

Windows

This chapter describes how your application can use the Window Manager to create and manage windows. Windows delineate the space within which the user enters or views information, and every Macintosh application that has a user interface should use windows to communicate with the user. Any piece of information that your application presents to the user should be displayed in a window. Similarly, any piece of information that your application solicits from the user should involve the user performing appropriate actions (such as typing or clicking) in a window.

There are two general kinds of windows: document windows and dialog boxes. Document windows are used primarily to allow the user to enter and manipulate information, such as text, graphics, or other data. Often, but not always, the information in a document window can be stored in a file, from which the user can later retrieve it. Dialog boxes are used for many other purposes, such as alerting the user of unusual occurrences, soliciting information from the user, and displaying various application settings or user preferences.

This chapter focuses on techniques for handling windows in general, with particular emphasis on document windows. It shows how to

- determine the type of a window
- create and display windows
- handle events in windows
- close and remove windows

For specific information about dialog windows, see the chapter “Dialog Boxes” later in this book. For a complete description of the capabilities of the Window Manager and for code samples illustrating more advanced window-handling techniques, see the chapter “Window Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*.

About Windows

A *window* is a user interface element that delimits an area on the screen in which the user can enter or view information. Here “information” is intended quite broadly; for example, an application that draws mazes and allows the user to trace a path through the maze by moving the cursor can reasonably be thought of as displaying information (the maze) and allowing the user to enter information (the desired path through the maze). As a result, virtually any interaction with the user that happens outside the menu bar and menus should occur within a window.

The system software provides a wide array of types of window to accommodate the many uses they can have. Window types are distinguished by their appearance and behavior. Some windows have title bars and others do not. Some windows can be moved around on the screen by the user and others cannot. In your choice of a window type, you should be guided by the behavior your application supports in that window.

Windows

Note

You can, if necessary, define your own custom types of windows, with an appearance and behavior unlike the windows provided by the system software. For compatibility reasons, however, this practice is generally discouraged. ♦

As indicated earlier in this chapter, the many types of windows are divided loosely into document windows and dialog boxes. The distinction between windows and dialog boxes is to some degree arbitrary, but in general, you use the Dialog Manager to create and manage dialog boxes and the Window Manager to create and manage document windows. The Dialog Manager essentially just provides a “front-end” to other Toolbox managers, including the Window Manager, the Control Manager, the Event Manager, and TextEdit. The Dialog Manager makes it very easy to create and handle user actions in windows containing controls, text boxes, and other dialog items. However, because dialog boxes are also windows, you might need to use some Window Manager routines as well to manipulate dialog boxes. For example, you can hide a dialog box by calling the `HideWindow` routine (there is no `HideDialog` routine).

When you are designing your application, you need to decide whether to use the Dialog Manager or the Window Manager to create and manage any particular window. For some types of windows, the decision is obvious. For document windows that can contain variable amounts of data and therefore probably require scroll bars and a size box, you’ll want to use the Window Manager. For simple windows that contain a message and possibly a few buttons, you’ll probably want to use the Dialog Manager. As a dialog box becomes more and more complex, however, you’ll want to consider using the Window Manager and other Toolbox managers instead. The Window Manager provides the greatest control over the appearance and behavior of a window. In particular, any time you need to do moderately complex drawing in the window, you should probably use the Window Manager (and `QuickDraw`) instead of the Dialog Manager.

