Hidden Surfaces

Introduction

We must determine what is visible within a scene from a chosen viewing position

For 3D worlds this is known as **visible surface detection** or **hidden surface elimination**

Importance

- Drawing polygonal faces on screen consumes CPU cycles
  - Illumination
- We cannot see every surface in scene
  - We don’t want to waste time rendering primitives which don’t contribute to the final image.

Importance

- Opaque objects that are closer to the eye and in the line of sight of other objects will block those objects or portions of those objects from view.
- In fact, some surfaces of these opaque objects themselves are not visible because they are eclipsed by the objects' visible parts.
- The surfaces that are blocked or hidden from view must be "removed" in order to construct a realistic view of the 3D scene (see Fig. 1-3 where only three of the six faces of the cube are shown).
- The identification and removal of these surfaces is called the hidden-surface problem.

Importance

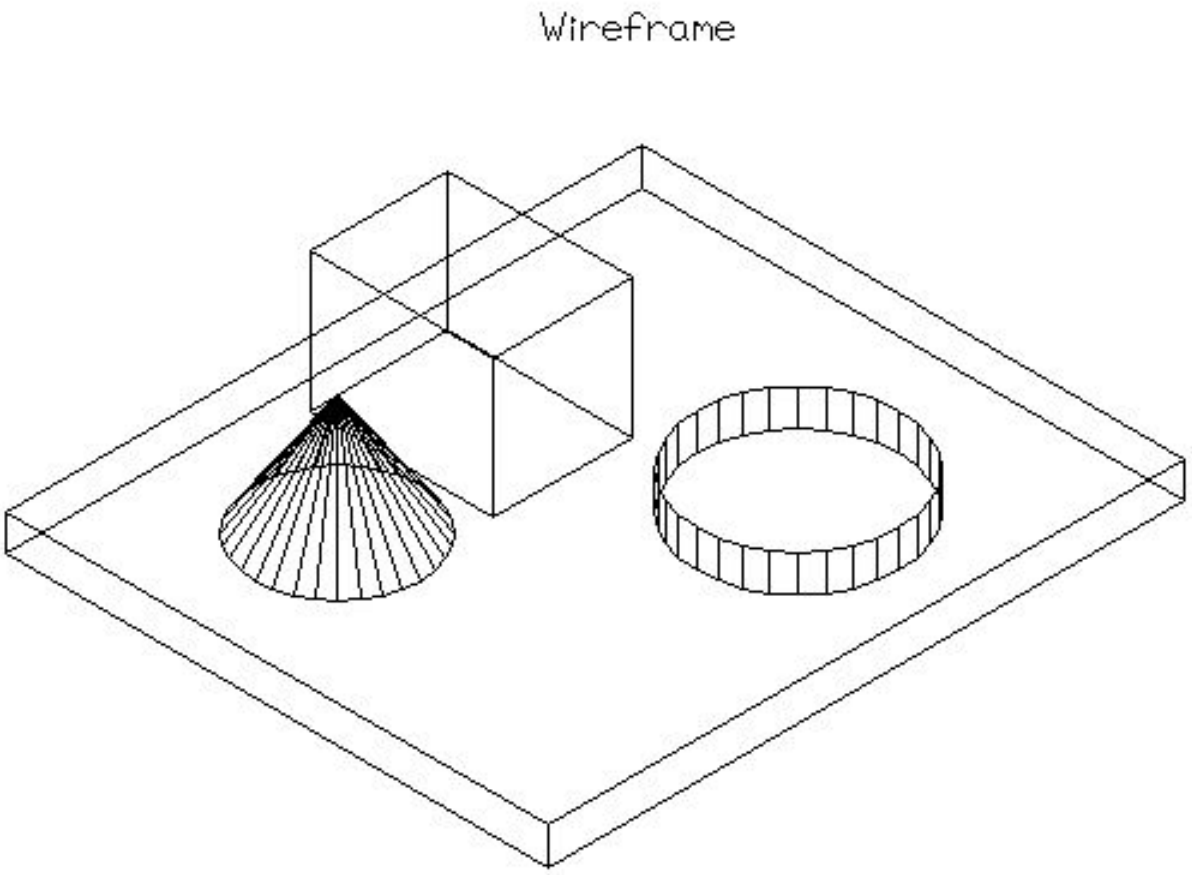
- We assume that all coordinates (x, y, z) are described in the normalized viewing coordinate system (Chap. 8).
- Any hidden-surface algorithm must determine which edges and surfaces are visible either from the center of projection for perspective projections or along the direction of projection for parallel projections.
- The question of visibility reduces to this:
- given two points P1(xl ,yl ,zl) andP2(x2,y2,z2), does either point obscure the other? This is answered in two steps:
  1. Are P1 and P2 on the same projection line?
- 2. If not, neither point obscures the other. If so, a depth comparison tells us which point is in front of the other.

