VISIBLE SURFACE DETERMINATION

The task is to determine which parts of an object's surface are visible to the viewer, that is, which parts are not obscured by other objects. In most of the algorithms presented here, we will assume that the objects are represented by polygon meshes. However, some algorithms can also be adapted to other representations without major changes.

Visibility algorithms fall into two groups: *object-space methods* and *image-space methods*. We will discuss examples from both groups.

1. Using Normal Vectors

This is an object-space method suitable for mesh representations. Assume that objects are represented by closed, oriented manifold meshes, so that for each face we can readily obtain its vertices in counterclockwise order when viewed from outside.

Once the viewer's position in the scene has been chosen, the mesh faces can be divided into two groups:

- back-facing faces – the outward normal vector and the direction from the viewer to the face form an acute angle,
- front-facing faces – the outward normal vector and the direction from the viewer to the face form an obtuse angle.

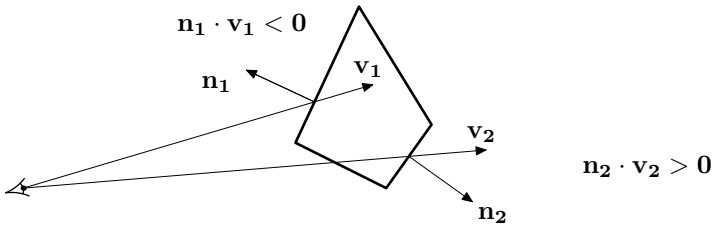


Figure 1. Front-facing and back-facing faces

The dot product tells us whether the angle is acute or obtuse; see the figure. Only front-facing faces can be visible, so back-facing faces need not be considered further.

If the scene consists of a single convex polyhedron, or several convex polyhedra whose projections do not overlap, this algorithm solves the visibility problem completely. In practice, scenes are usually much more complex. Face normals can therefore be used in a preprocessing step for more advanced algorithms, which then need to consider only front-facing faces.

Let us examine how to compute face normals in more detail. Let $A_0A_1 \dots A_{n-1}$ be the vertices of a polygon listed in counterclockwise order. We need a vector perpendicular to the polygon and pointing outwards from the polyhedron. In the ideal case, where the polygon is convex and all its vertices lie in one plane, we can take the direction vectors of any two consecutive edges and compute their cross product:

$$\mathbf{n} = (A_i - A_{i-1}) \times (A_{i+1} - A_i).$$

The vector $\mathbf{n}$ then gives the required normal direction. Thus, if $\mathbf{v}$ is the vector pointing from the viewer towards the face, the face $A_0A_1 \dots A_{n-1}$ is front-facing if $\mathbf{n} \cdot \mathbf{v} < 0$, and back-facing if $\mathbf{n} \cdot \mathbf{v} > 0$.

If we drop the convexity requirement, the polygon may have reflex interior angles. To obtain the correct normal direction, we must ensure that the cross product uses the direction vectors of

adjacent edges meeting at a convex vertex. We do this by choosing edges whose shared vertex has an extreme value (a maximum or minimum) of one coordinate among all the polygon's vertices.

The vertices of a polygon may also fail to lie exactly in one plane, owing to rounding errors or inherent inaccuracies in mesh construction. In this case, the normal direction depends on which pair of adjacent edges we choose. We can then take the average or sum of all such "normals" as a representative normal direction. If $A_0, A_1, \ldots, A_{n-1}$ are the polygon's vertices listed in counterclockwise order, the resulting vector is

$$\mathbf{n} = \sum_{i=0}^{n-1} (A_i - A_{i-1}) \times (A_{i+1} - A_i),$$

where indices are taken modulo n.

Remark 1. The normal of an object, such as a polygon, is often understood to be a unit vector with a specified orientation. In this section, we have not normalised the vectors because their lengths do not affect the visibility tests. It is therefore reasonable to omit this operation.

