An Introduction to Ray Tracing

Edited by
ANDREW S. GLASSNER

Xerox PARC
3333 Coyote Hill Road
Palo Alto CA 94304
USA

ACADEMIC PRESS
Harcourt Brace Jovanovich, Publishers
London · San Diego · New York · Berkeley · Boston
Sydney · Tokyo · Toronto

1 An Overview of Ray Tracing

ANDREW S. GLASSNER

1 IMAGE SYNTHESIS

1.1 Introduction

Ray tracing is a technique for *image synthesis*: creating a 2-D picture of a 3-D world.

In this article we assume you have some familiarity with basic computer graphics concepts, such as the idea of a *frame buffer*, a *pixel*, and an *image plane*. We will use the term pixel in this article to describe three different, related concepts: a small region of a monitor, an addressable location in a frame buffer, and a small region on the image plane in the 3-D virtual world. Typically, these three devices (monitor, frame buffer, and image plane) are closely related, and the region covered by a pixel on one has a direct correspondence to the others. We will find it convenient to sometimes blur the distinction between these different devices and refer to the image plane as 'the screen.'

Most computer graphics are created for viewing on a flat screen or piece of paper. A common goal is to give the viewer the impression of looking at a photograph (or movie) of some three-dimensional scene. Our first step in simulating such an image will be to understand how a camera records a physical scene onto film, since this is the action we want to simulate.

After that we'll look at how the ray tracing algorithm simulates this physical process in a computer's virtual world. We'll then consider the issues that arise when we actually implement ray tracing on a real computer.

1.2 The Pinhole Camera Model

Perhaps the simplest camera model around is the *pinhole camera*, illustrated in *Figure 1*. A flat piece of photographic film is placed at the back of a light-proof

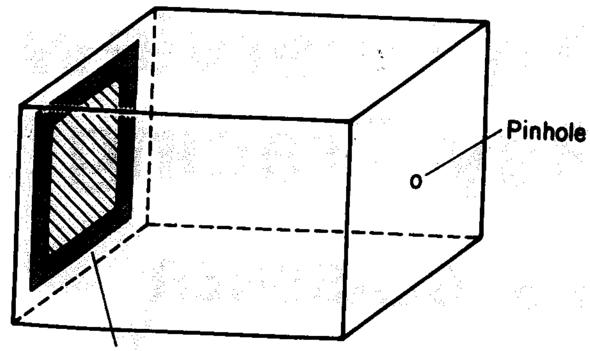


Fig. 1. The pinhole camera model.

box. A pin is used to pierce a single hole in the front of the box, which is then covered with a piece of opaque tape. When you wish to take a picture, you hold the camera steady and remove the tape for a while. Light will enter the pinhole and strike the film, causing a chemical change in the emulsion. When you're done with the exposure you replace the tape over the hole. Despite its simplicity, this pinhole camera is quite practical for taking real pictures.

The pinhole is a necessary part of the camera. If we removed the box and the pinhole and simply exposed the entire sheet of film to the scene, light from all directions would strike all points on the film, saturating the entire surface. We'd get a blank (white) image when we developed this very overexposed film. The pinhole eliminates this problem by allowing only a very small number of light rays to pass from the scene to the film, as shown in *Figure 2*. In particular, each point on the film can receive light only along the line joining that piece of film and the pinhole. As the pinhole gets bigger, each bit of the film receives more light rays from the world, and the image gets brighter and more blurry.

Although more complicated camera models have been used in computer graphics, the pinhole camera model is still popular because of its simplicity and wide range of application. For convenience in programming and

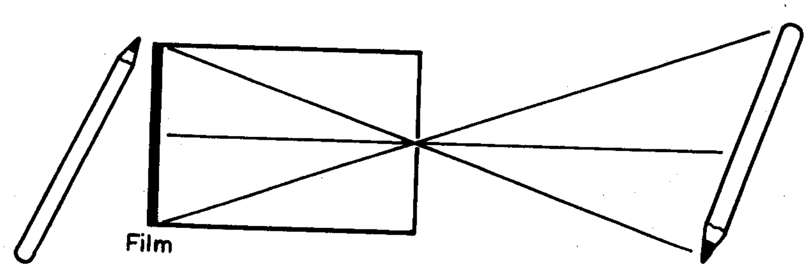


Fig. 2. The pinhole only allows particular rays of light to strike the film.

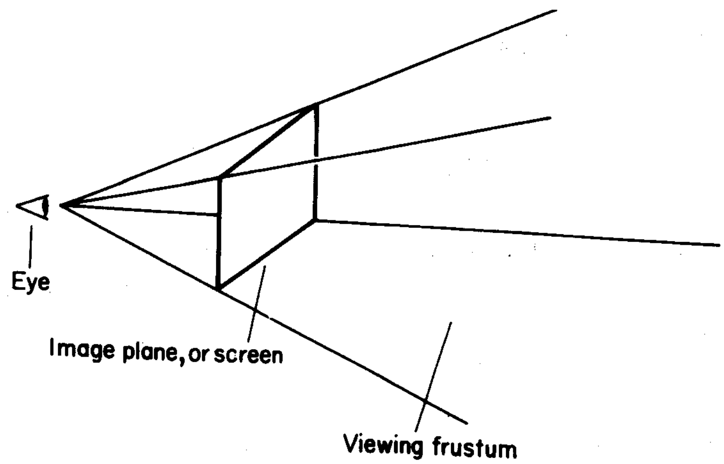


Fig. 3. The modified pinhole camera model as commonly used in computer graphics.

modeling, the classic computer graphics version of the pinhole camera moves the plane of the film out in *front* of the pinhole, and renames the pinhole as the *eye*, as shown in *Figure 3*. If we built a real camera this way it wouldn't work well at all, but it's fine for a computer simulation. Although we've moved things around, note that each component of the pinhole camera is accounted for in *Figure 3*. In particular, the requirement that all light rays pass through the pinhole is translated into the requirement that all light rays pass through the eyepoint. For the rest of our discussion we will use this form of the pinhole camera model.

You may want to think of the model in *Figure 3* as a Cyclopean viewer looking through a rectangular window. The image he sees on the window is determined by where his eye is placed and in what direction he is looking.

In *Figure 3* we've drawn lines from the eye to the corners of the screen and then beyond. You can think of these lines as the edges of walls that include the eye and screen. The only objects which the eye can directly see (and thus the film directly image) are those which lie within all four of the walls formed by these bounds. We also arbitrarily say that the only objects that can show up on the image plane are those in *front* of the image plane, i.e. those on the other side of the plane than the eye. This makes it easy to avoid the pitfall of having our whole image obscured by some large, nearby object. The eye also cannot directly see any objects behind itself.

All these conditions mean that the world that finally appears on the screen lies within an infinite pyramid with the top cut off (such a point-less pyramid is called a *frustum*). The three-dimensional volume that is visible to the eye, and may thus show up on the screen, is called the *viewing frustum*. The walls that form the frustum are called *clipping planes*. The plane of the screen is called the *image plane*. The location of the eye itself is simply referred to as the *eye position*.

1.3 Pixels and Rays

When we generate an image we're basically determining what color to place in each pixel. One way to think of this is to imagine each pixel as a small, independent window onto the scene. If only one color can be chosen to represent everything visible through this window, what would be the correct color? Much of the work of 3-D computer graphics is devoted to answering that question.

One way to think about the question is within the context of the pinhole camera model. If we can associate a region of film with a given pixel, then we can study what would happen to that region of the film in an actual physical situation and use that as a guide to determine what should happen to its corresponding pixel in the computer's virtual world. If we use the computer graphics pinhole camera of *Figure 3*, this correspondence is easy.

In *Figure 4*, one pixel in particular and its corresponding bit of film have been isolated. A small distribution of light rays can arrive from the scene, pass through the pinhole, and strike the film. After the exposure has completed and the pinhole is covered, that small region of film has absorbed many different rays of light. If we wish to describe the entire pixel with a 'single' color, a good first approximation might be to simply average together all the colors of all the light that struck it.

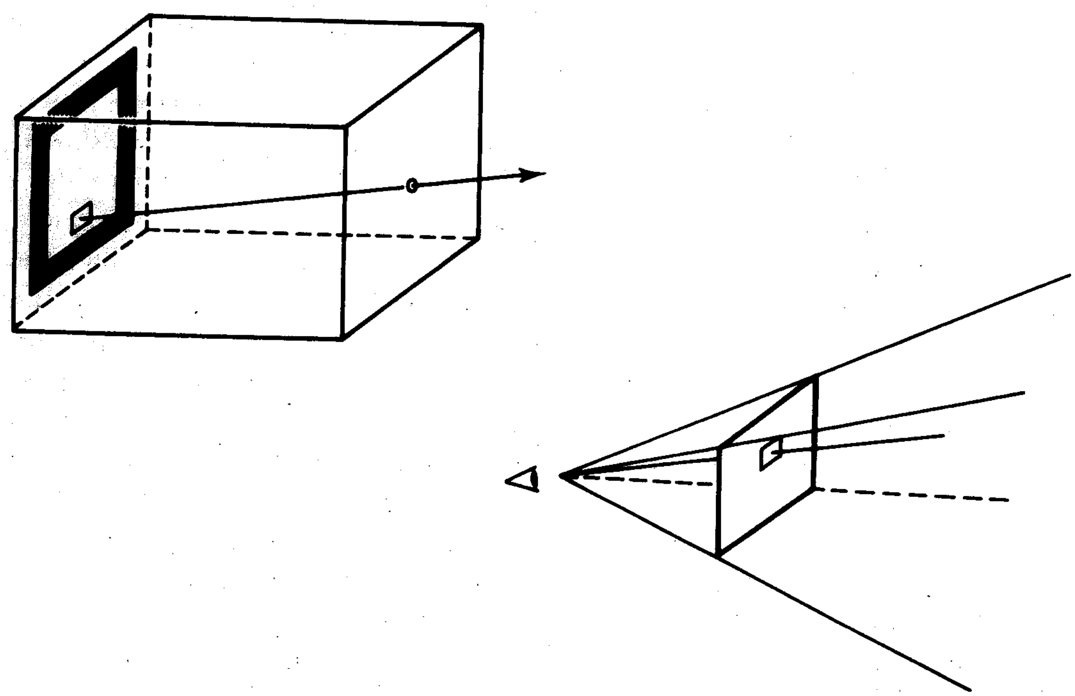


Fig. 4. Every pixel on the screen in the computer graphics camera model corresponds directly to a region of film in the pinhole camera.

