CS 465 Lecture 9: Ray tracing

Steve Marschner
Cornell University

October 10, 2003

We have discussed images, how to represent them, how to process, them, how to display them, and so on. But we have used natural images (like the background in the brush assignment) or images that are computed from very simple 2D functions (like the airbrush in the brush assignment).

For much of the rest of the course we'll be concerned with computing images of *3D* scenes, where the images is meant to look (to some degree of approximation) like the view from a camera or through the human eye. This is known as *3D rendering*, and the resoning behind how we do it is based on simple approximations of the behavior of light in the real word: how it bounces around, reflecting off surfaces, until it enters the eye or camera so that we can see it and it contributes to the image.

Where does an image come from—a photograph, for instance? A camera responds to light that enters through the lens and hits the sensor inside, causing a response that's digitized and written into an image file. Your eye works the same way: light enters through the lens and hits the retina at the back of your eye, where there are cells that sense the light and send the signals to your brain.

So the image is formed by light that enters the eye or camera. But where does that light come from? Some objects generate light themselves: the fluorescent tubes in the ceiling of the lecture hall, for instance. But the vast majority of objects—walls, chairs, chalkboards, people—don't emit light, but we still see them because they reflect some of the light that was emitted by the light sources. So most of rendering is about figuring out how light gets from sources to the eye by reflecting off stuff along the way.

The first rendering method we'll discuss is *ray tracing*, which takes the most straightforward approach possible: if you want to know what color goes at a particular point in an image,

1. figure out what object is seen at that point in the image,
2. figure out how that object is illuminated, and
3. deduce how much light is reflected toward the eye.

That reflected light is the color that belongs at that point in the image.

Looking ahead a bit to put this method in context, there are two main approaches to rendering:

- *Image-order* rendering: for each pixel, ask "what object is visible at this pixel?" and write that color of that objectq into the image. Ray tracing works this way.

- *Object-order* rendering: for each object, ask "at what pixels is this object visible?" and write the appropriate colors into all those pixels. The real-time rendering done in graphics hardware works this way.

As a general rule it is possible to achieve a broader range of effects more simply with image-order methods, but object-order methods are considerably faster for reasonably sized scenes.

Perspective

The question of where objects in a 3D scene appear in a 2D view of that scene is a very old one, worked on since ancient times. The basic answers have been known since the Rennaissance.

In the 15th century, artists (notably Leonardo da Vinci) developed a set of rules about how to draw a 2D view of a 3D scene (involving parallel lines converging at *vanishing points*), which they called the laws of perspective. Perspective is based on the fundamental insight that what's going on is a *projection* through a *viewpoint*: a point in 3D space maps to the point in the 2D image that is the intersection of a line through the 3D point and a *view plane*. If you look through a window and trace the shapes of the objects you see on the glass, the window is the view plane and what you end up with is a perfect perspective rendition of the scene.

Why should this be true? Light travels along straight lines, and you only see light that's traveling along lines through your viewpoint. When you look at the traced image, objects in the tracing share the same lines as the corresponding real objects.

Another way to think of it: a 2D view of a 3D scene is a function from lines through the viewpoint to light intensity, or color. Without bringing in extra knowledge, we can't tell *where* on the line the light came from. So if we want to replace the scene by an image painted on a flat rectangle, we want the color on the painting to match the color on the the object that projects to that point along a line through the viewpoint.

Ray tracing

The first step I posed above in ray tracing is to figure out what scene point is visible at a particular point in the image. This idea of perspective as a projection along lines through the viewpoint leads directly to the answer: define a line through the viewpoint and the point on the image plane, and find the closest intersection of that ray with the scene. Really, we want the closest intersection that occurs *in front of* the eye, since we don't see objects that are behind us—that means we're really talking about the intersection of a *ray*, or half line, with the scene. The endpoint of the ray is the viewpoint, and the direction is determined by the image point.

