17
Rendering

Rendering is a general term for methods that display a realistic-looking three-dimensional solid object on a two-dimensional output device (normally screen or paper). Perhaps the simplest way to render an object is to display its surface as a wireframe. The next step in rendering is to display, as a wireframe, only those parts of the surface that would be visible in real life. More realistic rendering is achieved by shading—computing the amount and color of the light emitted from every point of the surface. Complete realism may be achieved by simulating surface texture, reflections from neighboring surfaces, and shadows cast by all the objects in the scene.

Note. For years, the goal of rendering was to produce images that looked as real as possible. This never-ending quest for realism provided powerful impetus for legions of programmers, researchers, and engineers. The harder it was for a human observer to decide whether a given image was real or synthetic, the better the rendering was considered. However, because the human mind is always in motion—always looking for new ways, new explanations, and new achievements—it is no wonder that at a certain point in the history of computer graphics, several researchers decided to explore other ways to render images. This trend has become known as non-photorealistic rendering and it includes rendering images in terms of dots and strokes, making them look like comics or like watercolor paintings, and applying interesting and artistic distortions.

The last few sections of this chapter discuss approaches to non-photorealistic rendering. Readers looking for more on this topic can try [Strothotte and Schelchtweg 02].

D. Salomon, *The Computer Graphics Manual*, Texts in Computer Science,
DOI 10.1007/978-0-85729-886-7_17, © Springer-Verlag London Limited 2011

17.1 Introduction

Producing a computer-generated image is a multistep process whose main steps are as follows:

1. The designer/user has to specify the objects in the scene (the image), their shapes, positions, orientations, and surface color/texture.

2. He should select the viewer's position and direction of view. The computer then transforms the points defining each object to create a perspective projection.

3. An algorithm should now be executed, to determine what parts of each object are visible to the viewer. This is the hidden-surface removal problem (often referred to as visible surface determination). All parts of all objects in the image must be checked, and only those that are visible to the viewer are actually displayed.

4. The objects in the scene are displayed by simulating lighting. The designer/user has to define the light source (or sources), their positions, shapes, intensities, and colors. The light emanating from any surface in the scene is a combination of (1) light coming from light sources (direct lighting) and reflected by the surface, (2) light coming from other surfaces (indirect) and reflected by our surface, (3) light generated by the surface (if it happens to be one of the light sources), and (4) light transmitted by the surface (if it happens to be transparent or translucent).

5. The image (complete scene) is now displayed by rendering software that computes the amount and color of light reaching the viewer's eye from any point in the image, and then displays that point.

Current (2010) computers often have special hardware to implement perspective projections, hidden-surface elimination, and direct illumination. Everything else requires software. The most important rendering task done by software is illumination, both direct and indirect. The latter is important when the image contains shiny surfaces, each reflecting some of the others. Several methods for indirect illumination are currently popular, *ray tracing* (Section 17.5), *photon mapping* (Section 17.6), and *radiosity*. These methods use very different approaches to compute the light reflection.

Ray tracing was introduced by Turner Whitted of Bell Laboratories in 1979. The main idea is to trace the path of a light ray from the eye of the observer through each pixel on the screen into the image. If the ray strikes a surface, the algorithm spawns reflected or refracted rays which, in turn, are traced to see if they intersect any other surfaces. The final color and intensity of each pixel are determined by adding up the light contributed by each spawned ray.

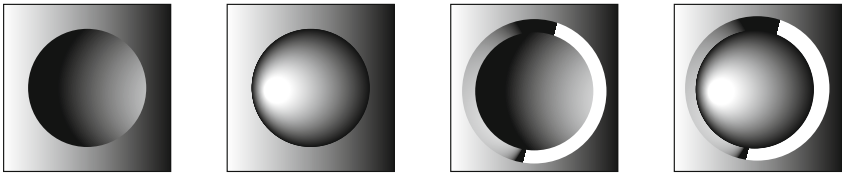
Ray tracing produces realistic images but is view dependent. This means that the entire computation must be repeated when the viewer's position is changed. Ray tracing is also too slow to generate real-time sequences of pictures, since each image may take minutes or more to compute.

The radiosity method, developed at Cornell in 1984, is view independent: Given an image, the calculations need be made only once. Once the global illumination has been determined, it is easy to create a series of images by moving the viewer to different viewpoints. Indeed, the method can be used to generate real-time sequences of images, which makes it useful for applications such as flight simulation and architecture (walking through a newly designed structure).

Radiosity uses conservation of energy to compute the light intensity for each surface in a scene that consists of ideal diffuse surfaces (either light sources or reflectors). An equation is written for the radiosity of each surface in the image—the intensity of light emanating from the surface—as a function of the radiosity of all the other surfaces.

17.2 A Simple Shading Model

The simplest shading technique simulates light reflection from the surface to be rendered. It assumes a light source (which itself may not have to be displayed) at a certain point. It assumes that the viewer is located at the center of perspective and that there is an environment around the object—such as walls and furniture—that can reflect more light on the object and can cast shadows. It then uses a mathematical model to calculate the intensity (and color) of the light reflected from every pixel on the surface of the object. Such a calculation may be very complex, depending on the model used. Figure 17.1 shows how a flat drawing can be made to look real (i.e., three-dimensional) by simulating reflection. It is easy to tell which of the four buttons are convex and which are concave. It is also easy to tell that the bevels around the buttons in Figure 17.1c,d face the viewer.



(a) (b) (c) (d)

Figure 17.1: Light Reflection from Buttons.

The resulting shaded image depends on the following entities:

- The light source. Its intensity, color, shape, direction, and distance. It can be a point source, or a large source, such as a window or a light fixture.
- The surface of the object. It can vary from shiny to dull, from smooth to rough, and from bright to dark. It can have several colors, can be opaque, transparent or translucent (diffusing light so that objects beyond it cannot be clearly distinguished).
- The environment. Objects seen in empty space, without any background to reflect light on them, look harsh. Imagine a spaceship in deep space, away from any reflecting planets. Those parts of the ship illuminated by direct starlight are very bright, while parts that are in the shade are completely dark. The result is that we see the ship mostly in black and white, with few grays or colors. A realistic shading model should therefore consider light reflection from other objects and from nearby walls.

In general, a ray of light striking a surface is partly absorbed, partly transmitted (and also refracted), and partly reflected. The following three sections briefly discuss these phenomena. Following these, we describe the simple shading model.

17.2.1 Absorption

Light absorption is a phenomenon that depends on the material and on the wavelength of the light. Material that absorbs all wavelengths except blue (which it reflects or transmits) looks blue. Thus, absorption of certain wavelengths of light determines the color and brightness of the surface.

17.2.2 Refraction

When light moves from one medium to another, such as from air to glass to water, its speed changes. The denser the medium, the slower the light travels. When we talk about the speed of light, we implicitly mean its speed in vacuum (where it is fastest). The result of the speed change is that a beam of light changes its direction of motion and bends when it enters a different medium (Figure 17.2). This phenomenon is called *refraction* (see Plate B.3). Notice that it affects all electromagnetic radiation (X-rays, radio waves, microwaves, etc.), not just light.

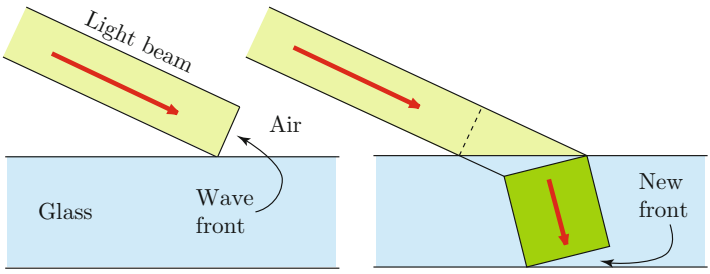


Figure 17.2: Bending of Light as a Result of Speed Change.

Figure 17.3a shows why refraction is important in computer graphics. When a ray of light moves from air to glass and again to air, it bends twice, in opposite directions, so it comes out of the glass in its original direction but with its position shifted. An observer looking at an object through the glass will therefore see the object shifted away from its original position. Thus, realistic-looking computer-generated images should simulate refraction.

Figure 17.3e shows a ray of light traveling in the air entering a slab of glass at an angle α to the normal of the glass surface. Inside the glass, the ray bends and it now moves at an angle β with respect to the normal. The rule of refraction, discovered experimentally by the Dutch mathematician Willebrord Snell in 1621, is

$$\frac{\sin\alpha}{\sin\beta} = \frac{C_1}{C_2} = C,$$

where C_1 and C_2 are the speeds of light in air and glass, respectively, and C, their ratio, is called the *refraction coefficient* of air and glass. This is Snell's law. (The *index of refraction* of a medium M is defined as the ratio of the speed of light in a vacuum and the speed of light in M. Hence, the refraction coefficient of two media is the ratio of their refraction indexes.)

Willebrord Snellius (born Willebrord Snel van Royen) was a Dutch astronomer and mathematician who lived and worked in Leiden from 1580 to 1626 (he succeeded his father as professor of mathematics at the University of Leiden).

For centuries, his name has been attached to the law of refraction of light, but it is now known that this law was understood empirically in ancient times by Ptolemy, was mentioned in the middle ages by Witelo, and was first described rigorously by Ibn Sahl in 984.

In addition to this well-known law, Snell's name is also known from a method for determining the radius of the Earth that he originated and executed in 1615. The idea was to employ triangulation to measure the distance of one point on the Earth from the parallel of latitude of another point. Another achievement of Snell was an algorithm for computing the value of π.

How does the change of speed cause the light to change its direction? This can be explained by means of a general physical principle called *the principle of least time*. It was proposed by Pierre Fermat around 1650, so it is sometimes called *Fermat's principle*. It says that light chooses the particular path in air and glass that takes the *shortest time* to traverse. Using this principle, it is easy to prove that the path of least time is the one obeying Snell's law. Figure 17.3b shows an analogous situation. A lifeguard is stationed on a beach and there is a swimmer in the water. The swimmer starts drowning and the lifeguard starts running toward him. The best path for the lifeguard (from the point of view of the swimmer) is that of least time. Path b is a straight line. This may be the intuitive choice of many lifeguards, but it may not be the path of least time since swimming is slower than running. Path d minimizes the swimming time, but there is no guarantee that it is the right path. Intuitively, it seems that the right path is somewhere between paths b and d since it is clear that paths such as a and e require longer times.

We now show that the path of least time is the one that satisfies Snell's law. The proof is short and elegant and its geometry is shown in Figure 17.3c. We assume that the best path is the one that hits the water at point $\mathbf{P}$, and we then try another path that hits at point $\mathbf{Q}$, close to $\mathbf{P}$. Figure 17.3d shows how the curve of travel time versus point of hit has a minimum at $\mathbf{P}$. Since point $\mathbf{Q}$ is close to $\mathbf{P}$ and since the curve is continuous, we expect only a very small difference in the travel times of the rays that hit the water at points $\mathbf{P}$ and $\mathbf{Q}$. Another way to express this is to say that we expect the travel times of the two rays to be essentially the same in the first approximation, because the curve of Figure 17.3d is close to flat (horizontal) at point $\mathbf{P}$.

