Indian Institute of Technology Bombay

Department of Mechanical Engineering

Project: Ray Tracing Engine

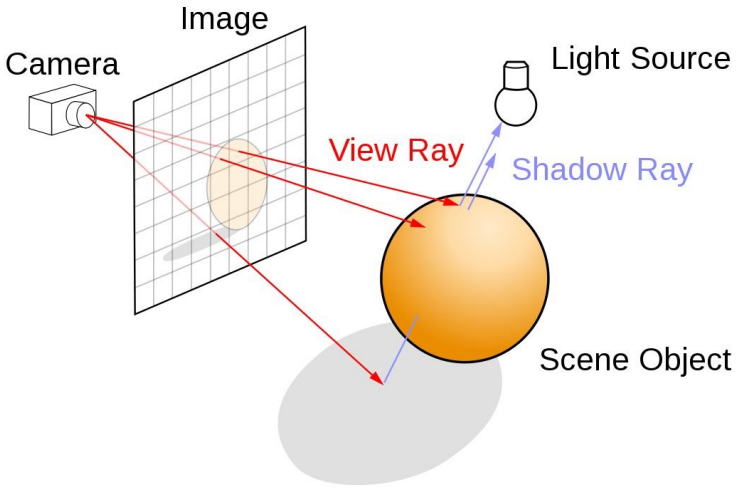
Instructor: Prof. S.S. Pande

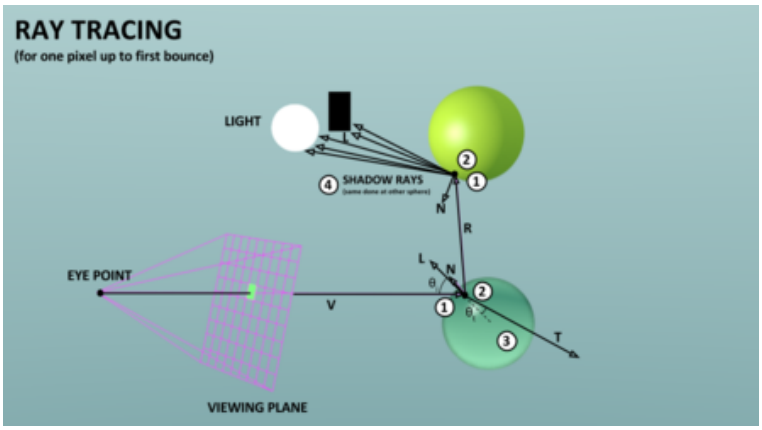
Roll Number	Name
190100129	Prathyush Thunguri
190100118	Anusha Mantha
190100111	Shreyansh Goyal
190100087	Tejaswi Pagadala

Introduction

The project topic is Ray tracing Engine, hence we are giving a brief information about ray tracing.

Ray tracing is a rendering technique that can realistically simulate the lighting of a scene and its objects by rendering physically accurate reflections, refractions, shadows, and indirect lighting. Ray tracing generates computer graphics images by tracing the path of light from the view camera (which determines your view into the scene), through the 2D viewing plane (pixel plane), out into the 3D scene, and back to the light sources. As it traverses the scene, the light may reflect from one object to another (causing reflections), be blocked by objects (causing shadows), or pass through transparent or semi-transparent objects (causing refractions). All of these interactions are combined to produce the final color and illumination of a pixel that is then displayed on the screen. This reverse tracing process of eye/camera to light source is chosen because it is far more efficient than tracing all light rays emitted from light sources in multiple directions.





The ray-tracing algorithm builds an image by extending rays into a scene and bouncing them off surfaces and towards sources of light to approximate the color value of pixels.

Abstract

Problem Statement:

Given information about a 3D object (position, shape , color, size), our task is to render the same image in 2D, given the camera and the light source and also make the rendered image look realistic and convey as much information as possible subject to the constraints.

