Radar Interferometer Ray-Tracing Modeling

Neha Bagde

May 13th, 2022

1 Introduction

The Plume Surface Interactions (PSI) group spearheaded by Professors Villafañe, Rovey and Elliot looks to study the aerodynamic and granular dynamics phenomena that occurs during the propulsive landing of a spacecraft of a planetary surface. This topic is especially relevant for landings on the Moon and Mars, as well as other celestial bodies in outer space.

The expected results of this project would include creating an experimental PSI database in Martian and Lunar environments to further validate PSI modeling studies at NASA or other institutions. Another goal is to develop new flight instruments that are applicable to Lunar and Martian missions. It would also include creating new and innovative diagnostic techniques for future PSI ground experiments.

One critical component of the PSI group is the research that is focused toward Radar Interferometer Ray tracing Modeling. The radar interferometer emits a beam of millimeter waves in a 15° cone. However, based on previous experimental observations, the secondary reflections of the beam on nearby surfaces can cause inaccuracies in the measurements.

The goal of this research work is to develop a ray-tracing modeling tool that is able to predict the location and intensity of the reflections for a particular experimental configuration.

2 Phase 1 Objectives

The main objectives of the first phase of this project included understanding the basics of ray tracing through introductory classes such as CS 419 offered by the University of Illinois, learning more about radars, radar interferometry, and their applications as they pertain to this project, and lastly, building familiarity with triangular meshes.

The CS 419 course introduced the basic ray tracing method. The method described how a camera shoots a ray through a screen, towards the object. From that ray, a second ray is produced and pointed towards the light source. If the second ray points at the light source without intersecting with the object, the first ray is not in the shadow. If the shadow ray does intersect, the first ray is in shadow. Fig.1 visualizes this processes. The second blue ray in Fig.1 points directly toward the light source without intersecting with the object, hence proving that the first blue ray is not in the shadow. On the other hand, the second green ray points towards the light source while intersecting with the object, indicating that the first green ray is in shadow.

