the area light will be split before the adaptive phase begins. For example for adaptive 0 a minimum of 4 rays will be shot at the light. For adaptive 1 a minimum of 9 rays will be shot (adaptive 2 gives 25 rays, adaptive 3 gives 81 rays, etc). Obviously the more shadow rays shoot the slower the rendering will be.

The number of rays never exceeds the values specified for rows and columns of points. For example area_light x,y,4,4 specifies a 4 by 4 array of lights. If adaptive 3 is specified, it would mean that it needs at least a 9 by 9 array. In this case no adaptive sampling is done. The 4 by 4 array is used.

The circular command has been added to area lights in order to better create circular soft shadows. With ordinary area lights the pseudo-lights are arranged in a rectangular grid and thus project partly rectangular shadows around all objects, including circular objects.

By including the circular tag in an area light, the light is stretched and squashed so that it looks like a circle: this way, circular or spherical light sources are better simulated.

Circular area lights can be ellipses: the AXIS_1_VECTOR and AXIS_2_VECTOR define the shape and orientation of the circle; if the vectors are not equal, the light source is elliptical in shape.

By including the orient modifier in an area light, the light is rotated so that for every shadow test, it always faces the point being tested. The initial orientation is no longer important, so you only have to consider the desired dimensions (area) of the light source when specifying the axis vectors. In effect, this makes the area light source appear 3-dimensional (e.g. an area_light with perpendicular axis vectors of the same size and dimensions using circular *and* orient simulates a spherical light source). Orient can be used with "circular" lights only, the two axes of the area light must be of equal length and the two axes of the area light should use an equal number of samples, and that number should be greater than one

.

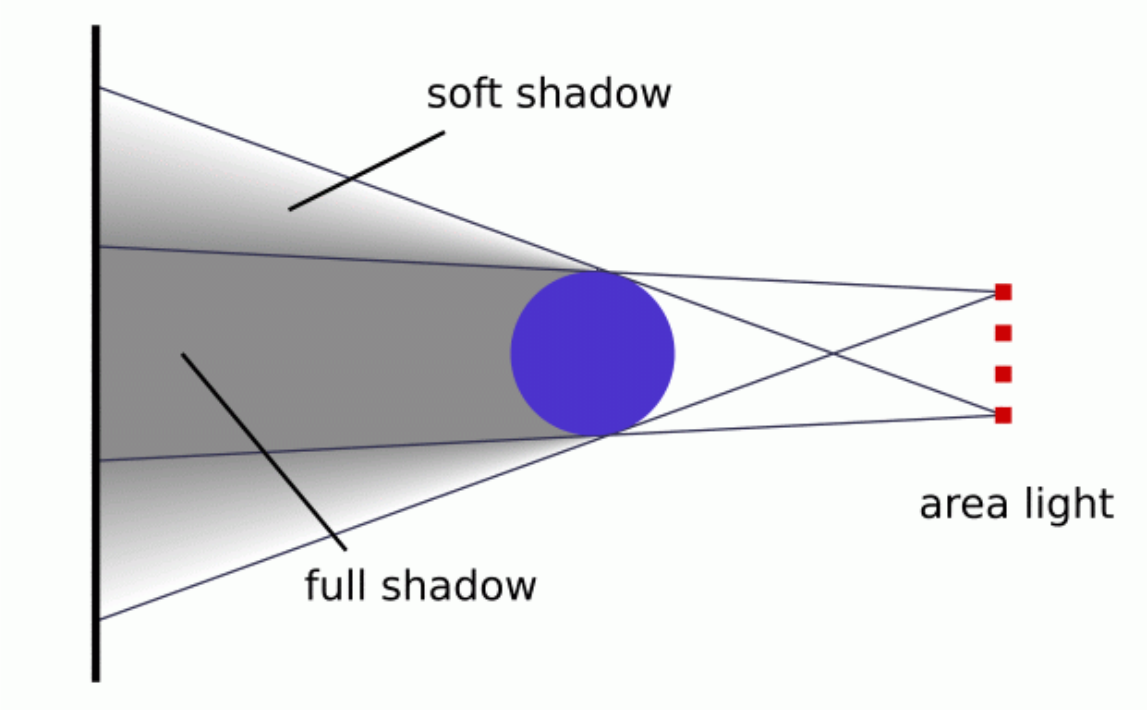


Fig. 11.7 : Soft Shadow

11.3 Attenuation

In real life light fades at distance. To simulate this POV-Ray uses fade_distance and fade_power that defines attenuation by,

$$attenuation = \frac{2}{1 + \left(\frac{d}{fadeXdistance}\right)^{fadeXpower}}$$

Where d is the distance traveled by the light.

12.0 POV-RAY GLOBAL ILLUMINATION MODEL

Pov-ray is a recursive raytracer[26,27]. It traces rays backwards from out into the scene. Rays start their journey from the pinhole camera. The user specifies the location of the camera, light sources, and objects as well as the surface texture properties of objects, their interiors (if transparent) and any atmospheric media such as fog, haze, or fire. A ray is shot through each of the pixels through the viewing window. If the ray intersects any objects then the color of the surface at that point is calculated. Otherwise atmospheric calculations are performed on that point. Each time a ray hits an object, a ray is sent to each of the lights on the scene to find the contribution of lights on that point. For reflective, transparent or refractive objects new rays are spawned and the reflective and refractive rays contribute to determine the color of that point. The rays thus form a ray tree.

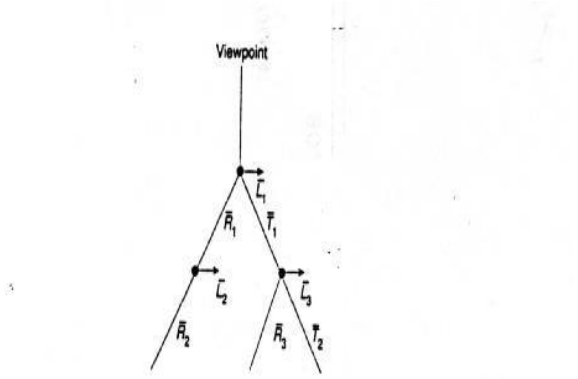


Fig. 12.1: The Ray Tree

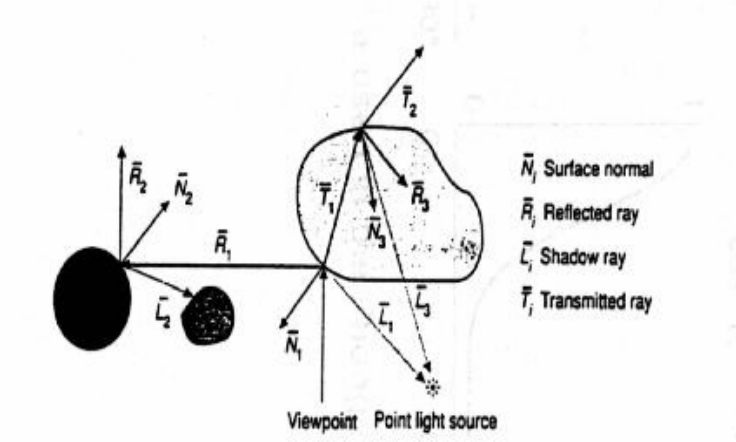


Fig. 12.2 : Rays recursively spawn

The ray tree is formulated in a bottom up manner and each node's intensity is computed as a function of its children's intensity.

