

# User Interface Code

---

This appendix shows the source code that manages the basic setup and user interface for the Venn Diagrammer application.

```

PROGRAM VennDiagrammer;
  USES
    Global, Utilities, Dialog, Preferences, VennProcs;

  VAR
    gDone:      Boolean;

{DoInitManagers: initialize Toolbox Managers}
  PROCEDURE DoInitManagers;
  BEGIN
    MaxApplZone;           {extend heap zone to limit}
    MoreMasters;           {get 64 more master pointers}

    InitGraf(@thePort);    {initialize QuickDraw}
    InitFonts;              {initialize Font Manager}
    InitWindows;           {initialize Window Manager}
    InitMenus;              {initialize Menu Manager}
    TEInit;                 {initialize TextEdit}
    InitDialogs(NIL);       {initialize Dialog Manager}

    FlushEvents(everyEvent, 0); {clear event queue}
    InitCursor;             {initialize cursor to arrow}
  END;

{DoSetupMenus: set up the menu bar}
  PROCEDURE DoSetupMenus;
  VAR
    menuBar:      Handle;
  BEGIN
    menuBar := GetNewMBar(rMenuBar);
    IF menuBar = NIL THEN
      DoBadError(eCantFindMenus);

    SetMenuBar(menuBar);
    DisposeHandle(menuBar);
    AppendResMenu(GetMenuHandle(mApple), 'DRVr');
  
```

User Interface Code

```

    DrawMenuBar;
END;

{DoUpdate: update a window}
PROCEDURE DoUpdate (myWindow: WindowPtr);
    VAR
        myHandle:    MyDocRecHnd;
        myRect:      Rect;           {tool rectangle}
        origPort:    GrafPtr;
        origPen:      PenState;
        count:        Integer;
    BEGIN
        GetPort(origPort);           {remember original drawing port}
        SetPort(myWindow);

        BeginUpdate(myWindow);       {clear update region}
        EraseRect(myWindow^.portRect);

        IF IsAppWindow(myWindow) THEN
            BEGIN
                {Draw two lines separating tools area from work area.}
                GetPenState(origPen); {remember original pen state}
                PenNormal;             {reset pen to normal state}
                WITH myWindow^ DO
                    BEGIN
                        MoveTo(portRect.left, portRect.top + kToolHt);
                        Line(portRect.right, 0);
                        MoveTo(portRect.left, portRect.top + kToolHt + 2);
                        Line(portRect.right, 0);
                    END;

                {Redraw the tools area in the window.}
                FOR count := 1 TO kNumTools DO
                    BEGIN
                        SetRect(myRect, kToolWd * (count - 1), 0, kToolWd * count,
                               kToolHt);
                        DoPlotIcon(myRect, gToolsIcons[count], myWindow, srcCopy);
                    END;

                {Redraw the status area in the window.}
                myHandle := MyDocRecHnd(GetWRefCon(myWindow));
                DoStatusText(myWindow, myHandle^.statusText);
            END;
        END;
    END;

```

User Interface Code

```

        {Draw the rest of the content region.}
        DoVennDraw(myWindow);

        SetPenState(origPen);           {restore previous pen state}
    END; {IF IsAppWindow}

    EndUpdate(myWindow);
    SetPort(origPort);                 {restore original drawing port}
END;

{DoCreateWindow: create a new window}
FUNCTION DoCreateWindow: WindowPtr;
    VAR
        myPointer: Ptr;
        myWindow: WindowPtr;
        myHandle: MyDocRecHnd;
    BEGIN
        myPointer := NewPtr(sizeof(WindowRecord));
        IF myPointer = NIL THEN
            exit(DoCreateWindow);

        myWindow := GetNewWindow(rVennD, myPointer, WindowPtr(-1));
        IF myWindow <> NIL THEN
            BEGIN
                SetPort(myWindow);
                myHandle := MyDocRecHnd(NewHandleClear(sizeof(MyDocRec)));

                IF myHandle <> NIL THEN
                    BEGIN
                        HLockHi(Handle(myHandle));
                                                {lock the data high in the heap}
                        SetWRefCon(myWindow, LongInt(myHandle));
                                                {attach data handle to window record}

                        DoSetWindowTitle(myWindow);           {set the window title}

                        {Define initial window settings.}
                        WITH myHandle^^ DO
                            BEGIN
                                figure := 1;
                                mood[1] := 1;
                                mood[2] := 1;
                                mood[3] := 1;