This is the core idea of the ray tracing algorithm: to find out how much light arrives at a particular point from a particular direction, you intersect a ray with the scene and do some computations based on the object you hit:

for each pixel (i, j)
  $\mathbf{o}$ = viewpoint
  $\mathbf{d}$ = direction for pixel (i, j)

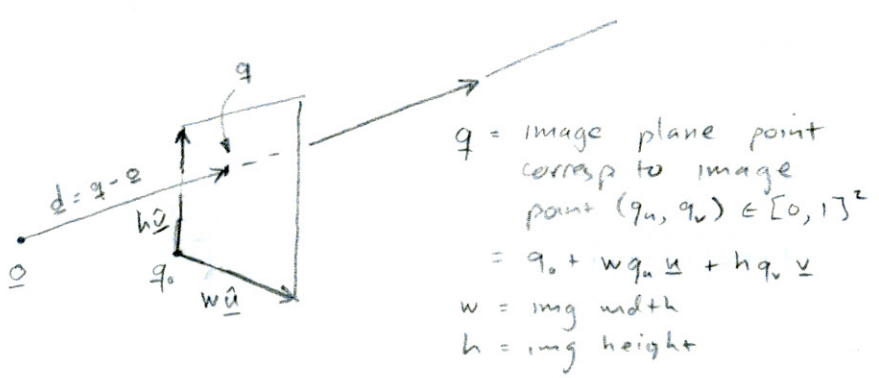
$\mathbf{p}$ = intersection of ray $\{\mathbf{o} + t\mathbf{d} | t \in [0, \infty)\}$ with scene
image$[i, j]$ = color reflected from $\mathbf{p}$ in direction $-\mathbf{d}$

There are three key tasks to be taken care of:

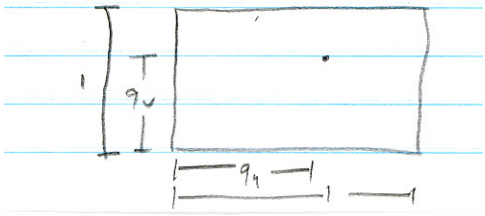
- Generating eye rays
- Ray intersection
- Shading

Generating eye rays

One can do this in various ways, but the simplest is to use the window analogy directly:



We can easily compute $\mathbf{q}$ if we have a basis for the view rectangle. Let $\mathbf{q}_0$ be the lower left corner of the image plane, and let $\hat{\mathbf{u}}$ and $\hat{\mathbf{v}}$ be the vectors along the bottom and left edges. Then if (q_u, q_v) are the coordinates of a point in the image, defined in the unit square:



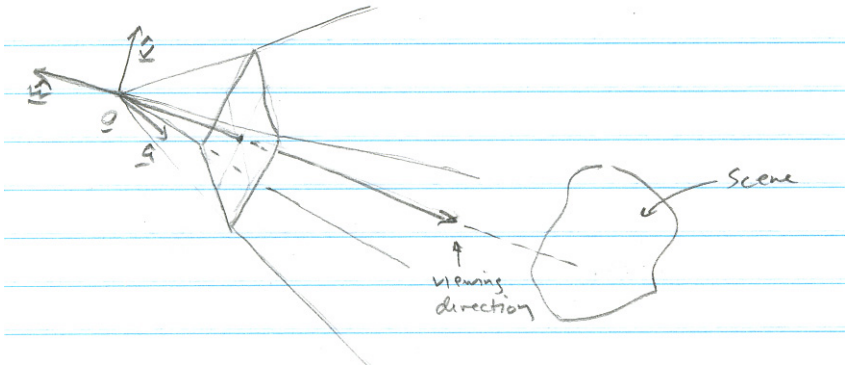
and w and h are the width and height of the rectangle, then $\mathbf{q} = \mathbf{q}_0 + wq_u\hat{\mathbf{u}} + hq_v\hat{\mathbf{v}}$.

This is all very well, but where is the image plane? Really, the question is, How do we specify a camera? There are two parts to this: the *camera pose* (how the camera is positioned in space) and the camera's *intrinsic parameters* (characteristics of the camera itself—e. g. the field of view).

Camera pose Pose boils down to a coordinate frame: the viewpoint (which is effectively the position of the camera) and an ONB that tells which way the camera is looking and which way is up.

By the way, the viewpoint is also called the “eye point,” the “camera position,” and the “center of projection” or COP.

