Chapter 9

RECURSIVE RAY TRACING

9.1 Simplification of the illumination model

The light that reaches the eye through a given pixel comes from the surface of an object. The smaller the pixel is, the higher the probability that only one object affects its color, and the smaller the surface element that contributes to the light ray. The energy of this ray consists of three main components. The first component comes from the own emission of the surface. The second component is the energy that the surface reflects into the solid angle corresponding the the pixel, while the third is the light energy propagated by refraction. The origin of the reflective component is either a lightsource (primary reflection) or the surface of another object (secondary, ternary, etc. reflections). The origin of the refracted component is always on the surface of the same object, because this component is going through its interior. We have seen in chapter 3 that the intensity of the reflected light can be approximated by accumulating the following components:

- an ambient intensity I_0, which is the product of the ambient reflection coefficient k_a of the surface and a global ambient intensity I_a assumed to be the same at each spatial point

- a diffuse intensity I_d, which depends on the diffuse reflection coefficient k_d of the surface and the intensity and incidence angle of the light reaching the surface element from any direction

- a specular intensity I_s, which depends on the specular reflection coefficient k_s of the surface and the intensity of the light. In addition, the value is multiplied by a term depending on the angle between the theoretical direction of reflection and the direction of interest and a further parameter n called the specular exponent

- a reflection intensity I_r, which is the product of the (coherent) reflective coefficient k_r of the surface and the intensity of the light coming from the inverse direction of reflection.

Refracted light can be handled similarly.

The following simplifications will be made in the calculations:

- Light rays are assumed to have zero width. This means that they can be treated as lines, and are governed by the laws of geometric optics. The ray corresponding to a pixel of the image can be a line going through any of its points, in practice the ray is taken through its center. A consequence of this simplification is that the intersection of a ray and the surface of an object becomes a single point instead of a finite surface element.

- Diffuse and specular components in the reflected light are considered only for primary reflections; that is, secondary, ternary, etc. incoherent reflections are ignored (these can be handled by the radiosity method). This means that if the diffuse and specular components are to be calculated for a ray leaving a given surface point, then the possible origins are not searched for on the surfaces of other objects, but only the lightsources will be considered.

- When calculating the coherent reflective and refractive components for a ray leaving a given surface point, its origin is searched for on the surface of the objects. Two rays are shot towards the inverse direction of reflection and refraction, respectively, and the first surface points that they intersect are calculated. These rays are called the children of our original ray. Due to multiple reflections and refractions, child

rays can have their own children, and the family of rays corresponding to a pixel forms a binary tree. In order to avoid infinite recurrence, the depth of the tree is limited.

- Incoherent refraction is completely ignored. Implying this would cause no extra difficulties — we could use a very similar model to that for incoherent reflection — but usually there is no practical need for it.

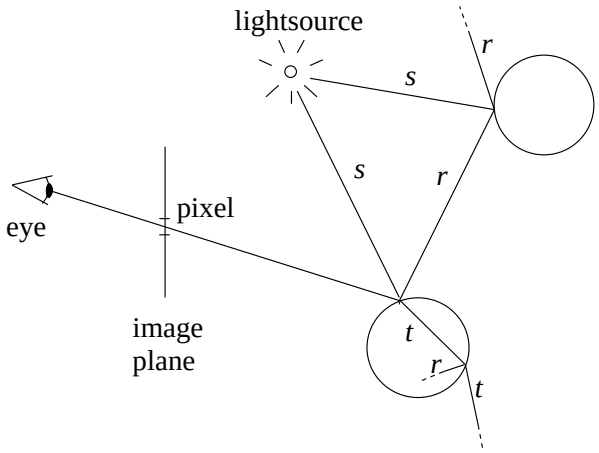

Figure 9.1: Recursive ray tracing

These concepts lead us to recursive ray tracing. Light rays will be *traced backwards* (contrary to their natural direction), that is from the eye back to the lightsources. For each pixel of the image, a ray is shot through it from the eye, as illustrated in figure 9.1. The problem is the computation of its color (intensity). First we have to find the first surface point intersected by the ray. If no object is intersected, then the pixel will either take the color of the background, the color of the ambient light or else it will be black. If a surface point is found, then its color has to be calculated. This usually means the calculation of the intensity components at the three representative wavelengths (R, G, B), that is, the *illumination equation* is evaluated in order to obtain the intensity values. The intensity corresponding to a wavelength is composed of ambient, diffuse, specular, coherent reflective and coherent refractive components. For calculating the diffuse and specular

components, a ray is sent towards each lightsource (denoted by s in figure 9.1). If the ray does not hit any object before reaching the lightsource, then the lightsource illuminates the surface point, and the reflected intensity is computed, otherwise the surface point is in shadow with respect to that lightsource. The rays emanated from the surface point towards the lightsources are really called **shadow rays**. For calculating coherent reflective and refractive components, two rays are sent towards the inverse direction of reflection and refraction, respectively (denoted by r and t in figure 9.1). The problem of computing the color of these child rays is the same as for the main ray corresponding to the pixel, so we calculate them recursively:

```
for each pixel p do
    r = ray from the eye through p;
    color of p = Trace(r, 0);
endfor
```

The subroutine **Trace**(r, d) computes the color of the ray r (a dth order reflective or refractive ray) by recursively tracing its reflective and refractive child rays:

```
Trace(r, d)
    if d > d_max then return background color; endif
    q = Intersect(r);                          // q: object surface point
    if q = null then
      return background color;
    endif
    c = AccLightSource(q);                                  // c: color
    if object (q) is reflective (coherently) then
      r_r = ray towards inverse direction of reflection;
      c += Trace(r_r, d + 1);
    endif
    if object (q) is refractive (coherently) then
      r_t = ray towards inverse direction of refraction;
      c += Trace(r_t, d + 1);
    endif
    return c;
end
```

The conditional return at the beginning of the routine is needed in order to avoid infinite recurrence (due to total reflection, for example, in the interior of a glass ball). The parameter $d_{\max}$ represents the necessary "depth" limit. It also prevents the calculation of too "distant" generations of rays, since they usually hardly contribute to the color of the pixel due to attenuation at object surfaces. The function **Intersect**(r) gives the intersection point between the ray r and the surface closest to the origin of r if it finds it, and *null* otherwise. The function **AccLightSource**(q) computes the accumulated light intensities coming from the individual lightsources and reaching the surface point q. Usually it is also based on function **Intersect**(r), just like **Trace**(r):

```
AccLightSource(q)
    c = ambient intensity + own emission;          // c: color
    for each lightsource l do
        r = ray from q towards l;
        if Intersect(r) = null then
            c += diffuse intensity;
            c += specular intensity;
        endif
    endfor
    return c;
end
```

The above routine does not consider the illumination of the surface point if the light coming from a lightsource goes through one or more transparent objects. Such situations can be approximated in the following way. If the ray r in the above routine intersects only transparent objects with transmission coefficients $k_t^{(1)}, k_t^{(2)}, \ldots, k_t^{(N)}$ along its path, then the diffuse and specular components are calculated using a lightsource intensity of $k_t^{(1)} \cdot k_t^{(2)} \cdot \ldots \cdot k_t^{(N)} \cdot I$ instead of I, where I is the intensity of the lightsource considered. This is yet another simplification, because refraction on the surface of the transparent objects is ignored here.

