Chapter 5

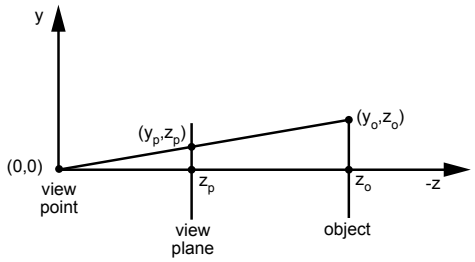
Generalizing the Raycaster

In Section 2.1 of the introductory raycasting chapter, we showed the basic raycasting algorithm in pseudocode. Although the algorithm was complete, it was not fully general, since it made the following assumptions:

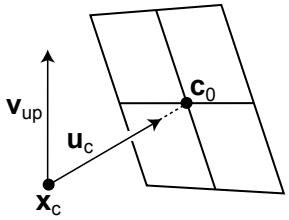
1. The camera is at the origin $(0, 0, 0)$.
2. The viewplane is parallel to the x-y plane (i.e., perpendicular to the z axis).
3. The viewplane is centered at z_p, along the negative z axis.
4. Every ray hits an object.
5. Given a hitpoint $\mathbf{x}$ in space, we know the color at that point.

In order to give us the foundations for a much more general algorithm, let us look at these assumptions and relax them.

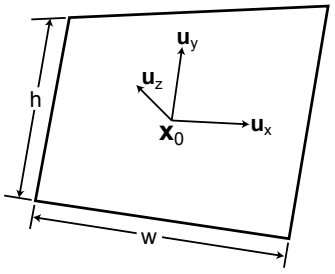
5.1 Camera at the origin, viewplane at z_p, parallel to the x-z plane



The convention that we start with is that the camera is at $(0, 0, 0)$ pointing down the negative z axis, and the viewplane center is at some position z_p (which is negative) along the z axis. So, if we know the coordinates of a pixel's center relative to the center of the viewplane, say (c_x, c_y), then the center of the pixel in 3D space is given by (c_x, c_y, z_p).



If we set the camera in an arbitrary location $\mathbf{x}_c = (x_c, y_c, z_c)$, and we specify a direction vector $\mathbf{u}_c$ specifying the direction in which the camera is aimed, then if the viewscreen distance is z_p (now a positive distance) then the center of the viewscreen will be at $\mathbf{c}_0 = \mathbf{x}_c + z_p\mathbf{u}_c$. The distance z_p is also known as the *focal length* of the camera.



If we specify what direction is up for the camera by an additional direction vector $\mathbf{v}_{up}$, then we can construct a coordinate system at the viewscreen center using cross products:

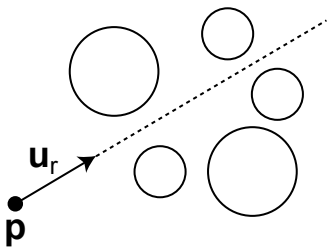
$$\begin{aligned} \mathbf{u}_x &= (\mathbf{u}_c \times \mathbf{v}_{up})/\|\mathbf{u}_c \times \mathbf{v}_{up}\|, \\ \mathbf{u}_z &= -\mathbf{u}_c, \\ \mathbf{u}_y &= \mathbf{u}_z \times \mathbf{u}_x. \end{aligned}$$

Horizontal direction on the viewscreen is $\mathbf{u}_x$, vertical direction is $\mathbf{u}_y$, and the surface normal direction of the viewscreen is $\mathbf{u}_z$. All that is left is to specify, as input parameters, the viewscreen width w and height h.

Thus, our algorithm for raycasting with an arbitrary camera would be:

```
// assume that z_p, x_c, u_c, v_up and M, N, h, w are given
c_0 = x_c + z_p * u_c; // screen center
u_x = (u_c × v_up) / ‖u_c × v_up‖; // horizontal screen direction
u_y = −u_c × u_x; // vertical screen direction
u_z = −u_c; // normal to the viewscreen
dydi =h / float(M); // vertical distance between pixels
dxdj = w / float(N); // horizontal distance between pixels
for(i = 0; i < M; i++){ // for each row
   py = −h / 2 + dydi * (i + 0.5); // y increment of row
   for(j = 0; j < N; j++){ // for each column
      px = −w / 2 + dxdj * (j + 0.5); // x increment of column
      p = c_0 + px * u_x + py * u_y; // pixel center in space
      if(orthographic)
         u_r = u_c; // orthographic ray vector
      else
         u_r = (p − x_c) / ‖p − x_c‖; // perspective ray vector
      x = shoot(p, u_r); // return position of first hit
      image[i][j] = shade(x); // pixel colored by object hit
   }
}
```

5.2 Every ray hits an object



The assumption is that `shoot(p, u_r)` always returns a position that was hit by the ray. Of course it is quite possible that a ray might travel through the scene and hit nothing. In this case, the simple thing is to have shoot return a special value that you can test to see if the ray hit nothing.

One common modification to shoot is to have it return the ray parameter t, which is the distance along the ray of an object that is hit, instead of returning a position $\mathbf{x}$. If you do this, then you can make the return value positive infinity to indicate no hit. If your programming language uses the IEEE floating point format, it will have a definition for infinity, and it can be used directly in computations. It is then an easy matter to have `shade()` assign a background color, for example a "sky" color, to the pixel if the value of t is infinity.

5.3 Given a point in space, we know the color

To eliminate the assumption that we know the color for a particular hit point in space, we can further modify `shoot()` and `shade()`. If we have `shoot()` return both the ray parameter t and an identifier o for the object hit, then we can modify `shade()` to find the color of the exact point on the object. The code might work something like this:

```
(t, o) = shoot(p, u_r); // o is object hit
if(t == INFINITY)
  image[i, j] = background_color;
else{
  x = p + t * u_r;
  image[i, j] = shade(x, o);
}
```

An even more general implementation might have `shade()` take t, $\mathbf{p}$ and $\mathbf{u}_r$ as parameters, instead of the hitpoint $\mathbf{x}$, so that it could use distance from the camera t as a shading parameter. In that case, the test for infinity could be done directly in the `shade()` routine.

The chapter on shading give the details of how the `shade()` routine might be implemented.