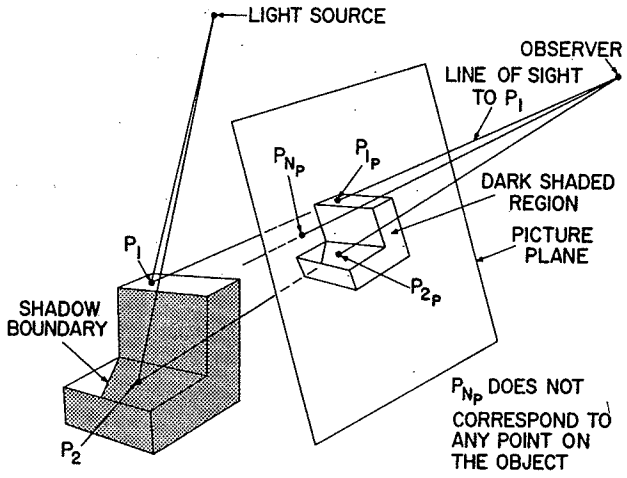

Fig. 16.51 Determining whether a point on an object is in shadow. (Courtesy of Arthur Appel, IBM T.J. Watson Research Center.)

16.51, 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 16.52 shows two pictures that Appel rendered with this algorithm, using a pen plotter. He simulated a halftone pattern by placing a different size "+" at each pixel in the grid, depending on the pixel's intensity. To compensate for the grid's coarseness, he drew edges of visible surfaces and of shadows using a visible-line algorithm.

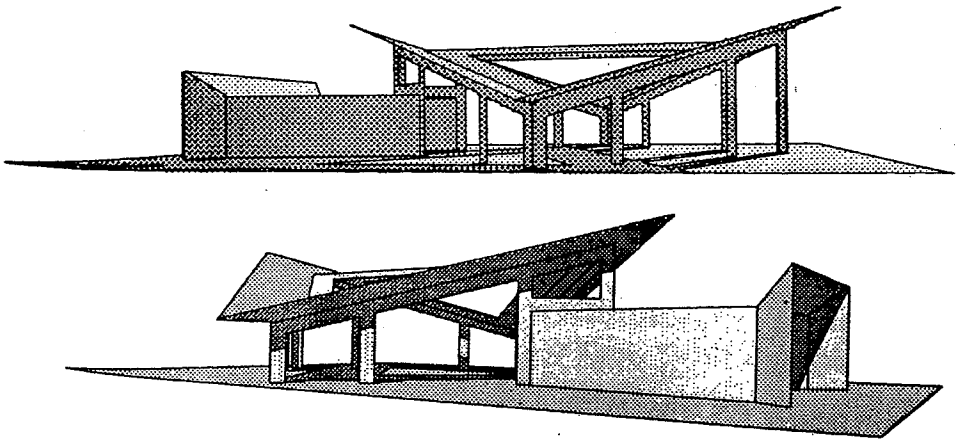

Fig. 16.52 Early pictures rendered with ray tracing. (Courtesy of Arthur Appel, IBM T.J. Watson Research Center.)

The illumination model developed by Whitted [WHIT80] and Kay [KAY79a] fundamentally extended ray tracing to include specular reflection and refractive transparency. Color Plate III.10 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. 16.53. 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 16.1.4. 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 16.5.2. (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. 16.54. The rays thus form a *ray tree*, such as that of Fig. 16.55. 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. Color Plate III.11(a) and (b) were made with a recursive ray-tracing algorithm.

We can represent Whitted's illumination equation as

$$I_\lambda = I_{a\lambda}k_aO_{d\lambda} + \sum_{1 \le i \le m} S_i f_{att_i} I_{p\lambda_i}[k_dO_{d\lambda}(\overline{N} \cdot \overline{L}_i) + k_s(\overline{N} \cdot \overline{H}_i)^n] + k_sI_{r\lambda} + k_tI_{t\lambda}, \quad (16.55)$$

where $I_{r\lambda}$ is the intensity of the reflected ray, k_t is the *transmission coefficient* 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. (16.55) 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. (16.24), he also made it a continuous function

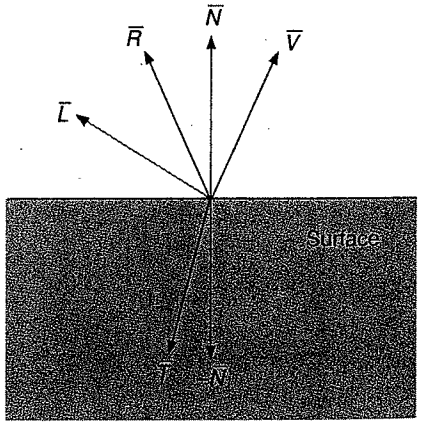

Fig. 16.53 Reflection, refraction, and shadow rays are spawned from a point of intersection.