It can be seen that the function **Intersect**(r) is the key to recursive ray tracing. Practical observations show that 75–95% of calculation time

is spent on intersection calculations during ray tracing. A **brute force** approach would take each object one by one in order to check for possible intersection and keep the one with the intersection point closest to the origin of r. The calculation time would be proportional to the number of objects in this case. Note furthermore that the function **Intersect**(r) is the only step in ray tracing where the complexity of the calculation is inferred from the number of objects. Hence optimizing the time complexity of the intersection calculation would optimize the time complexity of ray tracing — at least with respect to the number of objects.

9.2 Acceleration of intersection calculations

Let us use the notation $Q(n)$ for the time complexity ("query time") of the routine **Intersect**(r), where n is the number of objects. The brute force approach, which tests each object one by one, requires a query time proportional to n, that is $Q(n) = O(n)$. It is not necessary, however, to test each object for each ray. An object lying "behind" the origin of the ray, for example, will definitely not be intersected by it. But in order to be able to exploit such situations for saving computation for the queries, we must have in store some preliminary information about the spatial relations of objects, because if we do not have such information in advance, all the objects will have to be checked — we can never know whether the closest one intersected is the one that we have not yet checked. The required preprocessing will need computation, and its time complexity, say $P(n)$, will appear. The question is whether $Q(n)$ can be reduced without having to pay too much in $P(n)$.

Working on intuition, we can presume that the best achievable (worst-case) time complexity of the ray query is $Q(n) = O(\log n)$, as it is demonstrated by the following argument. The query can give us $n + 1$ "combinatorially" different answers: the ray either intersects one of the n objects or does not intersect any of them. Let us consider a weaker version of our original query: we do not have to calculate the intersection point exactly, but we only have to report the index of the intersected object (calculating the intersection would require only a constant amount of time if this index

is known). A computer program can be regarded as a numbered list of instructions. The computation for a given input can be characterized by the sequence $i_1, i_2, \ldots, i_m$ of numbers corresponding to the instructions that the program executed, where m is the total number of steps. An instruction can be of one of two types: it either takes the form $X \leftarrow f(X)$, where X is the set of variables and f is an algebraic function, or else it takes the form "IF $f(X) \circ 0$ THEN GOTO i_{yes} ELSE GOTO i_{no}", where $\circ$ is one of the binary relations $=, <, >, \leq, \geq$. The first is called a *decision* instruction; the former is called a *calculational* instruction. Computational instructions do not affect the sequence $i_1, i_2, \ldots, i_m$ directly, that is, if i_j is a calculational instruction, then i_{j+1} is always the same. The sequence is changed directly by the decision instructions: the next one is either i_{yes} or i_{no}. Thus, the computation can be characterized by the sequence $i_1^D, i_2^D, \ldots, i_d^D$ of decision instructions, where d is the number of decisions executed. Since there are two possible further instructions (i_{yes} and i_{no}) for each decision, all the possible sequences can be represented by a binary tree, the root of which represents the first decision instruction, the internal nodes represent intermediate decisions and the leaves correspond to terminations. This model is known as the **algebraic decision tree model** of computation. Since different leaves correspond to different answers, and there are $n+1$ of them, the length $d_{\max}$ of the longest path from the root to any leaf cannot be smaller than the depth of a balanced binary tree with $n+1$ leaves, that is $d_{\max} = \Omega(\log n)$.

The problem of intersection has been studied within the framework of computational geometry, a field of mathematics. It is called the **ray shooting problem** by computational geometers and is formulated as "given n objects in 3D-space, with preprocessing allowed, report the closest object intersected by any given query ray". Mark de Berg [dB92] has recently developed efficient ray shooting algorithms. He considered the problem for different types of objects (arbitrary and axis parallel polyhedra, triangles with angles greater than some given value, etc.) and different types of rays (rays with fixed origin or direction, arbitrary rays). His most general algorithm can shoot arbitrary rays into a set of arbitrary polyhedra with n edges altogether, with a query time of $O(\log n)$ and preprocessing time and storage of $O(n^{4+\varepsilon})$, where ε is a positive constant that can be made as small as desired. The question of whether the preprocessing and storage complexity are optimal is an open problem. Unfortunately, the complexity

of the preprocessing and storage makes the algorithm not too attractive for practical use.

There are a number of techniques, however, developed for accelerating intersection queries which are suitable for practical use. We can consider them as **heuristic methods** for two reasons. The first is that their approach is not based on complexity considerations, that is, the goal is not a worst-case optimization, but rather to achieve a speed-up for the majority of situations. The second reason is that these algorithms really do not reduce the query time for the worst case, that is $Q(n) = O(n)$. The achievement is that average-case analyses show that they are better than that. We will overview a few of them in the following subsections.

9.2.1 Regular partitioning of object space

Object coherence implies that if a spatial point p is contained by a given object (objects), then other spatial points close enough to p are probably contained by the same object(s). On the other hand, the number of objects intersecting a neighborhood δp of p is small compared with the total number of objects, if the volume of δp is small enough. It gives the following idea for accelerating ray queries. Partition the object space into disjoint cells $C_1, C_2, \ldots, C_m$, and make a list L_i for each cell C_i containing references to objects having non-empty intersection with the cell. If a ray is to be tested, then the cells along its path must be scanned in order until an intersection with an object is found:

```
Intersect(r)
    for each cell C_k along r in order do
        if r intersects at least one object on list L_k then
            q = the closest intersection point;
            return q;
        endif
    endfor
    return null;
end
```

Perhaps the simplest realization of this idea is that the set of cells, $C_1, C_2, \ldots, C_m$, consists of congruent axis parallel cubes, forming a regular

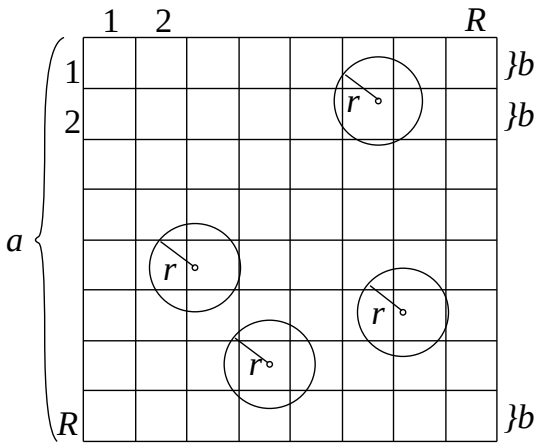

Figure 9.2: Regular partitioning of object space

spatial grid. The outer cycle of the above routine can then be implemented by an incremental line drawing algorithm; Fujimoto *et al.* [FTK86], for instance, used a 3D version of DDA (digital differential analyzer) for this task. If the resolution of the grid is the same, say R, in each of the three spatial directions, then $m = R^3$. The number of cells, k, intersected by a given ray is bounded by:

$$k \leq 1 + 7(R - 1) \tag{9.1}$$

where equality holds for a ray going diagonally (from one corner to the opposite one) through the "big cube", which is the union of the small cells. Thus:

$$k = O(R) = O(\sqrt[3]{m}). \tag{9.2}$$

If we set $m = O(n)$, where n is the number of objects, and the objects are so nicely distributed that the length of the lists L_i remains under a constant value ($|L_i| = O(1)$), then the query time $Q(n)$ can be as low as $O(\sqrt[3]{n})$. In fact, if we allow the objects to be only spheres with a fixed radius r, and assume that their centers are uniformly distributed in the interior of a cube of width a, then we can prove that the expected complexity of the query time can be reduced to the above value by choosing the resolution R properly, as will be shown by the following **stochastic analysis**. One more

assumption will obviously be needed: r must be small compared with a. It will be considered by examining the limiting case $a \to \infty$ with r fixed and n proportional to a^3. The reason for choosing spheres as the objects is that spheres are relatively easy to handle mathematically.