The proof should therefore figure out the difference between the travel times along the two paths and set that difference to zero. This will generate an equation whose solution should produce Snell's law. The first step is to draw a perpendicular to LP

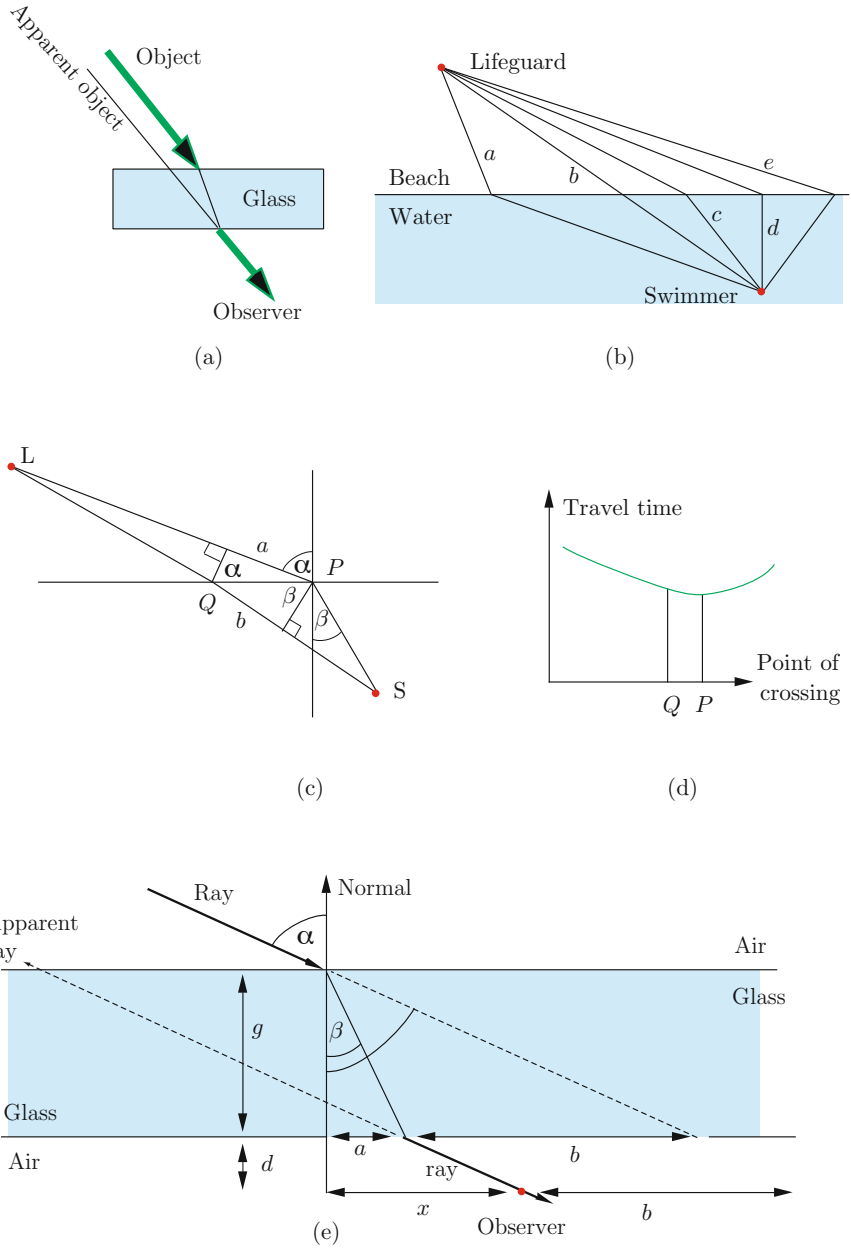


Figure 17.3: Refraction.

that passes through point **Q**. This shows that path LPS has to travel on the beach a distance of a units longer than path LQS. It takes a/C_1 time units to travel distance a. The second step is to draw a perpendicular to line QS that passes through point **P**. This shows that path LQS has to travel in the water a distance b longer than path LPS. It takes b/C_2 time units to travel distance b. Denoting by d the distance PQ, we get $\sin\alpha = a/d$ and $\sin\beta = b/d$ from elementary trigonometry. We can now write

$$\begin{aligned}
&\text{time}(a) = \text{time}(b)\\
&\Rightarrow \frac{a}{C_1} = \frac{b}{C_2}\\
&\Rightarrow \frac{d\sin\alpha}{C_1} = \frac{d\sin\beta}{C_2}\\
&\Rightarrow \frac{\sin\alpha}{C_1} = \frac{\sin\beta}{C_2}\\
&\Rightarrow \frac{\sin\alpha}{\sin\beta} = \frac{C_1}{C_2}.
\end{aligned}$$

◇ **Exercise 17.1:** Prove Snell's law using just elementary calculus and trigonometry.

Snell's law is mathematically simple, so people generally like to think of it as an explanation of refraction. A ray of light hits a glass surface and bends by the right amount, depending on the angle of incidence. It is easy for us to imagine that the light "knows" at what angle it hits and what medium it is entering, so it changes its direction of motion accordingly. The least-time principle, on the other hand, even though more elegant, is harder to accept as an explanation. The problem is: How does light know in advance what the least time path is? When we humans are faced with such a problem, we have to try different paths, we hesitate, we need to perform calculations, but light does not hesitate, does not seem to try different paths, and always selects the right path confidently.

Quantum electrodynamics provides a completely different explanation to refraction. Advanced readers are referred to pages 49–52 of [Feynman 85]. Another book by Feynman, *The Feynman Lectures on Physics*, (volume 1, chapter 26, page 5) describes a few common phenomena, such as a mirage, that are caused by refraction.

Figure 17.3e illustrates the refraction problem as it typically occurs in practice (i.e., in computer graphics applications). A ray of light passes through a thick slab of glass and is observed on the other side of the glass. The known quantities are the angle of incidence α, the thickness g of the glass, the vertical distance d between the observer and the glass, and the refraction coefficient $C = C_1/C_2$. The unknown quantity is x, the horizontal distance between the observer and the point of incidence. Once x is known, we know where the observer should be positioned in order to see the (refracted) ray. The derivation is elementary and uses similar triangles:

$$\begin{aligned}
&a = g\tan\beta, \quad a + b = g\tan\alpha,\\
&b = g\tan\alpha - a = g\tan\alpha - g\tan\beta, \quad x + b = (g + d)\tan\alpha,\\
&x = g\tan\alpha + d\tan\alpha - b
\end{aligned}$$

$$
\begin{aligned}
&= g\tan\alpha + d\tan\alpha - g\tan\alpha + g\tan\beta \\
&= d\tan\alpha + g\tan\beta \\
&= d\tan\alpha + \frac{g\sin\alpha}{\sqrt{C^2 - \sin^2\alpha}}.
\end{aligned} \tag{17.1}
$$

The last equality is true because

$$
\begin{aligned}
\tan\beta = \frac{\sin\beta}{\cos\beta} &= \frac{\sin\alpha\sin\beta}{\sin\alpha\sqrt{1-\sin^2\beta}} \\
&= \frac{\sin\alpha\sin\beta}{\sqrt{\sin^2\alpha - \sin^2\alpha\sin^2\beta}} \\
&= \frac{\sin\alpha}{\sqrt{\left(\frac{\sin\alpha}{\sin\beta}\right)^2 - \sin^2\alpha}} = \frac{\sin\alpha}{\sqrt{C^2 - \sin^2\alpha}}.
\end{aligned}
$$

It is easy to test Equation (17.1) for the case $C_1 = C_2$, where the light goes from a medium M to the same medium M. In this case, there should be no refraction, so the equation should yield $\alpha = \beta$. Substituting $C = 1$ in Equation (17.1) yields

$$
x = d\tan\alpha + \frac{g\sin\alpha}{\sqrt{1-\sin^2\alpha}} = d\tan\alpha + \frac{g\sin\alpha}{\cos\alpha} = (d+g)\tan\alpha.
$$

Figure 17.3e shows that $x = (d+g)\tan\alpha$ implies $b = 0$ or $\alpha = \beta$.

◇ **Exercise 17.2:** The SI (Système International) definition of the meter was adopted at the 1983 Conference Générale des Poids et Mesures. It says "the meter is the length of the path traveled by light in vacuum during a time interval of 1/299,792,458 of a second." This defines the speed of light in vacuum to be exactly 299,792,458 m per s. The speed of light in typical glass fiber is roughly 33% less, or about 200,000 km per s. The refraction coefficient C from vacuum to glass is, therefore, approximately 1.5. Using Equation (17.1), calculate and plot the distance x as a function of the angle of incidence α for the case $d = 0$ and $g = 1$.

17.2.3 Reflection

When a ray of light hits a mirror, it is reflected. The direction of reflection is determined by the following simple rule: The angle of reflection equals the angle of incidence (the angles are measured between the rays of light and the normal to the surface). This rule can also be elegantly deduced from Fermat's principle. Figure 17.4 shows a ray traveling from point **A** to a mirror M, getting reflected, and arriving at point **B**. What path takes the ray from **A** to the mirror and to **B** in the least time?

Consider path ADB. The travel time from **A** to **D** is minimal, but the travel time from **D** to **B** is much longer. If we move a bit to the right and let the ray hit the mirror at, say, **E**, we slightly increase the travel time AE but greatly decrease the travel time EB. To find the best point, we use an elegant "trick." We construct an imaginary point $\mathbf{B}'$ on the other side of the mirror, at the same distance as **B**. The total travel time AEB

equals the travel time AEB′, since the speed of light on both sides of the mirror is the same. It is now clear that the minimal-time path AB′ is also the minimal distance path AB′, i.e., a straight line. We denote by **C** the point where this line intercepts the mirror. Since line ACB′ is straight, and since BF=FB′, we conclude that angle BCF equals angle B′CF which, in turn, equals angle ACM. This implies that the angle between direction AC and the normal equals the angle between direction CB and the normal.

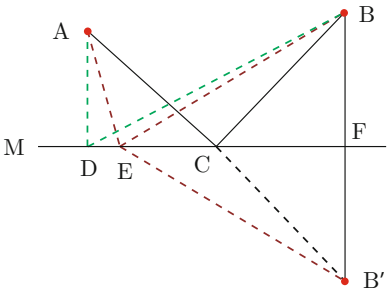


Figure 17.4: Least Time in Reflection.

In the case of reflection, the light speed is always the same, so times are proportional to distances. In the case of refraction, however, the path of minimal time is different from that of minimal distance.

An ideal mirror reflects all the light that hits it, and each ray hitting a point on the surface is reflected in one direction, such that the angle of reflection equals that of incidence. If a viewer happens to be in that direction, looking at the point, he will see a reflection of the light source at the point (in the color of the light source, not that of the surface). Such an ideal reflection is called *specular*. Determining the specular reflection from a point requires the knowledge of the normal to the surface at the point, and the position of the viewer.

An ideal dull surface reflects each ray of light in all directions, because every point on the surface has many microfacets pointing in different directions. A viewer always sees the same intensity reflected from a given point, regardless of his position. He still sees different reflections from different points, since some points may be farther away from the light source, or may be pointing away from it. This type of reflection is called *diffuse* (Figure 17.5). The banana and apple in Figure 17.5 are examples of diffuse and specular surfaces, respectively, while the orange features both types of reflection.

> It depends on what you call normal.
> —Keanu Reeves (as Scott Favor) in *My Own Private Idaho* (1991).

Figure 17.5a illustrates strong diffuse reflection, where the angle θ between the direction L of the light source and the normal is small. The blue circular arc indicates equal (and strong) reflection in all directions. Part (c) of the figure shows weak reflection as a result of a large angle between L and the normal. The blue arc is still circular but is small. Figure 17.5b demonstrates specular reflection. The angle between the direction L

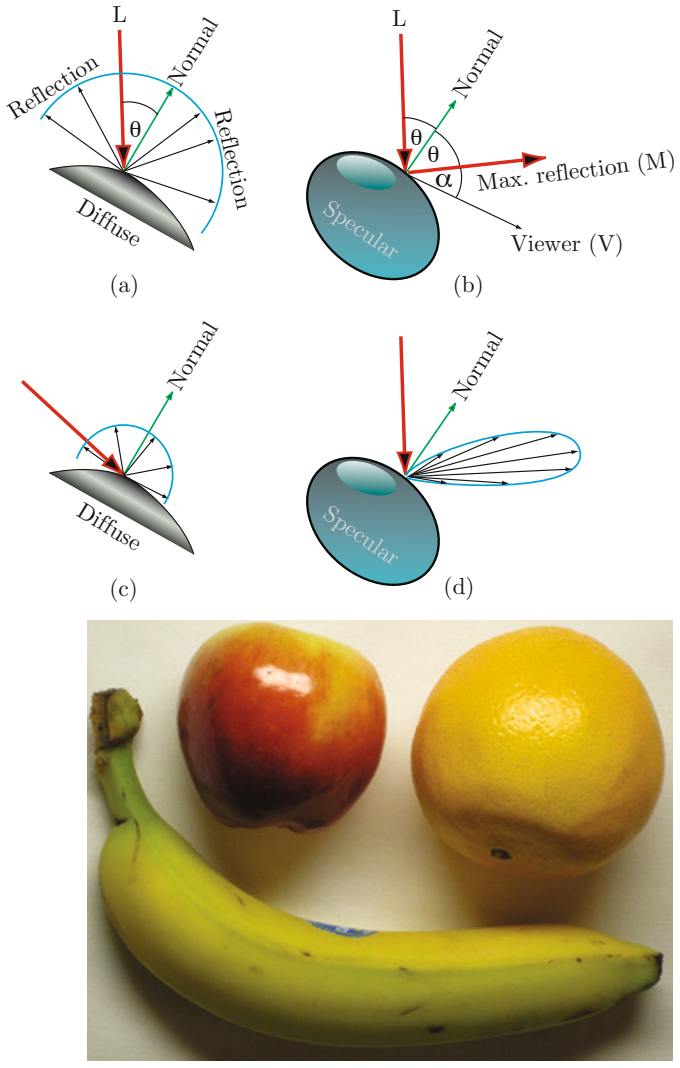


Figure 17.5: Diffuse and Specular Reflection.

of the light source and the normal is denoted by θ. The direction of maximum reflection forms the same angle θ on the other side of the normal. A viewer at V will see the reflection drop as α gets bigger. Part (d) of the figure shows an elongated blue arc that

indicates the direction of maximum reflection and how the reflection intensity drops off quickly as the viewer moves away from this direction.