Note

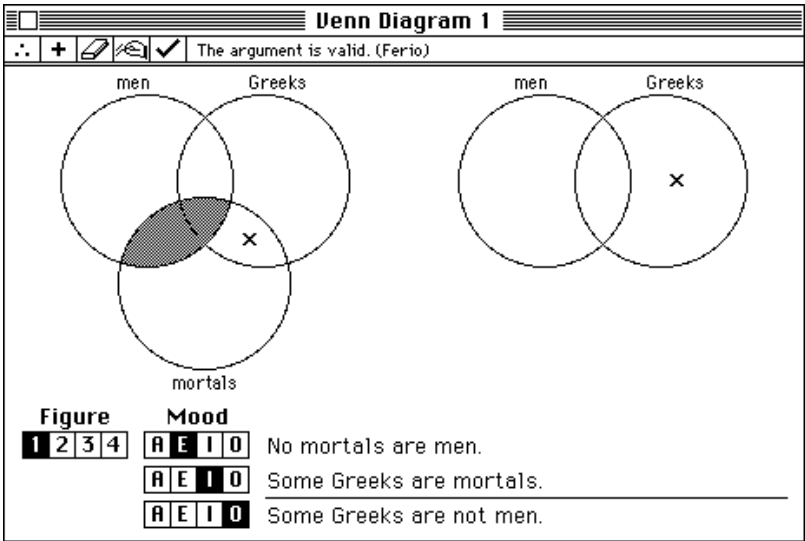
For a more detailed list of factors that can effect the decision whether to use the Dialog Manager or the Window Manager (and other Toolbox managers) to manage a window, see the chapter “Dialog Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*. ♦

Window Parts

The Window Manager defines and supports a set of standard window elements through which the user can manipulate windows. It’s important that your application follow the standard conventions for drawing, moving, resizing, and closing windows. By presenting the standard interface, you make experienced users instantly familiar with many aspects of your application, allowing them to focus on learning its unique features.

The Venn Diagrammer application supports two kinds of windows, a single dialog box for setting general preferences and an unlimited number of document windows for evaluating categorical syllogisms. A sample document window is shown in Figure 6-1.

Figure 6-1 A Venn diagram window



This window contains only two special elements defined by the Window Manager, a title bar and a close box. The **title bar** displays the name of the window and indicates whether it's active or not. The Window Manager displays the title of the window in the center of the title bar, in the system font and system size. If the system font is in the Roman script system, the title bar is 20 pixels high.

The **close box** offers the user a quick way to close a window. If the user clicks the close box, your application should react exactly as if the user had chosen the Close command from the File menu.

Note

Venn Diagrammer's use of standard window elements is purposely restricted to the title bar and close box. Your application's windows should include as many of the standard window elements as are appropriate. ♦

The window shown in Figure 6-1 also contains a number of elements that are defined and managed by the Venn Diagrammer application. Immediately under the title bar is a row of five tools, which allow the user to manipulate the Venn diagram without leaving the window. To the right of the tools is a status area, where the Venn Diagrammer application displays information and other feedback to the user. In Figure 6-1, the status area contains a message indicating that the syllogism under consideration is valid; the status area also shows the traditional name of that valid syllogism (Ferio).

Underneath the tools area and the status area, the document window contains two sets of overlapping circles, which show the Venn diagram for the syllogism's premises and conclusion. The user can alter the contents of any region of overlap by clicking in that area. Shading indicates that the region is known to be empty; an X indicates that the

Windows

region is known to contain something; the lack of either shading or an X indicates that the contents of the region are unknown.

The user can alter the syllogism under consideration by changing the figure of the syllogism and the mood of any of the three statements in the syllogism. Any changes in the figure or mood are instantly reflected in the syllogism shown in the bottom center of the window.

Window Records

You've already seen, in skeletal form at least, how to create a window by calling `NewWindow` (see Listing 1-1 on page 3). When you call `NewWindow`, the Window Manager creates in your application heap a new *window record* that contains information about the new window. The Window Manager defines a window record using the `WindowRecord` data structure, shown in Listing 6-1.

Listing 6-1 The `WindowRecord` data structure

```

TYPE WindowRecord =
  RECORD
    port:          GrafPort;          {window's graphics port}
    windowKind:    Integer;            {class of the window}
    visible:        Boolean;            {visibility}
    hilited:        Boolean;            {highlighting}
    goAwayFlag:    Boolean;            {presence of close box}
    spareFlag:      Boolean;            {presence of zoom box}
    strucRgn:       RgnHandle;          {handle to structure region}
    contrRgn:       RgnHandle;          {handle to content region}
    updateRgn:      RgnHandle;          {handle to update region}
    windowDefProc: Handle;              {handle to window definition }
                                      { function}
    dataHandle:     Handle;              {handle to window state }
                                      { data record}
    titleHandle:    StringHandle;        {handle to window title}
    titleWidth:     Integer;              {title width in pixels}
    controlList:    ControlHandle;        {handle to control list}
    nextWindow:     WindowPeek;          {pointer to next window }
                                      { record in window list}
    windowPic:      PicHandle;           {handle to optional picture}
    refCon:         LongInt;             {storage available to your }
                                      { application}

  END;