2. Z-Buffer

The *z-buffer*, also called the *depth buffer*, is an example of an image-space method. It is widely used because it is easy to implement. Together with polygonal representations and Phong shading (which you will encounter next semester), it forms the basis of many graphics applications. The Z-buffer can also be used effectively with other representations.

2.1. **Basic Algorithm.** Each object, such as a polygon, is projected along the z axis onto the xy plane (as if its z coordinate were "discarded") and rasterised. For each polygon, the following are computed:

(1) the coordinates (x, y) of all pixels covered by the polygon's projection,
(2) the original z coordinate corresponding to each (x, y) position,
(3) the colour (intensity) at each (x, y) position.

The z coordinates computed in step (2) are stored in the *z-buffer*, a two-dimensional array with the same dimensions as the rendering window. The colours computed in step (3) are stored in the *frame buffer*, which holds the image displayed in the window. The basic structure of the algorithm is as follows:

```
// initialisation:
for all x, y
    z-buffer[x, y] := maximum depth (an initialisation value)
    optionally initialise the frame buffer (fill it with the background colour)
// rendering:
for each polygon in the scene
    find all raster positions (x, y) covered by the polygon's projection
    for each relevant (x, y) position
        compute the corresponding z coordinate (such that (x, y, z) lies in the polygon)
        if z < z-buffer[x, y]
            z-buffer[x, y] := z
            frame-buffer[x, y] := colour (intensity) computed for the polygon at (x, y)
```

Now that the overall structure is clear, we can describe the z-buffer algorithm in more detail. Each polygon is processed using a horizontal *sweep line*, which moves from the polygon's minimum to its maximum y coordinate. For a given polygon and a fixed y value, we store the x coordinates of the endpoints of each segment where the polygon's projection intersects the current sweep line, together with the corresponding z coordinates. For a non-convex polygon, the intersection may consist of several segments. These data are sufficient if the entire polygon is rendered in a single colour. If shading is required, we must also determine the colour at each point, so the scanline data additionally include the colours at the segment endpoints.

```
for all x, y
   z-buffer[x, y] := maximum depth (an initialisation value)
   optionally initialise the frame buffer (fill it with the background colour)
for each polygon in the scene
   let y_min and y_max be the minimum and maximum y coordinates of pixels
         in the polygon's rasterised projection
   // construct the span list:
   for all raster values y ∈ ⟨y_min, y_max⟩
      find the intersection of the current sweep line and the polygon's projection
            (one or more segments)
      store the x and z coordinates of the segment endpoints in span-list[y]:
            x_left, x_right, z_left, z_right
      (if shading is required, also store the endpoint colours (intensities) I: I_left, I_right)
   // process the segments in the list:
   for all y ∈ ⟨y_min, y_max⟩
      for each segment in span-list[y]
         read x_left, x_right, z_left, z_right (and, if applicable, I_left, I_right)
         for all raster values x ∈ ⟨x_left, x_right⟩
            determine the z coordinate at (x, y) by linear interpolation
            if z < z-buffer[x, y]
               z-buffer[x, y] := z
               frame-buffer[x, y] := colour (intensity) I
```

As already mentioned, the main advantage of this visibility algorithm is its simplicity, which accounts for its popularity and widespread use. One disadvantage is its memory consumption. To address this, the scene can be processed in separate parts. Another limitation is that the basic z-buffer algorithm cannot handle translucent objects on its own: the z-buffer does not record which object lies immediately behind a translucent surface. One possible modification is to store, at each z-buffer entry, not only the z coordinate of the nearest object but also a list of pointers to all objects whose projections cover that pixel, sorted by their z coordinates at that position. Such a version clearly requires a great deal of memory. Also note that the z-buffer often renders pixels, and even entire objects, that are absent from the final image because nearer objects overwrite them later. This also occurs in some of the other algorithms discussed below.

