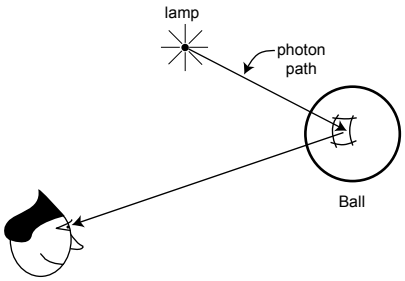
Chapter 4

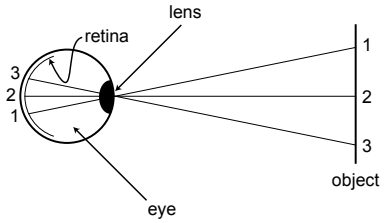
Raycasting

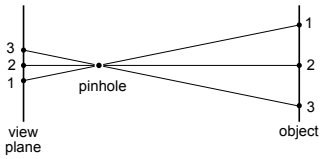
4.1 Raycasting foundations



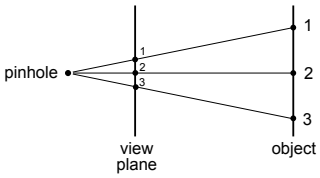
When you look at an object, like the ball in the picture to the left, what do you see? You do not actually see the ball itself. Instead, what you see is the light that is reflected from the surface of the ball. A photon of light leaves a light source, like the lamp in the figure, is reflected off the surface of the ball and travels to your eye. The cells on the retina of your eye are sensitive to the reception of photons, which your mind interprets as point flashes of colored light. The color is dependent on the wavelength of the photon, and the brightness of the light is dependent on the flux of the photons (the number of photons per second reaching your eye). Now of course, the lamp is sending many photons that are reflected from around that spot on the ball and into your eye, so your eye actually sees an area of the ball's surface.

Your eye has a lens and pupil with a small radius. Thus, to a first approximation, it can be thought of as having pin-hole optics. As in the picture to the right, light from one point on a surface arrives at one point on the retina in the back of your eye. This is depicted in the diagram to the right, where the images of points 1, 2 and 3 on a planar surface are projected to distinct points on the retina.



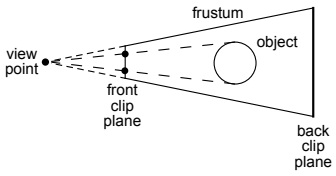


screen behind pinhole



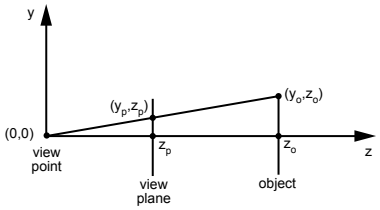
screen in front of pinhole

In computer graphics, it is usual to replace the eye, which is close to a sphere, with the a pupil letting light in, with a plane and a pinhole. So the model looks like the drawing above on the left. Note that the pinhole optics inverts the image, so in computer graphics it is usual to move the view plane in front of the pinhole, to remove the inversion, as in the drawing above on the right. This is the standard pinhole camera model of computer graphics.



Later, when we study view volumes and perspective projection, we will learn how to capture this geometry in a matrix describing a view frustum, which is a pyramid with its apex cut off. This matrix will provide a mathematical description of how an object located inside the frustum would be projected onto the front clip plane, which serves as the view plane.

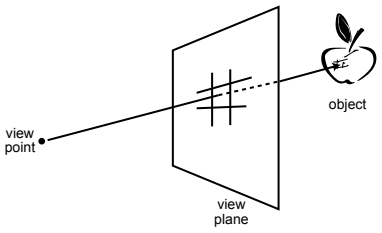
The geometry of the perspective projection is quite simple, involving the use of similar triangles and ratios. Let us look at this in two dimensions using the z axis as a distance from the viewpoint and the y axis to be height above the viewpoint, and letting the viewpoint be at the origin. Now, if the



viewplane is at distance z_p from the view point, all points projected onto this view plane will have z coordinate z_p. If an object point is at distance z_0 from the viewpoint, and has y coordinate y_0, then its projection y_p onto the viewplane will be determined by the ratios of sides of similar triangles: $\{(0,0),(0,z_p),(y_p,z_p)\}$, and $\{(0,0),(0,z_0),(y_0,z_0)\}$. So we have

