# Ray Tracing Fundamentals

Charles Zitella

April 2024

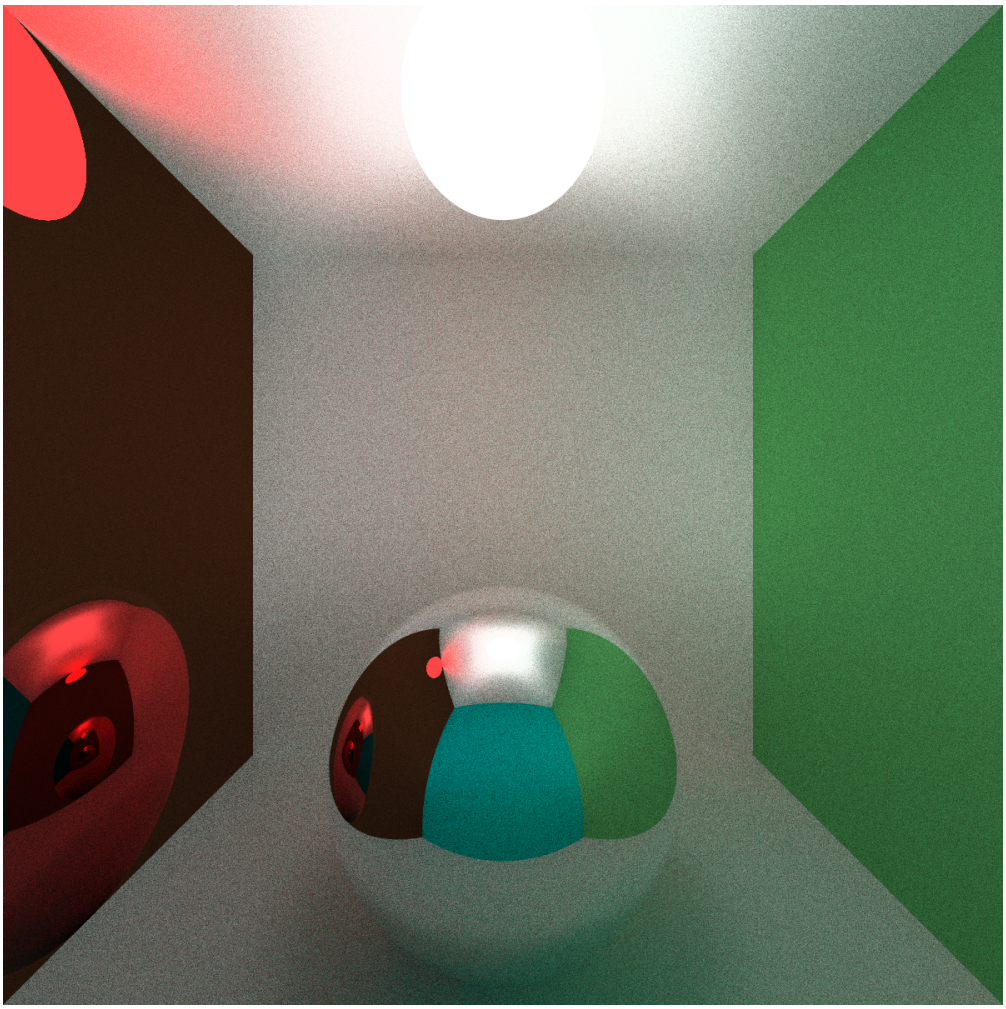


Figure 1: Final Render

# 1 Mini-Manual

Objects are created by appending them to the python list "objects". In order to run the program, you must run "main.py". Currently a scene with with a cornell box and sphere is set up but that can be changed by adjusting values in "main.py". There is no interface, image will be saved to image.png while loading and mutliple renders together will be saved to Final.png.

# 2 Creation of the Image

The way the image was created was using Matplotlib and its function "imsave", which converts a 3d array into an image. Each item in the array is a 3d set of values representing the rgb colour values for a given pixel [r, g, b]. The 3d array itself is created using the numpy module, for its efficiency over python lists.