2.2. **Z-Buffer with a Sweep Line (Scanline).** In this variant of the z-buffer algorithm, the scene is processed in parts whose projections occupy a window one pixel high and as wide as the original window. The z-buffer therefore contains one entry per column and is reinitialised after each row has been processed. This modification saves memory. In the implementation, the order of the loops is reversed: the outer loop iterates over y coordinates rather than polygons, and the buffers are initialised inside this loop.

2.3. **Scanline Algorithm with Intervals.** The idea is to determine visibility once within each interval of a scanline, rather than separately at every point. The following example illustrates this approach (see the figure).

Suppose the scene consists of two polygons. The scanline is the projection of a plane $y = k$, where the value is constant. The intersection of this plane with the scene may, for example, consist of two line segments. To determine which segment is visible when looking along the z axis, we project their endpoints onto the x axis. These projected endpoints divide the axis into several intervals $(\langle x_1, x_3 \rangle, \langle x_3, x_2 \rangle, \dots)$. If, for example, segment AB is visible at a point inside one of these intervals, then it is visible throughout that interval, whose boundaries are the projected endpoints. We therefore do not need further visibility tests at the other points of the segment within the interval; we only need to render them.

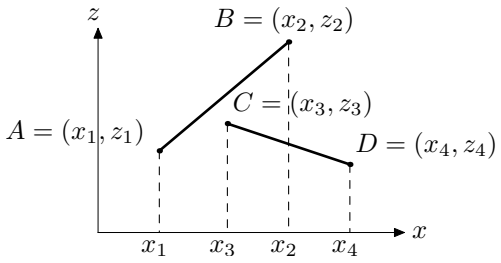


FIGURE 2. Intervals on a scanline

// scene preprocessing:
for each polygon
  record the scanlines it intersects (the y values it spans)
// rendering:
for each scanline (that is, for each raster value y)
  initialise the z-buffer and frame buffer
  for each relevant polygon
    find the intersection of its projection and the scanline (one or more segments)
  divide the scanline into intervals
  for each interval
    find the nearest segment (for example, by testing the interval's midpoint)
    render the segment within the interval, applying shading if required

This algorithm is faster than the scanline z-buffer algorithm only when the scene is relatively simple. Since the scenes being modelled are generally complex, implementing this refinement is usually not worthwhile.

2.4. **Ray Casting.** We have already encountered this algorithm in the context of CSG representations. Here, we can view it as a z-buffer approach in which the scene is divided into parts whose projections each occupy a single pixel of the rendering window.

3. Painter's Algorithms

A painter's algorithm renders objects much as a painter would paint them on a canvas: the most distant objects are drawn first, followed by nearer objects, which progressively overwrite distant, hidden parts. These algorithms require an analysis of the scene and are therefore classified as object-space methods.

3.1. **Depth-Sorting Algorithm.** In depth sorting, all objects are placed in a linear order, approximately according to their distance from the viewer. For example, if the scene is projected onto the xy plane, with the viewer looking along the negative z axis, objects are sorted by their z coordinate. In the following, we assume that objects are represented by polygon meshes. To determine the rendering order of two polygons, the following tests are performed; passing any one test is sufficient:

(1) if the polygons' z ranges do not overlap, their rendering order can be determined,
(2) if the bounding boxes of their projections onto the xy plane do not overlap, either rendering order is valid,
(3) if one polygon lies entirely in front of or behind the plane containing the other polygon, their rendering order can be determined,
(4) if the polygons' projections do not overlap, either rendering order is valid.

Test (4) is a more precise version of test (2). Both are included because test (2) is much faster and may make the more expensive test (4) unnecessary.

For test (3), we need the equation of the plane containing one of the polygons. First, we find a normal vector $\mathbf{n} = (n_1, n_2, n_3)$ as in the normal-based visibility method. We can then proceed in either of two ways. In the first approach, we compute the coefficient d so that $n_1x + n_2y + n_3z + d = 0$ is the required plane equation, by substituting the coordinates of a polygon vertex lying in the plane. We then test all vertices of the other polygon: their coordinates are substituted into the left-hand side of the plane equation. If all the resulting values have the same sign, the entire polygon lies on one side of the plane and the rendering order can be determined.