The recursive ray tracing process is very slow and it can also get bogged down to tracing multiple reflection and refraction rays. To prevent this from happening the maximum depth of the ray tree can be specified. This parameter is called max_trace_level. A weight is assigned to the rays each time a ray starts tracing. The weight of the rays keeps dropping down along the tree. The parameter adc_bailout, stops a ray from being traced if its weight value is lower than the value specified by adc_bailout, which means the contribution of the ray for that pixel is very low.

For every pixel in the final image one or more viewing rays are shot from the camera, into the scene to see if it intersects with any of the objects in the scene. These "viewing rays" originate from the viewer, represented by the camera, and pass through the viewing window (representing the final image).
Every time an object is hit, the color of the surface at that point is calculated. For this purpose rays are sent backwards to each light source to determine the amount of light coming from the source. These "shadow rays" are tested to tell whether the surface point lies in shadow or not. If the surface is reflective or

transparent new rays are set up and traced in order to determine the contribution of the reflected and refracted light to the final surface color.
Special features like inter-diffuse reflection (radiosity), atmospheric effects and area lights make it necessary to shoot a lot of additional rays into the scene for every pixel.

12.1 Finish[28]

The finish properties of a surface can greatly affect its appearance. How does light reflect? What happens in shadows? What kind of highlights are visible. To answer these questions you need a finish.

The syntax for finish is as follows:

```
FINISH:
   finish { [FINISH_IDENTIFIER] [FINISH_ITEMS...] }
FINISH_ITEMS:
   ambient COLOR | diffuse Amount | brilliance Amount |
   phong Amount | phong_size Amount | specular Amount |
   roughness Amount | metallic [Amount] | reflection COLOR |
   crand Amount | conserve_energy BOOL_ON_OF |
   reflection { Color_Reflecting_Min [REFLECTION_ITEMS...] }|
   irid { Irid_Amount [IRID_ITEMS...] }
REFLECTION_ITEMS:
   COLOR_REFLECTION_MAX | fresnel BOOL_ON_OFF |
   falloff FLOAT_FALLOFF | exponent FLOAT_EXPONENT |
   metallic FLOAT_METALLIC
IRID_ITEMS:
   thickness Amount | turbulence Amount
```

Data Structure

```
struct Finish_Struct
{
  SNGL Diffuse, Brilliance;
  SNGL Specular, Roughness;
  SNGL Phong, Phong_Size;
  SNGL Irid, Irid_Film_Thickness, Irid_Turb;
  SNGL Temp_Caustics, Temp_IOR, Temp_Dispersion, Temp_Refract, Reflect_Exp;
  SNGL Crand, Metallic;
  RGB Ambient, Reflection_Max, Reflection_Min;
  SNGL Reflection_Falloff;
  int Reflection_Type;
  SNGL Reflect_Metallic;
  int Conserve_Energy;
};
```

12.1.1 Metallic Highlight Modifier

The keyword metallic may be used with phong or specular highlights. This keyword indicates that the color of the highlights will be calculated by an empirical function that models the reflectivity of metallic surfaces.

Normally highlights are the color of the light source. Adding this keyword filters the highlight so that white light reflected from a metallic surface takes the color specified by the pigment

12.2 Lighting and Shading

The lighting calculations performed by POV-Ray in sequential order are described below.

First the normal on the intersection point is determined from the intersection point and object. The direction of the normal is flipped if its points away from the direction.

The weight is the contribution of a ray. When a ray is first shot it is set to 1. The value decreases for reflective and refractive rays.

12.2.1 Reflectivity determination

Before performing any color contribution on lighting POV-Ray determines the reflectivity of the surface if the surface is specularly reflective.

The weight of the reflective ray is,

```
Temp_Weight_Max = max3(Reflection_Max[pRED], Reflection_Max[pGREEN], Reflection_Max[pBLUE]);
Temp_Weight_Min = max3(Reflection_Min[pRED], Reflection_Min[pGREEN], Reflection_Min[pBLUE]);
*weight = *weight * max(Temp_Weight_Max, Temp_Weight_Min);
```

POV-Ray can determine the reflectivity of a surface in two ways.

The standard reflectivity is computed by,

```
If (fabs(Reflection_Falloff-1.0)>EPSILON)
    Reflection_Frac = pow(1.0 - cos_angle, Reflection_Falloff);
else
    Reflection_Frac = 1.0 - cos_angle;
if(fabs(Reflection_Frac)<EPSILON)
```

```
    Assign_RGB(reflectivity, Reflection_Min);

else if (fabs(Reflection_Frac-1.0)<EPSILON)

    Assign_RGB(reflectivity, Reflection_Max);

else

    CRGBLinComb2(reflectivity, Reflection_Frac, Reflection_Max, (1 - Reflection_Frac),
Reflection_Min);
```

falloff sets a falloff exponent in the variable reflection. This is the exponent telling how fast the reflectivity will fall off, i.e. linear, squared, cubed, etc. reflection min is how reflective the surface will be when seen from a parallel angle and reflection max is how reflective it will be if seen from the 90 degree angle.

Another way is by computing Fresnel reflectivity terms,

```
DBL sqx = Sqr(ior) + Sqr(cos_angle) - 1.0;
      if (sqx > 0.0)
      {
        g = sqrt(sqx);
        F = 0.5 * (Sqr(g - cos_angle) / Sqr(g + cos_angle));
        F = F * (1.0 + Sqr(cos_angle * (g + cos_angle) - 1.0) / Sqr(cos_angle * (g - cos_angle) +
1.0));

        F=min(1.0,max(0.0,F));
        CRGBLinComb2(reflectivity,F,Reflection_Max,(1.0-F),Reflection_Min);
      }
      else
        Assign_RGB(reflectivity, Reflection_Max);
```

Where F is the fresnel reflectivity term[29]

If the surface is metallic POV-Ray performs some calculations to determine the color of the reflected ray,

```
DBL R_M=Layer->Finish->Reflect_Metallic;

    DBL x = fabs(acos(Cos_Angle_Incidence)) / M_PI_2;
    DBL F = 0.014567225 / Sqr(x - 1.12) - 0.011612903;
    F=min(1.0,max(0.0,F));

    ListReflec[layer_number][0]*=
      (1.0 + R_M * (1.0 - F) * (LayCol[0] - 1.0));
    ListReflec[layer_number][1]*=
      (1.0 + R_M * (1.0 - F) * (LayCol[1] - 1.0));
    ListReflec[layer_number][2]*=
      (1.0 + R_M * (1.0 - F) * (LayCol[2] - 1.0));
```

The attenuation for the ray is computed by,

```
Att = (1.0 - (LayCol[pFILTER]*max3(LayCol[0],LayCol[1],LayCol[2]) + LayCol[pTRANSM]))
```

12.2.2 Ambient color

The light seen in dark shadowed areas comes from diffuse reflection off of other objects. This light cannot be directly modeled using ray-tracing. *A*mbient lighting is used to simulate the light inside a shadowed area.

Ambient light is light that is scattered everywhere in the room. It bounces all over the place and manages to light objects up a bit even where no light is directly shining. Computing real ambient light would take far too much time, so POV-Ray simulate ambient light by adding a small amount of white light to each texture whether or not a light is actually shining on that texture. The ambient keyword controls the amount of ambient light.

A color can be specified rather than a float after the ambient keyword in each finish statement.

