Raytracing Project – Christian Metzler

1. (Fundamental) Rendering loop

The rendering loop is the basis for each rendering process.

```
with open(filename, "w") as f:
    f.write(f"P3\n{width} {height}\n255\n")
    for i in range(height):
        for j in range(width):
            f.write(f"{color[0]} {color[1]} {color[2]} ")
        f.write('\n')
```

This code is used for generating a PPM image file, which is a simple file format for storing images.

```
with open(filename, "w") as f:
```

This line opens a file named filename in write mode ("w") and assigns it to a variable named f. The with statement is used to ensure that the file is properly closed after it's used, even if an error occurs.

```
f.write(f"P3\n{width} {height}\n255\n")
```

This line writes the PPM header information to the file, which includes the format identifier (P3), the image dimensions (width and height), and the maximum color value (255).

```
for i in range(height): for j in range(width): f.write(f"{color[0]}
{color[1]} {color[2]} ") f.write('\n')
```

This is the main rendering loop that writes the pixel values to the file. The loop iterates over each row (i) and column (j) in the image, and writes the RGB color values (color[0], color[1], and color[2]) to the file for each pixel. The f.write('\n') at the end of each row writes a newline character to the file to start a new row.

Using files instead of print to render multiple scenes with after each other automatically This code is useful for rendering multiple scenes and saving each one as a separate image file. By changing the filename variable each time the loop is run, you can automatically save each scene as a new file. This is more efficient and easier to manage than printing the image data to the console or terminal, especially if you need to render a large number of images.

As for now the calculated picture is only the default color we set.

2. (Fundamental) Camera

The next step is to define a camera.

```
camera = Camera(focal_length=1, aspect_ratio=(16 / 9), image_width=400)
```

The camera object in this code is an instance of the Camera class, which represents the virtual camera used to generate the image. The camera has several attributes that define its properties:

focal_length: This is the distance from the camera to the image plane, which affects the field of view and perspective of the image. A shorter focal length produces a wider field of view, while a longer focal length produces a narrower field of view.

aspect_ratio: This is the ratio of the width to the height of the image, which determines the shape of the image. A wider aspect ratio produces a wider image, while a narrower aspect ratio produces a taller image.

image_width: This is the width of the image in pixels, which determines the resolution of the image.

Initializing the camera outside of the main rendering loop is useful because it allows you to reuse the same camera settings for multiple scenes without having to recreate the camera object each time. This is more efficient and easier to manage than creating a new camera object for each scene, especially if you need to render a large number of scenes. Additionally, initializing the camera outside of the loop allows you to easily adjust the camera settings and see the effects on all scenes, rather than having to modify the camera settings for each individual scene.

The camera class is implemented as it is shown below.

```
class Camera():
    def __init__(self, focal_length, aspect_ratio, image_width,
```

```
origin=Point([0, 0, 0])):
        self.image_heigth = int(image_width / aspect_ratio)
        self.image_width = image_width
        self.focal_length = focal_length
        self.aspect_ratio = aspect_ratio
        self.origin = origin

        self.viewport_height = 2.0
        self.viewport_width = aspect_ratio * self.viewport_height
```

```
class Ray():
    def __init__(self, origin, direction):
        self.origin = origin
        self.direction = direction

    def calculate_position(self, t):
        return self.origin + t * self.direction
```

In ray tracing, a ray is typically represented by its origin and direction. The Ray class in the code represents a ray in three-dimensional space, where origin is a three-dimensional point representing the starting point of the ray, and direction is a three-dimensional vector representing the direction in which the ray is pointing.

The calculate_position method of the Ray class is used to determine the position of the ray at a given time t along its direction. This is useful for finding where the ray intersects with objects in the scene, such as geometry or lights.

The method simply calculates the position of the ray at the given time t by multiplying the direction vector by t and adding the result to the origin point. This results in a new three-dimensional point that represents the position of the ray at that time.

For example, if the origin of a ray is at point (0,0,0) and its direction is (1,0,0), then calling calculate_position(2) on that ray would return the point (2,0,0), which is two units along the x-axis from the origin.

