algorithms may perform extra work when modeling diffuse phenomena that change little over large areas of an image or between images made from different viewpoints. On the other hand, view-independent algorithms model diffuse phenomena efficiently but require overwhelming amounts of storage to capture enough information about specular phenomena.

Ultimately, all these approaches attempt to solve what Kajiya [KAJI86] has referred to as the **rendering equation,** which expresses the light being transferred from one point to another in terms of the intensity of the light emitted from the first point to the second and the intensity of light emitted from all other points that reaches the first and is reflected from the first to the second. The light transferred from each of these other points to the first is, in turn, expressed recursively by the rendering equation. Kajiya presents the rendering equation as

$$I(x, x') = g(x, x')\left[\epsilon(x, x') + \int_S \rho(x, x', x'')\, I(x', x'')dx''\right], \qquad (14.31)$$

where x, x', and $x''$are points in the environment; $I(x, x')$ is related to the intensity passing from x' to x; $g(x, x')$ is a geometry term that is 0 when x and x' are occluded from each other, and $1 / r^2$ when they are visible to each other, where r is the distance between them; and $\epsilon(x, x')$ is related to the intensity of light that is emitted from x' to x. The initial evaluation of $g(x, x')\epsilon(x, x')$ for x at the viewpoint accomplishes visible-surface determination in the sphere about x. The integral is over all points on all surfaces S. $\rho(x, x', x'')$ is related to the intensity of the light reflected (including both specular and diffuse reflection) from x'' to x from the surface at x'. Thus, the rendering equation states that the light from x' that reaches x consists of light emitted by x' itself and light scattered by x' to x from all other surfaces, which themselves emit light and recursively scatter light from other surfaces.

As we shall see, how successful an approach is at solving the rendering equation depends in large part on how it handles the remaining terms and the recursion, on what combinations of diffuse and specular reflectivity it supports, and on how well the visibility relationships between surfaces are modeled.

14.7 RECURSIVE RAY TRACING

In this section, we extend the basic ray-tracing algorithm of Section 13.4 to handle shadows, reflection, and refraction. This simple algorithm determined the color of a pixel at the closest intersection of an eye ray with an object, by using any of the illumination models described previously. To calculate shadows, we fire an additional ray from the point of intersection to each of the light sources. This is shown for a single light source in Fig.14.33, which is reproduced from a paper by Appel [APPE68]—the first paper published on ray tracing for computer graphics. If one of these **shadow rays** intersects any object along the way, then the object is in shadow at that point and the shading algorithm ignores the contribution of the shadow ray's light source.

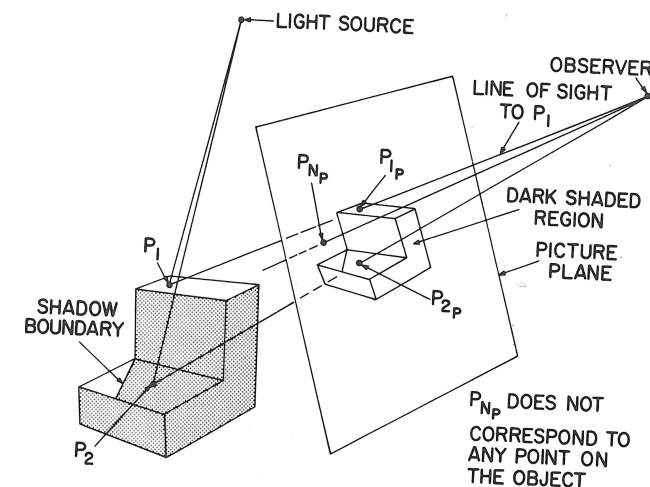

Figure 14.33 Determining whether a point on an object is in shadow. (Courtesy of Arthur Appel, IBM T.J. Watson Research Center.).