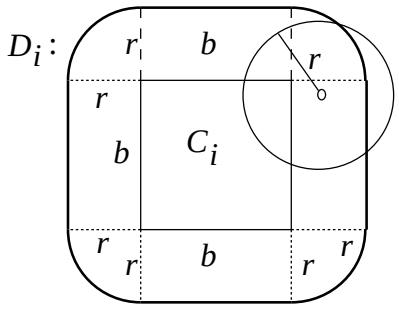

Figure 9.3: Center of spheres intersecting a cell

If points $p_1, \ldots, p_n$ are independently and **uniformly distributed** in the interior of a set X, then the probability of the event that $p_i \in Y \subseteq X$ is:

$$\Pr\{p_i \in Y\} = \frac{|Y|}{|X|} \tag{9.3}$$

where $|\cdot|$ denotes volume. Let X be a cube of width a, and the resolution of the grid of cells be R in all three spatial directions. The cells $C_1, C_2, \ldots, C_m$ will be congruent cubes of width $b = a/R$ and their number is $m = R^3$, as shown in figure 9.2. A sphere will appear on the list L_i corresponding to cell C_i if it intersects the cell. The condition of this is that the center of the sphere falls into a rounded cube shaped region D_i around the cell C_i, as shown in figure 9.3. Its volume is:

$$|D_i| = b^3 + 6b^2r + 3br^2\pi + \frac{4r^3\pi}{3}. \tag{9.4}$$

The probability of the event that a list L_i will contain exactly k elements — exploiting the assumption of uniform distribution — is:

$$\begin{aligned} \Pr\{|L_i| = k\} &= \binom{n}{k} \Pr\{p_1, \ldots, p_k \in D_i \ \wedge \ p_{k+1}, \ldots, p_n \notin D_i\} \\ &= \binom{n}{k} \left(\frac{|D_i \cap X|}{|X|}\right)^k \left(\frac{|X \setminus D_i|}{|X|}\right)^{n-k}. \end{aligned} \tag{9.5}$$

If D_i is completely contained by X, then:

$$\Pr\{|L_i| = k\} = \binom{n}{k} \left(\frac{|D_i|}{|X|}\right)^k \left(1 - \frac{|D_i|}{|X|}\right)^{n-k}. \tag{9.6}$$

Let us consider the limiting behavior of this probability for $n \to \infty$ by setting $a \to \infty$ ($|X| \to \infty$) and forcing $n/|X| \to \rho$, where ρ is a positive real number characterizing the density of the spheres. Note that our uniform distribution has been extended to a **Poisson point process** of intensity ρ. Taking the above limits into consideration, one can derive the following limiting behavior of the desired probability:

$$\Pr'\{|L_i| = k\} = \lim_{\substack{|X|\to\infty \\ n/|X|\to\rho}} \Pr\{|L_i| = k\} = \frac{(\rho|D_i|)^k}{k!} e^{-\rho|D_i|}. \tag{9.7}$$

Note that the rightmost expression characterizes a Poisson distribution with parameter $\rho|D_i|$, as the limit value of the binomial distribution on the right-hand side of expression 9.6 for $n \to \infty$ and $n/|X| \to \rho$. The expected length of list L_i is then given by the following formula:

$$\mathrm{E}[|L_i|] = \sum_{k=1}^{\infty} k \cdot \Pr'\{|L_i| = k\} = \rho|D_i|. \tag{9.8}$$

Substituting expression 9.4 of the volume $|D_i|$, and bearing in mind that $n/|X| \to \rho$ and $|X| = a^3 = R^3 b^3$ hence $b^3 \to n/\rho R^3$, we can get:

$$\mathrm{E}[|L_i|] = \frac{n}{R^3} + 6\rho^{1/3} r \frac{n^{2/3}}{R^2} + 3\rho^{2/3} r^2 \pi \frac{n^{1/3}}{R} + \rho \frac{4r^3\pi}{3} \quad (1 \le i \le R^3). \tag{9.9}$$

for the expected asymptotic behavior of the list length. This quantity can be kept independent of n (it can be $O(1)$) if is R chosen properly. The last term tends to be constant, independently of R. The first term of the sum requires $R^3 = \Omega(n)$, at least. The two middle terms will also converge to a constant with this choice, since then $R^2 = \Omega(n^{2/3})$ and $R = \Omega(n^{1/3})$. The conclusion is the following: if our object space X is partitioned into congruent cubes with an equal resolution R along all three spatial directions, and R is kept $R = \Omega(\sqrt[3]{n})$, then the expected number of spheres intersecting any of the cells will be $O(1)$, independent of n in the asymptotic sense. This implies furthermore (cf. expression 9.1) that the number of cells along the

path of an arbitrary ray is also bounded by $O(\sqrt[3]{n})$. The actual choice for R can modify the constant factor hidden by the "big O", but the last term of the sum does not allow us to make it arbitrarily small. The value $R = \lceil \sqrt[3]{n} \rceil$ seems to be appropriate in practice ($\lceil \cdot \rceil$ denotes "ceiling", that is the smallest integer above or equal). We can conclude that the expected query time and expected storage requirements of the method are:

$$\mathrm{E}[Q(n)] = O(R(n)) = O(\sqrt[3]{n}) \quad \text{and} \quad \mathrm{E}[S(n)] = O(n) \tag{9.10}$$

respectively, for the examined distribution of sphere centers. The behavior

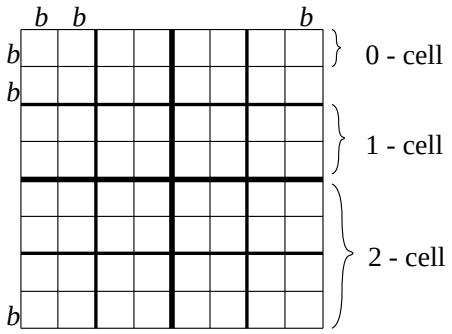

Figure 9.4: j-cells

of the preprocessing time $P(n)$ depends on the efficiency of the algorithm used for finding the intersecting objects (spheres) for the individual cells. Let us consider the 8 neighboring cells of width b around their common vertex. Their union is a cube of width $2b$. An object can intersect any of the 8 cells only if it intersects the cube of width $2b$. Furthermore, considering the union of 8 such cubes, which is a cube of width $4b$, a similar statement can be made, etc. In order to exploit this idea, let us choose $R = 2^K$ with $K = \lceil (\log_2 n)/3 \rceil$, in order to satisfy the requirement $R = \Omega(\sqrt[3]{n})$. The term j-cell will be used to denote the cubes of width $2^j b$ containing 2^{3j} cells of width b, as shown in figure 9.4. Thus, the smallest cells C_i become 0-cells, denoted by $C_i^{(0)}$ ($1 \leq i \leq 2^{3K}$), and the object space X itself will appear as the sole K-cell. The preprocessing algorithm will progressively refine the partitioning of the object space, which will consist of one K-cell in the first

step, 8 $(K-1)$-cells in the second step, and $2^{3K} = O(n)$ 0-cells in the last step.