This method is used extensively in the ray tracing process, such as for finding intersections between rays and scene objects or determining the path of light rays in the scene.

3. (Fundamental) Objects: shape

Now we want to render some actual objects. For this we need to define how the renderer reacts when a ray hits an object.

```
def ray_color(ray, world):
    hit_context = world.get_nearest_hit(ray)
    if hit_context["distance"] > 0:
        return hit_context["color"]
    else:
        unit_direction = Vec.normalize(ray.direction)
        t = 0.5 * (unit_direction[1] + 1.0)
        color = (1.0 - t) * Color([1.0, 1.0, 1.0]) + t * Color([0.5, 0.7,
1.0])
        return color
```

The ray_color function takes a ray and a world as input and returns the color of the ray after it intersects with objects in the scene.

The function first uses the get_nearest_hit method of the world object to determine the nearest intersection of the ray with objects in the scene. If there is an intersection (hit_context["distance"] > 0), the function returns the color of the intersected object (hit_context["color"]).

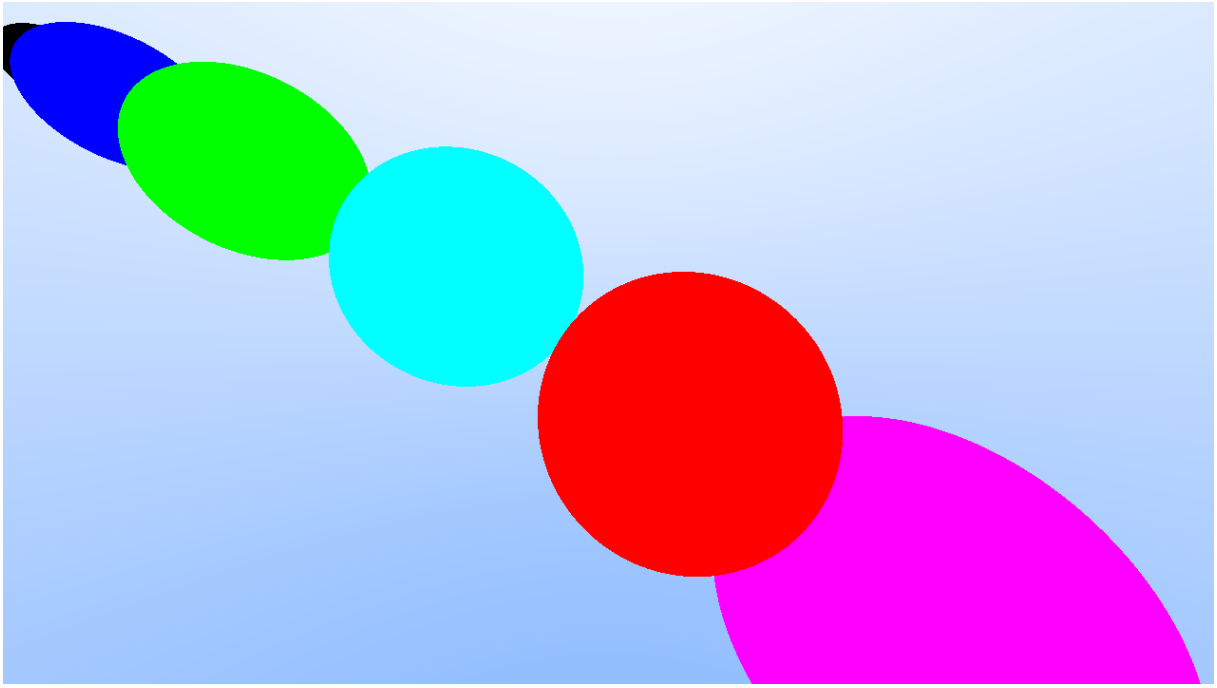
If the ray does not intersect with any objects in the scene, the function calculates a background color based on the direction of the ray. The background color is a gradient that blends from white at the bottom of the image to light blue at the top of the image.

To calculate the background color, the function first normalizes the direction vector of the ray using the Vec.normalize method. The function then calculates a value t that represents the vertical position of the ray in the image, with values ranging from 0 at the bottom of the image to 1 at the top of the image. This is done by adding 1 to the y-coordinate of the normalized direction vector and dividing the result by 2 (t = 0.5 * (unit_direction[1] + 1.0)).