This 'averaging' of the light in a pixel is in fact the way we eventually determine a single color for the pixel. The mathematics of the averaging may get somewhat sophisticated (as we'll see in later chapters), but we'll always be looking at lots of light rays and somehow combining their colors.

From this discussion we can see that the eventual goal is to fill in every pixel with the right color, and the way to find this color is to examine all the light rays that strike that pixel and average them together somehow. From now on, when we refer to the 'pinhole camera model,' or even just 'the camera,' we'll be referring to the computer graphics version of *Figure 3*.

2 TRACING RAYS

2.1 Forward Ray Tracing

We saw in the last section that one critical issue in image synthesis is the determination of the correct color for each pixel, and that one way to find that color is to average together the colors of the light rays that strike that pixel in the pinhole camera model. But how do we find those rays, and what colors are they? Indeed, just what do we mean by the 'color' of a light ray?

The color of a ray is not hard to define. We can think of a light ray as the straight path followed by a light particle (called a *photon*) as it travels through space. In the physical world, a photon carries energy, and when a photon enters our eye that energy is transferred from the photon itself to the receptor cells on our retina. The color we perceive from that photon is related to its energy. Different colors are thus carried to our retina by photons of different energies.

One way to talk of a photon's energy is as energy of vibration. Although photons don't actually 'vibrate' in any physical sense, vibration makes a useful mathematical and intuitive model for describing a photon's energy. In a vibrating photon model, different speeds of vibration are related to different energies, and thus different colors. For this reason we often speak of a given color as having a certain *frequency*. Another way to describe the rate of vibration is with the closely related concept of *wavelength*. For example, we can speak of frequency and say that our eyes respond to light between about 360 and 830 terahertz (abbreviated THz; 1 THz = 10^{12} cycles per second). Alternatively, we can speak of wavelength and describe the same range as 360–830 nanometers (1 nm = 1 billionth of a meter). In mathematical formulae, it is typical to use the symbol f to represent the frequency of a photon, and λ to represent its wavelength.

Generally speaking, each unique frequency has an associated energy, and thus will cause us to see an associated color. But colors can add both on film and in the eye; for example, if a red photon and a green photon both arrive at our eye simultaneously, we will perceive the sum of the colors: yellow.

Consider a particular pixel in the image plane. Which of the photons in a three-dimensional scene actually contribute to that pixel?

Figure 5 shows a living room, consisting of a couch, a mirror, a lamp, and a table. There's also a camera, showing the position of the eye and the screen.

Photons must begin at a light source. After all, exposing a piece of film in a completely dark room doesn't cause anything to happen to the film; no light hits it. If the lamp in the living room is off, then the room is completely dark and our picture will be all black. So imagine that the light is on. The lamp contains a single, everyday white light bulb. The job of the bulb is to create photons at all the visible frequencies and send them out in all directions. In order to get a feel for how the photons eventually contribute to the photograph, let's follow a few photons in particular.

We will not consider all the subtleties and complexities that actually occur when light bounces around in a three-dimensional scene; that discussion could fill several books! Instead, we'll stick to the most important concepts.

Let's say photon A is colored blue (that is, if the photon struck our eye we would say that we were looking at blue light). It leaves the light source in the direction of the wall, and then strikes the wall. Some complicated things can happen when the photon hits the wall's surface, which we'll talk about later in

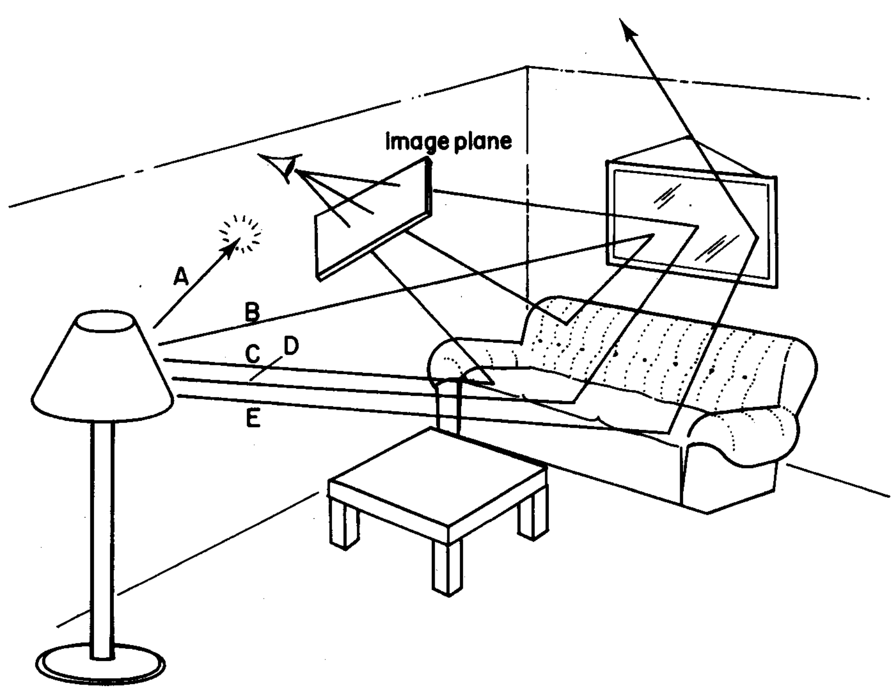


Fig. 5. Some light rays (like A and E) never reach the image plane at all. Others follow simple or complicated routes.

the Surface Physics section. For now, we'll say that the light hits the wall, and is mostly absorbed. So photon A stops here, and doesn't contribute to the picture.

Photon C is also blue. It leaves the light and strikes the couch. At the couch, the photon is somewhat absorbed before being reflected. Nevertheless, a (somewhat weaker) blue photon leaves the couch and eventually passes through the screen and into our eye. So that's why we get to actually see the couch: light from the light source strikes the couch and gets reflected to our eye through the screen.

The reflection can get more complex. Photon B is reflected off the mirror before it hits the couch; its path is light, mirror, couch, film. Alternatively, photon D leaves the light source, strikes the couch, and then strikes the mirror to be reflected onto the film. Photon E follows a very similar path, but it never strikes the film at all. Other photons may follow much more complicated paths during their travels.

So in general, photons leave the light source and bounce around the scene. Usually, the light gets a little dimmer on each bounce, so after a couple of bounces the light is so dim you can't seen it anymore. Only photons that eventually hit the screen and then pass into our eye (when they're still bright) actually contribute to the image. You might want to look around yourself right now, identify some light source, and imagine the paths of some photons as they leave that light, bounce around the objects near you, and eventually reach your eye. Notice that if you're looking into a mirror, you can probably see some objects in the mirror that you can't see directly. The photons are leaving the light source, hitting those objects, then hitting the mirror, and eventually finding your eyes.

We've just been *ray tracing*. We followed (or traced) the path of a photon (or ray of light) as it bounced around the scene. More specifically, we've been *forward ray tracing*; that is, we followed photons from their origin at the light and into the scene, tracing their path in a forward direction, just as the photons themselves would have travelled it.

2.2 Forward Ray Tracing and Backward Ray Tracing

The technique of forward ray tracing described above is a first approximation to how the real world works. You might think that simulating this process directly would be a good way to make pictures, and you would be pretty much correct. But there is a problem with such a direct simulation, and that's the amount of time it would take to produce an image. Consider that each light source in a scene is generating possibly millions of photons every second, where each photon is vibrating at a slightly different frequency, going in a slightly different direction. Many of these photons hit objects that you would never see at all, even indirectly. Other just pass right out of the scene, for example by flying out through a window. If we were to try to create a picture by actually following photons from their source, we would find a depressingly small number of them ever hit the screen with any appreciable intensity. It might take years just to make one dim picture!

The essential problem is not that forward ray tracing is no good, but rather that many of the photons from the light source play no role in a given image. Computationally, it's just too expensive to follow useless photons.

The key insight for computational efficiency is to reverse the problem, by following the photons backwards instead of forwards. We start by asking ourselves, "Which photons *certainly* contribute to the image?" The answer is those photons that actually strike the image plane and then pass into the eye. Each of those photons travelled along some path before it hit the screen; some may have come directly from a light source, but most probably bounced around first.

Let's consider a particular point on the image plane. We can easily find the path followed by a photon that hit that point on the screen and then our eye: it's the line joining our eye and that point on the film, as shown in *Figure 6*. We know that the path of the photon is a line bounded at one end (where it strikes our eye), but the photon could have started anywhere along the line. The formal term for a line that has one endpoint fixed is a *ray*.

So if some photon actually did contribute to our view of the image at that point, it came along the ray joining our eye and that point on the film. But what object did that photon come from? If we extend the ray into the world, we can look for the nearest object along the path of the ray. The photon must have come from this object.

Consider the ray in *Figure 7*, which shows a light ray joining a sphere and the eye, passing through the image plane. It is the *possible* path of a photon; we don't know if any photon actually took that path. But if any photon hit that piece of the screen and then our eye, it had to come along that line from the sphere to our eye. So our new plan will be to ask if any photons actually did come along that path.

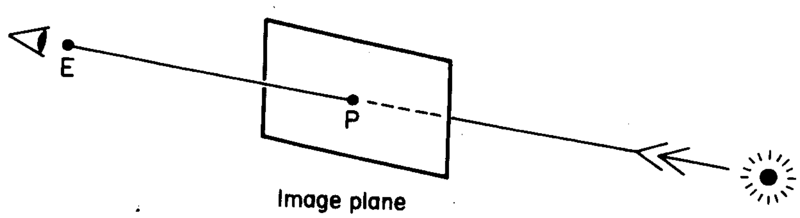


Fig. 6. A photon bringing light to the eye (at E) arrives by passing through point P on the image plane. The photon's path is along the straight line joining E and P.

