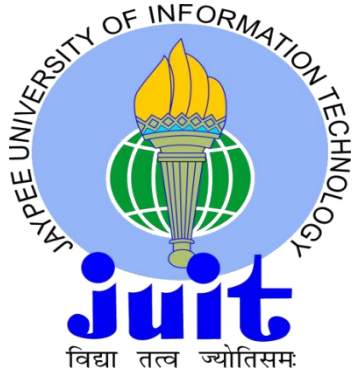
RAY TRACING USING JAVA

Submitted in partial fulfillment of the Degree of

Bachelor of Technology

Computer Science and Engineering



MAY-2014

Under the Supervision of

Dr. Pardeep Kumar

By

Mayank (101243)

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY,

WAKNAGHAT

TABLE OF CONTENTS

Chapter no.	Topics	Page No.
	Certificate	ii
	Acknowledgement	iii
	Summary	iv
	List of figures	v
1.	Introduction	1
2.	Brief survey of ray tracing	2
3.	Mathematics Involved	9
4.	Components of lighting	13
5.	Implementation in Java	20
6.	Result	36
7.	Conclusion	43
8.	References	45

CERTIFICATE

This is to certify that the work titled "**RAY TRACING USING JAVA**" submitted by **Mayank** in partial fulfilment for the award of degree of **B.Tech Computer Science and Engineering** of **Jaypee University of Information Technology, Waknaghat** has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

Signature of Supervisor ……

Name of Supervisor ……..

Designation ……..

Date ……..

ACKNOWLEDGEMENT

I express sincere appreciation to **Dr. Pardeep Kumar** for his guidance throughout the research and preparation of the thesis. I would like to thank him for his helpful comments, suggestions and giving me chance to work together.

Signature of the student ……..

Name of Student ……..

Date ……..

SUMMARY

Objective:

To develop a platform for ray tracing where we create a scene using a text file and then our java application will create a realistic image of the scene.

Description:

Ray tracing is a technique to produce a realistic image virtually. This method converts a 3 dimensional image to 2 dimensional image to make it suitable for display on the computer screen. Ray tracing takes a lot of time in rendering the image, so it makes this technique suitable for applications where image can be generated before its actual use, for example television animation.

In this project I have developed a Java application for ray tracing in which we define a scene in a text file. This text file serves as an input to the application which then decodes this scene into the various individual objects and lights. Then we take a simple camera as our viewer that defines our image plane. We render the continuous image pixel by pixel by shooting a ray from each pixel to the image plane and then we check for the intersections. Finally we calculate the colour at that pixel and display it.

LIST OF FIGURES

S.No.	Description	Page no.
1.	The basic rendering method	5
2.	Illustration of rays, vectors and angles required in ray tracing	6
3.	Ambient Light	14
4.	Diffuse light scattering	15
5.	The Diffuse Term	15
6.	Calculating diffuse lighting	16
7.	The specular term	16
8.	Example of different shininess exponents	17
9.	Calculating specular term	18
10.	Lighting effect of the three components	18
11.	Putting the lighting terms together	19
12.	The Original Scene before applying lighting and shadows	40
13.	Scene after applying lighting	41
14.	Final rendered image	42

CHAPTER 1: INTRODUCTION

Ray tracing is a method to produce virtually realistic images and is based on global illumination rendering method. This method traces rays of light from the eye (the camera) to the image plane into the scene. Then these rays are checked against all objects present in the scene to determine if they intersect any of these. If the ray passes through the scene without any intersection, then that pixel is shaded with the background colour. Ray tracing's primary advantage is that it is relatively straightforward to compute shadows and reflections.

Ray tracing was first developed in 1960s by scientists of Mathematical Applications Group and is a point sampling algorithm where we sample a continuous image by producing one or more rays from each pixel. Hence, leading to one of the disadvantage – aliasing. Aliasing is corroborated in computer graphics by spiky edges.

This method is capable of rendering a very high degree of visual realism as it simply captures the natural method of image generation through the eye but at a great computational cost. This makes ray tracing suitable for applications where the image can be rendered beforehand, as in television animation, and poorly suited for dynamic applications like gaming where speed is of utmost importance.

A ray tracing program mathematically identifies the path that each ray follows in reverse direction i.e. from the eye back to its point of origin. Each path consist of one or more straight line components and nearly always involves refraction, reflection, or shadow effects from points within the scene. Rays are assigned a colour based on the pigments the objects in the scene that the ray passes through and each pixel on the display corresponds to a ray.

In the last decade or so the efficiency and the quality of computer graphics has increased significantly. However the need for a higher quality interactive graphics still persists. We need high quality interactive graphics effects such as reflections, shadows, refractions etc. The complex implementations of these effects can only be an approximation due to limitations of the rendering methods. Ray tracing has been a standard for non-interactive computer graphics. Recent research has shown that ray tracing is also possible for interactive applications too. In this thesis we discover more about ray tracing and its implementation in Java.

CHAPTER 2: BRIEF SURVEY OF RAY TRACING

It necessary to be introduced with few terms before going into depth of ray tracing. “Ray tracing” actually covers a huge area in field of computer graphics, ranging from the basic concept of efficiently finding an intersection between a ray and set a primitives (plane and sphere in this project) to recursively tracing the images for realistic reflections. In computer graphics we typically want to know how our three dimensional image looks through a virtual camera. The process of computing image that such a virtual camera produces is called a rendering.

The current standard rendering method is a local illumination rendering method known as rasterization. This means that only the light from the light source is accounted for. Reflected rays and refracted rays do not contribute to the image.

To make the scene look more real we use ray tracing as it is a global illumination rendering method i.e. it takes into account the reflected and refracted rays too. This is requires for advanced effects such as reflection and shadows. Ray tracing works by tracing the path of light. We follow the path of rays of light

Of all the rays of light that is produced by light sources a lot of rays end up going away from our virtual camera. We only want to know which light rays enter our virtual camera and hence contribute to our image, so we follow the light backwards. This means that we start at our eye (the virtual camera) and trace, the ray determines the colour of the pixel we want to know about. When we encounter the point where the light is coming from we want to know the colour at that point. To calculate this colour we need to know what the incoming light at that point was. We recursively trace the rays from the light that falls on that point. Remember in a global illumination method the light can originate both from the light source or light reflected from other objects.

The main problem in ray tracing is to find the nearest intersection of the ray with an object. Though this is not covered in the implementation, I suggest using spatial index structure. Using sich structure we can check if a ray is in the vicinity of the object before we can check for an intersection.