We’ll use the letters $\hat{\mathbf{u}}$, $\hat{\mathbf{v}}$, and $\hat{\mathbf{w}}$ for the camera basis, with the convention that $\hat{\mathbf{u}}$ and $\hat{\mathbf{v}}$ are the horizontal and vertical axes of the image plane (so that $\hat{\mathbf{v}}$ is the direction the camera sees as “up”) and $-\hat{\mathbf{w}}$ is the direction the camera is looking. Why minus? Because we want a right-handed basis and making $\hat{\mathbf{u}}$ or $\hat{\mathbf{v}}$ opposite from the usual coordinates in the plane would be even more awkward.

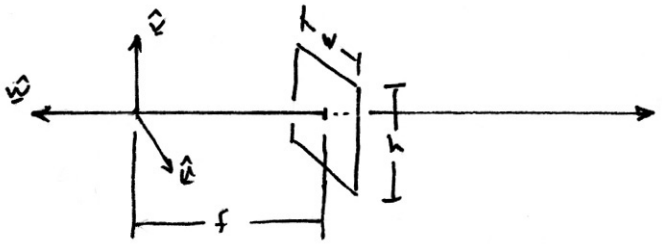


So for pose all we need to do is store the eye point and these three vectors. We can use them to find the image plane (by moving along $-\hat{\mathbf{w}}$ from the eye point) and to move around on the image plane (by adding $\hat{\mathbf{u}}$ and $\hat{\mathbf{v}}$).

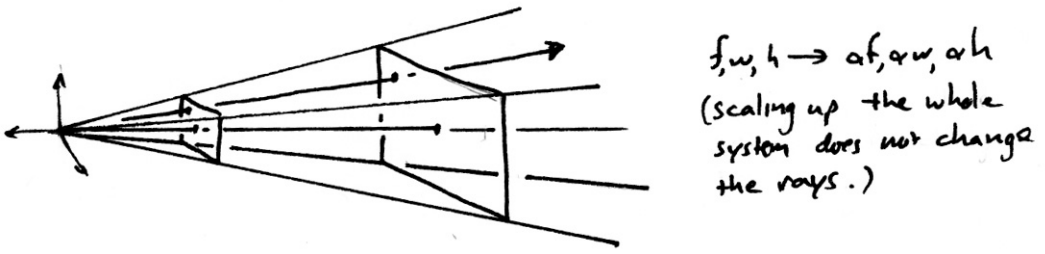
Intrinsic parameters Intrinsic parameters boil down to specifying the view rectangle. There are times when it’s useful to have the view rectangle tilted, off center, etc.—but for now we will go with the basic configuration:

- View plane is perpendicular to the $\hat{\mathbf{w}}$ axis (that is, parallel to $\hat{\mathbf{u}}$ and $\hat{\mathbf{v}}$).
- The $\hat{\mathbf{w}}$ axis pierces the view rectangle exactly at the center.

This leaves only three parameters that can vary: the width and height of the rectangle, and the distance from the eye point to the image plane:



This is really just two parameters, though, because scaling all three together does not change the picture (each pixel still maps to the same ray):



Camera specifications The question remains (for both aspects of the camera) how to specify the parameters in a way that's convenient for a user of the program. We don't want the user to have to come up with the vectors $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ and ensure that they are an ONB, for instance. It's much more intuitive to generate the frame from some more comprehensible pieces.

