18 Visible Surface Determination

We are surrounded by objects of every size, shape, and color, but we don't see all of them. Nearby objects tend to obscure parts of distant objects. The visibility problem, the problem of deciding which elements of a rendered scene are visible by a given observer, and which are hidden, has to be solved by any three-dimensional graphics program or software package.

Older texts on computer graphics tend to refer to this topic as "hidden-surface elimination" but the title of this chapter is the opposite. Instead of hidden surface we use visible surface and instead of elimination we have determination. This usage reflects the nature of the algorithms discussed in this chapter. Instead of drawing the complete surfaces and then eliminating certain parts, these algorithms determine which parts of a surface are visible, and then draw these parts pixel by pixel. (Also, if the objects are wireframes, we generally draw the hidden parts in gray or dashed, instead of eliminating them.)

In the discussion that follows we assume that the scene to be displayed consists of n objects (surface patches), each of which may partly obscure parts of objects located behind it. There are two main approaches to determining the visible parts of surfaces as follows:

- The pixels of the display monitor are scanned one by one, and for each pixel the software determines the particular object (if any) that the viewer will see at the pixel. The software constructs a vector (a ray) from the viewer to the pixel on the screen, and continues that vector into the scene until it hits an object at a point P or until it gets out of the scene. In the former case, the pixel is painted the correct color, depending on the light source and the normal to the surface at P. In the latter case, the pixel is painted the background color. Such an algorithm is referred to as an "image precision" method.

As the ray penetrates deeper and deeper into the scene in small steps, it has to be checked against each of the n objects at each step, so the total time for processing the vector is proportional to n. There is such a ray for each pixel, so the total time of an image precision algorithm is about $p \cdot n$, where p is the number of pixels on the screen. Generally, p is on the order of a few millions, and n may be several dozen to several thousand surface patches.

- Each object B is compared with all the other objects to determine which parts of B are unobstructed by any other objects. The pixels of the screen are not used in such an algorithm, which is therefore named "object precision."

Each of the n objects is compared with the $(n-1)$ other objects, so the total time complexity of an object-precision algorithm is $n(n-1)$.

Figure 18.1 shows a scene with two objects, a triangle obscuring part of a rectangle behind it. Once the scene has been rotated, it is easy to see how two rays from the viewer hit the triangle. The upper ray would have hit the rectangle too, but the algorithm must stop it when it hits the surface nearest the viewer.

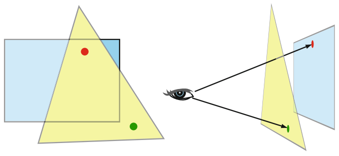


Figure 18.1: Rays Hitting Two Objects.

The main advantage of an object-precision algorithm is obvious. Such an algorithm is executed at the resolution of the objects (surfaces), so once the algorithm does its job, the entire scene can be displayed and printed quickly at different resolutions. Once the output resolution is known, an extra step is required in which the results of the algorithm are adapted to the specific resolution. In contrast, an image-precision method depends on the number of pixels of the output device, and so has to be repeated each time the scene needs to be displayed or printed at a different resolution.

Any algorithm for visible-object determination must examine the original, three-dimensional coordinates of points. Such an algorithm may include a depth-comparison step in which it has to decide whether two points lie on the same line from the observer. This basic decision is based on Equation (6.3), which relates the projected coordinates (x^*, y^*) of a point (x, y, z) to its original, three-dimensional coordinates. Given two points $\mathbf{P}_1 = (x_1, y_1, z_1)$ and $\mathbf{P}_2 = (x_2, y_2, z_2)$, the equation implies that they are on the same ray from the viewer if $x_1^* = x_2^*$ and $y_1^* = y_2^*$ or

$$\frac{x_1}{k+z_1} = \frac{x_2}{k+z_2} \quad \text{and} \quad \frac{y_1}{k+z_1} = \frac{y_2}{k+z_2},$$

where k is the distance of the observer from the projection plane (Figure 6.34a). Unfortunately, such a step requires four divisions, but if $\mathbf{P}_1$ is later compared with another point, only two divisions are needed.

18.1 Ray Casting

Ray casting is a conceptually simple but computationally intensive image-precision algorithm. The basic idea is to cast a ray from the viewer's location $(0, 0, -k)$, through a pixel $(x, y, 0)$ on the screen, and to extend the ray until it intercepts a surface. This is repeated for every pixel on the screen.

The parametric equation $R(t)$ of such a ray is $(0, 0, -k) + t\alpha$ where $\alpha = (x, y, k)$ is a vector that points from the viewer to pixel $(x, y, 0)$. If we divide α by its magnitude, then t, the parameter, is also the length of the ray.

The main computational step is to check for an intersection of the ray with each object. Each time an intersection is found, the value of t is saved. When all the objects have been checked in this way, the smallest value of t is used to select the nearest object.

Because of its complexity, ray casting is used mostly in ray tracing, for accurate rendering of arbitrary objects that may transmit and refract light.

18.2 Z-Buffer Method

The Z-buffer algorithm (also titled depth buffer) is an image-precision method originated in 1974 by Edwin Catmull and Wolfgang Straßer, working independently. We first describe this simple idea assuming that the scene to be rendered consists of flat polygons. In spite of the name Z-buffer, this algorithm requires two buffers, a Z-buffer Z for saving the z coordinates of various pixels on the polygons and a frame (or color) buffer F for saving the colors of the pixels. The dimensions of each buffer equal those of the screen (or other output device) on which the scene is projected.

We further assume that the scene is a viewing volume fully contained between two planes as discussed in Section 6.11. All z coordinates of points on objects are less than a maximum value K. All the Z-buffer entries are initialized to K and all the frame buffer entries are initialized to the background color. During the algorithm, each entry of the Z-buffer either stays the same (with K, or background) or its value decreases as smaller and smaller z coordinates of pixels are stored in it. At the end, an entry either contains K or the z coordinate of the pixel closest to the observer.