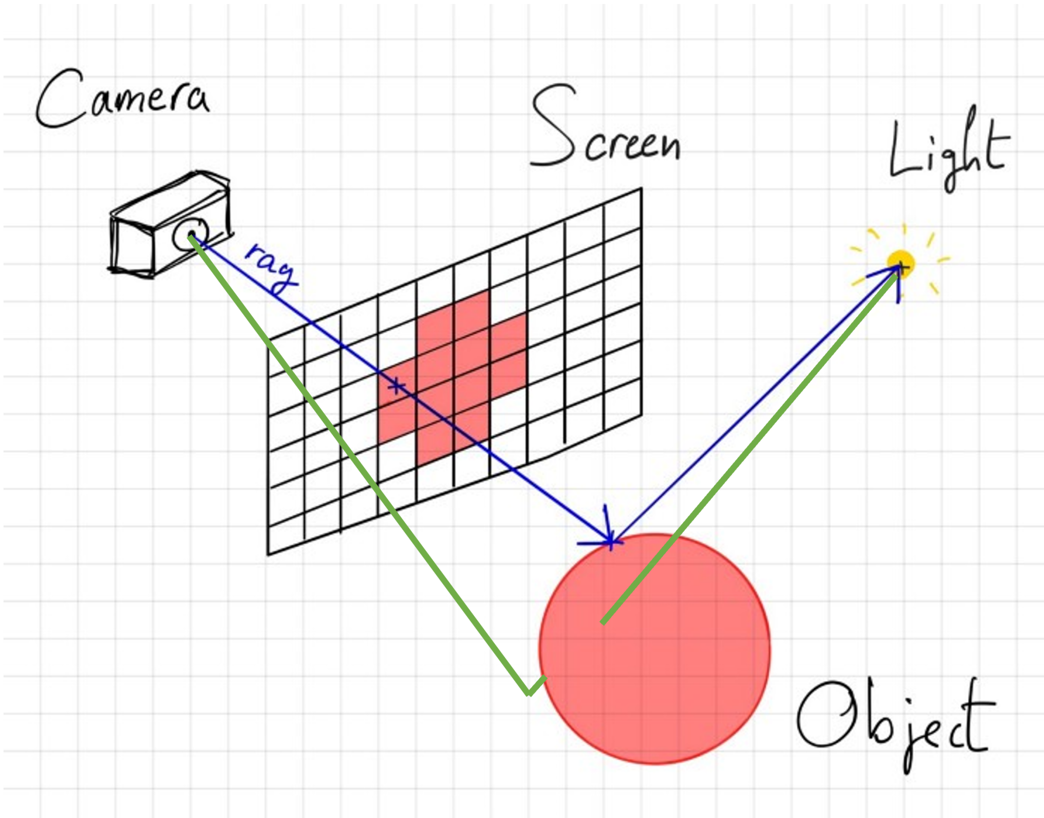

Figure 1: Ray Tracing Method Diagram

The CS 419 course also introduced the Phong Reflection Model. This model assumes light off an object is reflected in three ways: ambient, diffuse, and specular. In ambient reflection, all surfaces and orientations are illuminated equally. In diffuse reflection, shading is produced by illuminating dull and smooth sections of an object. In specular reflection, illumination is produced based on the position and intensity of the light source. These three components are summed to calculate the overall illumination of an object, as shown in Eqn.1.

$$I_p = k_a i_a + \sum_{m \epsilon \text{lights}} (k_d (L_m \cdot N)_{im_d} + k_s (R \cdot V)^{\alpha}_{im_s} \quad (1)$$

where,

k_a: ambient reflection constant
k_d: diffuse reflection constant
k_s: specular reflection constant
α: shininess constant
i_a: ambient intensity constant
i_d: diffuse intensity constant

i_s: specular intensity constant
lights: all light sources in scene
L_m: direction vector from point on object toward light source
N: normal at the specific point on object
R_m: direction that perfectly reflected ray of light would take from the specific point on the object
V: direction pointing towards the camera

Using the information learned about ray tracing and the Phong Reflection Model, I was able to generate a ray-traced image of a phong-illuminated sphere, as shown in Fig.2. The code used to produce Fig.2 is included in the Appendix as well. The primary functions used in this code are the "sphere-intersect" and "nearest-intersect" functions.

The "sphere-intersect" function accepts a ray origin, ray direction, a sphere center point, and sphere radius. The function, then uses the quadratic equation to calculate the determinant and minimum distance if an intersection between a ray and the sphere occurs. If the determinant, denoted as "delta" in the code, is negative, there is no intersection between the ray and sphere. If it is exactly zero, the ray is tangent to the sphere and still not intersecting. If the determinant is positive, there are two intersection points between the ray and the sphere. We only want the first intersection point, which will be associated with the minimum distance.

The "nearest-intersect" function is used when there are multiple objects in the scene. In the event a ray makes an intersection with multiple objects, the function identifies the object closest to the ray and uses the intersection point associated with that object. Therefore, this function aids in correctly process the placement and orientation of objects in the scene.

After defining these primary functions, the image is then generated by parsing through each screen pixel, creating a ray associated with each pixel, and identifying intersection with the object. Once the function has intersection points, the Phong Reflection model is applied by assigning the appropriate illumination based on pre-defined object specifications and the intensity and position of the light source. It should be noted that in the code provided, the Blinn-Phong Reflection model was applied instead to make the code less computationally intensive. The main different between the Phong Reflection model and Blinn-Phong Reflection model is the use of a "half-way vector," H, defined in Eqn.2. In the Phong illumination equation, (R $\cdot$ V) can be replaced with (H $\cdot$ N)

$$H = \frac{L + V}{||L + V||} \tag{2}$$

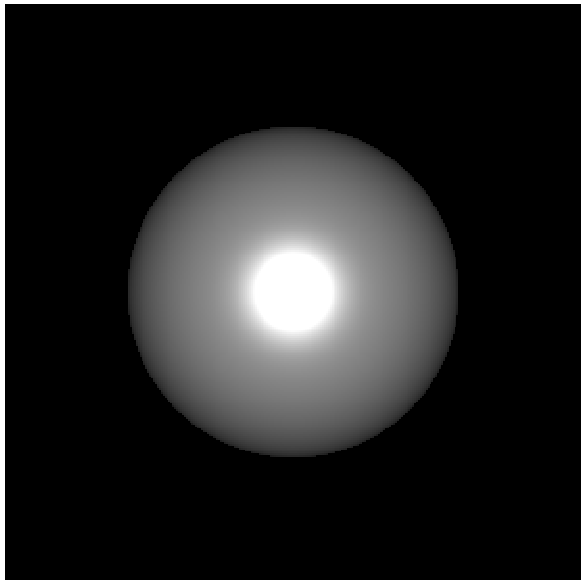

Figure 2: Ray-Traced Sphere

To verify the implementation of the code, a range-power diagram (Fig.3) was produced, highlighting how the power decreased as the distance from the center point increased.