17.2.4 A Reflection Model

We now realize that every point on a surface emits three types of light: diffuse and specular reflected light, and transmitted (refracted) light. Each light ray leaving the surface is a sum of these three contributions. We are now ready to discuss a simple shading model that simulates only reflection (no refraction) but produces acceptable results.

Diffuse reflection. The intensity of diffuse reflection from a point depends on the intensity I_p and direction $\mathbf{L}$ of the light source, on the direction $\mathbf{N}$ of the normal to the surface at the point, and on the coefficient of diffuse reflection k_d (a user-selected parameter between 0 and 1). According to Lambert's law, the intensity is $I_p k_d \cos\theta$.

Figure 17.6 illustrates this law. In Figure 17.6a, a wide light beam hits a surface while traveling parallel to the normal. In Figure 17.6b, the same beam hits the surface at an angle θ to the normal. Elementary trigonometry shows that the surface area covered by the beam is now greater by a factor of $1/\cos\theta$. Since the same amount of light is now spread over a larger area, the reflection is weaker, by the same factor.

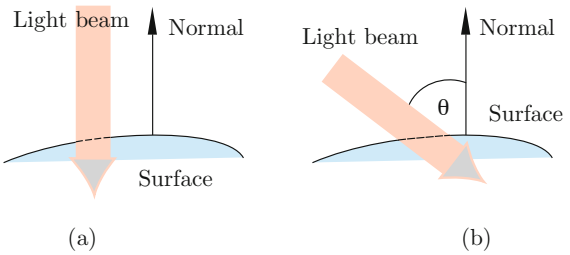


Figure 17.6: Diffuse Reflection at an Angle.

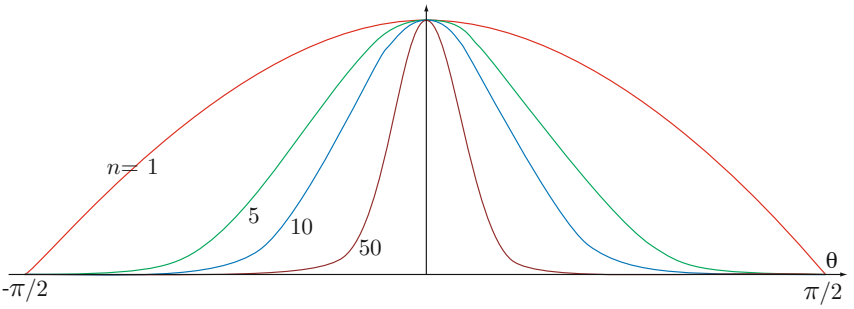
If $\mathbf{L}$ and $\mathbf{N}$ are unit vectors, then $\cos\theta = \mathbf{L} \bullet \mathbf{N}$. To simplify the calculation, we can sometimes assume that the light source is at infinity, i.e., all the light rays arriving at the surface are parallel, so $\mathbf{L}$ is the same for all points on the surface.

The intensity that reaches the viewer depends on his distance R from the point, so our model of diffuse reflection should be modified to give $I_p k_d(\mathbf{L} \bullet \mathbf{N})/R^2$ as the intensity of light reaching the viewer from a given point. Notice that R varies as the viewer moves around the surface from point to point.

Specular reflection. Looking at a shiny surface, we may see a highlight at a certain point. The reflection at that point is strong and it has the color of the light source, instead of that of the surface. The highlight also has the shape of the light source. As we move around the surface, the highlight moves with us on the surface. This is specular reflection.

The simple specular reflection model assumes $\mathbf{L}$ and $\mathbf{N}$ as before, and a viewer at $\mathbf{V}$. The reflection is mostly in direction $\mathbf{M}$ (Figure 17.5d), and the intensity seen by the

viewer depends on the material and on the angle α (Figure 17.5b,d). The smaller α, the stronger the intensity seen by the viewer from this particular point on the surface. The intensity is therefore proportional to $\cos\alpha$. In order to include the properties of the material, we use the term $\cos^n\alpha$, where n is an integer that depends on the material. For a perfect mirror, $n = \infty$ implying $\cos\alpha = 0$ and specular reflection that is strictly in the $\mathbf{M}$ direction. For a rougher surface, n is normally in the range 1–10.




```
g1=Plot[{Red,Cos[t]},{t,-Pi/2,Pi/2}];
g2=Plot[Cos[t]^5,{t,-Pi/2,Pi/2}];
g3=Plot[Cos[t]^10,{t,-Pi/2,Pi/2}];
g4=Plot[Cos[t]^50,{t,-Pi/2,Pi/2}, PlotRange->All];
Show[g1,g2,g3,g4,PlotRange->All]
```

Figure 17.7: The Behavior of $\cos^n\theta$.

Figure 17.7 illustrates the behavior of $\cos^n\theta$ for $n = 1, 5, 10, 50$. It is clear that for large values of n, the function is almost always zero.

The intensity of specular reflection that reaches the viewer is (assuming that $\mathbf{M}$ and $\mathbf{V}$ are unit vectors) $I_p k_s \cos^n\alpha/R^2 = I_p k_s(\mathbf{M}\bullet\mathbf{V})^n/R^2$. The quantity k_s is the coefficient of specular reflection, a user-controlled parameter in the interval $[0,1]$ that can be varied to simulate various materials.

Transparent objects transmit light, but also reflect some of it. We know from everyday experience that, looking through a sheet of glass, the angle between the line of sight and the glass surface determines how clearly we see through the glass. When a ray of light strikes the glass at a 90° angle, almost all of it is transmitted and refracted; very little is absorbed or reflected. The opposite is true when the ray hits the glass at a grazing angle. Such a ray is mostly reflected. Thus, a transparent object reflects light in a special way. The amount of reflection depends in a complex way on the angle of incidence and also on the wavelength. We say that such a surface has a coefficient of specular reflection that's a function of both the angle and the wavelength.

In practice, it is slow to compute vector $\mathbf{M}$ because it has to point in a certain direction, and also be in the same plane as $\mathbf{L}$ and $\mathbf{N}$, so the dot product $\mathbf{M}\bullet\mathbf{V}$ should preferably be replaced by a simpler expression that employs just the vectors $\mathbf{L}$, $\mathbf{N}$, and

$\mathbf{V}$. We note that

$$\begin{aligned}\mathbf{V} \bullet \mathbf{L} &= \cos(2\theta + \alpha) \\ &= \cos(2\theta)\cos(\alpha) - \sin(2\theta)\sin\alpha \\ &= \cos\alpha[\cos^2(\theta) - \sin^2\theta] - 2\sin\theta\cos\theta\sin\alpha,\end{aligned}$$

and

$$\begin{aligned}\mathbf{N} \bullet \mathbf{V} &= \cos(\theta + \alpha) \\ &= \cos\theta\cos\alpha - \sin\theta\sin\alpha.\end{aligned}$$

Combining these expressions yields $\mathbf{M} \bullet \mathbf{V} = 2(\mathbf{N} \bullet \mathbf{L})(\mathbf{N} \bullet \mathbf{V}) - \mathbf{V} \bullet \mathbf{L}$. The intensity of specular reflection can now be expressed as

$$\frac{I_p k_s [2(\mathbf{N} \bullet \mathbf{L})(\mathbf{N} \bullet \mathbf{V}) - \mathbf{V} \bullet \mathbf{L}]^n}{R^2}.$$

The computation of specular reflection is more intensive than in the case of diffuse reflection, because it involves determining the vectors $\mathbf{N}$ and $\mathbf{V}$ for every pixel (if the light source cannot be assumed to be at infinity, then $\mathbf{L}$ also has to be recomputed for each pixel).

Example: Figure 17.8 shows a surface with a normal in the y direction, $\mathbf{N} = (0, 1, 0)$, a light source at 45° in the xy plane $\mathbf{L} = (-0.7071, 0.7071, 0)$, and a viewer at 30° from the x axis in the same plane, $\mathbf{V} = (0.866, 0.5, 0)$. We get $\mathbf{N} \bullet \mathbf{L} = 0.7071$, $\mathbf{N} \bullet \mathbf{V} = 0.5$, and $\mathbf{V} \bullet \mathbf{L} = -0.2588$, which yields $2(\mathbf{N} \bullet \mathbf{L})(\mathbf{N} \bullet \mathbf{V}) - \mathbf{V} \bullet \mathbf{L} = 2 \times 0.7071 \times 0.5 + 0.2588 = 0.966$. A relatively high reflection (96.6% of the maximum value), because the viewer is only 15° away from the direction of maximum reflection. This large value is normally reduced when the effects of $I_p k_s$ and R^2 are included. The effect of n can now easily be illustrated.

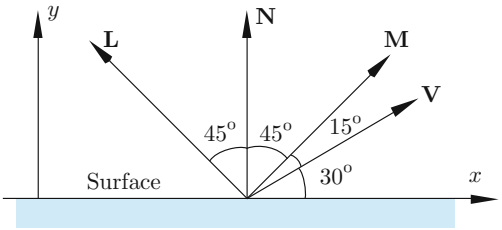


Figure 17.8: Specular Reflection Example.

For $n = 10$, the value above goes down to 70.7%, and for $n = 100$ (a very shiny surface), it drops all the way down to 3.14%! With a shiny surface, even an offset of 15° is enough to reduce the reflection highlight to almost nothing.

Ambient Reflection. In most cases, we can account for multiple reflections from nearby objects with the simple model $I_a k_a$, where I_a is the intensity of *ambient reflection* and k_a is the coefficient of this reflection (a parameter that depends on the material).

In summary, our simple shading model thus assigns to each pixel an intensity I of reflected light reaching the viewer of

$$I = I_a k_a + \frac{I_p[k_d(\mathbf{L} \bullet \mathbf{N}) + k_s(\mathbf{M} \bullet \mathbf{V})^n]}{R^2}.$$

Color Shading. The reflection coefficients k_d and k_a depend on the color of the incident light. A shading model for a color output device should therefore compute three intensities I_R, I_G, and I_B for each pixel, and use them to determine the color of the pixel. For example, $I_G = I_{aG}k_{aG} + I_{pG}[k_{dG}(\mathbf{L} \bullet \mathbf{N}) + k_s(\mathbf{M} \bullet \mathbf{V})^n]/R^2$.

More sophisticated rendering models can compute shadows and take into account multiple reflections, transparent objects, and shadows (Section 17.5). They can also deal with complex, nonsolid objects such as waves, smoke, and clouds. Because of these models, rendering is considered a computationally intensive application.

17.3 Gouraud and Phong Shading

A polygonal surface is especially easy to shade, because we can assume that all the pixels of a polygon reflect the same amount of light. The particular shading model being used should therefore be applied just once to each polygon. The resulting surface, however, looks angular and unnatural. Fortunately, there are two simple methods that result in a better looking surface by smoothing out the shading. These are the Gouraud and Phong shading algorithms. The former interpolates intensities and is discussed below. The latter interpolates normal vectors and its implementation details are similar.

Gouraud's method [Gouraud 71] smooths out the shading of a polygonal surface by computing reflection intensities at the corner points of each polygon and interpolating these intensities at every pixel on the polygon. It proceeds in four steps as follows:

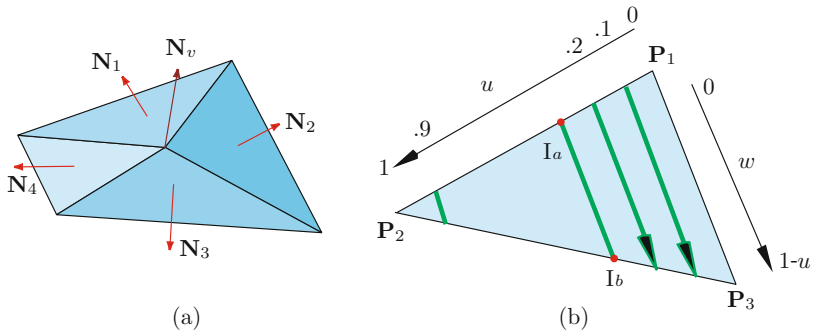


Figure 17.9: Gouraud Shading.