Computing the ambient contribution value of the pixel I_a,

I_a = a * k * I * C

Here,

a=att
k=surface ambient reflection coefficient
I=light intensity
C=Surface Color

12.2.3 Diffuse Color

When light reflects off of a surface the laws of physics say that it should leave the surface at the exact same angle it came in. This is similar to the way a billiard ball bounces off a bumper of a pool table. This perfect reflection is called *specular reflection*. However, only very smooth polished surfaces reflect light in this way. Most of the time, light reflects and is scattered in all directions by the roughness of the surface. This scattering is called *diffuse reflection* because the light diffuses or spreads in a variety of directions. It accounts for the majority of the reflected light we see.

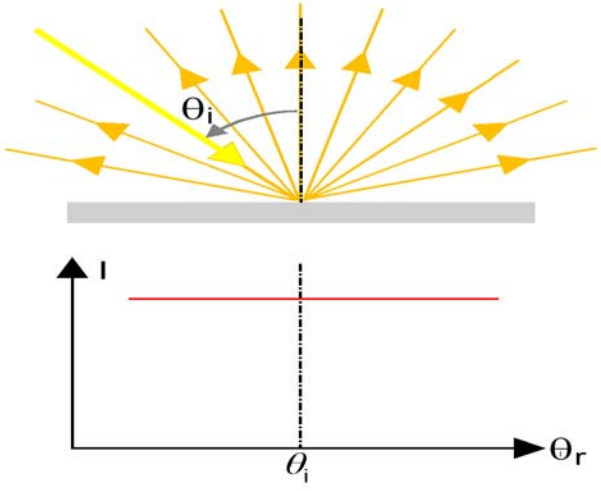


Fig. 12.3 : Diffuse Reflection

Calculate the diffuse color component I_d given by:

I_d = a * d * I * C * (N . L) ^ b

where d : surface's diffuse reflection coefficient
b : surface's brilliance
C : surface's color
N : surface's normal vector
L : light vector (pointing at the light)
I : intensity of the incoming light
a : attenuation factor

12.2.4 Specular highlight

The specular keyword in a finish statement produces a highlight which is very similar to Phong highlighting but it uses slightly different model. The specular model more closely resembles real specular reflection and provides a more credible spreading of the highlights occurring near the object horizons.

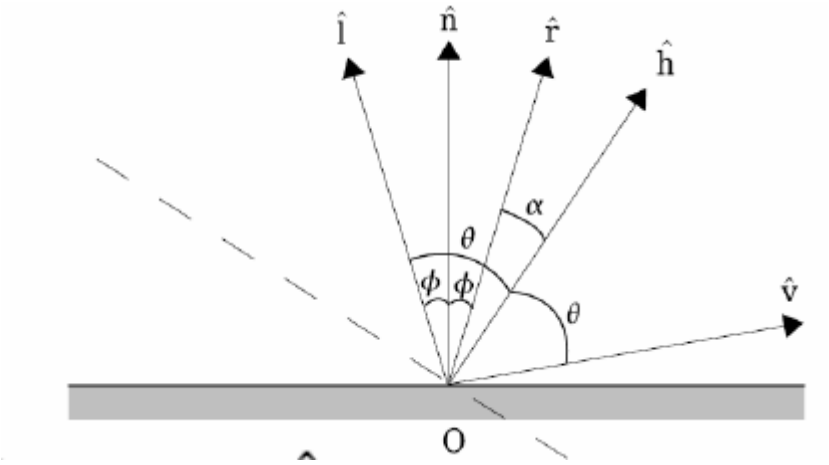


Fig. 12.4 : Specular Reflection

Calculate the specular reflected color component I_s given by:

I_s = s * C * (H . N) ^ (1 / r)

where s : surface's specular reflection coefficient
r : surface's roughness
C : surface's color/light color depending on the metallic flag
N : surface's normal
H : bisection vector between V and L

The bisecting vector H is calculated by

H = (L - V) / sqrt((L - V).(L - V))

```
if (Finish->Metallic > 0.0)
    {

      /*Calculate the reflected color by interpolating between
      the light source color and the surface color according
      to the (empirical) Fresnel reflectivity function.*/


      VDot(NdotL, Layer_Normal, Light_Source_Ray->Direction);

      x = fabs(acos(NdotL)) / M_PI_2;

      F = 0.014567225 / Sqr(x - 1.12) - 0.011612903;

      F=min(1.0,max(0.0,F));
      Cs[pRED]      =   Light_Colour[pRED]      *   (1.0   +   Finish->Metallic   *   (1.0   -   F)   *
(Layer_Pigment_Colour[pRED]   - 1.0));
```

```
      Cs[pGREEN] = Light_Colour[pGREEN] * (1.0 + Finish->Metallic * (1.0 - F) *
(Layer_Pigment_Colour[pGREEN] - 1.0));
      Cs[pBLUE]  = Light_Colour[pBLUE]  * (1.0 + Finish->Metallic * (1.0 - F) *
(Layer_Pigment_Colour[pBLUE]  - 1.0));

      CRGBAddScaledEq(Colour, Intensity, Cs);
    }
    else
    {
      Colour[pRED]   += Intensity * Light_Colour[pRED];
      Colour[pGREEN] += Intensity * Light_Colour[pGREEN];
      Colour[pBLUE]  += Intensity * Light_Colour[pBLUE];
```

12.2.5 Phong Highlights

The phong keyword in the finish statement controls the amount of Phong highlighting on the object. It causes bright shiny spots on the object that are the color of the light source being reflected.

The Phong method measures the average of the facets facing in the mirror direction from the light sources to the viewer.

Calculate the phong reflected color component I_p given by:

I_p = p * C * (R . L) ^ s

where p : surface's phong reflection coefficient
s : surface's phong size
C : surface's color/light color depending on the metallic flag
R : reflection vector
L : light vector (pointing at the light)

The reflection vector is calculated from the surface normal and the viewing vector (looking at the surface point):

R = -2 * (V . N) * N + V, with R . R = 1

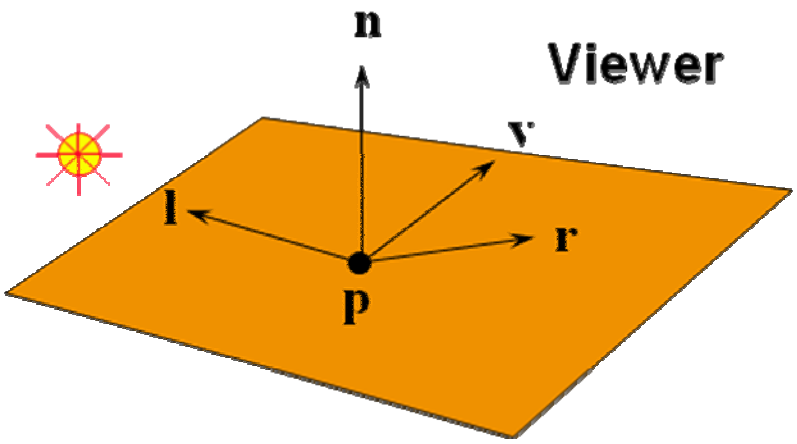


Fig. 12.5 : Phong Model

Same calculations for specular reflection is performed if the surface is metallic.

12.2.6 Specular Reflection

When light does not diffuse and it *does* reflect at the same angle as it hits an object, it is called *specular reflection*.

