Programming Windows Using State Tables

This data structure can simplify the structure of interactive graphics programs

Michael A. Bertrand & William R. Welch

This article presents a Windows-based program called "Draw" that uses state tables to implement interactive drawing tools in an economical, consistent fashion. Draw renders four kinds of geometric figures: rectangles, rounded rectangles, ellipses, and lines. Each type is associated with a drawing tool that's accessed by means of a menu choice (see Figure 1). Our implementation uses state tables to encapsulate program control flow in a single data structure (an array of pointers to functions). Using this technique, you can easily extend the program to support other kinds of geometric figures, as long as the user interaction for the new types is similar to the types described here.

Before discussing the details of our implementation, it is useful to review some of the basic concepts behind Windows programs.

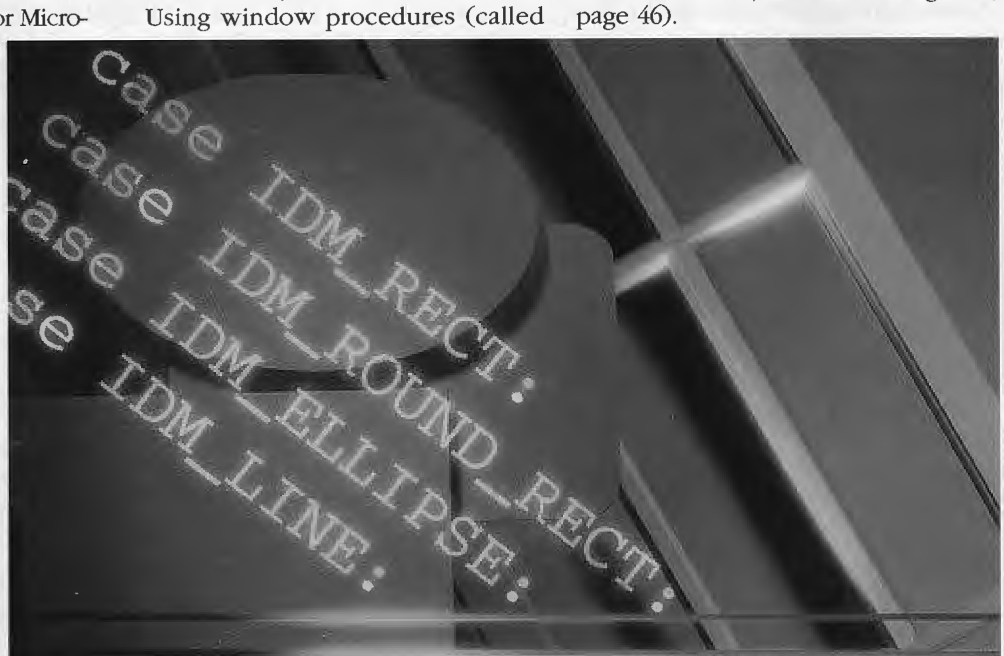
Event-Driven Programming

As more and more programmers are finding out, writing programs for Microsoft Windows and other event-driven GUIs is very different from writing traditional DOS programs. In a Windows program, your program does not have a single line of control, flowing from beginning to middle to end. Rather, it responds to all manner of events (or, in Windows parlance, messages) that are sent by the system to all applications, at arbitrary or unpredictable times. This event-driven structure follows the pattern of interaction of a real-world user driving an interactive graphics application: Any one event, such as a mouse movement, is about as likely to occur as any other (say, a keystroke or a menu choice).

Using window procedures (called *Wndprocs*), your application is able to respond to all of these events or messages as they occur. This is not merely a suggestion, but an implementation requirement. Each type of window in a Microsoft Windows application must have a procedure associated with it that receives all messages sent by environment to that class of window. The messages correspond to external events (mouse movements, mouse clicks, keystrokes) as well as internal events (for example, the message that asks the application to redraw its screen display, or a message sent by another application, and so on).

With this bit of background, we can now discuss the Draw program. Draw consists of a single header file, Draw.h (Listing One, page 45), and a single C-language source file, Draw.c (Listing Two, page 45). There is also a makefile (Listing Three, page 46) and two files required by Windows: the definition file, Draw.def (Listing Four, page 46), and the resource file, Draw.rc (Listing Five, page 46).

