
**RYAN INTERNATIONAL SCHOOL CBSE, SURAT**

Academic year 2020-2021

# Raytracing in Python

*A project report*

Name: Urjasvi Suthar

Class: XII-C

Roll no.: 38

# INDEX

# RYAN INTERNATIONAL SCHOOL CBSE, SURAT


## BONAFIDE CERTIFICATE

This is to certify that the project report entitled “Raytracing in python” submitted by “Urjasvi Suthar” project considered as the part of the practical exam of AISSCE conducted by CBSE is a bonafide record of the work carried out under our guidance and supervision at **RYAN INTERNATIONAL SCHOOL CBSE, SURAT.**

This project report is evaluated by us on _____ .

INTERNAL EXAMINER EXTERNAL EXAMINER

HEAD OF THE INSTITUTION

# Acknowledgment

I would like to thank my subject teacher for giving me the opportunity to select the topic of my interest for this project. I've enjoyed making this project as it is based on the topic of my interest and group work.

I have put in hard work and dedication to make sure that this project will be informative and enjoyable.

# Overview of Raytracing

So, what is raytracing? Raytracing (CGI) is a rendering technique used to render virtual scene with realistic lighting effect. It is capable of simulating optical effects such as reflection, refraction and dispersion (Chromatic aberration). It is used in movies and tv shows as VFX. Video games are starting to get raytracing support by using beefy GPUs that have hardware accelerated raytracing support.

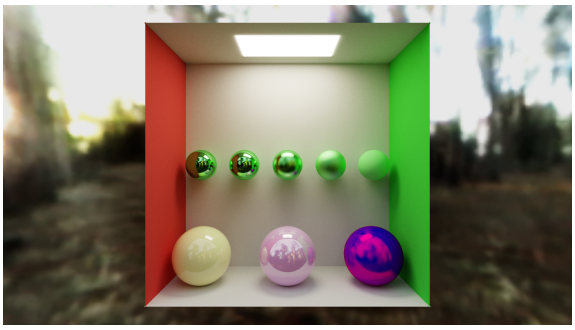
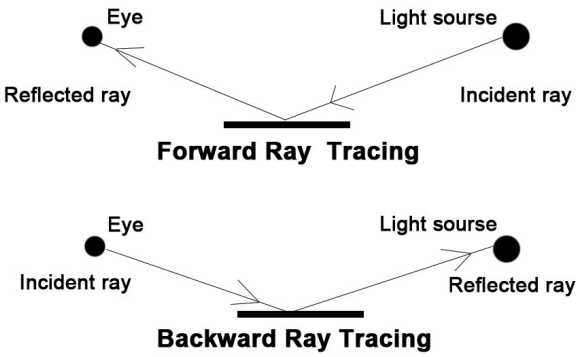


Image showcasing reflection

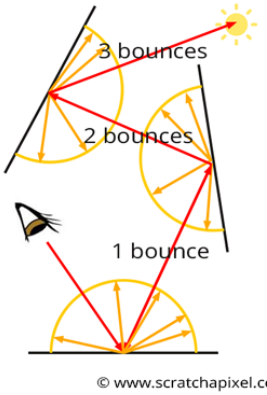
Image showcasing reflection, refraction and color bleeding.

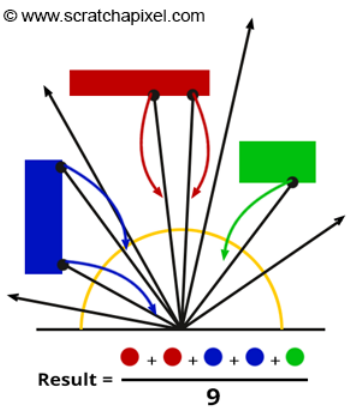
Image showcasing glass material with roughness.

In real-life, when light is turned on, its emit ‘light rays’ which bounce around and then finally enter our eye which make us ‘see’ things. Raytracing is similar to this, instead of emitting from the light, we shoot rays from our ‘eye’. The reason being that if we shoot ray from light, then it have large chances of missing our eye. This would be really inefficient. Raytracer in which rays are emitted from light source is called ‘Forward raytracer’, and raytracer in which rays are emitted from eye is called ‘Backward raytracer’.




