Chapter 6

VISIBILITY CALCULATIONS

In order to be able to calculate the color of a pixel we must know from where the light ray through the pixel comes. Of course, as a pixel has finite dimensions, there can be an infinite number of light rays going into the eye through it. In contrast to this fact, an individual color has to be assigned to each pixel, so it will be assumed, at least in this chapter, that each pixel has a specified point, for example its center, and only a single light ray through this point is to be considered. The origin of the ray — if any — is a point on the surface of an object. The main problem is finding this object. This is a geometric searching problem at discrete positions on the image plane. The problem of finding the visible surface points can be solved in one of two ways. Either the pixels can be taken first and then the objects for the individual pixels. In this case, for each pixel of the image, the object which can be seen in it at the special point is determined; the object which is closest to the eye will be selected from those falling onto the pixel point after projection. Alternatively, the objects can be examined before the pixels. Then for the whole scene the parts of the projected image of the objects which are visible on the screen are determined, and then the result is sampled according to the resolution of the raster image. The first approach can solve the visibility problem only at discrete points and the accuracy of the solution depends on the resolution of the screen. This is why it is called an **image-precision** method, also known as an image-space, approximate, finite-resolution or discrete method. The second approach handles the visible parts of object projections at the precision of the object description, which is limited only by the finite precision of floating point calculations in the computer. The algorithms falling into this class are categorized as **object-precision** algorithms, alternatively as object-space, exact, infinite-resolution or continuous methods [SSS74].

The following pseudo-codes give a preliminary comparison to emphasize the differences between the two main categories of visibility calculation algorithms. An image-precision algorithm typically appears as follows:

```
ImagePrecisionAlgorithm
    do
        select a set P of pixels on the screen;
        determine visible objects in P;
        for each pixel p ∈ P do
            draw the object determined as visible at p;
        endfor
    while not all pixels computed
end
```

The set of pixels (P) selected in the outer loop depends on the nature of the algorithm: it can be a single pixel (ray tracing) or a row of pixels (scan-line algorithm) or the pixels covered by a given object (z-buffer algorithm). An object-precision algorithm, on the other hand, typically appears as follows:

```
ObjectPrecisionAlgorithm
    determine the set S of visible objects;
    for each object o ∈ S do
        for each pixel p covered by o do
            draw o at p;
        endfor
    endfor
end
```

If N, R^2 are the number of objects and the number of pixels respectively, then an image-precision algorithm always has a lower bound of $\Omega(R^2)$ for its running time, since every pixel has to be considered at least once. An object-precision algorithm, on the other hand, has a lower bound of $\Omega(N)$ for its time complexity. But these bounds are very optimistic; the first one does not consider that finding the visible object in a pixel requires more and more time as the number of objects grows. The other does not give any indication of how complicated the objects and hence the final image can be. Unfortunately, we cannot expect our algorithms to reach these lower limits.

In the case of image-space algorithms, in order to complete the visibility calculations in a time period proportional to the number of pixels and independent of the number of objects, we would have to be able to determine the closest object along a ray from the eye in a time period independent of the number of objects. But if we had an algorithm that could do this, this algorithm could, let us say, be used for reporting the smallest number in a non-ordered list within time period independent of the number of list elements, which is theoretically impossible. The only way of speeding this up is by preprocessing the objects into some clever data structure before the calculations but there are still theoretical limits.

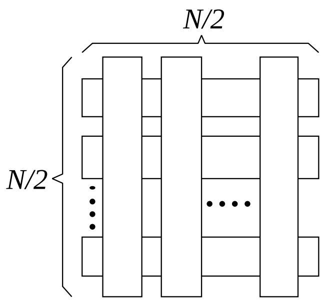

Figure 6.1: Large number of visible parts

In the case of object-space algorithms, let us first consider an extreme example, as shown in figure 6.1. The object scene is a grid consisting of $N/2$ horizontal slabs and $N/2$ vertical slabs in front of the horizontal ones. If the projections of the two groups fall onto each other on the image plane, then the number of the separated visible parts is $\Theta(N^2)$. This simple example shows that an object-precision visibility algorithm with a worst-case running time proportional to the number of objects is impossible, simply because of the potential size of the output.

Since the time spent on visibility calculations is usually overwhelming in 3D rendering, the speed of these algorithms is of great importance. There is no optimal method in either of the two classes (possessing the above-mentioned lower limit speed). This statement, however, holds only if the examinations are performed for the worst case. There are algorithms that have optimal speed in most cases (average case optimal algorithms).

6.1 Simple ray tracing

Perhaps the most straightforward method of finding the point on the surface of an object from where the light ray through a given pixel comes, is to take a half-line starting from the eye and going through (the center of) the pixel, and test it with each object for intersection. Such a ray can be represented by a pair $(\vec{s}, \vec{d})$, where $\vec{s}$ is the starting point of the ray and $\vec{d}$ is its direction vector. The starting point is usually the eye position, while the direction vector is determined by the relative positions of the eye and the actual pixel. Of all the intersection points the one closest to the eye is kept. Following this image-precision approach, we can obtain the simplest ray tracing algorithm:

```
for each pixel p do
    r = ray from the eye through p;
    visible object = null;
    for each object o do
        if r intersects o then
            if intersection point is closer than previous ones then
                visible object = o;
            endif
        endif
    endfor
    if visible object ≠ null then
        color of p = color of visible object at intersection point;
    else
        color of p = background color;
    endif
endfor
```

When a ray is to be tested for intersection with objects, each object is taken one by one, hence the algorithm requires $O(R^2N)$ time (both in worst and average case) to complete the rendering. This is the worst that we can imagine, but the possibilities of this algorithm are so good — we will examine it again in chapter 9 on recursive ray tracing — that despite its slowness ray tracing is popular and it is worth making the effort to accelerate it. The algorithm shown above is the "brute force" form of ray tracing.

The method has a great advantage compared to all the other visible surface algorithms. It works directly in the world coordinate system, it can realize any type of projection, either perspective or parallel, without using transformation matrices and homogeneous division, and finally, clipping is also done automatically (note, however, that if there are many objects falling outside of the viewport then it is worth doing clipping before ray tracing). The first advantage is the most important. A special characteristic of the perspective transformation — including homogeneous division — is that the geometric nature of the object is generally not preserved after the transformation. This means that line segments and polygons, for example, can be represented in the same way as before the transformation, but a sphere will no longer be a sphere. Almost all types of object are sensitive to perspective transformation, and such objects must always be approximated by transformation-insensitive objects, usually by polygons, before the transformation. This leads to loss of geometric information, and adversely affects the quality of the image.

The key problem in ray tracing is to find the intersection between a ray $\vec{r}(\vec{s}, \vec{d})$ and the surface of a geometric object o. Of all the intersection points we are mainly interested in the first intersection along the ray (the one closest the origin of the ray). In order to find the closest one, we usually have to calculate all the intersections between $\vec{r}$ and the surface of o, and then select the one closest to the starting point of $\vec{r}$. During these calculations the following parametric representation of the ray is used:

$$\vec{r}(t) = \vec{s} + t \cdot \vec{d} \quad (t \in [0, \infty)). \tag{6.1}$$

The parameter t refers to the the distance of the actual ray point $\vec{r}(t)$ from the starting point $\vec{s}$. The closest intersection can then be found by comparing the t values corresponding to the intersection points computed.

6.1.1 Intersection with simple geometric primitives

If object o is a sphere, for example, with its center at $\vec{c}$ and of radius R, then the equation of the surface points $\vec{p}$ is:

$$|\vec{p} - \vec{c}| = R \tag{6.2}$$

where $|\cdot|$ denotes vector length. The condition for intersection between the sphere and a ray $\vec{r}$ is that $\vec{p} = \vec{r}$ for some $\vec{p}$. Substituting the parametric expression 6.1 of ray points for $\vec{p}$ into 6.2, the following quadratic equation is derived with parameter t as the only unknown:

$$(\vec{d})^2 \cdot t^2 + 2 \cdot \vec{d} \cdot (\vec{s} - \vec{c}) \cdot t + (\vec{s} - \vec{c})^2 - R^2 = 0 \tag{6.3}$$

This equation can be solved using the resolution formula for quadratic equations. It gives zero, one or two different solutions for t, corresponding to the cases of zero, one or two intersection points between the ray and the surface of the sphere, respectively. An intersection point itself can be derived by substituting the value or values of t into expression 6.1 of the ray points. Similar equations to 6.2 can be used for further quadratic primitive surfaces, such as cylinders and cones.

The other type of simple primitive that one often meets is the planar polygon. Since every polygon can be broken down into triangles, the case of a triangle is examined, which is given by its vertices $\vec{a}, \vec{b}$ and $\vec{c}$. One possibility of calculating the intersection point is taking an implicit equation — as in the case of spheres — for the points $\vec{p}$ of the (plane of the) triangle. Such an equation could look like this:

$$((\vec{b} - \vec{a}) \times (\vec{c} - \vec{a})) \cdot (\vec{p} - \vec{a}) = 0 \tag{6.4}$$

which, in fact, describes the plane containing the triangle. Substituting the expression of the ray into it, a linear equation is constructed for the unknown ray parameter t. This can be solved easily, and always yields a solution, except in cases where the ray is parallel to the plane of the triangle. But there is a further problem. Since equation 6.4 describes not only the points of the triangle, but all the points of the plane containing the triangle, we have to check whether the intersection point is inside the triangle. This leads to further geometric considerations about the intersection point $\vec{p}$. We can check, for example, that for each side of the triangle, $\vec{p}$ and the third vertex fall onto the same side of it, that is:

$$\begin{aligned} ((\vec{b}-\vec{a}) \times (\vec{p}-\vec{a})) \cdot ((\vec{b}-\vec{a}) \times (\vec{c}-\vec{a})) &\geq 0, \\ ((\vec{c}-\vec{b}) \times (\vec{p}-\vec{b})) \cdot ((\vec{b}-\vec{a}) \times (\vec{c}-\vec{a})) &\geq 0, \\ ((\vec{a}-\vec{c}) \times (\vec{p}-\vec{c})) \cdot ((\vec{b}-\vec{a}) \times (\vec{c}-\vec{a})) &\geq 0 \end{aligned} \tag{6.5}$$

The point $\vec{p}$ falls into the triangle if and only if all the three inequalities hold.

An alternative approach is to use an explicit expression of the inner points of the triangle. These points can be considered as positive-weighted linear combinations of the three vertices, with a unit sum of weights:

$$\begin{aligned} \vec{p}(\alpha,\beta,\gamma) &= \alpha \cdot \vec{a} + \beta \cdot \vec{b} + \gamma \cdot \vec{c}, \\ \alpha,\beta,\gamma &\geq 0, \\ \alpha+\beta+\gamma &= 1 \end{aligned} \tag{6.6}$$

The coefficients α, β and γ are also known as the **baricentric coordinates** of point $\vec{p}$ with respect to the spanning vectors $\vec{a}, \vec{b}$ and $\vec{c}$ (as already described in section 5.1). For the intersection with a ray, the condition $\vec{p} = \vec{r}$ must hold, giving a linear equation for the four unknowns α, β, γ and t:

$$\begin{aligned} \alpha \cdot \vec{a} + \beta \cdot \vec{b} + \gamma \cdot \vec{c} &= \vec{s} + t \cdot \vec{d}, \\ \alpha+\beta+\gamma &= 1 \end{aligned} \tag{6.7}$$

The number of unknowns can be reduced by merging the second equation into the first one. Having solved the merged equation, we have to check whether the resulting intersection point is inside the triangle. In this case, however, we only have to check that $\alpha \geq 0, \beta \geq 0$ and $\gamma \geq 0$.

The two solutions for the case of the triangle represent the two main classes of intersection calculation approaches. In the first case, the surface of the object is given by an implicit equation $F(x, y, z) = 0$ of the spatial coordinates of the surface. In this case, we can always substitute expression 6.1 of the ray into the equation, getting a single equation for the unknown ray parameter t. In the other case, the surface points of the object are given explicitly by a parametric expression $\vec{p} = \vec{p}(u, v)$, where u, v are the surface parameters. In this case, we can always derive an equation system $\vec{p}(u, v) - \vec{r}(t) = \vec{0}$ for the unknowns, u, v and t. In the first case, the equation is only a single one (although usually non-linear), but objects usually only use a portion of the surface described by the implicit equation and checking that the point is in the part used causes extra difficulties. In the second case, the equation is more complicated (usually a non-linear equation system), but checking the validity of the intersection point requires only comparisons in parameter space.

6.1.2 Intersection with implicit surfaces

In the case where the surface is given by an implicit equation $F(x,y,z) = 0$, the parametric expression 6.1 of the ray can be substituted into it to arrive at the equation $f(t) = F(x(t), y(t), z(t)) = 0$, thus what has to be solved is:

$$f(t) = 0 \tag{6.8}$$

This is generally non-linear, and we cannot expect to derive the roots in analytic form, except in special cases.

One more thing should be emphasized here. From all the roots of $f(t)$, we are interested *only* in *its real roots* (complex roots have no geometric meaning). Therefore the problem of finding the real roots will come to the front from now on.

Approximation methods

Generally some **approximation method** must be used in order to compute the roots with any desired accuracy. The problem of approximate solutions to non-linear equations is one of the most extensively studied topics in computational mathematics. We cannot give here more than a collection of related theorems and techniques (mainly taken from the textbook by Demidovich and Maron [DM87]). It will be assumed throughout this section that the function f is continuous and continuously differentiable.

A basic observation is that if $f(a) \cdot f(b) < 0$ for two real numbers a and b, then the interval $[a, b]$ contains at least one root of $f(t)$. This condition of changing the sign is sufficient but not necessary. A counter example is an interval containing an even number of roots. Another counter example is a root where the function has a local minimum or maximum of 0 at the root, that is, the first derivative $f'(t)$ also has a root at the same place as $f(t)$. The reason for the first situation is that the interval contains more than one root instead of an *isolated* one. The reason for the second case is that the root has a *multiplicity* of more than one. Techniques are known for both isolating the roots and reducing their multiplicity, as we will see later.