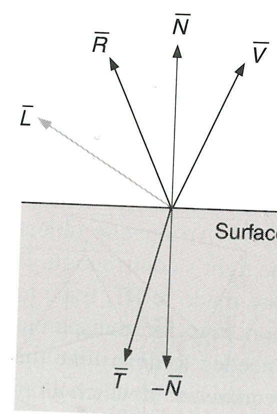

Figure 14.34 Reflection, refraction, and shadow rays are spawned from a point of intersection.

The illumination model developed by Whitted [WHIT80] and Kay [KAY79a] fundamentally extended ray tracing to include specular reflection and refractive transparency. Color Plate 41 is an early picture generated with these effects. In addition to shadow rays, Whitted's recursive ray-tracing algorithm conditionally spawns **reflection rays** and **refraction rays** from the point of intersection, as shown in Fig. 14.34. The shadow, reflection, and refraction rays are often called **secondary rays**, to distinguish them from the **primary rays** from the eye. If the object is specularly reflective, then a reflection ray is reflected about the surface normal in the direction of $\overline{R}$, which may be computed as in Section 14.1.4. If the object is transparent, and if total internal reflection does not occur, then a refraction ray is sent into the object along $\overline{T}$ at an angle determined by Snell's law, as described in Section 14.5.2. (Note that your incident ray may be oppositely oriented to those in these sections.)

Each of these reflection and refraction rays may, in turn, recursively spawn shadow, reflection, and refraction rays, as shown in Fig. 14.35. The rays thus form a **ray tree,** such as that of Fig. 14.36. In Whitted's algorithm, a branch is terminated if the reflected and refracted rays fail to intersect an object, if some user-specified maximum depth is reached or if the system runs out of storage. The tree is evaluated bottom-up, and each node's intensity is computed as a function of its children's intensities.

We can represent Whitted's illumination equation as

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + \sum_{1 \le i \le m} S_i f_{att_i} I_{p\lambda_i} [k_d O_{d\lambda} (\overline{N} \cdot \overline{L}_i) + k_s (\overline{N} \cdot \overline{H}_i)^n] + k_s I_{r\lambda} + k_t I_{t\lambda}\, , \qquad (14.32)$$

where $I_{r\lambda}$ is the intensity of the reflected ray, k_t is the **transmission coefficient**

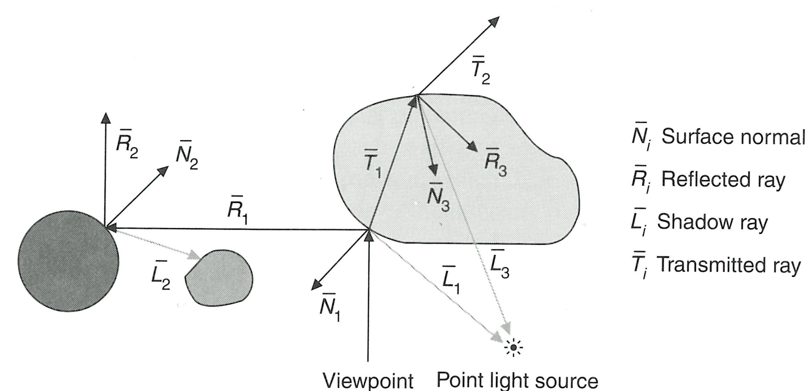

Figure 14.35 Rays recursively spawn other rays.

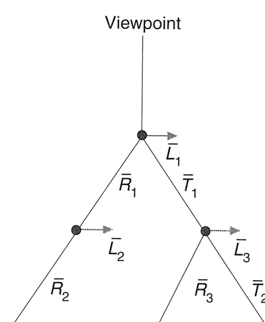

Figure 14.36
The ray tree for Fig. 14.35.

ranging between 0 and 1, and $I_{t\lambda}$ is the intensity of the refracted transmitted ray. Values for $I_{r\lambda}$ and $I_{t\lambda}$ are determined by recursively evaluating Eq. (14.32) at the closest surface that the reflected and transmitted rays intersect. To approximate attenuation with distance, Whitted multiplied the I_λ calculated for each ray by the inverse of the distance traveled by the ray. Rather than treating S_i as a delta function, as in Eq. (14.22), he also made it a continuous function of the k_t of the objects intersected by the shadow ray, so that a transparent object obscures less light than an opaque one at those points it shadows.

