10

Hidden Lines and Hidden Surfaces

10.1 INTRODUCTION

Whenever 3D objects are projected on 2D view plane, lines and surfaces that are closer to viewer block the lines and surfaces that are far from the viewer. The blocked or hidden surfaces must be removed in order to render a realistic image. The identification and removal of those surfaces is called hidden surface problem. The solution involves the determination of depth and visibility for all the points/lines/surfaces in the picture. Further, it may be possible that hidden lines/ hidden surfaces may occur when a 3D object is viewed directly.

We are interested to determine which lines or surfaces of the objects are visible, along the lines of projection in case of parallel projection or from centre of projection in case of perspective projection, so that we can display only *visible lines* or *surfaces*. This process is called **visible line/visible surface determination**, or **hidden line/hidden-surface elimination.**

A number of algorithms have been derived for identification of visible objects based upon different applications. Some methods are more efficient requiring more memory and more processing time. Some of these are applicable to special type of objects, but none of them is the best.

Based upon whether to deal with objects directly or with their project images, visible-surface algorithms are broadly classified as:

Object-space Method

These methods are implemented on physical coordinate system in which the object is described. These methods are independent of display devices. Effort required for implementation of such methods is a function n^2, where n is the number of objects.

Image-space Method

These methods are implemented on screen coordinate system. These methods depend on the resolution of display devices. The effort is function of $n.N$, where N is the number of pixels in the display and n is the number of objects.

Generally, $n.N$ is larger than n^2 . Most visible surface algorithms used image-surface algorithms. These methods are more efficient than object-space method because it is easier to take advantage of coherence in a raster scan implementation of an image space algorithm.

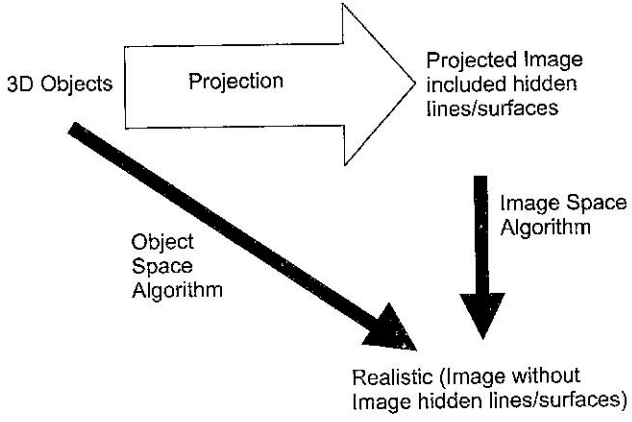


FIGURE 10.1 Illustration to Achieve Realistic image after Removal hidden lines/hidde- Surfaces Through Image Space Algorithm and Object Space Algorith-

Principle of Hidden-surface Elimination Methods

All hidden lines/surface elimination methods based upon principle whenever a picture contains opaque object and surfaces, those that are closer to the eye are in the line of sight of other objects will block those object from view.

Given two points $P1(x1, y1, z1)$ and $P2(x2, y2, z2)$. To check either point obscures the other? In case $P1$ and $P2$ are on same projection line i.e., for an orthographic projection on to the xy plane, and if $z1<z2$, then $P1$ obscures $P2$, as shown in Fig. 10.2(a). In case of perspective projection, 10.2(b), the calculations are more complex. Here we can take the help of perspective to parallel transformation.

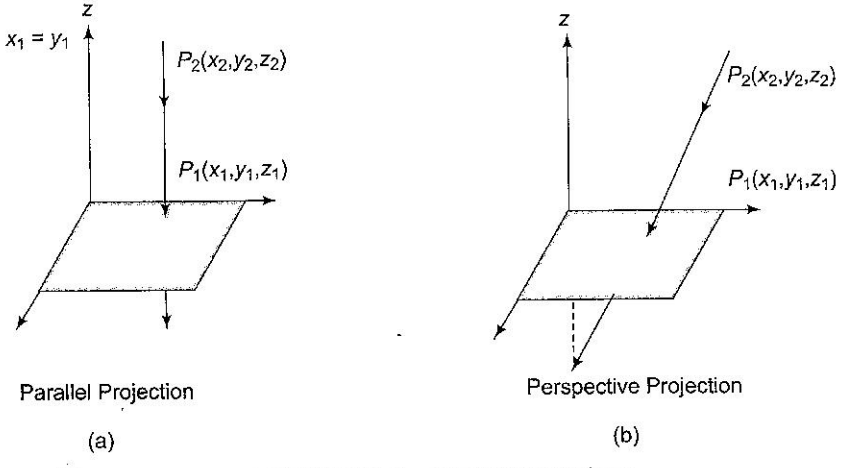


FIGURE 10.2 Depth Comparison

All hidden line/hidden surface algorithms involve sorting. The order in which sorting of the geometric coordinates occur is generally immaterial to the efficiency of the algorithms. The principal sort is based on the geometric distance of a volume, surface, edge or point from the viewpoint. After establishing the distance or depth priority, it remains to sort laterally and vertically to determine whether in fact an object is observed by those closer to the viewpoint. The efficiency of a hidden line/ hidden surface algorithm depends significantly on the efficiency of the sorting process. Coherence i.e., the tendency for the characteristics of a scene to be locally constant, is used to increase the efficiency of the sort.

10.2 Z-BUFFER ALGORITHM (DEPTH BUFFER ALGORITHM)

The z-buffer is one of the simplest algorithms of the hidden surface removal. The technique was originally proposed by Catmull and is an image space method. The z-buffer is a simple extension of the frame buffer idea. A frame buffer is used to store the attributes (intensity) of each pixel in image space. The z-buffer is a separate depth buffer used to store the z-coordinate or depth of every visible pixel in image space as illustrated in Fig. 10.3(a) and 10.3(b).

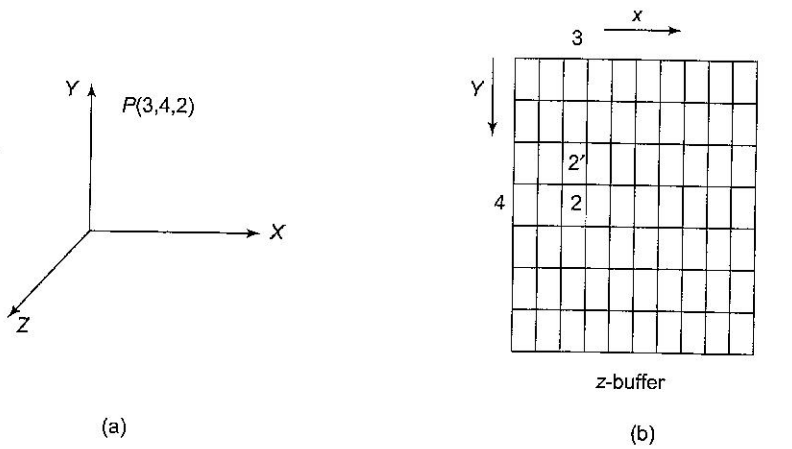


FIGURE 10.3 Representation of a 3D point in z-buffer

The depth or z-value of a new pixel to be written to a frame buffer is compared to the depth or z-value of the pixel already stored in the z-buffer. If the comparison indicates that the z-value of the new pixel is greater than z-value already stored in the z-buffer, then z-buffer is updated with the new z-value and intensity of new pixel is written in frame buffer, otherwise no action is taken.

Figure 10.4 depicts the triangle XYZ covering some portion of rectangle ABCD.

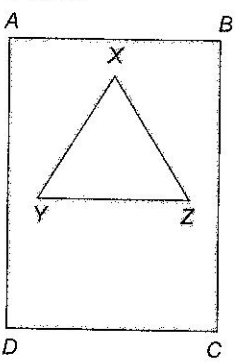


FIGURE 10.4

0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	1	1	1	1	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0

Frame buffer of Rectangle

(a)

0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	6	6	6	6	6	6	6	6	6	6	6	0
0	6	6	6	6	6	6	6	6	6	6	6	0
0	6	6	6	6	6	6	6	6	6	6	6	0
0	6	6	6	6	6	6	6	6	6	6	6	0
0	6	6	6	6	6	6	6	6	6	6	6	0
0	6	6	6	6	6	6	6	6	6	6	6	0
0	6	6	6	6	6	6	6	6	6	6	6	0
0	6	6	6	6	6	6	6	6	6	6	6	0
0	6	6	6	6	6	6	6	6	6	6	6	0
0	6	6	6	6	6	6	6	6	6	6	6	0
0	6	6	6	6	6	6	6	6	6	6	6	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0

Z-buffer of Rectangle

(b)

0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	2	0	0	0	0	0	0
0	0	0	0	0	2	2	2	0	0	0	0	0
0	0	0	0	2	2	2	2	2	0	0	0	0
0	0	0	2	2	2	2	2	2	2	0	0	0
0	0	2	2	2	2	2	2	2	2	2	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0