Mike teaches mathematics and programming at Madison Area Technical College, Madison, WI 53704. Bill is a freelance writer and programmer who holds a Ph.D. in biological science. He can be reached at 201 Virginia Terrace, Madison, WI 53705.



The Main Window Procedure

In general, every application has a main window and an associated main window procedure. If the application has other kinds of windows (known as child windows), each of these kinds will have a window procedure defined for it as well. Draw creates only one kind of window, so it has only a single window procedure, *WndProc*.

The function *WndProc* contains code to respond to Windows messages such as selecting a drawing tool from the menu, responding to mouse events, and repainting the window when it is moved or resized. *WndProc* passes mouse-button and mouse-move events to the function *Tool*, which manages drawing. *Tool* provides a template for interactive drawing tools and is the real heart of Draw.

When using Draw, you interactively display geometric figures by invoking three mouse events: left-button-down, mouse-move, and left-button-up. These three events produce the Windows messages WM_LBUTTONDOWN, WM_MOUSEMOVE, and WM_LBUTTONUP, respectively. As is common in Windows programs, *Tool* uses these messages as case constants in a switch statement. With the rectangle tool, for example, you first depress the left mouse button (WM_LBUTTONDOWN) to define the x and y coordinates (x1 and y1) of the initial corner of the figure. Then, as you move the mouse without releasing the button (dragging the cursor and producing a series of WM_MOUSEMOVEs), the program repeatedly erases and redraws the rectangle while the current mouse position defines the x and y coordinates (x2 and y2) of the rectangle corner opposite the initial corner. The final figure appears when you release the left mouse button (WM_LBUTTONUP).

A Simplifying Technique

Draw's four tools require a minimal amount of code. The key to this economy is the data structure *DrawFig*, which is an array of pointers to functions — one for each tool. All four tools work in exactly the same way (that is, left-button-down, mouse-move, left-button-up), and their functions have the same parameters and return a value of the same type. In choosing a tool through the menu, the program sets the value of the *DrawFig* index, *iFigType*. This value, in turn, determines which function is pointed to by the *DrawFig* array and used for the actual drawing in *Tool*.

Draw uses the standard Windows raster operation codes to "rubber band" a figure as the mouse cursor is dragged

Two of the functions that the *DrawFig* array points to, *Rectangle* and *Ellipse*, are standard Windows functions, that is, part of the native Application Program Interface (API). The other two functions that the *DrawFig* array points to, *DrawRoundRect* and *DrawLine*, are our own. This is because the native Windows functions to draw rounded rectangles (*RoundRect*) and lines (*MoveTo* and *LineTo*) have different parameters than *Rectangle* and *Ellipse*. To deal with this difference, we wrote the *DrawRoundRect* and *DrawLine* functions. These two have the same parameters as *Rectangle* and *Ellipse*, so all four functions can be included in the same array of pointers to functions, the *DrawFig* array.

This scheme of using the *DrawFig* array to point to tool functions that use the same three mouse events to draw figures has an important ramification: Other similarly behaving tools can be added to Draw by simply including pointers to the appropriate functions in the list of the *DrawFig* array initializers and including them in the menu. Additions might, for example, be tools for isosceles triangles, regular polygons, and parabolic segments.

Storing Figure Coordinates

In any Windows application, whenever the user moves a window or changes its size, Windows sends a WM_PAINT message to the application to erase and redisplay the entire output area of the window. Any figures produced by Draw will be erased, and Draw must redraw them if they are to stay on the screen as the location or size of the window is changed.

This restoration of the window contents can be accomplished only if Draw in some way saves the figures. This it does, in the externally defined structure *faList*, which is an array of structures of type FIGURE. Each FIGURE in *faList* contains a field (named *iType*) that indicates the type of figure (rectangle, rounded rectangle, ellipse, or line) and a structure (rsCoord) that contains the x and y coordinates of the two endpoints which define the location of the figure. Values for these variables are assigned — a new figure is saved — in this case block WM_LBUTTONUP of function *Tool*. Whenever *WndProc* gets a WM_PAINT message, it traverses *faList*, a simple graphics database, to restore the screen. The array *faList* is characteristic of the graphics programming approach known as vector-based or display-list oriented approach. (This technique is also sometimes loosely called object-oriented.) In this approach, a geometric figure is represented in the database, or display list, by a set of drawing commands and endpoint coordinates that determine how the list is traversed to display the figures. In Draw, the drawing command in the list is the type of figure (*iType*); more elaborate systems include attributes such as line width and line color.