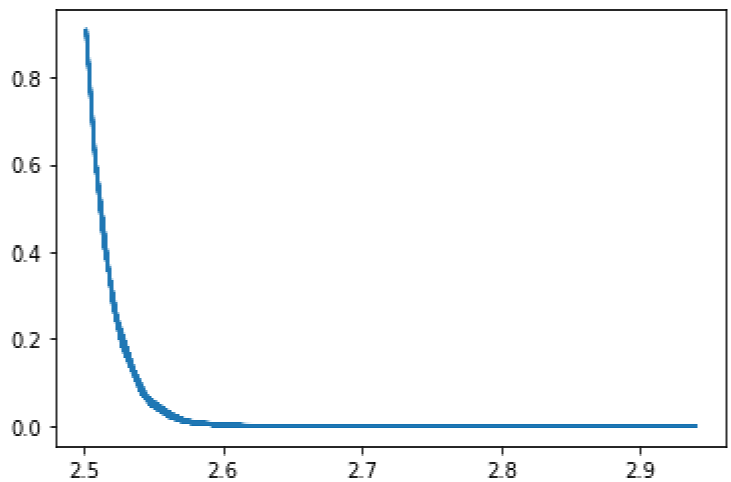

Figure 3: Range-Power Diagram of Sphere

Additionally, using concept covered in Texas Instruments paper, "High-Accuracy Distance Measurement using Millimeter-Wave Radar [3], a radar phase-shift diagram was produced as shown in Fig.4. The phase shift was calculated assuming a frequency of 60 GHz and Eqn.3 below.

$$\Delta\phi = \frac{2\pi f \Delta d}{c} \tag{3}$$

To perform ray-tracing on more complex objects, I learned how to generate and use triangular meshes in Trimesh, which is a Python library built to accomplish this. Trimesh has

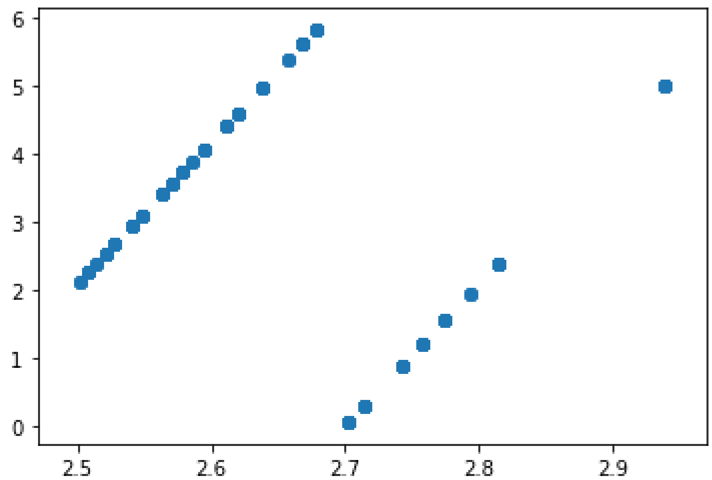

Figure 4: Radar Phase Shift for Sphere

a variety of commands and functions but the one I found most useful was "mesh.ray.intersect-locations". To use this function, you first have to load an STL file and generate a mesh. The function then identifies the mesh and accepts an array of pre-defined ray origins and ray directions to produce a list of intersection locations between the ray and mesh, index rays associated with the locations, and the index triangles the locations are on. Fig.5 aids in visualizing this process.

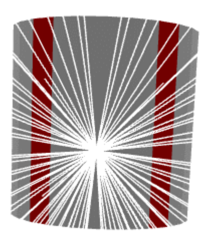

Figure 5: Visual Representation of "mesh.ray.intersect-locations"

In Fig.5 above, the white lines represent rays pointed toward the cylinder mesh. The red portions indicate that those mesh facets have intersections on them. The other rays do not intersect with the object at all and are ignored in the output of the function.