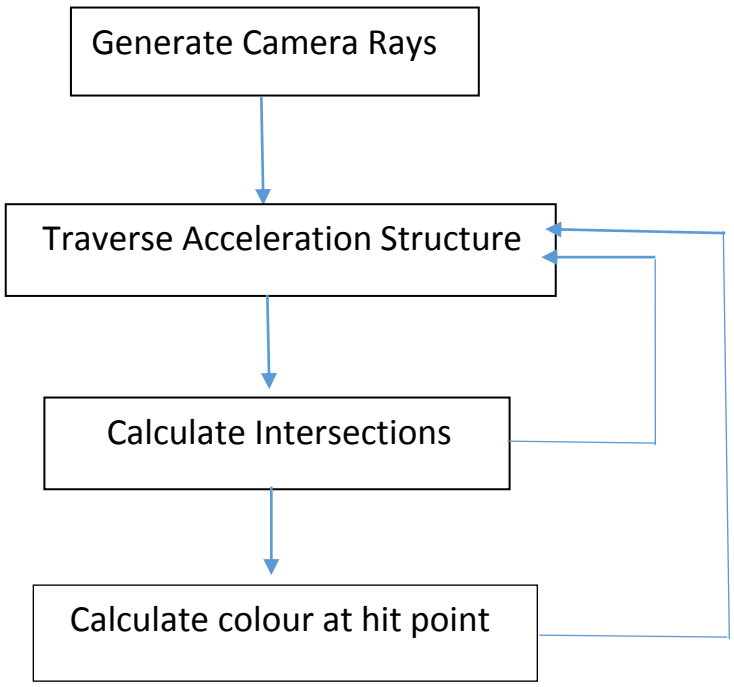


Fig 1. The basic rendering method

This simple but flexible rendering method makes ray tracing a much more suitable environment for advanced effects than rasterization. With rasterization we need complex and non-intuitive operations for such effects and often these effects are not possible at all because of the limitations of a local illumination rendering method.

Although ray tracing is a far more suitable environment for advanced effects it is traditionally known as being slow compared to rasterization rendering. The big difference between ray tracing and rasterization is that they work the other way around: Rasterization takes a primitive and draws it on screen, which is a very fast operation. With ray tracing we look for each pixel which primitive is under it. Suppose for example we would want to render a simple cube: With rasterization we would simply draw 16 triangles. With ray tracing we would need more computations as we would need to trace rays for each pixel on screen. This is why ray tracing is traditionally known as being “slow” compared to rasterization rendering.

Ray tracing is not actually "slow", the rendering time being logarithmic with the size of the scene. It does however have a high initial cost. With a very complex scene with advanced effects, it would be more efficient to use ray tracing than rasterization. This is because the rasterization approach would always draw all triangles and overwrite triangles which are further away. This means a lot of redundant operations. Additionally if the advanced effects are at all possible they would need multiple rendering passes. With ray tracing we would simply have no redundant calculations and would not need multiple rendering passes. With highly complex and high quality graphics the cost of rasterization rendering, with its redundant calculations and ineffective advanced effects, is higher than the cost of ray tracing.

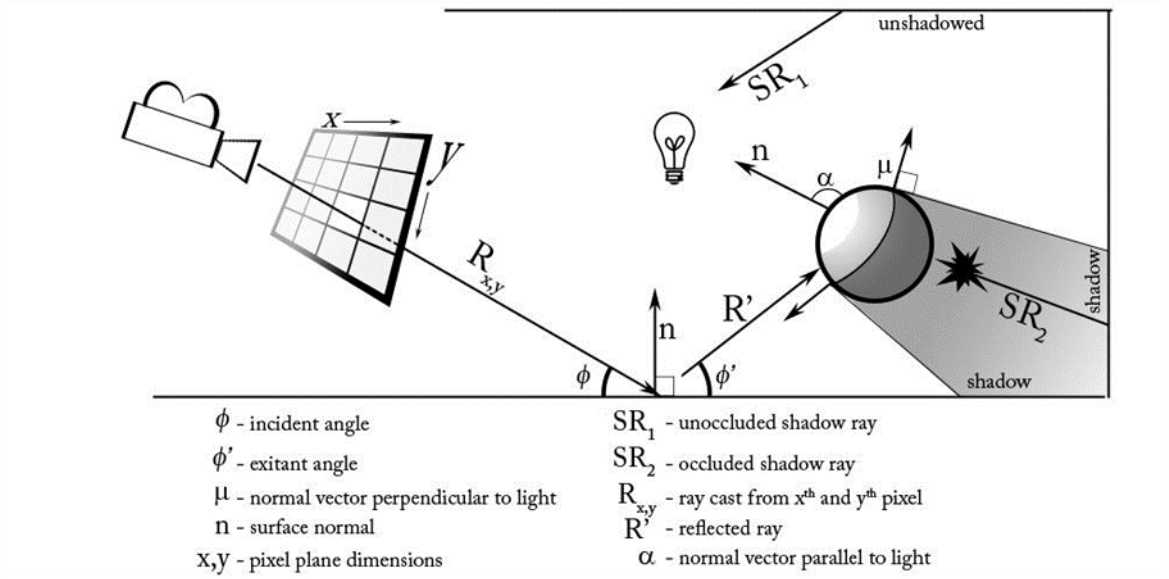


Fig 2. Illustration of rays, vectors and angles required in ray tracing

Ray-tracing finds its roots in the works of artists and mathematicians from centuries earlier. In Alberti's desire to understand the mathematics behind art and vision, he described sight and vision in a matter aligned perfectly with computer graphics, and especially ray-tracing. Alberti states: "Let us imagine the rays, like extended very fine threads gathered… going back together inside the eye where the sense of sight lines. They are like a trunk of rays

from which, like straight shoots, the rays are released and go out towards the surface in front of them"

According to Shirley and Morley (2003), ray-trace renderers are built upon a series of simple algorithms that are used, in turn, to generate digital images. In contrast to a rendering method like scan-line rendering, ray-trace rendering is becoming more-popular because of increased computing power and the renderers ability to cleanly solve problematic topics such as realistic material transparencies and object shadows.

Basic Ray-tracing Algorithm:

```
for(current pixel width : x)
    for(current pixel height : y)
     set tm = DOUBLEMAX;
        for(every object)
            if( ray intersects object)
               calculate t;
                if(t<tm)
                 set tm= t;
          if(tm=DOUBLEMAX)
           draw background colour at pixel x,y
           else
           draw closest shape at pixel x,y
Draw all pixels to image
```

The algorithmic process of ray-tracing is simple to understand. There exist objects to create and methods for describing their connections. The base list of required objects to create are: camera, ray (a position and direction), two dimensional array of pixels (an empty image), lights, and shapes. All objects in the scene are connected via independently calculated rays. In order for the renderer to see objects and render them, those objects must be in the line of sight of the camera. The line of sight is calculated as a ray, whose originating position is the camera and whose direction is determined by the location of the empty image. Rays are cast into the scene and wherever these line of sight rays intersect with various shapes, the renderer calculates which object is hit first and what color that object is. The color is calculated by casting rays, from a point on the surface, into the scene. Each surface point is checked for facing direction toward or away from the light sources. Any surface section facing toward a light whose view of the light is unobstructed will receive a lighting contribution. This contribution is based also on the amount to which the surface section points toward the light.

CHAPTER 3: MATHEMATICS INVOLVED

Equation of the ray,

Our ray can be defined with an origin and a direction. These are 3D vectors i.e. have three components(x, y, z). It will be useful later when our direction is normalised(whose length is 1). The path the ray takes can be represented as a parametric equation in t.

$$P = O + Dt$$

(P is the position, O is the origin, and D is the direction).

As our direction vector is normalised, the distance traveled by the ray from the origin is t.

Equation of Plane,

We can represent a normal to the plane using a vector(3D), and also a value to denote the distance to the origin (in the direction of the normal). Rather than the origin of the ray mentioned above, this is the distance to the origin at (0, 0, 0). E.g. if we look in the positive direction down the Z-axis from 0, we can amount to a wall 10 units away with a plane with a distance of 10 and a normal of (0, 0,-1).