```

Windows

As you can see, a window record consists of numerous fields that contain information about the window. The first field (`port`) contains the window's graphics port, a drawing environment with its own coordinate system. The graphics port in turn contains information about that drawing environment, such as the location of the port on the screen, the default size and font of any text that is to be drawn in the port, and so forth.

Because many of the operations you'll perform on windows are in reality operations on the window's graphics port, the Window Manager defines the data type `WindowPtr` as a pointer to the window's graphics port.

```
TYPE
```

```
    WindowPtr    = GrafPtr;
```

For example, each time you want to draw in a window, you need to make sure that the window is the current drawing port. To do so, you can simply pass the window pointer to the `QuickDraw` routine `SetPort`.

```
SetPort(myWindow);
```

You can do this because a window pointer is simply a pointer to a graphics port, which is the first field in a window record. Similarly, you can determine the location of the window on the screen by inspecting the `portRect` field of the graphics port. Recall that Listing 1-1 on page 3 centers the text within the window as follows:

```
WITH gWindow^.portRect DO          {set the position of the pen}
    MoveTo(((right - left) DIV 2) - (StringWidth(gString) DIV 2),
           (bottom - top) DIV 2);
```

Usually you don't need to access or directly modify fields in a window record. If you do need to examine the fields of the window record (other than those contained in the window's graphics port), you can use the `WindowPeek` data type:

```
TYPE
```

```
    WindowPeek    = ^WindowRecord;
```

A `WindowPeek` data type is a pointer to a window record.

Note

Don't get confused here. A window pointer is a pointer to the window's graphics port, not a pointer to the window record. The `WindowPeek` data type is so called because it lets you "peek" into the fields of the window record beyond the graphics port. ♦

Window Types

The `windowKind` field of a window record indicates the type of window that the window record describes. Your application can, if necessary, read the value in that field to determine how to handle a particular window.

Windows

When the Window Manager creates a new window for a desk accessory, it places a negative value (in particular, the reference ID of the desk accessory) in the `windowKind` field of the window. In all other cases, the Window Manager puts one of two constants into that field:

```
CONST
    dialogKind = 2;           {dialog or alert window}
    userKind   = 8;           {window created by an application}
```

You can rely on this behavior to determine what kind of window a given window pointer picks out. Listing 6-2 defines a function `IsAppWindow` that returns `TRUE` if the application created the specified window by calling a Window Manager routine directly. In the case of the Venn Diagrammer application, this means that the window is a document window.

Listing 6-2 Determining if a window is a document window

```
FUNCTION IsAppWindow (myWindow: WindowPtr): Boolean;
BEGIN
    IF myWindow = NIL THEN
        IsAppWindow := FALSE
    ELSE
        IsAppWindow := WindowPeek(myWindow)^.windowKind = userKind;
END;
```

Notice that `IsAppWindow` coerces the window pointer `myWindow` to the type `WindowPeek` before dereferencing it to examine the `windowKind` field.

You can define similar functions to identify dialog boxes and desk accessory windows. Listing 6-3 defines a function `IsDialogWindow` that returns `TRUE` if your application created the specified window by calling a Dialog Manager routine.

Listing 6-3 Determining if a window is a dialog box

```
FUNCTION IsDialogWindow (myWindow: WindowPtr): Boolean;
BEGIN
    IF myWindow = NIL THEN
        IsDialogWindow := FALSE
    ELSE
        IsDialogWindow := WindowPeek(myWindow)^.windowKind = dialogKind;
END;
```

Finally, Listing 6-4 defines a function `IsDAccWindow` that returns `TRUE` if the specified window was created by a desk accessory.