Moving forward with this, I was able to generate a ray-traced image of a cylinder mesh (Fig.6). The code used to produce Fig.6 is provided in the Appendix. The primary function used in this code is "triangle-intersect".

The "triangle-intersect" function accepts a mesh list, ray origins, and ray directions and outputs intersection locations, normals at those intersections, and the index of the intersecting mesh. The mesh list includes all the meshes present in the scene. The ray origins and ray directions are pre-defined based on the number of pixels in the screen.

It is important to note that this code does not include a "nearest-object" function seen in the code for a sphere. The "nearest-object" function looped through each ray origin and ray direction, accepting one combination at a time. This code is structured differently because the "mesh.ray.intersect-locations" function only accepts arrays. Therefore, intersections associated with all meshes have to be calculated first and stored in "active" arrays. Then, a mask is applied to identify all instances where the distances from the values stored in the locations output array are greater than the distances from the values in the corresponding "active" array. If the distance in the "active" array is greater, the previous distance in locations array stays. However, if the new distance in the "active" array is less, then it replaces the existing value in the locations array. The purpose behind this is to obtain all the minimum distances between the ray origins and intersection locations and the normals and meshes associated with those respective intersection locations.

After defining the "triangle-intersect" function, the ray-traced cylinder mesh was generated the same way as the sphere.

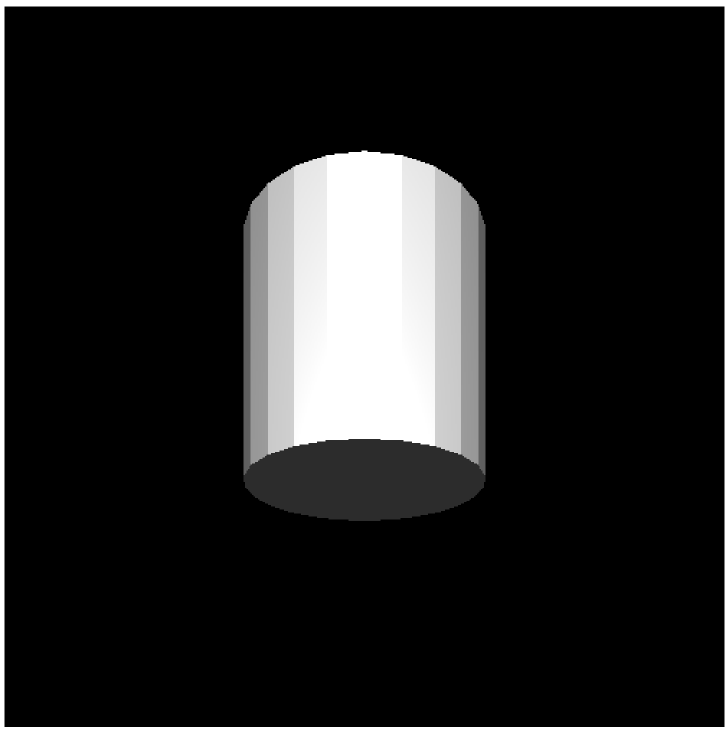

Figure 6: Ray-Traced Cylinder Vertical

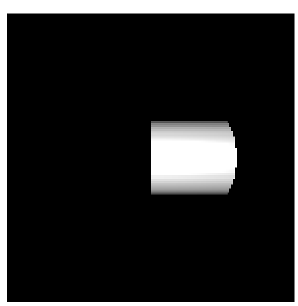

Figure 7: Ray-Traced Cylinder Horizontal

Similar to the sphere, a range-power diagram was generated for the cylinder mesh and shown in Fig.8.