The algorithm is best shown as a recursive algorithm, which preprocesses a list $L^{(j)}$ of objects with respect to a j-cell $C_i^{(j)}$. Provided that the object scene containing the objects $o_1, \ldots, o_n$ is enclosed by a cube (or rectangular box) X, it can be preprocessed by invoking a subroutine call **Preprocess**$(\{o_1, \ldots, o_n\}^{(K)}, X^{(K)})$ (with $K = \lceil (\log_2 n)/3 \rceil$), where the subroutine **Preprocess** is the following:

```
Preprocess(L^(j), C_i^(j))
    if j = 0 then L_i = L^(j); return ;
    for each subcell C_k^(j-1) (1 ≤ k ≤ 8) contained by C_i^(j) do
        L^(j-1) = {};
        for each object o on list L^(j) do
            if o intersects C_k^(j-1) then
              add o to L^(j-1);
            endif
        endfor
        Preprocess(L^(j-1), C_k^(j-1));
    endfor
end
```

The algorithm can be speeded up by using the trick that if the input list corresponding to a j-cell becomes empty ($|L^{(j)}| = 0$) at some stage, then we do not process the "child" cells further but return instead. The maximal depth of recurrence is K, because j is decremented by 1 at each recursive call, hence we can distinguish between $K+1$ different levels of execution. Let the level of executing the uppermost call be K, and generally, the level of execution be j if the superscript $^{(j)}$ appears in the input arguments. The execution time $T = P(n)$ of the preprocessing can be taken as the sum $T = T_0 + T_1 + \ldots + T_K$, where the time T_j is spent at level j of execution. The routine is executed only once at level K, 8 times at level $K-1$, and generally:

$$N_j = 2^{3(K-j)} \quad K \geq j \geq 0 \tag{9.11}$$

times at level j. The time taken for a given instance of execution at level j is proportional to the actual length of the list $L^{(j)}$ to be processed. Its

expected length is equal to the expected number of objects intersecting the corresponding j-cell $C_i^{(j)}$. Its value is:

$$\mathrm{E}[|L^{(j)}|] = \rho |D_i^{(j)}| \tag{9.12}$$

where $D_i^{(j)}$ is the rounded cube shaped region around the j-cell $C_i^{(j)}$, very similar to that shown in figure 9.3, with the difference that the side of the "base cube" is $2^{j-K}a$. Its volume is given by the formula:

$$|D_i^{(j)}| = 2^{3(j-K)}a^3 + 6 \cdot 2^{2(j-K)}a^2 r + 3 \cdot 2^{j-K} a r^2 \pi + \frac{4r^3\pi}{3} \tag{9.13}$$

which is the same for each j-cell. Thus, the total time T_j spent at level j of execution is proportional to:

$$N_j \rho |D_i^{(j)}| = \rho a^3 + 6 \cdot 2^{K-j} \rho a^2 r + 3 \cdot 2^{2(K-j)} \rho a r^2 \pi + 2^{3(K-j)} \rho \frac{4r^3\pi}{3}. \tag{9.14}$$

Let us sum these values for $1 \leq j \leq K-1$, taking the following identity into consideration:

$$2^i + 2^{i \cdot 2} + \ldots + 2^{i \cdot (K-1)} = \frac{2^{i \cdot K} - 2^i}{2^i - 1} \quad (i \geq 1), \tag{9.15}$$

where i refers to the position of the terms on the right-hand side of expression 9.14 ($i = 1$ for the second term, $i = 2$ for the third etc.). Thus the value $T_1 + \ldots + T_{K-1}$ is proportional to:

$$(K-1)\rho a^3 + 6 \cdot \frac{2^K - 2}{1} \rho a^2 r + 3 \cdot \frac{2^{2K} - 4}{3} \rho a r^2 \pi + \frac{2^{3K} - 8}{7} \rho \frac{4r^3\pi}{3}. \tag{9.16}$$

Since $K = O(\log n)$ and $a^3 \to n/\rho$, the first term is in the order of $O(n \log n)$, and since $n^{i/3} \leq 2^{iK} < 2n^{i/3}$, the rest of the terms are only of $O(n)$ (actually, this is in connection with the fact that the center of the majority of the spheres intersecting a cube lies also in the cube as the width of the cube increases). Finally, it can easily be seen that the times T_0 and T_K are both proportional to n, hence the expected preprocessing time of the method is:

$$\mathrm{E}[P(n)] = O(n \log n) \tag{9.17}$$

for the examined Poisson distribution of sphere centers.

We intended only to demonstrate here how a stochastic average case analysis can be performed. Although the algorithm examined here is relatively simple compared to those coming in the following subsections, performing the analysis was rather complicated. This is the reason why we will not undertake such analyses for the other algorithms (they are to appear in [Már94]).

9.2.2 Adaptive partitioning of object space

The regular cell grid is very attractive for the task of object space subdivision, because it is simple, and the problem of enumerating the cells along the path of a ray is easy to implement by means of a 3D incremental line generator. The cells are of the same size, wherever they are. Note that we are solely interested in finding the intersection point between a ray and the *surface* of the closest object. The number of cells falling totally into the interior of an object (or outside all the objects) can be very large, but the individual cells do not yield that much information: each of them tells us that there is no ray-surface intersection inside. Thus, the union of such cells carries the same information as any of them do individually — it is not worth storing them separately. The notion and techniques used in the previous subsection form a good basis for showing how this idea can be exploited.

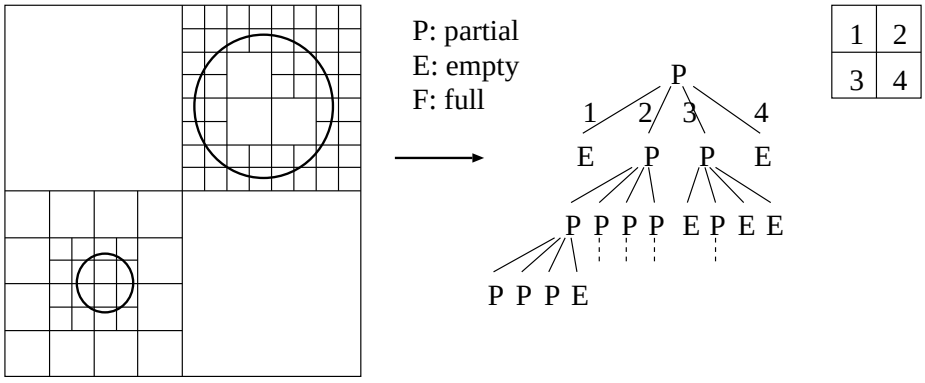

Figure 9.5: The octree structure for space partitioning

If our object space is enclosed by a cube of width a, then the resolution of subdivision, R, means that the object space was subdivided into congruent cubes of width $b = a/R$ in the previous subsection. We should remind the reader that a cube of width $2^j b$ is called a j-cell, and that a j-cell is the union of exactly 2^{3j} 0-cells. Let us distinguish between three types of j-cell: an *empty* cell has no intersection with any object, a *full* cell is completely contained in one or more objects, and a *partial* cell contains a part of the surface of at least one object. If a j-cell is empty or full, then we do not have to divide it further into $(j-1)$-cells, because the child cells would also be empty or full, respectively. We subdivide only partial cells. Such an uneven subdivision can be represented by an **octree** (octal tree) structure, each node of which has either 8 or no children. The two-dimensional analogue of the octree (the quadtree) is shown in figure 9.5. A node corresponds to a j-cell in general, and has 8 children ($(j-1)$-cells) if the j-cell is partial, or has no children if it is empty or full. If we use it for ray-surface intersection calculations, then only partial cells need have references to objects, and only to those objects whose surface intersects the cell.