The general equation for a plane is:

$$ax + by + cz + d = 0$$

Where d is the distance to the origin and a, b and c are the x, y and z components of the normal. To make the later equation more incisive, we can rewrite the plane equation in vector form as:

$$P \cdot N = -d$$

(P is the position, N is the normal, and d is the distance from the origin. Also the dot is the dot product)

From the above equations we know that t is the distance the ray has travelled from the origin. Now we want to find out how far the ray has travelled if it intersects our plane. Solving the ray and plane equations simultaneously will lead us to the required result.

The plane equation:

$$P \cdot N = -d$$

Taking P from the ray equation:

$$(O + Dt) \cdot N = -d$$

The dot product is distributive, so we simplify:

$$O \cdot N + (Dt) \cdot N = -d$$

Further:

$$O \cdot N + t(D \cdot N) = -d$$

Now arranging it,

$$t(D \cdot N) = -d - O \cdot N$$

A bit more…

$$t = \frac{-d - O \cdot N}{D \cdot N}$$

When the ray hits the plane, we can easily calculate the distance travelled by the ray using this formula. We can substitute this value into the ray equation to get the intersection point in 3D coordinates too.

The numerator and denominator are 0 in case the ray does not hit the plane.

After implementing what we have discussed so far, we will have a screen with half white (for the intersection hits) and half black (for the misses).

Equation of Sphere,

Sphere's Parametric form :

$(P-C).(P-C) = R^2$

P is the point on sphere, R is radius and C is the centre of sphere

Ray Equation in parametric form:

$P = O + Dt$

Substituting ray equation in sphere equation:

$(O+Dt-C).(O+Dt-C) - R^2 = 0$

Expanding :

$(D.D)\ t^2 + 2D.(O-C)t + (O-C).(O-C) - R^2 = 0$

This a quadratic equation in t which gives us two roots which signifies two intersection point on sphere.

Rewriting the equation as:

$at^2 + bt + e = 0$

Where:

$a = D^2$

$b = 2D.(O-C)$

$e = (O-C).(O-C) - R^2$

First we need to check whether ray is intersection with the sphere or not:

Determinant = $\sqrt{b^2 - 4ac}$

$\sqrt{b^2 - 4ac} < 0 \Rightarrow$ No intersection

$\sqrt{b^2 - 4ac} > 0 \Rightarrow$ Two solutions (enter and exit)

$\sqrt{b^2 - 4ac} = 0 \Rightarrow$ One solution (ray grazes sphere)

CHAPTER 4: COMPONENTS OF LIGHTING

Different types of light sources are used to give different effect:

1. **Ambient part**

This represents a fixed- intensity and also a fixed light source that affects each and every object in the scene equally. Ambient light is mainly used to give the scene a basic view of different objects present. It is very simple to implement and models how light is scattered or reflected many times producing a uniform effect.

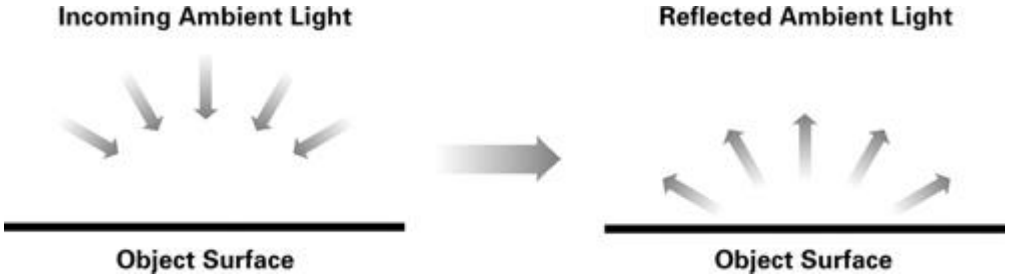


Fig 3. Ambient Light

The Mathematical term for ambient term,

ambient = K_a x GA

Where:

- K_a is the ambient reflectance of the material and
- GA is the Global Ambient i.e. the color of the incoming ambient light

2. **Diffuse Part**

The diffuse term is used for accounting the directed light reflected off a surface evenly in all the directions. Generally, diffuse surfaces are uneven on a minuscule scale, which reflect light in many directions. When incoming rays of light hit these aberrations, the light reflects in every directions.

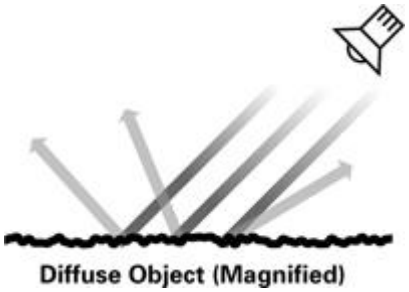


Fig 4. Diffuse Light Scattering

The intensity of light reflected directly depends on the angle of incidence of the light striking the surface. Surfaces having a dull finish, are said to be diffuse. The contribution of diffuse component at any particular point on a surface does not change, regardless of where the viewpoint is.

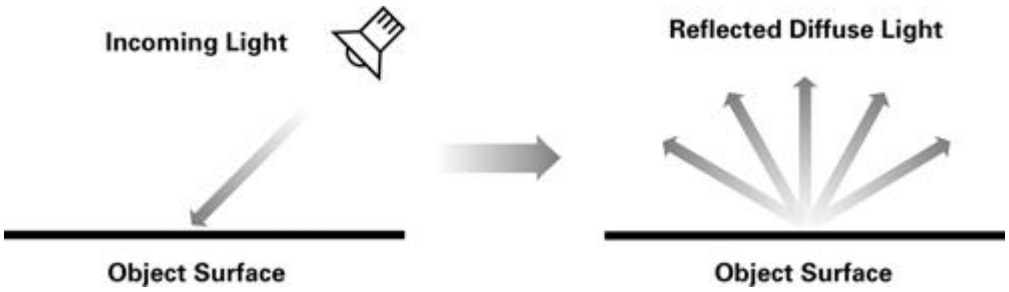


Fig 5. The Diffuse Term

The Mathematical term for diffuse term,

diffuse = K_d x LC x maximum(0, N.L)

Where:

- K_d is the diffuse colour of the material,
- LC is the light colour i.e. the colour of the incoming diffuse light,

- N represents the normalized surface normal,
- L denotes the normalized vector toward the light source

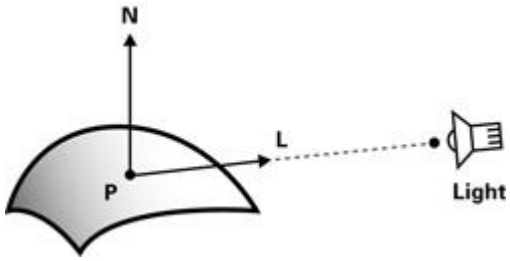


Fig 6. Calculating Diffuse Lighting

The angle between vectors N and L is calculated by taking the dot product. The greater the dot product, the smaller the angle will be between the vectors, and the surface will receive more incident light. Negative dot product denotes surfaces that faces away from the light, so the maximum(0,N.L) in the equation makes sure that these surfaces show no diffuse lighting.