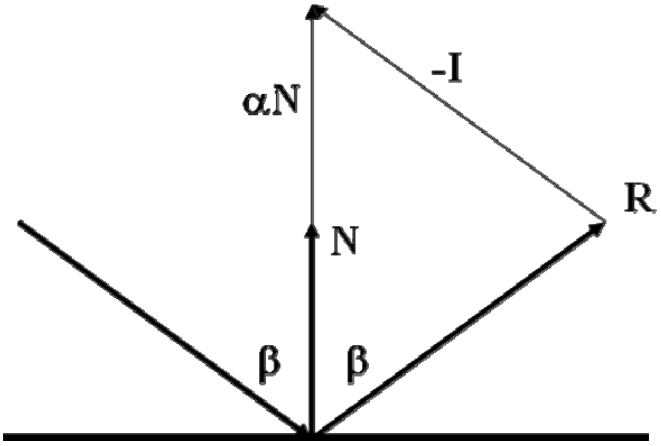


Fig. 12.6 : Calculations of reflected ray

The reflection vector is obtained by,

R = 2**(-I•N)N + I**

The new ray is traced back to the scene.

12.2.7 Refraction

When light passes through a surface either into or out of a dense medium the path of the ray of light is bent. Such bending is called *refraction*. The amount of bending or refracting of light depends upon the density of the material. Air, water, crystal and diamonds all have different densities and thus refract differently. The *index of refraction* or *ior* value is used by scientists to describe the relative density of substances. The ior keyword is used in POV-Ray in the interior to turn on refraction and to specify the ior value.

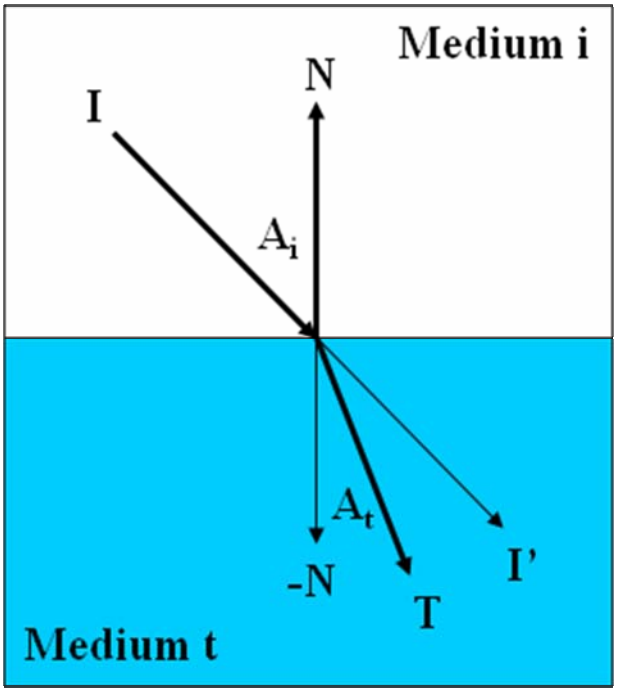


Fig. 12.7 : Calculations of Refracted Ray

Refracted ray β is obtained by the following equation,

$$\beta = \left(\frac{\mu_i}{\mu_t}\right)\cos(A_i) - \sqrt{1 + \left(\frac{\mu_i}{\mu_t}\right)^2 \left(\cos^2(A_i) - 1\right)}$$

A_i = **I•N**

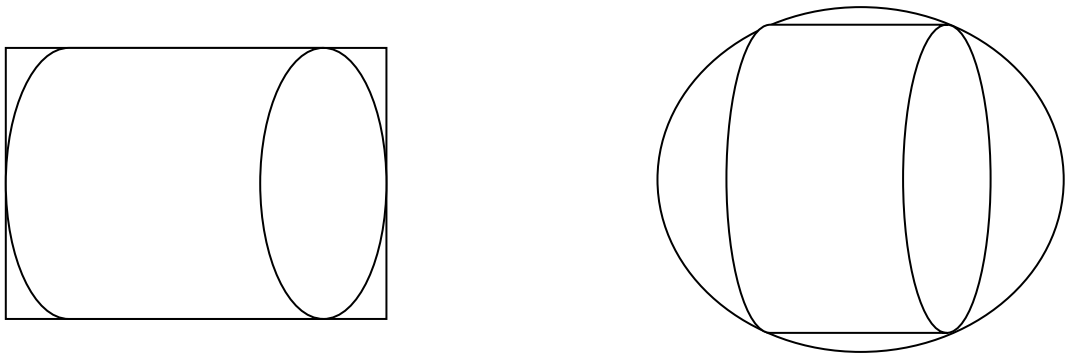
μ_i = refractive index of medium i

μ_t =refractive index of medium t

13.0 Ray Tracing Speed Ups

As we know ray tracing shoots a ray at each pixel of the view port. A 1024 by 1024 image of 100 objects would therefore require 100M intersection calculations. A lot of machines time and resources will be required to calculate those intersections.

It would be nice to know if the ray will actually hit an object before performing intersection calculations. There are several techniques and algorithms that try to determine it. They can be classified in to two broad catagories



Bounding box

Bounding Sphere

Fig. 13.1 : Bounding Shapes

Bounding box: A box that tightly encloses the object inside of it.

Bounding sphere: A sphere that tightly encloses the object inside it.

13.1 Spatial Partitioning

In spatial partitioning the space is divided in a top down approach. The bounding box of the scene is calculated first. In one approach, the bounding box is then divided into equal sized compartments. Each partition is associated with a list of objects that it contains either wholly or part of it. The lists are filled by assigning each object to the one or more partitions that contain it. Rays are shot at partitions after checking if the partition contains any object and the partitions are examined in order in which ray passes so that, once an intersection is found, no more partitions are checked . It is also necessary to determine if the intersection lies in that partition or not. The point of intersection and the rays ID can be cached with the object when the object is first encountered. Another approach is to divide the space unequally. An alternative adaptive subdivision method divides the scene using an octree.

These methods seem to be complicated to implement. An easier approach is to use bounding volume hierarchy.

13.2 Bounding Volume Hierarchy (BVH)

Bounding volumes provide a particular attractive way to decrease the amount of time needed for intersection calculations. Object are bounded by shapes like boxes or spheres, with which intersection calculations are less

expensive and before intersecting those objects, intersection with the bounding volume is performed. The ray does not need to be tested if an intersection calculation fails with the bounding volume. They bounding volumes are arranged in a tree like hierarchy known as bounding volume hierarchy.

A bounding-volume hierarchy is a tree structure on a set of geometric objects (the data objects). Each object is stored in a leaf of the tree. Each internal node stores for each of its children *O* an additional geometric object *V* (*O*), that encloses all data objects that are stored in descendants of *O*. In other words, *V* (*O*) is a bounding volume for the descendants of O.

POV-Ray uses a hierarchy of nested bounding boxes. This system compartmentalizes all finite objects in a scene into invisible rectangular boxes that are arranged in a tree-like hierarchy. Before testing the objects within the bounding boxes the tree is descended and only those objects are tested whose bounds are hit by a ray. This can greatly improve rendering speed. However for scenes with only a few objects the overhead of using a bounding system is not worth the effort. It is reasonable to use BVH if there are more than 5 objects in the scene.

POV-Ray uses boxes because the minimum (best-fit) bounding box for a given set of data objects is easy to compute, needs only few bytes of storage, and robust intersection tests are easy to implement and extremely fast.