	Mouse Event		
System State	WM_LBUTTONDOWN	WM_MOUSEMOVE	WM_LBUTTONUP
WAITING	DRAWING	—	—
DRAWING	—	—	WAITING

Table 1: *State table shows the changes from one system state to another (Waiting to Drawing and back), as triggered by the mouse events (left-button-down, move, left-button-up).*

			Mouse Event		
System State	WM_LBUTTONDOWN	WM_MOUSEMOVE	WM_LBUTTONUP	WM_RBUTTONDOWN	WM_RBUTTONUP
WAITING	DRAWING	—	—	—	—
DRAWING	—	—	WAITING	TRANSLATING	—
TRANSLATING	—	—	WAITING	—	DRAWING

Table 2: *This state table extends the relationships in Table 1 by adding two mouse events and another system state.*

(continued from page 40)

Rubber Banding Figures

Draw uses the standard Windows raster operation (*ROP2*) codes in order to "rubber band" a figure as the mouse cursor is dragged. This occurs in *Tool*, in the case block WM_MOUSEMOVE, which calls the Windows function *SetROP2* with argument R2_NOTXORPEN. This argument sets the XOR (eXclusive OR) drawing mode. Using the previous values of x2 and y2, the XOR mode causes the tool function called by the *DrawFig* array to erase the existing figure. *DrawFig* calls the tool function again, using the current values of x2 and y2 to draw the new figure. When the same figure is drawn twice in the same place in XOR drawing mode, figures in the background are left unchanged. Case WM_LBUTTONUP calls *ROP2* with argument R2_COPYPEN, setting the COPY drawing mode. In COPY mode, the background color fills the interior of the figure, erasing overlapped portions of any underlying figures.

System State Tables

The concept of "system state" is central to understanding Draw. It is the current state of the application that determines the response of the program to a mouse event. We use a single variable (*iState*, in *Tool*) to represent the system state.

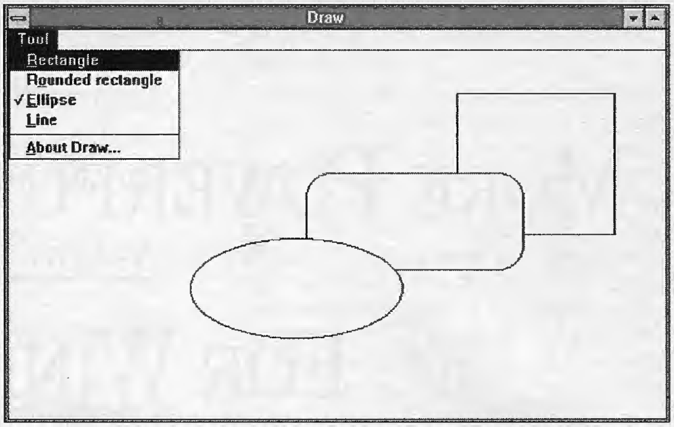


Figure 1: *Sample screen display for DRAW.EXE*

Table 1 shows Draw's state table. It shows how the two system states (WAITING and DRAWING) are related to the three mouse events (WM_LBUTTONDOWN, WM_MOUSEMOVE, WM_LBUTTONUP) that DRAW's tools use to display figures. When you use DRAW, you send a series of mouse events to *Tool*. *Tool*'s response to a given mouse event depends not only on that event, but also on the sequence of previous events. *Tool* records this sequence of mouse events as transitions in system state, and the state table documents these transitions.