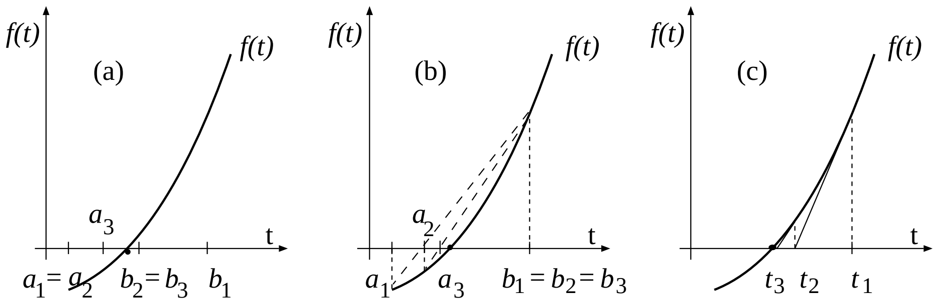

Figure 6.2: Illustrations for the halving (a), chord (b) and Newton's (c) method

If $f(a) \cdot f(b) < 0$ and we know that the interval $[a, b]$ contains exactly one root of $f(t)$, then we can use a number of techniques for approximating this root t^* as closely as desired. Probably the simplest technique is known as the **halving method**. First the interval is divided in half. If $f((a+b)/2) = 0$, then $t^* = (a+b)/2$ and we stop. Otherwise we keep that half, $[a, (a+b)/2]$ or $[(a+b)/2, b]$, at the endpoints of which $f(t)$ has opposite signs. This reduced interval $[a_1, b_1]$ will contain the root. Then this interval is halved in the same way as the original one and the same investigations are made, etc. Continuing this process, we either find the exact value of the root or produce a nested sequence $\{[a_i, b_i]\}$ of intervals of rapidly decreasing width:

$$b_i - a_i = \frac{1}{2^i}(b - a). \tag{6.9}$$

The sequence contracts into a single value in the limiting case $i \to \infty$, which value is the desired root:

$$t^* = \lim_{i\to\infty} a_i = \lim_{i\to\infty} b_i. \tag{6.10}$$

Another simple technique is the **method of chords**, also known as the method of proportional parts. Instead of simply halving the interval $[a, b]$, it is divided at the point where the function would have a root if it were linear (a chord) between $a, f(a)$ and $b, f(b)$. If — without loss of generality

— we assume that $f(a) < 0$ and $f(b) > 0$, then the ratio of the division will be $-f(a)/f(b)$, giving an approximate root value thus:

$$t_1 = a - \frac{f(a)}{f(b) - f(a)}(b - a). \tag{6.11}$$

If $f(t_1) = 0$, then we stop, otherwise we take the interval $[a, t_1]$ or $[t_1, b]$, depending on that at which endpoints the function $f(t)$ has opposite signs, and produce a second approximation t_2 of the root, etc. The convergence speed of this method is generally faster than that of the halving method.

A more sophisticated technique is **Newton's method**, also known as the method of tangents. It takes more of the local nature of the function into consideration during consecutive approximations. The basic idea is that if we have an approximation t_1 close to the root t^*, and the difference between them is δt, then $f(t^*) = 0$ implies $f(t_1 + \delta t) = 0$. Using the first two terms of Taylor's formula for the latter equation, we get:

$$f(t_1 + \delta t) \approx f(t_1) + f'(t_1) \cdot \delta t = 0 \tag{6.12}$$

Solving this for δt gives $\delta t \approx -f(t_1)/f'(t_1)$. Adding this to t_1 results in a new (probably closer) approximation of the root t^*. The general scheme of the iteration is:

$$t_{i+1} = t_i - \frac{f(t_i)}{f'(t_i)} \qquad (i = 1, 2, 3, \ldots). \tag{6.13}$$

The geometric interpretation of the method (see figure 6.2) is that at each approximation t_i the function $f(t)$ is replaced by the tangent line to the curve at $t_i, f(t_i)$ in order to find the next approximation value t_{i+1}. Newton's method is the most rapidly convergent of the three techniques we have looked at so far, but only if the iteration sequence 6.13 is convergent. If we are not careful, it can become divergent. The result can easily depart from the initial interval $[a, b]$ if for some t_i the value of $f'(t_i)$ is much less than that of $f(t_i)$. There are many theorems about "good" initial approximations, from which the approximation sequence is guaranteed to be convergent. One of these is as follows. If $f(a) \cdot f(b) < 0$, and $f'(t)$ and $f''(t)$ are non-zero and preserve signs over $a \leq t \leq b$, then, proceeding from an initial approximation $t_1 \in [a, b]$ which satisfies $f'(t_1) \cdot f''(t_1) > 0$, it is possible to compute the sole root t^* of $f(t)$ in $[a, b]$ to any degree of accuracy by using

Newton's iteration scheme (6.13). Checking these conditions is by no means a small matter computationally. One possibility is to use interval arithmetic (see section 6.1.3). There are many further approximation methods beyond the three basic ones that we have outlined, but they are beyond the scope of this book.

Reducing the multiplicity of roots of algebraic equations

The function $f(t)$ is algebraic in most practical cases of shape modeling and also in computer graphics. This comes from the fact that surfaces are usually defined by algebraic equations, and the substitution of the linear expression of the ray coordinates also gives an algebraic equation. The term **algebraic** means that the function is a polynomial (rational) function of its variable. Although the function may have a denominator, the problem of solving the equation $f(t) = 0$ is equivalent with the problem of finding the roots of the numerator of $f(t)$ and then checking that the denominator is non-zero at the roots. That is, we can restrict ourselves to equations having the following form:

$$f(t) = a_0 t^n + a_1 t^{n-1} + \ldots + a_n = 0 \tag{6.14}$$

The fundamental theorem of algebra says that a polynomial of degree n (see equation 6.14 with $a_0 \neq 0$) has exactly n roots, real or complex, provided that each root is counted according to its multiplicity. We say that a root t^* has **multiplicity** m if the following holds:

$$f(t^*) = f'(t^*) = f''(t^*) = \ldots = f^{(m-1)}(t^*) = 0 \quad \text{and} \quad f^{(m)}(t^*) \neq 0 \tag{6.15}$$

We will restrict ourselves to algebraic equations in the rest of the subsection, because this special property can be exploited in many ways.

Multiplicity of roots can cause problems in the approximation of the roots, as we pointed out earlier. Fortunately, any algebraic equation can be **reduced** to another equation of lower or equal degree, which has the same roots, each having a multiplicity of one. If $t_1^*, t_2^*, \ldots, t_k^*$ are the distinct roots of $f(t)$ with multiplicities of $m_1, m_2, \ldots, m_k$, respectively, then the polynomial can be expressed by the following product of terms:

$$f(t) = a_0 (t - t_1^*)^{m_1} (t - t_2^*)^{m_2} \cdots (t - t_k^*)^{m_k}, \quad \text{where} \ \ m_1 + m_2 + \ldots + m_k = n. \tag{6.16}$$

The first derivative $f'(t)$ can be expressed by the following product:

$$f'(t) = a_0(t - t_1^*)^{m_1-1}(t - t_2^*)^{m_2-1} \cdots (t - t_k^*)^{m_k-1} p(t) \tag{6.17}$$

where

$$p(t) = m_1(t - t_2^*) \cdots (t - t_k^*) + \ldots + (t - t_1^*) \cdots (t - t_{k-1}^*) m_k. \tag{6.18}$$

Note that the polynomial $p(t)$ has a non-zero value at each of the roots $t_1^*, t_2^*, \ldots, t_k^*$ of $f(t)$. As a consequence of this, the polynomial:

$$d(t) = a_0(t - t_1^*)^{m_1-1}(t - t_2^*)^{m_2-1} \cdots (t - t_k^*)^{m_k-1} \tag{6.19}$$

is the greatest common divisor of the polynomials $f(t)$ and $f'(t)$, that is:

$$d(t) = \gcd(f(t), f'(t)). \tag{6.20}$$

This can be computed using Euclid's algorithm. Dividing $f(t)$ by $d(t)$ yields the following result:

$$g(t) = \frac{f(t)}{d(t)} = (t - t_1^*)(t - t_2^*) \cdots (t - t_k^*) \tag{6.21}$$

(compare the terms in expression 6.16 of $f(t)$ with those in the expression of $d(t)$). All the roots of $g(t)$ are distinct, have a multiplicity of 1 and coincide with the roots of $f(t)$.

Root isolation

The problem of **root isolation** is to find appropriate disjoint intervals $[a_1, b_1], [a_2, b_2], \ldots, [a_k, b_k]$, each containing exactly one of the distinct real roots $t_1^*, t_2^*, \ldots, t_k^*$, respectively, for the polynomial $f(t)$.

An appropriate first step is to find a finite interval containing all the roots, because it can then be recursively subdivided. Lagrange's theorem helps in this. It states the following about the upper bound R of the positive roots of the equation: Suppose that $a_0 > 0$ in expression 6.14 of the polynomial and a_k ($k \geq 1$) is the first of the negative coefficients (if there is no such coefficient, then $f(t)$ has no positive roots). Then for the upper bound of the positive roots of $f(t)$ we can take the number:

$$R = 1 + \sqrt[k]{\frac{B}{a_0}} \tag{6.22}$$

where B is the largest absolute value of the negative coefficients of the polynomial $f(t)$. Using a little trick, this single theorem will be enough to give both upper and lower bounds for the positive and negative roots as well. Let us create the following three equations from our original $f(t)$:

$$\begin{aligned} f_1(t) &= t^n f\left(\frac{1}{t}\right) = 0, \\ f_2(t) &= f(-t) = 0, \\ f_3(t) &= t^n f\left(-\frac{1}{t}\right) = 0 \end{aligned} \tag{6.23}$$

Let the upper bound of their positive roots be R_1, R_2 and R_3, respectively. Then any positive root t^+ and negative root t^- of $f(t)$ will satisfy (R comes from equation 6.22):

$$\begin{aligned} \frac{1}{R_1} \leq \ & t^+ \ \leq R, \\ -R_2 \leq \ & t^- \ \leq -\frac{1}{R_3}. \end{aligned} \tag{6.24}$$

Thus we have at most two finite intervals containing all possible roots. Then we can search for subintervals, each containing exactly one real root. There are a number of theorems of numerical analysis useful for determining the number of real roots in a given interval, such as the one based on Sturm-sequences [Ral65, Ral69] or the Budan–Fourier theorem [DM87]. Instead of reviewing any of these here, a simple method will be shown which is easy to implement and efficient if the degree of the polynomial $f(t)$ is not too large.

The basic observation is that if t_i^* and t_j^* are two distinct roots of of the polynomial $f(t)$ and $t_i^* < t_j^*$, then there is definitely a value $t_i^* < \tau^* < t_j^*$ between them for which $f'(\tau^*) = 0$. This implies that each pair t_i^*, t_{i+1}^* of consecutive roots are separated by a value (or more than one values) τ_i^* ($t_i^* < \tau_i^* < t_{i+1}^*$) for which $f'(\tau_i^*) = 0$ ($1 \leq i \leq k-1$ where k is the number of distinct roots of $f(t)$). This is illustrated in figure 6.3. Note, however, that the contrary is not true: if τ_i^* and τ_j^* are two distinct roots of $f'(t)$ then there is not necessarily a root of $f(t)$ between them. These observations lead to a recursive method:

- Determine the approximate distinct real roots of $f'(t)$. This yields the values $\tau_1^* < \ldots < \tau_{n'}^*$, where $n' < n$ (n is the degree of $f(t)$). Then

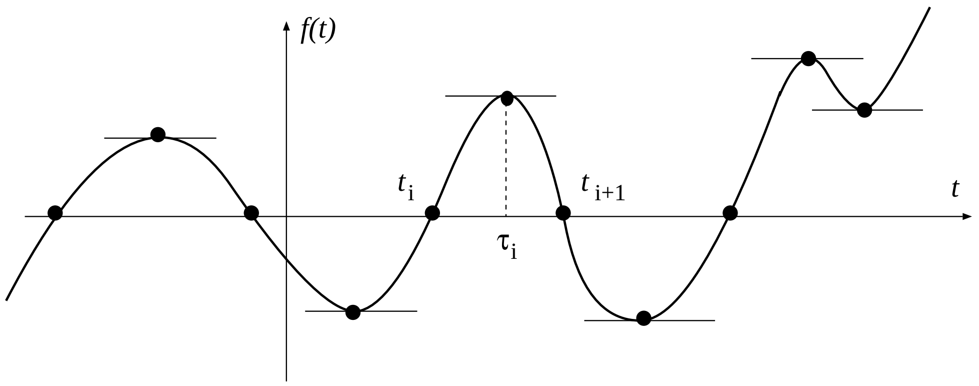

Figure 6.3: Roots isolated by the roots of derivative

each of the $n'+1$ intervals $[-\infty, \tau_1^*], [\tau_1^*, \tau_2^*], \ldots, [\tau_{n'}^*, \infty]$ contains either exactly one root or no roots of $f(t)$. If it is ensured that all the roots of $f(t)$ are of multiplicity 1 (see previous subsection) then it is easy to distinguish between the two cases: if $f(\tau_i^*) \cdot f(\tau_{i+1}^*) < 0$ then the interval $[\tau_i^*, \tau_{i+1}^*]$ contains one root, otherwise it contains no roots. If there is a root in the interval, then an appropriate method can be used to approximate it.

- The approximate distinct real roots of $f'(t)$ can be found recursively. Since the degree of $f'(t)$ is one less than that of $f(t)$ the recursion always terminates.
- At the point where the degree of $f(t)$ becomes 2 (at the bottom of the recursion) the second order equation can be solved easily.

Note that instead of the intervals $[-\infty, \tau_1^*]$ and $[\tau_{n'}^*, \infty]$ the narrower intervals $[-R_2, \tau_1^*]$ and $[\tau_{n'}^*, R]$ can be used, where R_2 and R are defined by equations 6.23 and 6.22.

An example algorithm

As a summary of this section, a possible algorithm is given for approximating all the real roots of a polynomial $f(t)$. It maintains a list L for storing the approximate roots of $f(t)$ and a list L' for storing the approximate roots of $f'(t)$. The lists are assumed to be sorted in increasing order. The notation $\deg(f(t))$ denotes the degree of the polynomial $f(t)$ (the value of n in expression 6.14):

```
Solve(f(t))
    L = {};
    if deg(f(t)) < 3 then
        add roots of f(t) to L                    // 0, 1 or 2 roots
        return L;
    endif
    calculate g(t) = f(t)/gcd(f(t), f'(t));       // eq. 6.21 and 6.20
    L' = Solve(g'(t));                            // roots of derivative
    add -R_2 and R to L';                         // eq. 6.22 and 6.23
    a = first item from L';
    while L' not empty do
        b = next item from L';
        if g(a) · g(b) < 0                        // [a, b] contains one root
            t = approximation of the root in [a, b];
            add t to L;
        endif
        a = b;
    endwhile
    return L;
end
```

6.1.3 Intersection with explicit surfaces

If we are to find the intersection point between a ray $\vec{r}(t)$ and an explicitly given free-form surface $\vec{s}(u, v)$, then, in fact, the following equation is to be solved:

$$\vec{f}(\vec{x}) = \vec{0}, \tag{6.25}$$

where $\vec{f}(\vec{x}) = \vec{f}(u, v, t) = \vec{s}(u, v) - \vec{r}(t)$, and the mapping $\vec{f}$ is usually non-linear. We can dispose of the problem of solving a non-linear equation system if we approximate the surface $\vec{s}$ by a finite number of planar polygons and then solve the linear equation systems corresponding to the individual polygons one by one. This method is often used, because it is straightforward and easy to implement, but if we do not allow such anomalies as jagged contours of smooth surfaces on the picture, then we either have to use a huge number of polygons for the approximation with the snag of having to check all of them for intersection, or we have to use a numerical root-finding method for computing the intersection point within some tolerance.