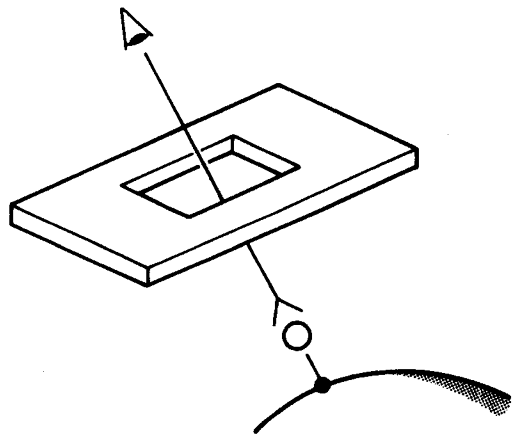


Fig. 7. A photon leaving the sphere could fly through a pixel and into the eye.

In this approach we're following rays not forward, from the light source to objects to the eye, but *backward*, from the eye to objects to the light source. This is a critical observation because it allows us to restrict our attention to rays that we know will be useful to our image—the ones that enter our eye!

Now that we've found the object a photon may have left to strike our eye, we must find out if any photon really did travel that path, and if so what its color is. We will address those topics below.

Because forward ray tracing is so expensive, the term 'ray tracing' in computer graphics has come to mean almost exclusively backward ray tracing. Unfortunately, some of the notions of backward ray tracing have led to some possibly confusing notation. Recall that we follow a ray backwards to find out where it may have begun. Nevertheless, we often carry out that search in a program by following the path of the light ray backwards, imagining ourselves to be riding along a path taken by a photon, looking for the first object along our path; this is the object from which the ray began. So we sometimes speak of looking for the "first object hit by the ray," or the "first object on the ray's path". What we're actually referring to is the object that may have radiated the photon that eventually travelled along this ray. This backwards point of view is prevalent in ray tracing literature and algorithms, so it may be best to think things through now and not get confused later. In summary, the "first object hit by a ray" means "the object which might have emitted that ray."

2.3 Ray Combination

When we want to find the color of a light ray, we need to find all the different light that originally contributed to it. For example, if a red light ray and a green light ray find themselves on exactly the same path at the same time, we might as well say that together they form a single yellow ray (red light and green light arriving at your eye simultaneously give the impression of yellow). So in *Figure 7*, where the light at a given pixel came from a sphere, we need to find a complete description of *all* the light leaving that point of the sphere in the direction of our eye. We'll see that we can rig our examination of the point so that we're only studying the light that will actually contribute to the pixel.

To aid in our discussion, we'll conceptually divide light rays into four classes: *pixel rays* or *eye rays* which carry light directly to the eye through a pixel on the screen, *illumination rays* or *shadow rays* which carry light from a light source directly to an object surface, *reflection rays* which carry light reflected by an object, and *transparency rays* which carry light passing through an object. Mathematically, these are all just rays, but it's computationally convenient to deal with these classes.

The pixel rays are the ones we've just studied; they're the rays that carry photons that end at the eye after passing through the screen (or in backwards ray tracing, they're rays that start at the eye and pass through the screen). Let's look at the other three types of rays individually.

The whole idea is to find out what light is arriving at a particular point on a surface, and then proceeding onward to our eye. Our discussion may be broken into two pieces: the illumination at a point on the object (which describes the incoming light), and the radiation of light from that point in a particular direction. We can determine the radiated light at a point by first finding the illumination at that point, and then considering how that surface passes that light on in a given direction (of course, if the object is a light source it could add some additional light of its own).

Knowing the illumination and surface physics at a point on a surface, we can determine the properties of the light leaving that point. We broke up rays into the three classes of shadow, reflection, and transparency because they're the three principle ways that light arrives at (and then leaves from) a surface. Some light comes directly from the light source and is then re-radiated away; the properties of this incoming light are determined by the shadows rays. Some light may strike the object and then be reflected; the reflection rays model this light. Lastly, some light comes from behind the object and may pass through; this light is modeled by the transparency rays.

2.4 Shadow and Illumination Rays

Imagine yourself on the surface of an object, such as point P in *Figure 8*. Is any light coming to you from the light sources? One way to answer that question is to simply look at each light. If you can see the light source, then there's a clear path between you and the light, and at least some photons will certainly travel along this path. If any opaque objects are in your way, then no light is coming directly from the light into your eye, and you are in shadow with respect to that light.

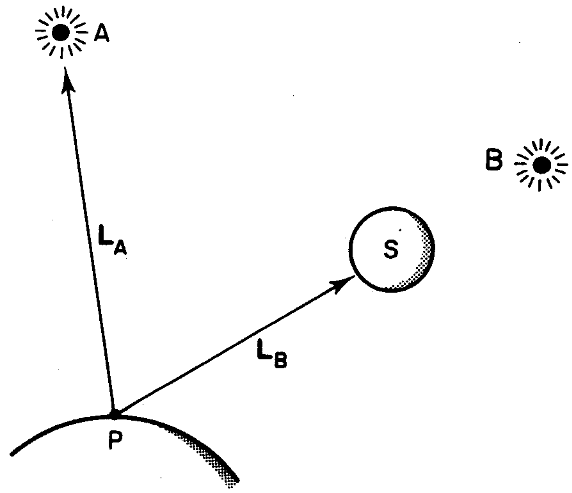


Fig. 8. To determine the illumination at a point P, we ask if photons could possibly travel from each light source to P. We answer this by sending shadow ray L_A towards light source A. It arrives at A, so P L_A is actually an illumination ray from P to A. But ray L_B is blocked from light source B by sphere S, so no light arrives at P from B.

We can simulate this operation of standing on the object and looking towards the light source with a light ray called a *shadow ray*. In practice, a shadow ray is like any other ray, except that we use it to 'feel around' for shadows; thus this kind of ray is sometimes also called a *shadow feeler*. Basically we start a ray at the object and send it to the light source (remember, we're following the paths of photons backwards). If this backwards ray reaches the light source without hitting any object along the way, then certainly some photons will come forwards along this ray from the light to illuminate the object. But if any opaque objects are in our way, then the light can't get through the intervening object to us; we would then be in shadow relative to that light source. *Figure 8* shows two shadow rays leaving a surface, ray $\mathbf{L_A}$ going to light source A, and ray $\mathbf{L_B}$ going to light source B. Ray $\mathbf{L_A}$ gets to its light source without interruption, but ray $\mathbf{L_B}$ hits an opaque object along the way. Thus we deduce that light can (and will!) arrive from light A, but not from light B.

When a shadow ray is able to reach a light source without interruption, we stop thinking of it as a 'shadow feeler' and turn it around, thinking of it as an *illumination ray*, which carries light to us from the light source.

In summary, the first class of illumination rays that contribute to the color of the light leaving an object are the light rays coming directly from the light source, illuminating the object. We determine whether there actually are any photons coming from a given light by sending out a shadow ray to each light source. If the ray doesn't encounter any opaque objects along the way to the light, that's our signal that photons will arrive from that light to the object. If instead there is an opaque object in the way, then no photons arrive and the object is in shadow relative to that light source.

Throughout this discussion we've only discussed what happens when the shadow ray hits a matte, opaque object. When it hits a reflective or transparent object the situation is much more complicated. For many years, people used a variety of ad hoc tricks to handle situations where shadow rays hit reflective or transparent surfaces. We now know some better ways to handle this situation; these will be discussed later in the book when we cover stochastic ray tracing.

2.5 Propagated Light

Recall that our overall goal is to find the color of the light leaving a particular point of a surface in a particular direction. We said that the first step was to find out which light was striking the object; some of that light would perhaps continue on in our direction of interest.

In the spirit of backward ray tracing, we'll look only for the incoming light that will make a difference to the radiated light in the direction we care about. After all, if some light strikes the surface but then proceeds away in a direction we don't care about, there's no need to really know much about that incoming light.

We will use the term *propagated light* to describe the illuminating light about which we care. Of all light that is striking a surface, which light is propagated just in our direction of interest? In ray tracing, we assume that most light interaction can be accounted for with four mechanisms of light transport (more about this in the Surface Physics chapter). For now, we'll concentrate exclusively on the two mechanisms called specular reflection and specular transmission—and since they're our only topics at the moment, well often leave off the adjective 'specular' in this section.

The general idea is that any illumination that falls on a surface and then is sent into our direction of interest either bounced off the surface like a basketball bouncing off a hardwood floor (reflection), or passed through the surface after arriving on the other side like a car driving through a tunnel (transmission). In the case of perfect (specular) reflection and transmission for a perfectly flat, shiny surface, there is exactly one direction from which light can arrive in order to be (specularly) reflected or transmitted into our eye.

When we are trying to determine the illumination at a point, recall that we originally found that point by following a ray to the object. Since we followed that ray backwards to the object, it is called the *incident ray*. Thus, our goal is to find the color of the light leaving the object in the direction opposite to the incident ray.

2.6 Reflection Rays

If we look at a perfectly flat, shiny table, we will see reflections of other objects in the tabletop. We see those reflections because light is arriving at the tabletop from the other objects, bouncing off of the tabletop, and then arriving in our eye. For a fixed eyepoint, each position on the table has exactly one direction from which light can come that will be bounced back into our eye.

For example, *Figure 9* shows a photon of light bouncing around a scene, ending up finally passing through the screen and into the eye. On its last bounce, the photon hit point P and then went into the eye. Photon B also hit point P, but it was bounced (or reflected) into a direction that didn't end up going into our eye. So for that eyepoint and that object, only a photon travelling along the path marked A could have been reflected into our direction of interest.

When we wish to find what light is reflected from a particular point into the direction of the incident ray, we find the *reflected ray* (or *reflection ray*) for that point and direction; this is the ray that can carry light to the surface that will be perfectly reflected into the direction of the incident ray. To find the color of the reflected ray, we follow it backwards to find from which object it began. The color of the light leaving that object along the line of the reflected ray is the color of that reflected ray. When we know the reflected ray's color, we can contribute it to any other light leaving the original surface struck by the incident ray.