To use Table 1, enter at the initial system state, WAITING, and read across to see the effect of mouse events. WM_MOUSEMOVE and WM_LBUTTONUP have no effect, but WM_LBUTTONDOWN causes a transition to a new system state, DRAWING, and starts the tool. Reenter the table at the new system state, DRAWING, and again read across. Now WM_LBUTTONDOWN and WM_MOUSEMOVE have no effect, but WM_LBUTTONUP causes a state transition back to WAITING, and stops the tool. *Tool* responds to only one sequence of mouse events: WM_LBUTTONDOWN, WM_MOUSEMOVE, WM_LBUTTONUP. This sequence is reflected in only one path through the state table: WAITING → DRAWING → WAITING.

System state can both determine the response to a mouse event and be determined by a mouse event. For example, in case WM_MOUSEMOVE of *Tool*, if *iState* equals DRAWING, the old figure is erased and the new one is drawn; if *iState* equals WAITING, there is no effect (break). By contrast, in case WM_LBUTTONDOWN, if *iState* equals WAITING, *iState* is changed to DRAWING and the endpoint coordinates are assigned.

Draw's tools are simple; thus Table 1 is correspondingly simple. Table 2 is a slightly more involved example that describes what would happen if two mouse events, right-button-down (WM_RBUTTONDOWN) and right-button-up (WM_RBUTTONUP), were added to translate (that is, change the location of) the figure being drawn. In this example, when you depress the right button while drawing, mouse moves translate the figure without changing its shape.

To use Table 2, enter at the initial system state, WAITING, and read across. WM_LBUTTONDOWN causes a transition to DRAWING and starts the tool, as before. WM_LBUTTONUP causes a transition back to WAITING, as before, and stops the tool. An intervening WM_RBUTTONDOWN, however, changes the state to TRANSLATING. In state TRANSLATING, WM_MOUSEMOVEs cause translations rather than rubber banding. WM_RBUTTONUP changes the state back to DRAWING. You can alternate between rubber banding (state DRAWING) and translating (state

(continued from page 42)

TRANSLATING) until the final WM_LBUTTONUP.

This expanded tool responds to the mouse-event sequence: WM_LBUTTONDOWN, WM_MOUSEMOVE, WM_RBUTTONDOWN, WM_MOUSEMOVE, WM_RBUTTONUP, WM_MOUSEMOVE, WM_LBUTTONUP, and this sequence is reflected in a path through the state table as: WAITING → DRAWING → TRANSLATING → DRAWING → WAITING.

In implementing the state tables, we coded them as two-dimensional switches, that is, nested switch statements. More elaborate tables might require an array-based approach. In Draw, the mouse event controls the outer switch statement, and the state variable controls the inner one. For Table 2, the skeleton for case WM_MOUSEMOVE is shown in Example 1. To fully flesh out the example, case blocks for WM_RBUTTONDOWN and WM_RBUTTONUP would have to be added to the switch statement that selects from *iMessage* in *Tool*. Also, the state variable *iState* would have to be changed accordingly.

```
case WM_MOUSEMOVE:
  switch (iState)
    {
    case WAITING:
      /* Tool not started; nothing to do. */
      break;  /* WAITING */
    case DRAWING:
      /* User is rubber banding. Erase old figure and draw new
         figure. Reset statics (x2,y2) to mouse coordinates. */

      ............ rubber banding code here ...........
      break;  /* DRAWING */
    case TRANSLATING:
      /* User is translating. Erase old figure and draw new
         figure. Reset statics (x1,y1) and (x2,y2) to translated
         values. */

      ............ translating code here ............
      break;  /* TRANSLATING */
    }  /* switch (iState) */
  break;  /* WM_MOUSEMOVE */
```

Example 1: *The skeleton for case WM_MOUSEMOVE*

Table 2 demonstrates that, as permissible sequences of mouse events and consequent system states are added to a program being developed, the complexity of the interactions increases rapidly. The code that describes these interactions necessarily becomes equally complex. Poorly managed complexity leads to intractability. State tables are a way to cut through this complexity. If state tables are first used to describe the interactions are constructed before the code is written they provide a guide for writing the code. New features can be added with only a minimal alteration of working code. State tables become a means of managing complexity and are therefore a valuable aid in writing and documenting Windows applications that make heavy use of mouse events.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 18**