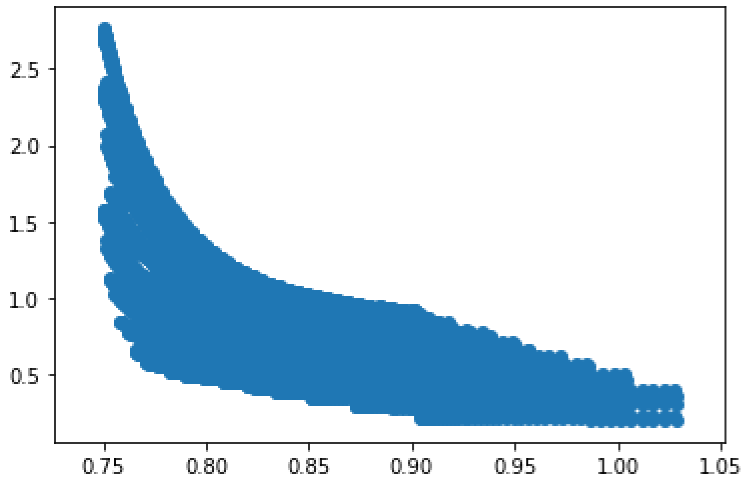

Figure 8: Range-Power Diagram for Cylinder Mesh

This range-power diagram was created for the cylinder mesh in the horizontal orientation. Unlike the sphere, this range-power diagram has several lines due to the number of facets on the mesh. To improve this diagram, it is possible to create a histogram and obtain the individual power values for a range of distances, rather than plot each distance.

To conclude, this project involved developing a ray-tracing model that can predict the intensities and locations of the reflections caused by the beam emitted by the radar interferometer. Although this goal was not completely met, the project is moving forward. The next steps of this project would include testing the code with multiple meshes in the scene. For example, a scene could have two cylinder meshes overlapping each other in some way and the code needs to correctly identify which meshes the rays are intersecting with. Another next step would be to test the code with complex meshes, such as the Stanford Bunny or Utah Teapot. Once it is confirmed that the code can manage multiple complex meshes, we can work towards recreating the lab set-up in CAD and applying the code to simulate how and where the light should hit.

3 References

[1] Aflak, Omar. “Ray Tracing From Scratch in Python | by Omar Aflak | The Startup | Medium.” Medium, The Startup, 26 July 2020

[2] Glassner, Andrew S. An Introduction to Ray Tracing.Elsevier, 1989.

[3] Ikram, Muhammad Z., Adeel Ahmad, and Dan Wang. "High-accuracy distance measurement using millimeter-wave radar." 2018 IEEE Radar Conference (RadarConf18). IEEE, 2018.

[4] Lu, Yifan, et al. "A 3-D Ray Tracing Model for Short-Range Radar Sensing of Hand Gestures." 2020 IEEE Asia-Pacific Microwave Conference (APMC). IEEE, 2020.

[5] Rasmont, Nicolas, et al. "Millimeter Wave Interferometry for Ejecta Concentration Measurements in Plume-Surface Interactions." AIAA SCITECH 2022 Forum. 2022.