Note the peculiar terminology of backward ray tracing: light arrives along the reflected ray and departs along the incident ray.

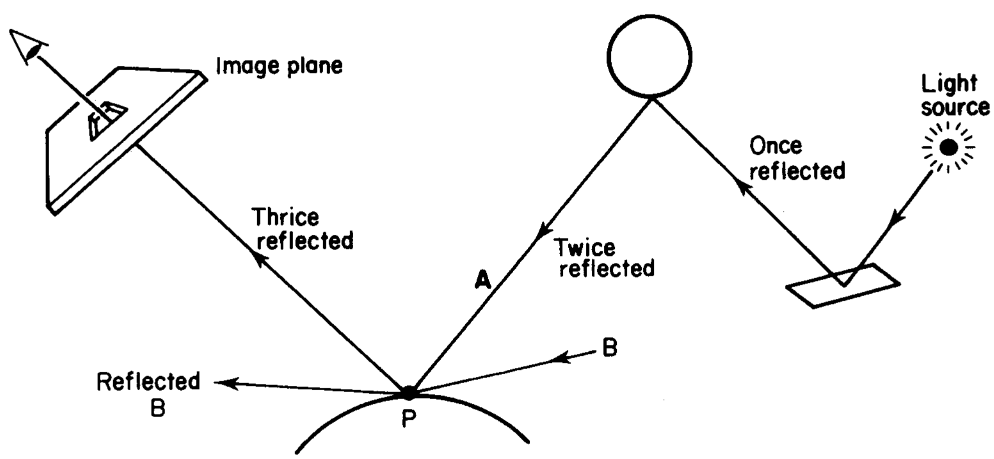


Fig. 9. The color of perfectly reflected light is dependent on the color of the object and the color of the incoming light that bounces off in the direction we care about. For example, at point P we want to know the color of the light coming in on ray **A**, since that light is then bounced into the eye.

Once we know the color of the light coming to the surface from the light sources, the reflected ray, and the transparency ray, we combine them according to the properties of the surface, and thus determine the total color leaving the surface in the direction of the incident ray.

We will see later in the book that more subtle effects can be accounted for if we use more than one transparency or reflected ray, sending them in a variety of (carefully chosen) directions and then weighting their results.

The subject of determining the way light behaves at a surface is called *surface physics*. This topic covers the geometry of light rays at a surface as well as what color changes happen to the light itself. We'll have an entire section of the course devoted to surface physics later on.

2.7 Transparency Rays

Just as there was a single direction from which light can be perfectly reflected into the direction of the incident ray, so is there a single direction from which light can be transmitted into the direction of the incident ray. The ray we create to determine the color of this light is called the *transmitted ray* or *transmitted ray*. *Figure 10* shows a possible path of a transmitted ray. Notice the bending, or *refraction*, of the light as it passes from one medium to another.

We follow the transmitted ray backwards to find which object might have radiated it, and then determine the color radiated by that object in the direction of the transmitted ray. When we know that color, we know the color of the transmitted ray, which (by construction) will be perfectly bent into the direction of the incident ray.

3 RECURSIVE VISIBILITY

The previous sections have discussed finding the color of light leaving a surface as a combination of different kinds of light arriving at the surface. In essence, the color of the radiated light is a function of the combined light from the light sources, light the object reflects, and light the object transmits. We found the colors of the reflected and transmitted light by finding the objects from which they started. But what was the color leaving this previous object? It was a combination of the light reaching it, which can be found with the same analysis.

This observation suggests a recursive algorithm, and indeed the whole ray tracing technique fits into that view very nicely.

The ray tracing process begins with a ray that starts at the eye; this is an *eye ray* or *pixel ray*. *Figure 11* shows one viewing set-up and a particular eye ray, labelled E.

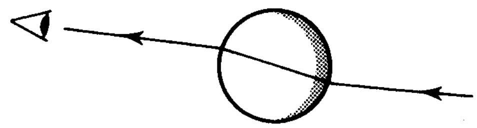


Fig. 10. Transmitted light arrives from behind a surface and passes through.

3.1 Surface Physics

We've mentioned above for reflection and transparency rays that we first find the direction they might have come from, and then look backwards along that path for a possible object at their source. The technique of determining these directions may be as simple or complicated as you like; we're approximating physical reality here, and physical reality is often complex in its details. The more accuracy you want from your model, the more detailed it will have to be. Happily, even fairly simple models seem to work very well for today's typical images.

The next step is the one that we'll repeat over and over again. We simply ask, "which object does this ray hit?" Remember that we're doing backwards ray tracing, so this question is really a confused form of the question, "given that a photon travelled along this ray to the eye, from which object did it start?"

In *Figure 11*, the eye ray hits plane 3, which we'll say is both somewhat transparent and reflective. We have two light sources, so we'll begin by sending out a shadow ray from plane 3 to each light: we'll call these rays $\mathbf{S}_1$ and $\mathbf{S}_2$. Since ray $\mathbf{S}_1$ reaches light A without interruption, we know that plane 3 is receiving light from light A. But ray $\mathbf{S}_2$ hits sphere 4 before it hits light B, so no illumination comes in along this path. Because plane 3 is both transparent and reflective, we also have to find the colors of the light it transmits and reflects; such light arrived along rays $\mathbf{T}_1$ and $\mathbf{R}_1$.

Following ray $\mathbf{T}_1$, we see that it hits sphere 6, which we'll say is a bit reflective. We send out two shadow rays $\mathbf{S}_3$ and $\mathbf{S}_4$ to determine the light hitting sphere 6, and create reflection ray $\mathbf{R}_2$ to see what color is reflected. Both $\mathbf{S}_3$ and $\mathbf{S}_4$ reach their respective light sources. Ray $\mathbf{R}_2$ leaves the scene entirely, so we'll say that it hits the surrounding world, which is some constant background color. That completes ray $\mathbf{T}_1$ from our original intersection with the primary ray $\mathbf{E}$.

Let's now go back and follow reflected ray $\mathbf{R}_1$. It strikes plane 9, which is a bit reflective and transparent. So we'll send out two shadow rays as always ($\mathbf{S}_5$ and $\mathbf{S}_6$), and reflected and transmitted rays $\mathbf{T}_2$ and $\mathbf{R}_3$. We'll then follow each of $\mathbf{T}_2$ and $\mathbf{R}_3$ in turn, generating new shadow and secondary rays at each intersection.

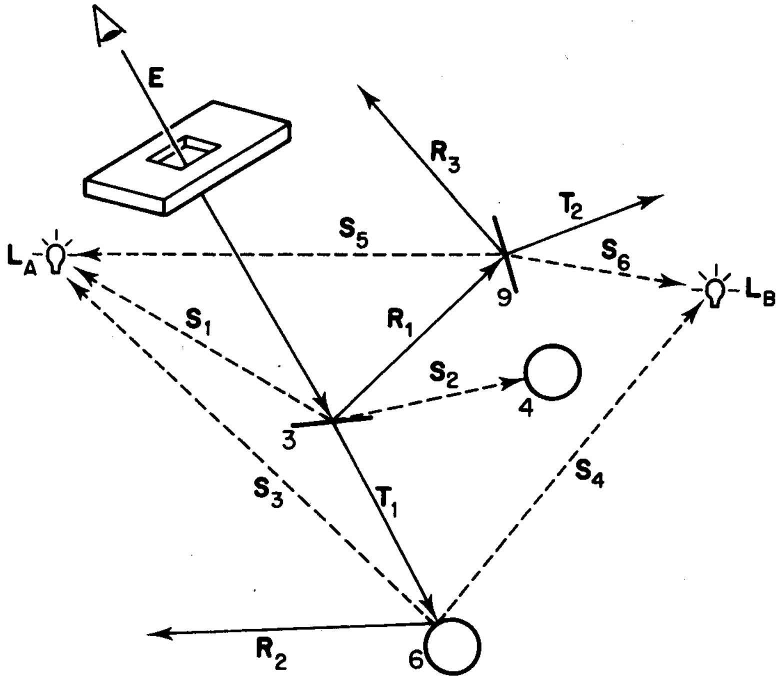


Fig. 11. An eye ray **E** propagated through a scene. Many of the intersections spawn reflected, transmitted, and shadow rays.

Figure 12 shows this whole process in a schematic form, called a *ray tree*. What ever causes the ray tree to stop? Like the non-opaque shadow ray question, the answers to this question are not easy. One ad hoc technique that usually works pretty well is to stop following rays either when they leave the scene, or their *contribution* gets too small. The former condition is handled by saying that if a ray leaves our world, then it just takes on the color of the surrounding background. The second condition is a bit harder.

How much contribution does ray **E** make to our picture? If it's the only ray at that pixel, then we'll use 100% of **E**; if the color **E** brings back is pure red, then that pixel will be pure red. But how about rays $\mathbf{T}_1$ and $\mathbf{R}_1$? Their contributions must be less than that of **E** since **E** is formed by adding them together. Let's arbitrarily say that plane 3 passes 40% of its transmitted light, and 20% of its reflected light (i.e. plane 3 is 40% transparent and 20% reflective).

Now recall that $\mathbf{T}_1$ is composed of the light radiated by sphere 6, given by $\mathbf{S}_3$, $\mathbf{S}_4$, and $\mathbf{R}_2$. Let's again be arbitrary, and say that object 6 is 30% transparent; thus $\mathbf{R}_2$ contributes 30% to $\mathbf{T}_1$. Since $\mathbf{T}_1$ contributes 40% to **E**, and $\mathbf{R}_2$ contributes 30% to $\mathbf{T}_1$, then $\mathbf{R}_2$ contributes only 12% to the final color of **E**. The farther down the ray tree we go, the less each ray will contribute to the color we really care about, the color of **E**.