Listing One *(Text begins on page 39.)*

```
/***********DRAW.H : header file for DRAW.C *************************/

#define WAITING  0   /* the possible values for variable iState in  */
#define DRAWING  1   /* Tool() are WAITING and DRAWING              */

/*  These constants are the possible values for iMenuChoice, the variable
 *  recording the user's menu choice. The old menu choice must be stored
 *  so the check mark can be removed from the menu when a new menu choice
 *  is made. Do not change. */
#define IDM_RECT        100
#define IDM_ROUND_RECT  101
#define IDM_ELLIPSE     102
#define IDM_LINE        103
#define IDM_ABOUT       104

/* These constants are the possible values for iFigType, the variable
 *  recording the current FIGURE, as chosen through the menu. The value is
 *  also stored in the iType field in faList[] and is used to determine
 *  which drawing function is called upon from DrawFig[], the array of
 *  pointers to functions; since these values are indices into an array,
 *  starting at 0, they may not be changed. */
#define FT_RECT         (IDM_RECT       - IDM_RECT)
#define FT_ROUND_RECT   (IDM_ROUND_RECT - IDM_RECT)
#define FT_ELLIPSE      (IDM_ELLIPSE    - IDM_RECT)
#define FT_LINE         (IDM_LINE       - IDM_RECT)

/* maximum number of FIGUREs in faList[] */
#define MAX_FIGS 1000

/* FIGUREs in faList[]: rectangle, rounded rectangle, ellipse, line */
typedef struct
{ int  iType;
  RECT rsCoord;
} FIGURE;

/* global variables */
FIGURE faList[MAX_FIGS];    /* List of FIGUREs */
int    iListSize;           /* tally number of displayed FIGUREs */
HANDLE hInst;               /* current instance */
RECT   rClient;             /* client area in scr coords for ClipCursor() */

/* function prototypes */
long FAR  PASCAL WndProc(HWND hWnd, unsigned iMessage, WORD wParam,
                                                      LONG lParam);
void NEAR PASCAL Tool(HWND hWnd, unsigned iMessage, LONG lParam,int iFigType);
BOOL FAR  PASCAL DrawRoundRect(HDC hDC, int x1, int y1, int x2, int y2);
BOOL FAR  PASCAL DrawLine(HDC hDC, int x1, int y1, int x2, int y2);
BOOL FAR  PASCAL AboutDraw(HWND hDlg, unsigned message, WORD wParam,
                                                      LONG lParam);
/* DrawFig[] is an array of pointers to FAR PASCAL functions, each with parms
 * (HDC,int,int,int,int) and returning BOOL. Note Rectangle() and Ellipse() are
 * MS Windows GDI calls, while DrawRoundRect() and DrawLine() are our calls. */
BOOL (FAR PASCAL *DrawFig[4])(HDC hDC, int x1, int y1, int x2, int y2)
     = {Rectangle, DrawRoundRect, Ellipse, DrawLine};
```