1. The normal vectors $\mathbf{N}_i$ are determined for all polygons i (Figure 17.9a).

2. Vertex normals $\mathbf{N}_v$ are computed for each vertex v by averaging the surface normals of all the polygons sharing the vertex. Figure 17.9a shows one such normal, calculated at the intersection of four triangles. Its value is $\mathbf{N}_v = (\mathbf{N}_1+\mathbf{N}_2+\mathbf{N}_3+\mathbf{N}_4)/4$.

3. Vertex intensities I_v are computed for all the vertices of the surface by using the normal vectors $\mathbf{N}_v$ of step 2 and any desired shading model.

4. Each polygon is shaded, scan line by scan line, by interpolating the reflection intensities at its vertices. If the polygon is a triangle, the scanning is done using Equation (9.5), which is duplicated here:

$$\mathbf{P}_1 + u(\mathbf{P}_2 - \mathbf{P}_1) + w(\mathbf{P}_3 - \mathbf{P}_1) = \mathbf{P}_1(1 - u - w) + \mathbf{P}_2 u + \mathbf{P}_3 w. \tag{9.5}$$

For each scan line, two intensities, I_a and I_b, are interpolated from the vertex intensities I_1, I_2, and I_3 (Figure 17.9b) and are then used to interpolate an intensity I_p for every pixel on the line. Figure 17.10a shows a procedure for scanning a triangle. Figure 17.10b shows the start and end points of each scan line.

```
procedure Gouraud(P1,P2,P3,I1,I2,I3);
real I; point P;
for u:=0 to 1 step 0.1 do
 for w:=0 to 1-u step 0.001 do
  I:=I1*(1-u-w)+I2*u+I3*w;
  P:=P1*(1-u-w)+P2*u+P3*w;
  Pixel(P,I);
end;
```

(a)

Value of u	Range of w	Scan Line from	Scan Line to
0	$0-1$	$\mathbf{P}_1$	$\mathbf{P}_3$
.1	$0-.9$	$.9\mathbf{P}_1 + .1\mathbf{P}_2$	$.1\mathbf{P}_2 + .9\mathbf{P}_3$
.2	$0-.8$	$.8\mathbf{P}_1 + .2\mathbf{P}_2$	$.2\mathbf{P}_2 + .8\mathbf{P}_3$
⋮			⋮
.9	$0-.1$	$.1\mathbf{P}_1 + .9\mathbf{P}_2$	$.9\mathbf{P}_2 + .1\mathbf{P}_3$
1	$0-0$	$\mathbf{P}_2$	$\mathbf{P}_2$

(b)

Figure 17.10: Scanning a Triangle.

◇ **Exercise 17.3:** Modify the above triangle scanning procedure to handle a four-sided polygon.

Gouraud's shading is a simple method that often produces good results. Its main drawback is the case where there should be a small shiny reflection (a highlight) inside

a polygon. This algorithm computes only the reflection intensities at the corners, so it never finds out about the highlight.

Phong shading is similar, but it overcomes this shortcoming by interpolating the normal vectors rather than the intensities. It ends up with an interpolated normal $\mathbf{N}_p$ at every pixel p and employs any shading method to calculate the reflection intensity from the point. This enhances mostly specular reflection.

17.4 Palette Optimization

Color lookup tables are discussed on Page 34. In this section, we assume a lookup table of 256 entries. Before displaying an image, the table has to be loaded with a palette of 256 colors, and *only* these colors can later be displayed.

Rendering consists of generating the surfaces and then displaying them. Displaying a surface is also a two-part task. First, a shading algorithm is executed to determine the ideal color of each pixel, and then another algorithm is needed, to pick up the best palette color, the one that's nearest the ideal color. It is therefore crucial to load the lookup table with the right 256 colors, the ones that best "represent" the image to be displayed. This is the problem of *palette optimization*.

A simple solution is to compute the distribution of colors (count the number of times each color occurs in the original image) and to load into the lookup table the 256 most common colors of the image. This simple solution has a serious drawback, outliers! A color that occurs in just a few pixels in the original image may be crucial to our understanding (or our enjoyment) of the image. If such a color (like the green in the figure) does not appear in the lookup table, the final, displayed image would look very different from the original. A better method should load the lookup table with the most common colors of the image, but also with a representative (or several representatives) of every other color that happens to appear in the image.

A good example is an image of a beach scene. The dominant colors are blue (water and sky), yellow (sand), and white (clouds). However, there may be a person in the image, wearing a green swimsuit. Clearly, the lookup table needs to have at least one shade of green in it, even though it is going to be used in only a few pixels.

> Examining an image, one would be at a loss to know whether it owed its shape to the original source of light or to the details of the intervening gravitational field. The only difference between the appearance of the surface of the lake and that of the night sky is that the former depends on reflection and the latter on refraction.
>
> —Hans Christian von Baeyer, *The Fermi Solution*, 1993.

A good, although not fast, palette optimization method is *median-cut color quantization*. It starts with the RGB cube (Section 21.6.1) where each axis is labeled from 0 to 255, and it ends up cutting the cube into 256 rectangular blocks, each containing about the same number of picture colors. In the beach example, there will be many blocks in the blue, yellow, and white regions of the RGB cube, but there will be at least

one block in the green region. The last step is to select the color in the middle of each block and to load the lookup table with those 256 colors. Here are the steps:

1. Determine the extreme values of red in the picture colors. If no color in the picture has red below, say, 18 and above 240, then those parts of the RGB cube corresponding to red below 18 and above 240 are ignored (we can imagine them being cut off the cube and thrown away, since the picture has no colors in those parts). The same thing is done for the green and blue dimensions of the RGB cube.

2. The longest side of the remaining cube is now determined. Let's say that it is the red side. All the colors in the picture are sorted by their red values and the median color is picked up. The median is that shade of red that has equal numbers of shades of red above and below it. If it is, for example, $(56, x, y)$, then there are equal numbers of colors in the picture with red < 56 and with red > 56. The RGB cube is now cut at red $= 56$, producing two rectangular blocks.

3. The above process is now applied to the two blocks created in step 2. Each is cut at a median, which produces four blocks.

4. The process of cutting blocks is repeated four more times, for a total of eight times, producing $2^8 = 256$ blocks. Because the cuts are always done at the median, each of the final 256 blocks has the same number of picture colors.

5. The center of each of the blocks is calculated, using the corner coordinates of the block, and is added to the color lookup table. An even better result is obtained by averaging, in each block, all the picture colors included in the block, but this is even more time-consuming.

17.5 Ray Tracing

The shading methods described so far share an important defect. They shade each object in the scene separately. If the objects are dull, this basic shading produces good results. In a scene with shiny objects, however, each object may be reflected in other objects, so shading objects separately may result in an image that looks wrong, artificial, and unreal. The ray tracing method discussed here is based on a completely different approach to shading and is especially successful in rendering complex scenes that consist of many colored, shiny, and transparent objects (see Plates A.2, G.4, H.3, and K.2).

The term "ray tracing" refers to any method that approaches a problem by computing the paths of rays or particles. In addition to its use in computer graphics, ray tracing is also employed in physics, mostly to analyze the behavior of optical devices. Notice that we can imagine a light ray as either a thin, mathematical line, or as a stream of photons traveling together along the same straight path until they are absorbed, reflected, or refracted (slowed down in dense material).

When a light ray (or equivalently, a beam of photons) hits the surface of an object, it may be (fully or partly) absorbed, reflected, or refracted. Some surfaces absorb part of the light and in response emit photons of longer wavelengths and in different directions. This phenomenon, called fluorescence, is rare and is generally disregarded by rendering software. Notice that conservation of energy applies to the energy of photons as well. If 40% of the light is absorbed by a surface and 60% is reflected by it, then nothing is refracted. Ray tracing can handle absorption, reflection, and refraction.

History. The history of ray tracing starts in 1968, with ray casting, This was a simple rendering method where rays are traced from the eye (more accurately, from an eyepoint), through the pixels of the final image, all the way back to the first surface they hit. Once the intersection point of a ray and a surface has been determined, the ray casting algorithm may employ any shading method to determine the color and brightness of the pixel through which the ray was sent.

In 1979, Turner Whitted extended ray casting to ray tracing. In hindsight, this was the obvious step, but ray tracing is computationally intensive and computer hardware in the late 1970s was much slower than today. Nevertheless, once the first step was taken, researchers and programmers immediately realized the advantages of the ray tracing approach (the simulation of refraction, multiple reflections, and shadows) and tried to implement it in full or in part.

> Teraflop club: /te'r*-flop kluhb/ [FLOP = Floating Point Operation] n. A mythical association of people who consume outrageous amounts of computer time in order to produce a few simple pictures of glass balls with intricate ray-tracing techniques. Caltech professor James Kajiya is said to have been the founder.
>
> —Eric Raymond, *The Hacker's Dictionary*.

A good, detailed reference for ray tracing is [Glassner 89], but see also chapter 8 of [Hill 06] for a simplified, two-dimensional example. Many other references are available in the standard computer graphics literature.

Ray tracing in computer graphics is based on the fact that whatever method we use for rendering a scene, we end up watching it as a rectangular set of pixels. Most often, an image is viewed on a screen where each pixel sends light of a certain color and intensity to the eye. Thus, the problem of rendering can be stated as follows: Determine the color and intensity of each pixel on the screen. In real life, light starts from a light source, it hits an object and is partly reflected. It may hit other objects and be reflected by them, and it may eventually, after bouncing around several times, emerge from a pixel on the screen, enter our eye, and be perceived by us as a small dot. The principle of ray tracing is to reverse this process. Instead of a light ray proceeding from its source to our eye, it is traced from our eye, through a pixel on the screen, to the objects that reflected it, and eventually to its source. The tracing makes it possible to determine what color and intensity reach our eye from any pixel. Thus, ray tracing simulates the paths of many rays of light backward, from their target to their source.

Figure 17.11 shows a simple scene with a few objects and four rays traced. Ray a is traced to the dodecahedron on the coffee table and from there to the ceiling, where it is traced back to the top of the light fixture. It is clear that after two reflections, the light has lost much of its intensity. so the pixel through which ray a emerges should appear dark. Ray b is traced directly to the light source, so the corresponding pixel should be painted bright. Both rays c and d emerge through their corresponding pixels and strike the camera after one reflection, so these pixels should be assigned shades between those of a and b, depending on how much light the wall and the dodecahedron reflect. The precise colors of the pixels depend on the amounts of diffuse and specular reflections

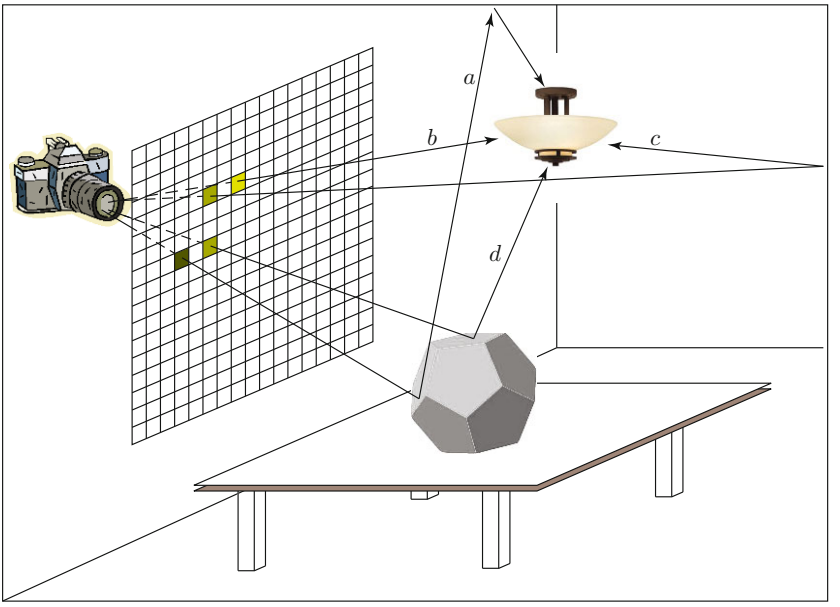


Figure 17.11: Ray Tracing: The Principle.

of the various objects. Recall that specular reflection involves highlights whose color is that of the light source regardless of the color of the object.