The algorithm processes the polygons one by one, in no particular order. Each polygon is scan-converted to three-dimensional pixels with coordinates (x, y, z) and each pixel is projected to a two-dimensional pixel with screen coordinates (x^*, y^*). For each of those pixels, its original (three-dimensional) z coordinate is compared to the value p that happens to be in entry $Z(x^*, y^*)$ in the Z-buffer. If $z < p$, then z is stored in $Z(x^*, y^*)$ because the current pixel is closer to the viewer than the pixel whose z coordinate is p. In addition, the algorithm has to compute the light (intensity and color) reflected from pixel (x, y, z) and store this value in $F(x^*, y^*)$.

At the end of the algorithm, when all the polygons have been scanned and all the pixels projected and compared, each entry in Z contains either K or the smallest z coordinate of all the pixels that projected to (x^*, y^*). Buffer F contains the final image and can be output (displayed or printed) directly.

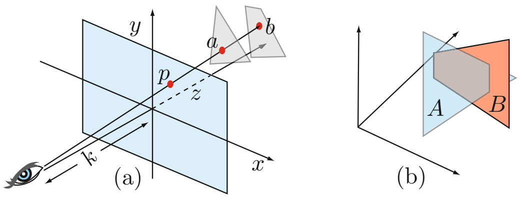


Figure 18.2: Two 3D Points Projected to the Same 2D Point.

Figure 18.2a illustrates how two different three-dimensional points a and b on two polygons can be projected to the same two-dimensional point p on the screen.

The main shortcoming of the Z-buffer algorithm is the unneeded rendering. If point b of Figure 18.2a is scanned and rendered before point a, then its color is computed and stored in the F buffer only to be erased when the color of a is stored at the same location. The algorithm can therefore be speeded up if the polygons that constitute the scene are sorted before the main loop starts, but such sorting is only approximate, because certain parts of polygon A may be in front of polygon B while other parts of A may be behind B (Figure 18.2b).

If the polygons are processed in sorted order, the main time-cosuming step is to scan a polygon pixel by pixel to determine the z coordinates of the pixels. It is in this step that we exploit the fact that our polygons are flat (planar). The implicit equation of a flat plane is (Section 9.2.2) $Ax + By + Cz + D = 0$. The special case where $C = 0$ is the xy plane itself, where we assume the screen to be located. Thus, the equations of the polygons that are of interest to us always satisfy $C \neq 0$. Dividing by C yields $z = -(Ax + By + D)/C = -\frac{A}{C}x - (By + D)/C$. We scan a polygon by y values where for each y, we scan a row of the polygon by varying x. Once we have determined the value of z for a point with coordinates (x, y), then determining $z1$ for any point $(x1, y)$ on the same scan line can be done by $z1 = z - \frac{A}{C}(x1 - x)$, an operation that requires two subtractions (or, if $x1 = x + 1$, only one subtraction) and a multiplication. Similarly, if all the z values for a scan line (i.e., a certain y) are saved, they can be used to determine z values of the next scan line (the one for $y + 1$).

Another advantage of flat polygons is that all the points on it have the same normal, which speeds up the calculations of light reflection. However, a polygon may be curved, in which case the z coordinate of any point on it is computed as a weighted average, as illustrated by Figure 18.3.

The steps are

$$\begin{aligned} z_a &= z_1 - (z_1 - z_4)\frac{y_1 - y}{y_1 - y_4}, \\ z_b &= z_2 - (z_2 - z_3)\frac{y_2 - y}{y_2 - y_3}, \\ z &= z_b - (z_b - z_a)\frac{x_b - x}{x_b - x_a}. \end{aligned}$$

The surface patches that constitute the scene do not have to be polygons. They may be any parametric surfaces, such as those discussed in Part III of the book. The

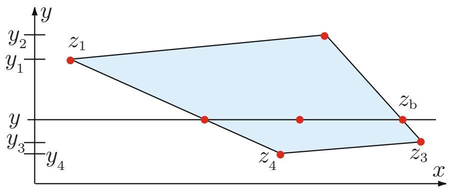


Figure 18.3: Interpolating z in a Curved Polygon.

difference between a polygon and a parametric surface patch is in the way they are scanned. A surface patch is scanned by varying its two parameters u and w in a double loop.

18.3 Explicit Surfaces

Section 8.11 introduces explicit surfaces, whose representation is $z = f(x, y)$ (see also Exercise 9.12 and Section 13.20). Such a function is single valued, because there is only one value of z for each pair (x, y) of coordinates. Thus, explicit surfaces are not general (for example, there cannot be a vertical line all of whose points have the same x and y coordinates) but are nevertheless useful because we often want to visualize the shape of a mathematical function of two variables.

An explicit surface is easy to plot as one or two families of contours, and such a plot is referred to as a wireframe or a mesh (Section 8.11.2). One family of contours is obtained when x is incremented in small steps, and a contour is drawn by varying y. The other family is plotted by changing the roles of x and y.

Here, we discuss an approach to determining the visible parts of such a wireframe. Figure 18.4 shows part of the explicit surface $z = \sqrt{x^2 + y^2} + \arctan(x, y)$ with and without the hidden parts, and it is obvious that determining the visible parts of such a surface greatly helps in visualizing its shape, extent in space, and details of small regions.

The main idea is illustrated in Figure 18.5. The top part of the figure shows five curves. We assume that each corresponds to an x value, i.e., each curve is of the form $z = f(x_i, y)$ where only y is varied. The order of the curves is obvious because of the use of perspective (the closer a curve is, the longer it is drawn), but the precise shape of this simple surface patch is unclear, especially around its center, where curves 4 and 5 meet and cross several times. In contrast, the bottom part of the figure is much clearer because only the visible parts are shown. It is easy to see how curves 1–3 hide two parts of curve 4, which in turn blocks from our view parts of curve 5.

This observation is the key to our method. We simply draw the contour curves from the nearest to the farthest, because each curve can obstruct only the ones behind it (the ones that haven't been drawn yet). Thus, the first curve (the one for the smallest value of x) is simply drawn without any tests. The second curve is also fully drawn, because the first curve cannot hide any parts of it. The first two curves can only intersect at

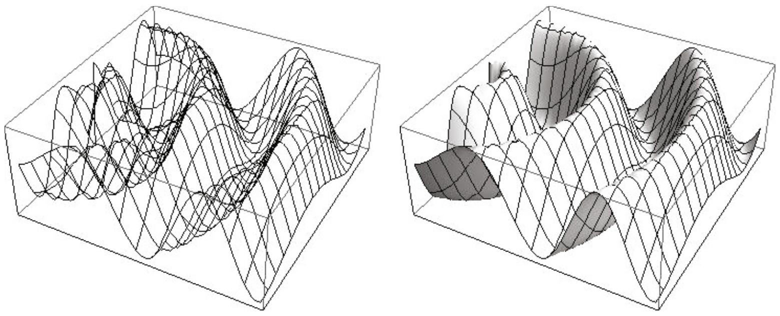


Figure 18.4: Explicit Surface with and without Hidden Parts.