The preprocessing routine that builds this structure is similar to the one shown in the previous subsection but with the above mentioned differences. If the objects of the scene X are $o_1, \ldots, o_n$, then the forthcoming algorithm must be invoked in the following form: **Preprocess**$(\{o_1, \ldots, o_n\}^{(K)}, X^{(K)})$, where K denotes the allowed number of subdivision steps at the current recurrence level. The initial value $K = \lceil (\log_2 n)/3 \rceil$ is proper again, since our subdivision can become a regular grid in the worst case. The algorithm will return the octree structure corresponding to X. The notation $L(C_i^{(j)})$ in the algorithm stands for the object reference list corresponding to the j-cell $C_i^{(j)}$ (if it is partial), while $R_k(C_i^{(j)})$ $(1 \leq k \leq 8)$ stands for the reference to its kth child (*null* denotes no child).

The algorithm is then the following:

```
Preprocess(L^(j), C_i^(j))
    if j = 0 then                                   // bottom of recurrence
        R_1(C_i^(j)) = ... = R_8(C_i^(j)) = null;
        L(C_i^(j)) = L^(j); return C_i^(j);
    endif
    for each subcell C_k^(j-1) (1 ≤ k ≤ 8) contained by C_i^(j) do
        L^(j-1) = {};
        for each object o on list L^(j) do
            if surface of o intersects C_k^(j-1) then add o to L^(j-1);
        endfor
        if L^(j-1) = {} then                        // empty or full
            R_k(C_i^(j)) = null;
        else                                        // partial
            R_k(C_i^(j)) = Preprocess(L^(j-1), C_k^(j-1));
        endif
    endfor
    return C_i^(j);
end
```

The method saves storage by its adaptive operation, but raises a new problem, namely the enumeration of cells along the path of a ray during ray tracing.

The problem of visiting all the cells along the path of a ray is known as **voxel walking** (voxel stands for "volume cell" such as pixel is "picture cell"). The solution is almost facile if the subdivision is a regular grid, but what can we do with our octree? The method commonly used in practice is based on a **generate-and-test** approach, originally proposed by Glassner [Gla84]. The first cell the ray visits is the cell containing the origin of the ray. In general, if a point p is given, then the cell containing it can be found by recursively traversing the octree structure from its root down to the leaf containing the point.

This is what the following routine does:

Classify(p, $C_i^{(j)}$)
  if $C_i^{(j)}$ is a leaf ($R_k(C_i^{(j)})$=*null*) **then return** $C_i^{(j)}$;
  for each child $R_k(C_i^{(j)})$ ($1 \leq k \leq 8$) **do**
    if subcell $R_k(C_i^{(j)})$ contains p **then**
      return Classify(p, $R_k(C_i^{(j)})$);
    endif
  endfor
  return *null*;
end

The result of the function call **Classify**(p, $X^{(K)}$) is the cell containing a point $p \in X$. It is *null* if p falls outside the object space X. The worst case time required by the classification of a point will be proportional to the depth of the octree, which is $K = \lceil (\log_2 n)/3 \rceil$, as suggested earlier. Once the cell containing the origin of the ray is known, the next cell visited can be determined by first generating a point q which definitely falls in the interior of the next cell, and then by testing to find which cell contains q. Thus, the intersection algorithm will appear as follows (the problem of generating a point falling into the next cell will be solved afterwards):

Intersect(r)
  $o_{\min}$ = *null*; // $o_{\min}$: *closest intersected object*
  p = origin of ray;
  C = **Classify**(p, $X^{(K)}$);
  while $C \neq$ *null* **do**
    for each object o on list $L(C)$ **do**
      if r intersects o closer than $o_{\min}$ **then** $o_{\min} = o$;
    endfor
    if $o_{\min} \neq$ *null* **then return** $o_{\min}$;
    q = a point falling into the next cell;
    C = **Classify**(q, $X^{(K)}$);
  endwhile
  return *null*;
end

There is only one step to work out, namely how to generate a point q which falls definitely into the neighbor cell. The point where the ray r exits the actual cell can easily be found by intersecting it with the six faces of the cell. Without loss of generality, we can assume that the direction vector of r has nonnegative x, y and z components, and its exit point e either falls in the interior of a face with a normal vector incident with the x coordinate axis, or is located on an edge of direction incident with the z axis, or is a vertex of the cell — all the other combinations can be handled similarly. A proper q can be calculated by the following vector sums, where $\vec{x}, \vec{y}$ and $\vec{z}$ represent the (unit) direction vectors of the coordinate axes, and $b = a2^{-K}$ is the side of the smallest possible cell, and the subscripts of q distinguish between the three above cases in order:

$$q_1 = e + \frac{b}{2}\vec{x}, \quad q_2 = e + \frac{b}{2}\vec{x} + \frac{b}{2}\vec{y} \quad \text{and} \quad q_3 = e + \frac{b}{2}\vec{x} + \frac{b}{2}\vec{y} + \frac{b}{2}\vec{z}. \quad (9.18)$$

For the suggested value of the subdivision parameter K, the expected query time will be $\mathrm{E}[Q(n)] = O(\sqrt[3]{n}\log n)$ per ray if we take into consideration that the maximum number of cells a ray intersects is proportional to $R = 2^K$ (cf. expression 9.1), and the maximum time we need for stepping into the next cell is proportional to K.

9.2.3 Partitioning of ray space

A ray can be represented by five coordinates, $x, y, z, \vartheta, \varphi$ for instance, the first three of which give the origin of the ray in the 3D space, and the last two define the direction vector of the ray as a point on the 2D surface of the unit sphere (in polar coordinates). Thus, we can say that a ray r can be considered as a point of the 5D ray-space $\Re^5 = E^3 \times O^2$, where the first space is a Euclidean space, the second is a spherical one, and their Cartesian product is a cylinder-like space. If our object space, on the other hand, contains the objects $o_1, \ldots, o_n$, then for each point (ray) $r \in \Re^5$, there is exactly one $i(r) \in \{0, 1, \ldots, n\}$ assigned, where $i(r) = 0$ if r intersects no object, and $i(r) = j$ if r intersects object o_j first. We can notice furthermore that the set of rays intersecting a given object o_j — that is the regions $R(j) = \{r \mid i(r) = j\}$ — form connected subsets of $\Re^5$, and $R(0) \cup R(1) \cup \ldots \cup R(n) = \Re^5$, that is, the $n + 1$ regions form a subdivision of the ray space. This leads us to hope that we can construct a ray-object

intersection algorithm with a good (theoretically optimal $O(\log n)$) query time based on the following **locus approach**: first we build the above mentioned subdivision of the ray space in a preprocessing step, and then, whenever a ray r is to be tested, we *classify* it into one of the $n+1$ regions, and if the region containing r is $R(j)$, then the intersection point will be on the surface of o_j. The only problem is that this subdivision is so difficult to calculate that nobody has even tried it yet. Approximations, however, can be carried out, which Arvo and Kirk in fact did [AK87] when they worked out their method called **ray classification**. We shall outline their main ideas here.