Prog.14.1 shows pseudocode for a simple recursive ray tracer. RT_trace determines the closest intersection the ray makes with an object and calls RT_shade to determine the shade at that point. First, RT_shade determines the intersection's ambient color. Next, a shadow ray is spawned to each light on the side of the surface being shaded to determine its contribution to the color. An opaque object blocks the light totally, whereas a transparent one scales the light's contribution. If we are not too deep in the ray tree, then recursive calls are made to RT_trace to handle reflection rays for reflective objects and refraction rays for transparent objects. Since the indices of refraction of two media are needed to determine the direction of the refraction ray, the index of refraction of the material in which a ray is traveling can be included with each ray. RT_trace retains the ray tree only long enough to determine the current pixel's color. If the ray trees for an entire image can be preserved, then surface properties can be altered and a new image recomputed relatively quickly, at the cost of only reevaluating the trees. Sequin and Smyrl [SEQU89] present techniques that minimize the time and space needed to process and store ray trees.

Program 14.1

Pseudocode for simple recursive ray tracing without antialiasing.

```
select center of projection and window on view plane;
for ( each scan line in image ) {
    for ( each pixel in scan line ) {
        determine ray from center of projection through pixel;
```



```
        pixel = RT_trace (ray, 1);
    }
}

/* Intersect ray with objects and compute shade at closest intersection. */
/* Depth is current depth in ray tree. */

RT_color RT_trace ( RT_ray ray, int depth )
{
    determine closest intersection of ray with an object;
    if ( object hit ) {
        compute normal at intersection;
        return RT_shade (closest object hit, ray, intersection, normal, depth);
    }
    else
        return BACKGROUND_VALUE;
}

/* Compute shade at point on object, tracing rays for shadows, reflection, refraction.*/

RT_color RT_shade (
    RT_object object,          /* Object intersected */
    RT_ray ray,                /* Incident ray */
    RT_point point,            /* Point of intersection to shade */
    RT_normal normal,          /* Normal at point */
    int depth )                /* Depth in ray tree */
{
RT_color color;                /* Color of ray */
RT_ray rRay, tRay, sRay;       /* Reflected, refracted, and shadow rays */
RT_color rColor, tColor;       /* Reflected and refracted ray colors */

    color = ambient term;
    for ( each light ) {
        sRay = ray to light from point;
        if ( dot product of normal and direction to light is positive ) {
            compute how much light is blocked by opaque and transparent surfaces,
            and use to scale diffuse and specular terms before adding them to color;
        }
    }
    if ( depth < maxDepth ) {          /* Return if depth is too deep. */
        if ( object is reflective ) {
            rRay = ray in reflection direction from point;
            rColor = RT_trace (rRay, depth + 1);
            scale rColor by specular coefficient and add to color;
        }
        if ( object is transparent ) {
            tRay = ray in refraction direction from point;
            if ( total internal reflection does not occur ) {
                tColor = RT_trace (tRay, depth + 1);
                scale tColor by transmission coefficient and add to color;
```

```
                }
            }
        }
        return color;        /* Return color of ray. */
    }
```

Figure 14.35 shows a basic problem with how ray tracing models refraction: The shadow ray $\overline{L}_3$ is not refracted on its path to the light. In fact, if we were to simply refract $\overline{L}_3$ from its current direction at the point where it exits the large object, it would not end at the light source. In addition, when the paths of rays that are refracted are determined, a single index of refraction is used for each ray.