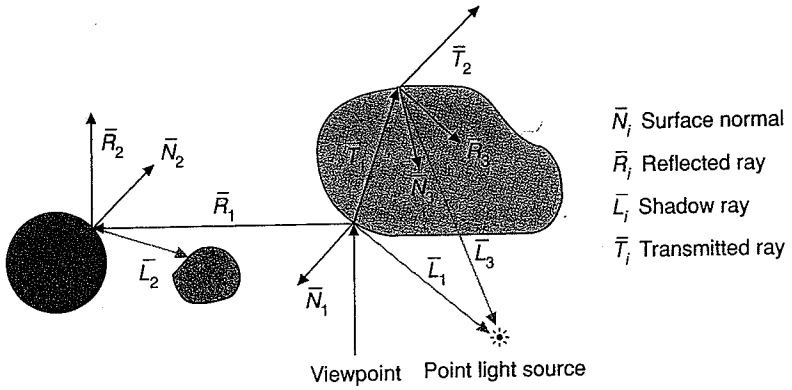

Fig. 16.54 Rays recursively spawn other rays.

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.

Figure 16.56 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

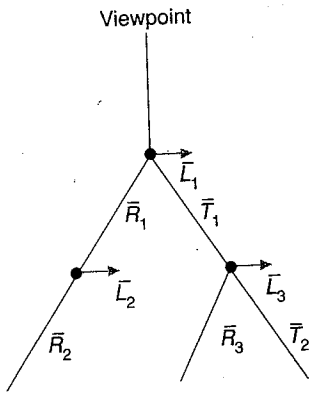

Fig. 16.55 The ray tree for Fig. 16.54.

```
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;
}  /* RT_trace */

/* Compute shade at point on object, tracing rays for shadows, reflection and 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;
        }
    }
```

Fig. 16.56 *(Cont'd.)*

```
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 */
} /* RT_shade */
```

Fig. 16.56 Pseudocode for simple recursive ray tracing without antialiasing.

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.

Figure 16.54 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. Later, we discuss some ways to address these failings.

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 15.10.1). 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. We provide a brief overview of these issues here, and discuss several parallel hardware implementations that take advantage of the algorithm's intrinsic parallelism in Section 18.11.2. For more detail, see [GLAS89].

16.12.1 Efficiency Considerations for Recursive Ray Tracing

Section 15.10.2 discussed how to use extents, hierarchies, and spatial partitioning to limit the number of ray–object intersections to be calculated. These general efficiency techniques are even more important here than in visible-surface ray tracing for several reasons. First, a quick glance at Fig. 16.55 reveals that the number of rays that must be processed can grow exponentially with the depth to which rays are traced. Since each ray may spawn a reflection ray and a refraction ray, in the worst case, the ray tree will be a complete binary tree with $2^n - 1$ rays, where the tree depth is n. In addition, each reflection or refraction ray that intersects with an object spawns one shadow ray for each of the m light sources. Thus, there are potentially $m(2^n - 1)$ shadow rays for each ray tree. To make matters worse, since rays can come from any direction, traditional efficiency ploys, such as clipping objects to the view volume and culling back-facing surfaces relative to the eye, cannot be used in recursive ray tracing. Objects that would otherwise be invisible, including back faces, may be reflected from or refracted through visible surfaces.

Item buffers. One way of speeding up ray tracing is simply not to use it at all when determining those objects directly visible to the eye. Weghorst, Hooper, and Greenberg [WEGH84] describe how to create an *item buffer* by applying a less costly visible-surface algorithm, such as the z-buffer algorithm, to the scene, using the same viewing specification. Instead of determining the shade at each pixel, however, they record in the item buffer pixel the identity of the closest object. Then, only this object needs to be processed by the ray tracer to determine the eye ray's exact intersection for this pixel, so that further rays may be spawned.

Reflection maps. Tracing rays can be avoided in other situations, too. Hall [HALL86] shows how to combine ray tracing with the reflection maps discussed in Section 16.6. The basic idea is to do less work for the secondary rays than for primary rays. Those objects that are not directly visible in an image are divided into two groups on the basis of an estimation of their indirect visibility. Ray tracing is used to determine the global lighting contributions of the more visible ones, whereas indexing into a suitably prepared reflection map handles the others. One way to estimate the extent to which an object is indirectly visible is to measure the solid angle subtended by the directly visible objects as seen from the centroid of the indirectly visible object. If the solid angle is greater than some threshold, then the object will be included in the environment to be traced by reflection rays (this environment