Deliverables:

1. Synthetic Experiments to understand the working of the chosen environment.
2. The python code corresponding to the algorithm (s) that were chosen
3. Rendered Images

Algorithm

The next few pages Contains the codes of our project

Importing Required Libraries

```
from tqdm import tqdm_notebook as tqdm
import numpy as np
import matplotlib.pyplot as plt
```

Setting Image Pixel Count

```
# Setting the pixel count
w = 1920 # Set the width of the image
h = 1080 # Set the height of the image
```

Defining Math functions

```
def normalize(x):
    x /= np.linalg.norm(x) # np.linalg.norm returns the l2 norm for the matrix
    return x               # This is normalizing with the l2 norm

def intersect_plane(O, D, P, N):
    # Every thing here is a 3x1 vector
    # Using Basic Geometry
    denom = np.dot(D, N)
    if np.abs(denom) < 1e-4:
        return np.inf
# This is because it means that the ray and the plane are almost parallel
#and hence no intersection
    d = np.dot(P - O, N) / denom
    if d < 0:  # Because this is a ray
        return np.inf
    return d
def intersect_triangle(O, D, P, N, p1, p2, p3):
    #check if this lies in the triangle
    denom = np.dot(D, N)
    if np.abs(denom) < 1e-4:
        return np.inf
# This is because it means that the ray and the plane are almost parallel
#and hence no intersection
    d = np.dot(P - O, N) / denom
    if d < 0:  # Because this is a ray
        return np.inf
    I = O + d*D
    # print(O+d*D)
    #Intersection point is known now
    a = np.cross((p1-p2),(p2-p3))
    total_area = np.sqrt(a.dot(a))/2
    a = np.cross((I-p2),(I-p3))
    area_1 = np.sqrt(a.dot(a))/2
    a = np.cross((I-p1),(I-p2))
    area_2 = np.sqrt(a.dot(a))/2
```

```
    a = np.cross((I-p1),(I-p3))
    area_3 = np.sqrt(a.dot(a))/2
    if(abs(total_area - area_1 - area_2 - area_3 )> 1e-8):
     return np.inf
    else: return d

def intersect_sphere(O, D, S, R):
    # Return the distance from O to the intersection of the ray (O, D) with the
    # sphere (S, R), or +inf if there is no intersection.
    # O and S are 3D points, D (direction) is a normalized vector, R is a scalar.
    #Using parameterized form of a ray, solves a quadratic that gives the value
    #both the intersection

    a = np.dot(D, D)
    OS = O - S
    b = 2 * np.dot(D, OS)
    c = np.dot(OS, OS) - R * R
    disc = b * b - 4 * a * c   # Evaluating if roots are real or not
    if disc > 0:
        distSqrt = np.sqrt(disc)
        q = (-b - distSqrt) / 2.0 if b < 0 else (-b + distSqrt) / 2.0
        t0 = q / a
        t1 = c / q
        t0, t1 = min(t0, t1), max(t0, t1)   # Sorts the parameter value
        if t1 >= 0:
            return t1 if t0 < 0 else t0
            #Since the first intersection is needed, we always look for
            # the smaller value
            #This may not be possible when the roots are of opposite sign,
            #hence return larger only when smaller one is negative
    return np.inf
```

Adding Utility functions

```
def intersect(O, D, obj):
# Depending on the obj whether sphere or plane returning the inersecting distance
# of O to intersection
# point on object
    if obj['type'] == 'plane':
        return intersect_plane(O, D, obj['position'], obj['normal'])
    elif obj['type'] == 'sphere':
        return intersect_sphere(O, D, obj['position'], obj['radius'])
    elif obj['type'] == 'triangle':
        return intersect_triangle(O, D, obj['position'],obj['normal'],
                                  obj['p1'],obj['p2'],obj['p3'])

def get_normal(obj, M):
    # Find normal.
    if obj['type'] == 'sphere':
        N = normalize(M - obj['position']) #Normal the sphere is evaluated
        #(based on current position and the center)
```

```
    elif obj['type'] == 'plane':
        N = obj['normal'] # normal of a plane returned from definition
    elif obj['type'] == 'triangle':
        N = obj['normal'] # normal of a plane returned from definition
    return N

def get_color(obj, M):# color of object at point M
    color = obj['color']
    if not hasattr(color, '__len__'): # If there is no color attached,
    #then use the color of M?
        color = color(M)
    return color
```