4 Appendix

4.1 Ray-Traced Sphere Code

```python
import numpy as np
import matplotlib.pyplot as plt

def norm(vector):
    return vector / np.linalg.norm(vector)

def reflected(vector, axis):
    return vector - 2 * np.dot(vector, axis) * axis

# function inputs: sphere center, sphere radius, ray origin point, ray direction vector
# function output: minimum distance if an intersection between ray and sphere, else function returns "None"
def sphere_intersect(c, r, ray_origin, ray_direction):
    b = 2 * np.dot(ray_direction, ray_origin - c)
    c = np.linalg.norm(ray_origin - c) ** 2 - r ** 2
    delta = b ** 2 - 4 * c
    if delta > 0:
        t1 = (-b + np.sqrt(delta)) / 2
        t2 = (-b - np.sqrt(delta)) / 2
        if t1 > 0 and t2 > 0:
            return min(t1, t2)
    return None

# function inputs: number of objects in scene, ray origin point, ray direction vector
# function outputs: nearest object and minimum distance at which the nearest object intersects with the ray
def nearest_intersected_object(objects, ray_origin, ray_direction):
    distances = []
    for obj in objects:
        distances.append(sphere_intersect(obj['c'], obj['r'], ray_origin, ray_direction))
    nearest_object = None
    min_distance = np.inf
    for index, distance in enumerate(distances):
        if distance and distance < min_distance:
            min_distance = distance
            nearest_object = objects[index]
    return nearest_object, min_distance

# number of pixels the screen will be split into
pixel_array = 300
```

```python

max_depth = 1

# camera perspective
camera = np.array([0, 0, 1])
screen = (-1, 1, 1, -1) # left, top, right, bottom

# radar simulator
light = { 'position': np.array([0, 0, 1]), 'ambient': np.array([1, 1, 1]), 'diffuse': np.array([1, 1, 1]), 'specular': np.array([1, 1, 1]) }

objects = [{ 'c': np.array([0, 0, -1]), 'r': 1, 'ambient': np.array([0.1, 0.1, 0.1]), 'diffuse': np.array([0.525, 0.525, 0.525]), 'specular': np.array([1, 1, 1]), 'shininess': 100, 'reflection': 0.5 }]

image = np.zeros((pixel_array, pixel_array, 3))
rays = []
intersection_distance_array = []
intersection_to_light_array = []
camera_to_light_array = []
power = []

# parsing through pixels from top to bottom
for i, y in enumerate(np.linspace(screen[1], screen[3], pixel_array)):
    # parsing through pixels from left to right
    for j, x in enumerate(np.linspace(screen[0], screen[2], pixel_array)):
        # screen is on origin
        pixel = np.array([x, y, 0])
        origin = camera # receiver
        rays.append(np.linalg.norm(pixel - origin))
        direction = norm(pixel - origin)

        color = np.zeros((3))
        reflection = 1

        for k in range(max_depth):
            # check for intersections
            nearest_object, min_distance = nearest_intersected_object(objects, origin, direction)
            if nearest_object is None:
                break

            intersection_distance_array.append(min_distance)
            intersection = origin + min_distance * direction
            normal_to_surface = norm(intersection - nearest_object['c'])
            shifted_point = intersection + 1e-6 * normal_to_surface
            intersection_to_light = norm(light['position'] - shifted_point)
```

```
            _,min_distance = nearest_intersected_object(objects, shifted_point, intersection_to_light)

            intersection_to_light_distance = np.linalg.norm(light['position'] - intersection)
            intersection_to_light_array.append(intersection_to_light_distance)

            camera_to_light_array.append(np.linalg.norm(intersection) + intersection_to_light_distance)
            is_shadowed = min_distance < intersection_to_light_distance

            if is_shadowed:
                break

            illumination = np.zeros((3))

            # ambiant
            illumination += nearest_object['ambient'] * light['ambient']


            # diffuse
            illumination += nearest_object['diffuse'] * light['diffuse'] * np.dot(intersection_to_light, normal_to_surface)

            # specular
            intersection_to_camera = norm(camera - intersection)
            H = norm(intersection_to_light + intersection_to_camera)
            illumination += nearest_object['specular'] * light['specular'] * np.dot(normal_to_surface, H) ** (nearest_object['shininess'] / 4)

            # reflection
            color += reflection * illumination
            power.append(np.sum(color)/3)
            reflection = nearest_object['reflection']

            # new ray origin and direction
            origin = shifted_point
            direction = reflected(direction, normal_to_surface)

        image[i, j] = np.clip(color, 0, 1)
    # print("%d/%d" % (i + 1, pixel_array))

plt.imsave('sphere.png', image)

plt.plot(intersection_distance_array, power)
plt.show()
```

```

f = 1*(10**9) # 60 GHz
c =  3*(10**8) # speed of light
phi_array = []
for i in range(0, len(intersection_distance_array)):
    d = intersection_distance_array[i]
    phi_array.append(np.mod((d/(c/f))*2*np.pi, 2*np.pi))

plt.scatter(intersection_distance_array, phi_array)
plt.show()
```

