Fast Bezier Curves in Windows

Michael Bertrand

C

Here's how to draw Bezier curves quickly enough to rubber-band them on-screen.

Bezier curves are widely used in computer graphics. PostScript uses these curves as building blocks, for example; defining even circles in terms of Beziers. Many modern PC and Mac illustration programs implement Bezier curves in a direct way, allowing users to create and interactively edit the curves to create complex images.

These curves possess several properties that have led to their widespread adoption in computer graphics applications:

- Because they are defined in terms of a few points, Bezier curves can be identified with these points in the graphics database.
- Efficient algorithms generate the entire curve from the defining points.
- The defining points intuitively describe the curve.

Four defining points—A Bezier curve is defined by four points called control1, handle1, handle2, and control2 (abbreviated here as ctrl1, hand1, hand2, and ctrl2). The curve begins at ctrl1 and proceeds toward hand1; at the other end, we can think of it as starting from ctrl2 and proceeding toward hand2. In each case, the curve gradually pulls away from the associated handle in its movement toward the other handle/control. The handles are points of attraction for the curve; the curve starts from ctrl1 toward hand1, but gradually pulls toward hand2 as the attractive force of hand1 diminishes and the attractive force of hand2 increases. The further away hand1 is from ctrl1, the longer the curve will be pulled toward ctrl1 before breaking toward hand2.

Bezier curves take several forms, as shown in the screen shot given in Figure 1. Vectors connecting each control point to its handle are shown as well as the Bezier curve they define. This example is about as complex as four-point Beziers get; more complex Bezier curves require a series of Beziers joined together in a continuous path.

de Casteljau Construction—Beziers must be drawn quickly to be dragged, or rubber-banded, on the screen, since the curves are being drawn and erased constantly with each mouse movement. We also need fast Bezier-drawing routines to render a complex image comprising perhaps hundreds of individual Beziers in a reasonable time.

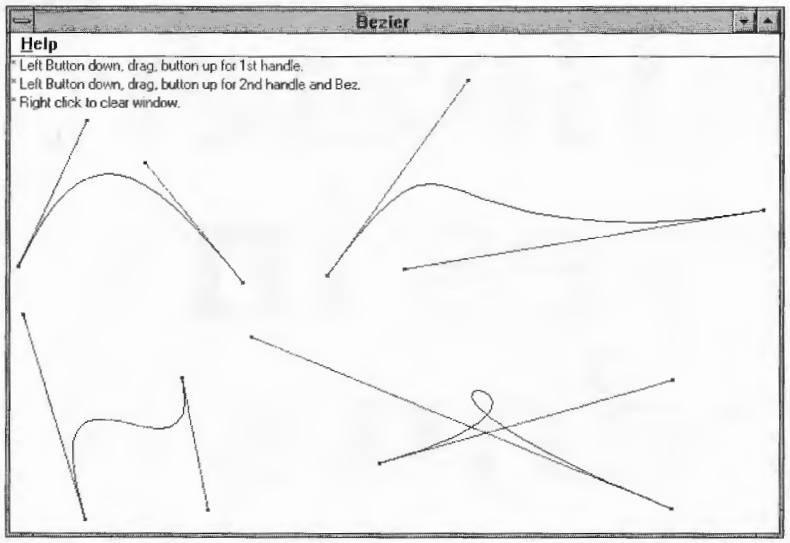
The de Casteljau algorithm is a fast integer-based method for calculating points along a Bezier curve, given the four original defining points. The calculated points can then be connected by line segments to give the impression of a smooth curve. The de Casteljau algorithm breaks a Bezier curve into two separate pieces, left and right, each of which is itself a Bezier curve.

