# 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. An object is a vector with at least five elements whose positional components are

| Index | Variable Reference | Range | Description |
|---|---|---|---|
| 1 | obj_tag | integer | Unique ID for each type of object |
| 2 | obj_transform | matrix | Transformation matrix |
| 3 | obj_inverse | matrix | Inverse of transformation matrix |
| 4 | obj_material | enclosed | Material specification for object |
| 5 | obj_shadow | 0 or 1 | Boolean, the object casts a shadow or not |

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
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 extending one unit in each axis direction, 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. Additional vector positional components are

| Index | Variable Reference | Range | Description |
|---|---|---|---|
| 6 | cylinder_minimum/cone_minimum | scalar | Bottom distance from origin, default -infinity |
| 7 | cylinder_maximum/cone_maximum | scalar | Top distance from origin, default infinity |
| 8 | cylinder_closed/cone_closed | 0 or 1 | Boolean, endcaps open (default) or closed |

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.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

A **color** is defined by a three element vector of RGB values between 0.0 and 1.0 inclusive. Hence Black would be 0 0 0 and White would be 1 1 1. There are many sources of predefined colors that you can use,

including the POV-ray ray tracer. Color definitions from that program are included in the Attributes namespace, which can be loaded from the rtattrib_ws.dyalog SALT file.

A **material** is defined as a vector with the following positional components

| Index | Variable Reference | Range | Description |
|---|---|---|---|
| 1 | material_color | enclosed | Base color of object |
| 2 | material_ambient | 0.0 to 1.0 | Degree of reflection from background light |
| 3 | material_diffuse | 0.0 to 1.0 | Reflection from light source based on surface angle |
| 4 | material_specular | 0.0 to 1.0 | Reflection from light source resulting in spot on surface |
| 5 | material_shininess | 10 to 200 | Default 200, no upper limit in practice |
| 1 | material_pattern | enclosed | Object is colored by a pattern |
| 6 | material_reflective | 0.0 to 1.0 | Material reflectivity from none (0) to fully (1) |
| 7 | material_refractive | 0.0 to 1.0 | Refractive index (water 1.33, glass 1.52) |
| 8 | material_transparency | 0.0 to 1.0 | From opaque (0) to clear (1) |

Consult the book or check out the various demo scenes to see some effects that can be achieved with the material specification!

## Patterns

A pattern is a vector with two colors and a Transformation Matrix. The vector positional components are

| Index | Variable Reference | Range | Description |
|---|---|---|---|
| 1 | pattern_type | integer | Unique ID for each type of pattern |
| 2 | pattern_transform | matrix | Transformation matrix |
| 3 | pattern_inverse | matrix | Inverse of transformation matrix |
| 4 | pattern_color1 | enclosed | Color or Pattern vector |
| 5 | pattern_color2 | enclosed | Color or Pattern vector |

Patterns establish a repetitive surface coloring. The basic patterns are

- *Stripe*, which alternates between two colors in the x-axis direction
- *Gradient*, which smoothly transitions from one color to the other in the x-axis direction
- *Ring*, which alternates between two colors moving out from the x- or z-axis
- *Checker*, which alternates between two colored squares in the x-z plane
- *Radial* Gradient, which combines the Ring and Gradient effect
- *Blend*, which takes two patterns and computes the average color each contributes
- *Perlin*, which produces a textured noise distribution of two colors
- *Perlin Gradient*, which produces a gradient textured noise distribution of two colors

All patterns accept two colors as arguments, but can also accept a pattern as argument as well. The Blend pattern is meant to take two patterns, since a blend of two colors would just be a uniform color. Patterns also accept a Transformation Matrix so that their default orientation can be changed. Striped

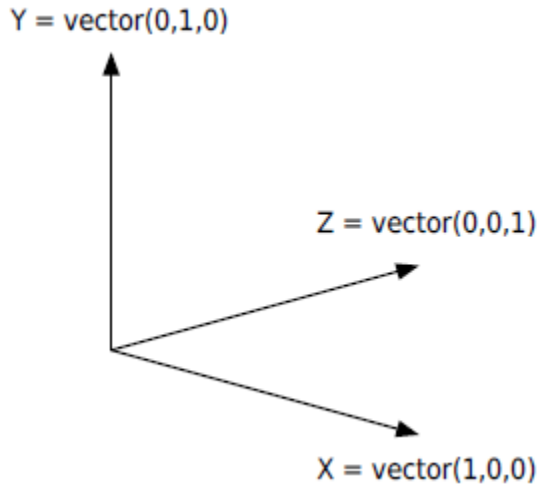
normally alternate along the x-axis, changing from one color to the other at each unit distance travelled. To alternate along the y-axis, changing four times each unit distance, you can apply the transformation matrix composed of scaling and rotation with the function

```
tm← (rotation_z ∘ 0.25) +.× scaling 0.25 1 1
obj← obj add pattern_transform tm
```