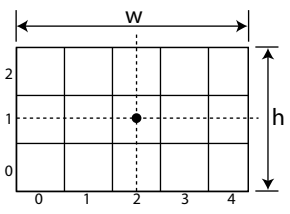
$$\frac{y_p}{z_p} = \frac{y_0}{z_0} \text{ or } y_p = \frac{z_p}{z_0} y_0$$

When we do a perspective projection, what we are doing, conceptually, is to use this geometric construction to project all of the points in the 3D model onto the view plane, to form the 2D image. This assumes that the view plane is parallel to the x-y plane, as in the drawing, and that the center of the view plane is on the z axis.



On the other hand, when we render an image using *raytracing*, we are computing the same projection, but we use a different approach to constructing it. In raytracing, we start from the viewpoint, send rays out through the viewplane, and determine which objects are "hit" by each ray. For example, the picture to the left shows one ray constructed between the viewpoint and a point on the apple. If we construct a correspondence between the view plane and the pixels in an image, we can "shoot" a ray from the viewpoint through every pixel on the viewplane, and see which object would be visible at that pixel. The color at the point "hit" by the ray would be used to color the pixel. This approach of ray "shooting" to find points on objects that project to a pixel on the viewscreen is called *raycasting*. Raycasting is the first step in raytracing, which allows us to calculate a correct shade for each pixel. For now, let us look just at the raycasting problem.

The diagram to the right will help us to understand raycasting geometry. Imagine that we have an image pixel array $M = 3$ pixels high by $N = 5$ pixels wide, arranged on a viewscreen whose width is w and whose height is h, measured in spatial coordinates (like centimeters). We assume the the camera is placed at the origin and aimed down the negative z axis. Thus, the center of the screen is $\mathbf{c} = (0, 0, -z_p)$. Now, if we consider pixel (i, j) where i is its row and



j is its column, then the center of pixel (i, j) is

$$\mathbf{p}_{ij} = \mathbf{c} - \begin{bmatrix} w/2 \\ h/2 \end{bmatrix} + \begin{bmatrix} \frac{w}{N}(j + 0.5) \\ \frac{h}{M}(i + 0.5) \end{bmatrix},$$

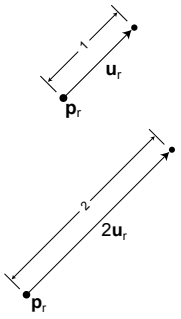
with measurements in centimeters.

To summarize, what we want to do in raycasting is to iterate over the pixels in the image, one at a time, shooting a ray through the center of the pixel out into the scene, looking for intersections between each ray and the scene geometry. For now, we assume that the viewpoint is at the origin $(0, 0, 0)$, and that the camera is aimed down the negative z axis. The raycasting algorithm to paint in all pixels of an $M \times N$ image on a $h \times w$ viewscreen looks like this:

```
pixheight = float(h) / float(M); // the height of one pixel
pixwidth = float(w) / float(N); // the width of one pixel
p_z = -z_p; // z coord is on screen
for(i = 0; i < M; i++){ // for each row
   p_y = c_y - h / 2 + pixheight * (i + 0.5); // y coord of row
   for(j = 0; j < N; j++){ // for each column
      p_x = c_x - w / 2 + pixwidth * (j + 0.5); // x coord of column
      u_r = p/||p||; // unit ray vector
      x = shoot(u_r); // return position of first hit
      image[i][j] = shade(x); // pixel colored by object hit
   }
}
```

In the algorithm, `shoot()` is a function that "shoots" the ray out into the scene, and returns the position of the first intersection of the ray with an object in the scene. The `shade()` function determines what color to assign the current pixel, based on the position of the "hit" returned by `shoot()`. Next, we will look at how to implement the `shoot()` function. Later we will learn how `shade()` might be implemented, and generalize the algorithm for a camera in any arbitrary position and aimed in any arbitrary direction

4.2 Ray construction

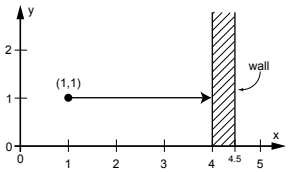


To understand raycasting, we first need to understand how to construct a ray, and then how to compute ray-object intersections. We define a ray to be a direction vector $\mathbf{u}_r$, an origination point $\mathbf{p}_r$. Distance along the ray is measured by parameter t.

