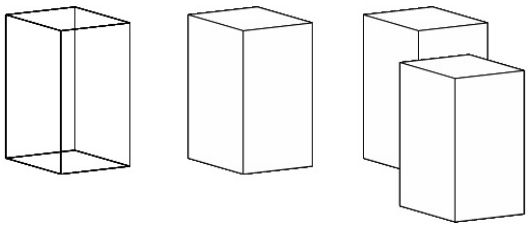
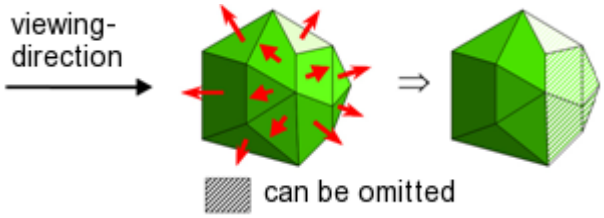
Visibility-Determination



To view scenes in a plausible and correct way, all parts which are not visible from a specific point of view, have to be omitted. In particular, these are the backsides of opaque objects or parts which are occluded by other objects. This is called hidden-line- or hidden-surface-removal.

Depending on the complexity of the scene, the types and data-structures of objects, the available hardware and requirements of the application, different *algorithms for visibility determination* can be used. When using object-space methods, the location of objects is compared to each other and only front (visible) parts are drawn. When using image-space methods, for every part of the image the visibility is determined separately. The following explanations do not take transparent objects into account.

Backface Detection (Backface Culling)



Backface Culling is ***not*** a complete visibility determination method. With it, simply all polygons, which have a normal vector pointing away from the viewer and therefore cannot be visible, are eliminated to reduce the cost of subsequent steps. On average, 50% of the polygons are removed by this procedure. To calculate which polygons will be removed when using orthographic projection, the scalar product of the view vector with the surface normal is calculated ($V_{view} \cdot N > 0 \rightarrow$ invisible); with perspective projection, the view point (x, y, z) is substituted into the plane equation ($Ax + By + Cz + D < 0 \rightarrow$ invisible). [assumptions as in "Polygon Lists".]

Z-Buffering (Depth Buffering)

The z-buffering algorithm solves the problem of visibility for a specific image resolution in the following way: For each point of the image, in addition to the color information, the information about the position of the displayed object is stored in an own storage. Since the viewing-direction is usually the (minus) z-direction, the x- and y-values correspond to the image-plane coordinates and only the z-value needs to be stored. So in addition to the image-buffer (frame buffer) an additional buffer is needed, which can store the z-value for each pixel. This buffer is called *z-buffer* or *depth-buffer*. Using a z-buffer all objects can be drawn in an arbitrary order. The z-values of the object which is to be drawn (which is a polygon in most cases) are calculated and compared to the z-values of the pixels onto which the object is going to be drawn. If a new z-value is nearer to the viewer (thus, usually larger), then the new object color is drawn there, replacing the old image value and also the z-value in the z-buffer is updated. Otherwise, the object is occluded and no action is necessary:

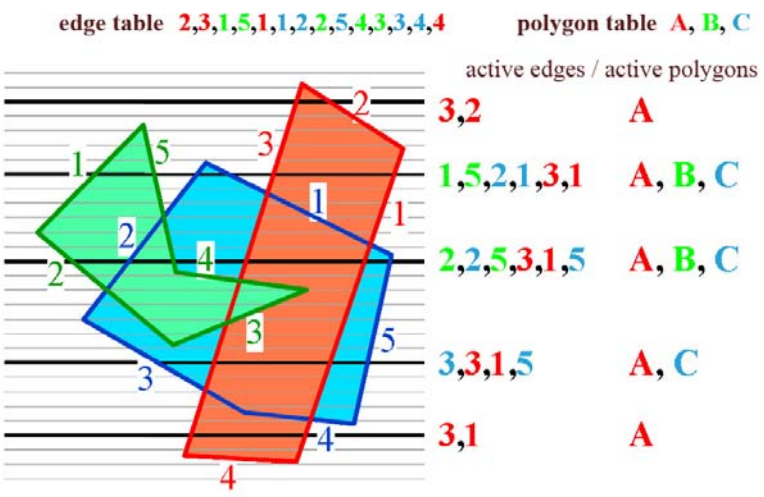
```
for all (x,y)  (* initializing the background *)
    depthBuff(x,y) = - ∞
    frameBuff(x,y) = backgroundColor
for each polygon P  (* loop over all polygons *)
    for each position (x,y) on polygon P
        calculate depth z
        if z > depthBuff(x,y) then
            depthBuff(x,y) = z
            frameBuff(x,y) = surfColor(x,y)
     (* else nothing! *)
```

The z-values can be efficiently calculated incrementally for flat polygons. The big advantage of z-buffering is that the objects (polygons) do not have to be sorted.

Scan-Line Method

With scan-line method the correct visibility is calculated for each horizontal line (in the example, from top to bottom, thus y decreasing). In doing so, it can be exploited that two consecutive pixel-lines (scan-lines), often have similar visibility properties.