So we can see that as we proceed down the ray tree, the contribution of

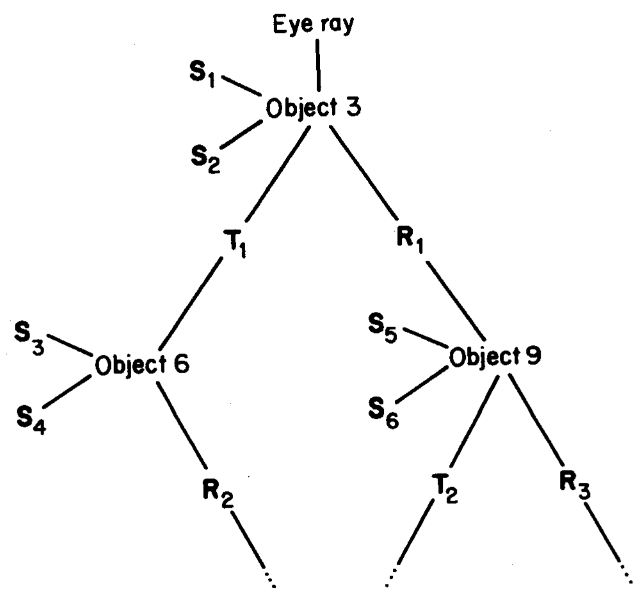


Fig. 12. The ray tree in schematic form.

individual rays to the final image becomes less and less. As a practical matter, we usually set a threshold of some kind to stop the process of following rays. It is interesting to note that although this technique, called *adaptive tree-depth control*, sounds plausible, and in fact works pretty well in practice, there are theoretical arguments that show that it can be arbitrarily wrong.

4 ALIASING

Synthesizing an image with a digital computer is very different from exposing a piece of film to a real scene. The differences are endless, although much of computer graphics research is directed to making the differences as small as possible. But there's a fundamental problem that we're stuck with: the modern digital computer cannot represent a continuous signal.

Consider using a standard tape recorder to record a trumpet. Playing the trumpet causes the air to vibrate. The vibrating air enters a microphone, where it is changed into a continually changing electrical signal. This signal is applied to the tape head, which creates a continually changing magnetic field. This field is recorded onto a piece of magnetic tape that is passing over the head.

Now let's consider the same situation on a digital computer. The music enters the microphone, and is changed into a continuous electrical signal. But the computer cannot record that signal directly; it must first turn it into a series of numbers. In formal terms, it *samples* the *signal* so that it can store it digitally. So our continuous musical tone has been replaced with a sequence of numbers. If we take enough samples, and they are of sufficiently high precision, then when we turn those numbers back into sound it will sound like the original music.

It turns out that these notions of 'enough samples' and 'sufficiently high precision' are critically important. They have been studied in detail in a branch of engineering mathematics called *signal processing*, from which computer graphics has borrowed many important results.

Let's look at a typical sampling problem by analogy. Imagine that you're at a county fair, standing by the carousel. This carousel has six horses, numbered 1 to 6, and it's spinning so that the horses appear to be galloping to the right. Now let's say that someone tells you that the carousel is making one complete rotation every 60 seconds, so a new horse passes by every 10 seconds. You decide to confirm this claimed speed of revolution.

Now just as you're watching horse 3 pass in front of you, someone calls your name. You turn and look for the caller, but you can't find anyone. It took you 10 seconds to look around. When you turn back to the carousel, you see that now horse 4 is in front of you. You might sensibly assume that the carousel has spun one horse to the right during your absence.

Say this happens again and again; you look away for 10 seconds, and then return. Each time you turn back you see the next-numbered horse directly in front of you. You could conclude that the carousel is spinning 1/6 of the way around every 10 seconds, so it takes 60 seconds to complete a revolution. Thus the claim appears true.

Now let's say that a friend comes back the next day to double-check your observations. As soon as she reaches the carousel (looking at horse 3) she hears someone calling her name. She looks around, but although you looked only for 10 seconds, your friend searches the crowd for 70 seconds. When she turns back to the carousel, she sees exactly what you saw yesterday; horse 4 is in front of her. If this happened again and again, she could conclude that the carousel is spinning 1/6 of the way around every 70 seconds. Thus she could reasonably state that the claim is false.

We know from your observations that it is certainly going faster than that, but there's no way for your friend to know that she's wrong if she only takes one look every 70 seconds.

In fact someone's measurements in such a situation can be arbitrarily wrong. Because as long as you regularly look at the carousel, look away, and look back, you have no idea what went on when you were looking away: I can always claim that it went around any number of full turns when you weren't looking!

The computer is prone to exactly the same problem. If it samples some signal too infrequently, the information that gets recorded can be wrong, just as our determination of the carousel's speed was wrong. The problem is that one signal (1/6 revolution every 10 seconds) is masquerading as another signal (1/6 revolution every 70 seconds); they're different signals, but after sampling we can't tell them apart. This problem is given the general term *aliasing*, to remind us that one signal is looking like another.

The problem of aliasing thoroughly permeates computer graphics. It shows up in countless ways, and almost always looks noticeably bad. The problem is that, if one is not careful, aliasing will almost always occur somewhere, simply due to the nature of digital computers and the nature of the ray tracing algorithm itself. Luckily, there are techniques to avoid aliasing, known collectively as *anti-aliasing* techniques. They are the weapons we employ to solve or reduce the aliasing problem.

We'll first look at some of the symptoms of aliasing, and then look briefly at some of the ways to avoid these problems.

4.1 Spatial Aliasing

When we get aliasing because of the uniform nature of the pixel grid, we often call that *spatial aliasing*. *Figure 13* shows a quadrilateral displayed at a variety of screen resolutions. Notice the chunky edges; this effect is colloquially called

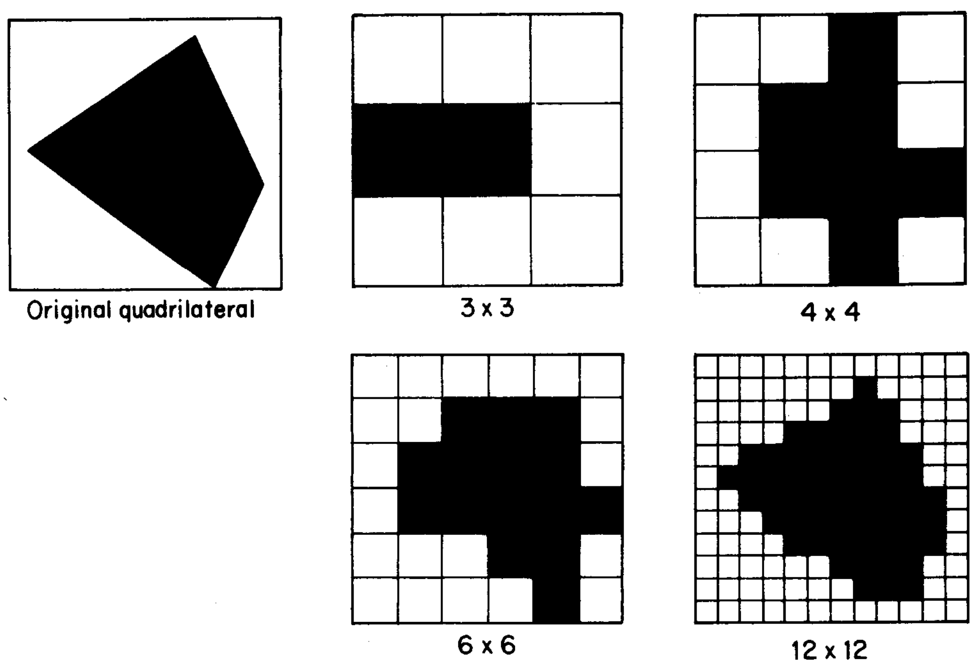


Fig. 13. A quadrilateral shown on grids of four different resolutions. Note that the smooth edges turn into stairsteps—commonly called 'jaggies.' No matter how high we increase the resolution, the jaggies will not disappear; they will only get smaller. Thus the strategy 'use more pixels' will never cure the jaggies!

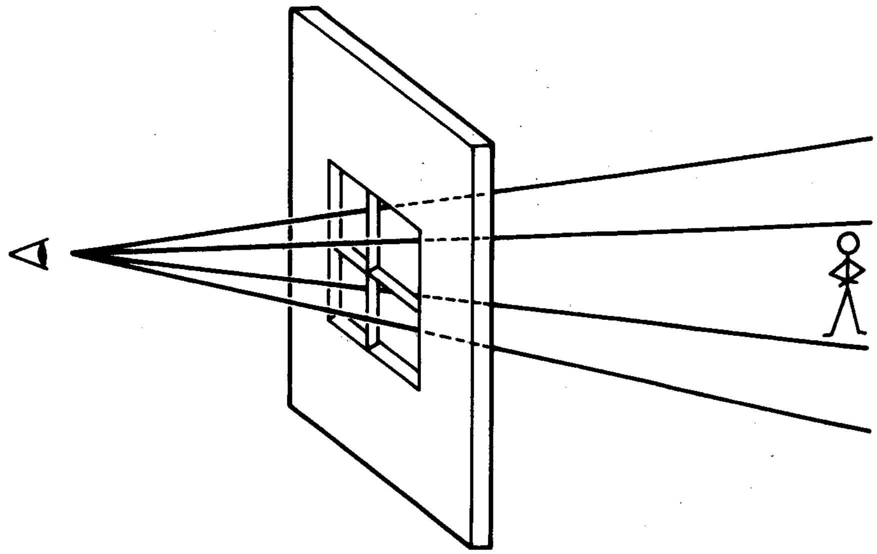


Fig. 14. No matter how closely the rays are packed, they can always miss a small object or a large object far enough away.

the jaggies, to draw attention to the jagged edge that should be smooth. Notice that the jaggies seem to become less noticeable at higher resolution. You might think that with enough pixels you could eliminate the jaggies altogether, but that won't work. Suppose you find that on your monitor can't see the jaggies at a resolution of 512 by 512. If you then take your 512-by-512 image to a movie theater and display it on a giant silver screen, each pixel would be huge, and the tiny jaggies would then be very obvious. This is one of those situations where you can't win; you can only suppress the problem to a certain extent.