```

User Interface Code

```

        isAnswerShowing := FALSE;
        isExistImport := gGiveImport;
    END;
    DoGetRandomTerms(myWindow);
    DoCalcAnswer(myWindow);

    {Position the window and display it.}
    DoPositionWindow(myWindow);
    ShowWindow(myWindow);

    END {IF myHandle <> NIL}
ELSE
    BEGIN
        {couldn't get a data record}
        CloseWindow(myWindow);
        DisposePtr(Ptr(myWindow));
        myWindow := NIL;      {so pass back NIL}
    END;
END;

DoCreateWindow := myWindow;
END;

{DoCloseDocWindow: dispose a document window and all its data structures}
PROCEDURE DoCloseDocWindow (myWindow: WindowPtr);
    VAR
        myHandle:   MyDocRecHnd;
    BEGIN
        IF myWindow = NIL THEN
            exit(DoCloseDocWindow)      {ignore NIL windows}
        ELSE
            BEGIN
                myHandle := MyDocRecHnd(GetWRefCon(myWindow));
                DisposeHandle(Handle(myHandle));
                CloseWindow(myWindow);    {close the window}
                DisposePtr(Ptr(myWindow)); {and release the storage}
            END;
        END;
    END;

{DoCloseWindow: close a window}
PROCEDURE DoCloseWindow (myWindow: WindowPtr);
    BEGIN
        IF myWindow <> NIL THEN
            IF IsDialogWindow(myWindow) THEN      {this is a dialog window}

```

User Interface Code

```

        HideWindow(myWindow)
    ELSE IF IsDAccWindow(myWindow) THEN      {this is a DA window}
        CloseDeskAcc(WindowPeek(myWindow)^.windowKind)
    ELSE IF IsAppWindow(myWindow) THEN      {this is a document window}
        DoCloseDocWindow(myWindow);
END;

{DoDrag: handle window dragging}
PROCEDURE DoDrag (myWindow: WindowPtr; mouseLoc: Point);
    VAR
        dragBounds: Rect;
    BEGIN
        dragBounds := GetGrayRgn^^.rgnBBox;
        DragWindow(myWindow, mouseLoc, dragBounds);
    END;

{DoGoAwayBox: process a click in close box}
PROCEDURE DoGoAwayBox (myWindow: WindowPtr; mouseLoc: Point);
    BEGIN
        IF TrackGoAway(myWindow, mouseLoc) THEN
            DoCloseWindow(myWindow);
        END;
    END;

{DoQuit: quit the program}
PROCEDURE DoQuit;
    VAR
        myWindow: WindowPtr;
    BEGIN
        myWindow := FrontWindow;           {close all windows}
        WHILE myWindow <> NIL DO
            BEGIN
                DoUpdate(myWindow);         {force redrawing window}
                DoCloseWindow(myWindow);
                myWindow := FrontWindow;
            END;
            gDone := TRUE;                  {set flag to exit main event loop}
        END;
    END;

{DoActivate: handle activate and deactivate events for the specified window}
PROCEDURE DoActivate (myWindow: WindowPtr; myModifiers: Integer);
    VAR
        myState: Integer;                  {activation state}
        myControl: ControlHandle;