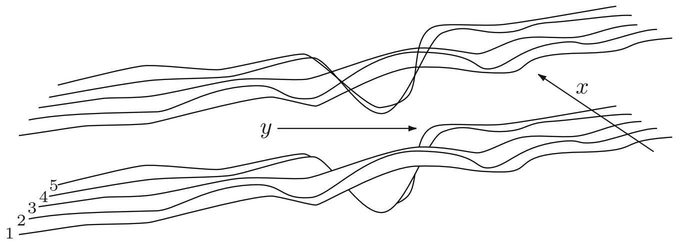
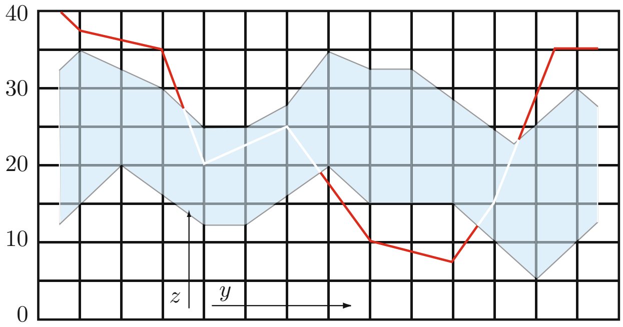


Figure 18.5: Five Curves of Constant x.

certain pixels. It is only when we get the the third curve that we have to start checking for visible parts.

The method described here (due to [Wright 73]) employs two arrays `minz` and `maxz`, as the silhouette of the surface. When the first curve $z = f(x_1, y)$ is drawn by varying y, its z coordinates are stored in both arrays. When the second curve $z = f(x_2, y)$ is drawn, its z coordinates are compared to what is already in the arrays. For each y value, if the new z is greater than `maxz[y]` or is less than `minz[y]`, the new z replaces the current value in the array. When the third curve $z = f(x_3, y)$ is drawn, only those z values that are greater than `maxz[y]` and less than `minz[y]` are drawn and they also replace the older values in the silhouette. As more contour curves are drawn, only their visible parts are actually drawn, and the silhouette is updated for each curve. Notice that `minz[y]`$\leq$`maxz[y]` for any `y`.

Figure 18.6 illustrates this process for numerous y values and shows how the two arrays are updated as y is incremented and more points of the curve are checked. Notice the hidden parts of the curve (in white) and the fact that sometimes the curve crosses the silhouette between y values. Thus, between the third and fourth y values, the curve drops suddenly from 35 to 20, crossing the top part of the silhouette at about $z = 28$. A sophisticated algorithm has to interpolate y values in such a case and locate the precise y value where the curve crosses the boundary silhouette.



`maxz`	35	32	30	25	25	28	35	32	32	29	25	25	30
`minz`	15	20	16	13	13	16	20	15	15	15	10	5	10
Curve	38	36	35	20	23	25	18	10	8	7	15	30	35
New `maxz`	38	36	35	25	25	28	35	32	32	29	25	30	35
New `minz`	15	20	16	13	13	16	18	10	8	7	15	5	10

Figure 18.6: Updating Arrays `maxz, minz`.

We now turn to the other family of contour curves, those for constant y values. The equation of such a curve is $z = f(x, y_i)$, where for each y_i, variable x is varied over its entire range. Figure 18.7 shows ten curves of constant y of the function $z = \sin(x + y^2)$. The algorithm is similar, with the exception that the curve closest to the observer is not the leftmost or rightmost curve, but the curve labeled 1. The closest curve is the one where, after the projection, the difference in the y coordinates between its extreme points is minimal. If the surface is rectangular and one boundary curve is perpendicular to the line of sight, then the projection of the curve of constant y that is closest to the observer is close to a straight line.

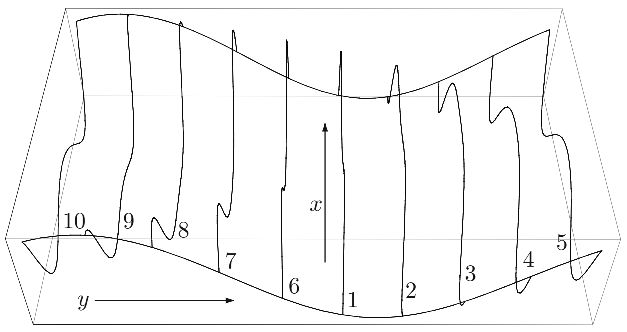


Figure 18.7: Curves of Constant y.

A look at Figure 18.7 makes it clear that each of the curves to the right of curve 1 can obstruct only curves to its right. Thus, they are drawn in the order shown in the figure, from 2 to 5. Each of the curves to the left of curve 1 can obstruct only curves to its left, so they have to be drawn from right to left, in the order shown, from 6 to 10.

The natural question at this point is how to combine the two families of curves to a single mesh. A little experimentation should convince the reader that simply superimposing the two families produces a wrong result, because each family contains curve segments that are hidden by the other family. The solution is to interleave the drawing of the two families as shown in Figure 18.8. The first family, of curves X_i of constant x, is plotted as before, but after each curve X_i, an entire set of short segments Y_j of constant y are plotted from X_i to X_{i+1}, using the *same* silhouette.