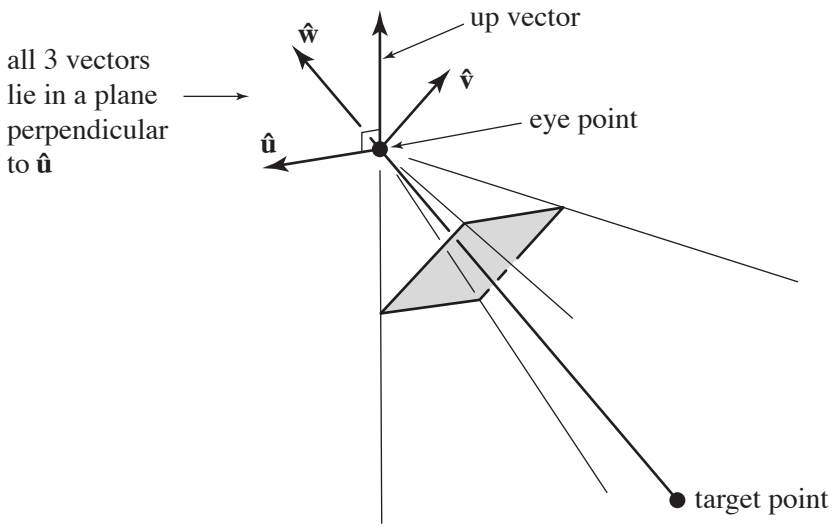
Recall there was a homework problem (from the book) about building a basis from two vectors, where the vectors $\mathbf{a}$ and $\mathbf{b}$ were given and we were supposed to come up with an ONB that had $\hat{\mathbf{w}}$ parallel to $\mathbf{a}$ and $\hat{\mathbf{u}}$ perpendicular to $\mathbf{a}$ and $\mathbf{b}$ (I changed the symbols from the problem to match the application here). Another way to say this is that $\hat{\mathbf{w}}$ is parallel to $\mathbf{a}$ and $\hat{\mathbf{v}}$ is in the plane defined by $\mathbf{a}$ and $\mathbf{b}$. The solution was

$$\hat{\mathbf{w}} = \frac{\mathbf{a}}{\|\mathbf{a}\|}$$
$$\hat{\mathbf{u}} = \frac{\mathbf{b} \times \hat{\mathbf{w}}}{\|\mathbf{b} \times \hat{\mathbf{w}}\|}$$
$$\hat{\mathbf{v}} = \hat{\mathbf{w}} \times \hat{\mathbf{u}}$$

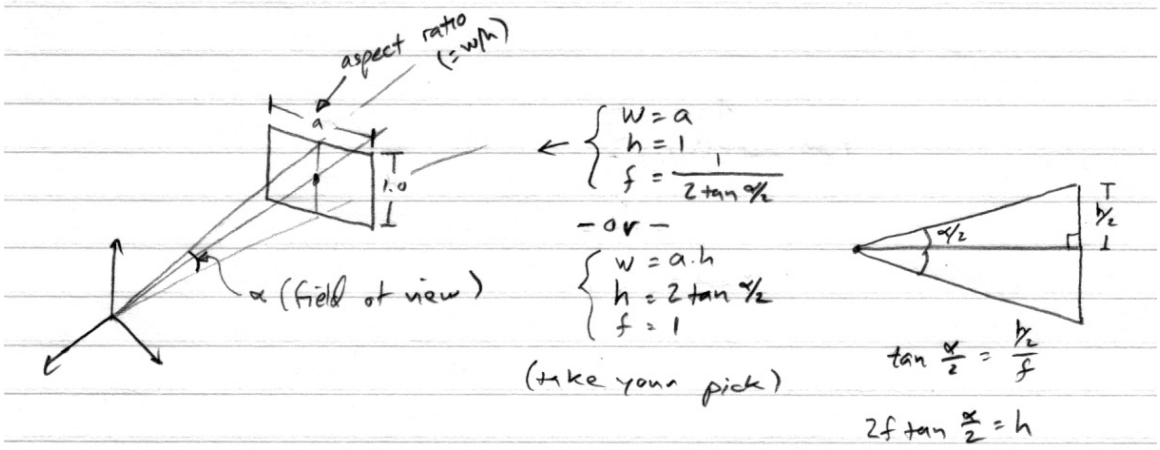
The standard way to specify a camera pose uses this approach, starting with three points:

- The *eye point* becomes the origin of the coordinate frame.
- The *target point* is where the camera is looking (should appear in the center of the image). This defines the camera's main axis: the $\hat{\mathbf{w}}$ axis points away from the target point.
- The *up vector* is a direction that should appear vertical in the image. This defines the orientation of the camera about its main axis: the $\hat{\mathbf{v}}$ axis lies in a plane defined by $\hat{\mathbf{w}}$ and the *up vector*.

So we get the camera basis using the above algorithm starting with $\mathbf{a} = -(\text{target point} - \text{eye point})$ and $\mathbf{b} = (\text{up vector})$.