```

User Interface Code

```

BEGIN
    myState := BAnd(myModifiers, activeFlag);

    IF IsDialogWindow(myWindow) THEN
        BEGIN
            myControl := WindowPeek(myWindow)^.controlList;
            WHILE myControl <> NIL DO
                BEGIN
                    HiliteControl(myControl, myState + 255 mod 256);
                    myControl := myControl^^.nextControl;
                END;
            END;
        END;
    END;

{DoDiskEvent: handle disk-inserted events}
    PROCEDURE DoDiskEvent (myEvent: EventRecord);
        VAR
            myResult: Integer;
            myPoint: Point;
        BEGIN
            IF HiWord(myEvent.message) <> noErr THEN
                BEGIN
                    SetPt(myPoint, 100, 100);
                    myResult := DIBadMount(myPoint, myEvent.message);
                END;
            END;
        END;

{MyModalFilter: a basic modal dialog filter function}
    FUNCTION MyModalFilter (myDialog: DialogPtr; VAR myEvent: EventRecord;
                           VAR myItem: Integer): Boolean;
        VAR
            itemType: Integer;
            itemHand: Handle;
            itemRect: Rect;
            myKey: Char;
            myIgnore: LongInt;
        BEGIN
            MyModalFilter := FALSE;           {assume we don't handle the event}

            CASE myEvent.what OF
                updateEvt:
                    BEGIN
                        IF WindowPtr(myEvent.message) <> myDialog THEN

```

## User Interface Code

```

        DoUpdate(WindowPtr(myEvent.message));
                                {update the window behind}
    END;
keyDown, autoKey:
    BEGIN
        myKey := char(And(myEvent.message, charCodeMask));

        {if Return or Enter pressed, do default button}
        IF (myKey = kReturn) OR (myKey = kEnter) THEN
            BEGIN
                GetDItem(myDialog, iOK, itemType, itemHand, itemRect);
                HiliteControl(ControlHandle(itemHand), 1);
                                {make button appear to have been pressed}
                Delay(kVisualDelay, myIgnore);
                HiliteControl(ControlHandle(itemHand), 0);
                MyModalFilter := TRUE;
                myItem := iOK;
            END;

            {if Escape or Cmd-. pressed, do Cancel button}
            IF (myKey = kEscape)
                OR ((myKey = kPeriod)
                    AND (BAnd(myEvent.modifiers, CmdKey) <> 0)) THEN
                BEGIN
                    GetDItem(myDialog, iCancel, itemType, itemHand,
itemRect);

                    HiliteControl(ControlHandle(itemHand), 1);
                                {make button appear to have been pressed}
                    Delay(kVisualDelay, myIgnore);
                    HiliteControl(ControlHandle(itemHand), 0);
                    MyModalFilter := TRUE;
                    myItem := iCancel;
                END;
            END;
        diskEvt:
            BEGIN
                DoDiskEvent(myEvent);
                MyModalFilter := TRUE;           {show we've handled the event}
            END;
        OTHERWISE
            ;
    END; {CASE}
END;

```

User Interface Code

```

{DoAboutBox: handle About... selections}
  PROCEDURE DoAboutBox (myWindow: WindowPtr);
    VAR
      myWindow:    WindowPtr;
      myDialog:    DialogPtr;
      myItem:      Integer;
  BEGIN
    myWindow := FrontWindow;
    IF myWindow <> NIL THEN
      DoActivate(myWindow, 1 - activeFlag);

    myDialog := GetNewDialog(rAboutDial, NIL, WindowPtr(-1));
    IF myDialog <> NIL THEN
      BEGIN
        SetPort(myDialog);
        DoDefaultButton(myDialog);

        REPEAT
          ModalDialog(@MyModalFilter, myItem);
        UNTIL myItem = iOK;

        DisposeDialog(myDialog);
        SetPort(myWindow);
      END;
    END;
  END;

{DoMenuAdjust: adjust menus by enabling and disabling items}
  PROCEDURE DoMenuAdjust;
    VAR
      myWindow:    WindowPtr;
      myMenu:      MenuHandle;
      count:       Integer;
  BEGIN
    myWindow := FrontWindow;

    IF myWindow = NIL THEN
      DisableMenuItem(GetMenuHandle(mFile), iClose)
    ELSE
      EnableMenuItem(GetMenuHandle(mFile), iClose);

    myMenu := GetMenuHandle(mVennD);
    IF IsAppWindow(myWindow) THEN

