Creating a Scene

This short document is meant to tell you how to describe and render a scene for a ray tracer program written in Dyalog APL. The program is based on the ray tracer described in the book **THE RAY TRACER CHALLENGE, A TEST-DRIVEN GUIDE TO YOUR FIRST 3D RENDERER** by Jamis Buck, available from The Pragmatic Bookshelf. For a detailed understanding of the program and how it works, buy the book!

https://pragprog.com/book/jbtracer/the-ray-tracer-challenge

I'll try to show you how you can create your own scenes using this program. For more details on my own implementation have a look at the Jupyter notebook `RayTracer-Code.ipynb` for the APL code and `RayTracer-Demo.ipynb` for the demo scenes.

Running the Demos

I assume you already have the Dyalog APL interpreter installed. If not, go to

https://www.dyalog.com/download-zone.htm

and register for a personal, non-commercial license (or licence, as they say). You'll find plenty of documentation on their site. I heartily recommend **MASTERING DYALOG APL** by Bernard Legrand, available as a PDF download from here.

Now take the SALT files `rtcode_ws.dyalog` and `rtdemo_ws.dyalog` from this repository and load them into your workspace and set the search path with the commands

```
]load 'path..to../rtcode_ws'
]load 'path..to../rtdemo_ws'
⎕PATH←'# #.RayTracer'
```

You may want to save your workspace under a convenient name at this point so you can return to the code in the future. If you do a `)objects` system command you'll see the two namespaces `RayTracer` and `RTDemo`. Within the `RTDemo` namespace are namespaces for each of the demo scenes.

Let's start with a simple scene that will render quickly. The Chapter 7 sample scene for Sphere objects can be accessed and rendered with the following commands

```
)cs #.RTDemo.Chapter07
)vars
c       floor   left    left_wall       m       middle  right
right_wall      t       w
```

The important variables to note are c (camera) and w (world). The same variables will be in all of the other demo namespaces, and they are all you need for now to render scenes. To render a scene and output the result as a PPM graphic file, use the following commands. The same commands, with an appropriate change to the output file name, can be used in all the other demo namespaces.

```
im← w render c
ppm← canvas_to_ppm im
'Chapter07.ppm' savePPM ppm
```

Have a look at your output image file with GIMP or other image viewer. Since PPM files are rather large, you may also want to convert your result to a PNG or JPG file, then delete the PPM file. If you plan on saving your workspace later, you'll proably want to clean things up by removing the rather large intermediate variables created during rendering using the erase command before moving to another namespace

```
)erase im ppm
```

Once you've played around and viewed some of the images, have a look at the Jupyter notebook file `RayTracer-Demo.ipynb` to see how the various scenes were constructed.

Objects

There are five main objects defined in this ray tracer; Sphere, Plane, Cube, Cylinder, and Cone. Every object has in common a *Transformation Matrix* (and its inverse), a *Material* specification, and a Boolean *Shadow* attribute. Some objects have additional attributes. For example, a Cylinder by default is open at each end, but can also be specified as having a cap on the end instead.

The Transformation Matrix defines how an object is scaled, rotated and translated within the ray tracer world space. More on this topic will be found in the Positioning section. A Material specification, described in the next section, defines an object's color, pattern, reflectivity, and so on. The Shadow attribute indicates whether or not the object casts a shadow. Not casting a shadow is useful if modeling the surface of a pool of water as reflective and transparent, but not casting a shadow on the bottom of the pool.

An object is represented by a nested vector and individual object attributes are addressed by an index value. The two that will be of common use are illustrated below. Note also how each object is created.

```
ball← sphere
floor← plane
box← cube
ball[#.RayTracer.obj_material]← matl
wall[#.RayTracer.obj_shadow]← 0
```

By default a Sphere has a radius of one, centered at the origin of ray tracer space. A Plane extends to infinity in the Y-Z axes. A cube has sides of length one, also centered at the origin. The Transformation Matrix, detailed in the Positioning section, can be used to scale a Sphere up or down, rotate a Plane so that it becomes a wall, or translate a Cube to some other point in ray tracer space.

Cylinder and Cone

A cylinder by default has a radius of one, is centered at the origin, and extends to infinity along the Y axis. Three additional attributes can modify the default appearance. The *Maximum* and *Minimum* attributes define the endpoints of the cylinder, while the Boolean *Closed* attribute indicates whether or not the ends of the cylinder are capped.