Scaling is more interesting for two-dimensional patterns like Ring

## 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.




A Point in space is a four element vector consisting of the (x, y, z) as the first three elements, while a Vector in space specifying the proportional relationship of the vector in the x-, y-, and z-axis direction. Each are constructed by the point and vector function, as in

```
p← point 1 5 ¯4
v← vector 1 1 0
```

An object is placed within the coordinate system using a *Transform Matrix* within the object. The default transform matrix is the identity matrix, which leaves the object at the origin and with default dimensions. An object's transform matrix is an arbitrary number of individual transform matrices multipled together, selected from the following

| Transform | Function | Description |
|---|---|---|
| Translation | `translation x y z` | Move object center in each axis direction |
| Scaling | `scaling x y z` | Scale object proportion in each axis direction |

| | | |
|---|---|---|
| Rotation X | `rotation_x angle` | Rotate object about the x-axis, angle in radians |
| Rotation Y | `rotation_x angle` | Rotate object about the y-axis, angle in radians |
| Rotation Z | `rotation_x angle` | Rotate object about the z-axis, angle in radians |
| Shearing | `shearing xy xz yx yz zx zy` | Skew object dimension in proportion to others |

Translation is the easiest transform to understand. To move a unit sphere from the origin to where it is centered at the point (5, 1, 3) use the translation function to generate the transform matrix with arguments 5 1 3. If you would also like to make the sphere twice as big, use the scaling function with equal values of 2 along the x, y, and z axes. Combine the matrix output of the two transform functions by using the inner product. The result would be written as

```
(translate 5 1 3) +.× scaling 2 2 2
```

Note, since APL evaluates an expression from right to left, the object is first scaled by the indicated amount and then translated to a new origin point. Rotations about an axis are affected by object position. If rotation is the first transform applied then the object is rotated about its own origin. If the object is first translated a given distance along a perpendicular axis, then rotation will also move the object origin in space.

## The World

The World is just a two element vector containing the lighting and object elements of a scene. The first World element is a vector of light vectors, while the second element is a vector of objects. A light is defined as a four element vector containing its *Location* and *Color* along with two other extensions, *Radius* and *Iteration*. The positional components of a light are

| Index | Variable Reference | Range | Description |
|---|---|---|---|
| 1 | light_point | enclosed | Origin of light |
| 2 | light_color | enclosed | Light color |
| 3 | light_radius | scalar | Radius of area light, default 0 for point light |
| 4 | light_iteration | scalar | Number of iterations to compute shadow for area light |

A point light is the simplest computationally and should always be used when viewing a scene under construction. Real light sources don't cast sharp shadows, so an area light provides more realistic gradation at the shadow boundary. A larger radius creates a more diffuse light, and a larger number of iterations make the shadow more realistic. Area lights increase rendering time in proportion to number of iterations.

A sample World construction might be

```
w← world
w[1]← ⊂⊂ (point ¯9 9 ¯9) point_light 1 1 1
w[2]← ⊂  floor room ball1 ball2 ball3
```

## The Camera

A camera is a vector that defines the point of view for rendering a scene and the size of the result in horizonal and vertical pixels. Most of the elements are set or computed when the camera is created and the viewpoint specified. There is no real need to access the content of a camera except through the two

functions, **camera** and **view_transform**, that construct a result. The **camera** function accepts the horizontal and vertical dimensions in pixels of the rendering canvas along with its field of view in radians. The **view_transform** function adds a transformation matrix to the camera which defines the origin and focal point of the camera. This function accepts the camera origin (*From*) the point of focus (*To*) and the direction of "Up" in the scene (*Up*). an example camera might be

```
c← camera 800 600 1.047
vt← view_transform (point 1 2 ¯5) (point 0 1 0) (vector 0 1 0)
c[#.RayTracer.camera_transform]← vt
c[#.RayTracer.camera_inverse]← ⌹vt
```

## 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
```

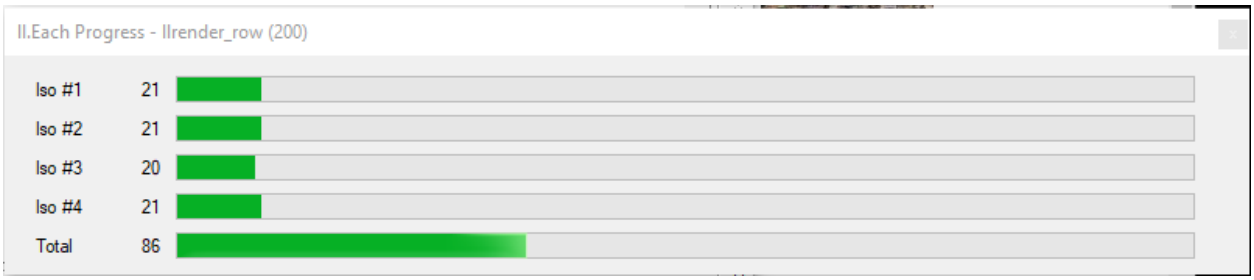
There are two additional rendering commands, one which renders a scene in parallel on multiple CPUs and one that renders a scene to a graphic window. The parallel renderer will automatically spawn a process for each CPU and then give each process a row to render until all of the rows in a canvas have 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.