The key to the efficiency of this algorithm is the astounding simplicity of the math, which involves taking simple averages to calculate de Casteljau construction points: First average the original defining points (the q's shown below are the averages), then average the averages (r's), then take a final average (s0):

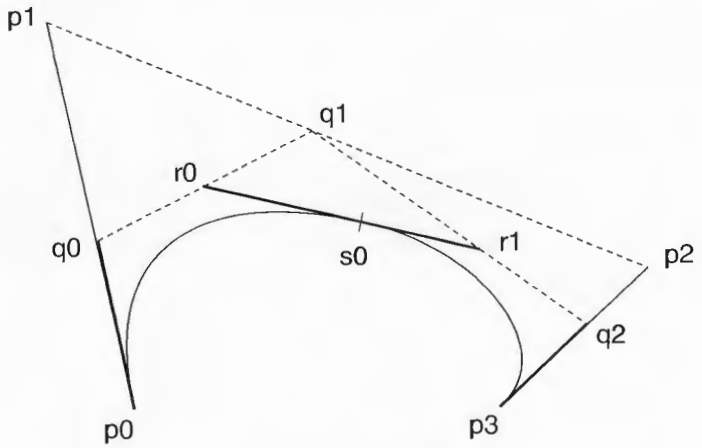
```
ctrl1 = p0
          q0
hand1 = p1        r0
             q1       s0
hand2 = p2        r1
             q2
ctrl2 = p3
```

That is: q0 = (p0 + p1)/2, r0 = (q0 + q1)/2, and so on. The actual calculations will be with coordinates, not points, but it helps to think in terms of points, keeping in mind that the "average" of two points is the point midway between the two. It turns out that s0 is the midpoint of the Bezier

▲ FIGURE 1—Sample Bezier Curves



▲ FIGURE 2—The de Casteljau Construction



curve. What's more, (p0, q0, r0, s0) are (ctrl, hand, hand, ctrl) for a Bezier coinciding with the left half of the original Bezier; and (p3, q2, r1, s0) are (ctrl, hand, hand, ctrl) for another Bezier coinciding with the right half of the original Bezier.

The procedure can be repeated: the midpoint of the left sub-Bezier is 1/4 of the way along the original curve; and the midpoint of the right sub-Bezier is 3/4 of the way along the original curve. The subdivision process can be repeated indefinitely to generate 2, 4, 8, 16,...2^n sub-Beziers and 3, 5, 9, 17,...2^n+1 points along the original curve. Like other subdivision processes, the de Casteljau algorithm lends itself to recursive implementation. Averaging entails dividing by 2, which can be done quickly as a shift right operation because we are dealing with integers.

The de Casteljau construction is illustrated in Figure 2, where the p's are the controls and handles of the original Bezier curve. q0 is midway between p0 and p1, r0 is midway between q0 and q1, and so on. Observe that :

1. s0 is the midpoint of the original Bezier curve.
2. (p0, q0, r0, s0) are (ctrl1, hand1, hand2, ctrl2) of a sub-Bezier coinciding with the left half of the original Bezier curve.
3. (p3, q2, r1, s0) are (ctrl1, hand1, hand2, ctrl2) of a sub-Bezier coinciding with the right half of the original Bezier curve.

After the first step illustrated here, we have 3 points on the curve :

```
The original control point, p0
The midpoint of the original Bezier curve
The original control point, p3
```

Applying the procedure again to the left and right sub-Beziers generates their midpoints, giving five points on the original Bezier. Subdividing these four sub-Beziers then gives us nine points on the original curve, and so on. Stopping at four subdivision levels and 17 points produces smooth curves at VGA resolution. The number of subdivisions, or recursive depth, is **BEZ_DEPTH** in the program. **NUM_BEZPTS** is the number of points generated along the Bezier curve, and is used to allocate an array to hold the Bezier points. Therefore, make sure that:

$$NUM\_BEZPTS >= 2^{BEZ\_DEPTH} + 1$$

Increasing **BEZ_DEPTH** results in more line segments in the Bezier curve, hence a smoother curve—at least up to a point. We reach diminishing returns in increasing **BEZ_DEPTH** too much, since the accumulated error of repeated averaging eventually throws the calculations off by one or more pixels. Remember that **BEZ_DEPTH** is an exponent, so increasing **BEZ_DEPTH** by 1 doubles the number of segments.

Since the recursion proceeds to the maximum depth down the far left branch, the first curve point actually generated is the point immediately following p0 = ctrl1 along the Bezier. The remaining points are also generated in order (from p0 = ctrl1 to p3 = ctrl2), a nice side effect of the recursive implementation. Another point is collected into an array every time the recursion reaches its finest subdivision level, and the points are in order!