Another aspect of the same problem is shown in *Figure 14*. Here a small object is falling between rays. Again, using more rays or pixels may diminish the problem, but it can never be cured that way. No matter how many rays you use, or how closely you space them together, I can always create an object that you'll miss entirely. You might think that if an object is that small, then it doesn't matter if it makes it into the image or not. Unfortunately, that's not true, and some good examples come from looking at temporal (or time) aliasing.

4.2 Temporal Aliasing

We often use computer graphics to make animated sequences. Of course, an animation is nothing more than many still frames shown one after another. It's tempting to imagine that if each still frame was very good, the animation would be very good as well. This is true to some extent, but it turns out that

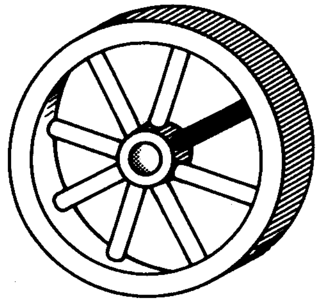


Fig. 15. A wheel with one black spoke.

when a frame is part of an animation (as opposed to just a single still, such as a slide), the notion of 'very good' changes. Indeed, new problems occur exactly because the stills are shown in an animated sequence: these problems fall under the class of *temporal aliasing* (temporal comes from the Latin *tempus*, meaning time).

Our example of the rotating carousel above was an example of this type of aliasing. Another, classical example of temporal aliasing is a spinning wheel. You may have noticed on television or in the movies that as a wagon wheel accelerates it seems to go faster and faster, and then it seems to slow down and start going backwards! When the wheel is going slowly, the camera can faithfully record its samples of the image on film (usually about 24 or 30 samples per second).

Figure 15 shows a wheel, with one spoke painted black. We're going to sample this clockwise-spinning wheel at 6 frames per second.

Figure 16(a) shows our samples when the wheel is spinning at 1 revolution per second; no problem, watching this film we would perceive a wheel slowly spinning clockwise. *Figure 16(b)* shows the same wheel at 3 revolutions per second: now we can't tell at all which way the thing is spinning. Finally, *Figure 16(c)* shows the same wheel at 5 revolutions per second; watching this film, we would believe that the wheel was spinning slowly backwards. This 'slowly backwards motion' is aliasing for the proper, forwards motion of the wheel.

The critical notion here is that things are happening too fast for us to record accurately.

Another problem occurs with the small objects mentioned in the previous section. As a very small object moves across the screen, it will sometimes be hit by a ray (and will thus appear in the picture), and sometimes it won't be hit by any rays. Thus, as the object moves across the screen it will blink on and off, or *pop*. Even for very small objects this can be extremely distracting, especially if they happen to contrast strongly with the background (like white stars in black space).

Another bad problem is what happens to some edges. *Figure 17* shows a horizontal edge moving slowly up the screen. Every few frames, it rises from

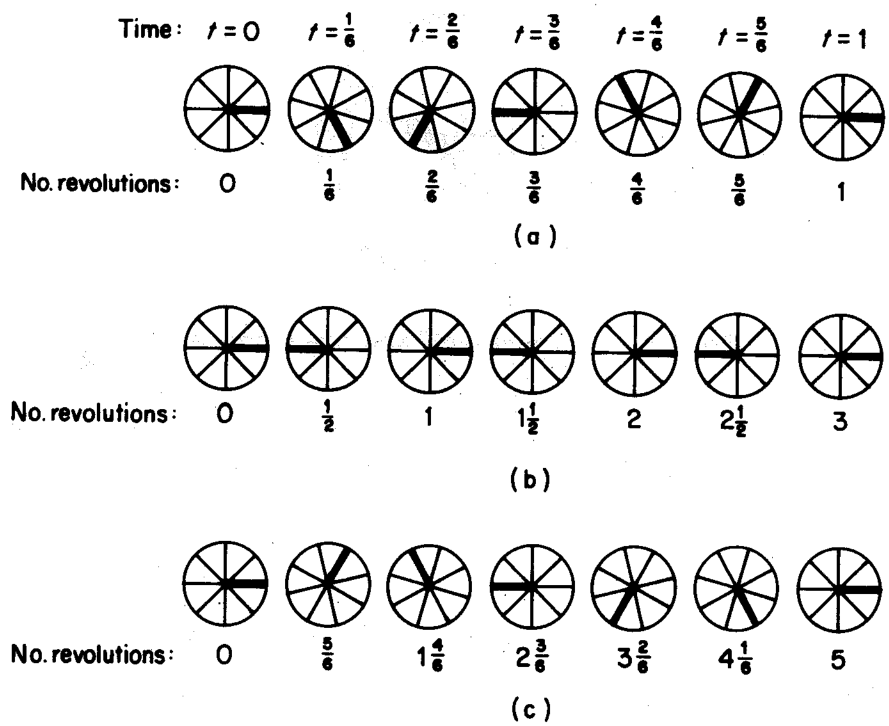


Fig. 16. A spinning wheel sampled at a constant 6 samples per second. In row (a) the wheel is spinning at 1 revolution per second and is correctly sampled. In row (b) the wheel is spinning at 3 revolutions per second; after sampling, we cannot tell in which direction the wheel is spinning! In row (c) the wheel is spinning at 5 revolutions per second, but appears to be spinning backwards at 1 revolution per second. Thus the very fast speed is aliasing as a slower speed after sampling.

one row of pixels to the next. This is another aspect of popping: the smoothly moving edge appears to jump from one line to the next in a very distracting manner.

Techniques that solve temporal aliasing problems usually create still frames that look blurry where things are moving fast. It's easy to see that this is just what happens when we use a camera to take a picture of quickly moving objects. Imagine taking a picture of a speeding race car as it whizzes past. Even though the shutter is open for a very brief moment, the car still moves fast enough to leave a streak, or blur, behind it on the film. Because of this characteristic of the frames, solutions to the problem of temporal aliasing are sometimes referred to as techniques for including *motion blur.*

4.3 Anti-aliasing

Aliasing effects can always be tracked down to the fundamental natures of digital computers and the point-sampling nature of ray tracing. The essential

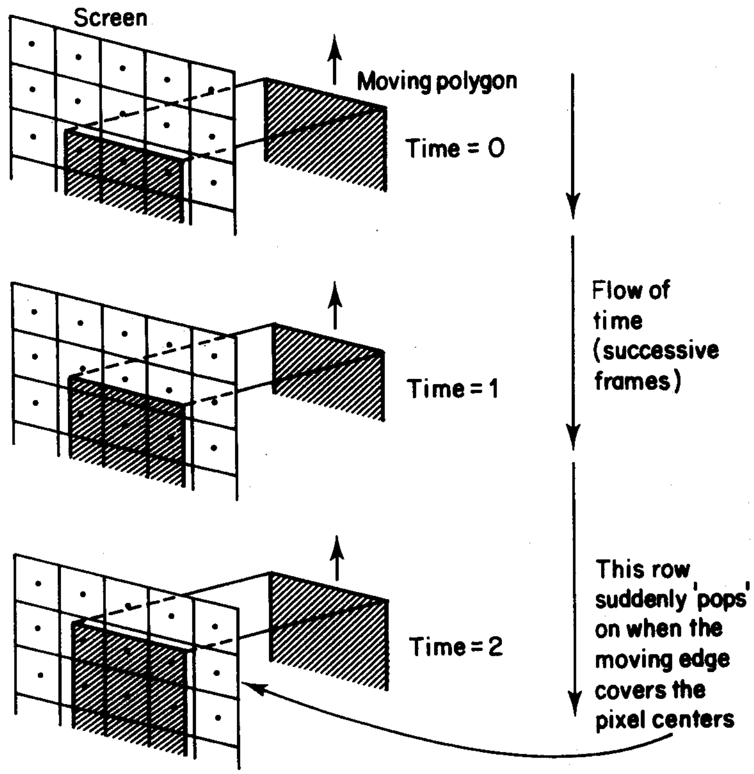


Fig. 17. A moving edge suddenly 'pops' when a new row of pixels is covered.

problem is that we're trying to represent continuous phenomena with discrete samples. Other aliasing effects abound in computer graphics; for example, frequency aliasing is very common but rarely handled correctly.

We will now consider several of the popular approaches to anti-aliasing. We'll focus on the problems of spatial aliasing, since they're easier to show on the written page than temporal aliasing. Nevertheless, many of these techniques apply to solving aliasing problems throughout computer graphics, and can be applied to advanced topics related to aliasing such as motion blur, correct texture filtering, and diffuse inter-reflections.

4.4 Supersampling

The easiest way to alleviate the effects of spatial aliasing is to use lots of rays to generate our image, and then find the color at each individual pixel by averaging the colors of all the rays within that pixel. This technique is called *supersampling*. For example, we might send nine rays through every pixel, and let each ray contribute one-ninth to the final color of the pixel.

Supersampling can help reduce the effects of aliasing, because it's a means for getting a better idea of what's seen by a pixel. If we send out nine rays in a given pixel, and six of the rays hit a green ball, and the other three hit a blue ball, the composite color in that pixel will be two-thirds green and one-third blue: a more 'accurate' color than either pure green or pure blue.

As we mentioned above, this technique cannot really *solve* aliasing problems, it just reduces them. Another problem with supersampling is that it's very expensive; our example will take nine times longer to create a picture than if we used just a single ray per pixel. But supersampling is a good starting point for better techniques.

4.5 Adaptive Supersampling

Rather than blindly firing off some arbitrary, fixed number of rays per pixel, let's try to concentrate extra rays where they'll do the most good. One way to go is to start by using five rays per pixel, one through each corner and one through the center, as in *Figure 18*. If each of the five rays is about the same color, we'll assume that they all probably hit the same object, and we'll just use their average color for this pixel.