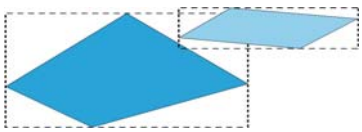
Starting with a table containing all edges of the polygons sorted by their maximum y-value and an associated polygon table, for every scan-line a list of active edges is created. This is done incrementally from the edges of the last scan-line: edges which have ended are eliminated and the next edges of the sorted edge table are checked, if they have already begun. After all intersection points from a scan-line with all edges are calculated in this way, they are sorted by their x-value (left to right). Between each pair of intersection points it has to be determined, which polygon is nearest to the viewer; this is the visible one and will be drawn along this scan-line.



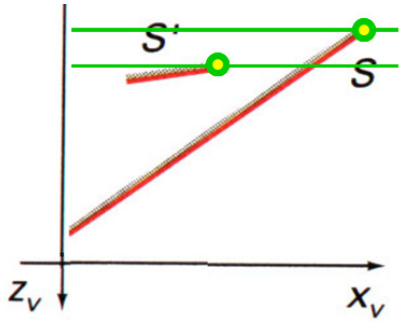
Depth-Sorting Method

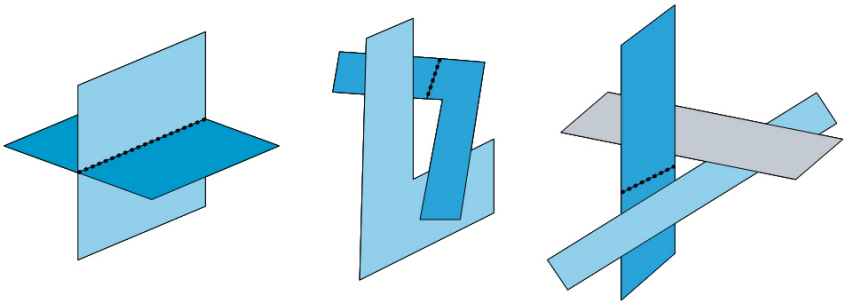
The principle of the depth-sorting algorithm is to sort all polygons from back to front and then draw them in this order. Since all hidden (occluded) parts are farer away than the occluding parts, the resulting image has correct visibility (*painter's algorithm*). The main expense is the sorting, which has to be done in such a way, that no polygon (partially) occludes any other one which comes later in the list (and is thus nearer to the viewer). This is done in two steps: first an approximate sorting is done quickly, and then it is checked, whether the sorting is right. If not, the list will be resorted.

1. Approximate sorting: sort the polygons according to the smallest z-value (farthest polygon).
2. Compare each polygon S with each (!) other polygon S':
   (* assuming that S lies behind S' *)
   (* now a series of tests follows with increasing complexity, until sorting is accepted as correct *)
   a. The biggest z-value of S is smaller than the smallest z-value of S'→ sorting correct.
   b. The x- or y-intervals do not cross → sorting correct.
   c. All vertices of S lie behind the plane of S' → sorting correct.
   d. All vertices of S' lie in front of the plane of S → sorting correct.
   e. The projections of S and S' onto the xy-plane do not intersect (see image to the right) → sorting correct.



   f. Sorting is probably wrong (see image where S' is hidden behind S) → swap and check sorting again. If the sorting is wrong again, then a special case has occured (see image) and has to be solved by splitting one of the polygons:

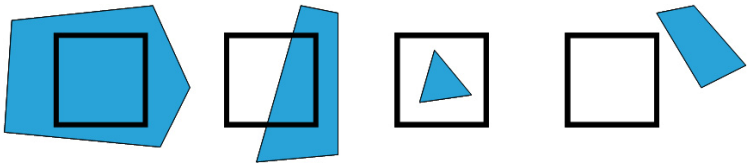




special cases that can only be solved by splitting a polygon

Area-Subdivision Method

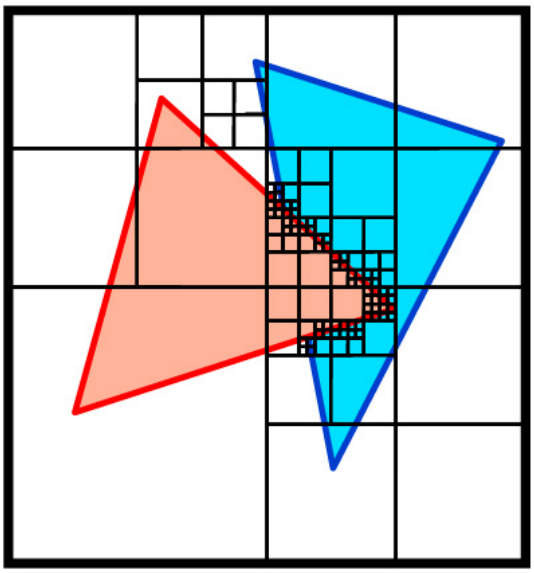
Similar to the quadtree representation of images, simple problems are solved in low resolution and more complicated ones are simplified by subdividing the area into 4 quarters. If applied recursively down to the image resolution, a pixel-accurate solution of the visibility problem is achieved.