Writing tools in Windows—We want to show off our fast Bezier-drawing through an interactive Bezier Tool. If the curve rubber-bands well on the screen, then we can claim to have a good algorithm. Interactive

tools in Windows are constructed with the concept of "system state." The **Window** procedure passes mouse messages to **BezTool()**, which maintains a key static variable, **iState**, which takes four values summarized in Table 1.

BezTool()'s action depends on **iState**, which in turn depends on the sequence of mouse messages that have recently streamed into the tool. Until the first **WM_LBUTTONDOWN** message is received, **iState** remains **NOT_STARTED** because nothing has been done. The first **WM_LBUTTONDOWN** triggers a state transition to **DRAG_HAND1**. In this state, the tool responds to **WM_MOUSEMOVE**s by rubber-banding the first handle in XOR mode. **WM_LBUTTONUP** then causes a state transition to **WAIT_FOR_CTRL2**. Nothing happens until another **WM_LBUTTONDOWN** is received, which changes **iState** to **DRAG_HAND2**; in this state, **WM_MOUSEMOVE** messages cause rubber-banding of both the second handle and the Bezier curve as a whole. **WM_LBUTTONUP** now causes a final state transition back to **NOT_STARTED**. The final handle and Bezier are frozen, and the tool is again ready to start another Bezier.

BezTool() calls **DrawHandle()** and **DrawBez()** to draw the figures (which in turn call Windows' GDI calls **MoveTo()**, **LineTo()**, and **Polyline()**). Each mouse move causes two calls to these routines. The first call draws over the figure exactly where it had been drawn the first time. Since we are in XOR drawing mode, drawing over the original figure erases it. The second call then draws at the new location. Static variables must be used if the user points are to be remembered for the next pass through the tool so previous figures can be erased.

Windows and graphics programming—Windows is a natural medium for this kind of programming. Mouse events are sent to our window procedure automatically, enabling us to build interactive mouse-driven tools. The GDI system provides line drawing, including the **R2_NOTXORPEN** ROP code which allows us to draw in XOR mode.

Windows has a built-in coordinate system and mapping modes so we can change our working range of numbers. The Beziers would not display nearly as nicely were we restricted to screen coordinates of about 500x500 pixels. By setting the **MM_ISOTROPIC** mapping mode and adjusting the Window Extent and Viewport Extent in **BezTool()**, we can expand the range to [-15,000, +15,000] which minimizes the negative side effects of calculations with small integers. ◆

Editor's note: To rebuild BEZ.EXE, you'll need several files in addition to BEZ.C, including some with no ASCII representation. These files are present in a file called BEZ.ZIP, contained within the listings archive for this issue, either on a Disk Subscription disk or from one of the online services and BBS systems that carry our listings.

References

Foley, James D., Andries van Dam, Steven K. Feiner, and John F. Hughes, *Computer Graphics : Principles and Practice*, Addison-Wesley (2nd ed., 1990), pp 507ff.

Michael Bertrand teaches Mathematics and programming at Madison Area Technical College, Madison, WI 53704.

▲ TABLE 1—IState's Four Possible Values

```
NOT_STARTED        :  tool has not been started
DRAG_HAND1         :  dragging handle1
WAIT_FOR_CTRL2     :  waiting for control2 to be entered
DRAG_HAND2         :  dragging handle2 and Bezier
```