Building Tree

The finite objects are first sorted along the longest axis. Then surface area heuristic is used to build the tree. This is because, the probability that a bounding volume is hit by a ray is roughly proportional to the bounding volumes surface area. The objects are split in a location so that, N1*A1 + N2*A2 is minimum where N1 and N2 are the number of objects in the two groups and A1 and A2 are the surface areas of the bounding boxes of the two groups. The process continues with objects until the bunching factor is reached which is 4. A node is created as the parent of the objects whose bounding volume holds all of them inside and the leaves are the objects. This process continues until the root of the tree is created. Once the tree is complete the infinite objects are stored on the first node of the tree. The node is flaged as infinite.

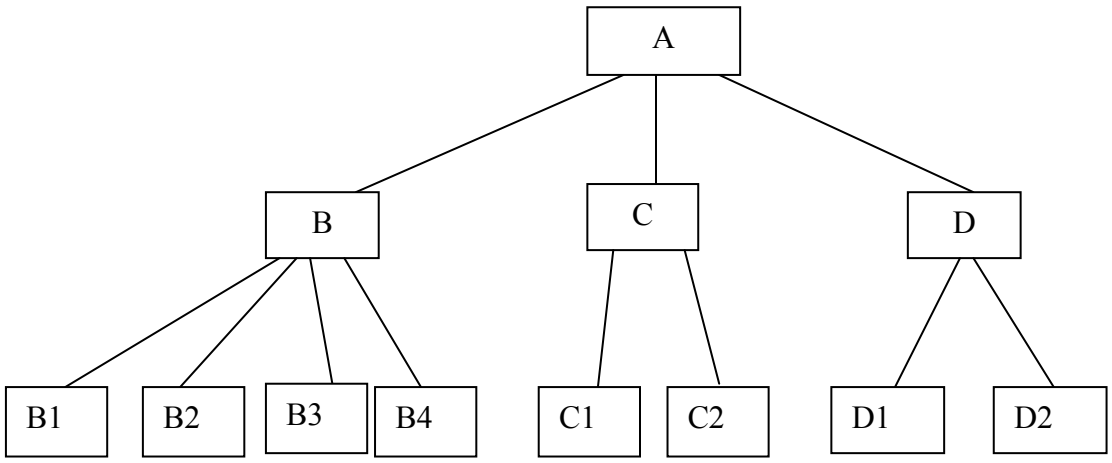


Fig. 13.2 : Bounding volume hierarchy

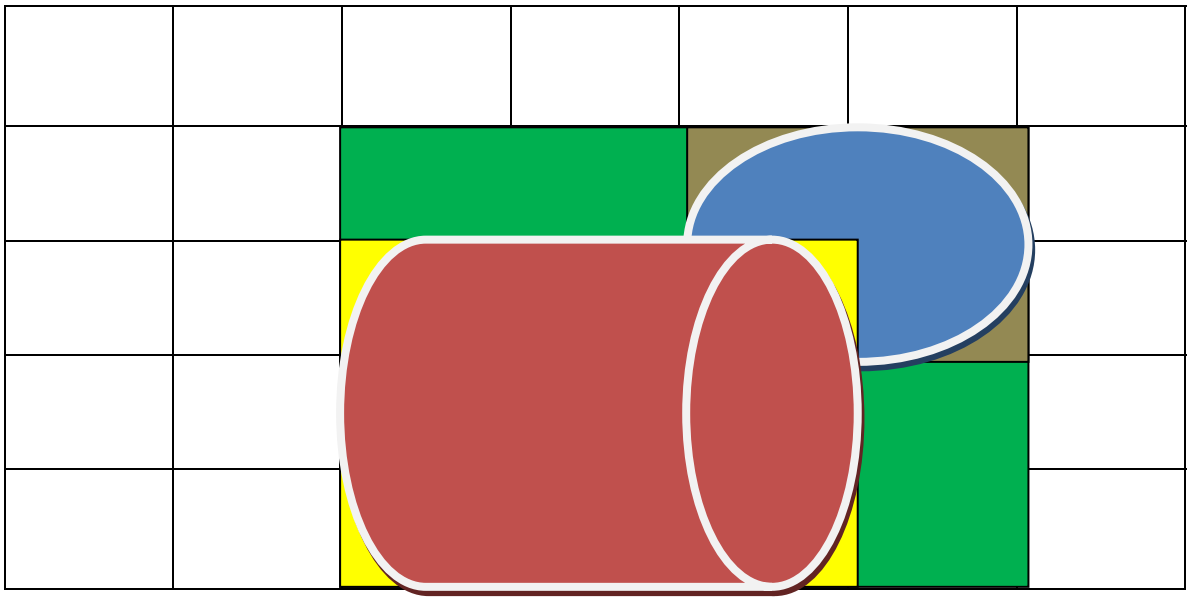


Fig. 13.3 : Vista buffer, projection of two bounding volumes projection enclosed by the parents projection

13.3 Vista buffer

The vista buffer is created by projecting the bounding box hierarchy onto the screen and determining the rectangular areas that are covered by each of the elements in the hierarchy. Only those objects whose rectangles enclose a given pixel are tested by the primary viewing ray. The vista buffer can only be used with perspective and orthographic cameras because they rely on a fixed viewpoint and a reasonable projection (i. e. straight lines have to stay straight lines after the projection).

Building Vista Buffer

Starting from the root of the bounding box tree the objects in each node is accessed. Only leaves are projected, not the nodes. The object is projected on the screen depending on the camera. Infinite objects such as planes are not projected.

Transform the points of the objects from the world co-ordinate system to the camera view volume or co-ordinate system. Then it is checked if all the points lie in front of the viewer. If the x and y co-ordinates are greater than -1then they are inside the camera view volume. This is because the camera lies at unit length distance from the origin. Then the x and y co-ordinate are checked if their absolute value is <=0.5(1+z), if it exceeds the value then the points are outside of the viewing volume and cannot be projected on the plane.

x_p=x/ 1+z

y_p=y/1+z

This is the projection of a point (x, y) in the near plane according to perspective projection.

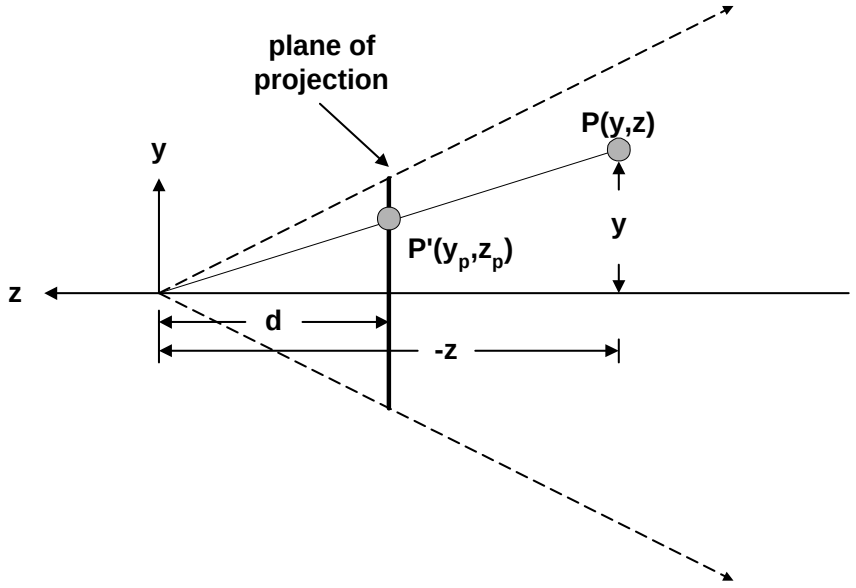


Fig.13.4 : Perspective Projection

The bounding volumes that are not totally inside the view volume may be partially inside. Each sides of the bounding volume is clipped by the top, bottom, left and right planes of the perspective view volume using Sutherland -Hodgman polygon clipping algorithm.