Frame buffer of Triangle

(c)

0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	8	0	0	0	0	0	0
0	0	0	0	0	8	8	8	0	0	0	0	0
0	0	0	0	8	8	8	8	8	0	0	0	0
0	0	0	8	8	8	8	8	8	8	0	0	0
0	0	8	8	8	8	8	8	8	8	8	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0

Z-buffer of Triangle

(d)

0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	1	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	2	1	1	1	1	1	0
0	1	1	1	1	2	2	2	1	1	1	1	0
0	1	1	1	2	2	2	2	2	1	1	1	0
0	1	1	2	2	2	2	2	2	2	1	1	0
0	1	2	2	2	2	2	2	2	2	2	1	0
0	1	1	1	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	1	1	1	1	1	0
0	1	1	1	1	1	1	1	1	1	1	1	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0

Frame buffer of Fig. 10.4
(e)

0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	6	6	6	6	6	6	6	6	6	6	6	0
0	6	6	6	6	6	8	6	6	6	6	6	0
0	6	6	6	6	8	8	8	6	6	6	6	0
0	6	6	6	8	8	8	8	8	6	6	6	0
0	6	6	8	8	8	8	8	8	8	6	6	0
0	6	8	8	8	8	8	8	8	8	8	6	0
0	6	6	6	6	6	6	6	6	6	6	6	0
0	6	6	6	6	6	6	6	6	6	6	6	0
0	6	6	6	6	6	6	6	6	6	6	6	0
0	6	6	6	6	6	6	6	6	6	6	6	0
0	6	6	6	6	6	6	6	6	6	6	6	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0

Z-buffer of Fig.10.4
(f)

FIGURE 10.5 Demonstration of Z-Buffer Method

Algorithm: Z-Buffer Algorithm for Hidden lines/Hidden Surface Removal

Step 1: Set the frame buffer to the background intensity color.

$$\text{frame\_buffer}(x,y) = I_{\text{background}}$$

Step 2: Set the z buffer to the minimum value. $Z_{\text{buffer}}(x,y)=0$

Step 3: For each pixel (x,y) in the polygon, calculate the depth $z(x,y)$ at that pixel.

Step 4: Compare the depth $z(x,y)$ with the value stored in the z-buffer at that location, $Z_{\text{buffer}}(x,y)$ to determine the visibility.

If $z(x,y) > Z_{\text{buffer}}(x,y)$, then write the polygon attributes (intensity, color, etc.) to the frame buffer at pixel location(x,y) and replace $Z_{\text{buffer}}(x,y)$ with $z(x,y)$ otherwise no action is taken i.e.,

$$Z_{\text{buffer}}(x,y) = z(x,y)$$

$$\text{frame\_buffer}(x,y) = I_{\text{proj}}(x,y)$$

where, $I_{\text{proj}}(x,y)$ is the projected intensity value for the surface at pixel position (x,y).

Step 5: Repeat steps 2 to 4 for all polygons in arbitrary order.

Step 6: Finally when all the polygons have been processed, the depth buffer contains depth value for the visible surfaces and the frame buffer contains the corresponding intensity values for those surfaces.

10.3 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.

Let $\mathbf{N} = (A, B, C)$ be the normal vector of a planar polygonal face, with $\mathbf{N}$ 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 $\mathbf{N}$ 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.

10.4 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. For example, when two polygons overlap, how do we decide which one obscures the other? (See Fig. 10.4.)

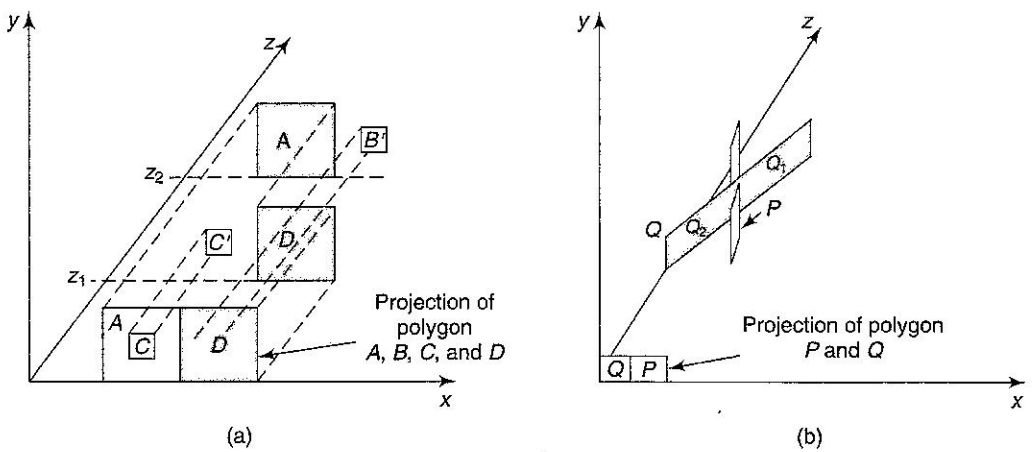


Fig. 10.4 *Projection of Opaque Polygons*

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.

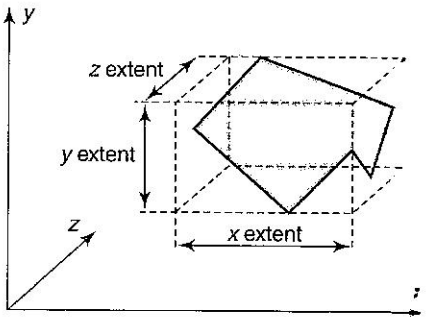


Fig. 10.5

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.

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.

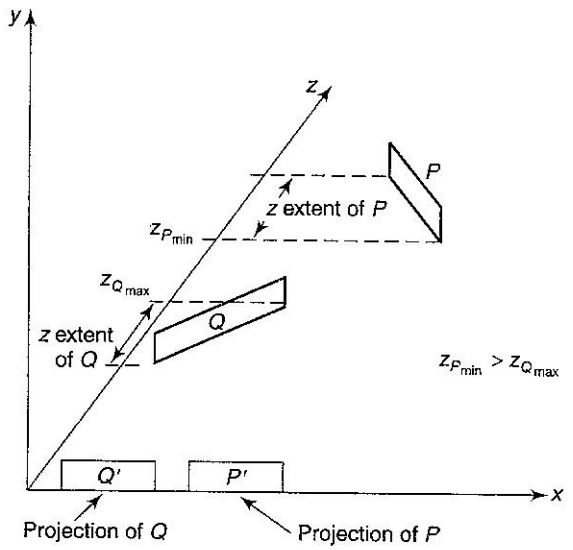


Fig. 10.6

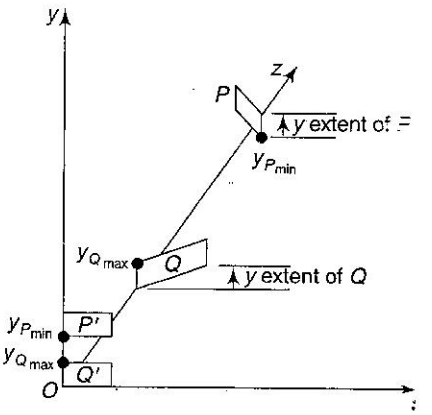


Fig. 10.7

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.

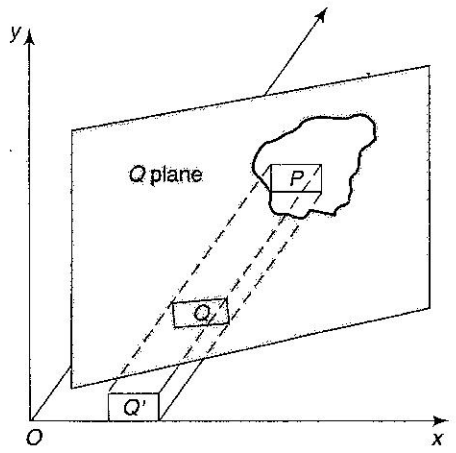


Fig. 10.8

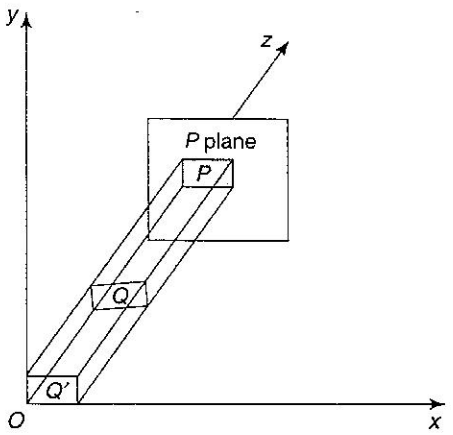


Fig. 10.9

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.

The 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 Chapter 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.