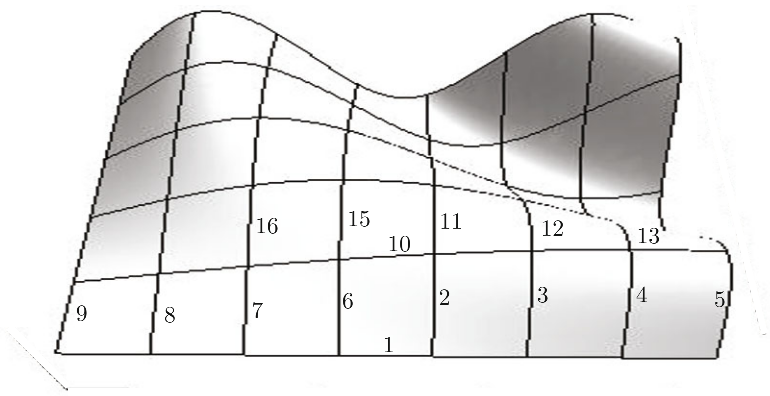


Figure 18.8: A Complete Mesh of Curves.

The approach described here is simple, but not general. It assumes that one family of curves is parallel to the line of sight of the observer and the other family is perpendicular to that line. A more general technique is described in [Anderson 82].

18.4 Depth-Sort Method

The depth-sort algorithm, due to [Newell et al. 72], assumes that the objects to be rendered are flat polygons (but the authors mention that polygons that are only slightly curved can also be rendered with this method). The main idea is to sort the polygons according to their maximal z coordinate, go over the list of polygons from maximum z (the farthest away from the observer) toward the observer, and render each polygon in the list. This kind of method is often referred to as the painter's algorithm, because an artist learns quickly to start a new canvas by painting distant objects first, followed by closer objects which may partially obstruct already-painted areas on the canvas.

The painter's algorithm can be used in any graphics application in which each of the objects that make up the scene is located in a plane of constant z (i.e., all the points of an object have coordinates (x, y, z_i) for the same z_i). Such applications are sometimes referred to as 2.5-dimensional and include mapping (cartography), printed circuit board design, fabric design, and cartoons.

In general, however, the painter's algorithm cannot always be used, because the z extents of the polygons in the scene may overlap, and may do so in simple or complex ways. This problem is illustrated in Figure 18.9. Part (a) of the figure shows four flat polygons sorted by their maximal z coordinates and numbered 1 through 4. They are located on the xz plane in order to simplify their spatial relationship, but in general they can be anywhere. It is obvious that the z extents of polygons 1 and 2 do not overlap, but those of polygons 3 and 4 do, albeit in a simple way. Thus, an important part of the sort-depth algorithm is to check for such overlaps. The point is that a simple overlap of the z extents of two polygons is easy to deal with. In such a case, one polygon partially obscures the other, so once we determine, for example, that A obscures B, we simply render B before A, regardless of their places in the sorted list of polygons.

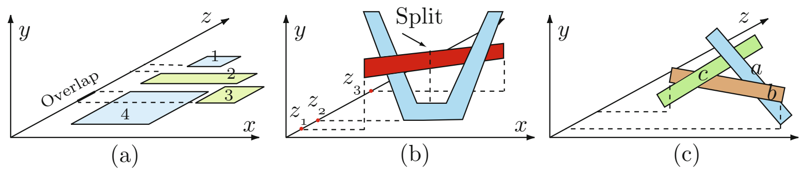


Figure 18.9: Various Polygons for Depth-Sort.

Parts (b) and (c) of the figure illustrate complex overlapping of the z extents of polygons. In part (b), one polygon passes through another. The depth-sort algorithm handles such a case by splitting one of the polygons by the plane of the other, as shown. Once this is done, the three resulting polygons partly obscure each other, but this is a simple overlap that is dealt with as discussed earlier. In part (c), the three polygons overlap in such a "clever" way that splitting any of them (splitting, say a by the plane of b) would still leave an overlap between the top part of a and polygon c, as well as between b and c.

We are now ready for the details of the algorithm.

The maximal z coordinate of each polygon is determined, the polygons are sorted by these coordinates, and are placed in a list.

The list is scanned from the end (largest z). Denote the last polygon in the list by P. Polygon P has to be tested against all the polygons preceding it in the list, because it may obscure any polygon that it overlaps. When P is tested against a polygon Q, the first test is z-extent overlap. If the smallest z coordinate of P is greater than the largest z coordinate of Q, then there is no overlap and P can be rendered before Q. (Rendering is done by scan converting P and determining the amount and color of light reflected from each pixel.) Once a polygon is rendered, it is deleted from the list, The polygon preceding it becomes the current one. If the list is empty, the algorithm stops.

If P overlaps Q, then the depth-sort algorithm performs up to five tests (listed here in order of increasing complexity) to determine the relation between P and Q. If any test succeeds, then P does not obscure Q. This does not mean that P can be rendered immediately, because it may obscure other polygons that it overlaps. P can be rendered only if the tests show that it does not obscure any of the polygons with which it overlaps. The five tests are as follows:

1. Do the x extents of P and Q not overlap?
2. Do the y extents of P and Q not overlap?
3. Is P entirely on the opposite side of Q's plane as is the observer (Figure 18.10a)?
4. Is Q entirely on the same side of P's plane as is the observer (Figure 18.10b)?
5. Do the projections of P and Q on the xy plane not overlap?

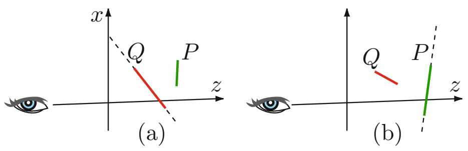


Figure 18.10: Illustrating Tests 3 and 4 of Depth-Sort.

◇ **Exercise 18.1:** Explain how to perform tests 3 and 4, i.e., how to verify that a given polygon L and the observer at $(0,0,-k)$ are on different sides (or the same side) of the plane of Q?

Again, if any test succeeds, then P does not obscure Q, but what if all five tests fail? In such a case, we tentatively assume that P does obscure Q. That does not mean that we can render Q immediately, because the relationship between P and Q may be complex, as in Figure 18.9b,c. Thus, we apply the same tests to Q. (Notice that tests 1, 2, and 5 are symmetric and therefore do not have to be repeated. Only tests 3 and 4 must be performed, and with P and Q switched.) If any test succeeds, then Q does not obscure P. Even in this case, Q cannot be rendered immediately, because it may obscure another polygon. Thus, if any test succeeds, Q is moved to the end of the list (and is also marked, as explained below) and it becomes the new P. If all the new tests fail, then we assume that, in addition to P obscuring Q, Q also obscures P. The relation between these two polygons resembles that of Figure 18.9b, so one of them has to be split by the plane of the other. The original split polygon is deleted from the list and its two descendants are inserted into the list in the correct z order.