Ray tracing is particularly prone to problems caused by limited numerical precision. These show up when we compute the objects that intersect with the secondary rays. After the x, y, and z coordinates of the intersection point on an object visible to an eye ray have been computed, they are then used to define the starting point of the secondary ray for which we must determine the parameter t (Section 13.4.1). If the object that was just intersected is intersected with the new ray, it will often have a small, nonzero t because of numerical-precision limitations. If not dealt with, this false intersection can result in visual problems. For example, if the ray were a shadow ray, then the object would be considered as blocking light from itself, resulting in splotchy pieces of incorrectly "self-shadowed" surface. A simple way to solve this problem for shadow rays is to treat as a special case the object from which a secondary ray is spawned, so that intersection tests are not performed on it. Of course, this does not work if objects are supported that really could obscure themselves or if transmitted rays have to pass through the object and be reflected from the inside of the same object. A more general solution is to compute abs(t) for an intersection, to compare it with a small tolerance value, and to ignore it if it is below the tolerance.

The paper Whitted presented at *SIGGRAPH '79* [WHIT80], and the movies he made using the algorithm described there, started a renaissance of interest in ray tracing. Recursive ray tracing makes possible a host of impressive effects—such as shadows, specular reflection, and refractive transparency—that were difficult or impossible to obtain previously. In addition, a simple ray tracer is quite easy to implement. Consequently, much effort has been directed toward improving both the algorithm's efficiency and its image quality. For more detail, see Section 16.12 of [FOLE90] and [GLAS89].

14.8 RADIOSITY METHODS

Although ray tracing does an excellent job of modeling specular reflection and dispersionless refractive transparency, it still makes use of a directionless ambient lighting term to account for all other global lighting contributions. Approaches based on thermal-engineering models for the emission and reflection of radiation eliminate the need for the ambient-lighting term by providing a more accurate treatment of interobject reflections. First introduced by Goral, Torrance,

Greenberg, and Battaile [GORA84] and by Nishita and Nakamae [NISH85], these algorithms assume the conservation of light energy in a closed environment. All energy emitted or reflected by every surface is accounted for by its reflection from or absorption by other surfaces. The rate at which energy leaves a surface, called its **radiosity,** is the sum of the rates at which the surface emits energy and reflects or transmits it from that surface or other surfaces. Consequently, approaches that compute the radiosities of the surfaces in an environment have been named **radiosity methods.** Unlike conventional rendering algorithms, radiosity methods first determine all the light interactions in an environment in a view-independent way. Then, one or more views are rendered, with only the overhead of visible-surface determination and interpolative shading.

14.8.1 The Radiosity Equation

In the shading algorithms considered previously, light sources have always been treated separately from the surfaces they illuminate. In contrast, radiosity methods allow any surface to emit light; thus, all light sources are modeled inherently as having area. Imagine breaking up the environment into a finite number n of discrete patches, each of which is assumed to be of finite size, emitting and reflecting light uniformly over its entire area. If we consider each patch to be an opaque Lambertian diffuse emitter and reflector, then, for surface i,

$$B_i = E_i + \rho_i \sum_{1 \le j \le n} B_j F_{j-i} \frac{A_j}{A_i}. \tag{14.33}$$

B_i and B_j are the radiosities of patches i and j, measured in energy/unit time/unit area (i.e., W/m^2). E_i is the rate at which light is emitted from patch i and has the same units as radiosity. ρ_i is patch i's reflectivity and is dimensionless. F_{j-i} is the dimensionless **form factor** or **configuration factor,** which specifies the fraction of energy leaving the entirety of patch j that arrives at the entirety of patch i, taking into account the shape and relative orientation of both patches and the presence of any obstructing patches. A_i and A_j are the areas of patches i and j.

Equation (14.33) states that the energy leaving a unit area of surface is the sum of the light emitted plus the light reflected. The reflected light is computed by scaling the sum of the incident light by the reflectivity. The incident light is in turn the sum of the light leaving the entirety of each patch in the environment scaled by the fraction of that light reaching a unit area of the receiving patch. B_jF_{j-i} is the amount of light leaving a unit area of A_j that reaches all of A_i. Therefore, it is necessary to multiply by the area ratio A_j / A_i to determine the light leaving all of A_j that reaches a unit area of A_i.

Conveniently, a simple reciprocity relationship holds between form factors in diffuse environments,

$$A_iF_{i-j} = A_jF_{j-i}. \tag{14.34}$$

Thus, Eq. (14.33) can be simplified, yielding