10.5 SCAN-LINE ALGORITHM

A scan-line algorithm consists essentially of two nested loops, an x-scan loop nested within a y-scan loop.

y Scan

For each y value, say, $y = \alpha$, intersect the polygons to be rendered with the scan plane $y = \alpha$. The scan plane is parallel to the xz plane, and the resulting intersections are line segments in this plane (see Fig. 10.10).

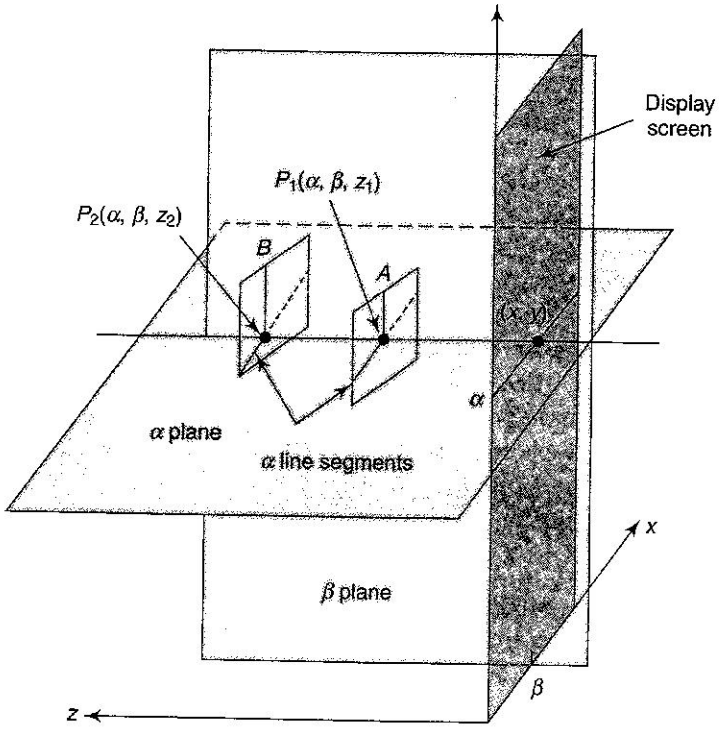


Fig. 10.10

x Scan

1. For each x value, say, $x = \beta$, intersect the line segments found above with the x-scan line $x = \beta$ lying on the y-scan plane. This intersection results in a set of points that lies on the x-scan line.
2. Sort these points with respect to their z coordinates. The point (x, y, z) with the smaller z value is visible, and the color of the polygon containing this point is the color set at the pixel corresponding to this point.

In order to reduce the amount of calculation in each scan-line loop, we try to take advantage of relationships and dependencies, called *coherences*, between different elements that comprise a scene.

Types of Coherence

1. *Scan-line coherence.* If a pixel on a scan line lies within a polygon, pixels near it will most likely lie within the polygon.
2. *Edge coherence.* If an edge of a polygon intersects a given scan line, it will most likely intersect scan lines near the given one.
3. *Area coherence.* A small area of an image will most likely lie within a single polygon.
4. *Spatial coherence.* Certain properties of an object can be determined by examining the *extent* of the object, that is, a geometric figure which circumscribes the given object. Usually the extent is a rectangle or rectangular solid (also called a *bounding box*).

Scan-line coherence and edge coherence are both used to advantage in scan-converting polygons (Chapter 3).

Spatial coherence is often used as a pre-processing step. For example, when determining whether polygons intersect, we can eliminate those polygons that don't intersect by finding the rectangular extent of each polygon and checking whether the extents intersect—a much simpler problem (see Fig. 10.11). [*Note*: In Fig. 10.11 objects A and B do not intersect; however, objects A and C, and B and C, do intersect. In pre-processing, corner points would be compared to determine whether there is an intersection. For example, the edge of object A is at coordinate $P_3 = (6, 4)$ and the edge of object B is at coordinate $P_4 = (7, 3)$.] Of course, even if the extents intersect, this does not guarantee that the polygons intersect. See Fig. 10.12 and note that the extents of A' and B' overlap even though the polygons do not.

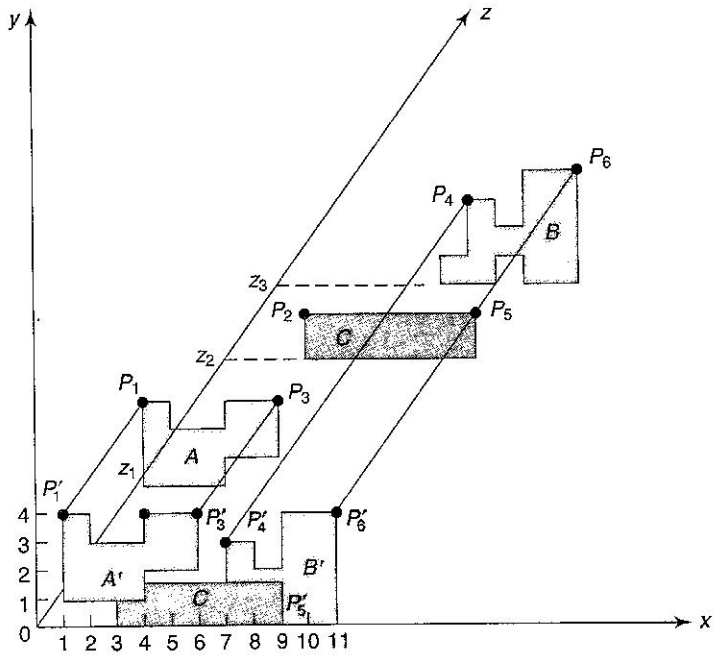
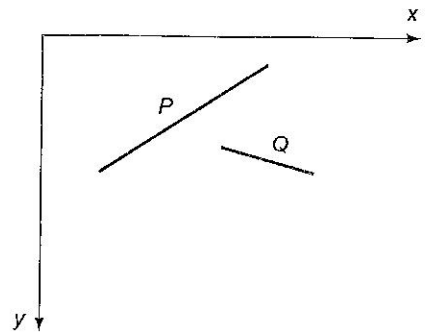


Fig. 10.11



Test 5: The projection of the polygons P and onto the xy screen does not overlap. This is checked by comparing each edge of one polygon against each edge of the other polygon to search for interactions.

Algorithm

Step 1: Sort all polygons into a polygon list according to Z_{max}. Starting from the end of the list, assign priorities for each polygon P.

Step 2: Find all polygons (preceding P) in the polygon list whose Z extents overlap that of P(test 0 is false).

Step 3: For each Q, perform tests 1 through 5.

(a) If every Q, test 1 to test 5 true, P does not obscure Q, 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. The polygon clipping techniques 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.

10.4 AREA-SUBDIVISION ALGORITHM

The area-subdivision algorithm was developed by Warnock. This algorithm follows the "***divide-and-conquer***" strategy.

This algorithm based on two steps:

[1] First decide which polygons are visible in area, they are displayed.

[2] Otherwise the area is divided into four equal areas, on each area those polygons are further tested to determine which ones should therfore be displayed. If a visibility descision cannot be made, this second area, is further subdivided either until a visibility decision can be made or until the screen area is a single pixel.

This method is also termed **Quadtree Method.**

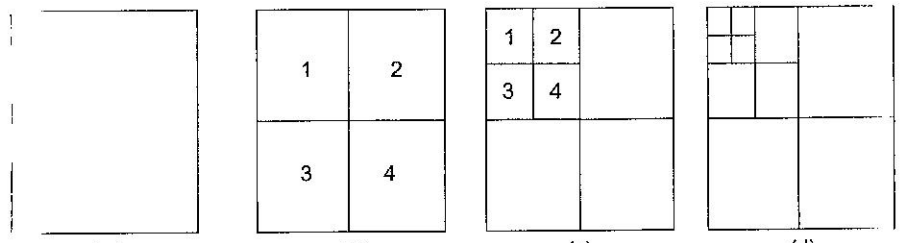


FIGURE 10.7

To check the visibility of a polygon within the viewport, its x-extent and y-extent are compared with the horizontal and vertical spans of the viewports, and classified as one of the following categories:

(a) Surrounding: A polygon surrounds a viewport if it completely encloses or covers the viewport. Refer to Fig. 10.8(a).

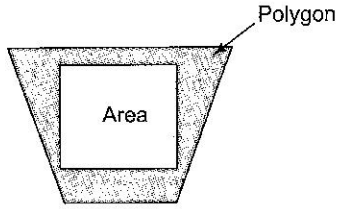


FIGURE 10.8(a) Surrounding

(b) Contained: A polygon is contained in a viewport if no part of it is outside any of the edges of the viewport. Refer to Fig. 10.8(b).

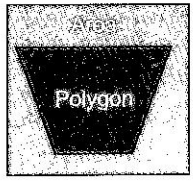
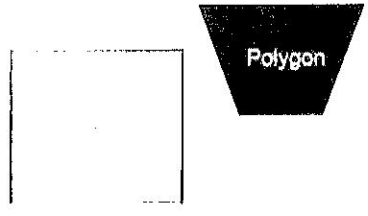


FIGURE 10.8(b) Contained

(c) Disjoint: A polygon is disjoint from the viewport if the x-extent and y-extents of the polygon do not overlap the viewport anywhere. Refer Fig. 10.7(c).