3. Specular Part

The specular term represents the light scattered from a surface mostly around the mirror direction. On very smooth and shiny surfaces, such as polished metals the specular term is most prominent. Figure 7 illustrates the concept of specular reflection

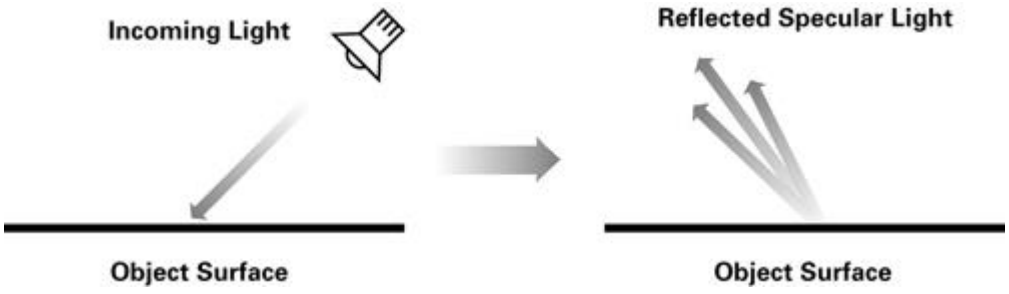


Fig 7. The Specular term

Unlike the diffuse and ambient lighting terms, the contribution of specular depends on the camera placement. If the camera is on a location that does not receive the reflected rays, the camera will not be able to see a specular highlight on the surface. The specular term also affected by the shininess component of the material and not only the specular colour component of the material. Less shiny surfaces have highlighted component spread on a larger area than compared to the shinier material where the highlight is tighter and to a point. We can see the difference between the different shininess components of material in the following figure.

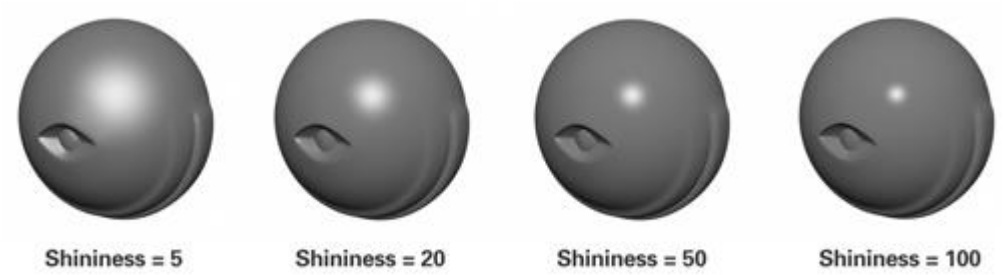


Fig 8. Example of different shininess exponents

The Mathematical term for specular term,

specular = K_s x LC x facing x (maximum(0, N · H)) $^{\text{shininess}}$

Where:

- K_s is the specular colour of the material,
- LC is light colour i.e. the the incoming specular light's colour,
- N represents the normalized surface normal,
- V denotes the normalized vector toward the viewpoint,
- L represents the normalized vector toward the light source,
- H denotes the normalized vector that is halfway between V and L,
- P is the point that is to be shaded, and
- facing is 0 if N · L is less than 0, and 1 otherwise.

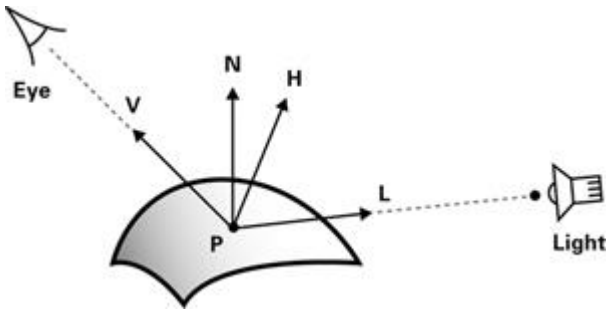


Fig 9. Calculating the Specular term

The specular appearance of the material becomes evident when the angle between the view vector V and the half-angle vector H is small. As H and V move farther apart, the exponentiation of the dot product of N and H makes sure that the specular appearance falls off quickly.

Also, if the diffuse term is zero the specular term is forced to zero because N · L (from diffuse part) is negative. This makes sure that specular highlights is not visible on surfaces that faces away from the light.

Adding the terms(Ambient, Diffuse and Specular) together,

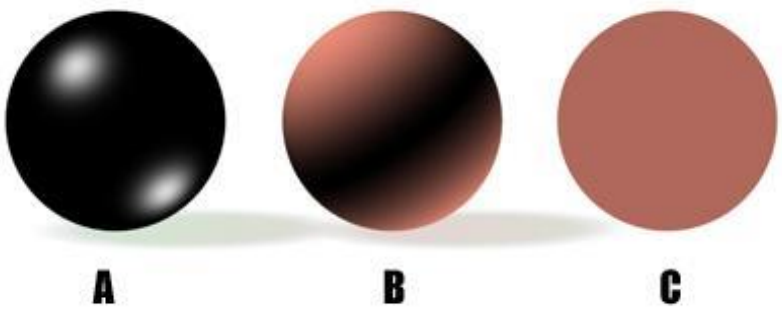


Fig 10. Lighting effect of the three components

A. Specular effect

B. Diffuse effect

C. Ambient effect

The individual effects are captured gain in Fig 9.

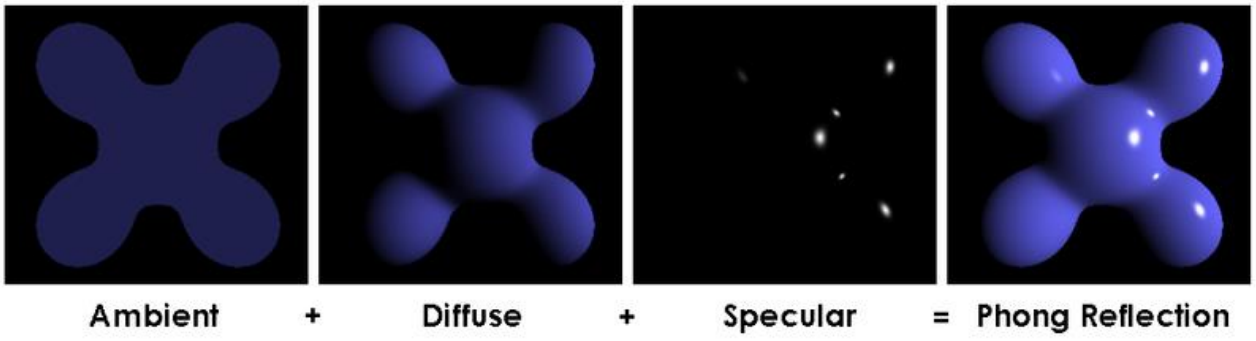


Fig 11. **Putting the lighting terms together**

Combining the ambient, diffuse, and specular terms gives the final lighting, as shown in Figure 10. Phong Reflection is the resulting Light effect.

CHAPTER 5: IMPLEMENTATION IN JAVA

Classes created:

1. SimpleCamera - SimpleCamera implements the viewer where the COP is on Z. A viewport is specified on the XY plane.
2. Vector – This class stores the 3D vector components and contains methods to manipulate the vectors.
3. Colour - Colour stores a RGB component and supports several methods to manipulate colour values
4. Sphere - Sphere represents a spherical object
5. DirectionalLight - DirectionalLight represents a light source that illuminates along parallel rays.
6. SimpleRay - SimpleRay implements the framework for the ray-tracing. It operates as an applet. It also implements a thread which periodically forces a screen repaint so that the user is kept aware of how far the ray-tracing has proceeded.
7. Light – This is superclass that each type of light must extend.
8. Material - Material describes the surface properties of an object using ambient, diffuse and specular components.
9. SceneReader - A class to read values from a file
10. SceneWriter - A simple output class to write values to a file.
11. Scene – Scene creates the scene and stores the different components in its respective classes
12. Point - Point stores and manipulates xyz triples representing 3 space points.
13. PointLight - PointLight represents a light sources that illuminates from a fixed origin
14. Object - Object superclass each type of object must extend and implement this class.

