Ray tracing introduction

The objective of a ray tracing program is to render a photo-realistic image of a virtual scene in 3 dimensional space. There are two major components in the process:

The virtual camera

1 - *The viewpoint*  
This is the location in 3-d space at which the viewer of the scene is located

2 - *The screen*  
This defines a virtual window through which the viewer observes the scene. The window can be viewed as a discrete 2-D pixel array (image). The *raytracing* procedure computes the color of each pixel. When all pixels have been computed, the image is written out as a .ppm file

The scene to be viewed

3 - *materials*  
One or more material definitions may be associated with each object. The material definition describes how the object interacts with a light. Among other things the material definition defines the color of the object.

4 - *light sources*  
Lights themselves are *not* visible, but they do illuminate objects and may be subject to shadowing. Lights may be white or colored.

5 – *visible objects*  
Reflective objects that are illuminated by the light sources
World and window coordinate systems

Two coordinate systems will be involved and it will be necessary to map between them:

1 - Window coordinates
   the coordinates of individual pixels in the virtual window. These are two dimensional (x, y) integer numbers. For example, if a 400 col x 300 row image is being created, the window x coordinates range from 0 to 399 and the window y coordinates range from 0 to 299. In the raytracing algorithm a ray will be fired through each pixel in the window. The color of the pixel will be determined by the color of the object(s) the ray hits.

2 - World coordinates
   the “natural” coordinates of the scene measured in feet/meters etc. Since world coordinates describe the entire scene these coordinates are three dimensional (x, y, z) floating point numbers.

For the sake of simplicity we will assume that

- the screen lies in the \( z = 0.0 \) plane
- the lower left corner of the window has world coordinates \((0.0, 0.0, 0.0)\)
- the lower left corner of the window has window (pixel) coordinates \((0, 0)\)
- the location of the viewpoint has a positive z coordinate
- all objects have negative z coordinates.
- lights may be located in either positive or negative z space.
Translating from pixel to world coordinates

**Problem:** Suppose the window is 640 pixels wide x 480 pixels high, and that the dimension of the window in world coordinates is 8 feet wide by 6 feet high. Find the world coordinates of the pixel at column 100 row 40.

**Possible Solution:** Compute the fraction or percentage of the complete x size that must be traversed to reach column 100. This value is 100/640 = 10 / 64 meaning column 100 is 10/64 of the way across the window. The x world coordinate of this location is therefore 10 / 64 of the total world distance across the window or (10/64)*8 = 10/8 = 1.25. Similarly the world y coordinate is (40 / 480 ) * 6 = (1 / 12) * 6 = 0.5.

A general formula for the procedure is thus:

\[
\text{world}_x = \text{world}_x \times \text{win}_x / (\text{win}_x)
\]

Thus the desired world coordinate is (1.25, 0.5, 0.0). (Recall the screen lies in the \(z = 0\) plane. Therefore the \(z\) world coordinate of every point in the window is 0.0).

**WARNING:** *Pixel dimensions are stored as integers.* You must ensure that the divisions shown above are done in floating point.
An alternative “world view”

If the above approach is used, then the pixel with x coordinate 0 clearly maps to world coordinate 0 as it apparently should. But if we are constructing a 640 pixel image, the maximum pixel coordinate is thus 639. And thus the corresponding world coordinate is:

\[ 8 \times 639 / 640 = 7.988 \text{ instead of } 8. \]

We can fix that by changing

\[ \text{world}_x = \text{world}_x \times \text{win}_x / (\text{win}_x - 1) \]

In this way pixel coordinate 0 maps to world coordinate 0 and pixel coordinate 639 maps to world coordinate 8. But then “nice” pixel coordinates such as 40 and 100 now map to really ugly numbers slightly larger than 1.25 and 0.5! Furthermore the image has no “center” pixel that maps to world coordinate (4.0, 3.0, 0.0)!