So how does backward raytracer work? Basically, we shoots ray from across our screen’s pixels to some distance, we check for object and ray intersection. If it intersect, we emit another ray from intersection point and we keep repeating it until repeating reaches N times (light bounce limit) or until it hit light sources. For each hit, we add material’s color and light intensity at that point. If it doesn’t intersect then we return background color.



## Requirements

Raytracing is computing heavy, so while this python program can run on any platform/hardware supporting python, better cpu means faster rendering time.

## Module Used

- math
- random
- time
- util

# Source code

```
--------------------------------------------------------------------------------
Main.py
--------------------------------------------------------------------------------

    # STD
    import math
    import time
    from random import random
    # Util
    import util.color
    from util.ray import ray
    from util.vector import vec3

    from Sphere import sphere
    from Hittable import hit_record
    from Material import material, lambertian, metal

    def shoot_ray(r, object_list, depth=0):
      rec = hit_record()
      for obj in object_list:
      rec = obj.hit(r, 0.01, rec.t, rec)
      if rec.hitted == True: break

      if rec.hitted == True:
      material = rec.material
      m_rec = material.scatter(r, rec)
      if depth < 50 and m_rec.bScattered == True:
             return m_rec.attenuation * shoot_ray( m_rec.scattered_ray, object_list,
depth + 1)
      else:
      return vec3(0.0, 0.0, 0.0)

      dir = r.direction()
      t = (dir.y + 1.0) * 0.5
      return vec3.interpolate(vec3(0.3, 0.5, 0.8), vec3(1.0, 1.0, 1.0), t)

    def renderer():
      aspect_ratio = 16/9
      image_width = 720
      image_height = int(image_width/aspect_ratio)
      num_samples = 50

      viewport_height = 2.0
      viewport_width = aspect_ratio * viewport_height
      focal_length = 1.0

      origin = vec3(0, 0, 0)
```

```
      horizontal = vec3(viewport_width, 0, 0)
      vertical = vec3(0, viewport_height, 0)
      lower_left_corner = origin - horizontal/2 - vertical/2 - vec3(0, 0, focal_length)

      object_list =[
            sphere(vec3(-0.5, -0.5, -1.5), 0.5, lambertian(vec3(1.0, 0.8, 1.0))),
      sphere(vec3(0.5, -0.5, -1.5), 0.5, metal(vec3(1.0, 1.0, 1.0), 0.0)),
      sphere(vec3(0.5, 0.5, -1.5), 0.5, lambertian(vec3(0.8, 1.0, 1.0))),
      sphere(vec3(-0.5, 0.5, -1.5), 0.5, metal(vec3(1.0, 1.0, 1.0), 0.3))
      ]

      file = open('output.ppm', 'w')

      file.write("P3\n{0:d} {1:d}\n255\n".format(image_width, image_height))

      for y in range(image_height-1, -1, -1):
            for x in range(image_width):
                  color = vec3(0.0, 0.0, 0.0)
                  for s in range(num_samples) :
                        u = (x + random()) / float(image_width)
                        v = (y + random()) / float(image_height)
                        r = ray(origin, lower_left_corner + u*horizontal + v*vertical
- origin)
                        color = color + shoot_ray(r, object_list)
                  color = color / float(num_samples)
                  util.color.write_color(file, color)

      file.close()

    start = time.time()
    renderer()
    end = time.time()
    print(str(end - start)+" secs took to render")
```