# 3 Trace Function

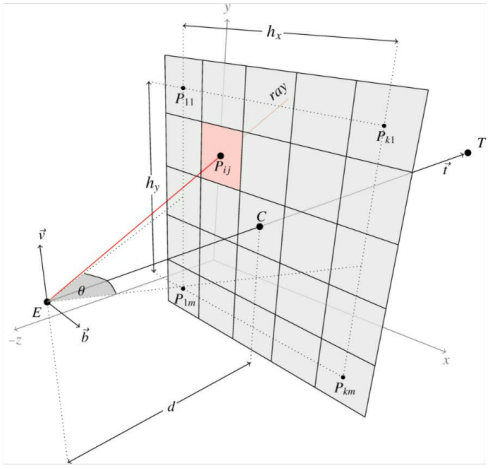



Figure 2: viewport

The trace function is the most fundamental function in the project. Simply put, it calculates the colour that should be attributed to each pixel. It does this by "shooting" a ray, from the origin/position of the camera (in this case (0,0,0)) towards the corresponding pixel on the viewport it is trying to find a colour for (see image above). The viewport represents the actual image that is being produced, and the camera is just the place to shoot rays from, therefore the viewport can be seen as broken up into pixels, but that just

tells us where to shoot the ray towards. After the ray is shot out, it must either intersect with an object or potentially shoot off into the abyss. The way the code is set up now, a ray shooting off into the abyss will just return the color black, but this could easily be changed for a sky-like colour, as can be seen in this photo:

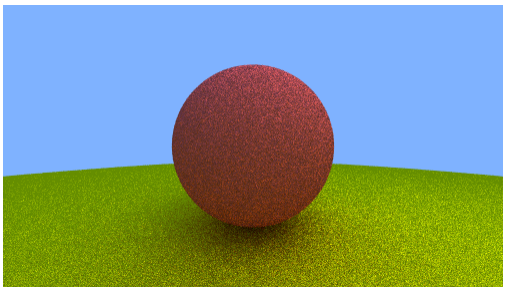


Figure 3: sphere with light source and sky

If the ray is to intersect an object, than a resulting ray is shot from the point of intersection. The helper function "collide" calculates the closest point of intersection using techniques I will describe later. The resulting ray's direction depends on the type of material, which I will discuss later. This process is repeated until the ray eventually either hits a light source, shoots off into the abyss, or reaches the max bounce limit. If no light source is reached, the colour black is returned. Every single bounce of the ray, it picks up information regarding the colour of the object it has hit.

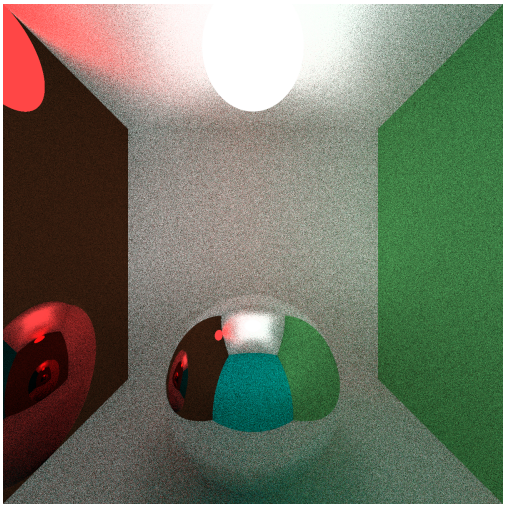
```
if hit[0]:
    normal = hit[1].normal(ray.at(hit[2]), ray)
    emittedlight = hit[1].emission * hit[1].emission_strength #emitted light, zero if not a light source
    total_light += emittedlight*colour #add emitted light
    colour *= hit[1].material.colour #pick up the colour of the object
```

Figure 4: calculations upon intersection

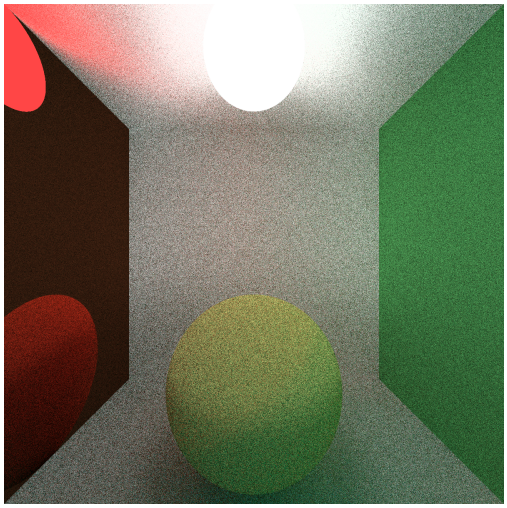
Keeping track of the colours allows for bounce lighting, contributing to the realism of the image. After a light source is hit, the loop exits and the "incoming_light" is returned as the colour found for the pixel. It is the summation of all the colours the ray hit multiplied by the strength of light source.