Finally, the function calculates the background color by interpolating between white and light blue based on the value of t. The resulting color is a blend of white and light blue, with more blue at the top of the image and more white at the bottom.

Overall, the ray_color function is responsible for computing the color of a given ray in the scene, taking into account the objects it intersects with and the background color. It is a fundamental step in the ray tracing process and is called repeatedly for each pixel in the image to generate the final rendered scene.

There is no bouncing of rays implemented yet.



The Sphere class represents a spherical object in the scene. It has three main attributes: color, center, and radius.

color: The color of the sphere, represented as an instance of the Color class.

center: The center point of the sphere, represented as an instance of the Point class.

radius: The radius of the sphere, represented as a float.

The Sphere class also has a method called hit_sphere, which takes a ray as input and returns a dictionary containing information about the intersection of the ray with the sphere.

```
def hit_sphere(self, ray: Ray):
    context = {"distance": -1}
    oc = ray.origin - self.center
    a = np.dot(ray.direction, ray.direction)
    b = 2.0 * np.dot(oc, ray.direction)
    c = np.dot(oc, oc) - self.radius * self.radius
    discriminant = b * b - 4 * a * c
    if (discriminant >= 0):
        t = (-b - np.sqrt(discriminant)) / (2.0 * a)
        if t > 0:
            context["distance"] = t
            context["intersection_point"] = ray.calculate_position(t)
            normal = Vec.normalize(context["intersection_point"] -
self.center)
            context["front_face"] = (np.dot(ray.direction, normal) < 0)
            context["normal"] = normal if context["front_face"] else -
normal
            context["color"] = self.color
    return context
```

The hit_sphere method first initializes a context dictionary with a default distance of-1, which will be used to store information about the intersection. It then calculates the components of the quadratic equation that describes the intersection of the ray with the sphere, using the ray's origin, direction, and the center and radius of the sphere.

If the discriminant of the quadratic equation is greater than or equal to zero, the method computes the distance to the nearest intersection of the ray with the sphere (t). If the distance is greater than zero, the method updates the context dictionary with information about the intersection, including the intersection point, surface normal, and whether the intersection is from the front or the back of the sphere. It also sets the color of the intersection to the color of the sphere.

Finally, the method returns the context dictionary, which can be used by the renderer to compute the color of the pixel at the intersection point.

Overall, the Sphere class and its hit_sphere method are responsible for determining if a given ray intersects with the sphere and returning information about the intersection if one occurs. This information is then used by the renderer to compute the final color of the pixel at the intersection point.

```
sp0 = Sphere(center=Point([-16, -8, -10]), radius=1, color=Color([0, 0,
0]))
```

Here is the initialization of black sphere with the position-16,-8,-10 and the radius 1.

4. (Fundamental) Enhancing camera rays and rendering

To generate multiple rays per pixel the main rendering loop has to be changed.

```
for h in range(height):
    for w in range(width):
```