15. Plane - Plane represents an infinite plane object

16. Ray - Ray stores a ray origin point and direction vector.

Class SimpleCamera

Constructor
`SimpleCamera()` Creates a camera with resolution of 200 by 200
`SimpleCamera(int a, int b)` Creates a camera with width, a and height, b.

Methods	
`void`	`initBasic(int a, int b)`
`Ray`	`ray(int x, int y)` Returns the ray number x, y from the model of camera
`void`	`setCOP(double cent)` Sets the centre of projection equal to cent on the Z axis
`void`	`setResolution(int x, int y)` Set the resolution of the camera in terms of the number of rays to cast.
`void`	`setVPWindow(double xmin, double xmax, double ymin, double ymax)` Set the viewport window on the XY plane

Class Vector

Variables	
`double`	`X` Holds the X component
`double`	`Y` Holds the Y component
`double`	`Z` Holds the Z component

Constructor

`Vector()` Creates a vector and initializes it to zero
`Vector(double s, double b, double c)` Creates a vector with the given a,b and c values
`Vector(Point p)` Point object is converted to a new vector
`Vector(Vector v)` Creates a vector by imitating an existing vector

Methods

`void`	`add(Vector a)` This vector is added to the given vector
`Vector`	`add(Vector a, Vector b)` The two vectors are added and written it to a given destination vector.
`void`	`add(Vector vd, Vector v1, Vector v2)` Two vectors are added and written it to a given destination vector.
`void`	`copy(Vector a)` An existing vector is copied to this vector
`void`	`cross(Vector a)` Cross product is calculated with this vector
`Vector`	`cross(Vector a, Vector b)` A new vector is returned with the cross product of the two given vectors.
`void`	`cross(Vector vd, Vector v1, Vector v2)` Cross product of the two given vectors is written to the given desination vector.
`double`	`dot(Point p)` Dot product of point and the vector is calculated.
`double`	`dot(Vector v)` Dot product of the two vectors
`double`	`dot(Vector v1, Vector v2)` Returns the dot product of the two given vectors.

void	`negate()` Vector is negated
Vector	`negate(Vector a)` Creates a new vector after negating a given vector.
double	`norm()` Length of this vector is calculated
void	`normalise()` Vector is normalized
Vector	`normalised()` Creates a new vector as the normalization of this vector
void	`print(SceneWriter ofs)` Prints a human readable form of the vector
void	`read(SceneReader ifs)` Read the vector from the given source
void	`scale(double sc)` The vector is scaled by the given factor sc
Vector	`scale(Vector a, double sc)` Scale method creates a vector by scaling a existing vector.
void	`scale(Vector d, Vector a, double sc)` Scale method that writes a vector after scaling the given vector.
Void	`Set(double a, double b, double c)` Vector is set to the given a, b and c values
double	`squarednorm()` Squared length of this vector is calculated
Vector	`subtract(Point a, Point b)` Creates a vector and initializes as the difference between points a and b.
void	`subtract(Vector a)` The given vector is subtracted from this vector
void	`subtract(Vector d, Point a, Point b)` The difference between two points is calculated and written it the destination vector.
void	`subtract(Vector d, Vector a, Vector b)` The difference between two vectors is calculated and written to the given destination vector.
void	`write(SceneWriter ofs)` Write the vector to the given destination

Class Colour

Field Summary

double	`B` Holds the blue component of the colour
double	`G` Holds the green component of the colour
double	`R` Holds the red component of the colour

Constructor Summary

`Colour()` Create a new colour and set it to the default colour (black)
`Colour(Colour c)` Create a new colour by copying an existing colour
`Colour(double r, double g, double b)` Create a new colour with the given red green and blue values

Method Summary

void	`add(Colour c)` Add the given colour to this colour
Colour	`add(Colour c1, Colour c2)` Implementation of colour add function that creates a new colour object from the result.
void	`add(Colour cd, Colour c1, Colour c2)` Implementation of colour add function that write the result of the addition to a given colour object.
void	`clamp()` Clamp each component of the colour to lie within the range 0.0 to 1.0
void	`copy(Colour c)` Copying an existing colour to this colour
boolean	`isBlack()` Test if the colour is black
void	`mult(Colour c)` Multiply the given colour to this colour

Colour	`mult(Colour c1, Colour c2)` Implementation of colour mult function that creates a new colour object from the result.
void	`mult(Colour cd, Colour c1, Colour c2)` Implementation of colour mult function that write the result of the multiplication to a given colour object.
void	`print(SceneWriter os)` Print a human readable version of the colour definition to the given destination
void	`read(SceneReader is)` Read the colour from the given source
void	`reset()` Reset the colour to the default colour (black)
void	`scale(Colour cd, Colour c1, double s)` Scale function that writes the colour scaled by the given factor to a given destination
Colour	`scale(Colour c1, double s)` Scale function that returns a new colour scaled by the given factor
void	`scale(double s)` Scale this colour by the given factor
void	`write(SceneWriter os)` Write the colour to the given destination

Class Sphere

Field Summary

Point	`Centre` Holds the point that defines the sphere centre
double	`Radius` Holds the radius of the sphere

Constructor Summary

`Sphere()` Create a default sphere
`Sphere(Material n, Point p, double r)` Create a sphere with the given material, centre and radius Note values are referenced not copied in the new object

Method Summary	
double	`intersect(Ray ray)` Find the intersection of the plane and a given ray.
Vector	`normal(Point p)` Find the normal of an object at the given point on its surface.
void	`print(SceneWriter os)` Print a human readable version of the sphere definition to the given destination
void	`read(SceneReader is)` Read the sphere from the given source
void	`write(SceneWriter os)` Write the sphere to the given destination

Class DirectionalLight

Field Summary	
Vector	`Direction` Stores the direction the light travels in.

Constructor Summary
`DirectionalLight()` Create a default light.
`DirectionalLight(Vector v)` Create a new directional light that shines along given direction vector.

Method Summary	
void	`print(SceneWriter os)` Print a human readable version of the light definition to the given destination
void	`read(SceneReader is)` Read the light from the given source.
void	`write(SceneWriter os)` Write the light to the given destination

Class SimpleRay

Method Summary	
void	`init()` Method for applet initialize The scene to read is taken from the "scene" property in the applet tag on the web page.
void	`paint(java.awt.Graphics g)` Paints the screen
void	`run()` Run method is called by the "kicker" thread.
void	`start()` Method called to start an applet after initialization.
void	`stop()` Method called to stop an applet after initialization.
void	`update(java.awt.Graphics g)`

Class Light

Field Summary	
Colour	`Intensity` Holds the colour of this light

Constructor Summary
`Light()`

Method Summary	
abstract void	`print(SceneWriter os)` Print a human readable version of the light definition to the given destination

abstract void	read(SceneReader is) Read the light from the given source
abstract void	write(SceneWriter os) Write the light to the given destination

Class Material

Field Summary

Colour	Ambient Holds the ambient component of this surface
Colour	Diffuse Holds the ambient component of this surface
double	Shininess Holds the shininess factor this surface
Colour	Specular Holds the specular component of this surface

Constructor Summary

Material() Create a default material
Material(Material m) Create a new material by copying an existing material.