Thus, all the points $\mathbf{x}$ lying on the ray satisfy the ray equation

$$\mathbf{x}(t) = \mathbf{p}_r + t\mathbf{u}_r,\ t \geq 0, \tag{4.1}$$

where each unique value of distance parameter t determines a unique point $\mathbf{x}$.



To determine if a ray hits an object, we look for any points on the surface of an object that satisfy Equation 4.1. For example, if the ray origination point is

$$p_r = \begin{bmatrix} 1 \\ 1 \end{bmatrix},$$

and the the ray is parallel to the x-axis then

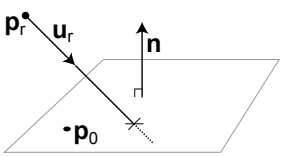
$$\mathbf{u}_r = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

Now, suppose that there is a vertical wall whose front face is at $x = 4$, and whose back face is at $x = 4.5$. In this case Equation 4.1 is satisfied (the wall is hit by the ray) for $3 \geq t \geq 3.5$ For raycasting we are usually only interested in the nearest "hit" point, so we take $t = 3$ as defining our point:

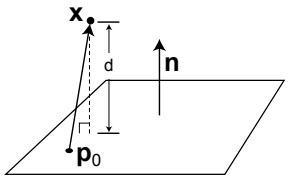
$$\mathbf{x} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} + 3 \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 4 \\ 1 \end{bmatrix}.$$

The wall is hit by the ray at point $\mathbf{x} = (4, 1)$, with ray parameter $t = 3$.

4.3 Ray-Plane intersection



Now, let us consider ray intersection with an infinite plane, arbitrarily positioned and oriented in 3D space. We can define the plane by a single point on the plane $\mathbf{p}_0$ and a unit surface normal vector $\mathbf{n}$ (i.e., a unit vector perpendicular to the surface). Together, these give the orientation of the plane and its position in space. We want to find out where the ray $(\mathbf{p}_r, \mathbf{u}_r)$ intersects the plane.



To see how to do this, we start with an easier problem. How far is a point $\mathbf{x}$ from the plane $(\mathbf{p}_0, \mathbf{n})$? This is straightforward to do using vector algebra. We first construct the vector from $\mathbf{p_0}$ to the point $\mathbf{x}$, i.e. $\mathbf{x} - \mathbf{p}_0$. The projection of this vector onto $\mathbf{n}$ gives the perpendicular distance

$$d = \mathbf{n} \cdot (\mathbf{x} - \mathbf{p}_0),$$

of $\mathbf{x}$ from the plane. So all we have to do is a vector subtract and a dot product to solve for the distance of an arbitrary point $\mathbf{x}$ from an arbitrary plane $(\mathbf{p}_0, \mathbf{n})$. Note that if $d > 0$ then $\mathbf{x}$ is above the plane by the distance d, if $d < 0$ then $\mathbf{x}$ is below the plane by the distance $|d|$, and if if $d = 0$ then $\mathbf{x}$ is exactly on the plane.