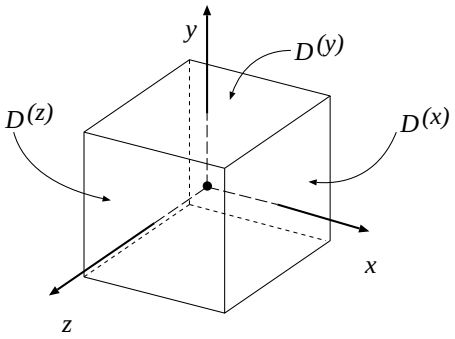

Figure 9.6: The direction cube

A crucial problem is that the ray space $\Re^5$ is not Euclidean (but cylinder-like), hence it is rather difficult to treat computationally. It is very inconvenient namely, that the points representing the direction of the rays are located on the surface of a sphere. We can, however, use a more suitable representation of direction vectors which is not "curved". Directions will be represented as points on the surface of the unit cube, instead of the unit sphere, as shown in figure 9.6. There are discontinuities along the edges of the cube, so the direction space will be considered as a collection $D^{(x)}, D^{(-x)}, D^{(y)}, D^{(-y)}, D^{(z)}, D^{(-z)}$ of six spaces (faces of the unit cube), each containing the directions with the main component (the one with the greatest absolute value) being in the same coordinate direction $(x, -x, y, -y, z, -z)$. If the object scene can be enclosed by a cube E — containing the eye as well — then any ray occurring during ray tracing must

fall within one of the sets in the collection:

$$H = \{E \times D^{(x)}, E \times D^{(-x)}, E \times D^{(y)}, E \times D^{(-y)}, E \times D^{(z)}, E \times D^{(-z)}\}. \quad (9.19)$$

Each of the above six sets is a 5D hypercube. Let us refer to this collection H as the bounding hyperset of our object scene.

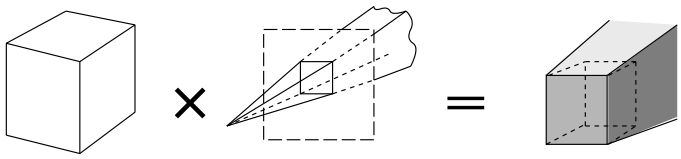

Figure 9.7: Beams of rays in 3D space

The hyperset H will be recursively subdivided into cells $H^{(1)}$, ..., $H^{(m)}$ (each being an axis parallel hypercube), and a *candidate list* $L(H^{(i)})$ will be associated with each cell $H^{(i)}$ containing references to objects that are intersected by any ray $r \in H^{(i)}$. Each such hypercube $H^{(i)}$ is a collection of rays with their origin in a 3D rectangular box and their direction falling into an axis parallel 2D rectangle embedded in the 3D space. These rays form an unbounded polyhedral volume in the 3D space, called a **beam**, as shown in figure 9.7. An object appears on the list associated with the 5D hypercube if and only if it intersects the 3D beam corresponding to the hypercube. At each step of subdivision a cell will be divided into two halves along one of the five directions. If we normalize the object scene so that the enclosing cube E becomes a unit cube, then we can decide to subdivide a 5D cell along one of its longest edges. Such a subdivision can be represented by a binary tree, the root of which corresponds to H itself, the two children correspond to the two halves of H, etc. In order to save computation, the subdivision will not be built completely by a separate preprocessing step, but rather the hierarchy will be constructed adaptively during ray tracing by **lazy evaluation**. Arvo and Kirk suggested [AK87] terminating this subdivision when either the candidate list or the hypercube fall below a fixed size threshold. The heuristic reasoning is that "a small candidate set indicates that we have achieved the goal of making the associated rays inexpensive to intersect with the environment", while "the hypercube size constraint is imposed to allow the cost of creating a candidate set to be

amortized over many rays" (cited from [AK87]). The intersection algorithm then appears as follows, where $R_l(H')$ and $R_r(H')$ denote the left and right children of cell H' in the tree structure, $n_{\min}$ is the number under which the length of an object list is considered to be as "small enough", and $w_{\min}$ denotes the minimal width of a cell (width of cells is taken as the smallest width along the five axes).

```
Intersect(r)
    H' = Classify(r, H);
    while |L(H')| > n_min and |H'| > w_min do
            H'_l, H'_r = two halves of H'; L(H'_l) = {}; L(H'_r) = {};
            for each object o on list L(H') do
                if o intersects the beam of H'_l then add o to L(H'_l);
                if o intersects the beam of H'_r then add o to L(H'_r);
            endfor
            R_l(H') = H'_l; R_r(H') = H'_r; H' = Classify(r, H');
    endwhile
    o_min = null;                          // o_min: closest intersected object
    for each object o on list L(H') do
            if r intersects o closer than o_min then o_min = o; endif
    endfor
    return o_min;
end
```

The routine **Classify**(r, H') called from the algorithm finds the smallest 5D hypercube containing the ray r by performing a binary search in the tree with root at H'.

9.2.4 Ray coherence theorems

Two rays with the same origin and slightly differing directions probably intersect the same object, or more generally, if two rays are close to each other in the 5D ray space then they probably intersect the same object. This is yet another guise of object coherence, and we refer to it as **ray coherence**. Closeness here means that both the origins and the directions are close. The ray classification method described in the previous section used a 5D subdivision along all the five ray parameters, the first three of

which represented the origin of the ray in the 3D object space, hence every ray originating in the object scene is contained in the structure, even those that have their origins neither on the surface of an object nor in the eye position. These rays will definitely not occur during ray tracing. We will define *equivalence classes* of rays in an alternative way: two rays will be considered to be equivalent if their origins are on the surface of the same object and their directions fall in the same class of a given partition of the direction space. This is the main idea behind the method of Ohta and Maekawa [OM87]. We will describe it here in some detail.

Let the object scene consist of n objects, including those that we would like to render, the lightsources and the eye. Some, say m, of these n objects are considered to be *ray origins*, these are the eye, the lightsources and the reflective/refractive objects. The direction space is partitioned into d number of classes. This subdivision can be performed by subdividing each face of the direction cube (figure 9.6) into small squares at the desired resolution. The preprocessing step will build a two-dimensional array $O[1 \ldots m, 1 \ldots d]$, containing lists of references to objects. An object o_k will appear on the list at $O[i, j]$ if there exists a ray intersecting o_k with its origin on o_i and direction in the jth direction class. Note that the cells of the array O correspond to the "equivalence classes" of rays defined in the previous paragraph. If this array is computed, then the intersection algorithm becomes very simple:

```
Intersect(r)
    i = index of object where r originates;
    j = index of direction class containing the direction of r;
    o_min = null;                          // o_min: closest intersected object
    for each object o on list O[i, j] do
        if r intersects o closer than o_min then
            o_min = o;
        endif
    endfor
    return o_min;
end
```