Listing 6-4 Determining if a window is a desk accessory window

```

FUNCTION IsDAccWindow (myWindow: WindowPtr): Boolean;
BEGIN
    IF myWindow = NIL THEN
        IsDAccWindow := FALSE
    ELSE
        IsDAccWindow := WindowPeek(myWindow)^.windowKind < 0;
    END;
END;

```

These three functions are used extensively throughout the code samples in the remainder of this chapter.

Note

The `IsDAccWindow` function is provided to help maintain compatibility with previous system software versions. When your application is running in System 7, it receives events only for its own windows and for windows belonging to desk accessories that were launched in its partition. ♦

Creating Windows

The Venn Diagrammer application allows the user to have multiple document windows (that is, multiple Venn diagram windows) on the desktop at the same time. Each different document window probably displays a different syllogism. As a result, the application needs some way to keep track of each window's current settings.

A standard way to do this is to make use of the `refCon` field in the window record. The `refCon` field is reserved specifically for use by applications, which can set the field (using the `SetWRefCon` procedure) to any 4-byte value. Often, applications store a handle to an application-defined data structure that describes the window. This data structure is often known as a *document record*. Given the window pointer, you can retrieve that handle by calling the `GetWRefCon` function.

The sample code in this book uses a document record of type `MyDocRec` (shown in Listing 6-5) to store information about the current contents of a Venn diagram window.

Listing 6-5 The structure of a document record for the Venn Diagrammer application

```

TYPE MyDocRec =
    RECORD
        figure:           Integer;           {the figure of the syllogism}
        mood:             ARRAY[1..3] OF Integer; {the moods of the statements}
        terms:            ARRAY[1..3] OF Str31;  {the three terms}
        statusText:       Str255;             {most recent status message}
    END;

```

Windows

```

userSolution:      MyDiagramState;      {user's diagram state}
realSolution:      MyDiagramState;      {answer's diagram state}
isAnswerShowing:   Boolean;              {is the answer showing?}
isExistImport:     Boolean;              {stmts imply exists subject?}
needsAdjusting:    Boolean;              {diagram needs adjusting?}
END;
MyDocRecPtr = ^MyDocRec;
MyDocRecHnd = ^MyDocRecPtr;

```

As you can see, the document record used by the Venn Diagrammer application contains fields that describe the current settings of the syllogism in the window, including the figure of the syllogism, the mood of each statement in the syllogism, and the terms used in those statements. The document record also contains fields that maintain information about the current appearance of the window, such as the status message most recently displayed in the window's status area (`statusText` field) and a Boolean value that indicates whether the answer is visible in the window (`isAnswerShowing` field). The Venn Diagrammer application uses that Boolean value to determine how to fill in the regions in the overlapping circles. If the value of `isAnswerShowing` is `TRUE`, the application displays the correct answer (encoded in the `realSolution` field); otherwise, the application displays the user's current answer (encoded in the `userSolution` field).

Note

The structure of the `MyDiagramState` data type is not shown in this book. ♦

The `MyDocRec` data structure also contains two other fields containing Boolean values. These specify whether the statements that make up the syllogism are to be interpreted as having existential import or not, and whether the window needs to be checked for automatic adjustment.

IMPORTANT

If a Venn diagram window contained `TextEdit` fields or controls (such as radio buttons or scroll bars), the document record could be expanded to include handles to those items. Also, if a file were associated with the window, you'd want the document record to include information about that file. In a nutshell, the document record can contain all relevant information about the window that isn't contained in the window record. ▲

The Venn Diagrammer application creates a document record every time it creates a document window, and it stores a handle to the document record in the `refCon` field of the window record. Listing 6-6 shows the `DoCreateWindow` routine, which creates a new document window. This function is called when the application is first launched and whenever the user chooses the `New` command from the `File` menu.