Method Summary

void	print(SceneWriter os) Print a human readable version of the material definition to the given destination
void	read(SceneReader is) Read the material from the given source
void	write(SceneWriter os) Write the material to the given destination

Class SceneReader

Method Summary	
void	`close()` Close the file when finished
boolean	`eof()` Return `true` if the end of file has been reached.
char	`readChar()` Read a `char` value from file.
double	`readDouble()` Read a `double` value from file.
float	`readFloat()` Read a `float` value from file.
int	`readInt()` Read an `int` value from file.
long	`readLong()` Read a `long` value from file.

Class SceneWriter

Constructor Summary
`SceneWriter(java.io.File f)` Construct SceneWriter object given a file.
`SceneWriter(java.io.OutputStream os)` Create a SceneWriter from a stream

Method Summary	
void	`flush()` Flush the writer
void	`writeChar(char c)` Write a char value to a file.

void	writeDouble(double d) Write a double value.
void	writeFloat(float f) Write a float value.
void	writeInt(int i) Write an int value to a file.
void	writeLong(long l) Write a long value.
void	writeNewline() Write a newline to a file.
void	writeString(java.lang.String s) Write a String value to a file.

Class Scene

Field Summary

Colour	Ambient

Method Summary

int	getNumberLights()
int	getNumberObjects()
boolean	intersect(Ray ray, Colour colour, int depth)
void	print(SceneWriter os)
void	read(SceneReader is)
void	setLight(int i, Light o)
void	setNumberLights(int n)

void	setNumberObjects(int n)
void	setObject(int i, Object o)
void	write(SceneWriter os)

Class Point

Field Summary

double	X Holds the X component
double	Y Holds the Y component
double	Z Holds the Z component

Constructor Summary

Point() Create a new point and set it to zero
Point(double x, double y, double z) Create a new point with the given x,y and z values
Point(Point p) Create a new point by copying an existing point

Method Summary

void	add(Point pd, Point p, Vector v) Function that write addition of a point by adding a vector to a given destiation point
Point	add(Point p, Vector v) Function create a new point by adding a vector to a point.
void	add(Vector v) Add the given offset vector to this point

void	`copy(Point p)` Copy an existing point to this point
boolean	`equals(Point p)` Test if two points are equal
void	`print(SceneWriter os)` Print a human readable version of the point definition to the given destination
void	`read(SceneReader is)` Read the point from the given source
void	`set(double x, double y, double z)` Set this point to the given x,y and z values
void	`write(SceneWriter os)` Write the point to the given destination

Class PointLight

Field Summary

Point	`Origin` Stores the origin of the point light.

Constructor Summary

`PointLight()` Create a default light.
`PointLight(Point p)` Create a new point light that at the given point.

Method Summary

void	`print(SceneWriter os)` Print a human readable version of the light definition to the given destination
void	`read(SceneReader is)` Read the light from the given source
void	`write(SceneWriter os)` Write the light to the given destination

Class Object

Field Summary	
Material	SurfaceMaterial Holds the material of this object

Method Summary	
abstract double	intersect(Ray ray) Find the intersection of an object and a given ray.
abstract Vector	normal(Point pt) Find the normal of an object at the given point on its surface.
abstract void	print(SceneWriter os) Print a human readable version of the object definition to the given destination
abstract void	read(SceneReader is) Read the object from the given source
abstract void	write(SceneWriter os) Write the object to the given destination

Class Plane

Field Summary	
double	Constant Holds the constant of the plane equation
Vector	Normal Holds the normal that defines this plane

Constructor Summary
Plane() Create a default plane
Plane(Material n, Vector v, double c) Create a plane with the given material and plane equation Note values are referenced not copied in the new object

Class Ray

Field Summary

Vector	`Direction` Holds the ray direction
Point	`Origin` Holds the ray origin

Method Summary

Point	`getPointAt(double t)` Get a point along a ray.

CHAPTER 6: RESULT

The scene file:

```
# number of objects
9
#right wall
1 -1.0  0.0  0.0       -2.0     0.9 0.9 0.3  0.8 0.8 0.1   0.0 0.0 0.0  2.0
#bottom wall
1  0.0  1.0  0.0       -2.0     0.2 0.2 0.2  0.1 0.1 0.1   0.7 0.7 0.7  2.0
#top wall
1  0.0 -1.0  0.0       -2.0     0.3 0.9 0.3  0.1 0.8 0.1   0.0 0.0 0.0  2.0
#left wall
1  1.0  0.0  0.0       -2.0     0.3 0.3 0.9  0.1 0.1 0.8   0.0 0.0 0.0  2.0
#back wall
1  0.0  0.0  1.0       -2.0     0.9 0.3 0.3  0.8 0.1 0.1   0.0 0.0 0.0  2.0
#sphere 1
0  1.2  0.4 -0.4           0.5       0.2 0.2 0.2  0.6 0.4 0.2   0.1 0.3 0.3  5.0
#sphere 2
0  0.0 -0.6  0.0           0.4       1.0 1.0 1.0  1.0 1.0 1.0   0.0 0.0 0.0  2.0
#sphere 3
0 -1.0  0.0 -1.0           1.0       0.0 0.0 0.0  0.0 0.0 0.0   0.8 0.8 0.8  5.0
#sphere 4
0  1.0 -0.6  0.0           0.5       0.2 0.2 0.2  0.6 0.4 0.2   0.1 0.3 0.3  5.0
```

#ambient colour

0.3 0.3 0.3

#number lights

2

light 1 - point light

0 0.3 0.5 0.5 -1.9 1.9 0.0

light 2 - directional light

1 0.8 0.3 0.3 1.9 0.0 0.0

Interpretation:

Scene has 9 objects

Scene has 2 lights

Number of objects 9

Plane with normal: [-1.0,0.0,0.0] and constant: -2.0

Material: ambient (0.9,0.9,0.3) diffuse (0.8,0.8,0.1) specular (0.0,0.0,0.0) shininess 2.0

Plane with normal: [0.0,1.0,0.0] and constant: -2.0

Material: ambient (0.2,0.2,0.2) diffuse (0.1,0.1,0.1) specular (0.7,0.7,0.7) shininess 2.0

Plane with normal: [0.0,-1.0,0.0] and constant: -2.0

Material: ambient (0.3,0.9,0.3) diffuse (0.1,0.8,0.1) specular (0.0,0.0,0.0) shininess 2.0

Plane with normal: [1.0,0.0,0.0] and constant: -2.0

Material: ambient (0.3,0.3,0.9) diffuse (0.1,0.1,0.8) specular (0.0,0.0,0.0) shininess 2.0

Plane with normal: [0.0,0.0,1.0] and constant: -2.0

Material: ambient (0.9,0.3,0.3) diffuse (0.8,0.1,0.1) specular (0.0,0.0,0.0) shininess 2.0