Newton's method is a classical numerical method for approximating any real root of a non-linear equation system $\vec{f}(\vec{x}) = \vec{0}$. If $[\partial\vec{f}/\partial\vec{x}]$ is the Jacobian matrix of $\vec{f}$ at $\vec{x}$, then the recurrence formula is:

$$\vec{x}_{k+1} = \vec{x}_k - \left[\frac{\partial\vec{f}}{\partial\vec{x}}\right]^{-1} \vec{f}(\vec{x}_k). \tag{6.26}$$

If our initial guess $\vec{x}_0$ is close enough to a root $\vec{x}^*$, then the sequence $\vec{x}_k$ is convergent, and $\lim_{k\to\infty} \vec{x}_k = \vec{x}^*$. The main problem is how to produce such a good initial guess for each root. A method is needed which always leads to reasonable starting points before performing the iterations. We need, however, computationally performable tests.

One possible method will be introduced in this chapter. The considerations leading to the solution are valid in the n-dimensional real space $\mathbf{R}^n$. For the sake of notational simplicity, the superscript ($\vec{\ }$) above vector variables will be omitted. They will be reintroduced when returning to our three-dimensional object space.

The method is based on a fundamental theorem of topology: **Schauder's fixpoint theorem** [Sch30, KKM29]. It states that if $X \subset \mathbf{R}^n$ is a convex and compact set and $g\colon \mathbf{R}^n \to \mathbf{R}^n$ is a continuous mapping, then $g(X) \subseteq X$ implies that g has a fixed point $\mathbf{x} \in X$ (that is for which $g(\mathbf{x}) = \mathbf{x}$). Let the mapping g be defined as:

$$g(\mathbf{x}) = \mathbf{x} - \mathbf{Y}f(\mathbf{x}), \tag{6.27}$$

where $\mathbf{Y}$ is a non-singular $n \times n$ matrix. Then, as a consequence of Schauder's theorem, $g(X) \subseteq X$ implies that there is a point $\mathbf{x}^* \in X$ for which:

$$g(\mathbf{x}^*) = \mathbf{x}^* - \mathbf{Y}f(\mathbf{x}^*) = \mathbf{x}^*. \tag{6.28}$$

Since $\mathbf{Y}$ is non-singular, it implies that $f(\mathbf{x}^*) = 0$. In other words, if $g(X) \subseteq X$, then there is *at least one* solution to $f(\mathbf{x}^*) = 0$ in X. Another important property of the mapping g is that if $\mathbf{x}^* \in X$ is such a root of f, then $g(\mathbf{x}^*) \in X$. This is so because if $f(\mathbf{x}^*) = 0$ then $g(\mathbf{x}^*) = \mathbf{x}^* \in X$. Thus we have a test for the existence of roots of f in a given set X. It is based on the comparison of the set X and its image $g(X)$:

- if $g(X) \subseteq X$ then the answer is positive, that is, X contains *at least one* root
- if $g(X) \cap X = \emptyset$ then the answer is negative, that is, X contains *no* roots, since if it contained one, then this root would also be contained by $g(X)$, but this would be a contradiction
- if none of the above two conditions holds then the answer is neither positive nor negative; in this latter case, however, the set X can be divided into two or more subsets and these smaller pieces can be examined similarly, leading to a recursive algorithm

An important question, if one intends to use this test, is that how the image $g(X)$ and its intersection with X can be computed. Another important problem, if the test gives a positive answer for X, is to decide where to start the Newton-iteration from. A numerical technique, called *interval arithmetic*, gives a possible solution to the first problem. We will survey it here. What it offers is its simplicity, but the price we have to pay is that we never get more than rough estimations for the ranges of mappings. The second problem will be solved by an interval arithmetic based *modification* of the Newton-iteration scheme.

Interval arithmetic

A branch of numerical analysis, called *interval analysis*, basically deals with real intervals, vectors of real intervals, and mappings from and into such objects. Moore's textbook [Moo66] gives a good introduction to it. Our overview contains only those results, which are relevant from the point of view of our problem. Interval objects will be denoted by capital letters.

Let us start with algebraic operations on intervals (addition, subtraction, multiplication and division). Generally, if a binary operation $\circ$ is to be extended to work on two real intervals $X_1 = [a_1, b_1]$ and $X_2 = [a_2, b_2]$, then the rule is:

$$X_1 \circ X_2 = \{x_1 \circ x_2 \mid x_1 \in X_1 \text{ and } x_2 \in X_2\} \qquad (6.29)$$

that is, the resulting interval should contain the results coming from all the possible pairings. In the case of subtraction, for example, $X_1 - X_2 = [a_1 - b_2, b_1 - a_2]$. Such an **interval extension** of an operation is *inclusion monotonic*, that is, if $X_1' \subset X_1$ then $X_1' \circ X_2 \subset X_1 \circ X_2$. Based on these operations, the interval extension of an algebraic function can easily be derived by substituting each of its operations by the corresponding interval extension. The (inclusion monotonic) interval extension of a function $f(x)$ will be denoted by $F(X)$. If $f(x)$ is a multidimensional mapping (where x is a vector) then $F(X)$ operates on vectors of intervals called *interval vectors*. The interval extension of a linear mapping can be represented by an *interval matrix* (matrix of intervals).

An interesting fact is that the Lagrangean mean-value theorem extends to the interval extension of functions (although it does not extend to ordinary vector-vector functions). It implies that if f is a continuously differentiable mapping, and F is its interval extension, then for all $\mathbf{x}, \mathbf{y} \in X$:

$$f(\mathbf{x}) - f(\mathbf{y}) \in F'(X)(\mathbf{x} - \mathbf{y}), \qquad (6.30)$$

where X is an interval vector (box), $\mathbf{x}, \mathbf{y}$ are real vectors, and F' is the interval extension of the Jacobian matrix of f.

Let us now see some useful definitions. If $X = [a, b]$ is a real interval, then its absolute value, width and middle are defined as:

$$\begin{aligned} |X| &= \max(|a|, |b|) && \text{(absolute value)}, \\ \mathrm{w}(X) &= b - a && \text{(width)}, \\ \mathrm{m}(X) &= (a + b)/2 && \text{(middle)} \end{aligned} \qquad (6.31)$$

If $X = (X_1, \ldots, X_n)$ is an interval vector, then its respective vector norm, width and middle vector are defined as:

$$\begin{aligned} |X| &= \max_i \{|X_i|\}, \\ \mathrm{w}(X) &= \max_i \{\mathrm{w}(X_i)\}, \\ \mathrm{m}(X) &= (\mathrm{m}(X_1), \ldots, \mathrm{m}(X_n)) \end{aligned} \qquad (6.32)$$

For an interval matrix $\mathbf{A} = [A_{ij}]$ the row norm and middle matrix are defined as:

$$\begin{aligned} \|\mathbf{A}\| &= \max_i \left\{ \sum_{j=1}^{n} |A_{ij}| \right\}, \\ \mathrm{m}(\mathbf{A}) &= [\mathrm{m}(A_{ij})] \end{aligned} \tag{6.33}$$

The above defined norm for interval matrices is very useful. We will use the following corollary of this definition later: it can be derived from the definitions [Moo77] that, for any interval matrix $\mathbf{A}$ and interval vector X:

$$\mathrm{w}(\mathbf{A}(X - \mathrm{m}(X))) \leq \|\mathbf{A}\| \cdot \mathrm{w}(X). \tag{6.34}$$

That is, we can estimate the width of the interval vector containing all the possible images of an interval vector $(X - \mathrm{m}(X))$ if transformed by any of the linear transformations contained in a bundle of matrices (interval matrix $\mathbf{A}$), and we can do this by simple calculations. Note, however, that this inequality can be used only for a special class of interval vectors (origin centered boxes).

Interval arithmetic and the Newton-iteration

We are now in position to perform the test $g(X) \subseteq X$ (equation 6.27) in order to check whether X contains a root (provided that X is a rectangular box): if the interval extension of $g(\mathbf{x})$ is $G(X)$, then $g(X) \subseteq G(X)$, and hence $G(X) \subseteq X$ implies $g(X) \subseteq X$.

Now the question is the following: provided that X contains a root, is the Newton-iteration convergent from any point of X? Another question is that how many roots are in X: only one (a unique root) or more than one? Although it is also possible to answer these questions based on interval arithmetic, the interested reader is referred to Toth's article [Tot85] about this subject. We will present here another method which can be called *an interval version* of the Newton-iteration, first published by Moore [Moo77]. In fact, Toth's work is also based on this method.

The goal of the following argument will be to create an iteration formula, based on the Newton-iteration, which produces a *nested sequence* of interval vectors:

$$X \supset X_1 \supset X_2 \supset \ \ldots \tag{6.35}$$

converging to the *unique* solution $\mathbf{x}^* \in X$ if it exists. A test scheme suitable for checking in advance whether a unique $\mathbf{x}^*$ exists will also be provided.

Based on the interval extension $G(X)$ of the mapping $g(\mathbf{x})$ (equation 6.27), consider now the following iteration scheme:

$$X_{k+1} = G(X_k) \quad \text{where } X_0 = X. \tag{6.36}$$

We know that if $G(X) \subseteq X$ then there is *at least one* root $\mathbf{x}^*$ of f in X. It is also sure that for each such $\mathbf{x}^*$, $\mathbf{x}^* \in X_k$ (for all $k \geq 0$), that is, the sequence of interval boxes contains each root. If, furthermore, there exists a positive real number $r < 1$ so that $\mathrm{w}\,(X_{k+1}) \leq r \cdot \mathrm{w}\,(X_k)$ for all $k \geq 0$, then $\lim\limits_{k\to\infty} \mathrm{w}\,(X_k) = 0$, that is, the sequence of interval vectors contracts onto a *single* point. This implies that if the above conditions hold then X contains a *unique* solution $\mathbf{x}^*$ and iteration 6.36 converges to $\mathbf{x}^*$. How can the existence of such a number r (the "contraction factor") be verified in advance?

Inequality 6.34 is suitable for estimating the width of an interval vector resulting from (the interval extension of) a *linear mapping* performed on a *symmetric* interval vector. In order to exploit this inequality, the mapping should be made linear and the interval vector should be made symmetric. Let the expression of mapping g be rewritten as:

$$g(\mathbf{x}) = \mathbf{x} - \mathbf{Y}\left(f(\mathrm{m}\,(X)) + f(\mathbf{x}) - f(\mathrm{m}\,(X))\right) \tag{6.37}$$

where X can be any interval vector. Following from the Lagrangean mean-value theorem:

$$g(\mathbf{x}) \in \mathbf{x} - \mathbf{Y} f(\mathrm{m}\,(X)) - \mathbf{Y} F'(X)\,(\mathbf{x} - \mathrm{m}\,(X)) \tag{6.38}$$

provided that $\mathbf{x} \in X$. Following from this, the interval extension of g will satisfy (decomposing the right-hand side into a real and an interval term):

$$G(X) \subseteq \mathrm{m}\,(X) - \mathbf{Y} f(\mathrm{m}\,(X)) + [\mathbf{1} - \mathbf{Y} F'(X)]\,(X - \mathrm{m}\,(X)) \tag{6.39}$$

where $\mathbf{1}$ is the unit matrix. Note that the interval mapping on the right-hand side is a linear mapping performed on a symmetric interval vector. Applying now inequality 6.34 (and because $\mathrm{w}\,(X - \mathrm{m}\,(X)) = \mathrm{w}\,(X)$):

$$\mathrm{w}\,(G(X)) \leq \|\mathbf{1} - \mathbf{Y} F'(X)\| \cdot \mathrm{w}\,(X) \tag{6.40}$$

that is, checking whether iteration 6.36 is convergent has become possible. One question is still open: how should the matrix $\mathbf{Y}$ be chosen. Since the structure of the mapping g (equation 6.27) is similar to that of the Newton-step (equation 6.26 with $Y = [\partial \vec{f}/\partial \vec{x}]^{-1}$), intuition tells that $\mathbf{Y}$ should be related to the inverse Jacobian matrix of f (hoping that the convergence speed of the iteration can then be as high as that of the Newton-iteration). Taking the inverse middle of the interval Jacobian $F'(X)$ seems to be a good choice.

In fact, Moore [Moo77] introduced the mapping on the right-hand side of equation 6.39 as a special case of a mapping which he called the **Krawczyk operator**. Let us introduce it for notational simplicity:

$$K(X, \mathbf{y}, \mathbf{Y}) = \mathbf{y} - \mathbf{Y} f(\mathbf{y}) + [\mathbf{1} - \mathbf{Y} F'(X)](X - \mathbf{y}), \tag{6.41}$$

where X is an interval vector, $\mathbf{y} \in X$ is a real vector, $\mathbf{Y}$ is a non-singular real matrix and f is assumed to be continuously differentiable. The following two properties of this mapping are no more surprising. The first is that if $K(X, \mathbf{y}, \mathbf{Y}) \subseteq X$ for some $\mathbf{y} \in X$, then there exists an $\mathbf{x} \in X$ for which $f(\mathbf{x}) = 0$. The second property is that if $\mathbf{x}^*$ is such a root with $f(\mathbf{x}^*) = 0$, then $\mathbf{x}^* \in K(X, \mathbf{y}, \mathbf{Y})$.

We are now ready to obtain the interval version of Newton's iteration scheme in terms of the Krawczyk operator. Note that this scheme is no else but iteration 6.36 modified so that detecting whether it contracts onto a single point become possible. Setting

$$\begin{aligned} X_0 &= X, \\ \mathbf{Y}_0 &= [\mathrm{m}\,(F'(X_0))]^{-1}, \\ r_i &= \|\mathbf{1} - \mathbf{Y}_i F'(X_i)\| \end{aligned} \tag{6.42}$$

the iteration is defined as follows:

$$X_{i+1} = K(X_i, \mathrm{m}\,(X_i), \mathbf{Y}_i) \cap X_i, \tag{6.43}$$

$$\mathbf{Y}_{i+1} = \begin{cases} [\mathrm{m}\,(F'(X_{i+1}))]^{-1}, & \text{if } r_{i+1} \le r_i; \\ \mathbf{Y}_i, & \text{otherwise} \end{cases}$$

The initial condition that should be checked before starting the iteration is:

$$K(X_0, \mathrm{m}\,(X_0), \mathbf{Y}_0) \subseteq X_0 \quad \text{and} \quad r_0 < 1 \tag{6.44}$$

If these two conditions hold, then iteration 6.43 will produce a sequence of *nested* interval boxes converging to the unique solution $\mathbf{x}^* \in X$ of the equation system $f(\mathbf{x}) = 0$.