Listing 6-6 Creating a new Venn diagram 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 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;
              isAnswerShowing := FALSE;
              isExistImport := gGiveImport;
            END;
          DoGetRandomTerms(myWindow);
          DoCalcAnswer(myWindow);

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

          END {IF myHandle <> NIL}
        ELSE

```

Windows

```

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

```

The `DoCreateWindow` function first attempts to allocate space in the heap for a window record by calling the Memory Manager's `NewPtr` function. If no space is available, `DoCreateWindow` exits and returns `NIL` to indicate that no new window was created. Otherwise, `DoCreateWindow` creates the new window, whose size and type are defined in a window resource of type `rVennD`.

```

CONST
    rVennD      = 131;                {resource ID of document window}

```

If the new window is successfully created, `DoCreateWindow` next tries to allocate space for a document record. Once again, if the space isn't available, `DoCreateWindow` takes care to dispose of the new window and return `NIL` to the calling routine. Otherwise, `DoCreateWindow` locks the handle to the document record high in the heap and attaches the document record to the window record by calling `SetWRefCon`.

Note

The document record data is locked at the top of the heap to help prevent heap fragmentation. See the chapter "Introduction to Memory Management" in *Inside Macintosh: Memory* for a discussion of when you need to lock data in the heap. ♦

The `DoCreateWindow` function next sets up the window's title (by calling the application-defined procedure `DoSetWindowTitle`) and initializes some of the fields in the document record. Then `DoCreateWindow` calls two further application-defined procedures (`DoGetRandomTerms` and `DoCalcAnswer`) to initialize the `terms` field and the `realSolution` field of the document record. (As for the `userSolution` field, the `NewHandleClear` function, which sets all bytes in the block to 0, automatically initializes it to encode an empty diagram, according to a clever scheme.)

The application-defined procedure `DoPositionWindow` sets the original position of the new window according to the user's expectations and good human interface design. Then `DoCreateWindow` calls the Window Manager procedure `ShowWindow` to display the window. The `ShowWindow` procedure generates and update event for the newly displayed window, thereby causing the Venn Diagrammer application to draw the content region of the window.

Windows

Note

The procedure `DoPositionWindow` is not defined in this book. For a discussion of how to determine the position of a new window, see the chapter “Window Manager” in *Inside Macintosh: Macintosh Toolbox Essentials*. ♦

Handling Window Events

Your application must be prepared to handle two kinds of window-related events:

- mouse and keyboard events in your application’s windows, which are reported by the Event Manager in direct response to user actions
- activate and update events, which are generated by the Window Manager and the Event Manager as an indirect result of user actions

Because Venn Diagrammer does not support text entry, the only relevant keyboard events it needs to handle are keyboard equivalents of menu commands. See the chapter “Menus” in this book for a description of how to handle those events.

This section shows how to handle mouse events as well as update and activate events.

Mouse Events

When your application is active, it receives notice of all mouse-down events in the menu bar, in one of its windows, or in any windows belonging to desk accessories that were launched in its partition. When it receives a mouse-down event, your application should call `FindWindow` to determine where the cursor was when the mouse button was pressed. The `FindWindow` function returns a *part code* that indicates the location of the cursor. These constants define the available part codes:

```
CONST inDesk      = 0;  {none of the following}
      inMenuBar    = 1;  {in menu bar}
      inSysWindow  = 2;  {in desk accessory window}
      inContent    = 3;  {anywhere in content region except size }
                          { box if window is active, }
                          { anywhere including size box if window }
                          { is inactive}
      inDrag       = 4;  {in drag (title bar) region}
      inGrow       = 5;  {in size box (active window only)}
      inGoAway     = 6;  {in close box}
      inZoomIn     = 7;  {in zoom box (window in standard state)}
      inZoomOut    = 8;  {in zoom box (window in user state)}
```

Windows

In addition to returning a part code as its function result, `FindWindow` also returns in its second parameter a pointer to a window, if the user presses the mouse button while the cursor is in a window. Listing 6-7 show how the Venn Diagrammer application handles mouse-down events.

Listing 6-7 Handling mouse-down events