Sphere with centre: [1.2,0.4,-0.4] and radius: 0.5

Material: ambient (0.2,0.2,0.2) diffuse (0.6,0.4,0.2) specular (0.1,0.3,0.3) shininess 5.0

Sphere with centre: [0.0,-0.6,0.0] and radius: 0.4

Material: ambient (1.0,1.0,1.0) diffuse (1.0,1.0,1.0) specular (0.0,0.0,0.0) shininess 2.0

Sphere with centre: [-1.0,0.0,-1.0] and radius: 1.0

Material: ambient (0.0,0.0,0.0) diffuse (0.0,0.0,0.0) specular (0.8,0.8,0.8) shininess 5.0

Sphere with centre: [1.0,-0.6,0.0] and radius: 0.5

Material: ambient (0.2,0.2,0.2) diffuse (0.6,0.4,0.2) specular (0.1,0.3,0.3) shininess 5.0

Ambient light is 0.3 0.3 0.3

Number of lights 2

Point light with colour: (0.3,0.5,0.5) and origin: [-1.9,1.9,0.0]

Directional light with colour: (0.8,0.3,0.3) and direction: [1.9,0.0,0.0]

Rendering of the scene:

1. The Original scene :

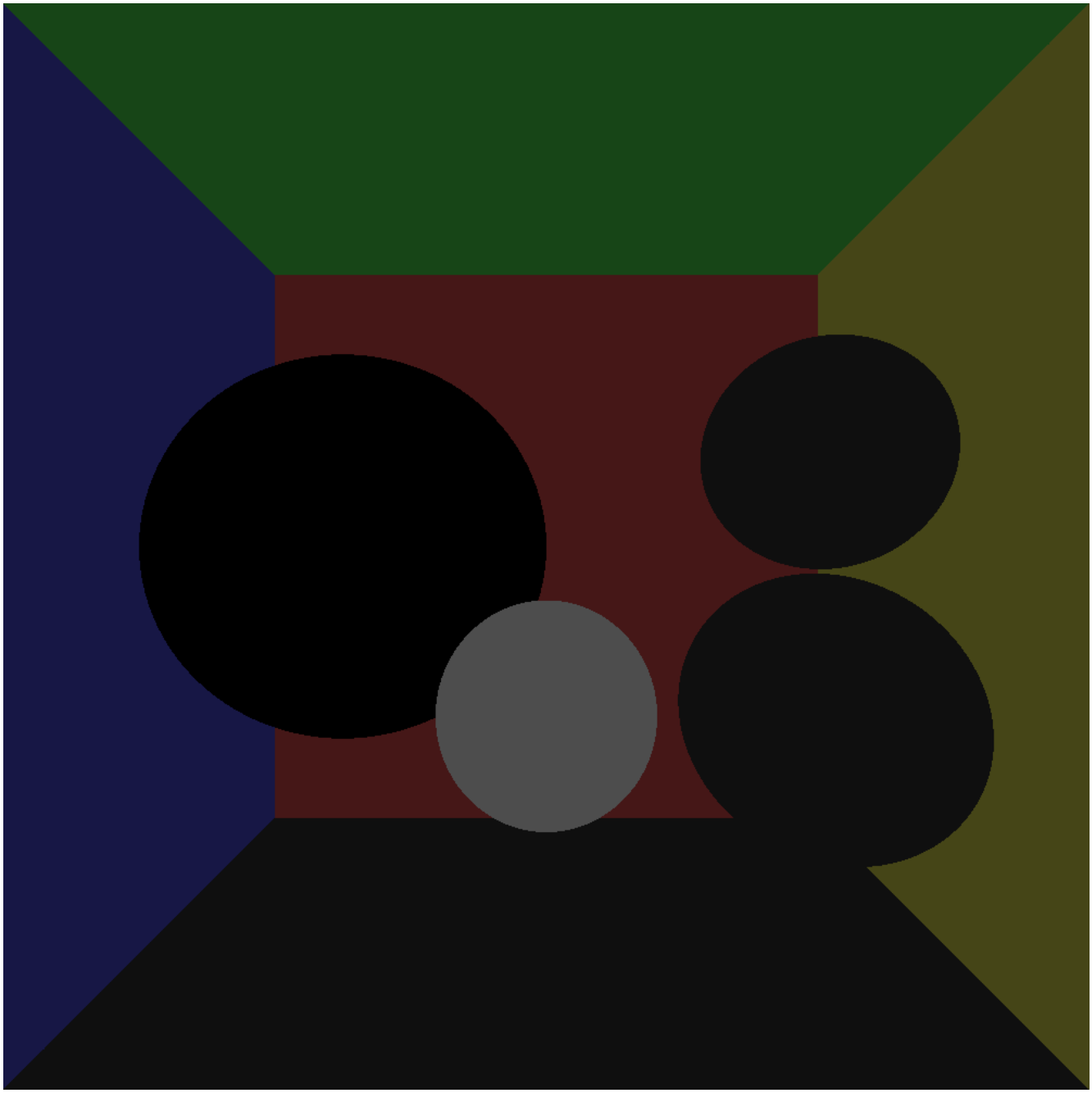


Fig 12. The Original Scene before applying lighting and shadows

2. After applying lighting

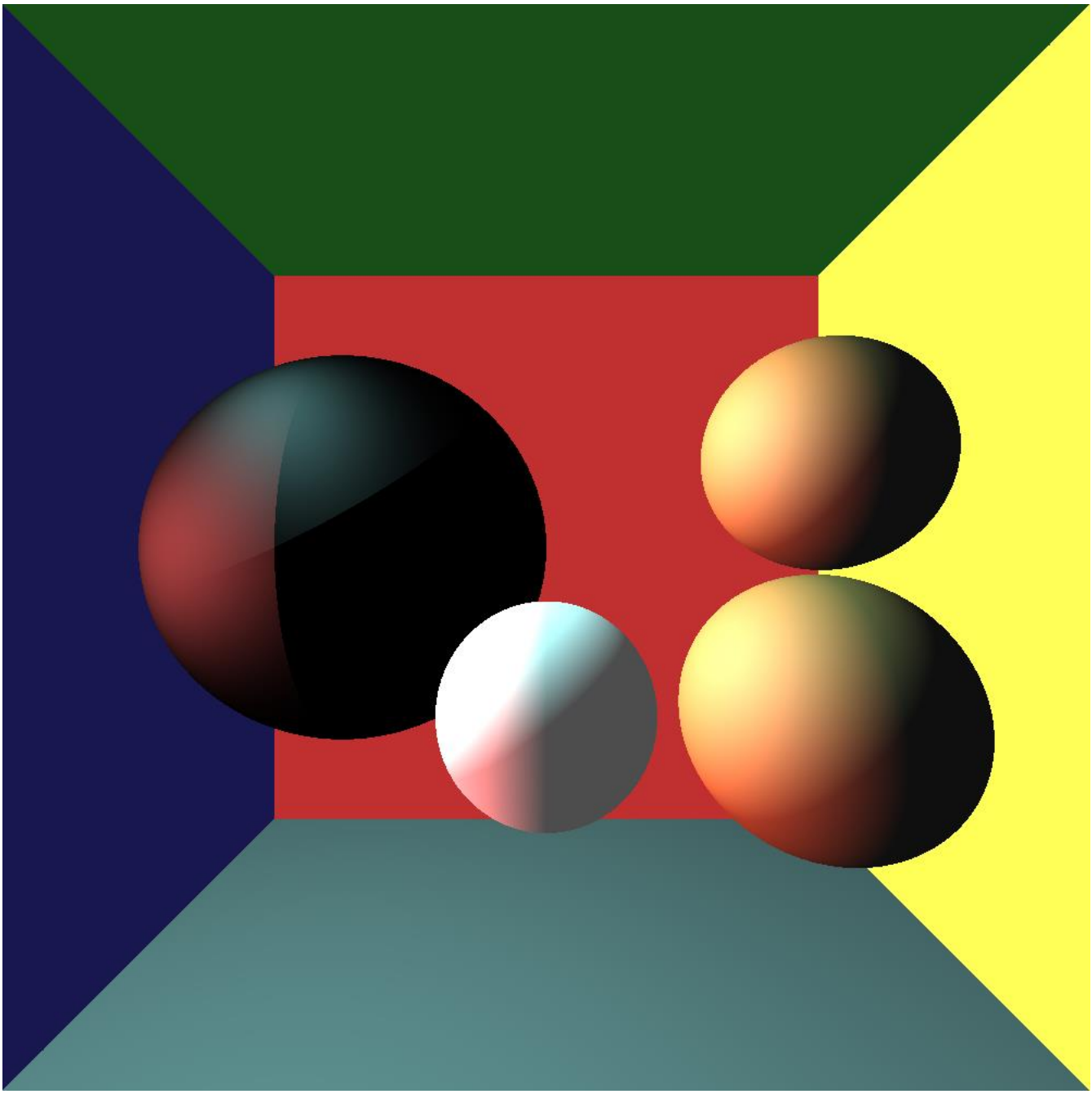


Fig 13. Scene after applying lighting

3. After applying shadows and reflections

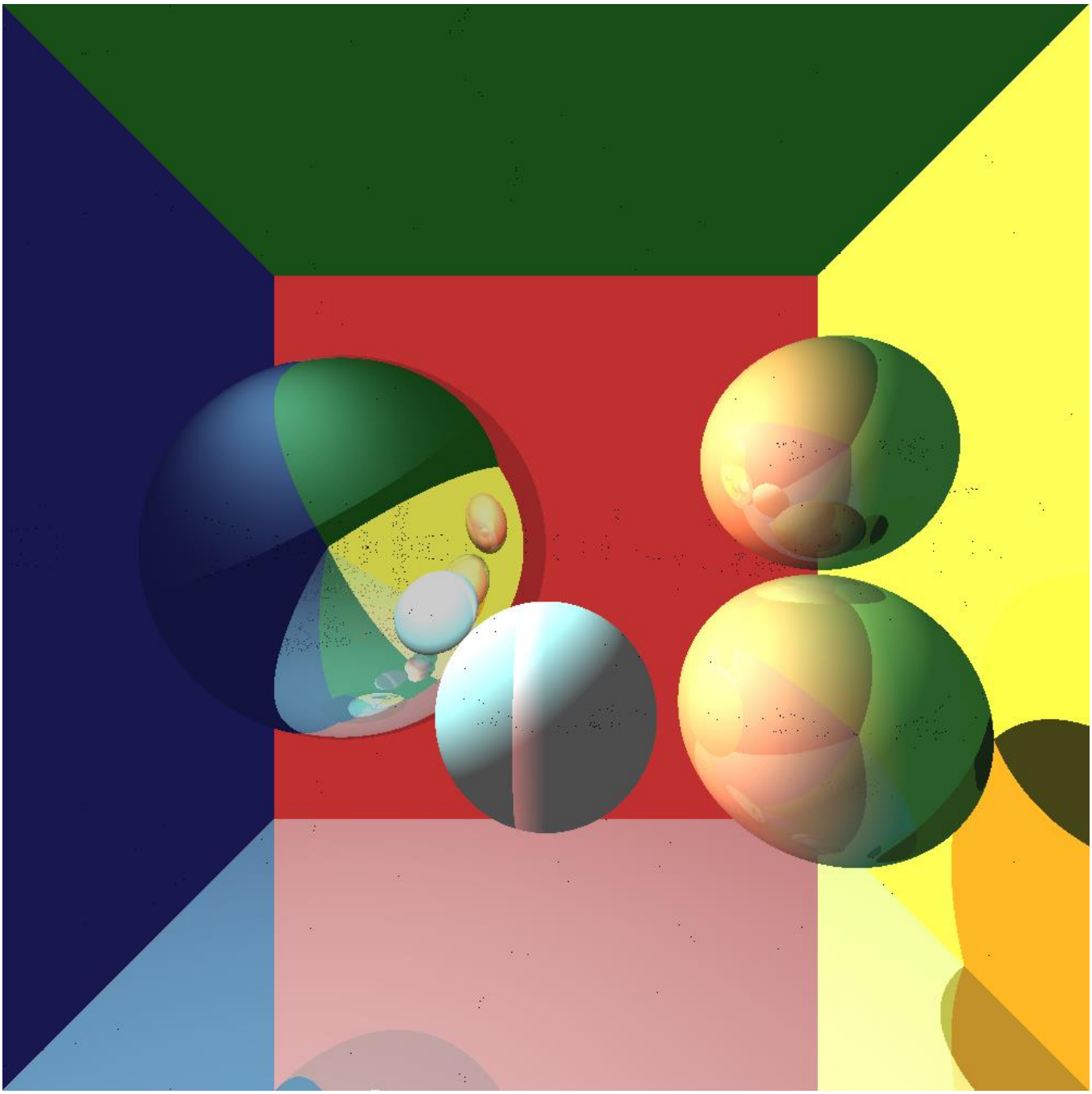


Fig 14. The final rendered image

CHAPTER 7: CONCLUSION

In this project we see how we can generate a realistic image through ray tracing by using backward tracing where we shoot a ray from the camera to the scene and see where it hits and then calculate the colour at that pixel. Ray tracing can be used in non-interactive applications where