End Listing One

Listing Two

```
/******* DRAW.C by Michael A. Bertrand and William R. Welch. *******/

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

int PASCAL WinMain(HANDLE hInstance, HANDLE hPrevInstance, LPSTR lpszCmdLine,
                                                         int nCmdShow)
  /*     hInstance     : current instance handle
   *     hPrevInstance : previous instance handle
   *     lpszCmdLine   : current command line
   *     nCmdShow      : display either window or icon
   */
{ static char szAppName [] = "Draw";
  static char szIconName[] = "DrawIcon";
  static char szMenuName[] = "DrawMenu";

  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;

  if (!hPrevInstance)              /* Register application window class. */
  { wc.style          = CS_HREDRAW | CS_VREDRAW;
    wc.lpfnWndProc    = WndProc;    /* function 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,              /* window class */
                      szAppName,              /* window caption */
                      WS_OVERLAPPEDWINDOW,    /* normal window style */
                      CW_USEDEFAULT,          /* initial x-position */
                      CW_USEDEFAULT,          /* initial y-position */
                      CW_USEDEFAULT,          /* initial x-size */
                      CW_USEDEFAULT,          /* initial y-size */
                      NULL,                   /* parent window handle */
                      NULL,                   /* window menu handle */
                      hInstance,              /* program instance handle */
                      NULL);                  /* create parameters */
  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)
  /*  IN:  hWnd      : handle to window
   *       iMessage  : message type
   *       wParam    : drawing tool selected from menu (when WM_COMMAND msg)
   *       lParam    : mouse coords (x == loword, y == hiword) */
{ static int  iMenuChoice = IDM_RECT;    /* default menu choice */
  static int  iFigType    = FT_RECT;     /* default figure type */
  HDC         hDC;              /* must generate our own handle to DC to draw */
  HMENU       hMenu;            /* handle for drop down menu */
  PAINTSTRUCT ps;               /* needed when receive WM_PAINT message */
  int         ndx;              /* to traverse faList[] when draw it */
  FARPROC     lpProcAbout;      /* pointer to "AboutDraw" function */
  POINT       pt;               /* for ClientToScreen() */

  switch(iMessage)
  {  case WM_SIZE:  /*convert client coords to scrn coords for ClipCursor()*/
         pt.x = pt.y = 0;
         ClientToScreen(hWnd, &pt);
         rClient.left = pt.x;
         rClient.top  = pt.y;
         pt.x = LOWORD(lParam);
         pt.y = HIWORD(lParam);
         ClientToScreen(hWnd, &pt);
         rClient.right  = pt.x;
         rClient.bottom = pt.y;
         break;
    case WM_COMMAND:
      switch(wParam)
      {  case IDM_RECT:
         case IDM_ROUND_RECT:
         case IDM_ELLIPSE:
         case IDM_LINE:
          /* New FIGURE chosen by user : uncheck old choice and check new
           * choice on menu; reset iMenuChoice according to user choice. */
          hMenu = GetMenu(hWnd);
          CheckMenuItem(hMenu, iMenuChoice, MF_UNCHECKED);
          CheckMenuItem(hMenu, iMenuChoice = wParam, MF_CHECKED);
          /* User has chosen new FIGURE : set iFigType accordingly. */
          iFigType = iMenuChoice - IDM_RECT;
          break;  /* case IDM_LINE ... */
        case IDM_ABOUT:
          /* "About" chosen by user : call "AboutDraw" function. */
          lpProcAbout = MakeProcInstance(AboutDraw, hInst);
          DialogBox (hInst, "AboutDraw", hWnd, lpProcAbout);
          FreeProcInstance(lpProcAbout);
          break; /* IDM_ABOUT */
       }  /* switch(wParam) */
      break;  /* WM_COMMAND */
    case WM_LBUTTONDOWN:
    case WM_MOUSEMOVE:
    case WM_LBUTTONUP:
      /* Mouse events passed on to Tool() for processing. */
      Tool(hWnd, iMessage, lParam, iFigType);
      break;  /* WM_LBUTTONDOWN... */
    case WM_PAINT:
      /* Repaint window when resized. */
      hDC = BeginPaint(hWnd, &ps);
      /* Draw list of FIGUREs. */
      for (ndx = 0; ndx < iListSize; ndx++)
        DrawFig[faList[ndx].iType](hDC, faList[ndx].rsCoord.left,
                                        faList[ndx].rsCoord.top,
                                        faList[ndx].rsCoord.right,
                                        faList[ndx].rsCoord.bottom);
      EndPaint(hWnd, &ps);
      break;  /* WM_PAINT */
    case WM_DESTROY:
      /* Destroy window when application terminated. */
      PostQuitMessage(0);
      break;  /* WM_DESTROY */
    default:
      return(DefWindowProc(hWnd, iMessage, wParam, lParam));
  }  /* switch(iMessage) */
  return(0L);
}
/**************************************************************/
void NEAR PASCAL Tool(HWND hWnd, unsigned iMessage, LONG lParam, int iFigType)
  /*       Process mouse event and draw.
   *  IN:  hWnd      : handle to window
   *       iMessage  : mouse event (WM_LBUTTONDOWN, WM_MOUSEMOVE, WM_LBUTTONUP)
   *       lParam    : mouse coords (x == loword, y == hiword)
   */
{ static int  x1, y1;      /* coordinates of button-down point */
  static int  x2, y2;      /* coordinates of mouse */
  static int  iState;      /* WAITING or DRAWING */
  HDC         hDC;         /* must generate our own handle to DC to draw */
  switch(iMessage)
  {  case WM_LBUTTONDOWN:
      /* Protect array from overflow : if array full, notify and out. */
      if (iListSize == MAX_FIGS)
      { MessageBox(hWnd,"Figure array full","Note",MB_ICONEXCLAMATION|MB_OK);
        break;  /* WM_LBUTTONDOWN */
      }
      /* If not drawing, reset iState and store button-down point. */
      if (iState == WAITING)
      { ClipCursor(&rClient);         /* restrict cursor */
        iState = DRAWING;             /* starting drag */
        x1 = x2 = LOWORD(lParam);     /* store user point in statics */
        y1 = y2 = HIWORD(lParam);
```