Importance

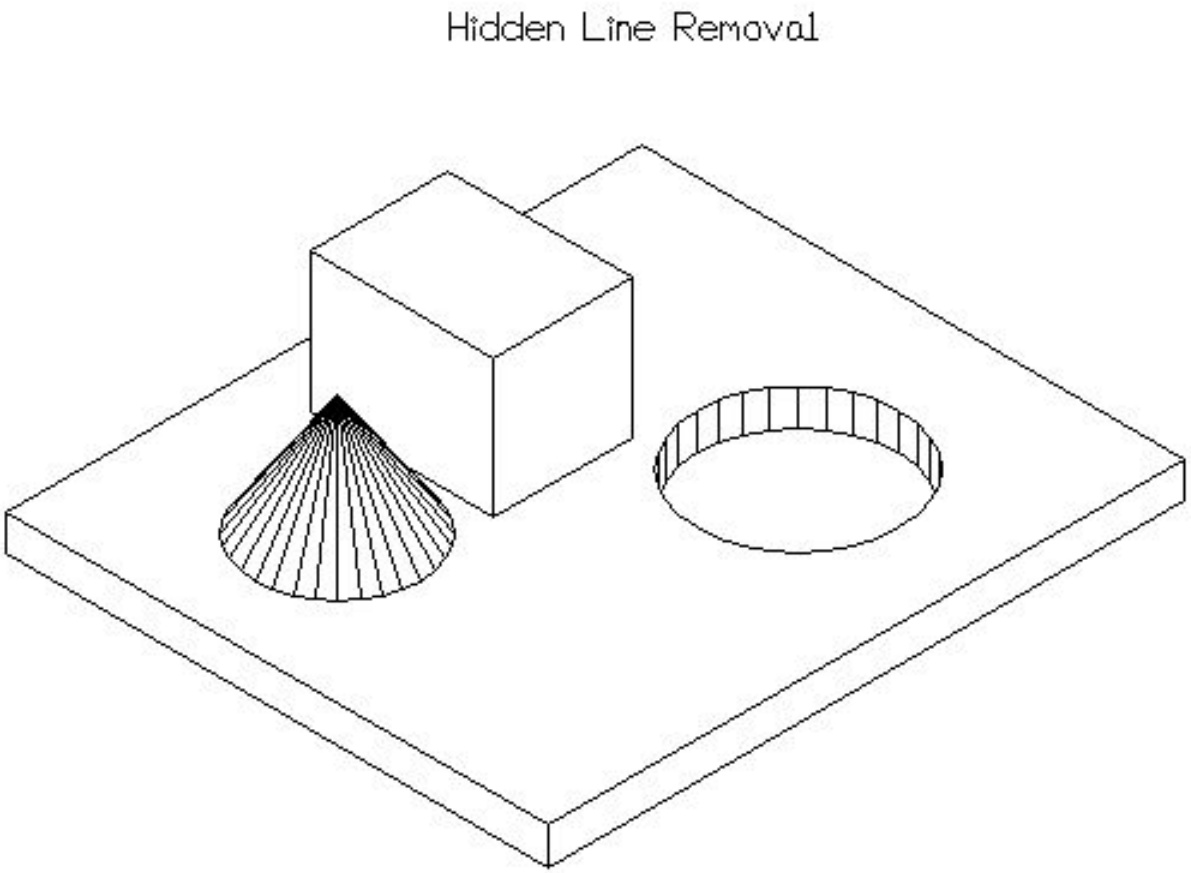
- For an orthographic parallel projection onto the xy plane, Pj and P2 are on the same projector if Xi = x2 and jj = y2. In this case, depth comparison reduces to comparing Zj and z2. If Zj < z2, then Pj obscures P2