```
PROCEDURE DoMouseDown (myEvent: EventRecord);
VAR
    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;
```

If the user clicks in the menu bar, `DoMouseDown` adjusts the menus and calls the application-defined routine `DoMenuCommand` to handle whatever menu command the user might choose. See the chapter “Menus” in this book for details on handling menu choices.

The `FindWindow` function returns the part code `inSysWindow` only when the user presses the mouse button while the cursor is in a window that belongs to a desk

Windows

accessory launched in your application's partition. You can then call the `SystemClick` procedure, passing it the event record and window pointer. The `SystemClick` procedure makes sure that the event is handled by the appropriate desk accessory. For more information about `SystemClick`, see the chapter "Event Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*.

If the user clicks in a window's drag region (identified by the part code `inDrag`), `DoMouseDown` calls the application-defined routine `DoDrag`, defined in Listing 6-8. The `DoDrag` procedure calls the Window Manager procedure `DragWindow`, which displays an outline of the window, moves the outline as long as the user continues to drag the window, and calls `MoveWindow` to draw the window in its new location when the user releases the mouse button.

Listing 6-8 Dragging a window

```
PROCEDURE DoDrag (myWindow: WindowPtr; mousetloc: Point);
    VAR
        dragBounds: Rect;
BEGIN
    dragBounds := GetGrayRgn^^.rgnBBox;
    DragWindow(myWindow, mousetloc, dragBounds);
END;
```

If the user clicks a window's close box (identified by the part code `inGoAway`), you can call an application-defined procedure to close that window. See "Closing Windows" beginning on page 128 for a discussion of how to close windows.

Finally, the `DoMouseDown` procedure defined in Listing 6-7 handles all user clicks in a window's content region either by selecting the window if it isn't already the frontmost window or by calling the routine `DoContentClick` defined in Listing 6-9.

Listing 6-9 Handling clicks in a window's content region

```
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.}
```

Windows

```

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;

        {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;

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

```

The general strategy employed in the `DoContentClick` procedure is to check each part of the content area that is meaningful to the application and determine whether the mouse click occurred there. Then `DoContentClick` reacts appropriately.

After setting the current drawing port to the specified window, `DoContentClick` calls the `GlobalToLocal` procedure to convert the mouse click location from global coordinates to local coordinates. Then `DoContentClick` checks whether the click occurred in the tools area of the window. If so, `DoContentClick` handles the tool selection by invoking the corresponding menu command and then exiting.

If the mouse click was in the status area of a window, `DoContentClick` simply exits. Otherwise, the user must have clicked somewhere in the content area below the tools and status area. In that case, `DoContentClick` calls the application-defined function `DoVennClick` to handle the event.

Windows

Note

The `DoVennClick` function is not defined in this book, but it's quite simple. It merely checks whether the click occurred in the figure icons, mood icons, or some part of the overlapping circles and, if so, changes the window's document record accordingly and invalidates any affected part of the screen. A portion of `DoVennClick` is shown in Listing 6-10. ♦

Update Events

The Event Manager sends your application an *update event* when part or all of your window's content region needs to be redrawn. Specifically, the Event Manager checks each window's update region every time your application calls `WaitNextEvent` and generates an update event for every window whose update region is not empty.

The Window Manager typically triggers update events when the moving and relayering of windows on the screen requires that one or more windows be redrawn. If the user moves a window that covers part of an inactive window, for example, the Window Manager first redraws the window frame. It then adds the newly exposed area to the window's update region, triggering an update event. In response, your application updates the content region.

Note

Your application can receive update events when it is in either the foreground or the background. In general, however, it doesn't matter whether your update routine is executed in the foreground or the background. ♦

Your application can also trigger update events itself by manipulating the update region. You can add areas to a window's update region by calling the Window Manager procedures `InvalRect` (to add a rectangle to the update region) and `InvalRgn` (to add an arbitrary region to the update region). For example, when the Venn Diagrammer application detects a mouse click in a figure icon, it reacts as shown in Listing 6-10.

Listing 6-10 Handling a click in a figure icon