Finally the point is mapped onto viewport

x_v = Screen_Width/2 + Screen_Width * x_p

y_v = Screen_Height/2 - Screen_Height * y_p

Only the minimum and maximum projection values are stored as the projections. For anti aliasing the projection is increased in the min by and max by2. If anti aliasing is turned off then increase is by 1.

Once all the leaves of a node are projected the minimum and maximum projection values are stored. The upper level nodes also follow the same procedure. So, every node in the tree contains the projections of each of its children nodes.

Pruning the Tree

Before ray tracing each pixel, it is checked which object lies on the pixel. First the root is checked. If the root of the vista tree does not lie on pixel then the whole tree is pruned. Otherwise the tree is traversed in breadth first manner. If the pixel lies inside a node, it is stored in a heap. The nodes in the heap are picked one by one and the children of the nodes are checked in similar fashion. If a leaf is encountered then the bounding box is checked for intersection. According to the intersection depth the objects are sorted in a queue. The objects with the lowest bounding box intersection depths are checked for intersection first and so on.

13.4 Light Buffer

After an intersection is found, a ray is sent back to the light source to determine the contribution of light. If the ray intersects any other opaque object along the path, the point is considered to be in shadow. It is computationally the most expensive process of ray tracing algorithm because each object in the entire environment must be tested to see if it occludes every light source for every ray intersection point.

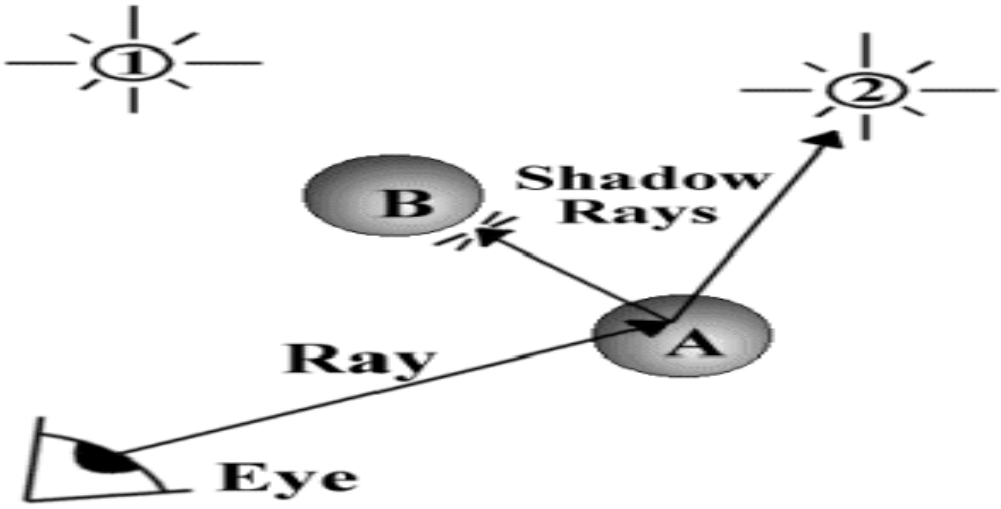


Fig. 13.5: Shadow ray

The light buffer is created by enclosing each light source in an imaginary box and projecting the bounding box hierarchy onto each of its six sides. Since this relies on a fixed light source, light buffers will not be used for area lights. It is also not used for parallel and fill lights. Reflected and transmitted rays do not take advantage of the light and vista buffer.

For each light source a light buffer is constructed. The light buffer is thought to be enclosed by a “hemicube”. The bounding volume hierarchy is projected on each of the six sides of the light buffer. The sides of the light buffer are divided into equal sized grids like the view port which is called the resolution.

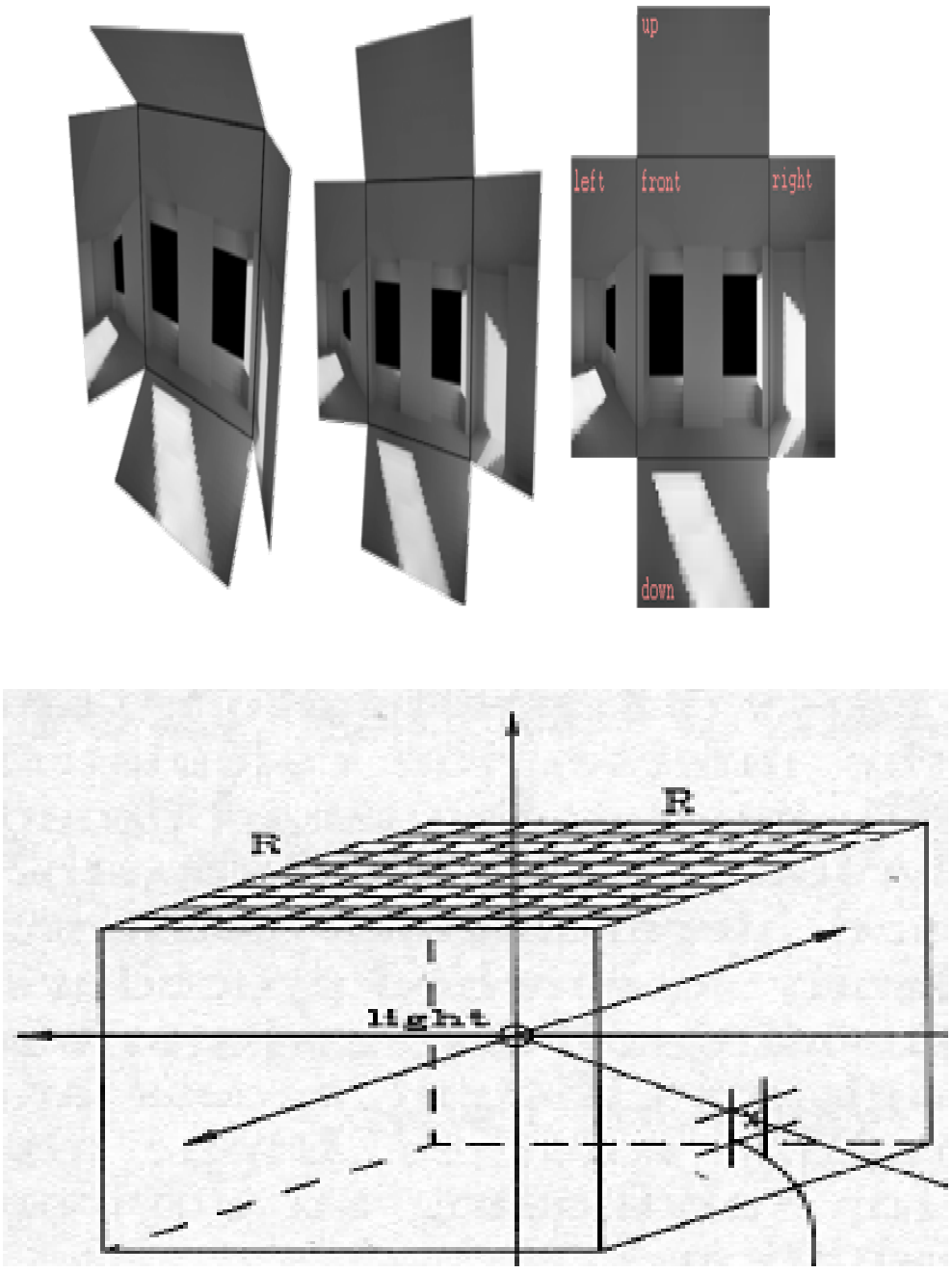


Fig. 13.6 : Hemicube and Light Buffer

The procedure of building the light tree is the same as the vista tree. Except that the axis perpendicular to the face is taken to be the depth axis or z axis and the co-ordinate values are changed. The bounding boxes are clipped in the lights viewing pyramid. The projection on the light buffer is

