RayTracer Tech Note

Ari Sloss

July 2020

1 Introduction

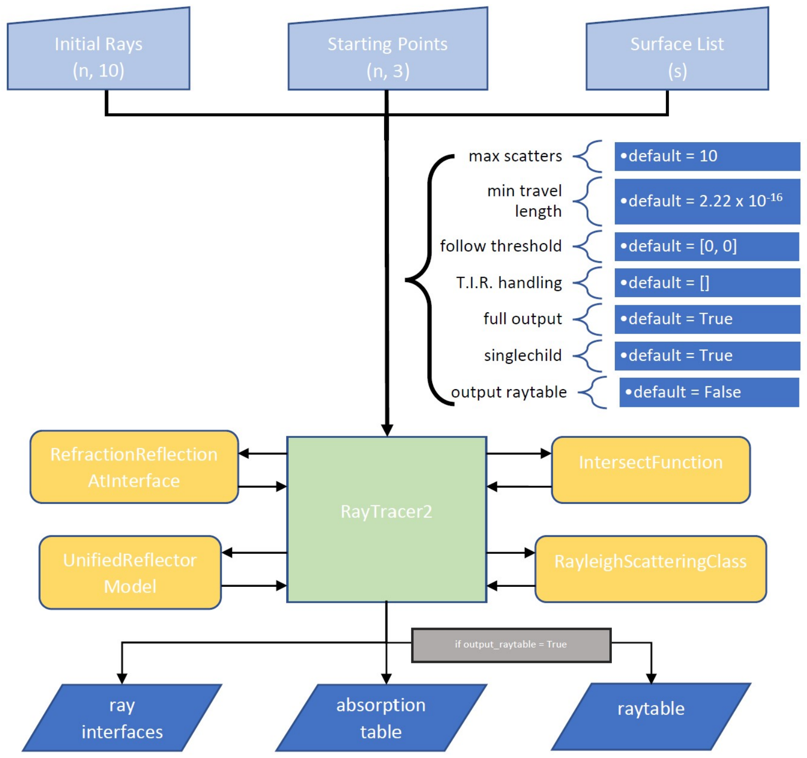
This ray tracing software is a suite consisting of the core engine - RayTracer2 - and accompanying modules for calculation and data storage, ported from MATLAB to Python. It requires the user to define a set of mathematical geometries and initial light rays, then calculates the paths of those rays as they propagate through space and are influenced by the geometries.

2 Running RayTracer2

A separate 'main' must be written to instantiate geometries and rays, call RayTracer2, and handle output data. Currently, this is done inside 'TestGeometry.py'.
RayTracer2 requires three input values, in the form of numpy arrays:

- $n \times 3$ array of ray starting points in Cartesian coordinates
- $n \times 10$ array of light rays giving the initial direction, intensity, and polarization of the light
- s-long array of surface geometries ('surface' objects)

where n is the number of rays. RayTracer2 can also take in seven additional input parameters to control such things as the maximum number of scatters or if to output a complete ray history. These are assigned defaults and do not need to be passed values in order to run RayTracer2. More detail about inputs can be found in the documentation for 'RayTracer2.py'.



The following files are required to be in the same file directory:

- RayTracer2
- RefractionReflectionAtInterface
- IntersectFunction
- rayInterfaces
- UnifiedReflectorModel
- surface
- RayleighScatteringClass
- finalRays (?)
- RayToShape
- All relevant RayToXXX functions (ex. RayToCylinder)
- File with your 'main' function

RayTracer2 and its modules can be treated as a black box, as all of the ray propagation calculations and physics are handled internally. The engine will output three numpy arrays:

- ray_interfaces - $k \times n$ array of rayInterfaces objects giving many details about each ray scatter
- absorption_table - $k \times 5 \times s \times 2$ array detailing absorption at each scatter
- raytable - $(k + 1) \times n \times 13$ array detailing the paths of each ray (not populated by default)

where k is the number of scatters. Again, more information can be found in the documentation.
Your 'main.py' file should look something like this:

```
import numpy as np

def main():
    # Define your ray starting points
    ray_startpoints = np.empty((n, 3)) # (x, y, z)
    # Assign values...

    # Define your starting rays
    rays = np.empty((n, 10))
    # Look at RayTracer2.py documentation for details

    # Define your surface geometries
    surface_list = []
    example_surface = surface.surface()
    # Assign 'surface' object parameters -- more detail in next section
    surface_list.append(example_surface)

    #Call RayTracer2
    [ray_interfaces, absorption_table, raytable] = \
    RayTracer2.RayTracer2(starting_points, rays, surface_list)

if __name__ == "__main__":
    main()
```

3 Making Surfaces

Each surface is stored as a 'surface' object, with the following properties:

- description - short description of the geometry
- n_outside - the index of refraction outside the surface (in the case of a plane, 'outside' is in the direction of the normal vector)
- n_inside - the index of refraction inside the surface
- surface_type - either 'normal', 'diffuse', 'retro', or 'unified' (see 'surface.py' documentation)
- absorption - a float from 0 to 1 defining the fraction of energy absorbed by the surface – 1 is a perfect absorber
- shape - defines what type of shape the surface is; currently a string
- param_list - Python list detailing the primitive surface
- inbounds_function - lambda function detailing the surface boundaries

These will need to be defined for every surface. 'shape', 'param_list', and 'inbounds_function' define the surface in space. Every surface is based on a primitive form of its shape. For example, a cylinder of finite length is specified from an infinite cylinder consisting of a point along its central axis, an orientation vector, and a radius. 'shape' simply tells 'RayToShape' which 'RayToXXX' function to call, and can currently be "cylinder", "plane", "sphere", "torus", or "quadsurface" (only 'cylinder', 'plane', and 'sphere' have been tested in Python). These will later be changed from strings to ints.