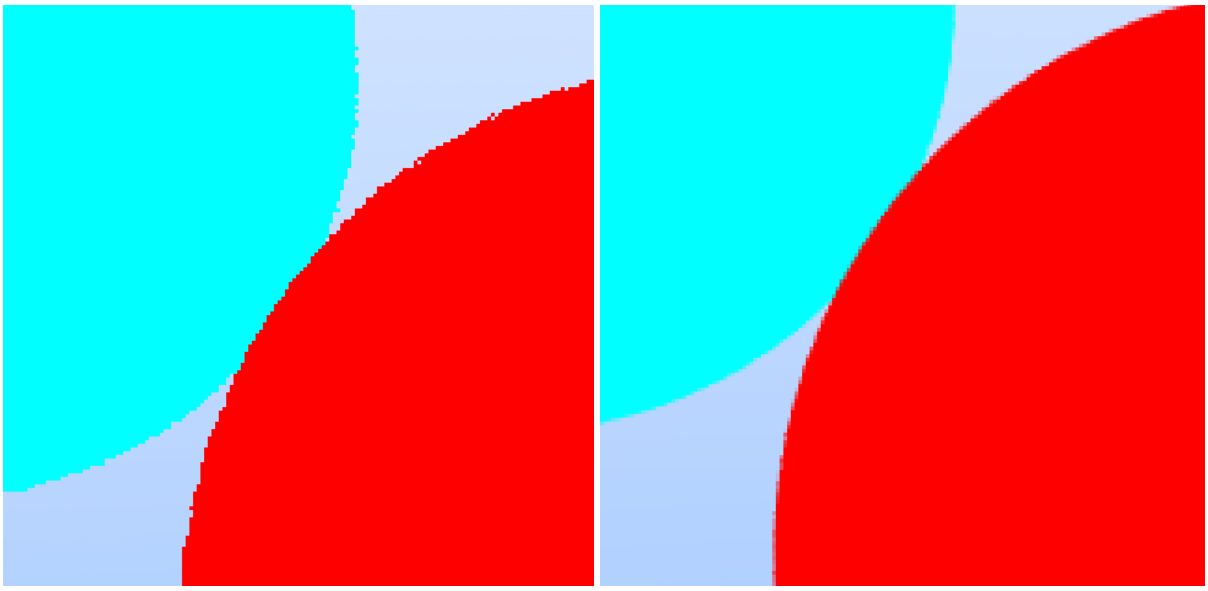
```
        colors = np.array([0, 0, 0], dtype="float64")
        for i in range(samples):
            dw = (w + np.random.random()) / (width - 1)
            dh = (h + np.random.random()) / (height - 1)
            ray = camera.get_ray(dh, dw)
            colors += ray_color(ray, world)
            color = np.clip(colors / samples, 0, 1)
        f.write(f"{int(255.99 * color[0])} {int(255.99 * color[1])}
{int(255.99 * color[2])} ")
    f.write("\n")
```

The main difference in this new rendering loop is the addition of a nested loop that samples multiple rays per pixel. Instead of just computing the color of each pixel based on a single ray, the loop generates samples number of rays for each pixel and averages their colors to produce a final color value.

The benefits of this approach are twofold. First, it reduces the amount of noise and improves the overall quality of the image. By sampling multiple rays, the renderer can better approximate the true color of each pixel and reduce the impact of random noise in the rendering process.

Second, it allows for better anti-aliasing of the image. Anti-aliasing is a technique used to smooth the edges of objects in the image and prevent jagged or pixelated edges. By sampling multiple rays per pixel, the renderer can better capture the subtle variations in color and brightness that occur along object edges and produce a smoother, more natural-looking image.

Overall, by sampling multiple rays per pixel, the new rendering loop can produce higher quality and more accurate images with better anti-aliasing and reduced noise.



Picture 1 sample pp vs picture 16 samples pp

One can see the differences mainly on the edges of the object. As we have not yet implemented any shading there is no difference on the texture.

5. (Fundamental) Object material: diffuse

Now we implement the first real physical surface. To achieve a scattering effect the rays hitting a surface will be randomly reflected. We implement an abstract Material class, which will contain all our different materials.

The code for the diffuse material class is shown below.

```
class Diffuse(Material):
    def __init__(self, color):
        super().__init__(color)

    def scatter(self, ray_in, context):
        new_direction = context["normal"] + Vec.random_in_unit_sphere()
        if Vec.near_zero(new_direction):
            new_direction = context["normal"]
        scattered_ray = Ray(origin=context["intersection_point"],
direction=new_direction)
        return scattered_ray, self.color
```