```

## User Interface Code

```

    FOR count := 1 TO kNumTools DO
        EnableMenuItem(myMenu, count)
    ELSE
        FOR count := 1 TO kNumTools DO
            DisableMenuItem(myMenu, count);

        IF IsDAccWindow(myWindow) THEN
            EnableMenuItem(GetMenuHandle(mEdit), 0)
        ELSE
            DisableMenuItem(GetMenuHandle(mEdit), 0);
        DrawMenuBar;
    END;

{DoMenuCommand: interpret and act on menu selections}
PROCEDURE DoMenuCommand (menuAndItem: LongInt);
    VAR
        myMenuNum: Integer;
        myItemNum: Integer;
        myResult: Integer;
        myDAName: Str255;
        myWindow: WindowPtr;
    BEGIN
        myMenuNum := HiWord(menuAndItem);
        myItemNum := LoWord(menuAndItem);
        GetPort(myWindow);

        CASE myMenuNum OF
            mApple:
                CASE myItemNum OF
                    iAbout:
                        BEGIN
                            DoAboutBox;
                        END;
                    OTHERWISE
                        BEGIN
                            GetMenuItemText(GetMenuHandle(mApple), myItemNum,
                                myDAName);
                            myResult := OpenDeskAcc(myDAName);
                        END;
                END;
            mFile:
                BEGIN
                    CASE myItemNum OF

```

User Interface Code

```

        iNew:
            myWindow := DoCreateWindow;
        iClose:
            DoCloseWindow(FrontWindow);
        iQuit:
            DoQuit;
        OTHERWISE
            ;
    END;
END;
mEdit:
    BEGIN
        IF NOT SystemEdit(myItemNum - 1) THEN
            ;
        END;
    mVennD:
        BEGIN
            myWindow := FrontWindow;
            CASE myItemNum OF
                iCheckVenn:
                    DoVennCheck(myWindow);
                iDoVenn:
                    DoVennAnswer(myWindow);
                iClearVenn:
                    DoVennClear(myWindow);
                iNextTask:
                    DoVennNext(myWindow);
                iCheckArg:
                    DoVennAssess(myWindow);
                iGetVennPrefs:
                    DoModelessDialog(rVennDPrefsDial, gPrefsDialog);
                OTHERWISE
                    ;
            END;
        END;
    END;

    OTHERWISE
        ;
END;
HiliteMenu(0);
END; {DoMenuCommand}

```

{DoContentClick: handle a mouse click in the content area of a window}

## User Interface Code

```

PROCEDURE DoContentClick (myWindow: WindowPtr; myEvent: EventRecord);
  VAR
    myRect:      Rect;                {temporary rectangle}
    count:       Integer;
  BEGIN
    IF NOT IsAppWindow(myWindow) THEN
      exit(DoContentClick);          {make sure it's a document window}

    SetPort(myWindow);              {set port to our window}
    GlobalToLocal(myEvent.where);

    {See if the click is in the tools area.}
    SetRect(myRect, 0, 0, kToolWd * kNumTools, kToolHt);
    IF PtInRect(myEvent.where, myRect) THEN
      BEGIN                          {if so, determine which tool was clicked}
        FOR count := 1 TO kNumTools DO
          BEGIN
            SetRect(myRect, (count - 1) * kToolWd, 0,
                      count * kToolWd, kToolHt);
            IF PtInRect(myEvent.where, myRect) THEN
              Leave;                  {we found the right tool, so stop looking}
            END;
          IF DoTrackRect(myWindow, myRect) THEN
            DoMenuCommand(BitShift(mVennD, 16) +
                          ((kNumTools + 1) - count)); {handle tools selections}
            exit(DoContentClick);
          END;
        END;
      BEGIN
        {See if the click is in the status area.}
        SetRect(myRect, kToolWd * kNumTools, 0,
                  myWindow^.portRect.right, kToolHt);
        IF PtInRect(myEvent.where, myRect) THEN
          BEGIN
            exit(DoContentClick);
          END;
        END;

        {The click must be in somewhere in the rest of the window.}
        DoVennClick(myWindow, myEvent.where);
      END;

    {DoMouseDown: process mouseDown events}
    PROCEDURE DoMouseDown (myEvent: EventRecord);
      VAR