If the rays have sufficiently different colors, then we'll subdivide the pixel into smaller regions. Then we'll treat each smaller region just as we did the whole pixel: we'll find the rays through the corners and center, and look at the resulting colors. If any given set of five rays are about the same color, then we'll average them together and use that as the color of the region; if the colors are sufficiently different, we'll subdivide again. The idea is that we'll send more rays through the pixel where there's interesting stuff happening, and in the boring regions where we just see flat fields of color we'll do no more additional work. Because this technique subdivides where the colors change, it

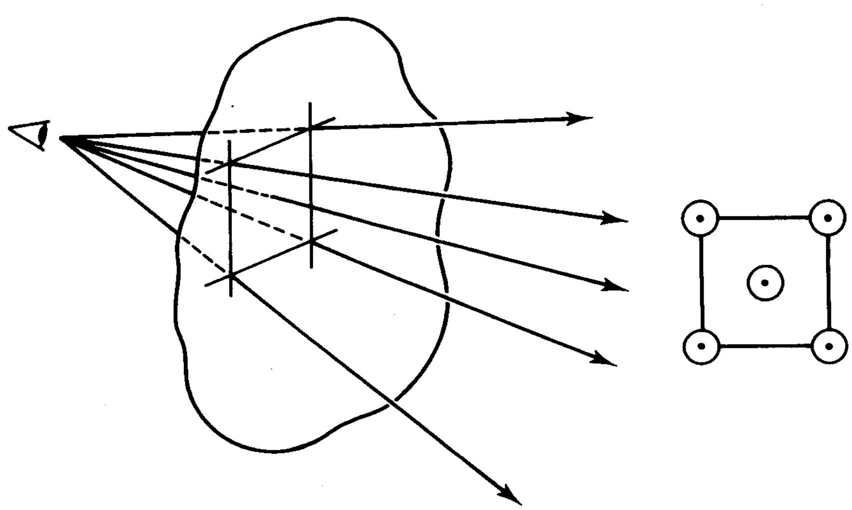


Fig. 18. Adaptive supersampling begins at each pixel by tracing the four corner rays and the center ray.

adapts to the image in a pixel, and is thus called *adaptive supersampling*. A detailed example of the process is shown in *Figure 19*.

This approach is easy, not too slow, and often works fairly well. But its fundamental assumption is weak. It's just not fair to assume that if some fixed number of rays are about the same color, that we have then sampled the pixel well enough. One problem that persists is the issue of small objects: little objects can slip through the initial five rays, and we'll still get popping as they travel across the screen in an animated sequence.

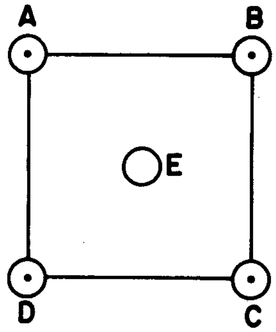
The central problems of adaptive supersampling are that it uses a fixed, arbitrary number of rays per pixel when starting off, and that it still uses a fixed, regular grid for sampling (although that grid gets smaller and smaller as we subdivide). Often this technique is fine when you need to quickly crank out a picture that just needs to look okay, but it can leave a variety of aliasing artifacts in your pictures. Happily, there are other approaches that solve aliasing problems better.

4.6 Stochastic Ray Tracing

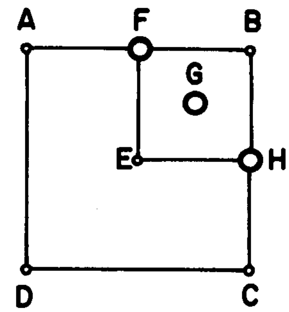
As we saw above, adaptive supersampling still ends up sending out rays on a regular grid, even though this grid is somewhat more finely subdivided in some places than in others. Thus, we can still get popping edges, jaggies, and all the other aliasing problems that regular grids give us, although they will usually be somewhat reduced. Let's get rid of the fixed grid, but continue to say that each pixel will initially be sampled by a fixed number of rays—we'll use nine. The difference will be that we'll scatter these rays evenly across the pixel. *Figure 20(a)* shows a pixel with nine rays plunked down more or less at random, except that they cover the pixel pretty evenly.

If each pixel gets covered with its nine rays in a different pattern, then we've successfully eliminated any regular grid. *Figure 20(b)* shows a small chunk of pixels, each sampled by nine rays, each of which is indicated by a dot. Now that we've gotten rid of the regular sampling grid, we've also gotten rid of the regular aliasing artifacts the grid gave us. Because we're randomly (or stochastically) distributing the rays across the space we want to sample, this technique is called *stochastic ray tracing*. The particular distribution that we use is important, so sometimes this technique is called *distributed ray tracing*.

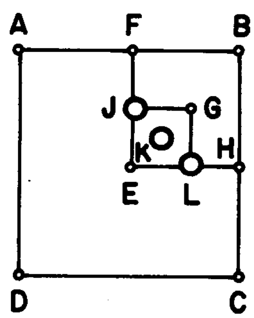
Let's consider another problem with the ray tracing algorithms described in preceding sections. Consider an incident ray which will carry light away from a somewhat bumpy surface. We'll see later in the course that when we consider diffuse reflection, there are many incoming rays that will send some of their energy away from the surface along the direction of the incident ray. There's no one 'correct' ray; they all contribute. One might ask which of these incoming rays should be followed? The answer that stochastic ray tracing



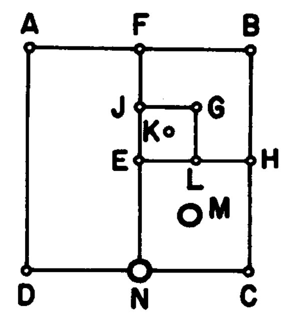
When we start a pixel, we trace rays through the four corners and the center. We then compare the colors of rays **AE**, **BE**, **CE**, and **DE**. Suppose **A** and **E** are similar and so are **D** and **E**, but both **BE** and **CE** are too different.



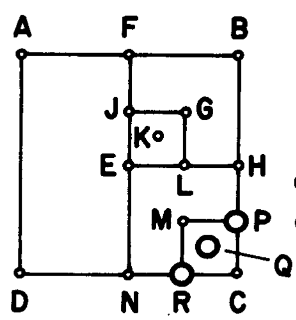
We'll start by looking more closely at the region bounded by **B** and **E**. We fire new rays **F**, **G**, **H** to find all four corners and the center of this region. We now compare **FG**, **BG**, **HG**, and **EG**. Suppose each pair is very similar, except **G** and **E**. So we look more closely at the region bounded by **G** and **E**.



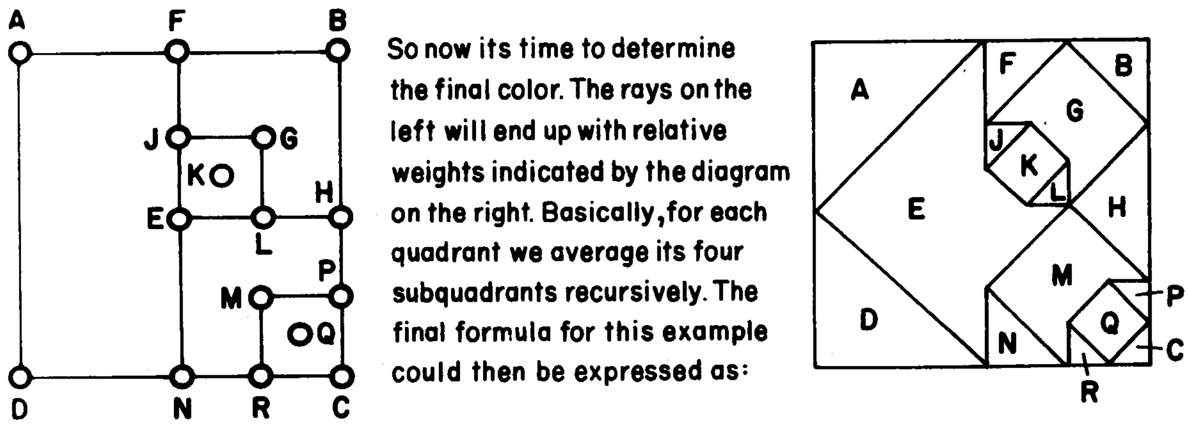
So now we fill in the square region bounded by **BE** with the three new rays **J**, **K**, and **L**. Let's suppose they're all sufficiently similar.



Now we return to the pair **CE** which we identified earlier. Since we already have **H**, we trace the new rays **M** and **N**. We compare the colors between **EM**, **HM**, **CM**, and **NM**. Suppose they are all similar except **CM**.



To complete the region we trace the new rays **P**, **Q**, and **R**. We compare **MQ**, **PQ**, **CQ**, and **RQ**. At this point we'll assume they're all sufficiently similar. These are no pairs of colors left to examine, so we're now done.



$$\frac{1}{4}\left(\frac{A+E}{2}+\frac{D+E}{2}+\frac{1}{4}\left[\frac{F+G}{2}+\frac{B+G}{2}+\frac{H+G}{2}+\frac{1}{4}\left\{\frac{J+K}{2}+\frac{G+K}{2}+\frac{L+K}{2}+\frac{E+K}{2}\right\}\right]\right.$$
$$\left.+\frac{1}{4}\left[\frac{E+M}{2}+\frac{H+M}{2}+\frac{N+M}{2}+\frac{1}{4}\left\{\frac{M+Q}{2}+\frac{P+Q}{2}+\frac{C+Q}{2}+\frac{R+Q}{2}\right\}\right]\right)$$

Fig. 19. Adaptive supersampling.

provides is that there is no single best incoming ray direction. Instead, choose a random ray direction. The next time you hit a surface and need to spawn new rays, choose a new random direction. The trick is to bias your random number selection in such a way that you send lots of rays in directions where it's likely a lot of light is arriving, and relatively few rays in directions where the incoming light is sparse.