The case illustrated by Figure 18.9c is more complex and requires special treatment. Polygon a obscures c, c obscures b, and b obscures a. If we move any of these polygons (say, a) to the end of the list, it may be placed in the correct order relative to b but not relative to c. We may then test b and c and decide to move b to the end of the list, thereby messing up the relation between a and b and causing an infinite loop.

The solution adopted by the depth-sort algorithm is to mark each polygon that is moved to the end of the list. Now, whenever the first round of five tests fails and polygon P is found to be already marked, it is not moved to the end of the list but is split instead and the resulting two polygons are inserted into the list according to their maximal z coordinates.

Complex, but this is the nature of visible surface determination algorithms.

◇ **Exercise 18.2:** Explain how to extend this algorithm to polygons with holes.

◇ **Exercise 18.3:** We see objects around us all the time, and they always obscure each other correctly, without the need for complex algorithms and slow calculations. What is the difference between rendering digital, imaginary objects of a scene and seeing real objects?

18.5 Scan-Line Approach

When the surface patches that constitute the scene are flat polygons, surface visibility can be determined by a scan-line type of algorithm, similar to the one described in Section 3.9.2 for filling a polygon. (The conscientious reader is advised to read that section before reading ahead.) The scene is painted scan line by scan line, where each line fills up spans of polygons. The spans are determined by the same algorithm that is used to fill a polygon, with the added task of deciding which of several z-overlapping polygons is the one visible to the observer at any point. We start with the basic idea of filling a polygon by scan lines.

The idea is to determine the locations of the polygon's interior pixels in each scan line (row), and then paint them with the fill color. The determination is done by computing the locations where a scan line intersects the edges of the polygon, as illustrated in Figure 18.11, duplicated here. The scan line at $y = 5$ computes four intersections that correspond to two spans, and paints three pixels in one span and four pixels in the next span of the polygon. The principle is to scan from left to right, to start painting pixels with the fill color as soon as the first edge is located, to stop filling when the next edge is found, and to alternate in this way until the last intersection point is found. An actual computer program may use a boolean variable (a flag), initialize it to false, toggle it each time the current scan line intersects a polygon edge, and fill pixels with the fill color when the flag is true.

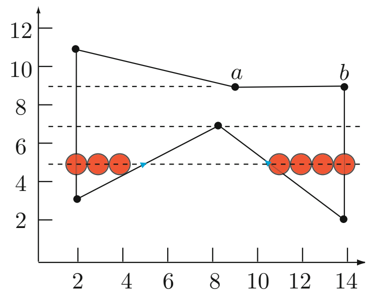


Figure 18.11: Pixels on a Scanline.

This straightforward process has to deal with the following problems:

- How to compute the intersection of each scan line with the polygon's edges. Scan line 5 in Figure 18.11 intersects the polygon at four points, two of which (indicated by small triangles) have noninteger x coordinates. We certainly do not want complex computations, involving floating-point numbers, just to determine the intersection points.

- How to identify the last intersection point of the scan line with an edge. If the algorithm cannot do this, it may have to continue scanning until the right edge of the entire display monitor is reached.

- The scan line at $y = 7$ intersects the polygon at three points, one of which corresponds to a vertex of the polygon. An odd number of intersections confuses the basic algorithm. Thus, an intersection of a scan line with a vertex should count as either zero or two intersections.

- Edge ab of the polygon is horizontal and may result in many intersection points. A correct algorithm should be able to deal with this case.

The following idea solves most of the problems above. Start with a list of the edges of the polygon. Each node in this list is a pair of vertices (actually, it is a pointer to a pair of vertices). Scan convert each edge with any scan-conversion method for straight lines (Section 3.1). The result of scan converting all the edges of a polygon is shown in Figure 18.12a. The points determined by the scan-conversion process are placed in a list T and are sorted by their y coordinates and, within each y, by their x coordinates. A careful check of such a list shows that it is not exactly what we had in mind. We need a list that will have an even number of points for each scan line (each y coordinate), but the list resulting from the pixels of Figure 18.12a is different.

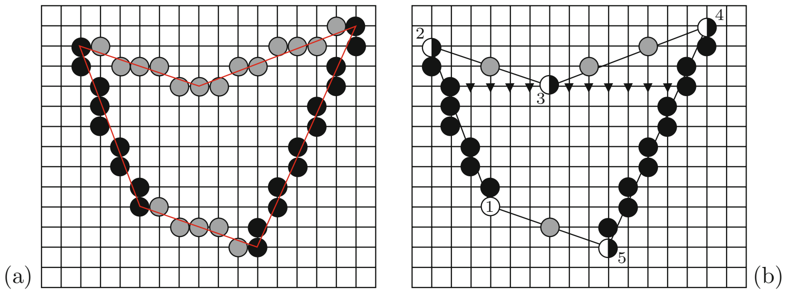


Figure 18.12: Boundary Pixels of a Polygon.

A closer look at the figure shows the source of the problem. For edges whose slope is greater than 1, the scan conversion produces one pixel per y value, but scanning a shallow edge produces one pixel for each x value (the gray pixels in Figure 18.12a). Thus, the basic scan-conversion algorithm has to be modified to produce one pixel per y value for any slope, as illustrated in Figure 18.12b. Notice that this modification results in holes and gaps in the boundary pixels, but each scan line now has an even number of pixels (the black/white pixels are counted twice, because each is created twice, from two edges).

This modification solves most of our problems, but we are not yet done. The white pixel (labeled 1) appears in list T twice, because it is a vertex (the common point of two edges), but it is obvious that we want it to appear in T only once, otherwise the third scan line from the bottom would have three points. The discussion in Section 3.9.2 explains the difference between this vertex (which is referred to as moderate) and the four other vertices, labeled 2 through 5, that we call extreme.