- For a perspective projection [see Fig. 10-1(6)], the calculations are more complex (Prob. 10.1). However, this complication can be avoided by transforming all three-dimensional objects so that parallel projection of the transformed object is equivalent to a perspective projection of the original object (see Fig. 10-2). This is done with the use of the perspective to parallel transform Tp

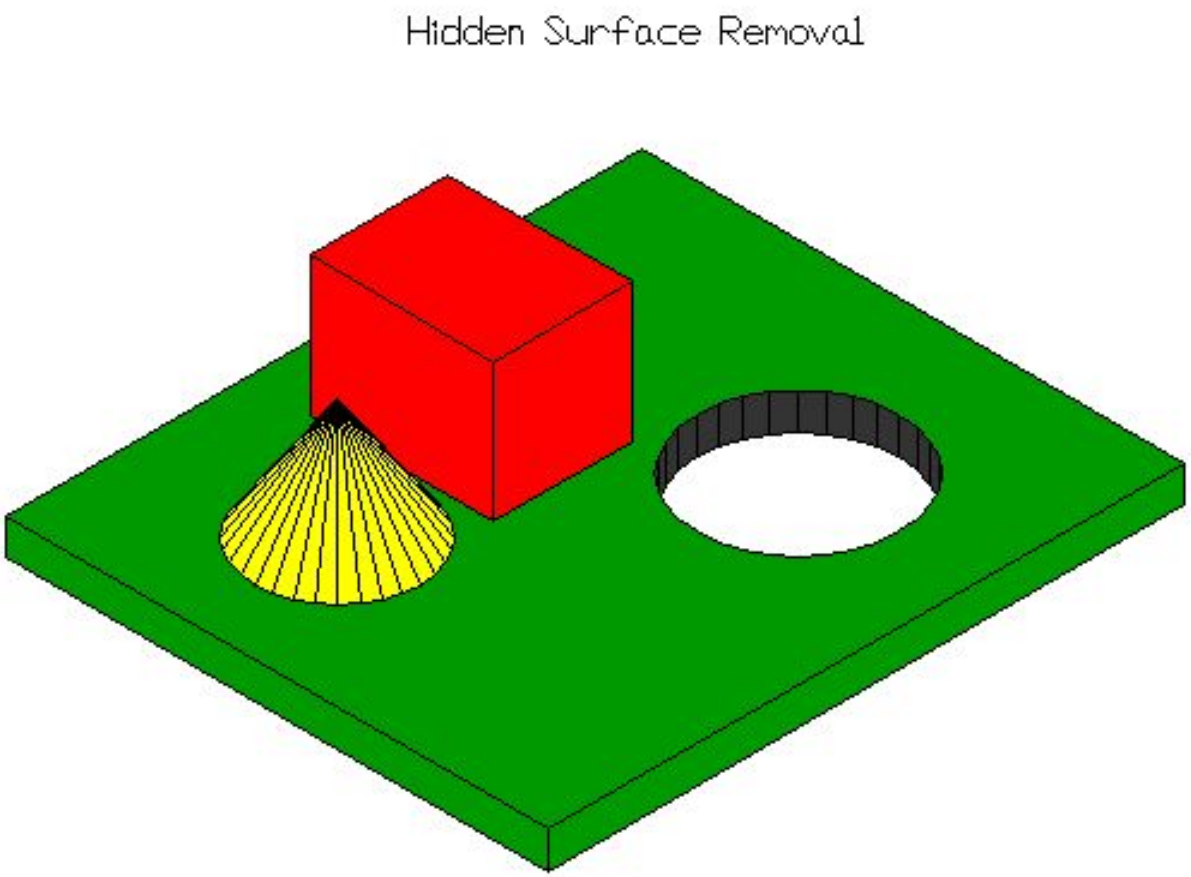
Hidden Lines



Hidden Lines Removed



Hidden Surfaces Removed



Hidden Surface Removal

- Goal: Determine which surfaces are visible and which are not.
- Other names:
  - **Visible-surface detection**
  - **Hidden-surface elimination**
- Display all visible surfaces, do not display any occluded surfaces.

Two Main Approaches

- **Object Space Methods**

  surface visibility is determined using continuous models in the object space (or its transformation) without involving pixelbased operations

  –Compares objects and parts of objects to each other within the scene definition

  –Operates on 3D object entities (vertices, edges, surfaces).

- **Image Space Methods:**

  pixel grid is used to guide the computational activities that determine visibility at the pixel level

  – Visibility is decided point- by-point at each pixel position on the projection plane.

  – Operates on 2D images (pixels).

Two Main Approaches

Object Space Method:

For each object in the scene do

Begin

1. Determine those part of the object whose view is unobstructed by other parts of it or any other object with respect to the viewing specification.

2. Draw those parts in the object color.

End