Let us return to our original problem of finding the intersection point (or all the intersection points) between a ray $\vec{r}(t)$ and an explicitly given surface $\vec{s}(u, v)$. Setting $\vec{f}(\vec{x}) = \vec{f}(u, v, t) = \vec{s}(u, v) - \vec{r}(t)$, the domain X where we have to find all the roots is bounded by some minimum and maximum values of u, v and t respectively. The basic idea of a possible algorithm is that we first check if initial condition 6.44 holds for X. If it does, then we start the iteration process, otherwise we subdivide X into smaller pieces and try to solve the problem on these. The algorithm maintains a list L for storing the approximate roots of $\vec{f}(\vec{x})$ and a list C for storing the candidate interval boxes which may contain solutions:

$C = \{X\}$; // *candidate list*
$L = \{\}$; // *solution list*
while C not empty **do**
  X_0 = next item on C;
  if condition 6.44 holds for X_0 **then**
    perform iteration 6.43 until $\mathrm{w}(X_k)$ is small enough;
    add $\mathrm{m}(X_k)$ to L;
  else if $\mathrm{w}(X_0)$ is not too small **then**
    subdivide X_0 into pieces $X_1, \ldots, X_s$;
    add $X_1, \ldots, X_s$ to C;
  endif
endwhile

6.1.4 Intersection with compound objects

In constructive solid geometry (CSG) (see subsection 1.6.2) compound objects are given by set operations ($\cup, \cap, \setminus$) performed on primitive geometric objects such as blocks, spheres, cylinders, cones or even halfspaces bounded by non-linear surfaces. The representation of CSG objects is usually a binary tree with the set operations in its internal nodes and the primitive objects in the leaf nodes. The root of the tree corresponds to the compound object, and its two children represent less complicated objects. If the tree possesses only a single leaf (and no internal nodes), then the intersection calculation poses no problem; we have only to compute the intersection between the ray and a primitive object. On the other hand, if two objects are combined by a single set operation, and all the intersection points are known to be on the surface of the two objects, then, considering the operation, one can easily decide whether any intersection point is on the surface of the resulting object. For example, if one of the intersection points on the first object is contained in the interior of the second one, and the combined object is the union ($\cup$) of the two, then the intersection point is not on its surface — it is internal to it — hence it can be discarded. Similar arguments can be made for any of the set operations and the possible in/out/on relationships between a point and an object.

These considerations lead us to a simple **divide-and-conquer** approach: if the tree has only a single leaf, then the intersection points between the ray and the primitive object are easily calculated, otherwise — when the root of the tree is an internal node — the intersection points are recursively calculated for the left child of the root, taking this child node as the root, and then the same is done with the right child of the root, and finally the two sets of intersection points are combined according the set operation at the root.

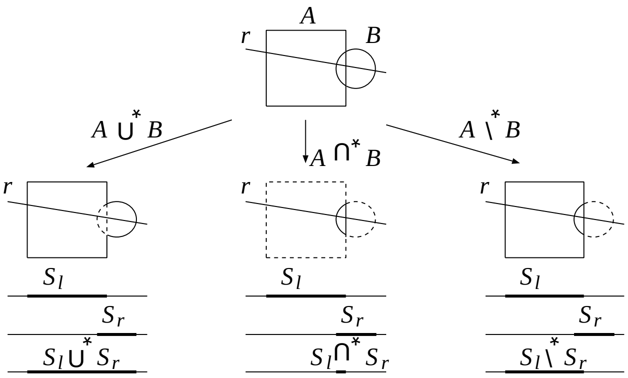

Figure 6.4: Ray spans and their combinations

A slight modification of this approach will help us in considering **regularized set operations** in ray-object intersection calculations. Recall that it was necessary to introduce regularized set operations in solid modeling in order to avoid possible anomalies resulting from an operation (see subsection 1.6.1 and figure 1.5). That is, the problem is to find the closest intersection point between a ray and a compound object, provided that the object is built by the use of regularized set operations. Instead of the isolated ray-surface intersection points, we had better deal with line segments resulting from the intersection of the ray and the solid object (more precisely, the closure of the object is to be considered, which is the complement of its exterior). The sequence of consecutive ray segments corresponding to an object will be called a **ray span**. If we take a look at figure 6.4, then we will see how the two ray spans calculated for the two child objects of a node can be combined by means of the set operation of the node. In fact, the result of the combination of the left span S_l and the right span S_r is $S_l \circ^* S_r$, where $\circ^*$ is the set operation ($\cup^*, \cap^*$ or $\setminus^*$). If we really implement the operation $\circ^*$ in the regularized way, then the result will be valid for regularized set operations. This means practically that all segments in a ray span must form a closed set with positive length. There are three cases when regularization takes place. The first is when the result span $S_l \circ^* S_r$ contains an isolated point ($\circ^*$ is $\cap^*$). This point has to be omitted because it would belong to a dangling face, edge or vertex. The second case is when the span contains two consecutive segments, and the endpoint of the first one coincides with the starting point of the second one ($\circ^*$ is $\cup^*$). The two segments have to be merged into one and the double point omitted, because it would belong to a face, edge or vertex (walled-up) in the interior of a solid object. Finally, the third case is when a segment becomes open, that is when one of its endpoints is missing ($\circ^*$ is $\setminus^*$). The segment has to be closed by an endpoint. The algorithm based on the concepts sketched in this subsection is the following:

CSGIntersec(ray, node)
    if node is compound **then**
        left span = **CSGIntersec**(ray, left child of node);
        right span = **CSGIntersec**(ray, right child of node);
        return CSGCombine(left span, right span, operation);
    else (node is a primitive object)
        return PrimitiveIntersec(ray, node);
    endif
end

The intersection point that we are looking for will appear as the starting point of the first segment of the span.

6.2 Back-face culling

It will be assumed in this and all the consecutive sections of this chapter that objects are transformed into the screen coordinate system, and that in the case of perspective projection the homogeneous division has also been performed. This means that objects have to be projected orthographically onto the image plane spanned by the coordinate axes X, Y, and the coordinate axis Z coincides with the direction of view.

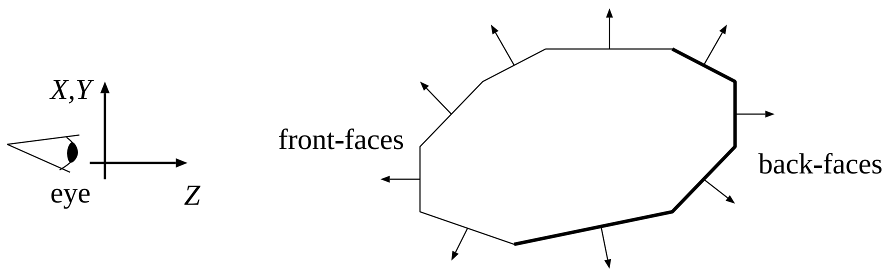

Figure 6.5: Normal vectors and back-faces

A usual agreement is, furthermore, that the normal vector at any object surface point (the normal vector of the tangent plane at that point) is defined so that it always points *outwards* from the object, as illustrated in figure 6.5. What can be stated about a surface point where the surface normal vector has a positive Z-coordinate (in the screen coordinate system)? It is definitely hidden from the eye since no light can depart from that point towards the eye! Roughly one half of the object surfaces is hidden because of this reason — and independently from other objects —, hence it is worth eliminating them from visibility calculations in advance. Object surfaces are usually decomposed into smaller parts called *faces*. If the normal vector at each point of a face has a positive Z-coordinate then it is called a **back-face** (see figure 6.5).

If a face is planar, then it has a unique normal vector, and the **back-face culling** (deciding whether it is a back-face) is not too expensive computationally. Defining one more convention, the vertices of planar polygonal faces can be numbered in counter-clockwise order, for example, looking from outside the object. If the vertices of this polygon appear in clockwise order on the image plane then the polygon is a back-face. How can it be detected? If $\vec{r}_1, \vec{r}_2, \vec{r}_3$ are three consecutive and non-collinear vertices of the polygon, then its normal vector, $\vec{n}$, can be calculated as:

$$\vec{n} = (-1)^c \cdot (\vec{r}_2 - \vec{r}_1) \times (\vec{r}_3 - \vec{r}_1) \tag{6.45}$$

where $c = 0$ if the inner angle at vertex $\vec{r}_2$ is less than π and $c = 1$ otherwise. If the Z-coordinate of $\vec{n}$ is positive, then the polygon is a back-face and can be discarded. If it is zero, then the projection of the polygon degenerates to a line segment and can also be discarded. A more tricky way of computing $\vec{n}$ is calculating the projected areas A_x, A_y, A_z of the polygon onto the planes perpendicular to the x-, y- and z-axes, respectively, and then taking $\vec{n}$ as the vector of components A_x, A_y, A_z. If the polygon vertices are given by the coordinates $(x_1, y_1, z_1), \ldots, (x_m, y_m, z_m)$ then the projected area A_z, for example, can be calculated as:

$$A_z = \frac{1}{2}\sum_{i=1}^{m}(x_{i\oplus 1} - x_i)(y_i + y_{i\oplus 1}) \tag{6.46}$$

where $i \oplus 1 = i + 1$ if $i < m$ and $m \oplus 1 = 1$. This method is not sensitive to collinear vertices and averages the errors coming from possible non-planarity of the polygon.

Note that if the object scene consists of nothing more than a single convex polyhedron, then the visibility problem can completely be solved by back-face culling: back-faces are discarded and non-back-faces are painted.

6.3 z-buffer algorithm

Another possible method for finding the visible object in individual pixels is that, for each object, all the pixels forming the image of the object on the screen are identified, and then, if a collision occurs at a given pixel due to overlapping, it is decided which object must be retained. The objects are taken one by one. To generate all the pixels that the projection of an object covers, **scan conversion** methods can be used to convert the area of the projections first into (horizontal) spans corresponding to the rows of the raster image, and then split up the spans into pixels according to the columns. Imagine another array behind the raster image (raster buffer), with the same dimensions, but containing distance information instead of color values. This array is called **z-buffer**. Each pixel in the raster buffer has a corresponding cell in the z-buffer. This contains the distance (depth) information of the surface point from the eye which is used to decide which pixel is visible. Whenever a new color value is to be written into a pixel during the raster conversion of the objects, the value already in the z-buffer is compared with that of the actual surface point. If the value in the z-buffer is greater, then the pixel can be overwritten, both the corresponding color and depth information, because the actual surface point is closer to the eye. Otherwise the values are left untouched.

The basic form of the z-buffer algorithm is then:

> Initialize raster buffer to background color;
> Initialize each cell of zbuffer[] to ∞;
> **for** each object o **do**
>   **for** each pixel p covered by the projection of o **do**
>   **if** Z-coordinate of the surface point $<$ zbuffer[p] **then**
>   color of p = color of surface point;
>   zbuffer[p] = depth of surface point;
>   **endif**
>   **endfor**
> **endfor**

The value ∞ loaded into each cell of the z-buffer in the initialization step symbolizes the greatest possible Z value that can occur during the visibility calculations, and it is always a finite number in practice. This is also an image-precision algorithm, just like ray tracing. Its effectiveness can be — and usually is — increased by combining it with **back-face culling**.

The z-buffer algorithm is not expensive computationally. Each object is taken only once, and the number of operations performed on one object is proportional to the number of pixels it covers on the image plane. Having N objects $o_1, \ldots, o_N$, each covering P_i number of pixels individually on the image plane, the time complexity T of the z-buffer algorithm is:

$$T = O\left(N + \sum_{i=1}^{N} P_i\right). \tag{6.47}$$

Since the z-buffer algorithm is usually preceded by a clipping operation discarding parts of objects outside the viewing volume, the number of pixels covered by the input objects $o_1, \ldots, o_N$ is $P_i = O(R^2)$ (R^2 is the resolution of the screen), and hence the time complexity of the z-buffer algorithm can also be written as:

$$T = O(R^2 N). \tag{6.48}$$

6.3.1 Hardware implementation of the z-buffer algorithm

Having approximated the surface by a polygon mesh, the surface is given by the set of mesh vertices, which should have been transformed to the screen coordinate system. Without loss of generality, we can assume that the polygon mesh consists of triangles only (this assumption has the important advantage that three points are always on a plane and the triangle formed by the points is convex). The visibility calculation of a surface is thus a series of visibility computations for screen coordinate system triangles, allowing us to consider only the problem of the scan conversion of a single triangle. Let the vertices of the triangle in screen coordinates be $\vec{r}_1 = [X_1, Y_1, Z_1]$, $\vec{r}_2 = [X_2, Y_2, Z_2]$ and $\vec{r}_3 = [X_3, Y_3, Z_3]$ respectively. The scan conversion algorithms should determine the X, Y pixel addresses and the corresponding Z coordinates of those pixels which belong to this triangle (figure 6.6). If the X, Y pixel addresses are already available, then the calculation of the corresponding Z coordinate can exploit the fact that the triangle is on a plane, thus the Z coordinate is some linear function of the X, Y coordinates. This linear function can be derived from the equation of the plane, using the notation $\vec{n}$ and $\vec{r}$ to represent the normal vector and the points of the plane respectively:

$$\vec{n} \cdot \vec{r} = \vec{n} \cdot \vec{r}_1 \quad \text{where} \quad \vec{n} = (\vec{r}_2 - \vec{r}_1) \times (\vec{r}_3 - \vec{r}_1). \tag{6.49}$$

Let us denote the constant $\vec{n} \cdot \vec{r}_1$ by C, and express the equation in scalar form, substituting the coordinates of the vertices ($\vec{r} = [X, Y, Z(X, Y)]$) and

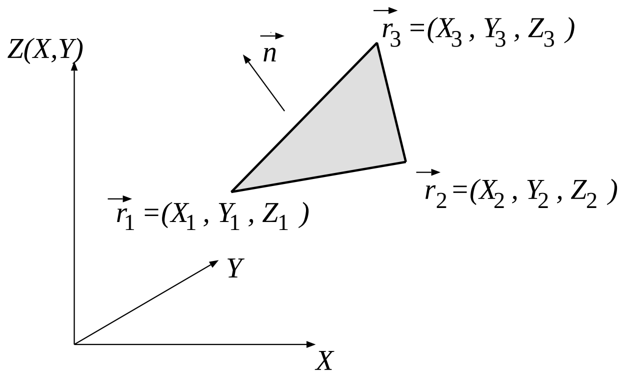

Figure 6.6: Screen space triangle

the normal of the plane ($\vec{n} = [n_X, n_Y, n_Z]$). The function of $Z(X,Y)$ is then:

$$Z(X,Y) = \frac{C - n_X \cdot X - n_Y \cdot Y}{n_Z}. \tag{6.50}$$

This linear function must be evaluated for those pixels which cover the pixel space triangle defined by the vertices $[X_1, Y_1]$, $[X_2, Y_2]$ and $[X_3, Y_3]$. Equation 6.50 is suitable for the application of the incremental concept discussed in subsection 2.3.2 on multi-variate functions. In order to make the boundary curve differentiable and simple to compute, the triangle is split into two parts by a horizontal line at the position of the vertex which is in between the other two vertices in the Y direction.

As can be seen in figure 6.7, two different orientations (called left and right orientations respectively) are possible, in addition to the different order of the vertices in the Y direction. Since the different cases require almost similar solutions, we shall discuss only the scan conversion of the lower part of a left oriented triangle, supposing that the Y order of the vertices is: $Y_1 < Y_2 < Y_3$.

The solution of the subsection 2.3.2 (on multi-variate functions) can readily be applied for the scan conversion of this part. The computational burden for the evaluation of the linear expression of the Z coordinate and for the calculation of the starting and ending coordinates of the horizontal spans of pixels covering the triangle can be significantly reduced by the incremental concept (figure 6.8).