(continued on page 46)

Listing Three *(Listing continued, text begins on page 39.)*

```
        }
        break;  /* WM_LBUTTONDOWN */
      case WM_MOUSEMOVE:
        /* If drawing, erase old figure and draw new one at mouse. */
        if (iState == DRAWING)
        { hDC = GetDC(hWnd);
          SetROP2(hDC, R2_NOTXORPEN);                    /* draw in XOR */
          DrawFig[iFigType](hDC, x1, y1, x2, y2);        /* erase old */
          x2 = LOWORD(lParam);                           /* get 2nd user pt */
          y2 = HIWORD(lParam);
          DrawFig[iFigType](hDC, x1, y1, x2, y2);        /* draw new */
          ReleaseDC(hWnd, hDC);
        }
        break;  /* WM_MOUSEMOVE */
      case WM_LBUTTONUP:
        /* If drawing, write in COPY mode and store in faList[]. */
        if (iState == DRAWING)
        { ClipCursor(NULL);              /* no longer restrict cursor */
          iState = WAITING;              /* ending draw */
          hDC = GetDC(hWnd);
          SetROP2(hDC, R2_COPYPEN);      /* COPY pen for final FIGURE */
          DrawFig[iFigType](hDC, x1, y1, x2, y2);  /* draw FIGURE */
          ReleaseDC(hWnd, hDC);
          faList[iListSize].iType = iFigType;      /* put FIGURE in faList[] */
          faList[iListSize].rsCoord.left   = x1;
          faList[iListSize].rsCoord.top    = y1;
          faList[iListSize].rsCoord.right  = x2;
          faList[iListSize].rsCoord.bottom = y2;
          iListSize++;        /* bump tally, since just added figure to list */
        }
        break;  /* WM_LBUTTONUP */
    }  /* switch(iMessage) */
}
/*****************************************************************/
BOOL FAR PASCAL DrawRoundRect(HDC hDC, int x1, int y1, int x2, int y2)
  /*  IN:   hDC    : display context in which to draw
   *        x1, y1 : coordinates of first corner
   *        x2, y2 : coordinates of second corner
   *  RET:  returns BOOL for consistency with GDI's Rectangle() and Ellipse()
   *  NOTE: GDI's RoundRect() is used to draw, but RoundRect() requires x-
   *        and y-diameters of ellipse used for rounding. This routine sets
   *        the common diameter equal to half the smallest side, then calls
   *        RoundRect(). Array DrawFig[] contains a pointer to this function.
   */
{ int dx, dy;          /* sides of rectangle, as positive values */
  int diameter;        /* diameter of circle used for rounding */

  dx = (x1 < x2) ? (x2 - x1) : (x1 - x2);    /* x-side of rect (positive) */
  dy = (y1 < y2) ? (y2 - y1) : (y1 - y2);    /* y-side of rect (positive) */
  diameter = (dx < dy) ? dx/2 : dy/2;         /* half smallest side */
  RoundRect(hDC, x1, y1, x2, y2, diameter, diameter);  /* call GDI */
  return(TRUE);
}
/*****************************************************************/
BOOL FAR PASCAL DrawLine(HDC hDC, int x1, int y1, int x2, int y2)
  /*
   *  IN:   hDC    : display context in which to draw
   *        x1, y1 : coordinates of first endpoint
   *        x2, y2 : coordinates of second endpoint
   *  RET:  returns BOOL for consistency w/GDI's Rectangle() and Ellipse().
   *  NOTE: Array DrawFig[] contains a pointer to this function.
   */
{ MoveTo(hDC, x1, y1);  /* MoveTo() and LineTo() are GDI calls. */
  LineTo(hDC, x2, y2);
  return(TRUE);
}
/*****************************************************************/
BOOL FAR PASCAL AboutDraw(HWND hDlg,unsigned iMessage,WORD wParam,LONG lParam)
  /*   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.
   */
{  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 */
}
```


