Ray Tracing without NumPy

Utilities.py

I do not see any improvements which can be done to this class.

Ray.py

In the at function a tuple can be used since it improves code readability. To make use of it replace the following line (line 8): return self.origin + t \* self.direction by the below line: return tuple(a + t \* b for a, b in zip(self.origin, self.direction)) The zip function is basically equating a as self.origin and b as self.direction.

In the copy function to store a copy of the attributes of ray 2 the copy function can be used. This is recommended because when it is being referred to directly any changes made to the original data in ray 2 will also effect the data stored inside the copy when that is not the intention as what you really want to do is to store a copy of the data at that moment and this data cannot be changed. To make use of this function you first need to import the copy function using the line: import copy in the first line. Then you need to replace the line (line 11): self.origin = ray2.origin and the line (line 12): self.direction = ray2.direction with the lines self.origin = copy.copy(ray2.origin) and self.direction = copy.copy(ray2.direction) respectively. In this way the attributes self.origin and self.direction are independent are not dependent on the attributes inside ray2 but the attributes of the Ray instance are independent. This means that if anything inside ray2 is changed, those changes will not take effect on ray.

Interval.py

In the clamp function (lines 14-19) it is being checked if the value of x is greater than self.\_t\_min and smaller than self.\_t\_max. If it falls in that range it returns x but if it is smaller than self.t\_min it returns self.t\_min and if it is larger than self.t\_max it returns self.\_t\_max. This is done using a lot of if statements which does not make the code concise and makes code repetitive. Rather than using a lot of if statements the min and max functions can be used. These simplify the function and reduces the code inside the function to just 1 line rather than 5. This is the way the function currently looks like:

def clamp(self, x):

if x < self.t\_min:

return self.t\_min

if x > self.t\_max:

return self.t\_max

return x

This is how it will look like after the min and max functions are used:

def clamp(self, x):

return max(self.t\_min, min(x, self.t\_max))

In this way it first measures the smallest value between x and self.t\_max and it then measures the largest value between self.t\_min and the result from the previous operation. This will always work because if x is smaller than self.t\_max the result will be self.t\_max from the first operation and also from the second since self.t\_max will always be larger than self.t\_max and vice versa if x is smaller than self.t\_min. While if it falls in the range, you will get x from the first operation and x from the second operation.

Vec3.py

Just like in the Ray.py class we can make use of the copy function to make a copy of v2 rather than manually referring to it. To do this we replace the lines:

self.e[0] = v2.e[0] (line 114)

self.e[1] = v2.e[1] (line 115)

self.e[2] = v2.e[2] (line 116)

with the lines:

self.e[0] = copy.copy(v2.e[0])

self.e[1] =copy.copy(v2.e[1])

self.e[2] = copy.copy(v2.e[2])

Apart from this I see nothing wrong with the code apart from the fact that NumPy can be used for various mathematical operations but since this version is without using NumPy it cannot be implemented.

Hittable.py

Rather than creating a new hit object every time the loop iterates, we can simply make use of the same rec object which was previously passed as a parameter of the function hit. This can be done by changing the line: temp\_rec = HitRecord() (line 42) to temp\_rec = rec. We can also remove the lines self.hit\_anything = False (line 43) and self.closest\_so\_far = ray\_t.t\_max (line 44) since they were already initialized in the constructor \_\_init\_\_. If they are not removed, they will be initialized again every time the function is called.

When making use of the copy function inside the hit function there is no need to specify that rec2=temp\_rec inside the for loop (line 50) but you can simply write temp\_rec only. In other words: rec.copy(rec2=temp\_rec) becomes rec.copy(temp\_rec).

Material.py

The decorator @abstractclassmethod in line 9 should be changed to @abstractmethod since that is the correct decorator to be used.

Just like the parameters r\_in and rec were annotated with Ray and HitRecord respectively even the parameter scattered can be annotated with Ray since when saying scattered you are referring to a ray hence it is most probably referring to a ray.

Metal.py

Rather than using an if statement to check if fuzz should be returned or 1 should be returned in the \_\_init\_\_ function in line 8 the min function can be used. To do so replace: self.fuzz = fuzz if fuzz < 1 else 1 with self.fuzz = min(fuzz, 1)

Lambertian.py

You can make the code shorter by implementing the if statement from lines 14-15 in 1 line by replacing the below part:

if (scatter\_direction.near\_zero()):

scatter\_direction = rec.normal

with just this 1 line:

scatter\_direction = rec.normal if scatter\_direction.near\_zero() else scatter\_direction

Sphere.py

An optimization to this class which can be done is that of calculating the square of the radius beforehand inside the \_\_init\_\_ function and storing it inside a variable. For this to work you need to add this line inside the \_\_init\_\_ function self.radius\_squared = radius\*\*2 so this function will not look like this:

def \_\_init\_\_(self, center, radius, mat):

self.center = center

self.radius = radius

self.mat = mat

self.radius\_squared = radius\*\*2

After this is done the line: c = oc.length\_squared() - self.radius\*\*2 (line 14) needs to be edited. We can remove the self.radius\*\*2 part and replace it with self.radius\_squared so the new line will now be this: c = oc.length\_squared() - self.radius\_squared.

A benefit of doing so is that in this way the square of the radius will only be calculated once and saved as opposed to having to calculate it every time it is used.

Dielectric.py

When calculating the value of r0 (lines 14-15) 2 lines of code are used which can be easily fitted into 1 lines of code making the code shorter by removing the below lines of code:

r0 = (1 - ref\_idx) / (1 + ref\_idx) (line 14)

r0 = r0 \* r0 (line 15)

and combining them into 1 line of code like this: r0 = ((1 - ref\_idx) / (1 + ref\_idx))\*\*2.

Camera.py

I do not see any improvements which can be done to this class.

main.Py

Rather than adding spheres one by one (lines 17-21), list comprehension can be used to add them all at once. The below code is removed:

world.add(Sphere(Point3(0.0, -100.5, -1.0), 100.0, material\_ground)) (line 17)

world.add(Sphere(Point3(0.0, 0.0, -1.0), 0.5, material\_center)) (line 18)

world.add(Sphere(Point3(-1.0, 0.0, -1.0), 0.5, material\_left)) (line 19)

world.add(Sphere(Point3(-1.0, 0.0, -1.0), -0.4, material\_left)) (line 20)

world.add(Sphere(Point3(1.0, 0.0, -1.0), 0.5, material\_right)) (line 21)

and it is replaced by the following code:

spheres = [Sphere(Point3(0.0, 0.0, -1.0), 0.5, material\_center), Sphere(Point3(-1.0, 0.0, -1.0), 0.5, material\_left), Sphere(Point3(-1.0, 0.0, -1.0), -0.4, material\_left), Sphere(Point3(1.0, 0.0, -1.0), 0.5, material\_right)]

world.add(\*spheres)

The star in the final line basically unpacks the list and adds every element one by one.