We can describe this problem mathematically as an integration problem, where we want to find the total light arriving at a given point. But because we can't solve the integration equation directly, we sample it randomly and hope that after enough random samples we'll start getting an idea of the answer. In

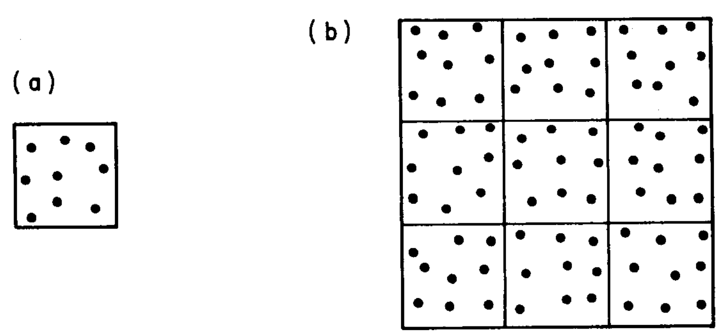


Fig. 20. We can use stochastic sample points within each pixel to help reduce spatial aliasing.

fact, our random selections can be carefully guided to help us obtain a good answer with a small number of samples.

The techniques of stochastic ray tracing lead to a variety of new effects that the deterministic ray tracing algorithms described above don't handle well or at all. For example, stochastic ray tracing helps us get *motion blur, depth of field*, and soft edges on our shadows (known as the *penumbra* region).

But although stochastic ray tracing solves many of the problems of regular ray tracing, we've picked up something new: *noise*. Since we're getting a better average with this technique, every pixel comes out more or less correct, but it's usually not quite right. This error isn't correlated to a regular grid like many of the other aliasing problems we've discussed, but instead it spreads out over the picture like static in a bad TV signal. It turns out that the human visual system is much more forgiving of this form of random noise than regular aliasing problems like the jaggies, so in this way stochastic ray tracing is a good solution to aliasing problems.

But we're still using all those rays for every pixel, even where we don't need them. Sometimes we do need them: consider a pixel that's looking out on a patchwork quilt, where one pixel sees just one red square. Then just one or two rays certainly give us the correct color in this pixel. On the other hand, consider a pixel that can see 16 differently colored squares of material. We'll need at least 16 rays, just to get one of each color. It's not clear from the above discussion how to detect when we need more samples, nor how to go about getting them.

4.7 Statistical Supersampling

One way to try getting just the right number of rays per pixel is to watch the rays as they come in. Imagine a pixel which has had four rays sent through it, distributed uniformly across the pixel. We can stop sampling if those four rays are a 'good enough' estimate of what's really out there.

We can draw upon the vast body of statistical analysis to measure the quality of our estimate and see whether some set of samples are 'good enough.' We'll look at the colors of the rays we've sent through the pixel so far, and perform some statistical tests on them. The results of these tests are a measure of how likely it is that these rays give us a good estimate of the actual color that pixel can see. If the statistics say that the estimate is probably poor, we'll send in more rays and run the statistics again. As soon as the color estimate is 'good enough,' then we'll accept that color for that pixel and move on. This is called *statistical supersampling*.

The important thing here is to determine how good is 'good enough.' In general, you can specify just how confident you'd like to be about each pixel. For example, you might tell your program to continue sending rays through pixels until the statistics say that it's 90% likely that the color you have so far is the 'true,' or correct, color. If you want the picture to finish faster, you might drop that requirement to 40%, but the quality will probably degrade.

4.8 The Rendering Equation

We can express how light bounces around in a scene mathematically, with a formula called *the rendering equation*. A solution to the rendering equation tells us just how light is falling on each of the objects in our scene. If one of those objects is an image plane, then the solution to the rendering equation is also a solution to the problem of computer graphics: what light is falling on that image plane?

The rendering equation is useful for several reasons. In one respect, it acts as a scaffolding upon which we can hang most of what we know about how light behaves when it bounces off a surface. In another respect, it tells us how light 'settles down' when a light source has been turned on in a scene and left on for a while. In this sense the rendering equation can help us use the power of *radiosity* techniques to model diffuse inter-reflections. The rendering equation also provides a nice synopsis of most of what we know about the behavior of light for image synthesis, and may provide for some new effects, such as caustics.

We mention the rendering equation here because one powerful way to solve it is by ray tracing. Specifically, an enhanced version of stochastic ray tracing (using techniques called *importance sampling* and *path tracing*) can help us find solutions to the rendering equation.

ANNOTATED BIBLIOGRAPHY

Key

[AA] Anti-aliasing; [AN] animation; [EF] efficiency; [OI] object intersections; [RT] ray tracing technique; [SH] shading; [VI] visibility.

Siggraph '77: *Comput. Graph.* **11**(2), July 1977.
Siggraph '78: *Comput. Graph.* **12**(3), August 1978.
Siggraph '79: *Comput. Graph.* **13**(2), August 1979.
Siggraph '80: *Comput. Graph.* **14**(3), July 1980.
Siggraph '81: *Comput. Graph.* **15**(3), August 1981.
Siggraph '82: *Comput. Graph.* **16**(3), July 1982.
Siggraph '83: *Comput. Graph.* **17**(3), July 1983.
Siggraph '84: *Comput. Graph.* **18**(3), July 1984.
Siggraph '85: *Comput. Graph.* **19**(3), July 1985.
Siggraph '86: *Comput. Graph.* **20**(4), August 1986.

1. Amanatides, J., Ray tracing with cones. Siggraph '84 [AA].
2. Barr, A.H., Ray tracing deformed surface. Siggraph '86 [OI].
3. Blinn, J.F., Models of light reflection for computer synthesized pictures. Siggraph '77 [SH].
4. Bouville, C., Bounding ellipsoids for ray–fractal intersection. Siggraph '85 [EF].
5. Bronsvoort, W.F. and Klok, F. Ray tracing generalized cylinders. *ACM Trans. Graph.* **4**(4), October 1985 [OI].
6. Bui-Tuong Phong, Illumination for computer generated pictures. *Commun. ACM* **18**(6), June 1975 [SH].
7. Clark, J.H., Hierarchical geometric models for visible surface algorithms. *Commun. ACM* **19**(10), October 1976 [EF].
8. Cook, R.L. and Torrance, K.E., A reflectance model for computer graphics. *ACM Trans. Graph.* **1**(1), January 1982 [SH].
9. Cook, R.L., Porter, T. and Carpenter, L., Distributed ray tracing. Siggraph '84 [RT].
10. Crow, F.C., 'The aliasing problem in computer-generated shaded images.' *Commun. ACM* **20**(11), November 1977 [AA].
11. Dippé, M.A.Z. and Wold, E.H., Antialiasing through stochastic sampling. Siggraph '85 [AA].
12. Fujimoto, A., Tanaka, T. and Iwata, K., ARTS: Accelerated Ray Tracing System. *IEEE Comput. Graph. Appl.* **6**(4), April 1986 [EF].
13. Glassner, A.S., Fast ray tracing by space subdivision. *IEEE Comput. Graph. Appl.* **4**(10), October 1984 [EF].
14. Glassner, A.S., Spacetime ray tracing for animation. *IEEE Comput. Graph. Appl.* **8**(2), March 1988 [EF, RT, AN].
15. Hall, R.A. and Greenberg D.P., A testbed for realistic image synthesis. *IEEE Comput. Graph. Appl.* **3**(8), November 1983 [SH].
16. Hanrahan, P., Ray tracing algebraic surface. Siggraph '83 [OI].
17. Heckbert, P.S. and Hanrahan, P., Beam tracing polygonal objects. Siggraph '84 [RT].
18. Joy, K.I. and Bhetanobhotla, M.N., Ray tracing parametric surface patches utilizing numerical techniques and ray coherence. Siggraph '86 [OI].
19. Kajiya, J.T., Ray tracing parametric patches. Siggraph '82 [OI].
20. Kajiya, J.T., New techniques for ray tracing procedurally defined objects. Siggraph '83 [OI].
21. Kajiya, J.T. and Von Herzen, B., Ray tracing volume densities, Siggraph '84 [OI].
22. Kajiya, J.T., The rendering equation. Siggraph '86 [RT].
23. Kay, D.S., Transparency, Refraction, and Ray Tracing for Computer Synthesized Images. M.S. Thesis, Cornell University, January 1979 [RT].
24. Kay, T.L. and Kajiya J., Ray tracing complex scenes. Siggraph '86 [EF].
25. Lee, M.E., Redner, R.A. and Uselton, S.P., Statistically optimized sampling for distributed ray tracing. Siggraph '85 [RT].
26. Newman, W. and Sproull R.F., *Principles of Interactive Computer Graphics*, 2nd Edition. McGraw-Hill, New York, 1979 [VI].
27. Potmesil, M. and Chakravarty, I., Synthetic image generation with a lens and aperture camera model. *ACM Trans. Graph.* **1**(2), April 1982 [VI].
28. Roth, S., Ray casting for modelling solids. *Comput. Graph. Image Process.* **18**, 1982 [RT].

29. Rubin, S.M. and Whitted T., A 3-dimensional representation for fast rendering of complex scenes. Siggraph '80 [EF].
30. Sederberg, T., Ray tracing steiner patches. Siggraph '84 [OI].
31. Speer, R., DeRose T.D., and Barsky B.A., A theoretical and empirical analysis of coherent ray tracing. *Proceedings of Graphics Interface 85*, May 1985 [EF].
32. Sweeney, M.A.J. and Bartels R.H., Ray tracing free-form B-spline surfaces. *IEEE Comput. Graph. Appl.* **6**(2), February 1986 [OI].
33. Toth, D.L., On ray tracing parametric surfaces. Siggraph '85 [OI].
34. Weghorst, H., Hooper G. and Greenberg D.P., Improved computational methods for ray tracing, *ACM Trans. Graph.* **3**(1), January 1984 [EF].
35. Wijk, J.J. Van, Ray tracing objects defined by sweeping planar cubic splines. *ACM Trans. Graph.* **3**(3), July 1984 [OI].
36. Wijk, J.J. Van, Ray tracing objects defined by sweeping a sphere. In *Proceedings of the Eurographics '84 Conference*, North-Holland, Amsterdam, 1984 [OI].
37. Whitted. T., An improved illumination model for shaded display. *Commun. ACM* **23**(6), June 1980 [RT].