(d) Overlapping or Intersecting: A polygon overlaps or intersects the current background if any of its side cuts the edges of viewport. Refer to Fig. 10.8(d)

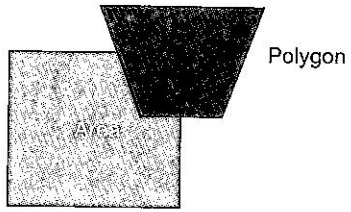


FIGURE 10.8(d) Intersecting

Disjoint polygons have no effect on area of interest. The part of intersecting polygon that is outside the area is also irrelevant, whereas the part of an intersecting polygon inside the interior to the area is the same as a contained polygon.

Algorithm

Step 1: Initialize the area to be the whole screen.
Step 2: Area does not need to be further subdivided if one of the following conditions is true:

a. All the polygons are **outside** with respect to the area.
b. There is only **one intersecting** or **contained** polygon.
c. There is **single surrounding** polygon, but no intersecting or contained polygon.
d. Surrounding surface **obscures** all other surfaces within the area boundaries

Step 3: For any remaining polygon, the viewport is divided into four quarters, and each viewport segment is evaluated for step 2. We continue this process until the subdivisions are easily analysed as belonging to single surface or until they are reduced to the size of a single pixel. A viewing area with resolution 1024 × 1024 could be required up to 10 subdivisions.
Step 4: If area is the pixel (x,y), compute $z(x,y)$ at (x,y) of all polygons. Set color of pixel with smallest z coordinate.

10.5 SCAN LINE ALGORITHM

Scan line algorithms, first developed by Wylie, Romnef, Evans and Erdahl, Birkright and Watkins.

It is an extension of the polygon scan conversion algorithm.. The difference is that here we deal with a set of polygons.

The following data structures are created:

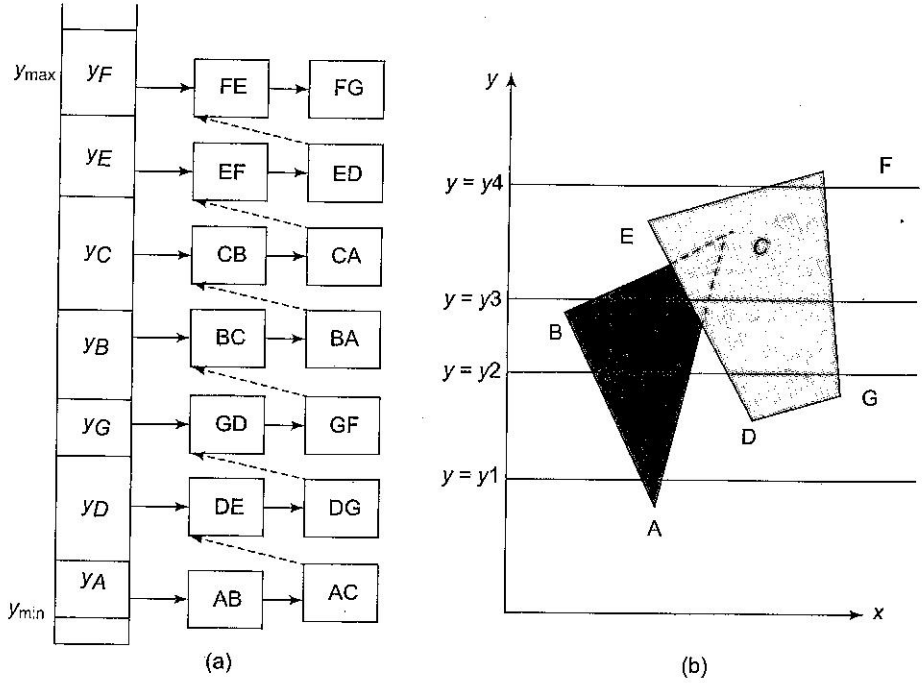
(a) Edge Table (ET)

It consists of all non-horizontal edges of all polygons projected on the view plane, horizontal edges are ignored. Each entry of the edge table contains the following informations

(i) The x coordinate of the end of the edge with the smaller y coordinate.
(ii) The y coordinate of the edge's other end (y_{max}).
(iii) The x increment, $\Delta x = \frac{1}{m}$ used in stepping in increment one scan line to next.
(iv) The polygon identification number, indicating the polygon to which the edge belongs.

ET entry	x	y_{max}	Δx	ID

Figure 10.9 depicts the rule to construct edge table (ET).



ET : AB → AC → DE → DG → GF → BC → EF → FE
(After removing the repeated edges)

FIGURE 10.9

(b) Polygon Table (PT)

It contains the following information for each polygon:

(i) The coefficient of the plane equation. This helps us to compute the z value(depth) at (x,y).

(ii) Color information for the polygon.

(iii) In-out Boolean *flag*, initialized to false and used during scan-line processing. This flag helps us to check whether a scan line in or out a polygon.

PT Entry

ID	Equation of polygon	Color	Status(IN/OUT)

(c) Active Edge Table (AET)

The AET contains only edges that cross the current scan line, sorted in order of increment x.

Scan line	*Entries*
$y = y1$	*AB,AC*
$y = y2$	*AB,AC, DE,FG*
$y = y3$	*BC,DE, AC, FG*
$Y=y4$	*EF,FG*

Algorithm

Step 1: Initialize each screen pixel to the background color.

Step 2: Activate edges whose $y = y_{min}$. Sort activates edges in order of increasing x.

Step 3: Process from left-to-right, each active edge list.

(a) *AEL* for scan line $y = y1$ contains edges entries *AB* and *AC* in that order. The edges are processed from left to right. In this case, only the flag for polygon *ABC* becomes true; thus, the Scan is "**in**" the polygon *ABC*. Therefore, no depth calculations are required. Intensity information for polygon *ABC* accessed from its *PT* as shown in Fig. 10.10.

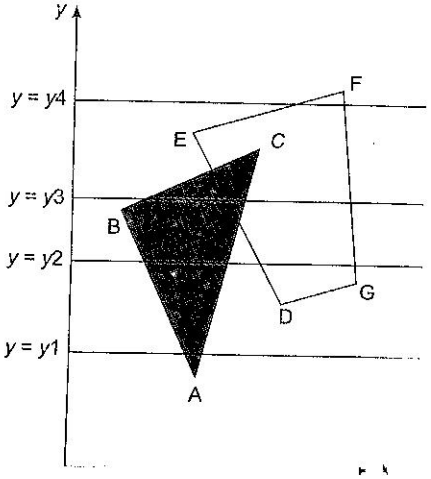


FIGURE 10.10

(b) *AEL* for scan line $y = y2$ contains edges entries *AB*, *AC*, *DE* and *FG* in that order. The edges are processed from left to right. In this case, processing proceeds much as before. There are two polygons on the scan line, but the scan is "**in**" only one polygon at a time.

In between the edges *AB* and *AC*, flag for polygon *ABC* becomes true; thus, the scan is "**in**" for the polygon *ABC*. Therefore, no depth calculations are required. Intensity information for polygon *ABC* accessed from its *PT*. Now because the scan is "**in**" only one polygon (*ABC*), it must be visible, so the shading for *ABC* is applied to the span from edge *AB* to the next edge in the *AET*, edge *AC*, at this edge flag for *ABC* is inverted to false, so that scan is no longer "**in**", as shown in Fig. 10.10(a).

Similarly, between the edges *DE* and *FG*, flag for only polygon *DEFG* becomes true; thus, the Scan is "**in**" for the polygon *DEFG*. Therefore, no depth calculations are required. Intensity information for polygon *DEFG* accessed from its *PT* as shown in Fig. 10.11(b).