Two Main Approaches

Image Space Method:

For each pixel in the image do

Begin

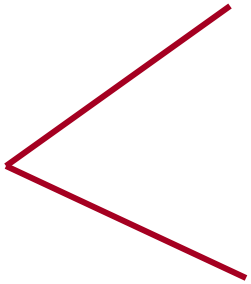
1. Here positions of various pixels are determined. It is used to locate the visible surface instead of a visible line. Each point is detected for its visibility. If a point is visible, then the pixel is on, otherwise off. Determine the object closest to the viewer that is pierced by the projector through the pixel
2. Draw the pixel in the object colour.

End

Different Visible Surface Detection Methods

- Back face Culling
- Hidden Object Removal: Painters Algorithm
- Z-buffer
- Scan-line
- Subdivision/ Warnock Algorithm
- Atherton-Weiler
- BSP Tree

Visible Surface Detection



Object space methods ex: back-face, painters algorithm

Image space methods ex: z-buffer, scan-line, subdivision

Z-Buffering

- We say that a point in display space is "seen" from pixel (x,y) if the projection of the point is scan converted to this pixel (Chap. 3).
- The Z-buffer algorithm essentially keeps track of the smallest z coordinate (also called the depth value) of those points which are seen from pixel (x, y).
- These Z values are stored in what is called the Z buffer.
- Let Z_{buf} (x, y) denote the current depth value that is stored in the Z buffer at pixel (x, y). We work with the (already) projected polygons P of the scene to be rendered. The Z-buffer algorithm consists of the following steps.

- 1. Initialize the screen to a background color. Initialize the Z buffer to the depth of the back clipping plane. That is, set

$$Z_{\text{buf}}(x, y) = Z_{\text{back}}, \qquad \text{for every pixel } (x, y)$$