◇ **Exercise 17.4:** Tracing rays of light backward seems counterintuitive. Why not start at the light source and trace each ray of light forward until it enters the eye through some pixel on the screen?

Even this simple example illustrates the main drawback of ray tracing. This approach to rendering requires the determination of a vast number of intersection points. The computational cost of ray tracing is therefore much higher than that of shading objects separately, one by one (scanline rendering). As rays are traced back deeper and deeper into the scene, they may intersect more and more objects. Current (2010) display monitors may consist of millions of pixels (a typical 1920×1200 display resolution corresponds to more than 2.3 million pixels), and each ray traced back through a pixel may be reflected several (even many) times before it ends up at a light source.

◇ **Exercise 17.5:** If fully traced, will every ray end up at a light source?

The tracing process may be speeded up considerably when we realize that most objects are not shiny and reflect only a small percentage of the light. Thus, with each reflection, the intensity of the ray that finally emerges from a pixel is greatly reduced. It therefore makes sense to stop the tracing when it becomes clear that the intensity of the final ray is below a certain threshold, and simply paint the corresponding pixel

black. Another shortcut is provided by the natural tendency of a light beam to diverge. We know that the intensity of a light beam falls off as the square of the distance. If the intensity of a ray falls to 1/2 of its original after traveling a distance d, then after traveling $3d$ units it drops to $(1/2)^3 = 1/8$ of its original. A practical ray tracing algorithm should therefore stop the tracing when the total distance traveled by a ray is greater than another user-controlled threshold parameter.

Ray tracing is therefore expensive, but its chief advantage is a high degree of realism, and after all, realism is the supreme goal of computer graphics (but see the beginning of this chapter for non-photorealistic rendering). The ray-tracing approach to rendering can simulate the important optical effects of reflection, refraction, scattering, chromatic aberration, and shadow casting. It often results in an image that is difficult or even impossible to distinguish from a photograph of a real scene.

In addition to realism, ray tracing can simulate the effects and constraints of a real, physical camera on an image. Among the important camera effects are limited depth of field (Section 26.4.7) and the shape of the aperture, which is typically a pentagon or a hexagon.

As if these advantages were not enough, ray tracing also determines those surfaces that are visible to the eye at its current location. Imagine a ray of light traced back from the eye through a pixel p. If this ray intersects surface s at point t, then no other points on s or on any other surface are visible to the eye at pixel p. Once rays have been traced from the eye to all the pixels on the display screen, only those surfaces that are visible to the eye will appear on the screen. Thus, the additional work required by ray tracing is somewhat alleviated by not having to have an extra step for visible surface determination.

The preceding discussion should convince the reader that ray tracing can also handle multiple reflections. Images with multiple reflections are not common, but they can be striking, colorful, complex, and beautiful. Try the following experiment. Take a small mirror and stand in front of a large mirror while holding the small mirror in front of you, facing the large mirror. Figure 17.12 and Plate G.4 illustrate the series of infinite reflections you see. This effect is simulated in ray tracing, except that the simulation (like everything else in real life) must be finite. It stops when either the number of reflections or the total distance traveled by the light exceeds a threshold parameter.

> Sophie Sheekhy stood in front of her mirror in her white shift. She stared at herself, and herself stared back at herself. The mirror on the pine chest reflected the cheval glass by the door so that she stood behind and behind herself, on a series of thresholds going white-green into diminishing infinity.
>
> —A. S. Byatt, *Angels and Insects* (1986).

Once the general approach of ray tracing is grasped, it is easy to understand the main steps of a ray tracing algorithm. Each ray sent from the eye through a pixel in the display screen must be traced. The algorithm must find the nearest object (if any) that it intersects and the precise intersection point. The simplest way of determining this is to test the ray for intersection with all the objects in the scene. With some thinking, however, we can easily improve this step. When the software generates an object, it determines the bounding box of the object (the smallest rectangular box into which the object fits). The ray tracing algorithm can use the bounding boxes to quickly eliminate

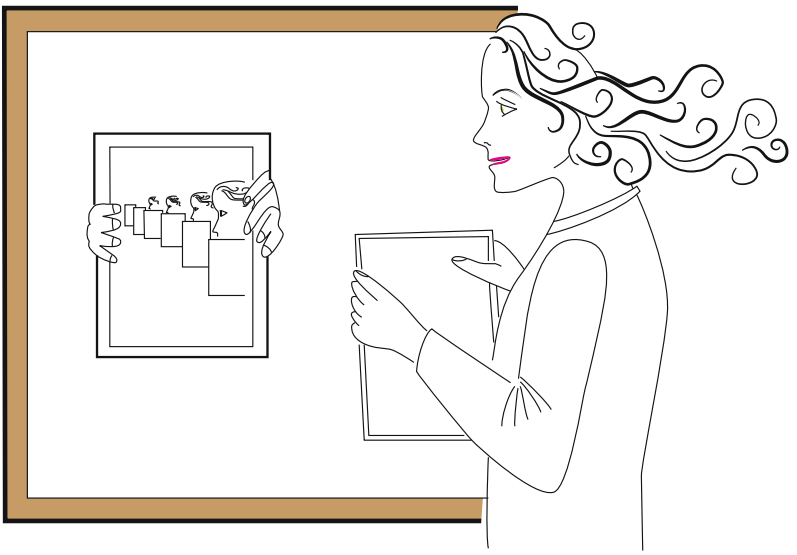


Figure 17.12: Ray Tracing: Infinite Reflections.

many objects from the tests. For example, if a ray enters the scene at a z coordinate of z_0 and it goes "up" in the scene (i.e., its z coordinate increases as the ray propagates), then the ray cannot intersect any object the z coordinate of whose bounding box is less than z_0.

It is also possible to have a hierarchy of bounding boxes. If the object is a chair, then each of its parts can have a bounding box, and the bounding box for the entire chair is constructed by selecting the extreme coordinates of the individual bounding boxes. The ray tracing algorithm tests the current ray against the general bounding box. If there is no match, the ray does not intersect the chair. If there is a match, there may be an intersection, and the algorithm tests the ray against each part's bounding box to find a possible match.

Another way to speed up the tracing of rays is parallel execution. The tracing of a ray determines the color of a pixel of the image. Each ray is therefore independent of all the other rays, which makes ray tracing a natural candidate for parallel execution. Imagine a parallel computer with N processors sharing a central memory. Each processor may be assigned an area of the image, and they can work in parallel, tracing N rays in the time it previously took to trace one ray. In practice, this type of computer suffers from memory contention. Several processors may try to access memory while memory is busy serving another processor. A solution may be to construct a parallel computer where each processor has its own memory.

The discussion of curves and surfaces in Part III of this book makes it clear that even though an image may be smooth and curved in principle, once it is rendered in the

computer it becomes a grid of pixels with a limited resolution. This is an important fact that simplifies the task of implementing ray tracing.

In addition to multiple reflections, ray tracing can also handle refraction and transparent objects. When an object is created, the software has to construct a table with the bounding box and other attributes of the object. Among other attributes, this table contains reflection and transparency information. For example, the object may reflect 10% of the light (2% diffuse and 8% specular), absorb 30%, and transmit the remaining 60%. The refraction index may be 2.418 (that of diamond) and the surface color may be blue (more precisely, 475 nm, or RGB $(0, 0, 255)$, or HSB $(240, 100, 100)$).

When the ray-tracing algorithm determines that a traced ray intersects an object, it has to check the object's data table for transparency (or refraction). If the object is transparent, the ray is split, as illustrated in Figure 17.13, into two secondary rays, a reflection ray and a transmission (or transparency) ray. Ray a in the figure is traced back to the prism, so part of it (b) is reflected and part (c) is refracted (some of the beam's energy may be absorbed). Similarly, ray d intersects the rectangular box, and the algorithm must create, and recursively trace, secondary rays e and f. It is now obvious that a ray tracing program must be recursive and may have to complete a huge amount of work, tracing many thousands of both primary and secondary rays.

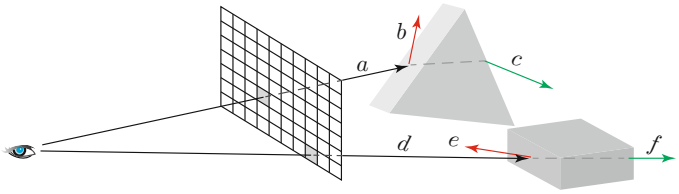


Figure 17.13: Ray Tracing: Refraction.

Shadows. In the presence of light, opaque objects cast shadows (see Plates A.2, G.1, G.2, I.7, L.1, and O.2), and this optical phenomenon can also be simulated in ray tracing.

Figure 17.14 illustrates how this is done. A ray a is traced into the scene. It hits a surface (the floor) and is reflected (b) as usual, in order to locate its next intersection. However, another *shadow* ray (c) is also created and is sent toward the light source (there is only one light source in the figure, but in principle there may be any number). This ray intersects the opaque rectangular box on its way to the source, which means that point p is in shadow. The pixel through which ray a passes should therefore be dark.

If there are several light sources, then a shadow ray has to be sent to each. Even more, if a light source has a finite extent (if it is more than a mathematical point), then several rays should be sent to it.

Creating and tracing shadow rays adds extra work, but also helps to reduce the total amount of work of the raytracer, because once it is known that p is in shadow, there is no need to create and trace ray b.

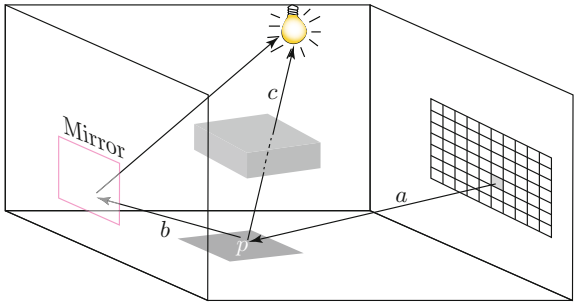


Figure 17.14: Ray Tracing: Shadows.

In scenes with shiny surfaces, the statement above may not be true. It may happen that ray b will intersect a very bright object, perhaps a mirror that reflects the light source directly to point p. In such a case, p will not be in complete shadow. In general, the knowledge that p is in direct shadow helps the ray tracing algorithm to select the correct color and brightness for p. See also the discussion of diffuse interreflection in Section 17.6.

(Here is a typical example of a point that is in shadow, yet is brightly illuminated. A room with one small window open. It is sunny outside, but the room is dark. A child is standing outside and is reflecting the sun into the room with a small mirror. Even though the room is generally dark, there is now a bright patch of light on one wall, and this increases the ambient light in the entire room.)

After reading the material in this section, it is useful to look at a list of the main steps of a ray tracing algorithm.

for all pixels p do
begin

1. Construct a ray $R(p)$ from the position of the eye through p.
2. Determine the intersections of $R(p)$ with all the objects in the scene.
3. Select the intersection point (if any) that is nearest the eye position.
4. Compute the color of pixel p. This is done recursively by constructing a reflection ray, a transmission (or transparency) ray, and shadow rays (one per light source) and following these rays recursively until each ends at a light source or bounces too many times or propagates too long.
5. Set p to the color determined in step 4.

end.

Clearly, most of the work is done in step 4, where one ray is followed and processed and new rays are created and pushed into the recursion stack to be processed later.

As if this algorithm is not complex enough, there are additional problems and points to consider. Perhaps the most important drawback of the "basic" ray tracing method is lack of sampling. Figure 17.15(a) illustrates this problem. The small blue and green objects are smaller than a pixel and are completely lost when rays are sent from the eye.

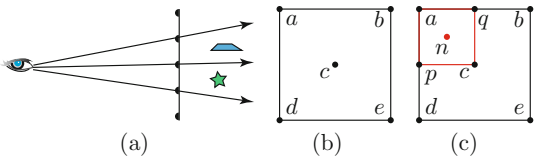


Figure 17.15: Ray Tracing: Small Objects and Adaptive Supersampling.