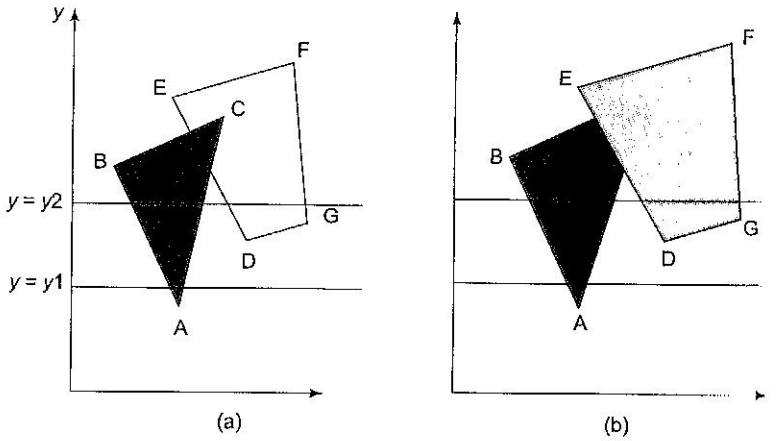


FIGURE 10.11

(c) *AEL* for scan line $y = y3$ contains edges entries *BC*, *DE*, *AC*, *FG* in that order. When this scan line entering polygon *ABC*, its flag becomes true. *ABC*'s shade is used for span up to the next edge, *DE*. At this point, the flag for *DEFG* also becomes true, so the scan line "**in**" two polygons simultaneously. Thus, we have seen that between the edges *ED* and *AC* the flags for both polygons are true. In this interval depth calculations must be made using the plane equations. On the basis of depths, we will decide whether *ABC* or *DEFG* is closer to the viewer. In this example we assume *DEFG* is closer to viewer. Therefore, the shading for *DEFG* is used for the span to edge *CB*, at which point *ABC*'s flag becomes false and the scan is again "**in**" only one polygon *DEFG* whose shade continues to be used up to edge *FG* as shown in Fig. 10.12.

Chapter Nine

Curve and Surface Design

One of the major concepts in computer graphics is *modeling* of objects. By this we mean numerical description of the objects in terms of their geometric property (size, shape) and how they interact with light (reflect, transmit). The focus of this chapter is on geometric representation of objects. We will discuss illumination and shading models in subsequent chapters.

A graphics system typically uses a set of primitives or geometric forms that are simple enough to be efficiently implemented on the computer but flexible enough to be easily manipulated (assembled, deformed) to represent or model a variety of objects. Geometric forms that are often used as primitives include, in order of complexity: points, lines, polylines, polygons, and polyhedra. More complex geometric forms include curves, curved surface patches, and quadric surfaces.

9.1 SIMPLE GEOMETRIC FORMS

Points and Lines

Points and lines are the basic building blocks of computer graphics. We specify a point by giving its coordinates in three- (or two-) dimensional space. A *line* or *line segment* is specified by giving its endpoints $P_1(x_1, y_1, z_1)$ and $P_2(x_2, y_2, z_2)$.

Polylines

A *polyline* is a chain of connected line segments. It is specified by giving the vertices (nodes) $P_0,...,P_N$ defining the line segments. The first vertex is called the *initial* or *starting point* and the last vertex, the *final* or *terminal point* (see Fig. 9.1).

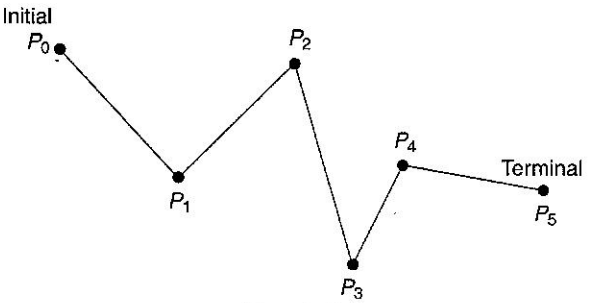


Fig. 9.1

Polygons

A *polygon* is a closed polyline, that is, one in which the initial and terminal points coincide. A polygon is specified by its *vertex list* $P_0,..., P_N, P_0$. The line segments $\overline{P_0P_1}, \overline{P_1P_2}, ..., \overline{P_NP_0}$ are called the *edges* of the polygon. (In general, we need not specify P_0 twice, especially when passing the polygon to the Sutherland-Hodgman clipping algorithm.)

A planar polygon is a polygon in which all vertices (and thus the entire polygon) lie on the same plane (see Fig. 9.2).

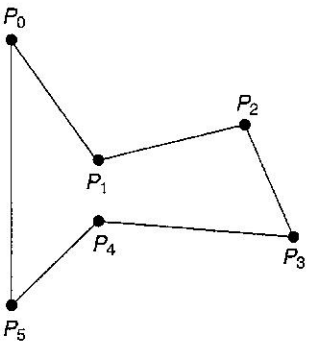
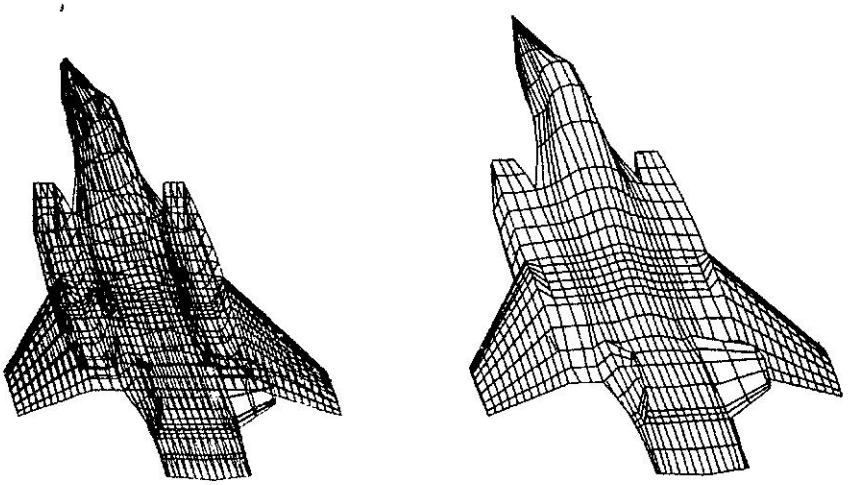


Fig. 9.2

9.2 WIREFRAME MODELS

A *wireframe model* consists of edges, vertices, and polygons. Here vertices are connected by edges, and polygons are sequences of vertices or edges. The edges may be curved or straight line segments. In the latter case, the wireframe model is called a *polygonal net* or *polygonal mesh* (Fig. 9.3).



Wire frame model
(a)

Hidden lines removed
(b)

Fig. 9.3

Splines can be either closed or open. They are the basic 2D shapes which will later be used to make 3D objects.

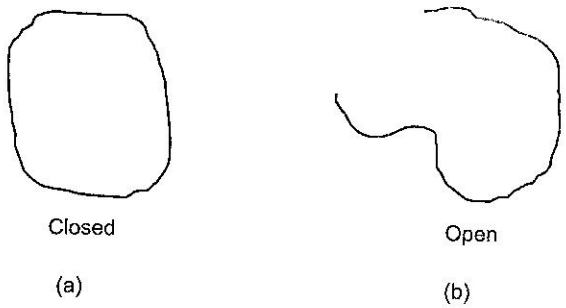


FIGURE 11.1 Spline Curves

Cubic splines are popular because they are simpler to compute, provide continuity of the curve, its slope, if necessary its curvature at appoint. Any curve may be built of a series of cubic segments.

The general equation of 2-D cubic spline is:

$$y = a_0 + a_1 t + a_2 t^2 + a_3 t^3 \qquad (11.1)$$

where, a_0, a_1, a_2, and a_3 are constants to be computed, thus we require four boundary conditions to evaluate four unknowns.

Parametric form of 2-D cubic spline is:

$$x = a_0 + a_1 u + a_2 u^2 + a_3 u^3$$
$$y = b_0 + b_1 u + b_2 u^2 + b_3 u^3$$

where, the parameter u lies between minimum and maximum values to cover the desired range of the curve.

Parametric form of 3D spline is

$$x = f(t) \quad , \quad y = g(t) \quad , \quad z = h(t)$$

where, (x,y,z) is a point on the curve corresponding to a unique value of $t(0 \le t \le 1)$.

In general, the function

$x = f(t)$ is polynomial function of degree n in t and is given by:

$$f(t) = a_0 + a_1 t + a_2 t^2 + a_3 t^3 + a_4 t^4 \ldots a_{n-1} t^{n-1} + a_n t^n$$

Similarly for $y = g(t)$ and $z = h(t)$.

The nature and shape of curve varies with value of n. We get straight line when $n = 1$; quadratic plane curve when $n = 2$; cubic space curve when $n = 3$; a quadratic space curve when $n = 4$, and so on.

Parametric form of 3D cubic spline is

$$x = a_0 + a_1 u + a_2 u^2 + a_3 u^3$$
$$y = b_0 + b_1 u + b_2 u^2 + b_3 u^3$$
$$z = c_0 + c_1 u + c_2 u^2 + c_3 u^3$$

where, the parameter u lies between minimum and maximum values to cover the desired range of the curve.