Z-Buffering

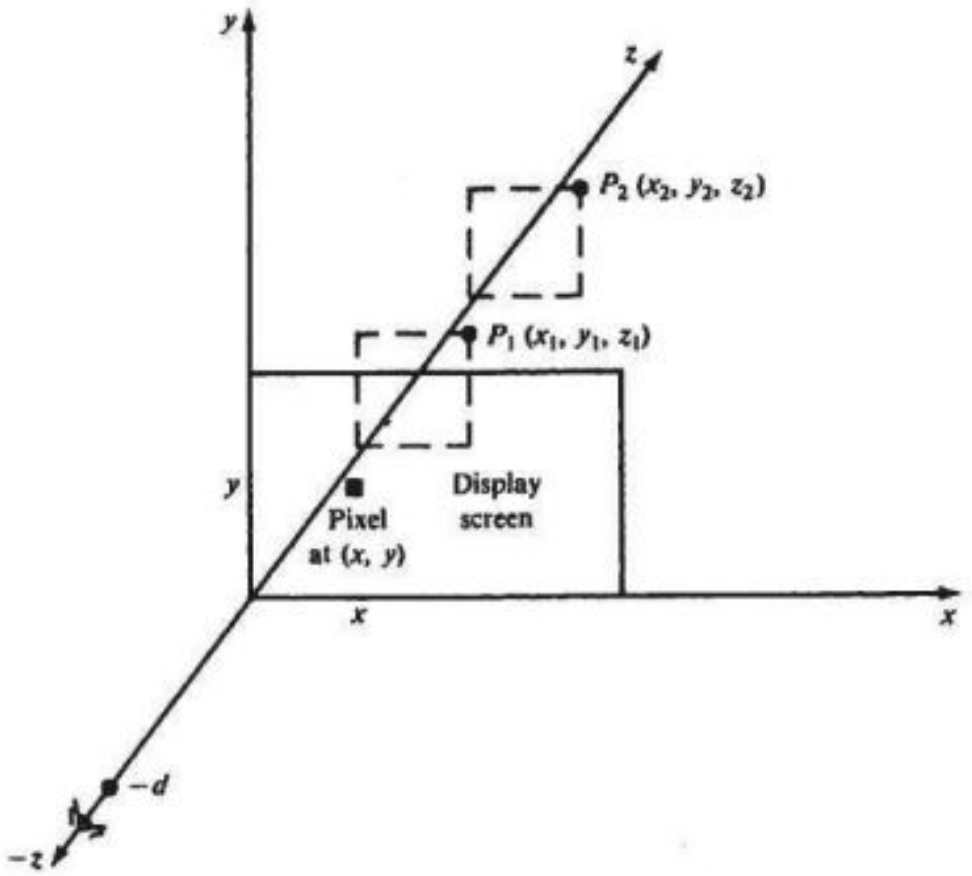
- Scan-convert each (projected) polygon P in the scene (Chap. 3) and during this scan-conversion process, for each pixel (x, y) that lies inside the polygon

(*a*) Calculate $Z(x, y)$, the depth of the polygon at pixel (x, y).

(*b*) If $Z(x, y) < Z_{\text{buf}}(x, y)$, set $Z_{\text{buf}}(x, y) = Z(x, y)$ and set the pixel value at (x, y) to the color of the polygon P at (x, y). In Fig. 10-3, points P_1 and P_2 are both scan-converted to pixel (x, y); however, since $z_1 < z_2$, P_1 will obscure P_2 and the P_1 z value, z_1, will be stored in the Z buffer.

- Although the Z-buffer algorithm requires Z-buffer memory storage proportional to the number of pixels on the screen, it does not require additional memory for storing all the objects comprising the scene. In fact, since the algorithm processes polygons one at a time, the total number of objects in a scene can be arbitrarily large.