$x_p = x/z;$

$y_p = y/z;$

$x_l = MAX\_BUFFER\_ENTRY * x_p$

$y_l = MAX\_BUFFER\_ENTRY * y_p$

Intersection with Light Tree

The shadow rays dominant axis determines which side of the light will be hit by the ray. The co-ordinate of the dominant axis is changed to be the depth and the other co-ordinates are converted to light buffers co-ordinate. After that it is the same procedure as the vista tree intersection. Once an intersection with the shadow ray is found, the object is cached. For the next intersected points of the object intersection calculations are performed with the cached object first. If intersection is not found then the light tree is intersected.

14.0 ANTI ALIASING[30]

The ray-tracing process is in effect a discrete, digital sampling of the image with typically one sample per pixel. Such sampling can introduce a variety of errors. This includes a jagged, stair-step appearance in sloping or curved lines, a broken look for thin lines, moiré patterns of interference and lost detail or missing objects, which are so small they reside between adjacent pixels. The effect that is responsible for those errors is called *aliasing*.

Anti-aliasing is any technique used to help eliminate such errors or to reduce the negative impact they have on the image. In general, anti-aliasing makes the ray-traced image look *smoother*. POV-Ray gives option to turn antialiasing on or off.

When anti-aliasing is turned on, POV-Ray attempts to reduce the errors by shooting more than one viewing ray into each pixel and averaging the results to determine the pixel's apparent color. This technique is called super-sampling and can improve the appearance of the final image but it drastically increases the time required to render a scene since many more calculations have to be done.

There are two types of super-sampling methods.

14.1 Adaptive Non-recursive Super Sampling

In the adaptive, non-recursive, super-sampling method, POV-Ray initially traces one ray per pixel. If the color of a pixel differs from its neighbors (to the left or above) by at least the set threshold value then the pixel is super-sampled by shooting a given, fixed number of additional rays. The default threshold is 0.3 but it can be changed.

The threshold comparison is computed as follows. If r1, g1, b1 and r2, g2, b2 are the rgb components of two pixels then the difference between pixels is computed by

```
diff = abs(r1-r2) + abs(g1-g2) + abs(b1-b2)
```

If this difference is greater than the threshold then both pixels are super-sampled. The rgb values are in the range from 0.0 to 1.0 thus the most two pixels can differ is 3.0. If the anti-aliasing threshold is 0.0 then every pixel is super-sampled. If the threshold is 3.0 then no anti-aliasing is done. Lower threshold means more anti-aliasing and less speed.

The non-recursive method, takes nine super-samples per pixel, located on a 3*3 grid. The number of rows and columns can be defined by an ini option. The number of rows and colums are specified by a single value, so it is always a square grid.

14.2 Adaptive recursive Super Sampling

The second, adaptive, recursive super-sampling method starts by tracing four rays at the corners of each pixel. If the resulting colors differ more than the threshold amount additional samples will be taken. This is done recursively, i.e. the pixel is divided into four sub-pixels that are separately traced and tested for further subdivision. The advantage of this method is the reduced number of rays that have to be traced. Samples that are common among adjacent pixels and sub-pixels are stored and reused to avoid re-tracing of rays. The recursive character of this method makes the super-sampling concentrate on those parts of the pixel that are more likely to need super-sampling.

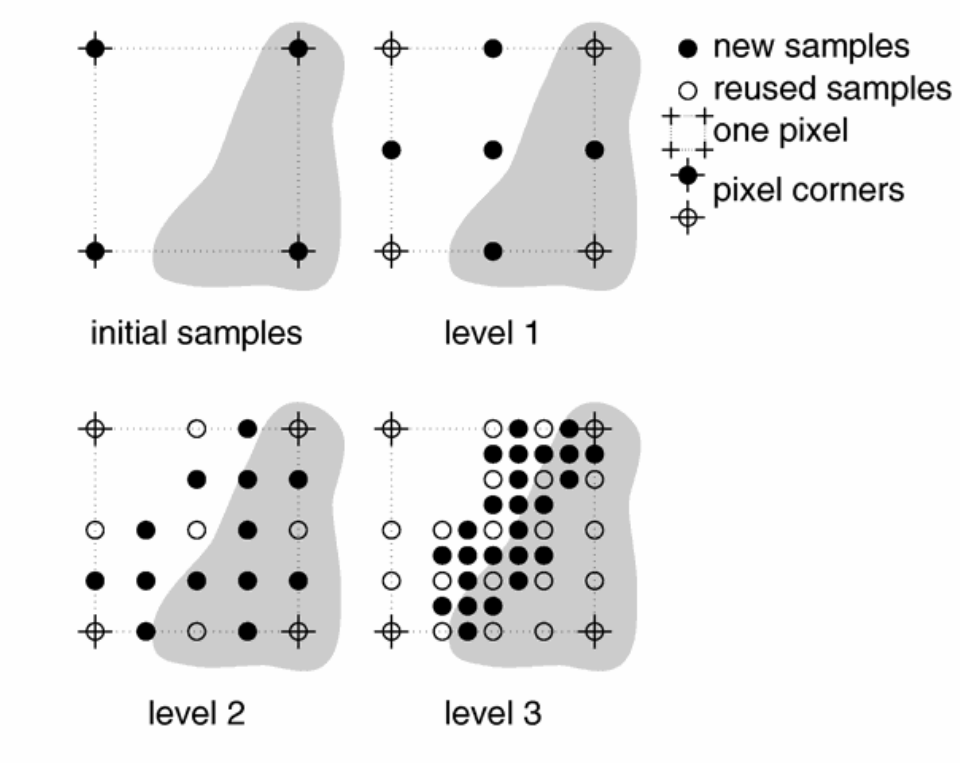


Fig. 14.1 : Adaptive Recursive Super Sampling

The maximum number of subdivisions can also be is specified but this is different from the adaptive, non-recursive method where the total number of super-samples is specified. A maximum number of *n* subdivisions results in a maximum number of samples per pixel that is given by the following table.

+Rn	Number of additional samples per super-sampled pixel for the non-recursive method +AM1	Maximum number of samples per super-sampled pixel for the recursive method +AM2
1	1	9
2	4	25
3	9	81
4	16	289
5	25	1089
6	36	4225
7	49	16641
8	64	66049
9	81	263169

14.3 Jittering

Another way to reduce aliasing artefacts is to introduce noise into the sampling process. This is called *jittering* and works because the human visual system is much more forgiving to noise than it is to regular patterns. The location of the super-samples is jittered or wiggled a tiny amount when anti-aliasing is used. Jittering can be turned on or off and the amount of jittering can be specified by an ini option as well. The default amount of 1.0 is the maximum jitter which will insure that all super-samples remain inside the original pixel.