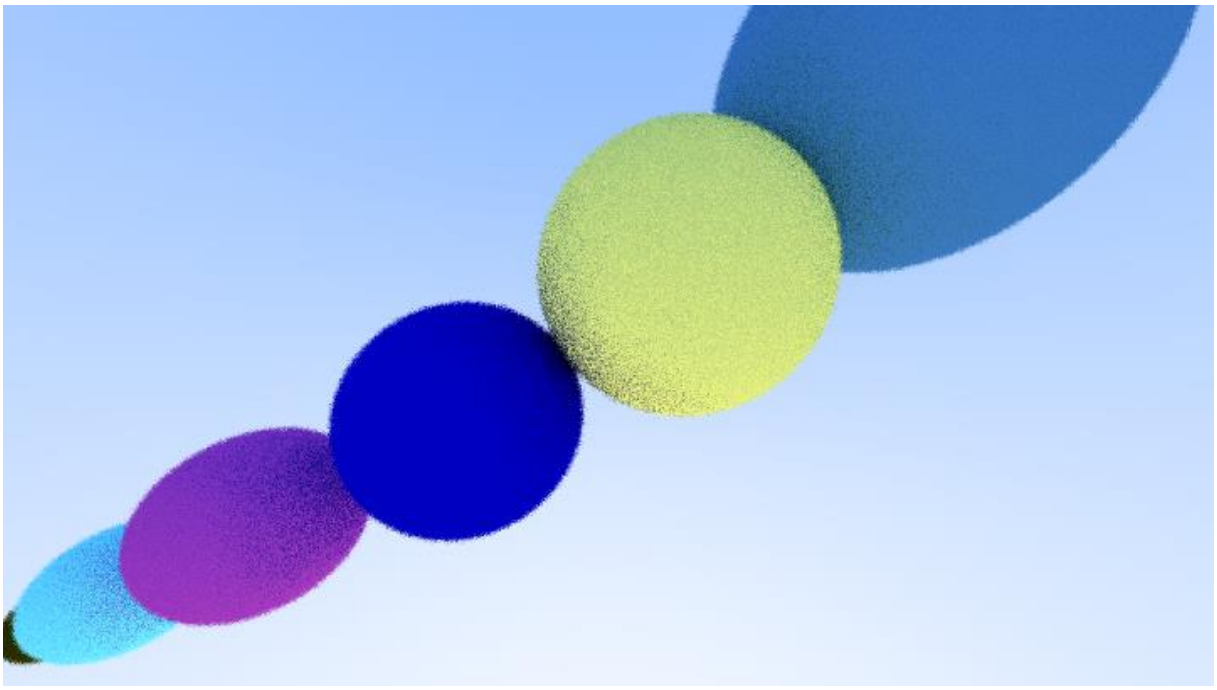
This code implements a Diffuse material for objects in a raytracer. A diffuse surface is a surface that scatters incoming light uniformly in all directions, rather than reflecting it at a single angle like a specular surface. This results in a matte, non-shiny appearance, and is a common material used for rough or textured surfaces like concrete or skin.

The scatter method of the Diffuse class implements this behavior by generating a new scattered ray that points in a random direction around the surface normal. The Vec.random_in_unit_sphere() method is used to generate a random direction that is constrained to lie within a unit sphere centered at the intersection point. This is because scattering should not result in rays pointing away from the surface, or else it would violate the conservation of energy.

However, if the new direction vector generated by Vec.random_in_unit_sphere() is near zero (i.e., it is very close to the surface normal), it means that the surface is nearly flat or smooth, and generating a random direction would not produce much scattering. In this case, the method sets the new direction to be the surface normal itself to simulate perfect diffuse reflection.

In summary, the Diffuse material simulates a rough, matte surface that scatters incoming light uniformly in all directions. It uses a random direction generator to generate scattered rays and handles the special case of surfaces that are nearly flat or smooth.

The resulting image is shown below.



6. (Fundamental) Object material: specular

The next material we are implementing a specular surface.

Specular surfaces are surfaces that reflect light in a highly directional manner, such as mirrors or polished metal. The reflection angle is determined by the angle of incidence and the surface normal, following the law of reflection. The specular effect is achieved by computing the reflection of the incoming ray at the intersection point of the surface and then tracing a new ray in that direction.

```
def scatter(self, ray_in: Ray, context: dict):
    reflected = self.reflect(Vec.normalize(ray_in.direction),
context["normal"])
    scattered = Ray(origin=context["intersection_point"],
direction=reflected)
    return scattered, self.color

def reflect(self, vector: Vector, normal: Vector):
    return vector - 2 * np.dot(vector, normal) * normal
```

In the code, the scatter method of the Specular material class takes in an incoming ray and a context dictionary that contains information about the intersection point of the ray with the surface, such as the surface normal and the intersection point itself. The method first calculates the direction of the reflected ray using the reflect method, which computes the reflection of the incoming ray around the surface normal. The method then creates a new Ray object with the origin at the intersection point and the direction of the reflected ray.