▲ LISTING 1—BEZ.C

```
/* BEZ.C : Program to draw Bezier curves and their handles
   interactively. User draws first handle by dragging, then second
   handle; the Bezier curve rubber-bands together with the second
   handle. Demonstrates the de Casteljau algorithm for fast
   calculation of Bezier points.
   Copyright (c) 1991, Michael A. Bertrand.   */

#include <windows.h>
#include "bez.h"

HPEN   hRedPen;            /* red pen for handles. */
int    LogPerDevice;       /* #logical units per device unit
                              (both axes). */
WORD   cxClient;           /* size of client area (x). */
WORD   cyClient;           /* size of client area (y). */
HANDLE hInst;              /* current instance */
POINT  BezPts[NUM_BEZPTS]; /* array of pts along Bezier curve */
POINT  *PtrBezPts;         /* pointer into BezPts[] array */

char Instr1[] =
  "* Left Button down, drag, button up for 1st handle.";
char Instr2[] =
  "* Left Button down, drag, button up for 2nd handle and Bez.";
char Instr3[] = "* Right click to clear window.";

int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance,
          LPSTR lpszCmdLine, int nCmdShow)
/*
  USE:  Register window and set dispach message loop.
  IN:   hInstance,hPrevInstance,lpszCmdLine,nCmdShow : standard
        WinMain parms
*/
{
  static char szAppName [] = "Bezier";
  static char szIconName[] = "BezIcon";
  static char szMenuName[] = "BezMenu";

  HWND     hWnd;    /* handle to WinMain's window */
  MSG      msg;     /* message dispached to window */
  WNDCLASS wc;      /* for registering window */

  /* Save instance handle in global var
     so can use for "About" dialog box. */
  hInst = hInstance;

  /* Register application window class. */
  if (!hPrevInstance)
    {
    wc.style         = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc   = WndProc;  /* fn to get window's messages */
    wc.cbClsExtra    = 0;
    wc.cbWndExtra    = 0;
    wc.hInstance     = hInstance;
    wc.hIcon         = LoadIcon(hInstance, szIconName);
    wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName  = szMenuName;  /* menu resource in RC file */
    wc.lpszClassName = szAppName;   /* name used in call to
                                       CreateWindow() */

    if (!RegisterClass(&wc))
      return(FALSE);
    }

    /* Initialize specific instance. */
    hWnd = CreateWindow(szAppName, szAppName, WS_OVERLAPPEDWINDOW,
                        CW_USEDEFAULT, CW_USEDEFAULT, CW_USEDEFAULT,
                        CW_USEDEFAULT, NULL, NULL, hInstance, NULL);

  ShowWindow(hWnd, nCmdShow); /* display the window */
  UpdateWindow(hWnd);         /* update client area; send WM_PAINT */

  /* Read msgs from app que and dispatch them to appropriate win
     function. Continues until GetMessage() returns NULL when it
     receives WM_QUIT. */
  while (GetMessage(&msg, NULL, NULL, NULL))
    {
    TranslateMessage(&msg);  /* process char input from keyboard */
    DispatchMessage(&msg);   /* pass message to window function */
    }
  return(msg.wParam);
}

long FAR PASCAL WndProc(HWND hWnd, unsigned iMessage,
                        WORD wParam, LONG lParam)
/*
  USE: Application's window procedure : all app's messages come
       here.
  IN:  hWnd,iMessage,wParam,lParam : standard Windows proc parameters
*/
{
  HDC         hDC;  /* must generate our own handle to DC to draw */
  PAINTSTRUCT ps;         /* needed when receive WM_PAINT message */
  FARPROC     lpProcAbout;      /* pointer to "AboutBez" function */

  switch(iMessage)
    {
    case WM_CREATE:
      /* Create hRedPen once and store as global. */
      hRedPen = CreatePen(PS_SOLID, 1, RGB(255, 0, 0));
      break;  /* WM_CREATE */

    case WM_SIZE:
      /* Get client area size into globals when window resized. */
      cxClient = LOWORD(lParam);
      cyClient = HIWORD(lParam);
      break;  /* WM_SIZE */

    case WM_COMMAND:
      if (wParam == IDM_ABOUT)
        {
        /* "About" menu item chosen by user :
           call "AboutBez" function. */
        lpProcAbout = MakeProcInstance(AboutBez, hInst);
        DialogBox (hInst, "AboutBez", hWnd, lpProcAbout);
        FreeProcInstance(lpProcAbout);
        }
      break;  /* WM_COMMAND */

    case WM_PAINT:
      /* Repaint instructions at upper left of window. */
      hDC = BeginPaint(hWnd, &ps);
      SelectObject(hDC, GetStockObject(ANSI_VAR_FONT));
      TextOut(hDC, 0,  0, Instr1, lstrlen(Instr1));
      TextOut(hDC, 0, 15, Instr2, lstrlen(Instr2));
      TextOut(hDC, 0, 30, Instr3, lstrlen(Instr3));
      EndPaint(hWnd, &ps);
      break;  /* WM_PAINT */

    case WM_LBUTTONDOWN:
    case WM_RBUTTONDOWN:
    case WM_MOUSEMOVE:
    case WM_LBUTTONUP:
      /* Mouse events passed on to BezTool() for processing. */
      BezTool(hWnd, iMessage, lParam);
      break;  /* WM_LBUTTONDOWN... */

    case WM_DESTROY:
      /* Destroy window & delete pen when application terminated. */
      DeleteObject(hRedPen);
      PostQuitMessage(0);
      break;  /* WM_DESTROY */
    default:
```

```
    return(DefWindowProc(hWnd, iMessage, wParam, lParam));
  } /* switch(iMessage) */
  return(0L);
}

void NEAR PASCAL BezTool(HWND hWnd, unsigned iMessage, LONG lParam)
/*
USE:  Process mouse event to draw handles and Bezier curve.
IN:   hWnd     : handle to window
      iMessage : mouse event (WM_LBUTTONDOWN, etc.)
      lParam   : mouse coords (x == loword, y == hiword)
NOTE: This is the interactive Bezier drawing tool which processes
WM_RBUTTONDOWN, WM_LBUTTONDOWN, WM_MOUSEMOVE, and WM_LBUTTONUP
messages. BezTool() is called repeatedly as the user draws. The
current state of the tool is maintained in the key static variable
iState. iState's value, as set last time thru the tool, determines
the tool's action this time thru. Bezier control and handle points,
as input by the user, are also maintained as statics so BezTool()
remembers them the next time thru.
*/
{
  HDC    hDC;       /* must generate our own handle to DC to draw */
  WORD   maxClient; /* larger of (cxClient, cyClient) */
  POINT  inPt;      /* incoming point */
  POINT  pts[2]; /* to get LogPerDevice, #logical units/dev. unit */
  /* user-entered Bez control & handle (1st): */
  static POINT ctrl1, hand1;
  /* user-entered Bez control & handle (2nd): */
  static POINT ctrl2, hand2;
  static int iState;  /* BezTool()'s state : DRAG_HAND1, etc. */

  hDC = GetDC(hWnd);

  /* Set extents and origin so will be working
     in range [-15000, +15000]. */
  SetMapMode(hDC, MM_ISOTROPIC);
  SetWindowExt(hDC, 30000, 30000);
  maxClient = (cxClient > cyClient) ? cxClient : cyClient;
  SetViewportExt(hDC, maxClient, -maxClient);
  SetViewportOrg(hDC, cxClient >> 1, cyClient >> 1);

  /* Calculate #logical units per device unit --
     will need later when draw little 3x3 boxes in DrawHandle(). */
  pts[0].x = pts[0].y = 0;
  pts[1].x = pts[1].y = 1;
  DPtoLP(hDC, pts, 2);
  LogPerDevice = (pts[1].x > pts[0].x) ? (pts[1].x - pts[0].x) :
                                         (pts[0].x - pts[1].x);

  /* Incoming point in device coordinates. */
  inPt.x = LOWORD(lParam);
  inPt.y = HIWORD(lParam);
  /* Convert to logical coordinates. */
  DPtoLP(hDC, &inPt, 1);

  switch(iMessage)
    {
    case WM_RBUTTONDOWN:
      /* Erase client area if not in middle of Bez. */
      if (iState == NOT_STARTED)
        InvalidateRect(hWnd, NULL, TRUE);
      break;  /* WM_RBUTTONDOWN */

    case WM_LBUTTONDOWN:
      switch(iState)
        {
        case NOT_STARTED:
          iState = DRAG_HAND1;          /* starting drag */
          hand1.x = ctrl1.x = inPt.x;  /* store user point
          hand1.y = ctrl1.y = inPt.y;     in statics */
          break;  /* NOT_STARTED */
        case WAIT_FOR_CTRL2:
          iState = DRAG_HAND2;          /* starting drag */
          hand2.x = ctrl2.x = inPt.x;  /* store user point
          hand2.y = ctrl2.y = inPt.y;     in statics */
          SetROP2(hDC, R2_NOTXORPEN);  /* draw in XOR */
          DrawBez(hDC, ctrl1, hand1, hand2, ctrl2);
          break;  /* NOT_STARTED */
        } /* switch(iState) */
      break;  /* WM_LBUTTONDOWN */

    case WM_MOUSEMOVE:
      switch(iState)
        {
        case DRAG_HAND1:
          SetROP2(hDC, R2_NOTXORPEN);     /* draw in XOR */
          DrawHandle(hDC, ctrl1, hand1);  /* erase old */
          hand1.x = inPt.x;               /* get new handle */
          hand1.y = inPt.y;
          DrawHandle(hDC, ctrl1, hand1);  /* draw new */
          break;  /* DRAG_HAND1 */
        case DRAG_HAND2:
          SetROP2(hDC, R2_NOTXORPEN);     /* draw in XOR */
          DrawHandle(hDC, ctrl2, hand2);  /* erase old */
          DrawBez(hDC, ctrl1, hand1, hand2, ctrl2);
          hand2.x = inPt.x;               /* get new handle */
          hand2.y = inPt.y;
          DrawHandle(hDC, ctrl2, hand2);  /* draw new */
          DrawBez(hDC, ctrl1, hand1, hand2, ctrl2);
          break;  /* DRAG_HAND1 */
        } /* switch(iState) */
      break;  /* WM_MOUSEMOVE */

    case WM_LBUTTONUP:
      switch(iState)
        {
        case DRAG_HAND1:
          iState = WAIT_FOR_CTRL2;
          SetROP2(hDC, R2_COPYPEN); /* COPY pen for final handle */
          DrawHandle(hDC, ctrl1, hand1);    /* draw in COPY mode */
          break;  /* DRAG_HAND1 */
        case DRAG_HAND2:
          iState = NOT_STARTED;
          SetROP2(hDC, R2_COPYPEN); /* COPY pen for final handle */
          DrawHandle(hDC, ctrl2, hand2);    /* draw in COPY mode */
          DrawBez(hDC, ctrl1, hand1, hand2, ctrl2);
          break;  /* DRAG_HAND2 */
        } /* switch(iState) */
      break;  /* WM_LBUTTONUP */
    } /* switch(iMessage) */

  ReleaseDC(hWnd, hDC);
}

BOOL FAR PASCAL AboutBez(HWND hDlg, unsigned iMessage,
                         WORD wParam, LONG lParam)
/*
USE:  Application's "About" dialog box function.
IN:   hDlg     : handle to dialog box
      iMessage : message type
      wParam   : auxiliary message info (act on IDOK, IDCANCEL)
      lParam   : unused
RET:  Return TRUE if processed appropriate message, FALSE otherwise.
NOTE: Closes "About" box only when user clicks OK button
      or system close. */
{
  switch (iMessage)
    {
    case WM_INITDIALOG:       /* initialize dialog box */
      return (TRUE);
    case WM_COMMAND:          /* received a command */
/* IDOK if OK box selected; IDCANCEL if system menu close command */
      if (wParam == IDOK || wParam == IDCANCEL)
        {
        EndDialog(hDlg, TRUE);  /* exit dialog box */
        return(TRUE);           /* did proccess message */
        }
      break;  /* WM_COMMAND */
    } /* switch (iMessage) */
  return (FALSE);               /* did not process message */
}
```

```
void NEAR PASCAL DrawBez(HDC hDC, POINT ctrl1, POINT hand1,
                         POINT hand2, POINT ctrl2)
/*
USE:  Draw Bezier curve given control and handle points.
IN:   ctrl1,hand1,hand2,ctrl2 : control and handle points for Bezier
NOTE: Set up, then call SubDivideBez(), the recursive de Casteljau
routine, generate points along the Bez. Windows' Polyline() displays
the Bez as a polygon. BEZ_DEPTH = recursive depth of de Casteljau.