Depending on the type of polynomials used and set of boundary conditions satisfied, spline curves classified into following categories:

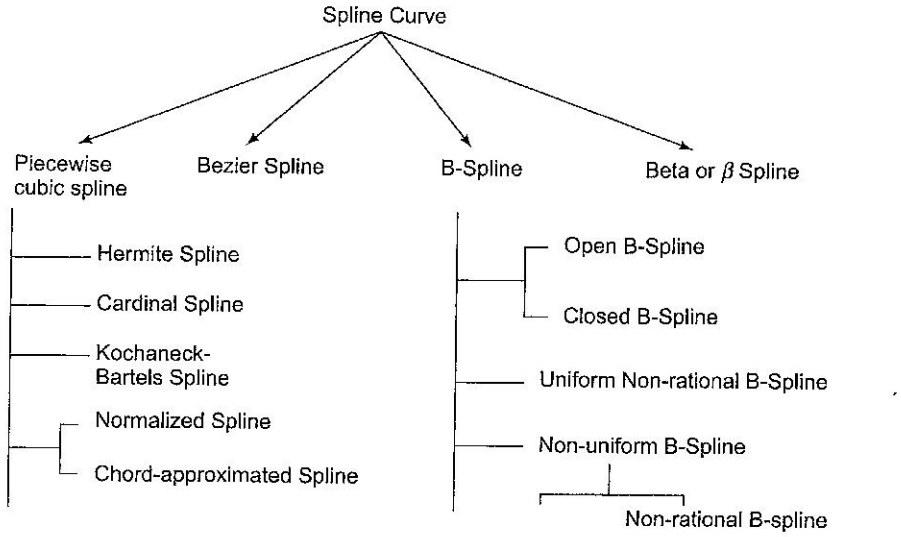


FIGURE 11.2 Classifications of Spline Curves

Above-mentioned spline curves can be put into following two basic categories depending on how they fit the given set of data points (control points):

11.2.4 Interpolated Spline

When curve passes through ***each*** control point, the resulting curve is said to interpolated spline.

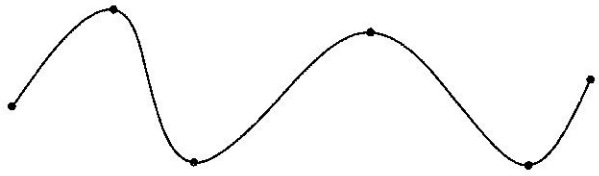


FIGURE 11.3 Interpolated Curve

11.2.5 Approximated Spline

When curve need not necessarily passing through each control point, it approximates the shape of the control polygon, the resulting curve is said to approximated spline

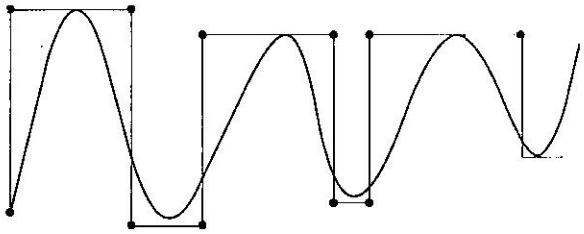


FIGURE 11.4 Approximate Curve

11.3 BEZIER CURVE AND BEZIER SURFACE

In 1970s, French Engineer, Pierre Bezier developed a method of fitting of polynomial curve. Bezier curves useful in innovative geometric deign of cars, planes architectural shape, etc.

In chapter 2, we have specified a line by the position of its end points. The Bresenham or anti-aliased algorithm simply draws a straight line between the two points. A Bezier curve allows you to specify, not only the end points of the line but also the direction of the line as it passes through the end points. The algorithm draws a curve that passes through the end points at an angle parallel to the specified ***direction.***

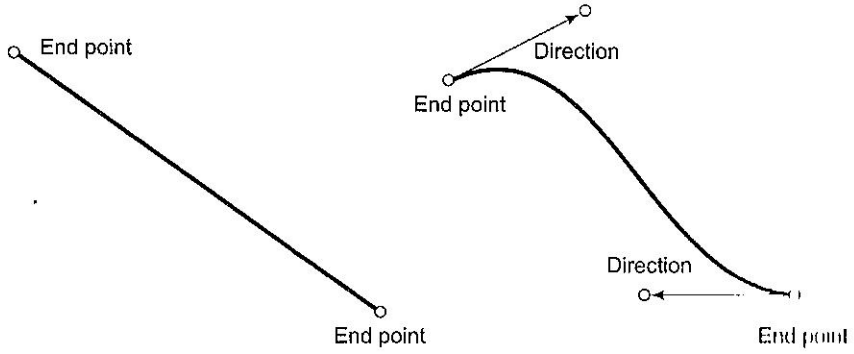


FIGURE 11.5

All sorts of curves can be specified with different direction vectors at the end points (Refer to Fig. 11.6(a)). Reflex curves appear when you set the vectors in opposite directions(Refer to Fig. 11.6(b)). Even loops can also be made(Refer to Fig. 11.6(c)).

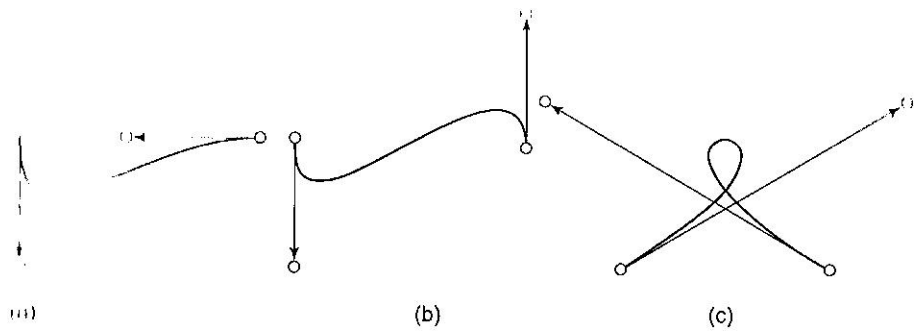


FIGURE 11.6 Different Types of Curves Based on Direction of Tangent

11.3.1 Bezier Curve

Definition

Given a set of $(n+1)$ control points P_0, P_1,P_2, ...P_n, a parametric Bezier curve (Bernstein-Bezier curve) segment is a weighted sum of $n + 1$ control points P_0, $P_1P_1P_2$, ...P_n that will fit to those points is mathematically defined by:

$$P(t) = \sum_{i=0}^{n} P_i B_{i,n}(t)$$

where, $B_{i,n}(t)$ are the Bezier blending function, also known as Bernstein Basis functions (weights) defined as follows:

$$B_{i,n}(t) = C\{n,_i)t^i (1-t)^{n-i}$$

$$C(n,i) = \frac{n!}{i!(n-i)!}$$

Thus, a Bezier curve can be seen as a weighted average of all of its control points. Because all of the weights are positive, and because the weights sum to one, the Bezier curve is guaranteed to lie within the convex hull of its control points.

The Bezier curve of order $n + 1$ (degree n) has $n + 1$ control points. Following are the first three orders of Bezier curve definitions:

Linear $P(t) = (1 - t)PO + tP1$

Quadratic $P(t) = (1-t)^2\ PO + 2(\ 1 - 1)\ tP_1 + t^2P_2$

Cubic $P(t) = (1-t)^3\ P_0 + 3\ (1-t)^2\ tP_1 + 3\ (1-t)\ t^2P^2 + t^3P_3$

$B(u)$ is a continuous function in 3 space defining the curve with N discrete control points P_k u 0 at the first control point ($k = 0$) and $u = 1$ at the last control point (k N)

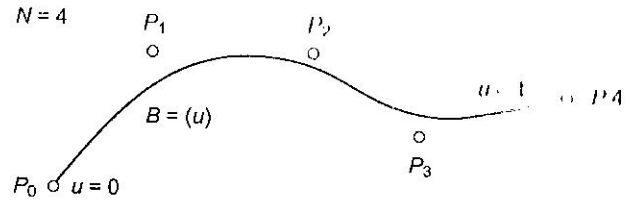


FIGURE 11.7 Bezier Curve

Properties of Bezier Curve

(1) Degree of polynomial blending function and thus Bezier curve is one less than number of control points.

(2) Bezier curve always passes through first control point P_0 and last control point P_n irrespective of intermediate points (P_1,P_2, ...P_{n-i}).

(3) Beizer blending functions are all positive and their sum is always equal to 1 i.e.,

$$\sum_{i=0}^{n} B_{i,n}(t) = 1 \text{ and for } n = 3 \text{ we have } \sum_{i=0}^{3} B_{i,3}(t) = 1$$

(4) Bezier curve always lies with convex hull of the control points.

(5) First derivatives of the Bezier curve at the end points can be calculated from control points coordinates as

$$P'(0) = -np_0 + np_1$$

$$P'(1) = -np_{n-1} + np_n$$

(6) Bezier curve can be translated and rotated by performing these operations on the control points.

Undesirable Properties of Bezier Curves are

(1) Numerical instability for large numbers of control points.

(2) Moving a single control point changes the global shape of the curve.

Points to Remember

- The curve, in general, does not pass through any of the control points except the first and last. From the formula $B(0) = P_0$ and $B(1) = P_N$.
- The curve is always contained within the convex hull of the control points, it never oscillates wildly away from the control points.
- If there is only one control point P_0, *i.e.*, $N = 0$ then $\boldsymbol{B}(u) = P_0$ for all u.
- If there are only two control points P_0 and P_1, *i.e.*, $N = 1$ then the formula reduces to a line segment between the two control points.

$$B(u) = \sum_{k=0}^{1} pk \frac{1}{k!(1-k)!} u^k (1-u)^{1-k} \qquad p(0) + u(p1 - p0)$$

the term

$$\frac{N!}{K!(N-K)!} u^K (1-u)^{N-K}$$

is called a blending function since it blends the control points to form the Bezier curve.