## 3.1 Materials

Materials of the objects have essentially one use, and that is to decide the direction of the subsequent ray after an intersection. I have provided an implementation of two types of materials with two different ray scattering methods; specular and diffuse.



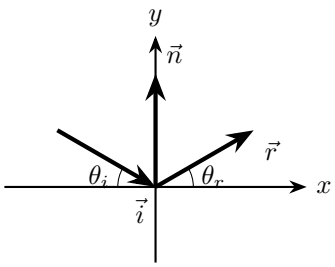
(a) Specular



(b) Diffuse

Figure 5: specular vs. diffuse materials

The specular material determines the bounced off ray using a simple reflection calculation across the normal.




looking at the diagram:

$$\vec{r} = -\vec{i} + \vec{i}\perp + \vec{i}\perp, \quad \vec{i}\perp = \vec{i} - (\vec{i}\cdot\vec{n}) * \vec{n}$$

$$\vec{r} = -\vec{i} - 2(-\vec{i} + (\vec{i}\cdot\vec{n}) * \vec{n})$$

$$\vec{r} = \vec{i} - 2(\vec{i}\cdot\vec{n}) * n$$

As for diffuse reflections, the way I implemented it was by using the hemisphere around the normal.

Essentially a random unit vector somewhere on the hemisphere is calculated and that is the direction of the bounced ray:

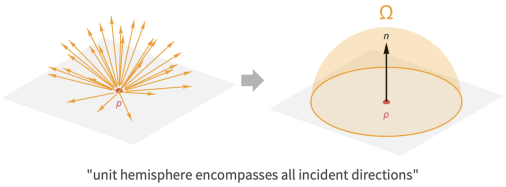



Figure 6: unit hemisphere distribution

These are the two types of materials I implemented due to their simplicity, but other types of materials could be implemented without real difficulty, i.e. using Snell's law for refraction, etc...

## 3.2 Anti-Aliasing

One technique used in a multitude of applications that I have applied here is Anti-Aliasing. Essentially, it is the blending of colours on sharp pixel boundaries, smoothing out jagged edges that are prevalent in low pixel counts. The way this is implemented in the code is simple. Instead of shooting out one ray per pixel, many rays are shot out and their colour values are averaged.

```
for s in range(samples_per_pixel):
    offset = np.array([random.random()-0.5, random.random()-0.5, 0])
    ray = Ray(center, first_pixel + (j+offset[0])*pixel_delta_u + (i+offset[1])*pixel_delta_v) #shoots ray from camera to pixel
    pixel_colour += trace(ray, objects)
pixel_colour *= pixel_samples_scale
```

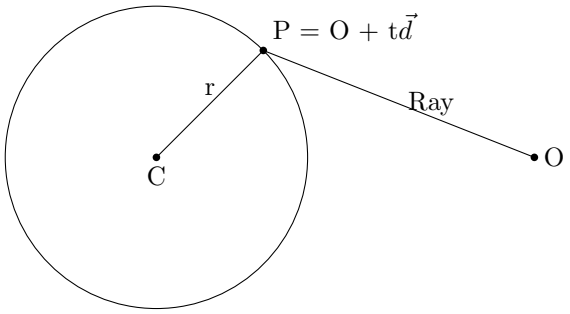
Figure 7: anti-aliasing implementation

Every pixel has "samples_per_pixel" amount of rays created, all with a random offset from the center of the pixel so that there is some variation. They each are traced until they return a colour value, which are all added up and averaged to find the average colour value for the pixel, rather than just the colour value that a single ray found.

# 4 Intersection

## 4.1 Sphere

A sphere is the easiest object to calculate intersection for, so naturally it is the one I will begin with. The following are the calculations required for finding if there is an intersection, as well as the point of intersection.