If an object in the scene is smaller than a pixel, the object cannot be seen, but in a realistic rendering it should not disappear completely. Instead, its color should affect the color of the pixel of which the object is part. This problem was immediately noticed by workers in the ray tracing area and several solutions have been proposed as follows:

- Send several thin rays through each pixel, trace all the rays to determine the colors they return, compute the average color, and assign it to the pixel. This color will reflect the fact that a small, invisible object exists at that pixel. If we decide to divide each pixel into $k \times k$ subpixels, then this approach increases the amount of work by a factor of k^2, a price that is justified only in those cases where the small objects are important. This approach is referred to as supersampling.

- Adaptive supersampling is more complex to implement, but is much faster. The idea is to identify those pixels that seem to cover small objects and concentrate on them. Figure 17.15(b) shows five rays traced through the four corners and the center of a square pixel. The rays are fully traced and the colors of the five points are saved. The four pairs of colors (a, c), (b, c), (d, c), and (e, c) are prepared and compared. If all pairs are equal (or if their differences are below a certain threshold parameter), then the algorithm assumes that there are no small objects behind the pixel and the pixel is painted color c. If two pairs, such as (b, c) and (a, c), are similar and the other two are different, as in Figure 17.15(c), then the algorithm assumes that there is a small object behind the red quadrant $aqpc$. In this case, the algorithm traces rays p, q, and n, constructs four color pairs, and performs similar color comparisons. This process can be repeated several times for smaller and smaller quadrants.

- Stochastic sampling. Instead of tracing rays in a regular pattern in each pixel, stochastic sampling creates rays that are distributed within the pixel in a semi-random pattern called a Poisson disk distribution or blue noise (see insert below). The process starts by generating a large number of random dots in the pixel, and then eliminating many of them such that the remaining dots are always separated by a certain minimum distance. This method is computationally expensive but is claimed to produce excellent results.

The term stochastic (from the Greek $\sigma\tau\acute{o}\chi o\varsigma$, meaning aim or guess) means random. A stochastic process is non-deterministic; the next state of the process is determined not just by deterministic rules, but also by some random element.

Poisson disk distribution

Section 21.3 discusses the structure of the human eye and its photosensitive cells, the rods and cones. Here, we concentrate on the spatial distribution of the cones (see also [Deering 05]). These cells are located in the retina and most of them are found in a sensitive part of the retina called the fovea (or yellow spot). The fovea is a small ($< 1\,\mathrm{mm}^2$) but very special region of the retina, where the cell concentration is highest and the sampling of light is maximal. There are about 200,000 cones per square mm in the fovea of an adult eye, to provide maximum image resolution and color sensitivity. In order to cover the space most efficiently, the cones are arranged in a hexagonal (honeycomb) pattern. (To visualize this pattern, hold a stack of round toothpicks in your hand and look at their tips.)

Outside the fovea, both the rods and cones are distributed less tightly. In the early 1980s, it became known that the distribution of cones outside the fovea is mostly random, but obeys a simple rule, individual cones are never closer than a certain distance. This distribution is often referred to as a Poisson disk distribution, but is also known as blue noise.

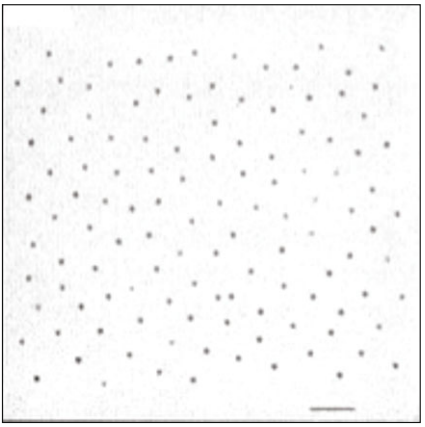


Figure 17.16: Poisson Disk Distribution of Points.

Caustics. Ray tracing is based on backward tracing of rays, from the eye to the light source. Is it computationally expensive, but is feasible. It is also possible to consider forward ray tracing, from the light source, through reflections, either to the eye of the observer or in another direction, where it is ignored by the algorithm. Clearly, the number of light rays is too great to completely simulate, but this approach can simulate the phenomenon of caustic better than ray tracing.

A caustic (in optics) is the envelope of light rays reflected or refracted by a curved surface or object. Such an envelope can be seen when it is projected on a surface.

Geometrically, a caustic is a curve or surface to which each of the light rays is tangent. The rainbow is a familiar caustic. A common example is shown in the figure, where the parallel light rays from the sun project the (wide) surface of the water onto a narrow pattern on the bottom of the glass and on the bright, diffuse surface. Often, light shining through ripples causes caustics in a shallow body of water.

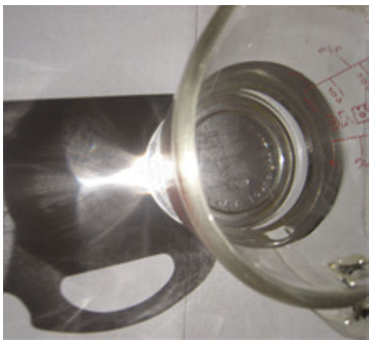


Figure 17.17: Caustic Cast by Water in a Glass.

17.6 Photon Mapping

Photon mapping is a two-pass rendering technique that combines the advantages of forward and backward ray tracings. In the first pass, photons are traced forward from the light source and are used to generate and collect illumination information which is stored in a data structure (the photon map). The second pass computes the actual rendering using data from the photon map. Rays are traced backward from the eye to objects in the scene to determine the surface points visible to the eye, and the photon map is then used to estimate the illumination at each visible point. Reference [Jensen 05] is by the developer of this method.

Photon mapping produces images that are especially striking and realistic in the following cases:

- Caustics. Those concentrated patterns of light reflected from and refracted by thick layers of glass or by the surface of water.

- Diffuse interreflection. This is a phenomenon where light reflected by a diffuse surface is reflected again by another surface such as the ground, a wall, or furniture. In this way, light reaches areas that are supposed to be in a shadow (i.e., points from which a light source is not visible). If the original diffuse surface is colored, the reflected light inherits that color, which is in turn given to the surrounding objects.

- Subsurface scattering. This phenomenon occurs where light hits the surface of an object and is scattered in various directions before being absorbed or reflected. Scattering happens when the light is reflected at various angles inside the object, perhaps as a result of the object having layers of varying refraction indexes. The light eventually emerges outside the object, but not at the refraction angle. Materials such as marble, skin, and milk exhibit this kind of optical behavior.

In the first pass of photon mapping, photons are sent from the various light sources into the scene. When a photon hits a surface, its direction and the coordinates of the intersection points are stored in the photon map. The data table of the surface is then examined to decide on the future of the photon. The table contains percentages of absorption, reflection, and refraction. A random number is then produced and is used to determine the future of the photon. The photon may be absorbed (in which case it simply disappears) or it may be reflected or refracted (in these cases its new direction is determined and the photon is traced further).

The second pass (rendering). The photon map is now used to estimate the illumination (color and brightness) of the display pixels, as in traditional ray tracing. For each pixel, a ray is traced from the eye, through the pixel, until it intersects an object in the scene at a point $\mathbf{P}$ or until it passes through the scene without hitting anything.

The algorithm now computes the amount and color of light emitted at $\mathbf{P}$ as the sum of direct and indirect terms. The indirect illumination is estimated from the photon map. For the direct illumination, the algorithm sends rays from $\mathbf{P}$ to each light source. If such a ray does not hit any object on its way, then the color, intensity, and distance of the light source are used to compute the direct illumination at $\mathbf{P}$.

17.7 Texturing

Texturing is a method commonly used to add realism to an image (see Plates A.6, A.8, D.2, H.5, I.4, I.7, J.1, J.3, K.2, and N.1). The idea is to create a table with texture values (black and white, or colors) and map it onto the surface. The texture table is just a small bitmap, where each entry describes the color of a pixel. In practice, the table is an array $T[m, n]$ of values. Texturing is done in one of two ways:

1. A surface $\mathbf{P}(u, w)$ is given and has to be textured. The entire surface has to be scanned and each pixel should be assigned a value from the texture table. Normally, the surface is much bigger than the table. The table covers only a small part of the surface, and several copies of the table have to be used to texture the entire surface. The surface is scanned by varying u and w independently from 0 to 1, and a function is needed that maps each pair (u, w) to a pair of indexes (i, j) in the texture table (where $1 \leq i \leq m$ and $1 \leq j \leq n$).

2. Several surfaces are given and the entire scene is shaded by ray tracing. Instead of shading each surface independently, we follow light rays from the eye of the observer through the screen and into the scene. Such a ray may hit a surface $\mathbf{P}(u, w)$ at point $\mathbf{P} = (x, y, z)$. We have to find the pair (u, w) that corresponds to this point and map this pair to a pair of indexes (i, j) in the texture table.

Regardless of the method used, texturing is affected by the topology of the surface. The texture table can be considered a flat rectangle. If the surface is anything other

than a flat plane, then mapping the table to the surface may introduce distortions (in the same way that mapping the spherical Earth to a flat sheet of paper always involves distortions). This is why there is no single mapping that is good for all surfaces. Mapping functions have to be derived for common, regular surfaces, such as a sphere, a cone, a cylinder, or a torus. When given a general surface, the user should decide which of the known mapping functions to use, based on the shape of the surface given.

Example: Mapping a cylinder. This is a simple example because it is easy to wrap a rectangle on a cylinder without distortions. We start with the parametric equation of the cylinder (Equation (Ans.7)):

$$\mathbf{P}(u,w) = \big(a(2u-1), R\sin w, R\cos w\big),$$

where $0 \le u \le 1$ and $0 \le w \le 2\pi$. This describes a cylinder that's $2a$ pixels long, with a radius R, centered on the origin and pointing in the x direction. We assume a texture table $T(i,j)$, where $0 \le i \le m-1$ and $0 \le j \le n-1$.

The quantity $\texttt{round}(u(2a-1))$ has integer values that vary from 0 (when $u=0$) to $2a-1$ (when $u=1$). We therefore define $i = \texttt{round}(u(2a-1)) \bmod m$. This assigns index i values between 0 and $m-1$ and causes several copies of the texture table to be laid side by side along the cylinder.

For j, we similarly start with $\texttt{round}(w(2\pi R-1))$. This quantity gets integer values in the range 0 (when $w=0$) to $2\pi R-1$ (when $w=1$). The index j itself is now defined as $j = \texttt{round}(w(2\pi R-1)) \bmod n$.

The surface can now be displayed by a double loop, on u and on w, where for each point, we perform two steps:

1. The normal vector at the point is calculated and a shading model is used to calculate the reflection intensity from the point.

2. A pair of indexes (i,j) is calculated, and the value $T(i,j)$ is used to modify the reflected intensity of step 1.

When ray tracing is used, we start with a ray that intersects the cylinder at a point (x,y,z). We need to find the values of (u,w) that correspond to that point. We consider the single equation

$$\big(a(2u-1), R\sin w, R\cos w\big) = (x,y,z),$$

a system of three equations in the two unknowns u and w. The solutions are

$$u = \left(\frac{x}{a}+1\right)/2, \qquad w = \arcsin(y/R) = \arccos(z/R).$$

Once u and w are known, a pair of indexes (i,j) can be calculated as above.

Example: Texturing a Sphere. A sphere of radius R, centered at the origin is expressed by:

$$(R\cos u\cos w, R\cos u\sin w, R\sin u),$$

where $-\pi/2 \le u \le \pi/2$ and $0 \le w \le 2\pi$. A large circle on this sphere has length $2\pi R$, so a meridian (constant w) from the south to the north pole covers πR pixels. The quantity $u+\pi/2$ varies in the range $[0,\pi]$, so $(u+\pi/2)R$ varies in the range $[0,\pi R]$. We, therefore, define the first mapping index as $i = \texttt{round}((u+\pi/2)R) \bmod m$.

Varying w for a constant u takes us along a latitude of radius $R\cos u$. This radius varies from R at the equator (where $u = 0$) to zero at the poles (where $u = \pi/2$ or $-\pi/2$). The circumference of this circle is, therefore, $2\pi R\cos u$, and half the circumference is $\pi R\cos u$.

Since w varies in the range $[0, 2\pi]$, the quantity $(w/2)R\cos u$ varies in the range $[0, \pi R\cos u]$. The second mapping index, j, is therefore defined as

$$j = \texttt{round}((w/2)R\cos u) \bmod n.$$

The distortion can be observed by noticing that when $u = \pi/2$, the general expression of the sphere reduces to point $(0, 0, R)$, the north pole, regardless of w. The first texture index for this point is $i = \text{round}(\pi R) \bmod m$. This is a number in the range $[0, m-1]$. However, the second texture index for this point is $j = \text{round}((w/2)R\cos(\pi/2)) \bmod n = 0$.