- The blending function is always a polynomial one degree less than the number of control points. Thus, 3 control points result in a parabola, 4 control points a cubic curve, etc.
- Closed curves can be generated by making the last control point the same as the first control point. First order continuity can be achieved by ensuring the tangent between the first two points and the last two points are the same.
- Adding multiple control points at a single position in space will add more weight to that point "pulling" the Bezier curve towards it.

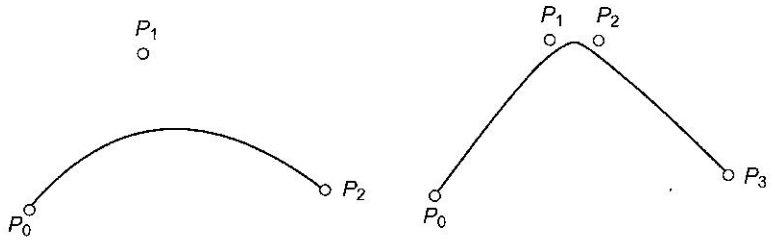


FIGURE 11.8 Effect on Shape of Curve to Add More Weight

- As the number of control points increases it is necessary to have higher order polynomials and possibly higher factorials. It is common therefore to piece together small sections of Bezier curves to form a longer curve. This also helps control local conditions; normally changing the position of one control point will affect the whole curve. Of course, since the curve starts and ends at the first and last control point it is easy to physically match the sections. It is also possible to match the first derivative since the tangent at the ends is along the line between the two points at the end. Second order continuity is generally not possible.

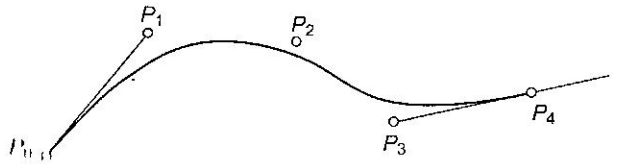


FIGURE 11.9 Role of Control Point Nearest to the End Points

11.3.3 Rational Bezier Curve

Definition

A "rational" Bezier curve is defined by

$$C(t) = \frac{\sum_{i=0}^{x} B_{i,P}(t) W_i \mathbf{P_i}}{\sum_{i=0}^{x} B_{i,P}(t) W_i},$$

where, P is the order, $B_{i,p}$ are the Bernstein polynomials, Pi are control points, and the weight W_i of Pi is the last ordinate of the homogeneous point Pi. These curves are closed under perspective transformations, and can represent conic sections exactly.

11.3.4 Bezier Surface

The Bezier surface is formed as the cartesian product of the blending functions of two orthogonal Bezier curves. The simplest way to construct a Bezier surface is as the tensor product of Bezier curves. A tensor product Bezier surface of order $n + 1$ is defined by $(n + 1)^2$ control points. It is called a Bezier patch.

$$B(u,v) = \sum_{i=0}^{Ni} \sum_{j=0}^{Nj} P_{i,j} \frac{Ni!}{i!(Ni-i)!} u^i (1-u)^{Ni-i} \frac{Nj!}{j!(Nj-j)} v^j (1-v)^{Nj-j}$$

$$0 \le u \le 1$$
$$0 \le v \le 1$$

where, $P_{i,j}$ is the i,jth control point. There are N_{i+1} and N_{j+i} control points in the i and j directions, respectively.

You can think about this as moving the control points of one Bezier curve along a set of Bezier curves to sweep out a surface. Continuity across a boundary between two Bezier patches is only guaranteed if each of the Bezier curves across the join obeys the curve continuity conditions.

The corresponding properties of the Bezier curve apply to the Bezier surface

- The surface does not, in general, pass through the control points except for the corners of the control point grid.
- The surface is contained within the convex hull of the control points.

Along the edges of the grid patch the Bezier surface matches that of a Bezier curve through the control points along that edge.

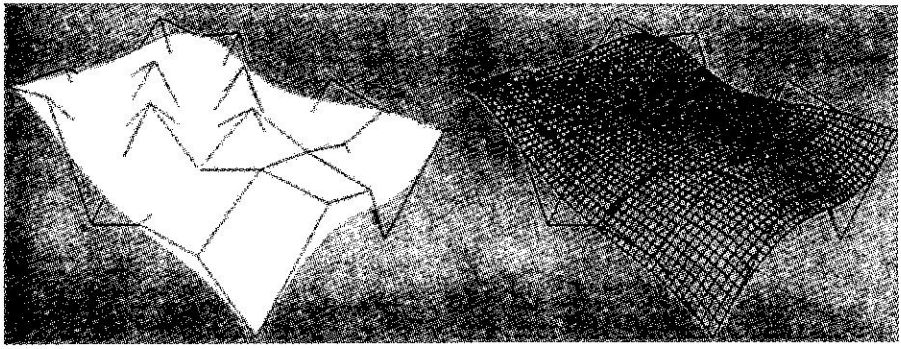


FIGURE 11.14

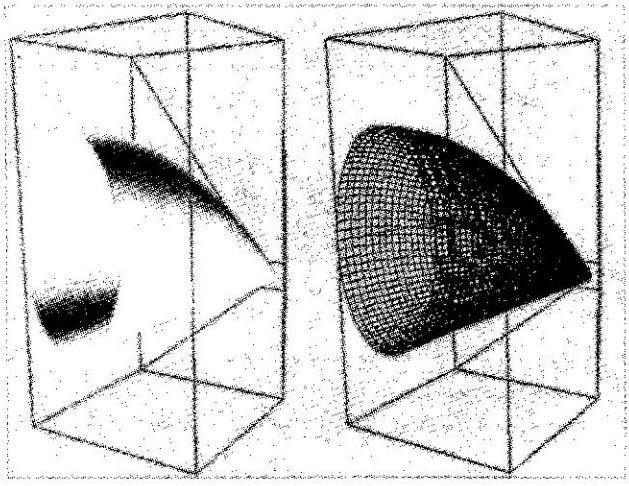


FIGURE 11.15

Closed surfaces can be formed by setting the last control point equal to the first. If the tangents also match between the first two and last two control points, then the closed surface will have first order continuity.

While a cylinder/cone can be formed from a Bezier surface, it is not possible to form a sphere.

11.4 B-SPLINE CURVES AND SURFACES

The name "B-spline" with the "*B*" standing for basis was defined by Sheonberg, in 1967.B-splines curves are widely used in computer aided design and manufacturing and are supported by OpenGL. B-splines are powerful tools for generating curves with many control points and provide advantages over Beizer curves.

Furthermore, curve designer has much flexibility in adjusting the curvature of a spline curve, and B-splines can be designed with sharp bend and even "corners".

B-spline curves are pulled towards the control points in much the same way that a Beizer curve is pulled towards its interior control points.

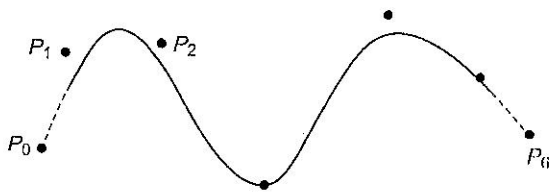


FIGURE 11.16

Unlike Bezier curve, B-spline curves do not necessarily interpolate their first and last control points(Refer to Fig. 11 .16).

A big advantage of B-spline curves over Bezier curves is that they act more flexibly and intuitively with large number of control points.

11.4.1 B-Spline Curves

Definition

Let a vector be known as the knot vector be defined $\boldsymbol{T} = \{t_0, t_1, ..., t_m\}$, where, $\boldsymbol{T}$ is a non-decreasing sequence with $t_i \in [0, 1]$, and define control points $\boldsymbol{P_0}$, ..., $\boldsymbol{P_n}$. Define the degree as

$$p \equiv m - \mathrm{n} - 1.$$

The "knots" $^{t}p^{+1}$, ..., $^{t}m - p - 1$ are called inter-knots.

Considering $P(t)$ as the parametric position vector of any point along the curve, then the general expression for a B-Spline curve of degree (d 1) is given by,

$$P(t) = \sum_{i=0}^{n} P_i B_{i,d}(t) \quad t_{\min} \le t \le t_{\max} \text{ and } 2 \le d \cdot n + 1$$

where, Pi (i = 0 to n) are the set of n + 1 control points and $Bi,d(t)$ are the n + 1 B-Spline blending functions (basis function) defined by the Cox deBoor recursion formula as,

$$B_{i,d}(t) = \left\{\frac{t - t_i}{t_{i+d-1} - t_i}\right\} B_{i,d-1}(t) + \left\{\frac{t_{i+d} - t_i}{t_{i+d} - t_{i+1}}\right\} B_{i+1,d-1}(t) \qquad (11.1)$$

where,

$$B_{i,1}(t) = 1, \text{ if } t_i \cdot t \cdot t_{i+1}$$
$$B_{i,1} \quad 0 \quad \text{otherwise}$$

When there are some repeated knots, some of the denominators above may be zero, we adopt the convention that 0/0 =0. Since, $N_{i,k}(u)$ will be identically zero when u_{i+k} u_i, this means that any term with denominator equal to zero may be removed.