In the second approach, we use the plane equation $\mathbf{n} \cdot (X - P) = 0$, where P is a point in the plane, $X = (x, y, z)$ is a vector of variables, and $\cdot$ denotes the dot product. We substitute the coordinates of the other polygon's vertices for X and check whether the resulting values of the left-hand side have the same sign. The two approaches differ in the number of elementary additions and multiplications. The first approach requires some initial operations to find d, but then uses fewer operations when evaluating the expression at each vertex.

These tests do not determine an order for every pair of polygons. More complex tests are generally not implemented, since some configurations cannot be ordered at all, such as non-convex polygons or polygons with cyclic overlap. If a polygon cannot be inserted into the ordered list, it is split into two parts by the plane of another polygon. The resulting pieces can then be ordered relative to that polygon using test (3).

3.2. **Using a BSP Tree.** Instead of arranging objects in a linear order, we organise them in a binary tree. The scene is recursively partitioned and encoded in a BSP tree so that, for example, the leaves contain convex objects or other simple objects that can be rendered individually with little difficulty in determining visibility. Choosing suitable splitting planes is the most demanding task. It is often impossible to find a plane that separates the objects into two non-empty groups without cutting any object. In that case, an object must be split into two parts by the plane.

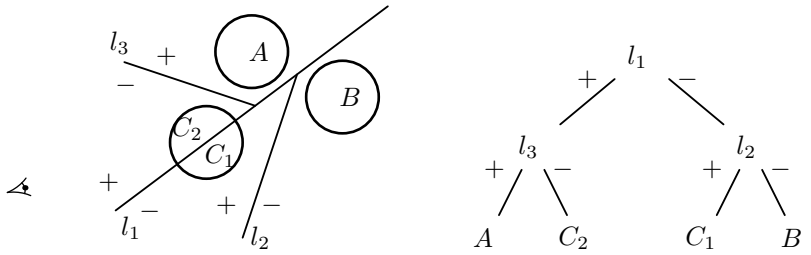


FIGURE 3. Organising a scene in a BSP tree

For polygonal representations, the leaves contain individual polygons, and the splitting planes are chosen to contain faces of the model. Some faces may be split into two smaller ones, increasing the total number of polygons. In practice, the total number of objects often approximately doubles, which is usually manageable.

Once the scene has been represented by a BSP tree, we choose the viewer's position and render the scene by recursively traversing the tree:

```
read the splitting plane at the root of the tree
if the viewer is in the positive half-space
   render the negative subtree first
   then render the positive subtree
if the viewer is in the negative half-space
   render the positive subtree first
   then render the negative subtree
```

In the example shown in the figure, we first render the scene in the negative subtree of node l_1. At node l_2, we again render the negative subtree first, whereas at node l_3, we render the positive subtree first. The objects are therefore rendered in the order B, C_1, A, C_2.

If the scene has been subdivided down to individual polygons, so that the leaves contain polygons rather than simple solid objects, and polygons are also stored at internal nodes, the polygon at each internal node is rendered between the two subtrees.

A major advantage is that the BSP tree is constructed independently of the viewer's position. This makes it fast to render animations in static scenes where the viewer moves through an unchanging environment, as in flight simulators and some computer games. The expensive pre-processing step, building the BSP tree, is performed at the beginning, followed by the interactive part of the program.

A painter's algorithm renders all objects in the scene. If rendering each pixel is expensive, for example because computing its colour is costly, the traversal can be reversed. After sorting, objects are rendered from front to back. An additional mask records whether each screen pixel has already been rendered; if it has, it is not rendered again.

3.3. **Using an Octree.** This approach is very similar to the previous one, but the scene is organised in an octree. It is often used when the input data are already represented by an octree. Once the viewer's position has been chosen, typically outside the scene, the octants are ordered for visibility, as in the BSP approach. The tree is then traversed in this order, rendering the corresponding subtrees.