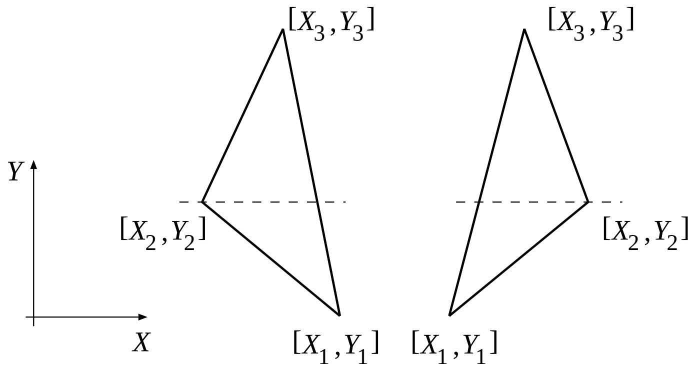

Figure 6.7: Breaking down the triangle

Expressing $Z(X+1,Y)$ as a function of $Z(X,Y)$, we get:

$$Z(X+1,Y) = Z(X,Y) + \frac{\partial Z(X,Y)}{\partial X} \cdot 1 = Z(X,Y) - \frac{n_X}{n_Z} = Z(X,Y) + \delta Z_X. \tag{6.51}$$

Since δZ_X does not depend on the actual X, Y coordinates, it has to be evaluated once for the polygon. In a scan-line, the calculation of a Z coordinate requires a single addition according to equation 6.51.

Since Z and X vary linearly along the left and right edges of the triangle, equations 2.33, 2.34 and 2.35 result in the following simple expressions in the range of $Y_1 \leq Y \leq Y_2$, denoting the K_s and K_e variables used in the general discussion by X_{start} and X_{end} respectively:

$$\begin{aligned}
X_{\text{start}}(Y+1) &= X_{\text{start}}(Y) + \frac{X_2 - X_1}{Y_2 - Y_1} = X_{\text{start}}(Y) + \delta X_Y^s \\
X_{\text{end}}(Y+1) &= X_{\text{end}}(Y) + \frac{X_3 - X_1}{Y_3 - Y_1} = X_{\text{end}}(Y) + \delta X_Y^e \\
Z_{\text{start}}(Y+1) &= Z_{\text{start}}(Y) + \frac{Z_2 - Z_1}{Y_2 - Y_1} = Z_{\text{start}}(Y) + \delta Z_Y^s
\end{aligned} \tag{6.52}$$

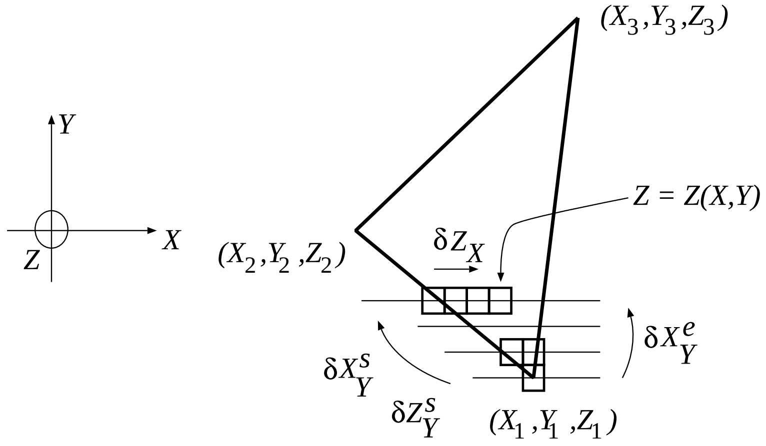

Figure 6.8: Incremental concept in Z-buffer calculations

The complete incremental algorithm is then:

$X_{\text{start}} = X_1 + 0.5$; $X_{\text{end}} = X_1 + 0.5$; $Z_{\text{start}} = Z_1 + 0.5$;
for $Y = Y_1$ **to** Y_2 **do**
  $Z = Z_{\text{start}}$;
  for $X = \text{Trunc}(X_{\text{start}})$ **to** $\text{Trunc}(X_{\text{end}})$ **do**
    $z = \text{Trunc}(Z)$;
    if $z <$ Zbuffer$[X, Y]$ **then**
      raster_buffer$[X, Y]$ = computed color;
      Zbuffer$[X, Y] = z$;
    endif
    Z += δZ_X;
  endfor
  X_{start} += δX_Y^s; X_{end} += δX_Y^e; Z_{start} += δZ_Y^s;
endfor

Having represented the numbers in a fixed point format, the derivation of the executing hardware for this algorithm is straightforward following the methods outlined in section 2.3 on hardware realization of graphics algorithms.

6.4 Scan-line algorithm

The visibility problem can be solved separately for each horizontal row of the image. This approach is a hybrid one, half way between image-precision and object-precision methods. On the one hand, the so-called **scan-lines** are discrete rows of the image, on the other hand, continuous calculations are used at object-precision within the individual scan-lines. Such a horizontal line corresponds to a horizontal plane in the screen coordinate system (see left side of figure 6.9). For each such plane, we have to consider the intersection of the objects with it. This gives two-dimensional objects on the scan plane. If our object space consists of planar polygons, then a set of line segments will appear on the plane. Those parts of these line segments which are visible from the line $Z = 0$ have to be kept and drawn (see right side of figure 6.9). If the endpoints of the segments are ordered by their X coordinate, then the visibility problem is simply a matter of finding the line segment with the minimal Z coordinate in each of the quadrilateral strips between two consecutive X values. If the line segments can intersect, then the X coordinates of the intersection points have also to be inserted into the list of segment endpoints in order to get strips that are homogeneous with respect to visibility, that is, with at most one segment visible in each.

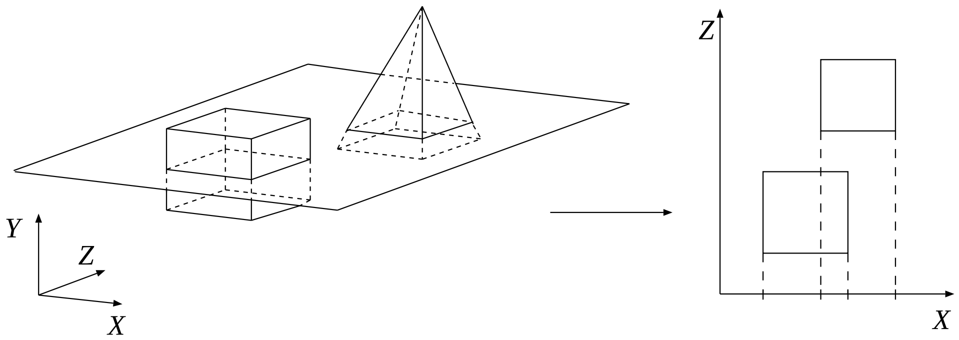

Figure 6.9: Scan-line algorithm

The basic outline of the algorithm is the following:

```
for Y = Y_min to Y_max do
    for each polygon P do
        compute intersection segments between P and plane at Y;
    endfor
    sort endpoints of segments by their x coordinate;
    compute and insert segment-segment intersection points;
    for each strip s between two consecutive x values do
        find segment in s closest to axis x;
        draw segment;
    endfor
endfor
```

If a given polygon intersects the horizontal plane at Y, it will probably intersect the next scan plane at $Y+1$, as well. This is one of the guises of the phenomenon called **object coherence**. The origin of it is the basic fact that objects usually occupy compact and connected parts of space. Object coherence can be exploited in many ways in order to accelerate the calculations. In the case of the scan-line algorithm we can do the following. Before starting the calculation, we sort the maximal and minimal Y values of the polygons into a list called the **event list**. Another list, called the active polygon list, will contain only those polygons which really intersect the horizontal plane at the actual height Y. A Y coordinate on the event list corresponds either to the event of a new polygon being inserted into the active polygon list, or to the event of a polygon being deleted from it. These two lists will then be considered when going through the consecutive Y values in the outermost loop of the above algorithm. This idea can be refined by managing an active edge list (and the corresponding event list) instead of the active polygon list. A further acceleration can be the use of differential line generators for calculating the intersection point of a given segment with the plane at $Y+1$ if the point at Y is known.

The time complexity of the algorithm in its "brute-force" form, as sketched above, is proportional to the number of rows in the picture on the one hand, and to the number of objects on the other hand. If the resolution of the screen is R^2, and the object scene consists of disjoint polygons having a total of n edges, then:

$$T = O(R \cdot n). \tag{6.53}$$

If the proposed event list is used, and consecutive intersection points (the X values at $Y+1$) are computed by differential line generators, then the time complexity is reduced:

$$T = O(n \log n + R \log n). \tag{6.54}$$

The $O(n \log n)$ term appears because of the sorting step before building the event list, the origin of the $O(R \log n)$ term is that the calculated intersection points must be inserted into an ordered list of length $O(n)$.

6.5 Area subdivision methods

If a pixel of the image corresponds to a given object, then its neighbors usually correspond to the same object, that is, visible parts of objects appear as connected territories on the screen. This is a consequence of object coherence and is called **image coherence**.

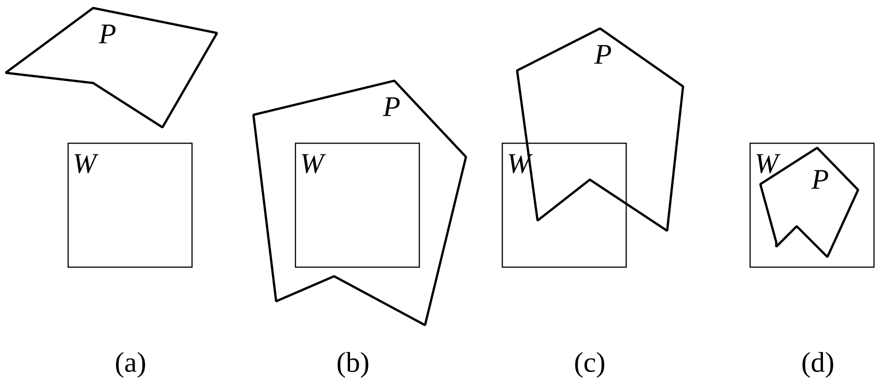

Figure 6.10: Polygon-window relations: distinct (a), surrounding (b), intersecting (c), contained (d)

If the situation is so fortunate — from a labor saving point of view — that a polygon in the object scene obscures all the others and its projection onto the image plane covers the image window completely, then we have to do no more than simply fill the image with the color of the polygon. This is the basic idea of **Warnock's algorithm** [War69]. If no polygon edge falls into the window, then either there is no visible polygon, or some polygon covers it completely. The window is filled with the background color in the first case, and with the color of the closest polygon in the second case. If at least one polygon edge falls into the window, then the solution is not so simple. In this case, using a **divide-and-conquer** approach, the window is subdivided into four quarters, and each subwindow is searched recursively for a simple solution. The basic form of the algorithm rendering a rectangular window with screen (pixel) coordinates X_1, Y_1 (lower left corner) and X_2, Y_2 (upper right corner) is this:

```
Warnock(X1, Y1, X2, Y2)
        if X1 ≠ X2 or Y1 ≠ Y2 then
           if at least one edge falls into the window then
              Xm = (X1 + X2)/2;
              Ym = (Y1 + Y2)/2;
              Warnock(X1, Y1, Xm, Ym);
              Warnock(X1, Ym, Xm, Y2);
              Warnock(Xm, Y1, X2, Ym);
              Warnock(Xm, Ym, X2, Y2);
              return ;
           endif
        endif
        // rectangle X1, Y1, X2, Y2 is homogeneous
        polygon = nearest to pixel (X1 + X2)/2, (Y1 + Y2)/2;
        if no polygon then
           fill rectangle X1, Y1, X2, Y2 with background color;
        else
           fill rectangle X1, Y1, X2, Y2 with color of polygon;
        endif
end
```

It falls into the category of image-precision algorithms. Note that it can handle non-intersecting polygons only. The algorithm can be accelerated by filtering out those polygons which can definitely not be seen in a given subwindow at a given step. Generally, a polygon can be in one of the following four kinds of relation with respect to the window, as shown in figure 6.10. A *distinct* polygon has no common part with the window; a *surrounding* polygon contains the window; at least one edge of an *intersecting polygon* intersects the border of the window; and a *contained* polygon falls completely within the window. Distinct polygons should be filtered out at each step of recurrence. Furthermore, if a surrounding polygon appears at a given stage, then all the others behind it can be discarded, that is all those which fall onto the opposite side of it from the eye. Finally, if there is only one contained or intersecting polygon, then the window does not have to be subdivided further, but the polygon (or rather the clipped part of it) is simply drawn. The price of saving further recurrence is the use of a scan-conversion algorithm to fill the polygon.

The time complexity of the Warnock algorithm is not easy to analyze, even for its initial form (sketched above). It is strongly affected by the actual arrangement of the polygons. It is easy to imagine a scene where each image pixel is intersected by at least one (projected) edge, from where the algorithm would go down to the pixel level at each recurrence. It gives a very poor worst-case characteristic to the algorithm, which is not worth demonstrating here. A better characterization would be an average-case analysis for some proper distribution of input polygons, which again length constraints of this book do not permit us to explore.

The Warnock algorithm recursively subdivides the screen into rectangular regions, irrespective of the actual shape of the polygons. It introduces superfluous vertical and horizontal edges. Weiler and Atherton [WA77] (also in [JGMHe88]) refined Warnock's idea from this point of view. The **Weiler–Atherton** algorithm also subdivides the image area recursively, but using the boundaries of the actual polygons instead of rectangles. The calculations begin with a rough **initial depth sort**. It puts the list of input polygons into a rough depth priority order, so that the "closest" polygons are in the beginning of the list, and the "farthest" ones at the end of it. At this step, any reasonable criterion for a sorting key is acceptable. The resulting order is not at all mandatory but increases the efficiency of the algorithm. Such a sorting criterion can be, for example, the smallest Z-value ($Z_{\min}$) for each polygon (or $Z_{\max}$, as used by the Newell–Newell–Sancha algorithm, see later). This sorting step is performed only once, at the beginning of the calculations, and is not repeated.

Let the resulting list of polygons be denoted by $L = \{P_1, \ldots, P_n\}$. Having done the sorting, the first polygon on the list (P_1) is selected. It is used to clip the remainder of the list into two new lists of polygons: the first list, say $I = \{P_1^I, \ldots, P_m^I\}$ $(m \leq n)$, will contain those polygons — or parts of polygons — that fall *inside* the clip polygon P_1, and the second list, say $O = \{P_1^O, \ldots, P_M^O\}$ $(M \leq n)$, will contain those ones that fall *outside* P_1. Then the algorithm examines the inside list I and removes all polygons located behind the current clip polygon since they are hidden from view. If the remaining list I' contains no polygon (the clip polygon obscures all of I), then the clip polygon is drawn and the initial list L is replaced by the outside list O and examined in a similar way to L. If the remaining list I' contains at least one polygon — that is, at least one polygon falls in front of the clip polygon — then it means that there was an error in the initial rough depth sort. In this case the (closest) offending polygon is selected as the clip polygon, and the same process is performed on list I' recursively, as on the initially ordered list L. Note that although the original polygons may be split into several pieces during the recursive subdivision, the clipping step (generating the lists I and O from L) can always be performed by using the original polygon corresponding to the actual clip polygon (which itself may be a clipped part of the original polygon). Maintaining a copy of each original polygon needs extra storage, but it reduces time.