The computation of the array O is based on the following geometric considerations. We are given two objects, o_1 and o_2. Let us define a set $V(o_1, o_2)$ of directions, so that V contains a given direction δ if and only if there exists

a ray of direction δ with its origin on o_1 and intersecting o_2, that is:

$$V(o_1, o_2) = \{\delta \mid \exists r : \mathrm{org}(r) \in o_1 \ \wedge \ \mathrm{dir}(r) = \delta \ \wedge \ r \cap o_2 \neq \emptyset\} \qquad (9.20)$$

where $\mathrm{org}(r)$ and $\mathrm{dir}(r)$ denote the origin and direction of ray r, respectively. We will call the set $V(o_1, o_2)$ the **visibility set** of o_2 with respect to o_1 (in this order). If we are able to calculate the visibility set $V(o_i, o_k)$ for a pair of objects o_i and o_k, then we have to add o_k to the list of those cells in the row $O[i, 1 \ldots d]$ of our two-dimensional array which have non-empty intersection with $V(o_i, o_k)$. Thus, the preprocessing algorithm can be the following:

Preprocess$(o_1, \ldots, o_n)$
    initialize each list $O[i, j]$ to $\{\}$;
    for each ray origin o_i $(1 \leq i \leq m)$ **do**
        for each object o_k $(1 \leq k \leq n)$ **do**
            compute the visibility set $V(o_i, o_k)$;
            for each direction class Δ_j with $\Delta_j \cap V(o_i, o_k) \neq \emptyset$ **do**
                add o_k to list $O[i, j]$;
            endfor
        endfor
    endfor
end

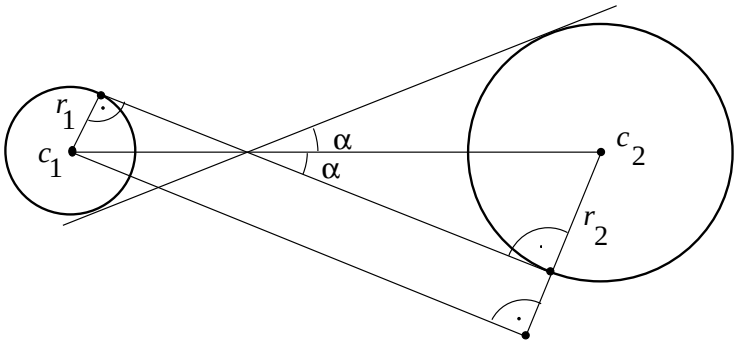

Figure 9.8: Visibility set of two spheres

The problem is that the exact visibility sets can be computed only for a narrow range of objects. These sets are subsets of the surface of the unit

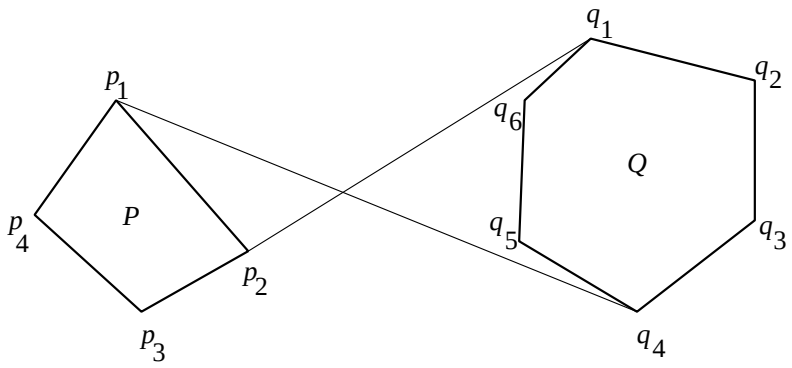

Figure 9.9: Visibility set of two convex hulls

sphere — or alternatively the direction cube. Ohta and Maekawa [OM87] gave the formula for a pair of spheres, and a pair of convex polyhedra. If S_1 and S_2 are spheres with centers c_1, c_2 and radii r_1, r_2, respectively, then $V(S_1, S_2)$ will be a spherical circle. Its center is at the spherical point corresponding to the direction $\vec{c_1c_2}$ and its (spherical) radius is given by the expression $\arcsin\{(r_1 + r_2)/|c_1 - c_2|\}$, as illustrated in figure 9.8. If P and Q are convex polyhedra with vertices $p_1, \ldots, p_n$ and $q_1, \ldots, q_m$, respectively, then $V(P, Q)$ will be the spherical convex hull of $n \cdot m$ spherical points corresponding to the directions $\vec{p_1q_1}, \ldots, \vec{p_1q_m}, \ldots, \vec{p_nq_1}, \ldots, \vec{p_nq_m}$ (see figure 9.9). It can be shown [HMSK92] that for a mixed pair of a convex polyhedron P with vertices $p_1, \ldots, p_n$ and a sphere S with center c and radius r, $V(P, S)$ is the spherical convex hull of n circles with centers at $\vec{p_1c}, \ldots, \vec{p_nc}$ and radii $\arcsin\{r/|p_1 - c|\}, \ldots, \arcsin\{r/|p_n - c|\}$. In fact, these circles are nothing else than the visibility sets $V(p_1, S), \ldots, V(p_n, S)$, corresponding to the vertices of P. This gives the idea of a generalization of the above results in the following way [MRSK92]: If A and B are convex hulls of the sets $A_1, \ldots, A_n$ and $B_1, \ldots, B_m$, respectively, then $V(A, B)$ will be the spherical convex hull of the visibility sets $V(A_1B_1), \ldots, V(A_1B_m), \ldots, V(A_nB_1), \ldots, V(A_nB_m)$. Note that disjointness for the pairs of objects was assumed so far, because if the objects intersect then the visibility set is the whole sphere surface (direction space).

Unfortunately, exact expression of visibility sets is not known for further types of objects. We can use approximations, however. Any object can be

enclosed by a large enough sphere, or a convex polyhedron, or a convex hull of some sets. The simpler the enclosing shell is, the easier the calculations are, but the greater the difference is between the real and the computed visibility set. We always have to find a trade-off between accuracy and computation time.

9.3 Distributed ray tracing

Recursive ray tracing is a very elegant method for simulating phenomena such as shadows, mirror-like reflections, and refractions. The simplifications in the illumination model — point-like lightsources and point-sampling (infinitely narrow light rays) — assumed so far, however, cause sharp shadows, reflections and refractions, although these phenomena usually occur in a **blurred** form in reality.

Perhaps the most elegant method among all the proposed approaches to handle the above mentioned blurred (fuzzy) phenomena is the so-called **distributed ray tracing** due to Cook *et al.* [CPC84]. The main advantage of the method is that phenomena like motion blur, depth of field, penumbras, translucency and fuzzy reflections are handled in an easy and somewhat unified way with no additional computational cost beyond those required by spatially oversampled ray tracing. The basic ideas can be summarized as follows. Ray tracing is a kind of **point sampling** and, as such, is a subject to aliasing artifacts (see chapter 11 on Sampling and Quantization Artifacts). The usual way of reducing these artifacts is the use of some post-filtering technique on an oversampled picture (that is, more image rays are generated than the actual number of pixels).

The key idea is, that oversampling can be made not only in space but also in the time (motion sampling), on the area of the camera lens or the entire shading function. Furthermore, "not everything must be sampled everywhere" but rather the rays can be *distributed*. In the case of motion sampling, for example, instead of taking multiple time samples at every spatial location, the rays are distributed in time so that rays at different spatial locations trace the object scene at different instants of time.