4. Warnock's Area-Subdivision Algorithm

This algorithm follows a divide-and-conquer strategy. If the scene to be rendered within a window is sufficiently simple, it is rendered directly. Otherwise, the window is divided into four quadrants, each of which is processed as a separate window. The following cases are considered simple enough to handle directly:

- the window is empty – fill the entire window with the background colour;
- only one object projects into the window – render that object and fill the rest with the background colour;
- several objects project into the window, but one object covers the entire window and is closest throughout it – fill the window with that object's colour.

In all other cases, subdivision continues until one of these simple cases is reached or the window is reduced to a single pixel, which is then assigned the colour of the nearest object at that position.

5. Wireframe Rendering

Some applications require a wireframe view rather than a shaded object, for example the edges of a polygonal representation, with hidden edges removed. One way to achieve this is to use a visible-surface algorithm and render each polygon by filling its interior with the background colour and drawing its boundary.

A more specialised approach first determines the visible portions of the edges by clipping away the hidden parts, then renders the remaining edge segments.

We can speed up this process in much the same way as face normals were used to discard back-facing polygons. Edges are divided into three groups:

- back edges – shared by two back-facing faces,
- front edges – shared by two front-facing faces,
- silhouette edges – shared by one front-facing face and one back-facing face.

Back edges are invisible and need not be considered further. Front and silhouette edges are potentially visible, so their visible portions must be determined. We will briefly mention two approaches.



FIGURE 4. An object with a hole; silhouette edges are highlighted on the right

5.1. **Roberts' Algorithm.** All potentially visible edges are clipped against all faces in the scene. Each edge is compared with each face to find intersections and determine which portions of the edge are not hidden by the face. Finally, all the remaining edge segments are rendered.

5.2. **Appel's Algorithm.** Potentially visible edges are clipped using silhouette edges rather than faces, because an edge's visibility can change only where its projection crosses the projection of a silhouette edge. This algorithm is relatively difficult to implement because many special cases must be handled.

6. Rendering the Graph of a Function

A specialised, efficient algorithm, known as the floating horizon algorithm, can be used to render the graph of a function $z = f(x, y)$ over a rectangular domain $\langle x_{min}, x_{max}\rangle \times \langle y_{min}, y_{max}\rangle$.

Each point $(x, y, f(x, y))$ on the surface is projected onto a two-dimensional image. Its coordinates in the projection plane, or rendering window, are denoted by (x', y'). First, we render the isoparametric curves $(x, y_i, f(x, y_i))$ for several fixed values y_i, proceeding from front to back. Let $y_0, y_1, \ldots, y_n \in \langle y_{min}, y_{max}\rangle$ be ordered by distance from the viewer: the point $(0, y_0, 0)$ is nearest, followed by $(0, y_1, 0)$, and so on up to $(0, y_n, 0)$. For each x' coordinate in the window, we store the minimum and maximum rendered values y'_{min}, y'_{max}.

```
// render the curve (x, y_0, f(x, y_0)), x ∈ ⟨x_min, x_max⟩
for all x ∈ ⟨x_min, x_max⟩
   let (x', y') be the projection of (x, y_0, f(x, y_0))
   y'_min[x'] := y', y'_max[x'] := y'
   render (x', y')
// render the curves (x, y_i, f(x, y_i)), i = 0 ..., n, x ∈ ⟨x_min, x_max⟩
for i = 1, ..., n (in this order)
   for all x ∈ ⟨x_min, x_max⟩
      let (x', y') be the projection of (x, y_i, f(x, y_i))
      if y' < y'_min[x']
         y'_min[x'] := y'
         render (x', y')
      if y' > y'_max[x']
         y'_max[x'] := y'
         render (x', y')
```

The isoparametric curves $(x_i, y, f(x_i, y))$ are then rendered in a similar manner.

KAGDM FMFI UK BRATISLAVA
Email address: jana.pilnikova@fmph.uniba.sk