Before starting, some givens:

$$P = O + t\vec{d}, \text{where } \vec{d} \text{ is the unit vector } \frac{\vec{OP}}{\|\vec{OP}\|} \text{ and } \|\vec{v}\|^2 = \vec{v} \cdot \vec{v}$$

Now let's start:

$$\begin{aligned} r = \|C - P\| \quad --> \quad (C-P)\cdot(C-P) &= r^2; \\ (C - (O + t\vec{d})) \cdot (C - (O + t\vec{d})) &= r^2 \\ (C - O - t\vec{d})) \cdot (C - O - t\vec{d})) &= r^2 \\ (-t\vec{d} + (C-O)) \cdot (-t\vec{d} + (C-O)) &= r^2 \\ t^2\vec{d}\cdot\vec{d} - 2t\vec{d}\cdot(C-O) + (C-O)\cdot(C-O) - r^2 &= 0 \end{aligned}$$

Looking closely, this is a quadratic equation with t as the variable, so in order to solve for t we must use the quadratic equation:

$$a = \vec{d}\cdot\vec{d}, \;\; b = -2\vec{d}\cdot(C-O), \;\; c = (C-O)\cdot(C-O) - r^2$$

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

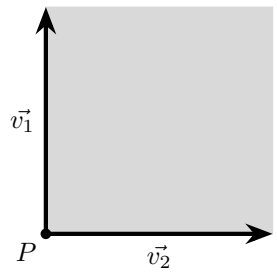
In order for there to be a valid t-value where the Ray intersects the sphere, the discriminant, $b^2 - 4ac$, must be positive so that its square root may be taken. Thus we can calculate if a sphere intersects a ray, and furthermore at what point P in space the Ray intersects the sphere, using the quadratic equation. One shortcut that can be taken is making a new variable, $h = \frac{-b}{2}$. This would make $h = \vec{d} \cdot (C - O)$. Adjusting the quadratic equation appropriately, we would get the following:

$$t = \frac{h \pm \sqrt{h^2 - ac}}{a}$$

This serves to save some calculation time in a process that requires us to calculate this thousands of times. As for how the scattered ray is calculated, that requires the normal of the sphere, which is easy on a sphere, seeing as it is just the vector from the center to the point of intersection.

## 4.2 Quad (Rectangle)

A bounded plane/rectangle is defined in this case by three things; a point, and two defining vectors:




we can calculate its interseciton using the equation for a plane: $Ax + By + Cz = D$. If [A,B,C] is the normal vector to the plane, we can easily find it using the cross product of $v_1$ and $v_2$. [x,y,z] is just a position vector, and since we are shooting our rays from the origin, we just have to check if there's a point along the ray that satisfies the equation.

$$Ax + By + Cz = D; \quad p = (x, y, z)$$

$$(n \cdot p) = D, \quad p = O + t\vec{d}$$

$$(n \cdot (O + t\vec{d})) = D$$

$$(n \cdot O) + (n \cdot t\vec{d}) = D$$

$$t = \frac{D - (n \cdot O)}{(n \cdot \vec{d})}$$

This checks if the ray intersects the plane, and with the t value we can find the point it intersects the plane at, but we still have to verify if it actually falls within the bounds of the rectangle. To do this we have to imagine a vector $\vec{v}$ from P in the definition of the rectangle and the point where the ray intersects the plane. This vector's x and y values must be defined by the plane.

Let P be point defined by rectangle, and p be the point of intersection:

$$\vec{v} = (P - p)$$

Looking at the diagram above, the horizontal part of $\vec{v}$ has to be shorter than the length of $\vec{v2}$, and vertical part shorter than the length of $\vec{v1}$:

$$\|v\| cos(\theta_1) < \|v_1\| \ \ \& \ \ \|v\| cos(\theta_2) < \|v_2\|$$

$$\|v_1\| * \|v\| cos(\theta_1) < \|v_1\| \|v_1\| \ \ \& \ \ \|v_2\| * \|v\| cos(\theta_2) < \|v_2\| \|v_2\|$$

$$(\vec{v_1} \cdot \vec{v}) < (\vec{v_1} \cdot \vec{v_1}) \ \ \& \ \ (\vec{v_2} \cdot \vec{v}) < (\vec{v_2} \cdot \vec{v_2})$$

Therefore, if both of the above are true, then the point $p$ is within the area of the rectangle defined by $P$, $\vec{v_1}$, and $\vec{v_2}$. Thus we can determine if a ray intersects a rectangle, and colour the pixels accordingly. As for how the resulting rays are scattered, it is identical to spheres, except the normal is calculated using the cross product of $\vec{v_1}$ and $\vec{v_2}$.