The reflect method computes the reflection of a given vector around a given surface normal using the formula:

reflected_vector = vector- 2 * np.dot(vector, normal) * normal

Here, vector is the incoming ray direction, normal is the surface normal at the intersection point, and np.dot() calculates the dot product of the two vectors. The resulting reflected vector is the direction of the reflected ray.

The specular effect can be achieved by recursively tracing rays of light reflecting off the surface. However, this can lead to infinite recursion and performance issues, so a maximum recursion depth or a termination condition is typically used to prevent this.

7. (Optional) Object material: specular transmission

For the next material we are adding specular transmission. Specular transmission is a phenomenon that occurs when light passes through a surface and is transmitted in a preferred direction, rather than being scattered in all directions. This phenomenon is commonly referred to as refraction.

The amount of light that is refracted depends on the angle of incidence, the refractive indices of the two media, and the wavelength of the light. The relationship between these factors is described by Snell's law:

n1 * sin(theta1) = n2 * sin(theta2)

where n1 and n2 are the refractive indices of the two media, theta1 is the angle of incidence, and theta2 is the angle of refraction.

This is implemented in the material sub class Transmissive.

```
def scatter(self, ray_in, context):
    refraction_ratio = 1 / self.ior if context["front_face"] else self.ior
    unit_direction = Vec.normalize(ray_in.direction)
    cos_theta = min(np.dot(-unit_direction, context["normal"]), 1.0)
    sin_theta = np.sqrt(1.0 - cos_theta * cos_theta)

    if (refraction_ratio * sin_theta > 1.0) or (self.reflectance(cos_theta,
refraction_ratio) > np.random.rand()):
        direction = super().reflect(unit_direction, context["normal"])
    else:
        direction = self.refract(unit_direction, context["normal"],
refraction_ratio)

    scattered = Ray(context["intersection_point"], direction)
    return scattered, self.color

def refract(self, uv, n, etai_over_etat):
    cos_theta = min(np.dot(-uv, n), 1.0)
    r_out_perp = etai_over_etat * (uv + cos_theta * n)
    r_out_parallel = np.sqrt(abs(1.0 - Vec.length_squared(r_out_perp))) * n
    return -(r_out_perp + r_out_parallel)
```

This code is implementing the scatter function for a Dielectric material. This function is responsible for computing the behavior of the light ray as it interacts with the material.