Z-Buffering: IS



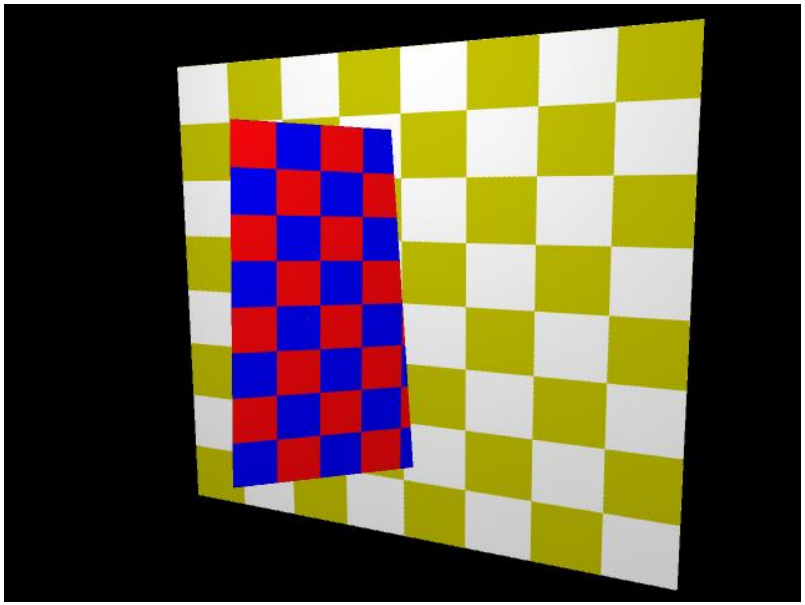
Z-Buffering: IS

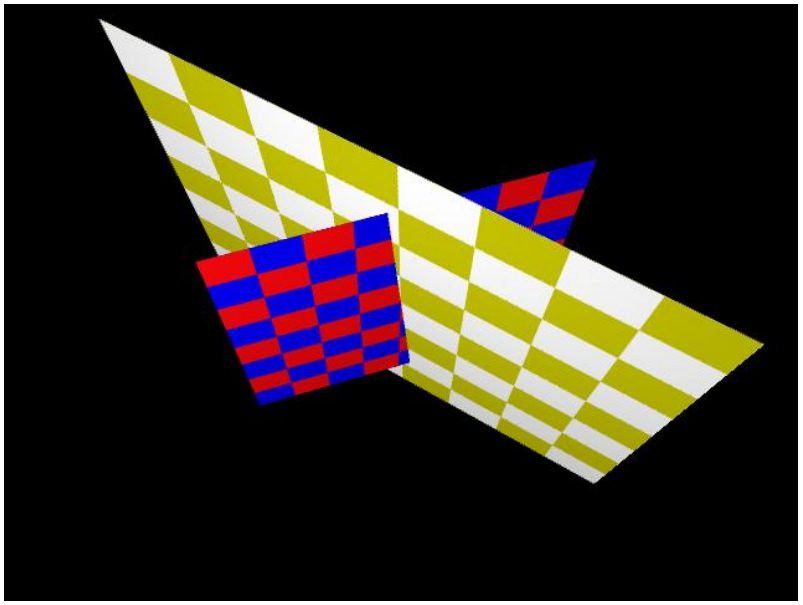
- Visible Surface Determination Algorithm.
- Determine which object is visible at each pixel.
- Order of polygons is not critical.
- Works for dynamic scenes.

Basic idea:

- Rasterize (scan-convert) each polygon, one at a time
- Object depth is usually measured from the view plane along the z-axis of a viewing system..
- Replace pixel with new color if z value is smaller. (i.e., if object is closer to eye)

Example





Goal is to figure out which polygon to draw based on which is in front of what. The algorithm relies on the fact that if a nearer object occupying (x,y) is found, then the depth buffer is overwritten with the rendering information from this nearer surface.

Z-buffering

- Need to maintain:
  - **Frame buffer** - contains color values for each pixel
  - **Z-buffer**- contains the current value of z for each pixel
- No sorting of objects required.
- Additional memory is required for the z-buffer.

Z-Buffering Algorithm

1. Initially each pixel of the z-buffer is set to the maximum depth value [infinite].
2. The image buffer is set to the background color.
3. Surfaces are rendered one at a time.
4. For the first surface, the depth value of each pixel is calculated.
5. If this depth value is smaller than the corresponding depth value in the z-buffer (ie. it is closer to the view point), both the depth value in the z-buffer and the color value in the image buffer are replaced by the depth value and the color value of this surface calculated at the pixel position.
6. Repeat step 4 and 5 for the remaining surfaces.
7. After all the surfaces have been processed, each pixel of the image buffer represents the color of a visible surface at that pixel.