Now, we can combine our ray equation with our distance equation to find the ray plane intersection. The intersection occurs at $d = 0$, so

$$\mathbf{n} \cdot [(\mathbf{p}_r + t\mathbf{u}_r) - \mathbf{p}_0] = 0,$$

or

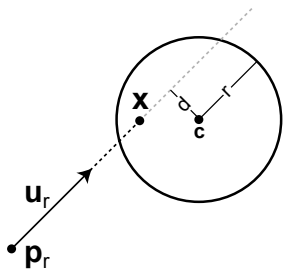
$$t(\mathbf{n} \cdot \mathbf{u}_r) = -\mathbf{n} \cdot (\mathbf{p}_r - \mathbf{p}_0),$$

or

$$t = -\frac{\mathbf{n} \cdot (\mathbf{p}_r - \mathbf{p}_0)}{\mathbf{n} \cdot \mathbf{u}_r}. \tag{4.2}$$

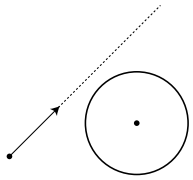
Equation 4.2 gives the value of t for which the ray intersects the plane. To find the intersection point $\mathbf{x}$ we simply apply this value of t to Equation 4.1. You should ask yourself what happens if the ray is parallel to the plane so that there is no intersection point? What happens to Equation 4.2 in this case? How would you test for this in a computer program?

4.4 Ray-Sphere intersection

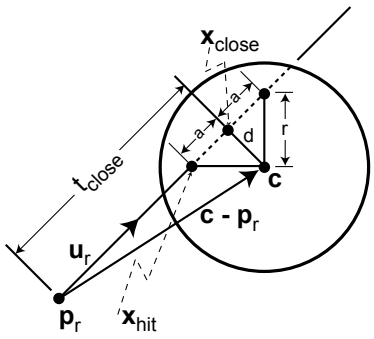


Now, let us look at another example that is very common in raycasting, that is the problem of ray-sphere intersection, diagrammed in the figure to the left.

To speed calculations, the ray-sphere intersection algorithm is usually broken into two steps. First, we do a quick calculation to see if there is any intersection at all. If there is none then we are done, as depicted in the picture to the right. If there is an intersection then we find the point $\mathbf{x}$ at which the ray first intersects the sphere. Here is how this is done.



Referring to the figure below, if we let d be the closest distance on the ray to the center of the sphere $\mathbf{c}$ then there will be no intersection if this distance is greater than the sphere radius r. As with ray-plane intersection, we can solve for this point of closest approach using a vector subtract and a dot product. Inspecting the figure, we can see that the value of ray parameter t at the closest approach is $t_{close} = \mathbf{u}_r \cdot (\mathbf{c} - \mathbf{p}_r)$. Thus, the position along the ray at closest approach is $\mathbf{x}_{close} = \mathbf{p}_r + t\mathbf{u}_r$. Now if $\|\mathbf{x}_{close} - \mathbf{c}\| > r$ we have no hit. If $\|\mathbf{x}_{close} - \mathbf{c}\| = r$ we exactly hit the circumference of the sphere, and $\mathbf{x}_{close}$ is the hit point. Only in the case with $\|\mathbf{x}_{close} - \mathbf{c}\| < r$ does the ray enter and then leave the sphere, and we have to solve for the closest hit point.



We could solve for the sphere intersection by using Equation 4.2, together with the sphere equation $(\mathbf{x} - \mathbf{c})^2 = r^2$, but it is more efficient to use what we have already computed. We already know $\mathbf{x}_{close}$, the closest point on the ray to the sphere center. The intersection point of the ray with the sphere must be on the ray, on either side of $\mathbf{x}_{close}$. Since these points are on the sphere surface, they must be the points where the distance from the center of the sphere is exactly the sphere radius r. Referring again to the figure above, if we find the distance a from $\mathbf{x}_{close}$ to the sphere surface then the two hit points will be at distance along the ray $t_{close} \pm a$. Let $d = \|\mathbf{x}_{close} - \mathbf{c}\|$ be the distance to the ray from the sphere center. Then by the Pythagorean theorem: $a^2 + d^2 = r^2$ or $a^2 = r^2 - d^2$, and

$$a = \sqrt{r^2 - d^2}.$$

It is clear that the nearest hit point is at distance along the ray $t_{close} - a$, and

so

$$\mathbf{x}_{hit} = \mathbf{p}_r + (t_{close} - a)\mathbf{u}_r.$$