There is, however, a more serious danger of clipping to the original copy of the polygons instead of their remainders! If there is *cyclic overlapping* between the original polygons, see figure 6.11 for example, then it can cause infinite recurrence of the algorithm. In order to avoid this, a set S of polygon names (references) is maintained during the process. Whenever a polygon P is selected as the clip polygon, its name (a reference to it) is inserted into S, and if it is processed (drawn or removed), its name is deleted from S. The insertion is done, however, only if P is not already in S, because if it is, then a cyclic overlap has been detected, and no additional recurrence is necessary because all polygons behind P have already been removed.

There is another crucial point of the algorithm: even if the scene consists only of convex polygons, the clipping step can quickly yield non-convex areas and holes (first when producing an outside list and then concavity is inherited by polygons in the later inside lists, as well). Thus, the polygon clipper has to be capable of clipping concave polygons with holes to both the inside and outside of a concave polygon with holes. Without going into further details here, the interested reader is referred to the cited work [WA77], and only the above sketched ideas are summarized in the following pseudo-code:

```
WeilerAtherton(L)
        P = the first item on L;
        if P ∈ S then draw P; return ; endif
        insert P into S;
        I = Clip(L, P);
        O = Clip(L, P̄);                          // P̄: complement of P
        for each polygon Q ∈ I;
            if Q is behind P then
                remove Q from I;
                if Q ∈ S then remove Q from S; endif
            endif
        endfor
        if I = {} then
            draw P;
            delete P from S;
        else
            WeilerAtherton(I);
        endif
        WeilerAtherton(O);
end
```

The recursive algorithm is called with the initially sorted list L of input polygons at the "top" level after initializing the set S to $\{\}$.

6.6 List-priority methods

Assume that the object space consists of planar polygons. If we simply scan convert them into pixels and draw the pixels onto the screen without any examination of distances from the eye, then each pixel will contain the color of the last polygon falling onto that pixel. If the polygons were ordered by their distance from the eye, and we took the farthest one first and the closest one last, then the final picture would be correct. Closer polygons would obscure farther ones — just as if they were painted an opaque color. This (object-precision) method, is really known as the **painter's algorithm**.

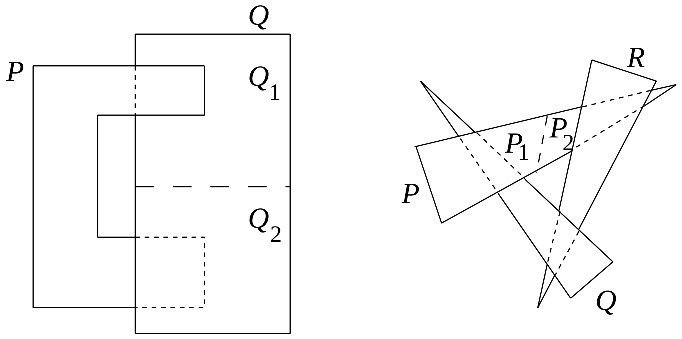

Figure 6.11: Examples for cyclic overlapping

The only problem is that the order of the polygons necessary for performing the painter's algorithm, the so-called **depth order** or **priority** relation between the polygons, is not always simple to compute.

We say that a polygon P *obscures* another polygon Q, if at least one point of Q is obscured by P. Let us define the relation $\succ$ between two polygons P and Q so that $P \succ Q$ if Q *does not obscure* P. If the relations $P_1 \succ P_2 \succ \ldots \succ P_n$ hold for a sequence of polygons, then this order coincides with the priority order required by the painter's algorithm. Indeed, if we drew the polygons by starting with the one furthest to the right (having the lowest priority) and finishing with the one furthest to the left, then the picture would be correct. However, we have to contend with the following problems with the relation $\succ$ defined this way:

1. If the projection of polygons P and Q do not overlap on the image plane, then $P \succ Q$ and $P \prec Q$, both at the same time, that is, the relation $\succ$ is *not antisymmetric*.

2. Many situations can be imagined, when $P \not\succ Q$ and $Q \not\succ P$ at the same time (see figure 6.11 for an example), that is, the relation $\succ$ is not defined for each pair of polygons.

3. Many situations can be imagined when a cycle $P \succ Q \succ R \succ P$ occurs (see figure 6.11 again), that is, the relation $\succ$ is *not transitive*.

The above facts prevent the relation $\succ$ from being an ordering relation, that is, the depth order is generally impossible to compute (at least if the polygons are not allowed to be cut). The first problem is not a real problem since polygons that do not overlap on the image plane can be painted in any order. What the second and third problems have in common is that both of them are caused by *cyclic overlapping* on the image plane. Cycles can be resolved by properly cutting some of the polygons, as shown by dashed lines in figure 6.11. Having cut the "problematic" polygons, the relation between resulting polygons will be cycle-free (transitive), that is $Q_2 \succ P \succ Q_1$ and $P_1 \succ Q \succ R \succ P_2$ respectively.

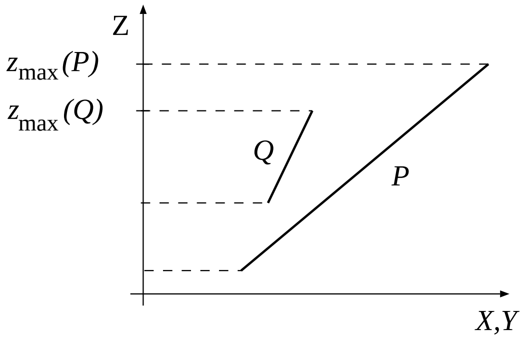

Figure 6.12: A situation when $z_{\max}(P) > z_{\max}(Q)$ yet $P \not\succ Q$

The **Newell–Newell–Sancha** algorithm [NNS72], [NS79] is one approach for exploiting the ideas sketched above. The first step is the calculation of an **initial depth order**. This is done by sorting the polygons according to their maximal z value, $z_{\max}$, into a list L. If there are no two polygons whose z ranges overlap, the resulting list will reflect the correct depth order. Otherwise, and this is the general case except for very special scenes such as those consisting of polygons all perpendicular to the z direction, the calculations need more care. Let us first take the polygon P which is the last item on the resulting list. If the z range of P does not overlap with any of the preceding polygons, then P is correctly positioned, and the polygon preceding P can be taken instead of P for a similar examination. Otherwise (and this is the general case) P overlaps a set $\{Q_1, \ldots, Q_m\}$ of polygons. This set can be found by scanning L from P backwards and taking the consecutive polygons Q while $z_{\max}(Q) > z_{\min}(P)$. The next step is to try to check that P *does not* obscure any of the polygons in $\{Q_1, \ldots, Q_m\}$, that is, that P is at its right position despite the overlapping. A polygon P does not obscure another polygon Q, that is $Q \succ P$, if any of the following conditions holds:

1. $z_{\min}(P) > z_{\max}(Q)$ (they do not overlap in z range, this is the so-called z minimax check);
2. the bounding rectangle of P on the x, y plane does not overlap with that of Q (x, y minimax check);
3. each vertex of P is farther from the viewpoint than the plane containing Q;
4. each vertex of Q is closer to the viewpoint than the plane containing P;
5. the projections of P and Q do not overlap on the x, y plane.

The order of the conditions reflects the complexity of the check, hence it is worth following this order in practice. If it turns out that P obscures Q ($Q \not\succ P$) for a polygon in the set $\{Q_1, \ldots, Q_m\}$, then Q has to be moved behind P in L. This situation is illustrated in figure 6.12. Naturally, if P intersects Q, then one of them has to be cut into two parts by the plane of the other one. Cycles can also be resolved by cutting. In order to accomplish this, whenever a polygon is moved to another position in L, we mark it. If a marked polygon Q is about to be moved again because, say $Q \not\succ P$, then — assuming that Q is a part of a cycle — Q is cut into two pieces Q_1, Q_2, so that $Q_1 \not\succ P$ and $Q_2 \succ P$, and only Q_1 is moved behind P. A proper cutting plane is the plane of P, as illustrated in figure 6.11.

Considering the Newell–Newell–Sancha algorithm, the following observation is worth mentioning here. For any polygon P, let us examine the two halfspaces, say H_P^+ and H_P^-, determined by the plane containing P. If the viewing position is in H_P^+, then for all $p \in H_P^+$, P *cannot obscure* p, and for all $p \in H_P^-$, p cannot obscure P. On the other hand, if the viewing position is contained by H_P^-, similar observations can be made with the roles of H_P^+ and H_P^- interchanged. A complete algorithm for computing the depth order of a set $S = \{P_1, \ldots, P_n\}$ of polygons can be constructed based on this idea, as proposed by Fuchs *et al.* [FKN80]. First P_i, one of the polygons, is selected. Then the following two sets are computed:

$$S_i^+ = (S \setminus P_i) \cap H_i^+, \quad S_i^- = (S \setminus P_i) \cap H_i^-, \quad (|S_i^+|, |S_i^-| \leq |S| - 1 = n - 1). \tag{6.55}$$

Note that some (if not all) polygons may be cut into two parts during the construction of the sets. If the viewing point is in H_i^+, then P_i cannot obscure any of the polygons in S_i^+, and no polygon in S_i^- can obstruct P_i. If the viewing point is in H_i^-, then the case is analogous with the roles of S_i^+ and S_i^- interchanged. That is, the position of P_i in the depth order is *between* those of the polygons in S_i^+ and S_i^-. The depth order in S_i^+ and S_i^- can then be recursively computed: a polygon P_j is selected from S_i^+ and the two sets S_j^+, S_j^- are created, and a polygon P_k is selected from S_i^- and the two sets S_k^+, S_k^- are created, etc. The subdivision is continued until the resulting set $S_\cdot^\cdot$ contains not more than one polygon (the depth order is then obvious in $S_\cdot^\cdot$; the dots in the subscript and superscript places stand for any possible value). This stop condition will definitely hold, since the size of both resultant sets $S_\cdot^+, S_\cdot^-$ is always at least one smaller than that of $S_\cdot^\cdot$, from which they are created (cf. equation 6.55).

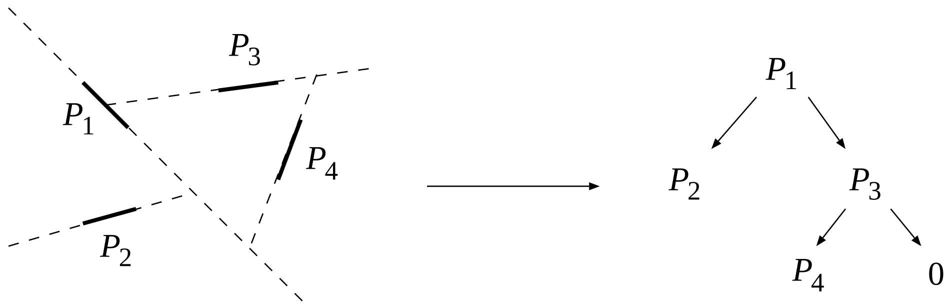

Figure 6.13: A binary space partitioning and its BSP-tree representation

The creation of the sets induces a subdivision of the object space, the so-called **binary space partitioning** (BSP) as illustrated in figure 6.13: the first plane divides the space into two halfspaces, the second plane divides the first halfspace, the third plane divides the second halfspace, further planes split the resulting volumes, etc. The subdivision can well be represented by a binary tree, the so-called **BSP-tree**, also illustrated in figure 6.13: the first plane is associated with the root node, the second and third planes are associated with the two children of the root, etc. For our application, not so much the planes, but rather the polygons defining them, will be assigned to the nodes of the tree, and the set S of polygons contained by the volume is also necessarily associated with each node. Each leaf node will then contain either no polygon or one polygon in the associated set S (and no partitioning plane, since it has no child). The algorithm for creating the BSP-tree for a set S of polygons can be the following, where $S(N), P(N), L(N)$ and $R(N)$ denote the set of polygons, the "cutting" polygon and the left and right children respectively, associated with a node N:

BSPTree(S)
  create a new node N;
  $S(N) = S$;
  if $|S| \leq 1$ **then**
    $P(N) = null$; $L(N) = null$; $R(N) = null$;
  else
    $P =$ **Select**(S); $P(N) = P$;
    create sets S_P^+ and S_P^-;
    $L(N) =$ **BSPTree**(S_P^+);
    $R(N) =$ **BSPTree**(S_P^-);
  endif
  return N;
end

The size of the BSP-tree, that is, the number of polygons stored in it, is on the one hand highly dependent on the nature of the object scene, and on the other hand on the "choice strategy" used by the routine **Select**. We can affect only the latter. The creators of the algorithm also proposed a heuristic choice criterion (without a formal proof) [FKN80], [JGMHe88] for minimizing the number of polygons in the BSP-tree. The strategy is two-fold: it minimizes the number of polygons that are split, and at the same time tries to maximize the number of "polygon conflicts" eliminated by the choice. Two polygons are in conflict if they are in the same set, and the plane of one polygon intersects the other polygon. What hoped for when maximizing the elimination of polygon conflicts is that the number of polygons which will need to be split in the descendent subtrees can be reduced. In order to accomplish this, the following three sets are associated with each polygon P in the actual (to-be-split) set S:

$$\begin{aligned} S_1 &= \left\{Q \in S \mid Q \text{ is entirely in } H_P^+\right\}, \\ S_2 &= \{Q \in S \mid Q \text{ is intersected by the plane of } P\}, \\ S_3 &= \left\{Q \in S \mid Q \text{ is entirely in } H_P^-\right\}. \end{aligned} \tag{6.56}$$

Furthermore, the following functions are defined:

$$\begin{aligned} f(P,Q) &= \begin{cases} 1, & \text{if the plane of } P \text{ intersects } Q; \\ 0, & \text{otherwise;} \end{cases} \\ I_{i,j} &= \sum_{P \in S_i} \sum_{Q \in S_j} f(P,Q) \end{aligned} \tag{6.57}$$

Then the routine **Select**(S) will return that polygon $P \in S$, for which the expression $I_{1,3} + I_{3,1} + w \cdot |S_2|$ is maximal, where w is a weight factor. The actual value of the weight factor w can be set based on practical experiments.

Note that the BSP-tree computed by the algorithm is *view-independent*, that is it contains the proper depth order for any viewing position. Differences caused by different viewing positions will appear in the manner of traversing the tree for retrieving the actual depth order. Following the characteristics of the BSP-tree, the traversal will always be an **inorder traversal**. Supposing that some action is to be performed on each node of a binary tree, the inorder traversal means that for each node, first one of its children is traversed (recursively), then the action is performed on the node, and finally the other child is traversed. This is in contrast to what happens with **preorder** or **postorder** traversals, where the action is performed before or after traversing the children respectively. The action for each node N here is the drawing of the polygon $P(N)$ associated with it. If the viewing position is in $H_{P(N)}^+$, then first the right subtree is drawn, then the polygon $P(N)$, and finally the left subtree, otherwise the order of the left and right children is back to front.