If anti-aliasing is not used one sample per pixel is taken regardless of the super-sampling method specified.

15.0 DISCUSSION

Ray tracing by no means is a fast process, but its ability to produce realistic images is what makes it an attractive rendering method. To balance between rendering speed and render quality, the methods in POV-Ray are optimized as much as possible and the more advance rendering techniques are avoided as they will take longer time to render. POV-Ray did not use microfacets for global illumination method; rather they used simple and conventional lighting methods. Which is why some of the methods used in POV-Ray does not produce very realistic results. The soft shadows generated by POV-Ray often has shadow banding and a very large area light about [15X15] needs to be defined for generation of convincing soft shadows. The larger the grid of the area lights the longer it takes to render. POV-Ray also turns off most of its advanced rendering techniques like, photon tracing that produces realistic reflective and refractive caustics. The future versions of POV-Ray will hopefully address these issues.

16.0 Conclusion

This paper studies the ray tracing data structures and methods employed by POV-Ray and gives references for a detailed study on the techniques. POV-Ray has a big list of features and it was not possible for me to research on all of them. To understand the rendering mechanism of POV-Ray I manually stepped through the source code. I studied the research papers and books from which ideas of the implementations were taken from and other parallel algorithms that exist. While researching for this thesis I learned about many advanced techniques. A lot of progressed was achieved in photorealistic rendering in recent years. Currently extensive research is going on to make ray tracing process faster so that, it can be used in real time applications.

17.0 BIBLIOGRAPHY

[1] J. D. Foley, A. van Dam, J. F. Hughes, Steven K. Feiner. “The Quest for Visual Realism” in *Computer Graphics: Principle and Practice*. 2nd Ed, Addison-Wesley, 1990, pp. 605-607.

[2] A. Appel, “Some Techniques of Shading Machine Renderings of Solids”, *SJCC*, 1968, 37-45.

[3] Mathematical Applications Group Inc., ”3-D Simulated Graphics Offered by Service Bureau,” *Datamation,* 13(1), February 1968,69.

[4] R. A. Goldstein, and R. Nagel , “3-D Visual Simulation”, *Simulation*, 16(1), January 1971, 25-31.

[5] T Whitted., “An improved illumination model for shaded display,” *CACM*, 23(6), 82, 147-153.

[6] D.S. Kay, *Transparency, Refraction and ray tracing for Computer Synthesized Images*. M.S Thesis, Program of Computer Graphics, Cornell University, Ithaca, NY, January 1979.

[7] “The Early history of POV-Ray,”
Internet:http://www.povray.org/documentation/view/3.6.1/7/, [April 26, 2009].

[8] E. A. Haines, “Essential Ray Tracing Algorithms,” in Glassner, A.S, ed., *An Introduction to Ray Tracing*, Academic Press, London, 1989, 33-77.

[9] P. Hanrahan, “A Survey of ray Tracing Intersection Algorithms,” in Glassner, A.S., ed., *An Introduction to Ray Tracing*, Academic press, London, 1989, 79-119.

[10] D.R. Peachy, “Solid Texturing of Complex Surfaces,” *SIGGRAPH* 85, 279-286.

[11] K Perlin., “An Image Synthesizer,” *SIGGRAPH* 85, 287-296.

[12], J.F Blinn., Simulation of Wrinkled Surfaces,” *SIGGRAPH* 78, 286-292.

[13] Steve Upstill, *RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics,* Addison-Wesley Longman Publishing Co., Inc, Boston, MA, USA, 1989

[14] J. D. Foley, A. van Dam, J. F. Hughes, Steven K. Feiner. “Visible Surface Determination” in *Computer Graphics: Principle and Practice*. 2nd Ed, Addison-Wesley, 1990, pp.704-711.

[15] E.A. Haines, D.P. Greenberg, “The Light Buffer: A Shadow Testing Accelarator,” *CG and A*, 6(9), September 1986, 6-16.

[16] POV-Ray 3.6 Documentation Authors, *“2.1.4 Features”, POV-Ray Help,* POV-Ray(Version 3.6),POV-Ray Pty. Ltd. (2004).

[17]Haines, Eric, "Point in Polygon Strategies," *Graphics Gems IV*, ed. Paul Heckbert, Academic Press, p. 24-46, 1994.

[18] J. D. Foley, A. van Dam, J. F. Hughes, Steven K. Feiner. “Illumination and Shading” in Computer Graphics: Principle and Practice. 2nd Ed, Addison-Wesley, 1990, pp. 741-745] or pattern mapping

[19] J. D. Foley, A. van Dam, J. F. Hughes, Steven K. Feiner. “Illumination and Shading” in *Computer Graphics: Principle and Practice*. 2nd Ed, Addison-Wesley, 1990, pp. 1015-1018

[20] POV-Ray 3.6.0 Documentation Authors, *“*3.4.11 Patterns*”, POV-Ray Help,* POV-Ray(Version 3.6),POV-Ray Pty. Ltd. (2004)

[21] POV-Ray 3.6.0 Documentation Authors, *“*3.4.1 Pigment”*, POV-Ray Help,* POV-Ray(Version 3.6),POV-Ray Pty. Ltd. (2004)

[22] POV-Ray 3.6.0 Documentation Authors, *“*3.4.2 Normal”*, POV-Ray Help,* POV-Ray(Version 3.6),POV-Ray Pty. Ltd. (2004)

[23] POV-Ray 3.6.0 Documentation Authors, *“*3.4.6 Layered textures”*, POV-Ray Help,* POV-Ray(Version 3.6),POV-Ray Pty. Ltd. (2004)

[24] POV-Ray 3.6.0 Documentation Authors,” 3.3.7 Light Sources”*, POV-Ray Help,* POV-Ray(Version 3.6),POV-Ray Pty. Ltd. (2004)

[25] J. D. Foley, A. van Dam, J. F. Hughes, Steven K. Feiner. “Illumination and Shading” in *Computer Graphics: Principle and Practice*. 2nd Ed, Addison-Wesley, 1990, pp.772-773

[26] POV-Ray 3.6 Documentation Authors, *"2.1 Program Description", POV-Ray Help,* POV-Ray(Version 3.6),POV-Ray Pty. Ltd. (2004)

[27] J. D. Foley, A. van Dam, J. F. Hughes, Steven K. Feiner. "Illumination and Shading" in *Computer Graphics: Principle and Practice*. 2nd Ed, Addison-Wesley, 1990, pp.776-785.

[28] POV-Ray 3.6 Documentation Authors, *"3.4.3 Finish", POV-Ray Help,* POV-Ray(Version 3.6),POV-Ray Pty. Ltd. (2004)

[29] J. D. Foley, A. van Dam, J. F. Hughes, Steven K. Feiner. "Illumination and Shading" in *Computer Graphics: Principle and Practice*. 2nd Ed, Addison-Wesley, 1990, pp.766-771

[30] POV-Ray 3.6 Documentation Authors, 3.2.1.8.4 Anti Aliasing Options*, POV-Ray Help,* POV-Ray(Version 3.6),POV-Ray Pty. Ltd. (2004)