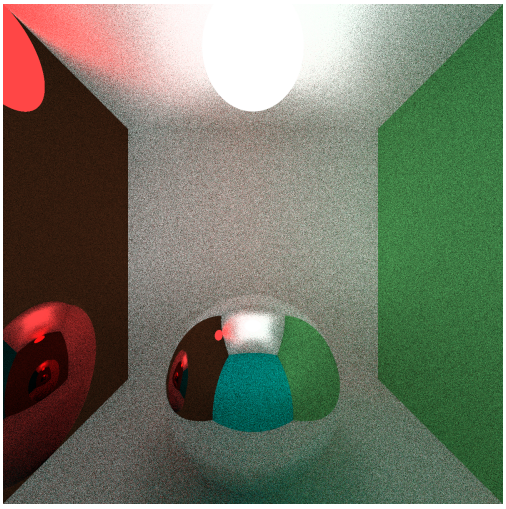
# 5 Noise

Noise is a biproduct of having many random bounces, since every bounce there could be a number of different random resulting paths leading to the possibility of each ray returning very different colour values. That's why very noisy images can sometimes be produced. The goal for a ray tracer is to find the blend between the strongest/most likely to hit colours. This can be achieved by layering many renders of the same image over each other, i.e. adding up all the colour values for each pixel and dividing by the amount of renders.

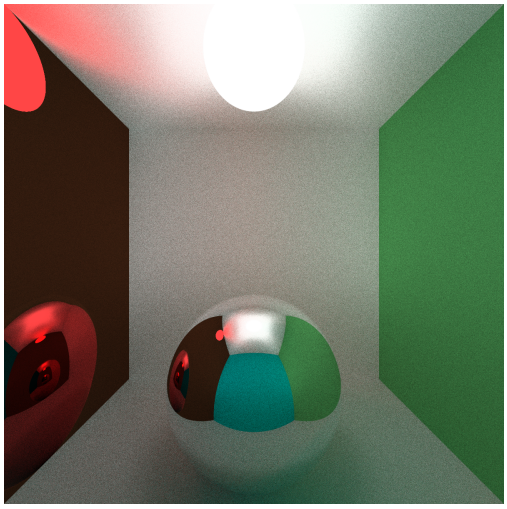
```
num_of_renders = 1 #number of renders to average
rend = np.zeros((height, width, 3))
tic = 0
for i in range(num_of_renders): #averages multiple renders to reduce noise
    im = render()
    rend += im
    w = rend/(i+1)
    plt.imsave('image.png', w) #saves image after each render
    print('Render', i+1, 'done')
    tic +=1
    print('Time elapsed:', (time.time()-start)/60, 'minutes')
    if time.time() - start > time_limit:
        print('Time limit reached')
        break
rend /= tic
plt.imsave('image.png', rend) #saves final image
print(f'{tic} renders done')
```

Figure 8: multiple renders implementation

The problem with multiple renders is the drastically increased total render time, with seven renders on the scene below taking 71 hours total to complete.



(a) one render



(b) seven renders

Figure 9: single vs. multiple renders

Although it took considerably more time, there is a pretty visible difference between one render and seven.

I would've liked to do more renders but the time was simply too absurd due to the inefficiency of the whole program running on a single cpu core.

# 6 What I Would Do Differently And Conclusion

There are several things I would do differently if I were to attempt the same project again. Firstly, I would look into hardware acceleration, specifically using the GPU, and also the possibility of multi core processing. As it stands, the thing I am most disappointed with is the absurd amount of time it requires to produce a usable image, although this was just a way to learn the basics not a real attempt at an extremely efficient ray tracer. I'm also disappointed with the amount of noise that remains even after multiple renders, so one think I might also look into is possible blending techniques and other solutions. I looked into bilateral filtering but it didn't help much. Lastly, base python with only matplotlib and numpy is not the ideal environment to create a program like this, but it works and helps easily understand the core concepts, which was the main goal.