Initial POINT ctrl1 loaded here, then recursive routine calculates
and loads the remaining 2^BEZ_DEPTH = (NUM_BEZPTS - 1) de Casteljau
pts. */
{
  PtrBezPts = BezPts;         /* init ptr to start of array */
  *PtrBezPts++ = ctrl1;       /* first control point special case */
  /* calc pts */
  SubDivideBez(ctrl1, hand1, hand2, ctrl2, BEZ_DEPTH);
  Polyline(hDC, BezPts, NUM_BEZPTS);  /* call Windows to draw */
}

void NEAR PASCAL SubDivideBez(POINT p0, POINT p1,
                              POINT p2, POINT p3, int depth)
/*
  USE:  Calculate de Casteljau construction points and break Bez
        in two.
  IN:   p0,p1,p2,p3 : control/handle/handle/control for Bez to
        subdivide depth: current recursive depth of algorithm.
  NOTE: Calculates the de Casteljau construction points so the
  Bezier can be subdivided into 2 parts (left, then right) by
  recursive calls to this routine. Recursion is broken off when
  depth, decremented once for each recursion level, becomes 0.
  This is the finest level of subdivision; the right-most point on
  the small subdivided Bezier is also a point on the original
  Bezier, so we load it into global array BezPts[] (thru PtrBezPts
  which points into the array). */
{
  /* de Casteljau construction points: */
  POINT q0, q1, q2, r0, r1, s0;

  /* depth == 0 means we are at the finest subdivision level:
  grab point into global array and return, breaking off recursion.
  */
  if (!depth)
    {
    *PtrBezPts++ = p3;
    return;
    }

  /* Calculate de Casteljau construction points as averages of
     previous points (ie., midway points); note shift right is
     fast division by 2. */
  /* q's are midway between 4 incoming control and handle points. */
  q0.x = (p0.x + p1.x) >> 1;  q0.y = (p0.y + p1.y) >> 1;
  q1.x = (p1.x + p2.x) >> 1;  q1.y = (p1.y + p2.y) >> 1;
  q2.x = (p2.x + p3.x) >> 1;  q2.y = (p2.y + p3.y) >> 1;

  /* r's are midway between 3 q's. */
  r0.x = (q0.x + q1.x) >> 1;  r0.y = (q0.y + q1.y) >> 1;
  r1.x = (q1.x + q2.x) >> 1;  r1.y = (q1.y + q2.y) >> 1;

  /* s0 is midway between 2 r's and is in middle of original Bez. */
  s0.x = (r0.x + r1.x) >> 1;  s0.y = (r0.y + r1.y) >> 1;

  /* Decrement depth; subdivide incoming Bez into 2 parts:
     left, then right.*/
  SubDivideBez(p0, q0, r0, s0, --depth);
  SubDivideBez(s0, r1, q2, p3,   depth);
}

void NEAR PASCAL DrawHandle(HDC hDC, POINT p, POINT q)
/*
  USE:  Draws handle on screen from p to q with hRedPen.
  IN:   hDC : handle to display context
        p,q : handle start and end points
  NOTE: Don't CreatePen or delete--these are done globally once
        only.
  Handles are drawn with little 3x3 pixel boxes at each end.
  Each pixel is LogPerDevice logical units; logical units must be
  used for the boxes since we are in MM_ISOTROPIC mapping mode.
*/
{
  HPEN origPen;     /* DC's orinal pen */
  int  xLeft;       /* left coord of little box at end of handle */
  int  xRight;      /* right coord of little box */
  int  y;           /* y coord of little box */

  /* Save original pen, select red pen. */
  origPen = SelectObject(hDC, hRedPen);

  /* Draw handle. */
  MoveTo(hDC, p.x, p.y);  LineTo(hDC, q.x, q.y);

  /* Set left and right coords around q.x (3 pixels). Remember
     Windows lines do not draw last pixel. */
  xLeft  = q.x - LogPerDevice;
  xRight = q.x + (LogPerDevice << 1);

  /* Init y coord 1 pixel below q.y. */
  y = q.y - LogPerDevice;

  /* Draw little box : 3x3 pixels. */
  MoveTo(hDC, xLeft, y); LineTo(hDC, xRight, y);  y += LogPerDevice;
  MoveTo(hDC, xLeft, y); LineTo(hDC, xRight, y);  y += LogPerDevice;
  MoveTo(hDC, xLeft, y); LineTo(hDC, xRight, y);  y += LogPerDevice;

  /* Re-select original pen. */
  SelectObject(hDC, origPen);
}
```