When ray tracing is used, a ray may intersect the sphere at point (x, y, z). We need to calculate the corresponding values of (u, w). The equation

$$(R\cos u\cos w, R\cos u\sin w, R\sin u) = (x, y, z)$$

is used as a system of three equations with the two unknowns u and w. The solutions are $u = \arcsin(z/R)$ and $\tan w = y/x \Rightarrow w = \arctan(y/x)$.

17.8 Bump Mapping

Shading models are supposed to create realistic-looking surfaces, but the simplest shading methods, such as the Gouraud and Phong models, result in surfaces that are ideally smooth and therefore look artificial, as if made of plastic. Real surfaces are not completely smooth. They may have stains, holes, or cracks in them. At the very least, a real surface features small irregularities. Bump mapping is a method that simulates such irregularities.

The principle is to perturb the normal to the surface at every point before the point is shaded and use the new normal to shade the point. The perturbation can be random, or it may be based on a table. A good example is the surface of a strawberry. It is shiny but not smooth. Anyone familiar with this fruit knows that its surface varies in a complex but regular way. It is possible to prepare a small table that describes the variation in surface height over a small region of the strawberry and apply the table repeatedly to create a "strawberry" effect on any surface.

If the bumps are to be random, no special algorithm is needed. A normal vector $\mathbf{N} = (N_x, N_y, N_z)$ is first normalized, then perturbed by adding random numbers to each of its three components. A component of a unit vector is always in the range $[-1, 1]$, so the random numbers should be drawn from a smaller interval, say, $[-0.001, 0.001]$.

> Bump: A small area raised above the level of the surrounding surface; protuberance.
> —A dictionary definition.

If the bumps are not random, we assume that a bump table $B(u,w)$ is given, that specifies the bump size for every surface point. Note that B is a table of numbers, not of points or vectors. The value of each table entry $B(u,w)$ indicates by how much the corresponding surface point $\mathbf{P}(u,w)$ is to be raised (or lowered, if $B(u,w)$ is negative). Such a bump table can be generated by a drawing/painting program, or by scanning a picture. The surface is denoted, as usual, by $\mathbf{P}(u,w)$, the two partial derivatives are denoted $\mathbf{P}_u(u,w)$ and $\mathbf{P}_w(u,w)$, the normal is the cross-product $\mathbf{N}(u,w) = \mathbf{P}_u(u,w) \times \mathbf{P}_w(u,w)$, and the unit normal vector is denoted by $\mathbf{n}(u,w)$. The mapping rule is as follows: Each surface point should be raised by $B(u,w)\mathbf{n}$ (note that this defines both the magnitude and direction of the raise). We use the notation $\mathbf{P}^*(u,w) = \mathbf{P}(u,w) + B(u,w) \cdot \mathbf{n}$, where $\mathbf{P}^*(u,w)$ is the new surface point. The new tangent vectors are

$$\begin{aligned}\mathbf{P}^*_u(u,w) &= \mathbf{P}_u(u,w) + B_u(u,w) \cdot \mathbf{n} + B(u,w) \cdot \mathbf{n}_u,\\ \mathbf{P}^*_w(u,w) &= \mathbf{P}_w(u,w) + B_w(u,w) \cdot \mathbf{n} + B(u,w) \cdot \mathbf{n}_w.\end{aligned}$$

Next, we note that the derivatives $\mathbf{n}_u$ and $\mathbf{n}_w$ of the unit normal depend on the curvature of the surface. If the surface is not highly curved, the magnitudes of those derivatives are small and they can be ignored. We can therefore write $\mathbf{P}^*_u \approx \mathbf{P}_u + B_u \cdot \mathbf{n}$ and $\mathbf{P}^*_w \approx \mathbf{P}_w + B_w \cdot \mathbf{n}$.

The normal $\mathbf{N}^*(u,w)$ to the new surface $\mathbf{P}^*(u,w)$ is the cross-product

$$\begin{aligned}\mathbf{N}^* &= \mathbf{P}^*_u \times \mathbf{P}^*_w\\ &= (\mathbf{P}_u + B_u \cdot \mathbf{n}) \times (\mathbf{P}_w + B_w \cdot \mathbf{n})\\ &= \mathbf{P}_u \times \mathbf{P}_w + B_u \cdot \mathbf{n} \times \mathbf{P}_w + B_w \cdot \mathbf{P}_u \times \mathbf{n}\\ &= \mathbf{N} + B_u \cdot \mathbf{n} \times \mathbf{P}_w + B_w \cdot \mathbf{P}_u \times \mathbf{n}\\ &= \mathbf{N} + \alpha\mathbf{P}_u + \beta\mathbf{P}_w.\end{aligned}$$

It is the sum of the original normal $\mathbf{N}$ and of two cross-products, each scaled by a derivative of B. The first cross-product is perpendicular to both $\mathbf{P}_w$ and $\mathbf{n}$, so it points in the direction of $\mathbf{P}_u$. The second cross product is perpendicular to both $\mathbf{P}_u$ and $\mathbf{n}$, so it points in the direction of $\mathbf{P}_w$.

The new normal is now used to shade the surface point. Note that we don't actually move or "bump" the surface point $\mathbf{P}(u,w)$ to $\mathbf{P}^*(u,w)$. We just compute a new normal $\mathbf{N}^*$ and use it to shade the point.

It is also important to note that the bump map B itself is not used by the algorithm, only its derivatives. In practice, good values for the derivatives can be obtained by subtracting

$$\begin{aligned}B_u(u,w) &= B(u+s,w) - B(u-s,w),\\ B_w(u,w) &= B(u,w+s) - B(u,w-s).\end{aligned}$$

where s is a convenient step size, and the indexes $u \pm s$, $w \pm s$ should be calculated modulo the size of the bump table.

17.9 Particle Systems

Part III of this book discusses curves and surfaces, but the surfaces it deals with are solid. They are not suitable for representing non-solid or particulate objects such as hair, fur, grass, dust, fire, sparks, smoke, fog, and water spray, as well as glowing trails and snow storms. Such objects can be created and manipulated by a technique called particle systems, the brainchild, in 1983, of computer graphics pioneer William Reeves [Reeves 83]. The main idea is to construct a non-solid object from a large number of particles that move and interact according to rules implemented by software. Typically, software for particle systems generates and manipulates three-dimensional particles, but two-dimensional particle systems are much simpler to implement and often produce satisfactory results.

The following is a list of the main components of a particle system:

- Particle. A particle is visible (but only for its lifetime) and has attributes such as shape (a small bitmap), color (or surface texture), size, weight, lifetime, and velocity. Any of these may be fuzzy, i.e., may be specified by a central value and a range of variability allowed around that value. Thus, the lifetime of a particle may be specified as 60 animation frames $\pm 25\%$, i.e., from $60 - 15$ to $60 + 15$ frames. A complex object is modeled by generating many particles and moving them together, much as a flock of birds or a school of fish, to achieve the desired visual effect. Once a particle is created, it behaves according to its built-in attributes and the user has no further control over it (this is appropriate, since a particle system may consist of many thousands of particles at any given time).

- Emitter. This component generates particles and emits them, but is itself invisible; it does not appear in the scene. An emitter has two sets of attributes. The first includes properties such as the path followed by the emitter, its shape, speed, and rate of particle creation. Also, an emitter may emit several types of particles. The second set consists of the attributes of the types of particles generated by the emitter. If the emitter has a shape (if it is more than just a point), it is typically a simple polyhedron whose every face emits particles in a direction perpendicular to the face.

- A prime (or super) emitter. This emitter creates emitters, which in turn spawn and spray particles. Thus, a single prime emitter may produce a shower of (invisible) emitters, each producing a shower of particles, similar to the effect produced by high-quality fireworks that explode several times to produce showers of showers of sparks.

- Deflector. This component deflects any particles hitting it. A deflector is a flat plane (in a two-dimensional particle system it is a line segment) that acts as a mirror, but is itself invisible. The attributes of a deflector are its position in space and its size.

- Absorber. An absorber is a plane, much like a deflector, but it absorbs any particles hitting it. A sophisticated particle system may allow a certain percentage of particles to pass through an absorber.

- Blocker. A blocker is a region of space in which particles are not visible. They disappear momentarily as if they are passing behind an unseen object, and then reappear. The background color is not affected by a blocker.

- Force. This is a vector associated with every point in space. When a particle is located at point **P**, it feels the force at that point, and this changes its velocity (direction and speed). Forces are used to simulate the effects of wind, gravity, friction, and electric and magnetic fields.

The software of a particle system consists of a loop where the following tasks are executed in each iteration:

- Update. The positions of all existing particles are updated based on rules that must include the effects of deflectors and forces. Particles past the end of their lives are deleted, as are also those particles that happen to hit an absorber. The positions of all emitters and prime emitters are also updated. New particles are produced from the emitters depending on each emitter's spawning rate. Collisions between particles may occur, but they are generally ignored because they are expensive to simulate and also because the number of particles may be huge and a viewer cannot follow the precise path of every particle to make sure all collisions are simulated properly.

- Rendering. If a particle is a single pixel or a very small bitmap, its color may not depend on the existing light sources and may be constant or may vary with time (for example, a particle in a gas flame may change color from blue to red). If the particle is bigger, it may have to be shaded according to the existing light sources. However, this type of shading does not require high-precision computations and may be done sloppily because of the large number of particles and because they often move at high speeds, so a viewer may not be able to tell when particle colors are wrong.

Similarly, visible surface determination may not be a problem. With thousands of particles, it is impractical to determine the visibility of every one. Thus, a particle system looks best in applications where visibility determination is irrelevant. Fog is a good example of such a case. The software simply renders every particle, without regard to its visibility, and the result looks fine, because of the large number of particles and because they are so similar (or even identical).

A particle system may be animated (also referred to as snow) or static (often called hair). In the former case, the life of a particle is distributed over time, while in the latter case the entire life (the path and color of the particle) is rendered when the particle is generated, so it may look like a thin strand of hair (where the thickness may vary along the path).

Figure 17.18 shows examples of (a) Snow flakes, (b) bubbles, and (c) water jet, constructed by the *particle illusion* software [wondertouch 10]. See color Plates B.2 and E.2 for more examples.

The following is quoted from *The Particle System API*, free particle software in C++ by David K. McAllister [McAllister 10]:

> The Particle System API allows C++ application developers to easily include dynamic simulations of groups of moving objects. The API is much lighter weight than a full physics engine. It is especially useful for eye candy in games and screen savers, but is also used in off-line animation software.
>
> With the Particle System API you create a group of particles, then describe the components of the particle effect using actions like Gravity(), Explosion(), Bounce(), etc. You apply the actions to the particle group at each time step,

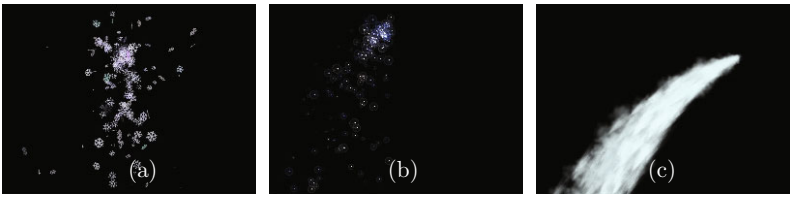


Figure 17.18: Particle System, an Example.

then read back the particle positions and other attributes into your app, or send them directly to the GPU as a vertex array or as geometry instances.

17.10 Mosaics

The ancients believed that the world is made up of four elements. They did not have much science, but they certainly had lots of art. One art form that was more developed and used in ancient times than today is mosaic. Mosaic is the art of constructing images (pictures or patterns) with small pieces of colored material, such as stone, glass, shells, minerals, tiles, or ceramics. The pieces can be squares, triangles, or hexagons (such shapes fill up a two-dimensional space), but they may also have irregular shapes. The pieces are embedded in mortar and each serves as a "pixel" of the mosaic. Figure 17.19 shows two beautiful mosaics uncovered by archeologists. Mosaics can be preserved, buried in dirt or clay, for long periods of times. The oldest known examples date back to the third century B.C.

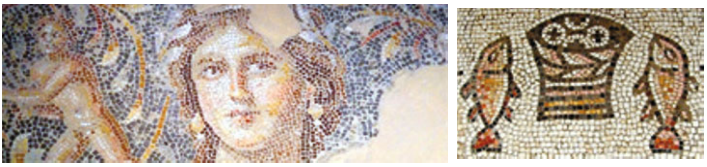


Figure 17.19: Ancient Mosaics.