```

User Interface Code

```

myPart:      Integer;
myWindow:    WindowPtr;
BEGIN
  myPart := FindWindow(myEvent.where, myWindow);
  CASE myPart OF
    inMenuBar:
      BEGIN
        DoMenuAdjust;
        DoMenuCommand(MenuSelect(myEvent.where));
      END;
    InSysWindow:
      SystemClick(myEvent, myWindow);
    inDrag:
      DoDrag(myWindow, myEvent.where);
    inGoAway:
      DoGoAwayBox(myWindow, myEvent.where);
    inContent:
      BEGIN
        IF myWindow <> FrontWindow THEN
          SelectWindow(myWindow)
        ELSE
          DoContentClick(myWindow, myEvent);
        END;
      OTHERWISE
        ;
      END;
  END;
END;

{DoKeyDown: respond to keyDown events}
PROCEDURE DoKeyDown (myEvent: EventRecord);
  VAR
    myKey:      char;
  BEGIN
    myKey := chr(BAnd(myEvent.message, charCodeMask));
    IF (BAnd(myEvent.modifiers, CmdKey) <> 0) THEN
      BEGIN
        DoMenuAdjust;
        DoMenuCommand(MenuKey(myKey));
      END;
    END;
  END;

{DoIdle: handle null events}
{currently we use this for auto-processing in Venn diagram windows}

```

## User Interface Code

```

PROCEDURE DoIdle (myEvent: EventRecord);
    VAR
        myWindow:    WindowPtr;
        myHandle:    MyDocRecHnd;
    BEGIN
        myWindow := FrontWindow;
        IF IsAppWindow(myWindow) THEN
            IF gAutoAdjust THEN
                BEGIN
                    myHandle := MyDocRecHnd(GetWRefCon(myWindow));
                    IF myHandle^.needsAdjusting THEN
                        DoVennIdle(myWindow);
                END;
            END;
        END; {DoIdle}

{DoOSEvent: handle OS events}
PROCEDURE DoOSEvent (myEvent: EventRecord);
    VAR
        myWindow:    WindowPtr;
    BEGIN
        CASE BSR(myEvent.message, 24) OF
            mouseMovedMessage:
                BEGIN
                    DoIdle(myEvent);      {right now, do nothing}
                END;
            suspendResumeMessage:
                BEGIN
                    myWindow := FrontWindow;
                    IF (BAnd(myEvent.message, resumeFlag) <> 0) THEN
                        DoActivate(myWindow, activeFlag)      {activate window}
                    ELSE
                        DoActivate(myWindow, 1 - activeFlag);  {deactivate window}
                    END;
                OTHERWISE
                    ;
            END;
        END;

{DoMainEventLoop: the main event loop}
PROCEDURE DoMainEventLoop;
    VAR
        myEvent:    EventRecord;
        gotEvent:    Boolean;           {is returned event for me?}

```

User Interface Code

```

BEGIN
  REPEAT
    gotEvent := WaitNextEvent(everyEvent, myEvent, 15, NIL);
    IF NOT DoHandleDialogEvent(myEvent) THEN
      IF gotEvent THEN
        BEGIN
          CASE myEvent.what OF
            mouseDown:
              DoMouseDown(myEvent);
            keyDown, autoKey:
              DoKeyDown(myEvent);
            updateEvt:
              DoUpdate(WindowPtr(myEvent.message));
            diskEvt:
              DoDiskEvent(myEvent);
            activateEvt:
              DoActivate(WindowPtr(myEvent.message),
                          myEvent.modifiers);
            osEvt:
              DoOSEvent(myEvent);
            keyUp, mouseUp:
              ;
            nullEvent:
              DoIdle(myEvent);
            OTHERWISE
              ;
          END; {CASE}
        END
      ELSE
        DoIdle(myEvent);
      UNTIL gDone;          {loop until user quits}
    END;

BEGIN
  DoInitManagers;          {initialize Toolbox managers}
  DoSetupMenus;            {initialize menus}

  gDone := FALSE;          {initialize global variables}
  gNumDocWindows := 0;     {initialize count of open doc windows}
  gPrefsDialog := NIL;     {initialize ptr to Preferences dialog}

  gAppsResourceFile := CurResFile; {get refnum of the app's resource file}
  gPreferencesFile := -1;   {initialize res ID of preferences file}

```

## User Interface Code

```
DoReadPrefs;                                {read the user's preference settings}

DoVennInit;
DoMainEventLoop;                            {and then loop forever...}
END.
```