We now have to add another rule or test to our fill algorithm in order to handle moderate vertices. Scan list T after it is sorted, looking for pairs of adjacent identical points. The two points of such a pair are endpoints of edges that meet at a vertex. If the vertex is moderate (located at the top of one edge and the bottom of the other edge), then one of the two identical points should be deleted from T. Experience with this algorithm (with polygons that have horizontal edges) suggests that we should be consistent, either always delete the point that is located at the bottom of its edge or always delete the point at the top of its edge. In the polygon of Figure 18.12b, we delete the point at the bottom of edge 1–2 and retain the point at the top of edge 5–1. Both points correspond to pixel 1, but now this pixel appears in list T only once.

Once list T has been constructed, the rest of the fill algorithm is straightforward. The software scans T, examines nodes, and toggles a flag each time a node is found. The list is sorted by scan lines (y values) and has an even number of points for each scan line. Each of these points causes the algorithm to toggle the flag and switch its behavior. The first point sets the flag to true and starts a fill, the second point clears the flag to false and terminates the fill, the third point starts another fill, and so on. The third scan line from the top (in Figure 18.12b) serves as an example. The first boundary pixel starts a string of four fill pixels (shown as triangles). Pixel 3 stops the fill, but the second occurrence of the same pixel starts another fill that includes this pixel and six more. Finally, the rightmost boundary pixel on this scan line stops the fill, and the next point encountered in T by the algorithm corresponds to the next scan line.

The detailed discussion in Section 3.9.2 shows also how to deal properly with horizontal edges. Simply delete the two endpoints of every horizontal edge from list T (rather, if the two endpoints of an edge have the same y coordinate, don't even place them in T).

An actual implementation of this visible surface determination algorithm employs the following data structures:

- A polygon table (PT). This includes the name of each polygon in the scene, as well as its color (for filling or shading), the boolean flag used by the algorithm, a pointer to the polygon's edge table (ET), and the four coefficients of its plane equation.

- An edge table (ET) for all the polygons in the scene. The ET is a list of nodes (often called buckets) for all the scan lines, i.e., all the y values spanned by the polygons in the scene. The bucket corresponding to k is the start of a list of (only non-horizontal) edges whose $y_{min} = k$. Thus, many buckets may be left empty. Each bucket of the ET list is either empty or points to a list of edges where each node contains the y_{max} of the edge, the x value of the bottom endpoint of the edge (the list is kept sorted by these values), and a term of the form $1/a$, where a is the slope of the edge (i.e.,

$(y_{max} - y_{min})/(x_{max} - x_{min})$). The discussion in Section 3.9.2 explains how the value $1/a$ is used to determine the intersections of the edge with consecutive scan lines. Notice that $1/a$ is 0 for vertical edges and is undefined for horizontal edges, but such edges are ignored by our algorithm anyway.

- Active-edge list (AEL). A list of the edges (from the ET) that intersect the current scan line. This list is updated for each scan line.

Figure 18.13 illustrates how these structures are used by the scan-line algorithm. We assume a simple case of two polygons that do not pierce each other.

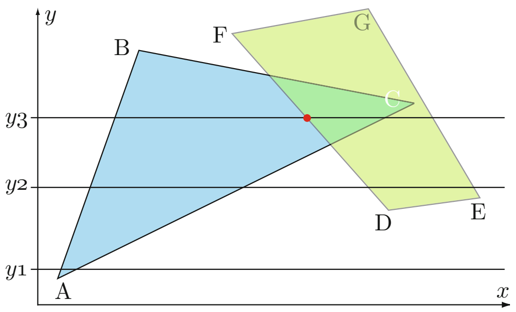


Figure 18.13: Two Polygons with Scan Lines.

For scan line $y1$, the AEL contains just the two edges AB and AC. The algorithm fills the short span between these edges with the fill color (or shade) for polygon ABC from the PT. This process involves switching the flag of polygon ABC twice, from false to true and back to false.

When the scan arrives at line $y2$, the AEL will contain the four edges AB, CD, DF, and EG (edge DE has already been deleted from the AEL, and edges BC and FG haven't been moved to the AEL yet). The algorithm fills the two resulting spans with the appropriate colors from the PT. The flag of polygon ABC switches values twice, as before, and the flag of polygon DEFG is also switched twice, but the two flags are never simultaneously true.

Scan line $y3$ illustrates the main decision our algorithm has to make. The intersection of this scan line with edge AB switches the flag of polygon ABC to true. The next intersection, of edge DF with the same scan line, switches the flag of polygon DEFG to true. The two flags are simultaneously true, implying that the scan line is now inside two polygons. The algorithm has to decide which polygon is closer to the observer at this point (indicated by a small red circle in the figure). The y coordinate of the point is $y3$ and its x coordinate is the x coordinate of the intersection of edge DF with scan line $y3$.

The equation of a plane (Section 4.4.1) is $Ax+By+Cz+D=0$. Once we substitute values for x and y, the equation yields a value for z. The scan-line visibility algorithm evaluates the plane equations of the two polygons at the point marked by a circle and selects the polygon for which the z value of the plane equation is the smaller (i.e., closer to the observer, who is located, as always, at $(0, 0, -k)$).

Assuming that polygon DEFG is closer to the observer, the algorithm switches the flag of ABC to false. There is now only one true flag, that of DEFG, and the algorithm fills the remainder of the current span (up to the intersection of scan-line $y3$ with edge EG) with the color of DEFG.

There may be more than two polygons sharing a scan line at a point, but the principle is the same. Whenever more than one flag is true at some point, evaluate the plane equations of all the relevant polygons at the point, select the one with the smallest z (in the simple case where the polygons do not penetrate each other there will be only one smallest z), set the flags of all the other polygons to false, and fill up the remainder of the span with the color of the selected polygon.

A complete scene rendering has to include the background, in addition to the objects. One way to include the background is to prepaint it the appropriate color or shade before the visible-surface algorithm starts. Another option is to switch to background painting between drawing spans of polygons.

It is possible to extend this method to curved surfaces by partitioning such a surface into polygons, each approximately flat.

18.6 Warnock's Algorithm

Warnock's algorithm is an example of an area-dubdivision method. The surfaces constituting the scene are assumed to be polygons and the idea is to divide the scene into smaller and smaller regions until a small enough region (sometimes as small as a single pixel) is reached where it is easy to decide which polygon (or part of a polygon) is the closest to the observer. The algorithm was developed as John Warnock's doctoral thesis (see [Warnock 69]) and it has a runtime of order $n\,p$, where n is the number of polygons and p is the number of pixels in the scene. We start with a simplified version.