Adding Ray Tracer

```
def trace_ray(rayO, rayD):
    # Find first point of intersection with the scene.
    t = np.inf
    for i, obj in enumerate(scene):
        t_obj = intersect(rayO, rayD, obj)
        if t_obj < t:
            t, obj_idx = t_obj, i
    # Return None if the ray does not intersect any object.
    if t == np.inf:
        return
    # Find the object.
    obj = scene[obj_idx]
    # Find the point of intersection on the object.
    M = rayO + rayD * t
    # Find properties of the object.
    N = get_normal(obj, M)
    color = get_color(obj, M)
    toL = normalize(L - M)
    toO = normalize(O - M)
    # Start computing the color.
    col_ray = ambient
    # Lambert shading
    col_ray += obj.get('diffuse_c', diffuse_c) * max(np.dot(N, toL), 0) * color
    # Blinn-Phong shading
    col_ray += obj.get('specular_c', specular_c) * max(np.dot(N, normalize(toL
                                        + toO)), 0)** specular_k * color_light
    return obj, M, N, col_ray
```

Double-click (or enter) to edit

Adding Object defining Functions

```
def add_sphere(position, radius, color):
    return dict(type='sphere', position=np.array(position),
```

```
        radius=np.array(radius), color=np.array(color), reflection=.5)
 # This function returns a dictionary with sphere of given radius, center and color

def add_plane(position, normal):
   # This function returns a dictionary of a defined plane
    return dict(type='plane', position=np.array(position),
        normal=np.array(normal),
        color=lambda M: (color_plane0
            if (int(M[0] * 2) % 2) == (int(M[2] * 2) % 2) else color_plane1),
        diffuse_c=.75, specular_c=.5, reflection=.25)

def add_triangle(p1,p2,p3,c):
  p1 = np.array(p1)
  p2 = np.array(p2)
  p3 = np.array(p3)
  norm = np.cross((p1-p2),(p2-p3))
  n = normalize(norm)
  return dict(type = 'triangle',position = np.array(p1),p1 = np.array(p1),
              color = np.array(c), p2 = np.array(p2), p3 = np.array(p3),
              normal = np.array(n),diffuse_c=.75, pecular_c=.5, reflection=.25)
```

Defining Objects

```
color_plane0 = 1. * np.ones(3)
color_plane1 = 0. * np.ones(3)
scene = [add_sphere([.75, .1, 1.5], .8, [0., 0., 1.]),
         add_sphere([-.75, .1, 2.25], .5, [0., 1., 0.]),
         add_sphere([-2.75, .1, 3.5], .8, [1., 0., 0.]),
         add_plane([0., -.5, 0.], [0., 1., 0.]),
         add_triangle([0., 1., 1.],[1., 0., 1.],[0., 0., 1.],[0., 1., 1.]),
         add_triangle([0., 1., 1.],[1., 0., 1.],[-0.75, 0.45, 2.],[1., 0., 1.]),
         add_triangle([0.4, 0.5, 1.],[1., 0., 1.],[-0.75, 0.45, 2.],[1., 1., 0.])
    ]
scene1 = [
         add_triangle([0., 1., 1.],[1., 0., 1.],[0., 0., 1.],[0., 1., 1.]),
         add_plane([0., -.5, 0.], [0., 1., 0.])
    ]
```

Defining Light position and color.

```
L = np.array([5., 5.,-10.])
color_light = np.ones(3)
```

Default light and material parameters.

```
ambient = 0.05
diffuse_c = 1.
specular_c = 1.
```

```
specular_k = 50

depth_max = 10  # Maximum number of light reflections.
col = np.zeros(3)  # Current color.
O = np.array([0., 0.35, -1.])  # Camera.
Q = np.array([0., 0., 0.])  # Camera pointing to.
img = np.zeros((h, w, 3))
r = float(w) / h
# Screen coordinates: x0, y0, x1, y1.
S = (-1., -1. / r + .25, 1., 1. / r + .25)
```