To do so, a method is needed which quickly discovers the location of a polygon's projection onto a (quadratic) image-window. There are four possibilities: (1) the polygon covers the whole window, (2) the polygon lies partly inside and partly outside the window, (3) the polygon is completely inside the window or (4) the polygon lies completely outside the window.

There are three simple visibility decisions:

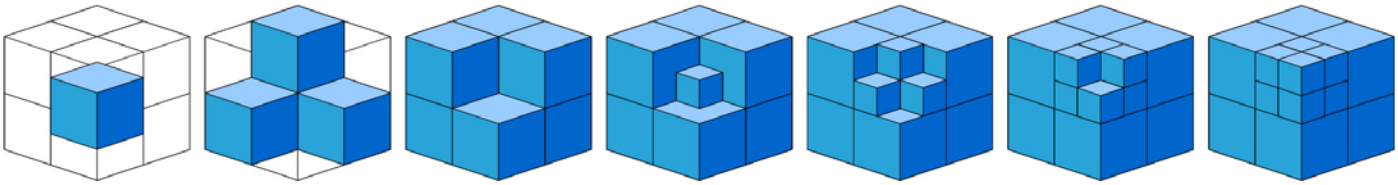
1. all polygons lie outside the window → done
2. only one polygon has an intersection with the window → draw this polygon
3. one polygon covers the whole window and lies in front of all other polygons in the window area → draw this polygon



If all three tests fail, then the window is divided into 4 quarters, which then will be processed recursively. Note that polygons which had already been completely outside a window, are also outside its child windows. Likewise, if a polygon has covered the whole window, all its child windows will also be covered by this polygon. If the child window has reached the size of only one pixel, the nearest polygon is chosen. As can be seen in the example, this happens along all edges at which the visibility changes.

Octree

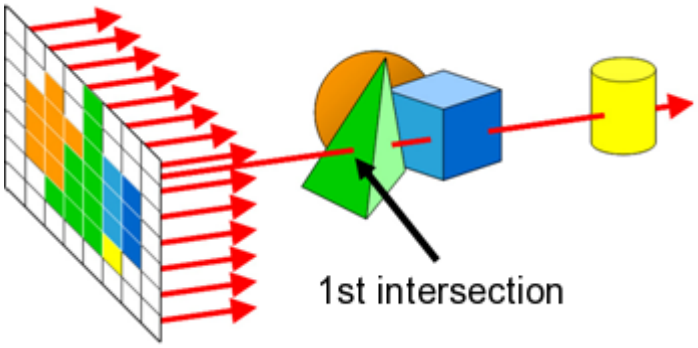
If the scene is represented as an octree instead of with polygons, then for every view direction the data structure already knows which side is in the front and which one is on the back. The data structure can be rendered recursively in the following way: first, in a cube, render the child cube which is farthest away from the viewer, then render the three next-nearer child cubes, then the next three, and finally the nearest one. One possible ordering can be seen in the example below:



Alternatively, the rendering can be done from front to back. Then all the areas on which something has been drawn already have to be stored, so that only those which will remain visible are drawn. The advantage of this data structure over others is that it implicitly knows which parts are in the front and which ones are in the back.

Ray-Casting

Ray-casting is a method of visibility determination that explicitly calculates for each pixel what is visible there. To do this, a *ray*, that is a straight line, is shot from every pixel into the scene. Thinking backwards, through this ray the light is transported from the scene onto the image plane or, respectively, the viewer. If this ray is intersected with all objects or polygons of the scene, a set of intersection points is obtained and the one which is nearest to the viewer is chosen. The color of the surface at this intersection point determines the color of the pixel through which the ray has been shot. If this is done for all pixels, the result for each pixel is the color of the nearest object, which is the visible one.



With ray-casting it is not only possible to render polygons in an easy way, but also other types of surfaces (like e.g. free-form-surfaces) for which the intersection with a line is computable. As explained later, usually the surface-normal is needed at the intersection points to be able to calculate useful shading. On the contrary, ray-casting is quite expensive since for every pixel (that are a few million for a full screen) an intersection has to be calculated for every object (and these can by thousands to millions). That is why an efficient implementation of the intersection test and further optimizations are necessary.

Ray-casting is a simple variant of ray-tracing, with which many other optical effects can be simulated. This will be explained in later chapter.

```
Ray-Casting =
for each pixel of the image-plane:
  generate a line through the pixel in viewing-direction ("viewing-ray")
  intersect the ray with all objects
  choose the nearest intersection point to the viewer
  color the pixel with the color of the surface at this intersect. point
```

Classification of the Algorithms

Let us categorize the algorithms according to if they operate in object- or image-space. This is sometimes ambiguous, but overall the following holds:

Object-space methods: backface culling, depth sorting, octree
Image-space methods: z-buffering, scanline method, area subdivision method, ray-casting