Z-Buffering: Algorithm

The z-buffer algorithm:

```
for each polygon P
    for each pixel (x, y) in P
            compute z_depth at x, y
            if z_depth < z_buffer (x, y)
            then set_pixel (x, y, color)
                    z_buffer (x, y) = z_depth
```

Z-Buffer Advantages

- Easy to implement
- No sorting of objects
- Diversity of primitives – not just polygons.
- Unlimited scene complexity
- Don’t need to calculate object-object intersections.
- Buffer may be saved with image for re-processing
- Amenable to scan-line algorithms
- Can easily resolve visibility cycles

Z-Buffer Disadvantages

- Waste time drawing hidden objects
- Requires a lot of memory
- Requires re-calculations when changing the scale
- Does not do transparency easily
- Prone to aliasing. Shadows are not easy

BACK-FACE REMOVAL

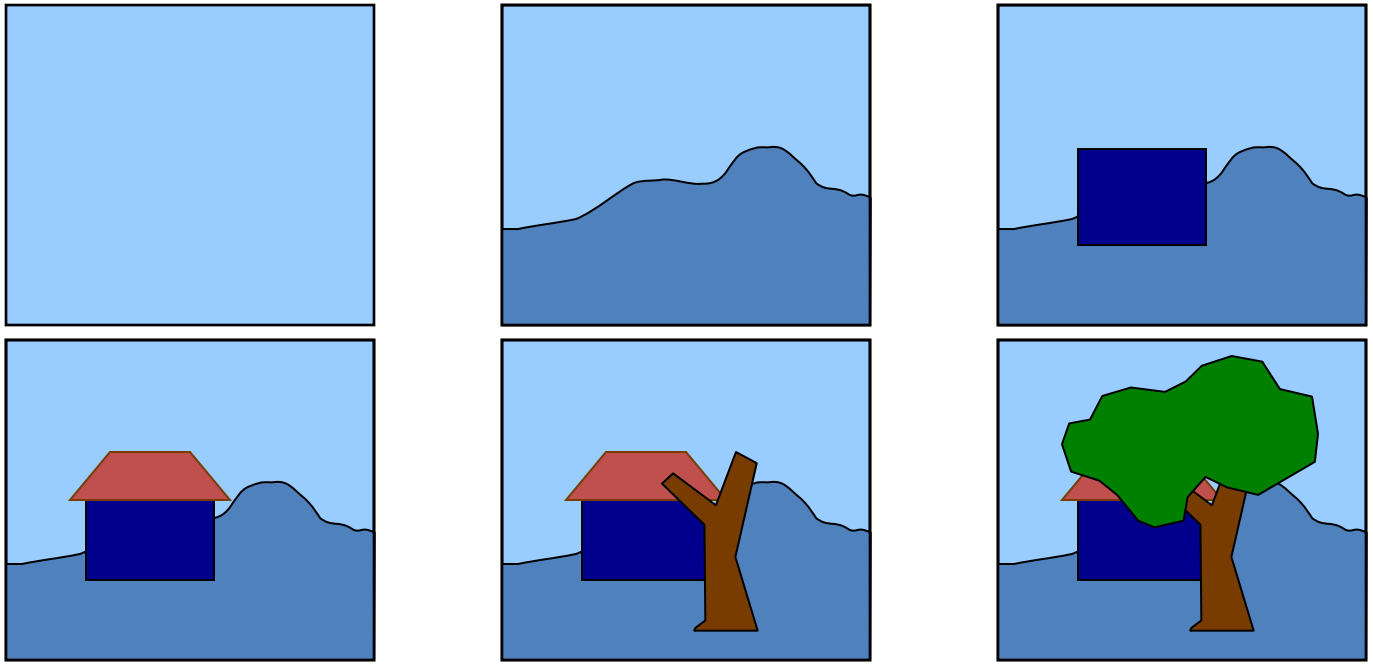
- Object surfaces that are orientated away from the viewer are called back-faces.
- The back-faces of an opaque polyhedron are completely blocked by the polyhedron itself and hidden from view.
- We can therefore identify and remove these back-faces based solely on their orientation without further processing (projection and scan-conversion) and without regard to other surfaces and objects in the scene.