Local control is an important feature of B-spline curves; it allows a designer or modified to edit one portion of a curve without causing changes to distant parts of the curve.

Properties of B-Splines Curves

(1) The polynomial curve has degree (d –1) and C^{d-2} continuity over the range of u.
(2) For (n + 1) control points, the curve is described with n +1 blending functions.
(3) Each blending function $B_{k,d}$ is defined over the d subintervals of the total range of u, starting of knot value u_k.
(4) The range of parameter u is divided into $n + d$ subintervals by the $n + d$ +1 values specified in the knot vector.
(5) With knot values labeled as $[u_0, u_1, u_2, ..., u_n]$ the resulting B-spline curve is defined only in the interval from knot value u_{d-1} up to knot value u_{n+1}.
(6) Each section of the spline curve is influenced by control points.
(7) Any one control point can affect the shape of at most d curve sections.
(8) A B-spline with no internal knot is a Bezier curve.

11.4.2 Classification of B-Spline Curves

- Uniform
- Non-uniform
- Open uniform

11.4.2.1 *Uniform, Periodic B-Splines*

A B-spline curve is called ***uniform B-Spline*** curve if the spacing between knot values is constant.

For example, we can set up a uniform knot vector as

$$[-2.0, -1.5, -1.0, -0.5, 0.0, 0.5, 1.0, 1.5]$$

Normally knot values are normalized to the range between 0 and 1 as in

$$[0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]$$

Uniform B-Splines have ***periodic blending functions***. That is, for a given value of n and d all blending functions have the same shape. In this case equation (11.1) B Spline blending functions defined by the Cox-deBoor recursion formula becomes:

$$B_{i,d}(u) = B_{i+1,d}(u + \Delta u) = B_{i+2,d}(u + 2u)$$

where, Δu is the interval between adjacent knot values.

Periodic splines are particularly used for generating close certain closed curves.

Now we will consider different cases when knots are uniformly spaced with knot vector equal to [0,1,2, ...ℓ].

Case I

Order $d = 1$ and Degree $k = 0$

In this blending function is defined as

$$B_{i,1}(u) = \begin{cases} 1 \text{ if } i \le u \le i+1 \\ 0 \text{ otherwise} \end{cases}$$

for $i = 0,1, ...\ell - 1$

Case II

Order $d = 2$ and degree $k = 1$

In this case blending function is defined as [From equation (11.3)]

$$B_{i,2}(u) = \frac{u-i}{1} B_{i,1}(u) + \frac{i+2-u}{1} B_{i+1,1}(u)$$

for $i = 0,1, \ldots \ell - 2$

In particular to the case $i = 0$, we have

$$B_{0,2}(u) = \frac{u}{1} B_{0,1}(u) + \frac{2-u}{1} B_{1,1}(u)$$

Considering separately the cases $0 \le u < 1$, $1 \le u < 2$, we have

$$B_{0,2}(u) = \begin{cases} u & \text{if } 0 \le u \le 1 \\ 2-u & \text{if } 1 \le u < 2 \\ 0 & \text{otherwise} \end{cases}$$

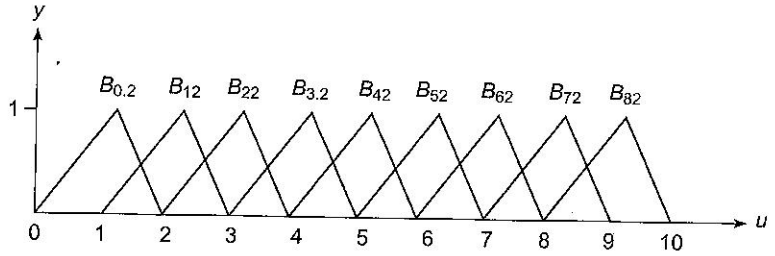


FIGURE 11.17 Periodic Blending Function for a Uniform, Degree one B-Spline

Case III

Order $d = 3$ and degree $k = 2$

In this case blending function is defined as [From equation (11.3)]:

$$B_{i,3}(u) = \frac{u-i}{2} B_{i,2}(u) + \frac{i+3-u}{2} B_{i+1,2}(u)$$

for $i = 0,1, \ldots \ell - 3$

In particular to the case $i = 0$, we have

$$B_{0,3}(u) = \frac{u}{2} B_{0,2}(u) + \frac{3-u}{2} B_{1,2}(u)$$

Considering separately the cases $0 \le u < 1$, $1 \le u \le 2$, $2 \le u < 3$, we have

$$B_{0,3}(u) = \begin{cases} \frac{1}{2}u^2 & \text{if } 0 \le u \le 1 \\ \frac{1}{2}u(2-u) + \frac{1}{2}(3-u)(u-1) = \frac{1}{2}(6u - 2u^2 - 3) & \text{if } 1 \le u < 2 \\ \frac{1}{2}(3-u)^2 & \text{if } 2 \le u < 3 \\ 0 & \text{otherwise} \end{cases}$$

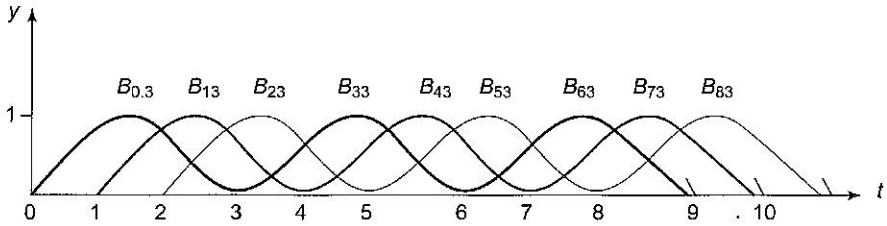


FIGURE 11.18 Periodic Blending Function for a Uniform, Degree Two B-Spline

Case IV

Cubic Periodic B-Spline Curve

As a special for a cubic periodic B-spline curve $d = 4$, then each blending function spans four subintervals of the total range of u; order $d = 4$ and degree $k = 3$.

In this case blending function is defined as [From equation (11.3)]:

$$B_{i,4}(u) = \frac{u-i}{3} B_{i,3}(u) + \frac{i+4-u}{3} B_{i+1,3}(u)$$

for $i = 0,1, \ldots \ell - 4$

If we are to fit the cubic to four control points, then we could use the integer knot vector [0,1,2,3,4,5,6,7] and $i = 0,1,2,3$.

11.4.2.2 Non-uniform B-Splines

A B-splines are called non-uniform B-splines if knot values have unequal spacing or multiple knot values.

For example,

[1,3,5,6,8,9,10]

[0|0,0,1,1,3,3|

Such types of B- plots provides increased flexibility in controlling a curve shape. With the unequally spaced intervals in the knot vectors, we can obtain

Chapter 11

Problem 11.1 Find the cubic Bezier curve with the following control points: $A(0,40)$, $B(40,40)$, $C(60,20)$, $D(60,-10)$?

Solution

Equation of cubic Bezier curve passes through A, B C, D is given by

$$B(t) = [t^3\ t^2\ t^1] \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 40 \\ 40 & 40 \\ 60 & 20 \\ 60 & -10 \end{bmatrix}$$

where $(0 \le t \le 1)$

$$B(t) = [t^3\ t^2\ t^1] \begin{bmatrix} 0 & 10 \\ -60 & -60 \\ 120 & 0 \\ 0 & 40 \end{bmatrix}$$

$$B(t) = [-60t^2 + 120t \quad 10t^3 - 60t^2 + 40]$$

Problem 11.2 Find the cubic B-spline curve with the following control points: $A(0,40)$, $B(40,40)$, $C(60,20)$, $D(60, -10)$?

Solution:

Equation of cubic B-spline curve passes through A, B, C, D is given by

$$B(t) = [t^3\ t^2\ t^1] \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} 0 & 40 \\ 40 & 40 \\ 60 & 20 \\ 60 & -10 \end{bmatrix}$$

where $(0 \le t \le 1)$

$$= [t^3\ t^2\ t^1] \frac{1}{6} \begin{bmatrix} 0 & 10 \\ -60 & -60 \\ 180 & -60 \\ 220 & 220 \end{bmatrix}$$

$$B(t) = [\frac{1}{6}(-60t^2 + 180t + 220) \quad \frac{1}{6}(10t^3 - 60t^2 - 60t + 220)]$$