The following algorithm draws the polygons of a BSP-tree N in their proper depth order:

BSPDraw(N)
  if N is empty **then return** ;
  if the viewing position is in $H^+_{P(N)}$ **then**
    BSPDraw($R(N)$); **Draw**($P(N)$); **BSPDraw**($L(N)$);
  else
    BSPDraw($L(N)$); **Draw**($P(N)$); **BSPDraw**($R(N)$);
  endif
end

Once the BSP-tree has been created by the algorithm **BSPTree**, subsequent images for subsequent viewing positions can be generated by subsequent calls to the algorithm **BSPDraw**.

6.7 Planar graph based algorithms

A **graph** G is a pair $G(V, E)$ in its most general form, where V is the set of vertices or nodes, and E is the set of edges or arcs, each connecting two nodes. A graph is planar if it can be drawn onto the plane so that no two arcs cross each other. A **straight line planar graph (SLPG)** is a concrete embedding of a planar graph in the plane where all the arcs are mapped to (non-crossing) straight line segments. Provided that the graph is connected, the "empty" regions surrounded by an alternating chain of vertices and edges, and containing no more of them in the interior, are called faces. (Some aspects of these concepts were introduced briefly in section 1.6.2 on B-rep modeling.)

One of the characteristics of image coherence is that visible parts of objects appear as connected territories on the screen. If we have calculated these territories exactly, then we have only to paint each of them with the color of the corresponding object. Note that although the calculations are made on the image plane, this is an object-precision approach, because the accuracy of the result — at least in the first step — does not depend on the resolution of the final image. If the object scene consists of planar polygons, then the graph of visible parts will be a straight line planar graph, also called the **visibility map** of the objects on the image plane. Its nodes and arcs correspond to the vertices and edges of polygons and intersections between polygons, and the faces represent homogeneous visible parts. We use the terms nodes and arcs of G in order to distinguish them from the vertices and edges of the polyhedra in the scene.

Let us assume in this section that the polygons of the scene do not intersect, except in cases when two or more of them share a common edge or vertex. This assumption makes the treatment easier, and it is still general enough, because scenes consisting of disjoint polyhedra fall into this category. The interested reader is recommended to study the very recent work of Mark de Berg [dB92], where the proposed algorithms can handle scenes of arbitrary (possibly intersecting) polygons. A consequence of our assumption is that the set of projected edges of the polygons is a superset of the set of edges contained in the visibility map. This is not so for the vertices, because a new vertex can occur on the image plane if a polygon partially obscures an edge. But the set of such new vertices is contained in the set of all intersection points between the projected edges. Thus we can first project all the polygon vertices and edges onto the image plane, then determine all the intersection points between the projected edges, and finally determine the parts that remain visible.

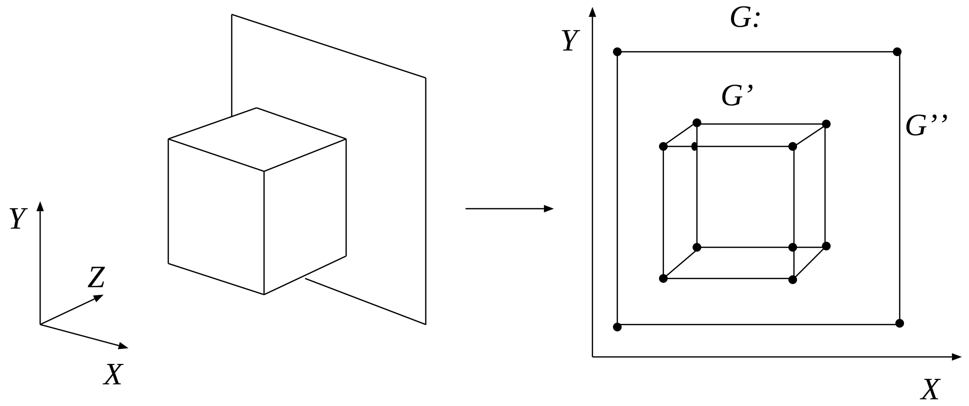

Figure 6.14: Example scene and the corresponding planar subdivision

In actual fact what we will do is to compute the graph G corresponding to the subdivision of the image plane induced by the projected vertices, edges and the intersection between the edges. This graph will not be exactly the visibility map as we defined above, but will possess the property that the visibility will not change within the regions of the subdivision (that is the faces of the graph). Once we have computed the graph G, then all we have to do is visit its regions one by one, and for each region, we select the polygon closest to the image plane and use its color to paint the region. Thus the draft of the drawing algorithm for rendering a set $P_1, \ldots, P_N$ of polygons is the following:

1. project vertices and edges of $P_1, \ldots, P_N$ onto image plane;
2. calculate all intersection points between projected edges;
3. compute G, the graph of the induced planar subdivision;
4. **for** each region R of G **do**
5. P = the polygon visible in R;
6.   **for** each pixel p covered by R **do**
7.   color of p = color of P;
8.   **endfor**
9. **endfor**

The speed of the algorithm is considerably affected by how well its steps are implemented. In fact, all of them are critical, except for steps 1 and 7. A simplistic implementation of step 2, for example, would test each pair of edges for possible intersection. If the total number of edges is n, then the time complexity of this calculation would be $O(n^2)$. Having calculated the intersection points, the structure of the subdivision graph G has to be built, that is, incident nodes and arcs have to be assigned to each other somehow. The number of intersection points is $O(n^2)$, hence both the number of nodes and the number of arcs fall into this order. A simplistic implementation of step 3 would search for the possible incident arcs for each node, giving a time complexity of $O(n^4)$. This itself is inadmissible in practice, not to mention the possible time complexity of the further steps. (This was a simplistic analysis of simplistic approaches.)

We will take the steps of the visibility algorithm sketched above one by one, and also give a worst-case analysis of the complexity of the solution used. The approach and techniques used in the solutions are taken from [Dév93].

Representing straight line planar graphs

First of all, we have to devote some time to a consideration of what data structures can be used for representing a straight line planar graph, say $G(V, E)$. If the "topology" of the graph is known, then the location of the vertices determines unambiguously all other geometric characteristics of the graph. But if we intend to manipulate a graph quickly, then the matter of "topological" representation is crucial, and it may well be useful to include some geometric information too. Let us examine two examples where the different methods of representation allow different types of manipulations to be performed quickly.

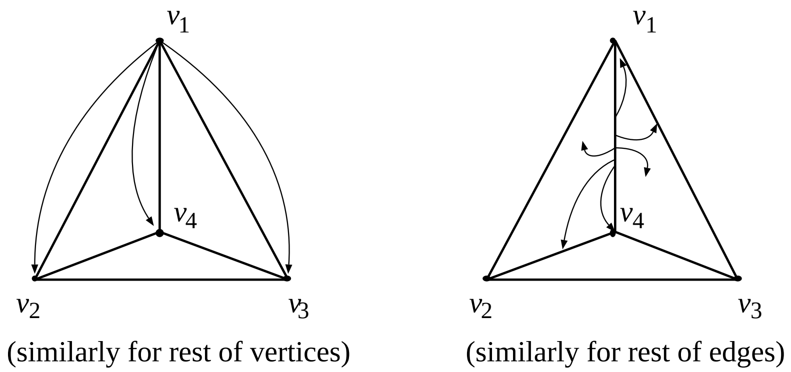

Figure 6.15: Adjacency lists and doubly connected edge list

The first scheme stores the structure by means of **adjacency lists**. Each vertex $v \in V$ has an adjacency list associated with it, which contains a reference to another vertex w, if there is an edge from v to w, that is $(v, w) \in E$. This is illustrated in figure 6.15. In the case of undirected graphs, each edge is stored twice, once at each of its endpoints. If we would like to "walk along" the boundary of a face easily (that is retrieve its boundary vertices and edges), for instance, then it is worth storing some extra information beyond that of the position of the vertices, namely the order of the adjacent vertices w around v. If adjacent vertices appear in counter clockwise order, for example, on the adjacency lists then walking around a face is easily achievable. Suppose that we start from a given vertex v of the face, and we know that the edge (v, w) is an edge of the face with the face falling onto the right-hand side of it, where w is one of the vertices on the adjacency list of v. Then we search for the position of v on the adjacency list of w, and take the vertex next to v on this list as w', and w as v'. The edge (v', w') will be the edge next to (v, w) on the boundary of the face, still having the face on its right-hand side. Then we examine (v', w') in the same way as we did with (v, w), and step on, etc. We stop the walk once we reach our original (v, w) again. This walk would have been very complicated to perform without having stored the order of the adjacent vertices.

An alternative way of representing a straight line planar graph is the use of **doubly connected edge lists (DCELs)**, also shown in figure 6.15. The basic entity is now the edge. Each edge e has two vertex references, $v_1(e)$ and $v_2(e)$, to its endpoints, two edge references, $e_1(e)$ and $e_2(e)$, to the next edge (in counter clockwise order, for instance) around its two endpoints $v_1(e)$ and $v_2(e)$, and two face references, $f_1(e)$ and $f_2(e)$, to the faces sharing e. This type of representation is useful if the faces of the graph carry some specific information (for example: which polygon of the scene is visible in that region). It also makes it possible to traverse all the faces of the graph. The chain of boundary edges of a face can be easily retrieved from the edge references $e_1(e)$ and $e_2(e)$. This fact will be exploited by the following algorithm, which traverses the faces of a graph, and performs an action on each face f by calling a routine **Action**(f). It is assumed that each face has an associated mark field, which is initialized to *non-traversed*. The algorithm can be called with any edge e and one of its two neighboring faces f ($f = f_1(e)$ or $f = f_2(e)$).

```
Traverse(e, f)
    if f is marked as traversed then return ; endif
    Action(f); mark f as traversed;
    for each edge e' on the boundary of f do
        if f1(e') = f then Traverse(e', f2(e'));
        else               Traverse(e', f1(e'));
    endfor
end
```

Note that the algorithm can be used only if the faces of the graph contain no holes — that is the boundary edges of each face form a connected chain, or, what is equivalent, the graph is connected. The running time T of the algorithm is proportional to the number of edges, that is $T = O(|E|)$, because each edge e is taken twice: once when we are on face $f_1(e)$ and again when we are on face $f_2(e)$.

If the graph has more than one connected component as the one shown in figure 6.14, then the treatment needs more care (faces can have holes, for example). In order to handle non-connected and connected graphs in a unified way, some modifications will be made on the DCEL structure. The unbounded part of the plane surrounding the graph will also be considered and represented by a face. Let this special face be called the *surrounding face*. Note that the surrounding face is always multiply connected (if the graph is non-empty), that is it contains at least one hole (in fact the edges of the hole border form the boundary edges of the graph), but has no boundary. We have already defined the structure of an edge of a DCEL structure, but no attention was paid to the structure of a face, although each edge has two explicit references to two faces. A face f will have a reference $e(f)$ to one of its boundary edges. The other boundary edges (except for those of the holes) can be retrieved by stepping through them using the DCEL structure. For the boundary of the holes, f will have the references $h_1(f), \ldots, h_m(f)$, where $m \geq 0$ is the number holes in f, each pointing to one boundary edge of the m different holes. Due to this modification, non-connected graphs will become connected from a computational point of view, and the algorithm **Traverse** will correctly visit all its faces, provided that the enumeration "**for** each edge e' on the boundary of f **do**" implies both the outer and the hole boundary edges. A proper call to visit each face of a possibly multiply connected graph is **Traverse**($h_1(F)$, F), where F is the surrounding face.

Step 1: Projecting the edges

Let the object scene be a set of polyhedra, that is, where the faces of the objects are planar polygons. Assume furthermore that the boundary of the polyhedra (the structure of the vertices, edges and faces) is given by DCEL structures. (The DCEL structure used for boundary representation is known as the **winged edge** data structure for people familiar with shape modeling techniques.) This assumption is important because during the traversal of the computed visibility graph we will enter a new region by crossing one of its boundary edges, and we will have to know the polygon(s) of the object scene the projection of which we leave or enter when crossing the edge on the image plane.

If the total number of edges is n, then the time T_1 required by this step is proportional to the number of edges, that is:

$$T_1 = O(n). \tag{6.58}$$

Step 2: Calculating the intersection points

The second step is the calculation of the intersection points between the projected edges on the image plane. In the worst case the number of intersection points between n line segments can be as high as $O(n^2)$ (imagine, for instance, a grid of $n/2$ horizontal and $n/2$ vertical segments, where each of the horizontal ones intersects each of the vertical ones). In this worst case, therefore, calculation time cannot be better than $O(n^2)$, and an algorithm that compares each segment with all other ones would accomplish the task in optimal worst-case time. The running time of this algorithm would be $O(n^2)$, independently of the real number of intersections. We can create algorithms, however, the running time of which is "not too much" if there are "not too many" intersections. Here we give the draft of such an **output sensitive algorithm**, based on [Dév93] and [BO79]. Let us assume that no three line segments intersect at the same point and all the $2n$ endpoints of the n segments have distinct x-coordinates on the plane, a consequence of the latter being that no segments are vertical. Resolving these assumptions would cause an increase only in the length of the algorithm but not in its asymptotic complexity. See [BO79] for further details. Consider a vertical line $L(x)$ on the plane at a given abscissa x. $L(x)$ may or may not intersect some of our segments, depending on x. The segments $e_1, \ldots, e_k$ intersecting $L(x)$ at points $(x, y_1), \ldots, (x, y_k)$ appear in an ordered sequence if we walk along $L(x)$. A segment e_i is said to be *above* e_j at x if $y_i > y_j$. This relation is a total order for any set of segments intersecting a given vertical line. A necessary condition in order for two segments e_i and e_j to intersect is that there be some x at which e_i and e_j appear as neighbors in the order. All intersection points can be found by sweeping a vertical line in the horizontal direction on the plane and always comparing the neighbors in the order for intersection. The order along $L(x)$ can change when the abscissa x corresponds to one of the following: the left endpoint (beginning) of a segment, the right endpoint (end) of a segment, and/or the intersection point of two segments. Thus our sweep can be implemented by stepping through only these specific positions, called *events*. The following algorithm is based on these ideas, which we can call as the **sweep-line** approach. It maintains a set Q for the event positions, a set R for the intersection points found and a set S for storing the order of segments along $L(x)$ at the actual position. All three sets are ordered, and for set S, succ(s) and prec(s) denote the successor and the predecessor of $s \in S$, respectively.

```
Q = the set of all the 2n segment endpoints;
R = {}; S = {};
sort Q by increasing x-values;
for each point p ∈ Q in increasing x-order do
    if p is the left endpoint of a segment s then
       insert s into S;
       if s intersects succ(s) at any point q then insert q into Q;
       if s intersects prec(s) at any point q then insert q into Q;
    else if p is the right endpoint of a segment s then
       if succ(s) and prec(s) intersect at any point q then
          if q ∉ Q then insert q into Q;
       endif
       delete s from S
    else              // p is the intersection of segments s and t, say
       add p to R; swap s and t in S;                  //say s is above t
       if s intersects succ(s) at any point q then
          if q ∉ Q then insert q into Q;
       endif
       if t intersects prec(t) at any q then
          if q ∉ Q then insert q into Q;
       endif
    endif
endfor
```

Note that the examinations "**if** $q \notin Q$" are really necessary, because the intersection of two segments can be found to occur many times (the appearance and disappearance of another segment between two segments can even occur $n-2$ times!). The first three steps can be performed in $O(n \log n)$

time because of sorting. The main loop is executed exactly $2n + k$ times, where k is the number of intersection points found. The time complexity of one cycle depends on how sophisticated the data structures used for implementing the sets Q and S are, because insertions and deletions have to be performed on them. R is not crucial, a simple array will do. Since the elements of both Q and S have to be in order, an optimal solution is the use of balanced binary trees. Insertions, deletions and searching can be performed in $O(\log N)$ time on a balanced tree storing N elements (see [Knu73], for instance). Now $N = O(n^2)$ for Q and $N = O(n)$ for S, hence $\log N = O(\log n)$ for both. We can conclude that the time complexity of our algorithm for finding the intersection of n line segments in the plane, that is the time T_2 required by step 2 of the visibility algorithm is:

$$T_2 = O((n + k)\log n). \tag{6.59}$$

Such an algorithm is called an **output sensitive** algorithm, because its complexity depends on the actual size of the output. It is generally worth mentioning that if we have a problem with a very bad worst-case complexity due to the possible size of the output, although the usual size of the output is far less, then we have to examine whether an output sensitive algorithm can be constructed.