A Cone is also centered at the origin and it extends upward and downward to infinity along the Y axis. At a unit distance above and below the origin the cone has widened to a radius of one. The Cone has the same three additional attributes as the Cylinder, namely *Maximum*, *Minimum*, and *Closed*. An example of creating each object and modifying their attributes is shown here

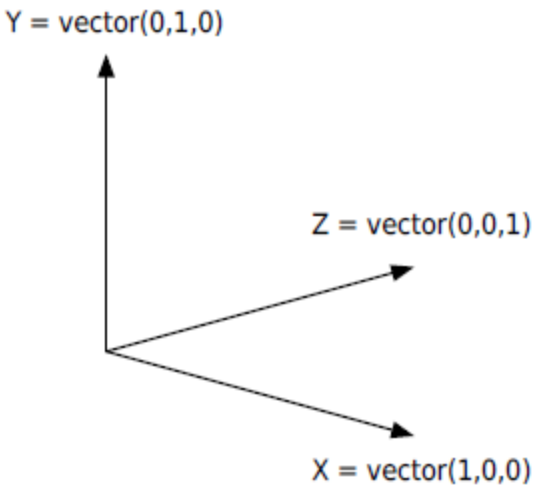
```
cyl1← cylinder
cone1← cone
cyl1[#.RayTracer.obj_material]← matl
cyl1[#.RayTracer.cylinder_maximum]← 5
cyl1[#.RayTracer.cylinder_minimum]← 2
cone1[#.RayTracer.cone_maximum]← 4
cone1[#.RayTracer.cone_minimum]← 0
cone1[#.RayTracer.cone_closed]← 1
```

Color and Materials

Patterns

Positioning

Once you've established the objects in a scene and their attributes you'll need to position them within the world space. The ray tracer uses the left-hand rule for defining axes and direction (look it up if you're not already familiar). The graphic below shows how the ray tracer will interpret coordinates.



The World

The Camera

Rendering

We've already seen how to render a scene in the introduction. The basic commands for the demo of Chapter 7 that we started with were

```
im← w render c
ppm← canvas_to_ppm im
'Chapter07.ppm' savePPM ppm
```

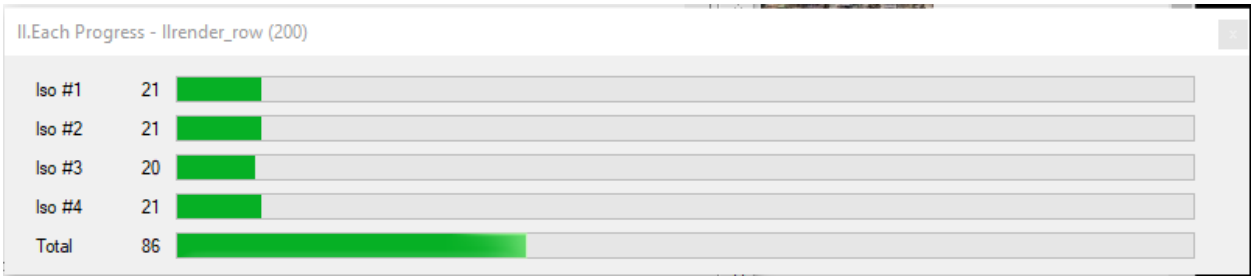
There are two additional rendering commands, one which renders a scene in parallel on multiple CPUs and one that renders a scene to a graphic window. The parallel renderer will automatically spawn a process for each CPU and then give each process a row to render until all of the rows in a canvas have been processed. On a four CPU machine the result is a speedup of about 2.5 times. The commands for parallel rendering using the **llrender** function is

```
im← w llrender c
ppm← canvas_to_ppm im
'Chapter07.ppm' savePPM ppm
```

To use the command you will need to load the necessary isolate workspace using

```
)CS #
)COPY isolate
```

A speedbar will appear containing a lane for each CPU and one for total scene rows rendered.



The graphic window render function is **wrender** and is used in the same manner as the render function, only it does not return a result. The command is simply

```
w wrender c
```

A window will appear, as shown below, and the scene will be rendered row by row from the top down. The Quit button will terminate any rendering in process and close the window.