BACK-FACE REMOVAL

- Let N = {A, B, C} be the normal vector of a planar polygonal face, with N pointing in the direction the polygon is facing.
- Since the direction of viewing is the direction of the positive z axis (see Fig. 10-3), the polygon is facing away from the viewer when $C > 0$ (the angle between N and the z axis is less than 90°).
- The polygon is also classified as a back-face when $C = 0$, since in this case it is parallel to the line of sight and its projection is either hidden or overlapped by the edge(s) of some visible polygon(s). Although this method identifies and removes back-faces quickly it does not handle polygons that face the viewer but are hidden (partially or completely) behind other surfaces. It can be used as a preprocessing step for other algorithms.

Painter's Algorithm

- Object-space algorithm
- Draw surfaces from back (farthest away) to front (closest):
  - Sort surfaces/polygons by their depth (*z* value)
  - Draw objects in order (farthest to closest)
  - Closer objects paint over the top of farther away objects



Painter’s Algorithm

- THE PAINTER'S ALGORITHM Also called the depth sort or priority algorithm
- the painter's algorithm processes polygons as if they were being painted onto the view plane in the order of their distance from the viewer.
- More distance polygons are painted first. Nearer polygons are painted on or over more distance polygons, partially or totally obscuring them from view.
  The key to implementing this concept is to find a priority ordering of the polygons in order to determine which polygons are to be painted (i.e., scan-converted) first.
- Any attempt at a priority ordering based on depth sorting alone results in ambiguities that must be resolved in order to correctly assign priorities.

Painter's Algorithm

- Assigning Priorities We assign priorities to polygons by determining if a given polygon P obscures other polygons. If the answer is no, then P should be painted first. Hence the key test is to determine whether polygon P does not obscure polygon Q.

The z extent of a polygon is the region between the planes $z = z_{min}$ and $z = z_{max}$ (Fig. 10-5). Here, z_{min} is the smallest of the z coordinates of all the polygon's vertices, and z_{max} is the largest.

Similar definitions hold for the x and y extents of a polygon. The intersection of the x, y, and z extents is called the *extent*, or bounding box, of the polygon.

- Testing Whether P Obscures Q Polygon P does not obscure polygon Q if any one of the following tests, applied in sequential order, is true.

Painter's Algorithm

Test 0: the z extents of P and Q do not overlap and $z_{Q_{max}}$ of Q is smaller than $z_{P_{min}}$ of P. Refer to Fig. 10-6.

Test 1: the y extents of P and Q do not overlap. Refer to Fig. 10-7.

Test 2: the x extents of P and Q do not overlap.

Test 3: all the vertices of P lie on that side of the plane containing Q which is farthest from the viewpoint. Refer to Fig. 10-8.

Test 4: all the vertices of Q lie on that side of the plane containing P which is closest to the viewpoint. Refer to Fig. 10-9.

Test 5: the projections of the polygons P and Q onto the view plane do not overlap. This is checked by comparing each edge of one polygon against each edge of the other polygon to search for intersections.

Painter's Algorithm

1. Sort all polygons into a polygon list according to z_{max} (the largest z coordinate of each polygon's vertices). Starting from the end of the list, assign priorities for each polygon P, in order, as described in steps 2 and 3 (below).

   2. Find all polygons Q (preceding P) in the polygon list whose z extents overlap that of P (test 0).
   3. For each Q, perform tests 1 through 5 until true.

      (a) If every Q passes, scan-convert polygon P.

      (b) If false for some Q, swap P and Q on the list. Tag Q as swapped. If Q has already been tagged, use the plane containing polygon P to divide polygon Q into two polygons. Q_1 and Q_2 [see Fig. 10-4(b)]. The polygon-clipping techniques described in Chap. 5 can be used to perform the division. Remove Q from the list and place Q_1 and Q_2 on the list, in sorted order.

Sometimes the polygons are subdivided into triangles before processing, thus reducing the computational effort for polygon subdivision in step 3.