Making a mosaic is time consuming and requires much effort and concentration, which is perhaps why today this art is not very popular. However, 1933 saw the birth of photomosaic, the modern form of this ancient art. A photomosaic is an image (the primary) that consists of small regions called tiles (normally equal-size squares or rectangles), each of which is a small image (taken from a large library of images) whose average color is similar to the average color of the tile in the primary image. When a photomosaic is viewed from a distance, the details of the small images become blurred and the eye perceives each as a large pixel. Naturally, a close examination reveals the details of the small components of the primary. These components resemble a huge

pile of images, but viewers generally agree that photomosaic is an art form. Reference [mosaic history 10] is a history of photomosaics.

(The term photomosaic has another meaning. It may also be used to describe a large photograph made of small, overlapping images that are stitched together. Such a panorama is common with photos taken by satellites or other space vehicles.)

To some viewers, a photomosaic may seem to reveal secrets in the primary image. This is especially true if the library images are somehow related to the primary, such as when (1) the primary is a person and the images are portraits of family and friends or (2) when the images are parts of the primary itself.

The main tasks in preparing a photomosaic are to partition the primary image into many small tiles, determine the average color of each tile, and search among the many images in the library for the image that offers the best match to the tile. It is easy to see why these tasks lend themselves to automatic execution by a computer. The first examples of computerized photomosaics appeared in 1993.

The detailed steps in creating a photomosaic by computer are as follows:

- Select the primary image and a large library (at least hundreds and preferably thousands) of images.

- Decide on the size and shape of the tiling grid. The triangle, square, and hexagon are the only regular polygons that, in a tessellation, fill the plane (Figure 17.20). Thus, the tiling grid may consist of one of them. If squares or rectangles are used as tiles, they may be laid in rows and columns or in a brick pattern, but their locations may also be random and irregular (a scattered layout). It is possible to create a photomosaic by placing many small squares or rectangles over the primary, and some may even be tilted. Such a random grid does not fully cover the plane, but areas in the primary image not covered by any tile are simply left in their original colors.

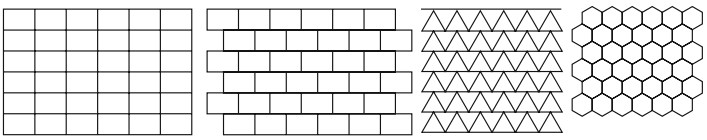


Figure 17.20: Regular Polygons.

- Once the size and shape of the tiles have been determined, each image in the library is scaled to this size and shape (this is possible only if the tiles are identical). It helps if all the library images have the same aspect ratio.

- The main loop starts at this point. It scans the primary tile by tile. The average color of the current tile is determined and the library is scanned for the image that best matches this color. This step must implement the following points:

1. For each library image, the software maintains a counter. Once an image has been selected, its count is incremented by 1. The software flags all the library images that are feasible matches to a given tile and selects the one with the smallest count.

This produces better results in cases where the same library image would be selected often, while other images that have the same average color would be ignored.

2. There must be a user-controlled threshold parameter for the software to decide on a match. If the difference between the average colors of a tile and of a library image is less than the threshold, the image is considered a feasible match.

3. Large values of the threshold lead to many tiles without a match, in which case the tile retains its original color. Thus, large values of the threshold parameter result in a mosaic where many tiles are not replaced by library images. In such a case, the final mosaic resembles the primary image, but close inspection shows the unmodified tiles, which to some users may look like cheating.

Here is how a tile T is matched to a library image I. We assume that both are squares of $n \times n$ pixels. Denote the color of a pixel P by C, and assuming that C is given as the 24 bits `R7R6R5R4R3R2R1R0`, `G7G6G5G4G3G2G1G0`, and `B7B6B5B4B3B2B1B0`, we interleave these bits to create the 24-bit integer

$$P = \texttt{G7R7B7G6R6B6G5R5B5G4R4B4G3R3B3G2R2B2G1R1B1G0R0B0}.$$

Section 2.13.1 shows why P is a good candidate for the title "average of the three color components of C." Once this is clear, it is not hard to see why the following expression—where $T(i,j)$ denotes the 24-bit color average of pixel $T(i,j)$ and similarly for pixel $I(i,j)$—is a reasonable measure of the difference of color averages

$$D(T,I) = \frac{\sum_{i=1}^{n}\sum_{j=1}^{n}\big(T(i,j)-I(i,j)\big)^2}{n^2}.$$

Any library image I where $D(T,I) <$ Threshold is a feasible match for tile T.

If the tiles are squares or rectangles, it may be possible to obtain much better results with variable-size tiles. The principle is to locate areas of high detail (or high noise) in the primary and partition each tile in these areas into four smaller tiles. In areas of very high image noise, a tile may be partitioned twice, resulting in 4×4 small tiles. Section 18.7 discusses the quadtree data structure, which is the natural choice for partitioning a region into four smaller regions.

In order to implement such a refinement, we need a criterion for measuring the amount of image noise in a given region. Two methods immediately suggest themselves as follows:

- Apply a Fourier transform or a wavelet transform (Chapter 25) to the pixels of the region in order to identify the image frequencies in the region. The more high-frequency data is obtained, the noisier the region.

- Apply an edge detection algorithm to determine the number and lengths of edges in the region. The more edges, the noisier the region.

An algorithm that employs this technique starts by partitioning the primary image into four quadrants. Each is checked for the amount of noise and, if needed, is partitioned into four subquadrants recursively. Partitioning stops when a "quiet" subquadrant is reached or when the subquadrants are too small (as determined by another user-controlled parameter).

Another approach to photomosaics is to partition the primary image into a small number of large tiles, and then convert each tile to a mosaic. This creates a mosaic of mosaics.

> Exploring this idea, however, I suggest an alternative: In this manner, weaving fictional elements in and out of each other, the mosaic is turned into a "mosaic of mosaics" where the themes of individual sections are more readily juxtaposed against each other.
>
> —J. M. McDermott.

The basic color matching described here is based on average colors and a threshold parameter, but this approach can be varied. The following is a short list of variations on the theme of tile–image matching.

- Have just one library image. This image replaces every tile in the primary and its color is corrected according to the color of the current tile.
- Have several or many library images and assign them to primary tiles at random. Again, when an image replaces a tile, its color is corrected.
- Select the library image for each tile manually. Clearly, a user can do a good job of matching a library image to a given tile, but only if the library (and the number of tiles) is small.
- Match library images to tiles by their edges. When the loop arrives at a tile, it checks the tile for edges and selects the library image that has the most similar edge structure. Such matching can produce excellent results, but edge detection is slow and not absolutely reliable (any edge-locating algorithm may find edges where there are none and may miss existing edges).

The last topic that needs to be covered is color matching. Given a primary tile and a matching library image with average colors T and I, respectively, we want to vary the colors of the image's pixels such that their new average will be T. The simplest method is to set the colors of all the pixels of the image to T, but this results in uniform library images and an ugly mosaic with many uniform tiles.

Better results are achieved when the difference $T - I$ is added to each image pixel (this is referred to as color shift). Assume that the image consists of n pixels, each with an average color p_i. The average color of the image is $I = \frac{1}{n}\sum p_i$ and the correction changes the color of a pixel from p_i to $q_i = p_i + T - I$. The new average color of the entire image is now

$$\frac{1}{n}\sum q_i = \frac{1}{n}\sum p_i + \frac{1}{n}\sum T - \frac{1}{n}\sum I = I + T - I = T.$$

However, some of the new pixel colors q_i may be outside the range of reproducible colors, so the color of such pixels should be corrected differently, by scaling rather than shifting. The scaling transformation is $q_i = p_i\frac{T}{I}$ and it tends to brighten the pixels (if $T > I$) or darken them (in the opposite case). Experience shows that color shifts, replaced by color scaling when necessary, result in reasonable color correction. (See Plate L.4 for an example of a computer-generated mosaic.)

17.10.1 ASCII Art

The ASCII code was developed in the early 1960s in response to the growing capabilities of printers. Older printers could print only digits and uppercase letters, but the printers available in the 1960s could also print lowercase letters and punctuation marks. Modems also came into use in the 1960s, which made it possible to send and receive data on telephone lines. This raised the need for control characters (symbols that do not get printed but act as commands when data is sent and received). The ASCII codes consist of seven data bits and a parity bit, and so can encode a set of $2^7 = 128$ characters. Of those, 95 characters are printable and the remaining 33 are control characters that specify commands such as carriage return, end of text, and ring the bell.

Naturally, the ASCII code was meant to input data into the computer (originaly from punched cards) and to send data to be printed. Like many other new concepts, ideas, and devices, some users tried to employ printers and printed text in new ways, and one of these new applications was ASCII art, a graphics technique that employs ASCII characters as large pixels of an image.

Images in ASCII art were created as early as 1966 by computer-art pioneer Kenneth Knowlton. These may be the earliest known examples of this art form.

A curiosity. ASCII art was used in the early days of e-mail to send images in addition to text. Today it is easy to attach an image to a message, so this application of ASCII art seems to be dead.

Figure 17.21 shows examples of ASCII art (see also Plate T.1).

Figure 17.21: ASCII Art.

The principle of ASCII art is to partition an image (the primary) into square tiles, measure the average color of a tile, and replace the tile with a character that best matches it. In the simplest version of this art, the tiles have the same dimensions, so it

makes sense to use a fixed-width (or non-proportional) font, where characters have the same width. A common example of such a font in current operating systems is Courier.

Before any primary image can be converted to ASCII, the set of ASCII printable characters has to be examined and the average gray of each character determined. There are only 95 printable ASCII characters and many may have similar average gray. Thus, there may be only 40–50 different gray averages in the character set. We denote the number of different gray averages by G.

The primary is often a color image, but the ASCII characters are black. Thus, the first step in matching a tile to a character is to determine the tile's brightness. This is done by converting the tile's color from its original color space to the YCbCr color space and scaling the luminance Y to the interval $[0, G-1]$. Once this is done, it is easy to match the tile to a character.

As with other forms of art, it is possible to modify the basic process in various ways. The following is a list of ideas for those keen on implementations:

- Partition the primary image into hexagons instead of squares. It is also possible to partition the primary into random squares.
- A binary version partitions the primary into small tiles and replaces each tile with a 0 or a 1.
- Another version employs thick and thin fonts. This increases the number of gray averages significantly.
- Quadtrees (Section 18.7) can be used, as discussed on Page 885, to partition the primary into variable-size squares according to the noise in individual parts of the primary. Large tiles are replaced with large characters (such as `M` and `#`) or characters from large-size fonts, while small tiles are replaced with small characters or characters from small-size fonts.

17.10.2 Mechanical Mirrors

Ancient mirrors were made of shiny, reflecting metal; then came glass. Today, in the age of computers and graphics, there are mechanical mirrors that can render an image in innovative ways, not just by displaying it on a monitor. Imagine standing in front of a large board covered with many small wood blocks, looking at your reflection. Such a wooden mirror, as well as other mechanical mirrors, are the brainchild of Daniel Rozin, a New-York artist [Rozin 11].

> Daniel Rozin is an artist, educator and developer, working in the area of interactive digital art. As an interactive artist Rozin creates installations and sculptures that have the unique ability to change and respond to the presence and point of view of the viewer. In many cases the viewer becomes the contents of the piece and in others the viewer is invited to take an active role in the creation of the piece. Even though computers are often used in Rozin's work, they are seldom visible.
>
> —From http://www.smoothware.com/danny/newbio.html

The principle is simple and attractive. The mirror consists of 830 wood blocks, that can be individually rotated under computer control. When rotated, a block reflects varying amounts of light, depending on the grain of the wood and the angle to which it

is set. A camera hidden in the mirror collects the image in front of the mirror, converts it to 830 large pixels, and uses the grayscale G of each pixel to rotate the corresponding wood block such that its reflection is close to G.

This process takes a fraction of a second and it is done in real time. As the observer moves in front of the mirror, its "reflection" is constantly updated. The image is not clear and sharp, as in a traditional, optical mirror. It appears ghostly and leaves a dark, haunted trace as it is updated.

It was a magnificent, sprawling artist's
rendering of an imagined fairy homeland.
—Terry Brooks, *Magic Kingdom For Sale, Sold!* (1986)