End Listing Two

Listing Three

```
#*************** DRAW : Make file for DRAW.C **************************
# To make program : NMAKE DRAW
# Linker and Resource Compiler: draw.exe depends on draw.obj draw.def draw.res
# Linker options as follows :
# /A:16 : align on paragraphs
# /CO   : add symbol information to EXE for CodeView
# /NOD  : don't search default libs (use only those in link response file)

draw.exe: draw.obj draw.def draw.res
    link /A:16 /CO /NOD draw,,, libw slibcew, draw.def
    rc draw.res

# Microsoft C Compiler : draw.obj contingent on draw.c, draw.h
# Compiler options as follows :
# -c  : compile only
# -Gs : remove stack probe before function calls
# -Gw : for MS Windows
# -Od : disable code optimization to help with debugging
# -W3 : highest warning level (flags ANSI incompatibilities)
# -AS : small model
# -Zp : pack structures (required by MS Windows)
# -Zi : add symbol information to OBJ for CodeView
draw.obj: draw.c draw.h
    cl -c -Gsw -Od -W3 -AS -Zpi draw.c

# Resource Compiler : draw.res contingent on draw.rc, draw.h
draw.res: draw.rc draw.h
    rc -r -v draw.rc
```

End Listing Three

Listing Four

```
;;*************DRAW.DEF : Definition file for DRAW.C*************
NAME         DRAW
DESCRIPTION  'MS Windows Draw Program (c) 1990 M. Bertrand & W. Welch'
EXETYPE      WINDOWS
STUB         'WINSTUB.EXE'
CODE         MOVEABLE PRELOAD
DATA         MOVEABLE PRELOAD SINGLE
HEAPSIZE     1024
STACKSIZE    4096
EXPORTS      WndProc
             AboutDraw
```

End Listing Four

Listing Five

```
/*********** DRAW.RC : resource file for DRAW.C ******************/

#include "windows.h"
#include "draw.h"

DrawIcon ICON DRAW.ICO

DrawMenu MENU
BEGIN
  POPUP "&Tool"
  BEGIN
    MENUITEM "&Rectangle",          IDM_RECT, CHECKED
    MENUITEM "R&ounded rectangle", IDM_ROUND_RECT
    MENUITEM "&Ellipse",            IDM_ELLIPSE
    MENUITEM "&Line",               IDM_LINE
    MENUITEM Separator
    MENUITEM "&About Draw...",      IDM_ABOUT
  END
END

/*  "AboutDraw" dialog box contains 3 types of controls :
 *     CTEXT to display centered text at x-coordinates 8, 24, 40, 56
 *     ICON to display DRAW's icon at coords relative (20,20)
 *     DEFPUSHBUTTON to display 32x14 OK push button at coords (60,74)
 */
AboutDraw DIALOG 30, 30, 150, 94
CAPTION "About Draw"
STYLE DS_MODALFRAME | WS_CAPTION | WS_SYSMENU
BEGIN
  CTEXT "Microsoft Windows"                      -1,  0,  8, 152,  8
  CTEXT "Draw"                                   -1,  0, 24, 152,  8
  CTEXT "Copyright (c) 1990"                     -1,  0, 40, 152,  8
  CTEXT "Michael A. Bertrand and William R. Welch" -1,  0, 56, 152,  8
  ICON  "DrawIcon"                               -1, 20, 20,  19, 26
  DEFPUSHBUTTON "&OK" IDOK, 60, 74, 32, 14,    WS_GROUP
END
```

End Listings