- Start with the entire monitor screen. This is the current region.
- If the current region is "sufficiently simple," then scan convert the polygon in it and shade the appropriate pixels. Else, divide the current region into four quadrants, and check each recursively for being "sufficiently simple."

A region is sufficiently simple if it includes no more than a single polygon or if it consists of one pixel. In the former case, the polygon should be clipped to the region, be scan converted, and its pixels shaded. In the latter case, where the region consists of the single pixel at (p, q), the software should compute the z coordinates at point (p, q) of those polygons that include point (p, q) and select the polygon with the smallest z coordinate (i.e., the closest to the viewer). That polygon then determines the shading of the pixel at (p, q).

The algorithm is recursive, but even for high-resolution displays, of 1,024 $\times$ 1,024 pixels, the depth of the recursion does not exceed 10.

Here is the algorithm in more detail. We start with a scene (a set of flat polygons embedded in three-dimensional space) and scan convert them to end up with two-dimensional polygons on the screen. Figure 18.14 illustrates the relationships between polygons and regions and shows four cases. A surrounding polygon (a) is one that completely contains the region. An intersecting polygon (b) is only partly located in the region. A contained polygon (c) is fully contained in the region, and a disjoint polygon (d) lies completely outside the region.

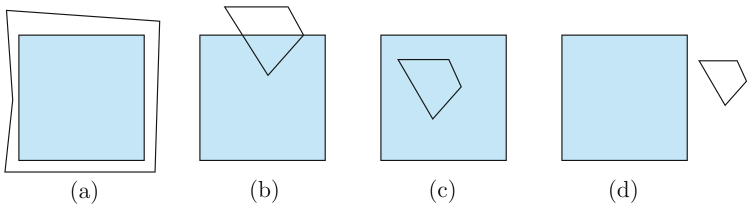


Figure 18.14: Relations between Polygons and Regions.

In a complex scene with many polygons, there may be several polygons in a region and their extents in space may overlap in complex ways. Warnock lists four cases where it is easy to identify the polygon that is closest to the observer and then shade it. If the current region does not conform to any of these cases, it is partitioned into four subregions and each is checked and processed recursively. The four cases are as follows:

1. The region is empty of polygons (all the polygons are disjoint from the region). This is the simplest case, and the region is simply filled with the background color.

2. Only one polygon intersects the region or is contained in it. This is also a simple case. The region is first filled with the background color, and then that part of the polygon contained in the region is scan converted and shaded.

3. A polygon surrounds the region, but no polygons intersect it or are contained in it. The region is shaded according to the properties of that polygon.

4. Several polygons may surround the region, may be contained in it, or may intersect it (they are referred to as the polygons associated with the region), but one of them surrounds the region such that it is in front of all the other ones. The region is shaded as in the previous case.

Case 4 is identified by the following test that is performed on all the polygons associated with the region. The plane equation of each polygon is derived and is computed at the four corners of the region. The result is four z coordinates. If there is a surrounding polygon whose four z coordinates are smaller (i.e., closer to the observer) than the z coordinates of any of the other polygons, then the test is a success, and it is easy to shade the region.

Figure 18.15a shows a simple example of this case. It is obvious that the large surrounding polygon is closer to the observer than the other two polygons. In part (b) of the figure, the left edge of the intersecting polygon is closer to the observer than the left edge of the surrounding polygon (the former polygon does not pierce the latter, but is located above it), so the test fails and the region has to be subdivided further.

Figure 18.16 is an example of this algorithm. It shows a simple scene that consists of two polygons. The entire display is subdivided several times and the number inside

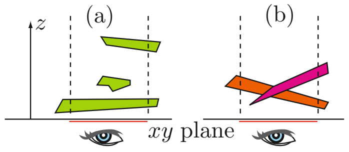


Figure 18.15: A Large Surrounding Polygon.

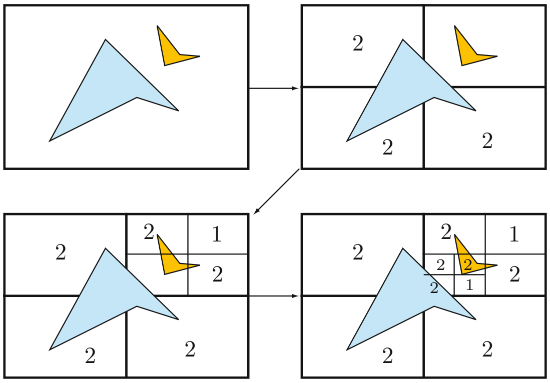


Figure 18.16: Warnock's Algorithm, an Example.

each region indicates the case (out of the four cases above) that corresponds to the region. A region without a number is subdivided in the next step.

18.7 Octree Methods

An octree is a special tree data structure that can represent a three-dimensional object. Because three-dimensional concepts, structures, and figures are often confusing, we start with a description of quadtrees, the two-dimensional cousins of octrees. References [Samet 90a,b] are detailed introductions to both quadtrees and octrees.

A quadtree is a tree data structure where a node is either a leaf or has exactly four children. A two-dimensional image can be stored in a quadtree by recursively creating and checking smaller and smaller quadrants. Given the bitmap of a monochromatic image, the process of constructing its quadtree starts by constructing a single node, the root of the final quadtree. If the image is uniform (all its pixels have the same color), then the quadtree consists of only the root, where the image color (0 for white and 1 for black) is stored. If the image is not uniform, the bitmap is partitioned into four quadrants that become the children of the root. A uniform quadrant is saved as a leaf child (containing the color of the quadrant) of the root. A nonuniform quadrant is saved as an (interior node) child of the root. Any nonuniform quadrants are then each recursively divided into four smaller subquadrants that are saved as four sibling nodes of the quadtree. Figure 18.17 shows a simple example.