we can generate images beforehand and then display it when needed, as it takes a lot of time in rendering it is not suitable for interactive applications however through parallelism we can still think of such implementations. In the program developed in this project we can define any scene consisting of plane and spheres through a data file containing its coordinates and RGB components of diffuse, ambient and specular lighting components. The number of recursive reflections depend on the exponential shininess component. Through suitable mathematical equations we can further extend this project to other primitives such as triangles, cylinders etc.

REFERENCES:

[1] WHITTED, T. An improved illumination model for shaded display. Communications of the ACM 32,6 (June 1980).

[2] Cohen, M. F., and Greenberg, D.P. The hemi-cube: A radiosity solution for complex environments. Computer Graphics (July 1985)

[3] SUTHERLAND, I.E., SPROULL, R.F., AND SCHUMACKER, R.A. A characterization of ten hidden surface algorithm. Comput. Surv. (Mar 1974)

[4] HALL, R.A. A methodology for realistic image synthesis. Master's thesis, Cornell University, Ithaca, New York (1983)

[5] Nishita, T., and Nakamae, E., Continuous tone representation of 3D objects taking account of shadows and interreflection, Computer Graphics (July 1985).

[6] Greg Ward, Francis M. Rubinstein, and Robert D. Clear. A Ray Tracing Solution for Diffuse Interreflection. Proc. SIGGRAPH , 1988.