'param_list' defines this primitive shape. It will look different for each shape type, but must be a list of numpy arrays. For a cylinder, the first value is a point on the central axis as a 1×3 numpy array, the second is a vector giving the direction of the central axis as a 1×3 numpy array, and the last is the radius in cm as a float.

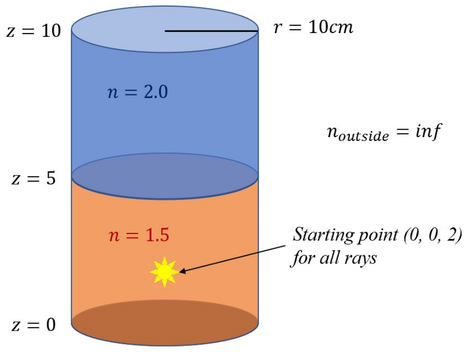
'inbounds_function' uses a lambda function on a numpy array to limit the primitive, infinite shape to the finite surface of interest. On the next page are examples from 'TestGeometry.py'. *Note:* **not** *the complete geometry*

```
bot_cyl = surface.surface()
bot_cyl.description = 'bottom cylinder along z-axis,\
    10cm radius from z=0 to z=5'
bot_cyl.shape = 'cylinder'
bot_cyl.param_list = [np.array([0, 0, 0]), np.array([0, 0, 1]), 10]
bot_cyl.inbounds_function = lambda p: np.reshape((p[:, 2, :] > 0) * \
    (p[:, 2, :] < 5), (np.size(p, 0), -1))
bot_cyl.n_outside = np.inf
bot_cyl.n_inside = 1.5
bot_cyl.surface_type = 'normal'
bot_cyl.absorption = 0

mid = surface.surface()
mid.description = 'middle disk centered on z-axis \
    with radius 10 and z=5'
mid.shape = 'plane'
mid.param_list = [np.array([0, 0, 5]), np.array([0, 0, 1])]
mid.inbounds_function = lambda p: np.reshape((p[:, 0] ** 2 + \
    p[:, 1] ** 2 < 100), (np.size(p, 0), -1))
mid.n_outside = 2
mid.n_inside = 1.5
mid.surface_type = 'normal'
mid.absorption = 0
```

The test geometry details a five surface cylinder with caps on each end and a disk in the middle; the bottom half of the cylinder, between the bottom and middle disks, has an index of refraction of 1.5 while the top half has an index of refraction of 2.0. This is why the two halves are defined as two separate cylinders, because each surface is passed a different value for 'n_inside'. The middle disk is necessary as a transition between mediums of different refractive indices. The top and bottom disks are perfect absorbers, the rest are perfect reflectors. RayTracer2 and the functions it calls have been debugged with this test geometry and shown to function properly.

4 Test Geometry Results

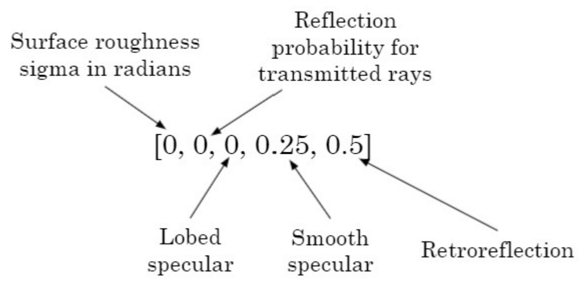


WIP

It works!

5 UnifiedReflectorModel

The 'UnifiedReflectorModel' is so called because it combines normal, diffuse, and retroreflection into one surface model. With it, one can represent any kind of reflecting/refracting surface.


Surface roughness sigma in radians
Reflection probability for transmitted rays
[0, 0, 0, 0.25, 0.5]
Lobed specular
Smooth specular
Retroreflection

A 'unified' surface is specified with `surface.surface_type = 'unified'` and `surface.unified_params = [x, y, z, v, w]`, with values as described above.

- x = surface roughness
- y = probability (as a decimal) for rays transmitted through the surface to reflect back out
- z = probability of lobed specular reflection
- v = probability of smooth specular reflection
- w = probability of retroreflection

The three probabilities z, v, and w must at most add to 1. If they do not, the remaining difference is the probability of normal refraction and reflection. Once the reflective properties of your material surface are known and translated into these parameters, your surface can be modeled.

[**Need more info on surface roughness from Eric**]

6 Handling RayTracer2 Output

WIP

See RayTracer2 documentation for output array specifics.

7 Lessons Learned and Useful Info

Tips:

- Use the numpy version of everything; arrays used as inputs are usually converted into numpy arrays, for example, but creating your arrays as numpy arrays instead of Python lists will be easier and allow for greater flexibility. Similarly, use numpy functions for operations on arrays.

- For debugging, I found printing the shapes of various input, output, and intermediate arrays to be very useful.

  ```
  *array*.shape
  ```

  is your friend!

- In numpy, arrays – or in this case, vectors – with 1 dimension have the shape (x,). These are treated by numpy as having shape (1,x) when used in array operations, which will give different results than may be expected or a ValueError because the expected shapes are not broadcastable. The solution is to add a dimension in order to make the shape (x,1). Using numpy indexing,

  ```
  *array*[:, np.newaxis] or *array*[:, None]
  ```

  will solve this problem, but I recommend using 'np.newaxis' because it is more explicit than 'None' (they are, however, the same thing).