---

---

Sphere.py

---

```
    # Std
    from math import sqrt
    # Util
    from util.ray import ray
    from util.vector import vec3

    import Hittable
    from Material import material, lambertian

    class sphere(Hittable.hittable):
      def __init__(self, pos, radius, mesh_material=lambertian(vec3(0.5, 0.5,
0.5))):
            self.pos = pos
            self.radius = radius
```

```
            self.mesh_material = mesh_material

      def hit(self, ray, t_min, t_max, hit_rec):
            oc = ray.origin() - self.pos
            ray_dir = ray.direction()
            a = vec3.dot(ray_dir, ray_dir)
            b = vec3.dot(oc, ray_dir)
            c = vec3.dot(oc, oc) - self.radius ** 2

            D = b**2 - a*c

            if D > 0.0 :
                  temp = ( -b - sqrt( D ) ) / a
                  if temp > t_min and temp < t_max :
                        hit_rec.t = temp
                        hit_rec.p = ray.point_at(hit_rec.t)
                        hit_rec.normal = vec3.normalize(hit_rec.p - self.pos)
                        hit_rec.material = self.mesh_material
                        hit_rec.hitted = True
                        return hit_rec

                  temp = ( -b + sqrt( D ) ) / a
                  if temp > t_min and temp < t_max :
                        hit_rec.t = temp
                        hit_rec.p = ray.point_at(hit_rec.t)
                        hit_rec.normal = vec3.normalize(hit_rec.p - self.pos)
                        hit_rec.hitted = True
                        hit_rec.material = self.mesh_material
                        return hit_rec

            hit_rec.hitted = False
            return hit_rec
```

---

Material.py

---

```
    # Std
    from abc import ABC
    # Util
    from util.ray import ray
    from util.vector import vec3

    class material_record :
      def __init__(self) :
            self.bScattered = False
            self.scattered_ray = ray(vec3(0.0, 0.0, 0.0), vec3(0.0, 0.0, 0.0))
            self.attenuation = vec3(0.0, 0.0, 0.0)

    # material interface
    class material(ABC) :
```

```
      def scatter(self, r, hit_rec) :
            raise NotImplementedError("Subclass must implement scatter method")

    class lambertian(material) :
      def __init__(self, albedo) :
            self.albedo = albedo

      def scatter(self, r, hit_rec) :
            target = hit_rec.p + hit_rec.normal + vec3.get_random_in_sphere()

            m_record = material_record()
            m_record.scattered_ray = ray(hit_rec.p, target - hit_rec.p)
            m_record.attenuation = self.albedo
            m_record.bScattered = True

            return m_record

    class metal(material) :
      def __init__(self, albedo, fuzz) :
            self.albedo = albedo
            self.fuzz = min(fuzz, 1.0)

      def scatter(self, r, hit_rec) :
            reflected = vec3.reflect(vec3.normalize(r.direction()),
hit_rec.normal)
            direction = reflected + self.fuzz * vec3.get_random_in_sphere()

            m_record = material_record()
            m_record.scattered_ray = ray(hit_rec.p, direction)
            m_record.attenuation = self.albedo
            m_record.bScattered = vec3.dot( m_record.scattered_ray.direction(),
hit_rec.normal ) > 0

            return m_record
```

---

Hittable.py

---

```
    # Std
    from abc import ABC
    # Util
    from util.ray import ray
    from util.vector import vec3

    from Material import material

    class hit_record():
    def __init__(self):
    self.p = vec3(0.0, 0.0, 0.0)
    self.normal = vec3(0.0, 0.0, 0.0)
```

```
    self.t = 100000000.0
    self.hitted = False
    self.material = material()

    class hittable(ABC):
    def hit(self, r, t_min, t_max, hit_rec):
    raise NotImplementedError('Subclass must implement hit method')
```

---

color.py

```
    def write_color(file, color):
      file.write("{0:d} {1:d} {2:d}\n".format(
      int(color.x*255), int(color.y*255), int(color.z*255)
    ))
```

---

ray.py

```
    class ray :
      def __init__(self, origin, direction) :
            self.A = origin
            self.B = direction
      def origin(self):
            return self.A
      def direction(self):
            return self.B
      def point_at(self, t):
            return self.A + self.B * t
```