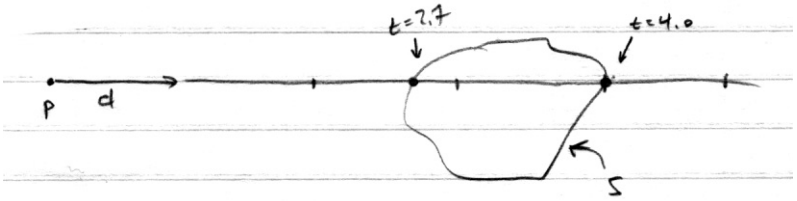


To specify intrinsics, we'll use a *vertical field of view* α and an *aspect ratio* a:



Ray intersection

Once we can generate eye rays, we need to be able to intersect them with the objects in the scene.



That is, solve for $\{t|\mathbf{p} + t\mathbf{d} \in S \text{and} t >= 0\}$ (remember that we can find the intersection point, given t.

This is a simple problem statement, but the problem is not always easy to solve. However, it is quite simple for a number of surfaces, particularly spheres and planes.

In general, an implicit representation of your surface is a good way to go whenever possible. In this case the problem boils down to solving the ray and surface equations simultaneously:

$$f(\mathbf{x}) = 0$$
$$\mathbf{x} = \mathbf{p} + t\mathbf{d}$$

but we can just eliminate $\mathbf{x}$ by substitution:

$$f(\mathbf{p} + t\mathbf{d}) = 0$$

and we now have a scalar equation in a single variable to solve.

Ray-sphere intersection For a sphere with center $\mathbf{c}$ and radius r the equation is:

$$\begin{aligned} f(\mathbf{x}) &= \|\mathbf{x} - \mathbf{c}\|^2 - r^2 = 0 \\ &= \langle \mathbf{p} + t\mathbf{d} - \mathbf{c}, \mathbf{p} + t\mathbf{d} - \mathbf{c} \rangle - r^2 \end{aligned}$$

This gives rise to a (scalar) quadratic equation:

$$\begin{aligned} 0 &= \langle \mathbf{p} - \mathbf{c} + t\mathbf{d}, \mathbf{p} - \mathbf{c} + t\mathbf{d} \rangle - r^2 \\ 0 &= \langle \mathbf{p} - \mathbf{c}, \mathbf{p} - \mathbf{c} \rangle + 2t\langle \mathbf{p} - \mathbf{c}, \mathbf{d} \rangle + t^2 \langle \mathbf{d}, \mathbf{d} \rangle - r^2 \\ 0 &= (\langle \mathbf{d}, \mathbf{d} \rangle)t^2 + (2\langle \mathbf{p} - \mathbf{c}, \mathbf{d} \rangle)t + (\langle \mathbf{p} - \mathbf{c}, \mathbf{p} - \mathbf{c} \rangle - r^2) \end{aligned}$$

which can be solved using the quadratic formula to obtain

$$t = \frac{-\langle \mathbf{p} - \mathbf{c}, \mathbf{d} \rangle \pm \sqrt{\langle \mathbf{p} - \mathbf{c}, \mathbf{d} \rangle^2 - \langle \mathbf{d}, \mathbf{d} \rangle (\langle \mathbf{p} - \mathbf{c}, \mathbf{p} - \mathbf{c} \rangle - r^2)}}{\langle \mathbf{d}, \mathbf{d} \rangle}.$$

[See the slides for the geometric version.]

Shading

In this lecture we are just doing the very basic version of shading, enough to get a ray tracer functioning.

We've said that all light originates from sources and gets to the camera by reflecting off surfaces. A few assumptions lead to the simplest shading algorithms:

1. *Direct lighting:* all light starts from a source, reflects off one surface, and goes to the eye. Light that reflects first off one surface and then off another is neglected.
2. *Point sources:* all light sources are very small and they radiate light uniformly in all directions.
3. *diffuse reflection:* when light reflects from a surface, it goes equally in all directions, regardless of where it came from. This means that a particular point on a surface has the same color from all viewpoints.

[The slides cover this part fairly well on their own.]