Looping through all pixels

```
for i, x in tqdm(enumerate(np.linspace(S[0], S[2], w)),total = w):
  for j, y in enumerate(np.linspace(S[1], S[3], h)):
        col[:] = 0    # Set the color of the pixel to black
        Q[:2] = (x, y)    # Point the camera to the required point
        D = normalize(Q - O)
        # DCS of the ray joining camera and point of interest
        depth = 0 # First, number of reflections is zero
        rayO, rayD = O, D  # Ray definition from camera to point of interest
        reflection = 1.
        # Loop through initial and secondary rays.""
        while depth < depth_max:   # Implement recursive
            traced = trace_ray(rayO, rayD)
# Returns object, point of intersection, normal at the point of intersection
            #and color of ray
            if not traced: # If traced is null
                break
            obj, M, N, col_ray = traced
            rayO, rayD = M + N * .0001, normalize(rayD - 2 * np.dot(rayD, N)*N)
             # Now the ray is reflected and initialized to OD for further
            depth += 1 # Since another reflection has occured
            col += reflection * col_ray
            reflection *= obj.get('reflection', 1.)
            # Taking into account the reflection of objects
        img[h - j - 1, i, :] = np.clip(col, 0, 1)
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:1: TqdmDeprecationWarnir
Please use `tqdm.notebook.tqdm` instead of `tqdm.tqdm_notebook`
  """Entry point for launching an IPython kernel.
```

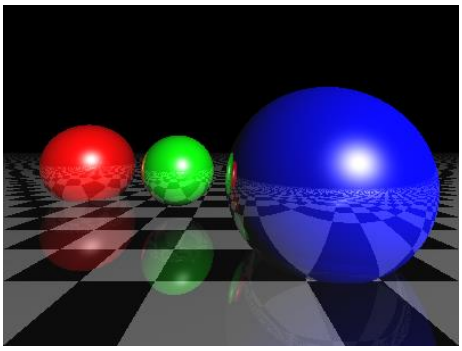
```
100%                                                1920/1920 [37:04<00:00, 1.02s/it]
```

```
plt.imsave('1980x1080_depth-10.png', img)
```

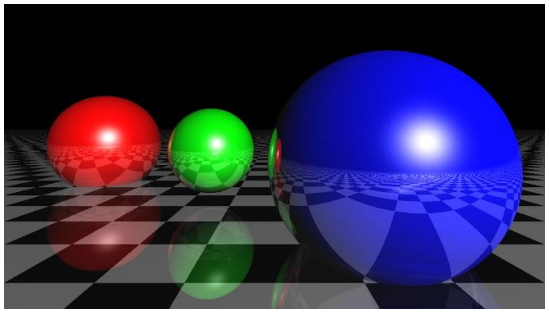
Results:

Resolution

400 x 300

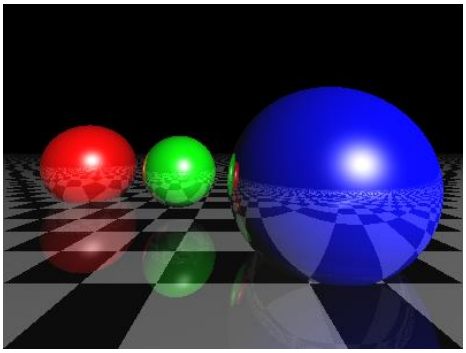


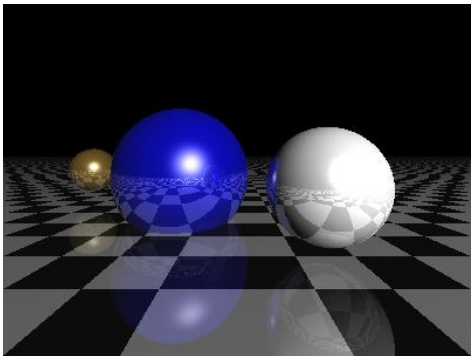
1920 x 1080



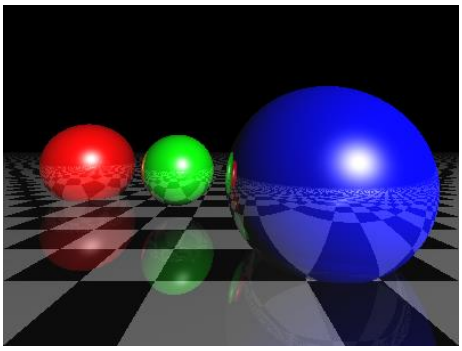
Sphere shape and color:

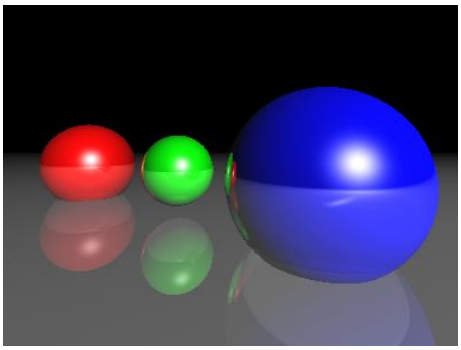
Original:





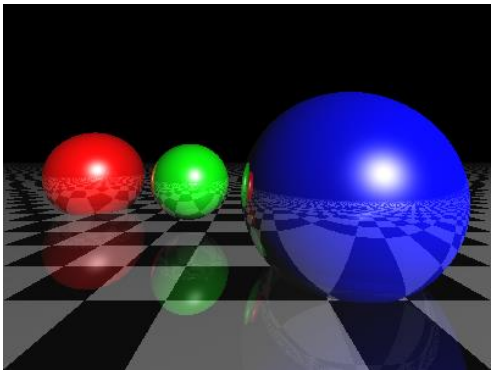
Plane Color

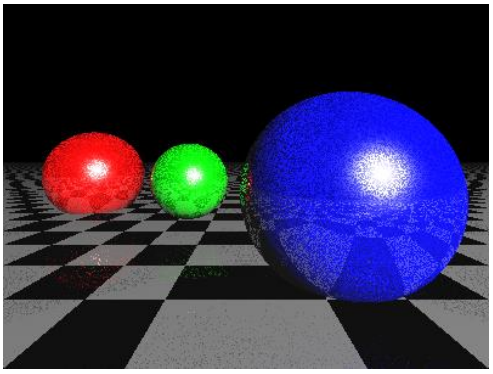




Addition of Diffusivity

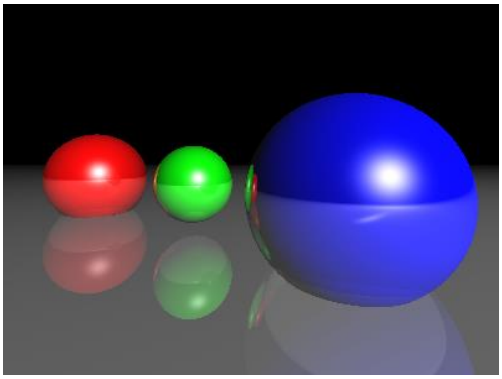
Without Diffusivity



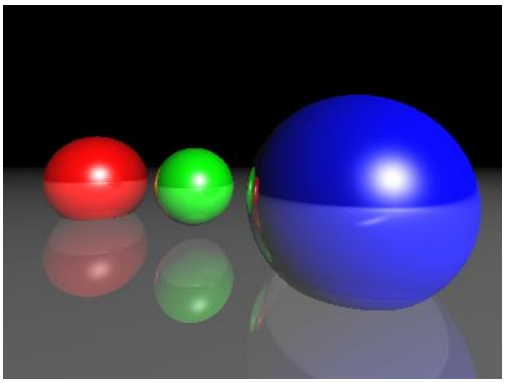


Effect of changing intersection accuracy

Moderately low:

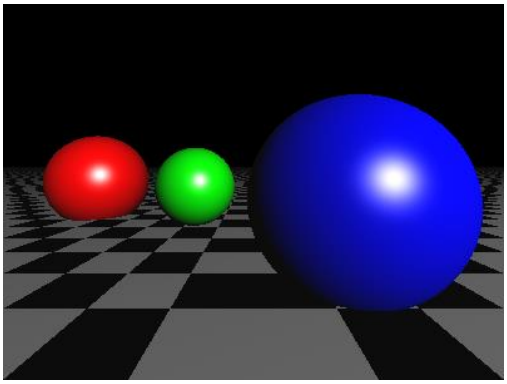


Extremely high:

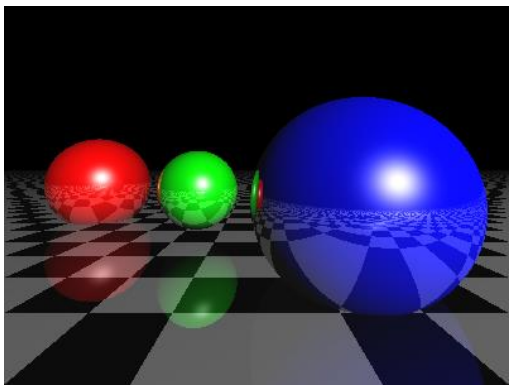


Depth

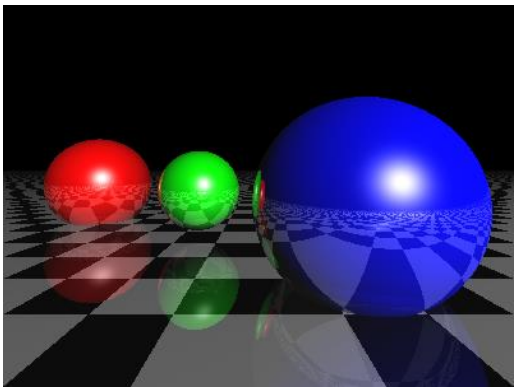
Depth = 1



Depth = 2

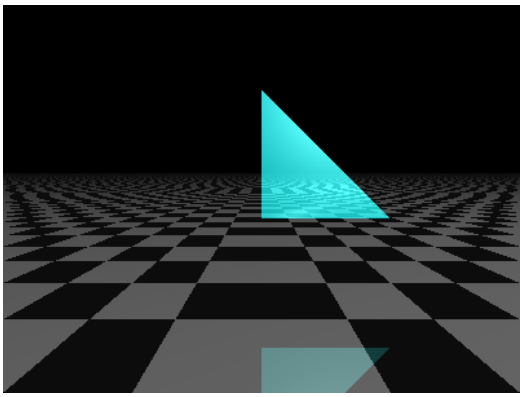


Depth = 15

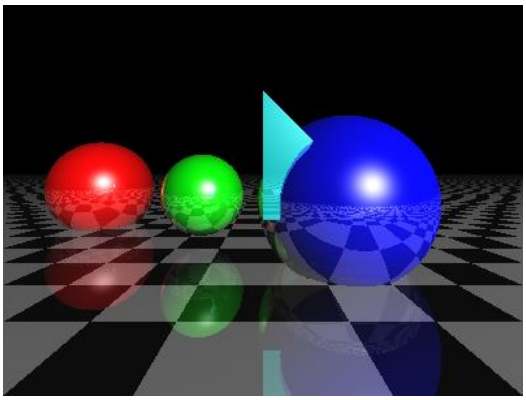


Adding Triangle:

Without intersection:

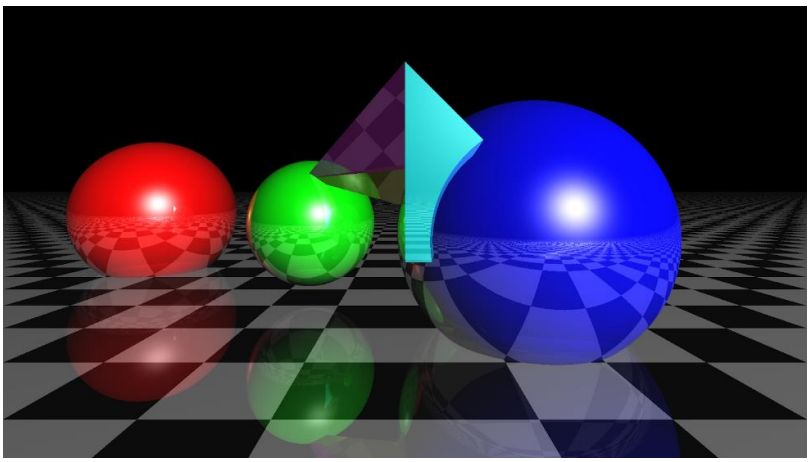


With intersection:



Everything combined:

1920x1080 at depth = 10



Conclusions:

1. When **diffusivity** is not added, the image's grain size is too large. Therefore, a smooth image is not obtained.

2. The more depth, the more real the image looks. It can be seen that at depth 2, a reflection of the floor is seen on the objects, and objects are seen on the floor. At depth 15, the reflection of reflection of objects on the floor can be seen on the objects.

3. Computation time increases

   (a) Increase the number of pixels

   (b) Increase the depth

   (c) As we add more shapes

4. This algorithm enables us to vary the shape, size, position, and color of the objects and the floor.