4.2 Ray-Traced Cylinder Mesh Code

```
import numpy as np
import matplotlib.pyplot as plt
import trimesh

def normalize(vector):
    return vector / np.linalg.norm(vector)

def reflected(vector, axis):
    return vector - 2 * np.dot(vector, axis) * axis

def triangle_intersect(mesh_list, ray_origins, ray_directions):
    locations = np.full((len(ray_origins), 3), np.inf)
    normals = np.full((len(ray_origins), 3), 0.0)
    intersected_mesh = np.full((len(ray_origins), 1), np.nan)

    for i in range(0, len(mesh_list)):
        active_mesh = mesh_list[i]
        active_locations = np.full((len(ray_origins),3), np.inf)
        active_normals = np.full((len(ray_origins),3), 0.0)
        active_index_tri = np.full((len(ray_origins),1), np.nan)
        shortlist_active_locations, index_ray, shortlist_active_index_tri = active_mesh.ray.intersects_location(ray_origins, ray_directions, multiple_hits=False)

        for k, index in enumerate(index_ray):
            #print(active_locations[index])
            #print(shortlist_active_locations[k])
            active_locations[index] = shortlist_active_locations[k]
            active_normals[index] = active_mesh.face_normals[shortlist_active_index_tri[k]]

        mask = np.abs(active_locations - ray_origins) < np.abs(locations - ray_origins)
```

```
        #print(mask)
        locations[mask] = active_locations[mask]
        normals[mask] = active_normals[mask]
        #print(normals)
        #print(active_normals)
        intersected_mesh[np.transpose(mask[:,0])] = i

    return locations, normals, intersected_mesh


# number of pixels the screen will be split into
width = 350
height = 350

camera = np.array([0, 0, 1])
ratio = float(width) / height
screen = (-1, 1 / ratio, 1, -1 / ratio) # left, top, right, bottom

mesh_list = [trimesh.load('cylinder_4.stl')]

# number of reflections allowed
max_depth = 3

# radar simulator
# light = { 'position': np.array([0, 0, 1]), 'ambient': np.array([1, 1, 1]), 'diffuse': np.array([1, 1, 1]), 'specular': np.array([1, 1, 1]) } # color vectors can be replaced by scalar values: DONE
light = { 'position': np.array([0, 0, 1]), 'ambient': 1, 'diffuse': 1, 'specular': 1} # color

# objects = {'file': 'cylinder_4.stl', 'ambient': np.array([0.1, 0.1, 0.1]), 'diffuse': np.array([0.5, 0.5, 0.5]), 'specular': np.array([1, 1, 1]), 'shininess': 100, 'reflection': 0.5 }
objects = {'mesh_index': 0, 'ambient': 0.1, 'diffuse': 0.5, 'specular': 1, 'shininess': 100, 'reflection': 0.5 }

image3 = np.zeros((width, height, 3))

intersection_distance_array = []
intersection_to_light_array = []
camera_to_light_array = []
distance = []
power = []

ray_origin = []
ray_direction = []
for yy in np.linspace(screen[1], screen[3], height):
```

```
    for xx in np.linspace(screen[0], screen[2], width):
        # screen is on origin
        pixel = np.array([xx, yy, 0])
        origin = camera # receiver
        ray_origin.append(origin)
        ray_direction.append(pixel - origin)

ray_origins = np.array(ray_origin)
ray_directions = np.array(ray_direction)

locations, normals, intersected_mesh = triangle_intersect(mesh_list, ray_origins, ray_directions)

# parsing through pixels from top to bottom
for i, y in enumerate(np.linspace(screen[1], screen[3], height)):
    # parsing through pixels from left to right
    for j, x in enumerate(np.linspace(screen[0], screen[2], width)):

        color = np.zeros((3))
        reflection = 1

        for k in range(max_depth):
            # check for intersections

            intersection = locations[i*width+j]
            distance.append(np.linalg.norm(intersection - ray_origins[0]))
            normal = normals[i*width+j]

            #intersection_distance_array.append(distance)
            #normal_to_surface = norm(intersection - normal)
            intersection_to_light = normalize(light['position'] - intersection) # change the name of "norm" function: DONE

            intersection_to_light_distance = np.linalg.norm(light['position'] - intersection)
            #intersection_to_light_array.append(intersection_to_light_distance)

            #camera_to_light_array.append(np.linalg.norm(intersection) + intersection_to_light_distance)

            illumination = np.zeros((3))

            # ambient
            illumination += objects.get('ambient') * light['ambient']

            # diffuse
```

```
            illumination += objects.get('diffuse') * light['diffuse'] * np.dot(intersection_to_light, normal)

            # specular
            intersection_to_camera = -1*ray_directions[i*width+j] # replaced this with ray_directions[i*width+j]: DONE
            H = normalize(intersection_to_light + intersection_to_camera) # H is halfway vector in the Blinn-Phong reflection model: DONE
            illumination += objects.get('specular') * light['specular'] * np.dot(normal, H) ** (objects.get('shininess') / 4)

            # reflection
            color += reflection * illumination
            power.append(np.sum(color)/3)
            reflection *= objects.get('reflection')

        image3[i, j] = np.clip(color, 0, 1)
    # print("%d/%d" % (i + 1, pixel_array))

plt.imsave('triangle_mesh.png', image3)

plt.scatter(distance, power)
plt.show()
```