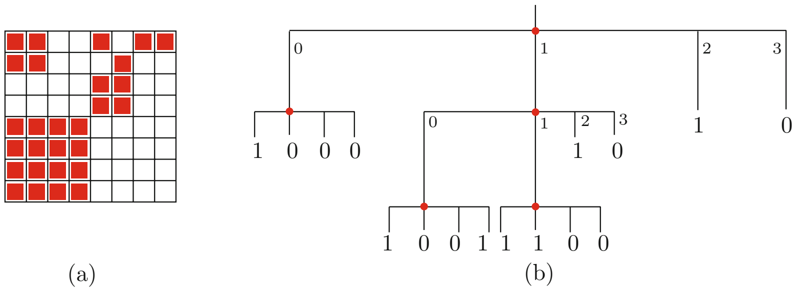


Figure 18.17: A Quadtree.

The 8×8 bitmap in 18.17a produces the 21-node quadtree of 18.17b. Sixteen nodes are leaves (each containing the color of one quadrant, 0 for white, 1 for black), and the other five (the circles) are interior nodes containing four pointers each. The quadrant numbering used is $\binom{0\,1}{2\,3}$, but see Exercise 18.4 for a more natural numbering scheme.

◇ **Exercise 18.4:** What is special about the particular quadrant numbering $\binom{1\,3}{0\,2}$?

The size of a quadtree depends on the complexity of the image. Assuming a bitmap size of $2^N \times 2^N$, one extreme case is a completely uniform image. The quadtree in this case consists of just one node, the root. The other extreme case is where each quadrant, even the smallest one, is nonuniform. The lowest level of the quadtree has, in such a case, $2^N \times 2^N = 4^N$ nodes. The level directly above it has a quarter of that number (4^{N-1}), and the level above that one has 4^{N-2} nodes. The total number of nodes in this case is $4^0+4^1+\cdots+4^{N-1}+4^N = (4^{N+1}-1)/3 \approx 4^N(4/3) \approx 1.33\times 4^N = 1.33(2^N\times 2^N)$. In this worst case the quadtree contains about 33% more nodes than the number of pixels (the bitmap size). Thus, representing such an image as a quadtree generates considerable expansion, but such images are rare.

An octree is the obvious extension of a quadtree to three dimensions. An octree is a data structure that can represent a three-dimensional object. In an octree, a node is either a leaf or has exactly eight children. The object is first surrounded by a bounding cube. It is then divided into eight octants, each nonuniform octant is divided into eight suboctants, and the process is repeated recursively until the subsuboctants reach a certain predetermined size. A flag of 1 (black) is stored in any leaf nodes that contain part of the object, and flags of 0 (white) are stored in all the empty leaves. Similar trees (N-trees) can, in principle, be constructed for N-dimensional objects.

Once the octree of an object is ready, it can be used to determine the visible parts of the object in parallel projection. Figure 18.18 illustrates the principle. The figure shows a simple case where the bounding cube is located on the positive side of the z axis with its axes parallel to the coordinate axes. The idea is to go over the cube from back (large z values) to front (small z). Thus, subcubes 1, 3, 5, and 7 of Figure 18.18 are considered first (in any order). For each subcube, the software has to go down the corresponding subtree of the octree, looking for leaf nodes. Each leaf node with a flag of 1 is parallel-projected onto the xy plane. Once the back layer of the cube is done, the software moves to the next layer (the one closer to the observer, in our example, the layer of subcubes 0, 2, 4, and 6) and proceeds in the same way.

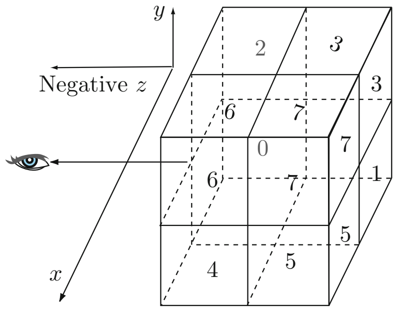


Figure 18.18: Octant Numbering.

◇ **Exercise 18.5:** Figure 18.19 shows a special case where the projection plane is perpendicular to the vector from one corner to the opposite corner of the bounding cube. Describe the best order of subcube processing in this case.

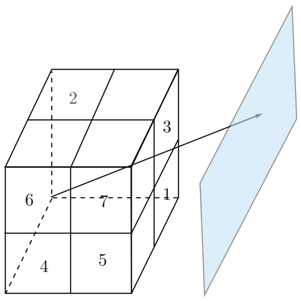


Figure 18.19: Nonstandard Projection.

18.8 Approaches to Curved Surfaces

The Z-buffer algorithm of Section 18.2 can also be applied to curved surfaces, but most of the other methods discussed earlier in this chapter are limited to polygonal surfaces. Part III of this book discusses several types of curved surfaces and this short section proposes general approaches to the problem of determining visibility of curved surface patches.

Perhaps the simplest approach to this problem is to partition a surface patch into many small facets, in the hope that each would be flat enough to allow an algorithm for polygonal surfaces to be applied. This can be done but it has the following shortcomings:

- The partitioning is time consuming.
- A curved surface may feature small regions that are highly curved. When an algorithm for polygonal surfaces is applied to such a region, the results are noticeably wrong.
- The boundaries between regions may result in artifacts due to the partitioning.

The spline surfaces of Chapter 12 feature tangent vector continuity at patch boundaries, and this feature makes it possible to develop a subdivision algorithm, similar to Warnock's Algorithm (Section 18.6). A spline surface patch P is partitioned recursively in the u and w parameters until a small patch is obtained, whose projection covers only one pixel. At this point, a Z-buffer-type method is applied to determine whether patch P is visible at that pixel. If yes, the pixel is shaded with a color determined by the surface properties of P.

Such a subdivision algorithm can be improved in various ways, but in principle it is slow. A better approach was taken by several researchers (see, for example, [Lane et al. 80]) who developed a scan-line based algorithm for curved surface patches. The main innovation of this method is to compute, for each scan line $y = y_0$, all the values of u and w for which the y component of the surface patch $\mathbf{P}(u,w) = \big(x(u,w), y(u,w), z(u,w)\big)$ equals y_0. This type of computation must be done numerically, and the developers of this algorithm employ for this purpose the well-known Newton–Raphson iterative method for finding roots (discussed in any text on numerical analysis). The Newton–Raphson method requires an initial guess of the solution, which for a scan-line method is easy; simply use the solution from the previous scan line.

In spite of its ingenuity, this scan-line method sometimes fails to find a solution, so other methods have to be tried.

An author in his book must be like God in the
universe, present everywhere and visible nowhere.
—Gustave Flaubert