Distributing the rays offers the following benefits at little additional cost:

- Distributing reflected rays according to the specular distribution function produces gloss (fuzzy reflection).
- Distributing transmitted rays produces blurred transparency.
- Distributing shadow rays in the solid angle of the lightsources produces penumbras.
- Distributing rays on the area of the camera lens produces depth of field.
- Distributing rays in time produces motion blur.

Oversampled ray traced images are generated by emanating more than one ray through the individual pixels. The rays corresponding to a given pixel are usually given the same origin (the eye position) and different direction vectors, and because of the different direction vectors, the second and further generation rays will generally have different origins and directions, as well. This spatial oversampling is generalized by the idea of distributing the rays. Let us overview what distributing the rays means in concrete situations.

Fuzzy shading

We have seen in chapter 3 that the intensity I^{out} of the reflected light coming from a surface point towards the viewing position can be expressed by an integral of an illumination function $I^{\text{in}}(\vec{L})$ ($\vec{L}$ is the incidence direction vector) and a reflection function over the hemisphere about the surface point (cf. equation 3.30):

$$I_r^{\text{out}} = k_r \cdot I_r^{\text{in}} + \int^{2\pi} I^{\text{in}}(\vec{L}) \cdot \cos\phi_{\text{in}} \cdot R^*(\vec{L}, \vec{V})\, d\omega_{\text{in}} \tag{9.21}$$

where $\vec{V}$ is the viewing direction vector and the integration is taken over all the possible values of $\vec{L}$. The coherent reflection coefficient k_r is in fact a δ-function, that is, its value is non-zero only at the reflective inverse of the viewing direction $\vec{V}$. Sources of second or higher order reflections

are considered only from this single direction (the incoming intensity I_r^{in} is computed recursively). A similar equation can be given for the intensity of the refracted light:

$$I_t^{\text{out}} = k_t \cdot I_t^{\text{in}} + \int^{2\pi} I^{\text{in}}(\vec{L}) \cdot \cos\phi_{\text{in}} \cdot T^*(\vec{L}, \vec{V})\, d\omega_{\text{in}} \tag{9.22}$$

where the integration is taken over the hemisphere below the surface point (in the interior of the object), I_t^{in} is the intensity of the coherent refractive (transmissive) illumination and k_t is the coherent transmission coefficient (also a δ-function).

The integrals in the above expressions are usually replaced by finite sums according to the finite number of (usually) point-like or directional lightsources. The effects produced by finite extent lightsources can be considered by distributing more than one shadow ray over the solid angle of the visible portion of each lightsource. This technique can produce **penumbras**. Furthermore, second and higher order reflections need no longer be restricted to single directions but rather the reflection coefficient k_r can be treated as non-zero over the whole hemisphere and more than one rays can be distributed according to its function. This can model **gloss**. Finally, distributing the refracted rays in a similar manner can produce **blurred translucency**.

Depth of field

Note that the usual projection technique used in computer graphics in fact realizes a pinhole camera model with each object in sharp focus. It is an idealization, however, of a real camera, where the ratio of the focal length F and the diameter D of the lens is a finite positive number, the so-called **aperture** number a:

$$a = \frac{F}{D}. \tag{9.23}$$

The finite aperture causes the effect called **depth of field** which means that object points at a given distance appear in sharp focus on the image and other points beyond this distance or closer than that are confused, that is, they are mapped to finite extent patches instead of points.

It is known from geometric optics (see figure 9.10) that if the focal length of a lens is F and an object point is at a distance T from the lens, then

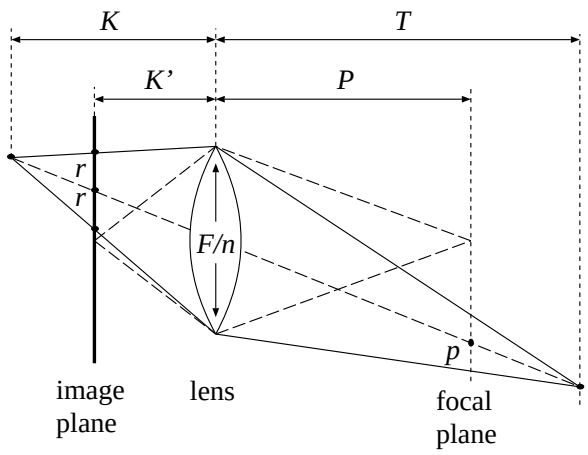

Figure 9.10: Geometry of lens

the corresponding image point will be in sharp focus on an image plane at a distance K behind the lens, where F, T and K satisfy the equation:

$$\frac{1}{F} = \frac{1}{K} + \frac{1}{T}. \tag{9.24}$$

If the image plane is not at the proper distance K behind the lens but at a distance K', as in figure 9.10, then the object point maps onto a circle of radius r:

$$r = \frac{1}{K} |K - K'| \frac{F}{n}. \tag{9.25}$$

This circle is called the **circle of confusion** corresponding to the given object point. It expresses that the color of the object point affects the color of not only a single pixel but all the pixels falling into the circle.

A given camera setting can be specified in the same way as in real life by the aperture number a and the *focal distance*, say P (see figure 9.10), which is the distance of those objects from the lens, which appear in sharp focus on the image (not to be confused with the focal length of the lens). The focal length F is handled as a constant. The plane at distance P from the lens is called the *focal plane*. Both the distance of the image plane from the lens and the diameter (D) of the lens can be calculated from these parameters using equations 9.24 and 9.23, respectively.

In depth of field calculations, the eye position is imagined to be in the center of the lens. First a ray is emanated from the pixel on the image plane through the eye position, as in usual ray tracing, and its color, say I_0 is computed. Let the point where this "traditional" ray intersects the focal plane be denoted by p. Then some further points are selected on the surface of the lens, and a ray is emanated from each of them through point p. Their colors, say $I_1, \ldots, I_m$, are also computed. The color of the pixel will be the average of the intensities $I_0, I_1, \ldots, I_m$. In fact, it approximates an integral over the lens area.

Motion blur

Real cameras have a finite *exposure time*, that is, the film is illuminated during a time interval of nonzero width. If some objects are in motion, then their image will be blurred on the picture, and the higher the speed of an object is, the longer is its trace on the image. Moreover, the trace of an object is translucent, that is, the objects behind it become partially visible. This effect is known as **motion blur**. This is yet another kind of integration, but now in time. Distributing the rays in time can easily be used for approximating (sampling) this integral. It means that the different rays corresponding to a given pixel will correspond to different time instants. The path of motion can be arbitrarily complex, the only requirement is the ability to calculate the position of any object at any time instant.

We have seen that distributed ray tracing is a unified approach to modeling realistic effects such as fuzzy shading, depth of field or motion blur. It approximates the analytic function describing the intensity of the image pixels at a higher level than usual ray tracing algorithms do. Generally this function involves several nested integrals: integrals of illumination functions multiplied by reflection or refraction functions over the reflection or transmission hemisphere, integrals over the surface of the lens, integrals over time, integrals over pixel area. This integral is so complicated that only approximation techniques can be used in practice. Distributing the rays is in fact a point sampling method performed on a multi-dimensional parameter space. In order to keep the computational time at an acceptably low level, the number of rays is not increased "orthogonally", that is, instead of adding more rays in each dimension, the existing rays are distributed with respect to this parameter space.