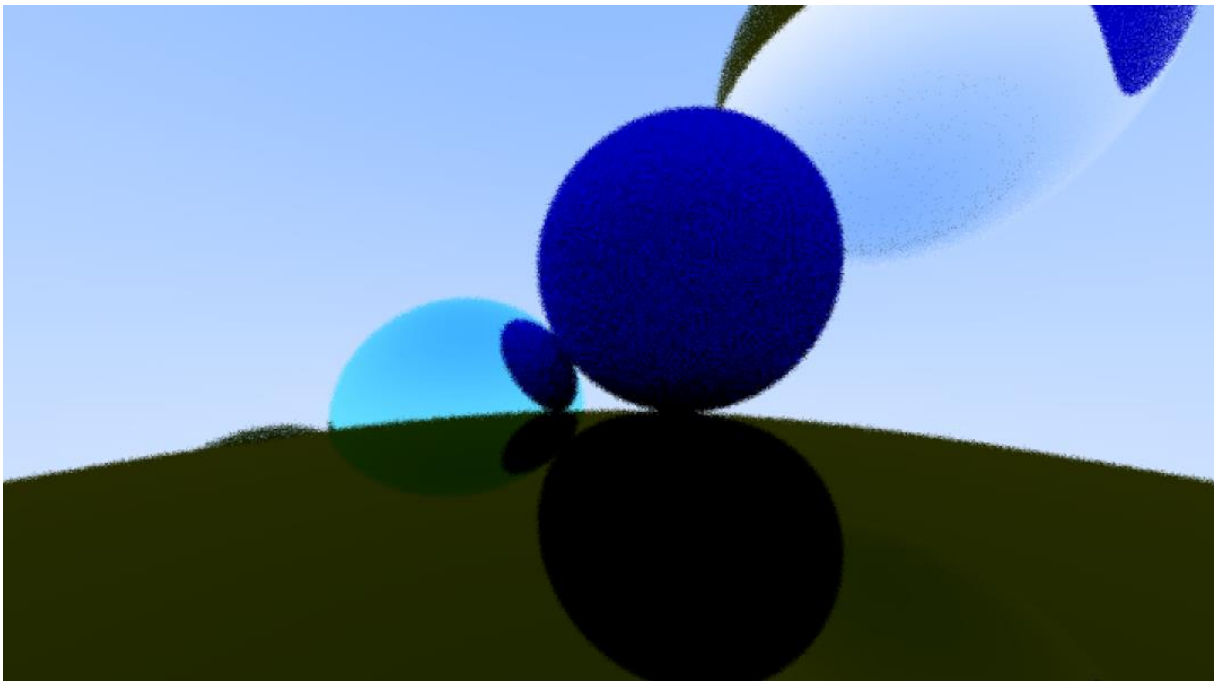
The code first calculates the refraction ratio of the material based on the angle of incidence of the incoming ray and whether the ray is entering or exiting the material. It then computes the angles of incidence and refraction using Snell's law.

Next, it checks whether the incoming ray is refracted or reflected based on the angle of incidence and the refractive index. If the incoming ray is reflected, it uses the reflect method to compute the reflection direction. If it is refracted, it uses the refract method to compute the refraction direction.

Finally, it returns a new ray with the computed direction, along with the color of the material.

The refract method takes in the unit vector uv representing the incoming ray, the surface normal n, and the ratio of the refractive indices of the two media (etai_over_etat). It first calculates the cosine of the angle between the incoming ray and the surface normal. It then calculates the perpendicular and parallel components of the outgoing ray based on this angle and the ratio of refractive indices. The outgoing ray is the sum of these two components, with the negative sign indicating that it is traveling in the opposite direction.

The final result looks like this.



The picture shows several translusive, diffuse and specular spheres.

9. (Optional) Positioning and orienting camera

In the final change for this project the camera model will be overhauled. The new camera will be defined by a look from and look at attribute.

```
cam1 = Camera(vfov=20, lookfrom=Point([0, 0, 0]), lookat=Point([0, 0, 1]))
```

This camera implementation allows the user to specify the vertical field of view (vfov) in degrees, as well as the lookfrom and lookat points in 3D space. By specifying these parameters, the camera is able to generate rays that emulate the behavior of a real camera lens.

The vfov parameter allows the user to control the amount of the scene that is visible in the rendered image. A larger vfov value results in a wider field of view, while a smaller value results in a narrower field of view. This allows the user to adjust the perspective of the rendered image to suit their needs.

The lookfrom and lookat points determine the position and orientation of the camera in the scene. By specifying these points, the user can control where the camera is located and what direction it is

facing. This allows the user to generate images from different viewpoints and angles, and to capture different aspects of the scene.

Overall, this camera implementation provides a flexible and customizable way to generate ray-traced images of 3D scenes.

To achieve this we have to update the Camera class.

```
class Camera():
    def __init__(self, vfov=90, aspect_ratio=(16 / 9), lookfrom=Point([0,
0, 0]), lookat=Point([0, 0, -1]),
                 vup=Vector([0, 1, 0])):
        theta = np.deg2rad(vfov)
        h = np.tan(theta / 2)
        viewport_height = 2 * h
        viewport_width = aspect_ratio * viewport_height
        w = Vec.normalize(lookfrom - lookat)
        u = Vec.normalize(np.cross(vup, w))
        v = -np.cross(w, u)

        self.origin = lookfrom
        self.horizontal = viewport_width * u
        self.vertical = viewport_height * v
        self.lower_left_corner = self.origin - self.horizontal / 2 -
self.vertical / 2 - w
```