---

vector.py

```
    from math import sqrt
    from random import random

    class vec3 :
      def __init__(self, x, y, z) :
```

```
        self.x = x
        self.y = y
        self.z = z

@staticmethod
def length(v) :
  return sqrt(v.x ** 2 + v.y ** 2 + v.z ** 2)

@staticmethod
def normalize(v) :
  v_len = vec3.length(v)
  if v_len == 0 :
        raise ZeroDivisionError
  unit_v = vec3(v.x, v.y, v.z) / v_len
  return unit_v

@staticmethod
def interpolate(v1, v2, t) :
  return v1 * t + v2 * (1.0 - t)

@staticmethod
def dot(v1, v2) :
  return v1.x * v2.x + v1.y * v2.y + v1.z * v2.z

@staticmethod
def reflect(v, n) :
  return v - 2 * vec3.dot(v, n) * n

@staticmethod
def get_random_in_sphere() :
  while True:
        v = vec3(random(), random(), random()) * 2.0 - 1.0
        if vec3.length(v) < 1.0 :
              return v

def __add__(self, other) :
  if isinstance(other, self.__class__) :
        return vec3( self.x + other.x, self.y + other.y, self.z + other.z )
  else :
        return vec3( self.x + other, self.y + other, self.z + other )

def __sub__(self, other) :
  if isinstance(other, self.__class__) :
        return vec3( self.x - other.x, self.y - other.y, self.z - other.z )
  else :
        return vec3( self.x - other, self.y - other, self.z - other )
def __mul__(self, other) :
  if isinstance(other, self.__class__) :
        return vec3( self.x * other.x, self.y * other.y, self.z * other.z )
  else :
        return vec3( self.x * other, self.y * other, self.z * other )
def __truediv__(self, other) :
  if isinstance(other, self.__class__) :
```

```
            return vec3( self.x / other.x, self.y / other.y, self.z / other.z )
        else :
            return vec3( self.x / other, self.y / other, self.z / other )

    __radd__ = __add__
    __rsub__ = __sub__
    __rmul__ = __mul__
```

---

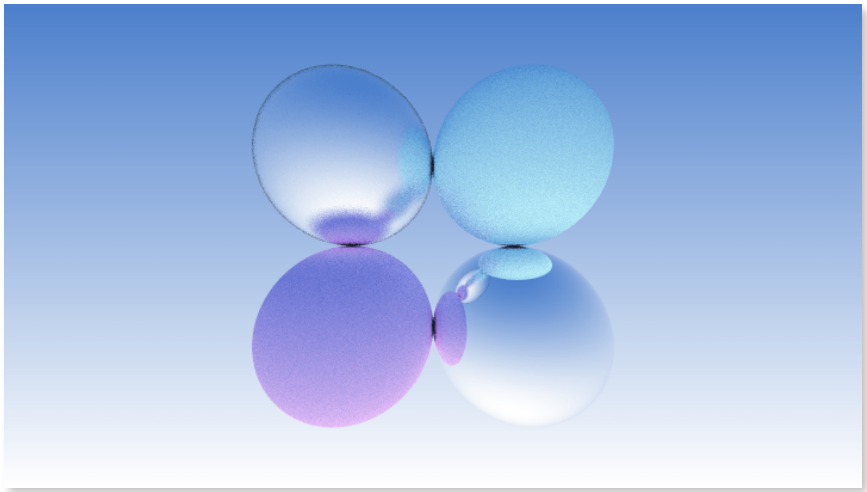
## Conclusion

If you were to run the program with 720p resolution with 50 samples on 2.2 GHz Quad-Core Intel Core i7, it would take you ~9 mins for render and that means it super slow.

To improve render time you can do following things:

- Python language optimization.
- Use manual managed memory programming languages like Rust and C/C++.
- Use acceleration structure like AABB.
- Use all CPU threads.
- Use GPU
  - Software rendering like GPGPU, Cuda.
  - Hardware accelerated raytracing.

Output (First sphere outline can be mitigated with more samples):



Code is available on github: https://github.com/BlackGoku36/PythonRaytracer

Download it and then run ‘Main.py’, it will take time to render and the image will appear in the same directory in ‘.ppm’ format. If your platform doesn’t support ‘.ppm’ format, you can download ‘.ppm viewer’ online.

## Bibliography

Peter Shirley, creating raytracer. In: https://raytracing.github.io/books/RayTracingInOneWeekend.html#overview

Snowapril, python translation. In: https://github.com/Snowapril/Raytracing-In-A-Weekend-Python