Step 3: Constructing the subdivision graph G

In step 3 of the proposed visibility algorithm we have to produce the subdivision graph G so that its faces can be traversed efficiently in step 4. A proper representation of G, as we have seen earlier, is a DCEL structure. It will be computed in two steps, first producing an intermediate structure which is then easily converted to a DCEL representation. We can assume that the calculations in steps 1 and 2 have been performed so that all the points — that is the projections of the $2n$ vertices and the k intersection points — have references to the edge(s) they lie on. First of all, for each edge we sort the intersection points lying on it (sorting is done along each edge, individually). Since $O(N \log N)$ time is sufficient (and also necessary) for sorting N numbers, the time consumed by the sorting along an edge e_i is $O(N_i \log N_i)$, where N_i is the number of intersection points to be sorted on e_i. Following from the general relation that if $N_1 + \ldots + N_n = N$, then

$$N_1 \log N_1 + \ldots + N_n \log N_n \leq N_1 \log N + \ldots + N_n \log N = N \log N, \tag{6.60}$$

the sum of the sorting time at the edges is $O(k \log k) = O(k \log n)$, since $N = 2k = O(n^2)$ (one intersection point appears on two segments). Having sorted the points along the edges, we divide the segments into subsegments at the intersection points. Practically speaking this means that the representation of each edge will be transformed into a doubly linked list, illustrated in figure 6.16. Such a list begins with a record describing its starting point.

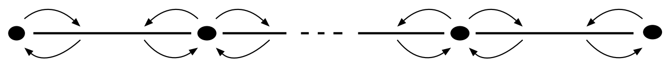

Figure 6.16: Representation of a subdivided segment

It is (doubly) linked to a record describing the first subsegment, which is further linked to its other endpoint, etc. The last element of the list stores the end point of the edge. The total time needed for this computation is $O(n + k)$, since there are $n + 2k$ subsegments. Note that each intersection point is duplicated although this could be avoided by modifying the representation a little. Note furthermore that if the real spatial edges corresponding to the projected edges $e_{i_1}, \ldots, e_{i_m}$ meet at a common vertex on the boundary of a polyhedron, then the projection of this common vertex is represented m times in our present structure. So we merge the different occurrences of each vertex into one. This can be done by first sorting the vertices in *lexicographic order* with respect to their x, y coordinates and then merging equal ones. Lexicographic ordering means that a vertex with coordinates x_1, y_1 precedes another one with coordinates x_2, y_2, if $x_1 < x_2$ or $x_1 = x_2 \wedge y_1 < y_2$. They are equal if $x_1 = x_2 \wedge y_1 = y_2$. The merging operation can be performed in $O((n+k)\log(n+k)) = O((n+k)\log n)$ time because of the sorting step. Having done this, we have a data structure for the subdivision graph G, which is similar to an adjacency list representation with the difference that there are not only vertices but edges too, and the neighbors (edges, vertices) are not ordered around the vertices. Ordering adjacent edges around the vertices can be done separately for each vertex. For a vertex v_i with N_i edges around it, this can be done in $O(N_i \log N_i)$ time. The total time required by the m vertices will be $O((n + k)\log n)$, using relation 6.60 again with $N_1 + \ldots + N_m = n + 2k$. The data structure obtained in this way is halfway between the adjacency list and the DCEL representation of G. It is "almost" DCEL, since edges appear explicitly, and each edge has references to its endpoints. The two reasons for incompleteness are that no explicit representation of faces appears, and the edges have no explicit reference to the edges next to them around the endpoints — the references exist, however, but only implicitly through the vertices. Since the edges are already ordered about the vertices, these references can be made explicit by scanning all the edges around each vertex, which requires $O(n+k)$ time. The faces can be constructed by first generating the faces of the connected components of G separately, and then merging the DCEL structure of the components into one DCEL structure. The first step can be realized by using an algorithm very similar to **Traverse**, since the outer boundary of each face can be easily retrieved from our structure, because edges are ordered around vertices. Assuming that the face references $f_1(e), f_2(e)$ of each edge e are initialized to *null*, the following algorithm constructs the faces of G and links them into the DCEL structure:

```
for each edge e do MakeFaces(e); endfor
MakeFaces(e)
    for i = 1 to 2 do
        if f_i(e) = null then
            construct a new face f;
            e(f) = e; set m (the number of holes in f) to 0;
            for each edge e' on the boundary of f do
                if f_1(e') corresponds to the side of f then
                    f_1(e') = f;
                else
                    f_2(e') = f;
                endif
                MakeFaces(e');
            endfor
        endif
    endfor
end
```

Note that the recursive subroutine **MakeFaces**(e) traverses that connected component of G which contains the argument edge e. The time complexity of the algorithm is proportional to the number of edges, that is $O(n+k)$, because each edge is taken at most three times (once in the main loop and twice when traversing the connected component containing the edge).

The resulting structure generally consists of more than one DCEL structure corresponding to the connected components of G. Note furthermore that the surrounding faces contain no holes. Another observation is that for any connected component G' of G the following two cases are possible: (1) G' falls onto the territory of at least one component (as in figure 6.14) and then it is contained by at least one face. (2) G' falls outside any other components (it falls into their surrounding face). In case (1) the faces containing G' form a nested sequence. Let the smallest one be denoted by f. Then for each boundary edge of G', the reference to the surrounding face of G' has to be substituted by a reference to f. Moreover, the boundary edges of G' will form the boundary of a hole in the face f, hence a new hole edge reference $h_{m+1}(f)$ (assuming that f has had m holes so far) has to be created for f, and $h_{m+1}(f)$ is to be set to one of the boundary edges of G'. In case (2) the situation is very similar, the only difference being that the surrounding face F corresponding to the resulting graph G plays the role of f. Thus the problem is first creating F, the "united" surrounding face of G, and then locating and linking the connected components of G in its faces. In order to accomplish this task efficiently, a **sweep-line** approach will be used.

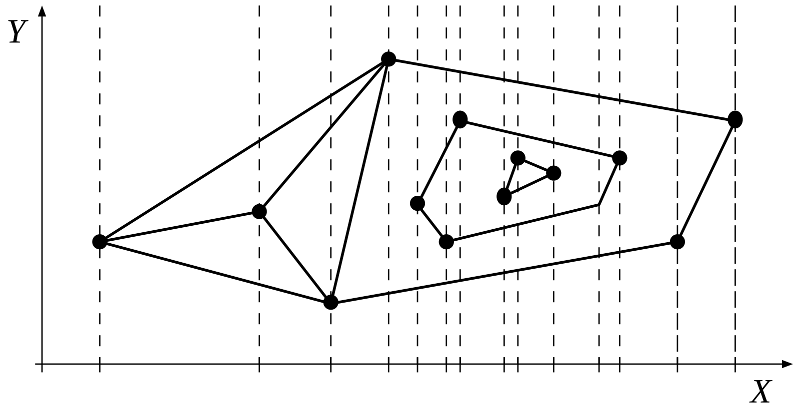

Figure 6.17: Slabs

The problem of locating a component, that is finding the face containing it, is equivalent to the problem of locating one of its vertices, that is, our problem is a point location problem. Imagine a set of vertical lines through each vertex of the graph G, as shown in figure 6.17. These parallel lines divide the plane into unbounded territories, called *slabs*. The number of slabs is $O(n+k)$. Each slab is divided into $O(n+k)$ parts by the crossing edges, and the crossing edges always have the same order along any vertical line in the interior of the slab. Given efficient data structures (with $O(\log(n+k))$ search time) for storing the slabs and for the subdivision inside the slabs, the problem of locating a point can be performed efficiently (in $O(\log(n+k))$ time). This is the basic idea behind the following algorithm which first determines the order of slabs, and then scans the slabs in order (from left to right) and incrementally constructs the data structure storing the subdivision. This data structure is a balanced binary tree, which allows efficient insertion and deletion operations on it. In order that the algorithm may be understood, two more notions must be defined. Each vertical line (beginning of a slab) corresponds to a vertex. The edges incident to this vertex are divided into two parts: the edges on the left side of the line are called *incoming edges*, while those on the right side are *outgoing edges*. If we imagine a vertical line sweeping the plane from left to right, then the names are quite apt. The vertex which is first encountered during the sweep — that is, the vertex furthest to the left — definitely corresponds to the boundary of the (first) hole of the surrounding face F, hence F can be constructed at this stage. (Note that this is so because the line segments are assumed to be straight.) Generally, if a vertex v with no incoming edges is encountered during the sweep (this is the case for the furthest left vertex too), it always denotes the appearance of a new connected component, which then has to be linked into the structure. The structure storing the subdivision of the actual slab (that is the edges crossing the actual slab) will be a balanced tree T.

The algorithm is the following:

> sort all the vertices of G by their (increasing) x coordinates;
> create F (the surrounding face);
> $T = \{\}$;
> **for** each vertex v in increasing x-order **do**
>   **if** v has only outgoing edges (a new component appears) **then**
> f = the face containing v (search in T);
>   mutually link f and the boundary chain containing v;
>   **endif**
>   **for** all the incoming edges e_{in} at v **do**
>   delete e_{in} from T;
>   **endfor**
>   **for** all the outgoing edges e_{out} at v **do**
>   insert e_{out} into T;
>   **endfor**
> **endfor**

The face f containing a given vertex v can be found by first searching for the place where v could be inserted into T, and then f can be retrieved from the edge either above or below the position of v in T. If T is empty, then $f = F$.

The sorting (first) step can be done in $O((n+k)\log(n+k)) = O((n+k)\log n)$ time; the main cycle is executed $O(n+k)$ times; the insertions into and deletions from T need only $O(\log(n+k)) = O(\log n)$ time. The time required to link the boundary of a connected component into the face containing it is proportional to the number of edges in the boundary chain, but each component is linked only once (when encountering its leftmost vertex), hence the total time required by linking is $O(n+k)$. Thus the running time of the algorithm is $O((n+k)\log n)$.

We have come up with a DCEL representation of the subdivision graph G, and we can conclude that the time T_3 consumed by step 3 of the visibility algorithm is:

$$T_3 = O((n+k)\log n). \tag{6.61}$$

Steps 4–9: Traversing the subdivision graph G

Note that it causes no extra difficulties in steps 1–3 to maintain two more references $F_1(e), F_2(e)$ for each edge e, pointing to the spatial faces incident to the original edge from which e has been projected (these are boundary faces of polyhedra in the object scene).

Steps 4–9 of the algorithm will be examined together. The problem is to visit each face of G, retrieve the spatial polygon closest to the image plane for the face, and then draw it. We have already proposed the algorithm **Traverse** for visiting the faces of a DCEL structure. Its time complexity is linearly proportional to the number of edges in the graph, if the action performed on the faces takes only a constant amount of time. We will modify this algorithm a little bit and examine the time complexity of the action. The basic idea is the following: for each face f of G, there are some spatial polygons, the projection of which completely covers f. Let us call them *candidates*. The projections of all the other polygons have empty intersection with f, hence they cannot be visible in f. Candidate polygons are always in a unique order with respect to their distance from the image plane (that is from f). The candidate polygon must always be retrieved at the first position. The candidate-set changes if we cross an edge of G. If we cross some edge e, then for each of the two spatial faces $F_1(e)$ and $F_2(e)$ pointed to by e there are two possibilities: either it appears as a new member in the set of candidates or it disappears from it, depending on which direction we cross e. Thus we need a data structure which is capable of storing the actual candidates in order, on which insertions and deletions can be performed efficiently, and where retrieving the first element can be done as fast as possible. The balanced binary tree would be a very good candidate were there not a better one: the **heap**. An N-element heap is a 1-dimensional array $H[1,\ldots,N]$, possessing the property:

$$H[i] \leq H[2i] \quad \text{and} \quad H[i] \leq H[2i+1]. \tag{6.62}$$

Insertions and deletions can be done in $O(\log N)$ time [Knu73], just as for balanced binary trees, but retrieving the first element (which is always $H[1]$) requires only constant time. Initializing a heap H for storing the candidate polygons at any face f can be done in $O(n \log n)$ time, since $N = O(n)$ in our case (from Euler's law concerning the number of faces, edges and vertices of polyhedra). This has to be done only once before the traversal, because

H can be updated during the traversal when crossing the edges. Hence the time required for retrieving the closest polygon to any of the faces (except for the first one) will not be more than $O(\log n)$. The final step is the drawing (filling the interior) of the face with the color of the corresponding polygon. Basic 2D scan conversion algorithms can be used for this task. An arbitrary face f_i with N_i edges can be raster converted in $O(N_i \log N_i + P_i)$ time, where P_i is the number of pixels it covers (see [NS79]). The total time spent on raster converting the faces of G is $O((n+k)\log n + R^2)$, since $N_1 + \ldots + N_m = 2(n + 2k)$, and $P_1 + \ldots + P_m \leq R^2$ (no pixel is drawn twice), where R^2 is the resolution (number of pixels) of the screen. Thus the time T_4 required by steps 4–9 of the visibility algorithm is:

$$T_4 = O((n + k)\log n + R^2). \tag{6.63}$$

This four-step analysis shows that the time complexity of the proposed visibility algorithm, which first computes the visibility map induced by a set of non-intersecting polyhedra having n edges altogether, and then traverses its faces and fills them with the proper color, is:

$$T_1 + T_2 + T_3 + T_4 = O((n + k)\log n + R^2), \tag{6.64}$$

where k is the number of intersections between the projected edges on the image plane. It is not really an output sensitive algorithm, since many of the k intersection points may be hidden in the final image, but it can be called an *intersection sensitive* algorithm.