We can get back our “nice” numbers and our center pixel by using the above strategy but always making the image size 1 more than a “nice size” (e.g. 801 x 601). Since the computer doesn't really care whether a number is ugly or nice, we will use this formulation.

\[
\begin{align*}
\text{world}_x &= \text{world}_x \times \text{win}_x / (\text{win}_x - 1) \\
\text{world}_y &= \text{world}_y \times \text{win}_y / (\text{win}_y - 1)
\end{align*}
\]
Computing the direction of a ray

**Problem:** Suppose the viewpoint is at location (4, 3, 6) in world coordinates. Compute a unit length vector from the viewpoint through the pixel at column 100 row 40.

**Solution:** We saw above that the world coordinates of the pixel are (approximately): (1.25, 0.5, 0). From page three we know that two points in 3-D space implicitly determine a vector pointing from one to the other. Given two points $P$ and $Q$ in 3-D Euclidean space, the vector

$$R = P - Q = (p_x - q_x, p_y - q_y, p_z - q_z)$$

represents the direction from $Q$ to $P$. Therefore the vector from the viewpoint to the point on the window is $(point - viewpoint)$ or:

$$(1.25, 0.5, 0) - (4, 3, 6) =$$

$$(1.25 - 4, 0.5 - 3, 0 - 6) = (-2.75, -2.50, -6.00)$$

The length of this vector is 7.06 and so a unit length vector in this direction is:

$$(-0.39, -0.35, -0.85)$$

If you have computed the direction correctly the $z$ component of the vector will always be negative. A good plan is therefore to include the line:

```c
assert(direction->z < 0);
```

The assert facility will *abort your program* if the condition is FALSE and will print the module and line number where the problem happened.

You might be tempted to also do:

```c
assert(vec_len(direction)) == 1.0);
```

but because floating point arithmetic is imprecise, that would not be a good idea.
The raytracing algorithm

The complete algorithm for the first version of the raytracer is summarized below:

Phase 1: Initialization

- load the model description containing camera, material, object, and light definitions
- print the camera, material, object and light descriptions to the err file

Phase 2: The raytracing procedure for building the image

for each pixel in the window
{
    initialize the color of the pixel to (0.0, 0.0, 0.0)
    compute the direction in 3-d space of a ray from the viewpoint through the pixel
    identify the first (closest) object hit by the ray
    make a copy of the ambient color of the material associated with the object
    scale the copy of the ambient color by 1.0 / distance(from_viewpt, to_hitpt)
    add the scaled value to the color of the pixel.
    convert the d_rgb pixel to i_rgb and store it in the image.
}

Phase 3: Writing out the image as a .ppm file

write .ppm header to the output file
write the image to the output file
Example input file and image

camera cam1
{
    pixeldim    640 480
    worlddim   8 6
    viewpoint  4 3 6
}

material green
{
    ambient 0 5 0
}

material yellow
{
    diffuse 4 4 0
    ambient 5 4 0
    specular 1 1 1
}

plane leftwall
{
    material green
    normal 3 0 1
    point 0 0 0
}

plane rightwall
{
    material yellow
    normal -3 0 1
    point 8 0 0
}

material gray
{
    ambient 2 2 2
}

plane floor
{
    material gray
    normal 0 1 0
    point 0 -0.2 0
}
The output image

The output produced by the input file on the previous page is shown below. Visible image corruption near the green-gray boundary courtesy of JPEG compression.

The color gradient (which is what provides the “three-D” effect) is achieved by dividing the base ambient reflectivity of the object \((0.5, 0)\) by the distance from the view point to the location at which the ray hits the object. Pixels near the green – yellow boundary are more distant from the view point than those near the edges of the images.
We can push the point of intersection of the planes even farther into negative $z$-space by reducing the $z$ component of the normal from 1 to 0.1. When we do this, the floor triangle becomes larger, and the intersection of the two planes becomes indistinct.

```plaintext
plane leftwall
{
    material green
    normal 3 0 0.1
    point 0 0 0
}

plane rightwall
{
    material yellow
    normal -3 0 0.1
    point 8 0 0
}
```