```
FOR count := 1 TO 4 DO
  BEGIN
    IF PtInRect(myPoint, gFigureRects[count]) THEN
      IF myHandle^.figure <> count THEN      {new rect differ from prev?}
        BEGIN
          InvalRect(gFigureRects[myHandle^.figure]);
          myHandle^.figure := count;
          InvalRect(gFigureRects[myHandle^.figure]);
          InvalRect(gTextBoxes[1]);          {invalidate premises}
          InvalRect(gTextBoxes[2]);
          DoCalcAnswer(myWindow);            {update the current answer}
```

Windows

```

        DoStatusText(myWindow, '');          {remove any existing message}
    END;
END;

```

Your general strategy should be to isolate all drawing that occurs in a document window into your application's update routine. Then, within any other routines, you redraw parts of the window, whenever necessary, by invalidating those parts to add them to the window's update region. Listing 6-11 shows the update routine for Venn Diagrammer.

Listing 6-11 Handling update events

```

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);
                    END;
                END;
            END;
        END;
    END;

```

Windows

```

        DoPlotIcon(myRect, gToolsIcons[count], myWindow, srcCopy);
    END;

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

    {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;

```

In response to an update event, your application calls `BeginUpdate`, draws the window's contents, and then calls `EndUpdate`. The `BeginUpdate` procedure limits the visible region to the intersection of the visible region and the update region. Your application can then update either the visible region or the entire content region—because `QuickDraw` limits drawing to the visible region, only the parts of the window that actually need updating are drawn. The `BeginUpdate` procedure also clears the update region. After you've updated the window, you call `EndUpdate` to restore the visible region in the graphics port to the full visible region.

As you can see in Listing 6-11, the Venn Diagrammer application draws the two lines separating the upper portion of the window's content region and redraws the tools icons. Then it redraws the most recently displayed status message (which it has saved in the window's document record). Finally, `DoUpdate` calls the application-defined routine `DoVennDraw` to draw the remainder of the content area (the overlapping circles, the figure and mood icons, the term labels on the circles, and the syllogism itself).

Note

The `DoVennDraw` routine is not shown in this book, but you've already seen portions of it in the chapter "Drawing" earlier in this book. ♦

Activate Events

The window in which the user is currently working is the *active window*. It's always the frontmost window on the desktop (unless your application supports "floating" windows) and is easily identified by the "racing stripes" in the title bar.

Your application activates and deactivates windows in response to *activate events*, which are generated by the Window Manager to inform your application that a window is becoming active or inactive. Each activate event specifies the window to be changed and the direction of the change (that is, whether it is to be activated or deactivated).

Windows

Your application also triggers activate events itself by calling the `SelectWindow` procedure. When it receives a mouse-down event in an inactive window, for example, your application calls `SelectWindow`, which brings the selected window to the front, removes the highlighting from the previously active window, and adds highlighting to the selected window (see Listing 6-7 on page 120). The `SelectWindow` procedure then generates two activate events: the first one tells your application to deactivate the previously active window; the second, to activate the newly active window.

When you receive the event for the previously active window, you need to do whatever is appropriate to make the window's contents appear inactive. Depending on the design of your application, you might need to

- hide the controls and size box
- remove or alter any highlighting of selections in the window

When you receive the event for the newly active window, you

- draw the controls and size box
- restore the content area as necessary, adding the insertion point in its former location and highlighting any previously highlighted selections

If the newly activated window also needs updating, your application also receives an update event, as described in the previous section, "Update Events."

Note

A switch to one of your application's windows from a different application is handled through suspend and resume events, not activate events. See the chapter "Processes" in this book for a description of how your application can handle suspend and resume events. ♦

Listing 6-12 illustrates the application-defined procedure `DoActivate`, which handles activate events.

Listing 6-12 Handling window activations and deactivations

```
PROCEDURE DoActivate (myWindow: WindowPtr; myModifiers: Integer);
  VAR
    myState:      Integer;           {activation state}
    myControl:    ControlHandle;
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);
        END
      END
    END
END;
```

Windows

```

        myControl := myControl^^.nextControl;
    END;
END;
END;

```

The `DoActivate` procedure is passed a window pointer and the `modifiers` field from the event record corresponding to the activate event. The `modifiers` field contains a bit (defined by the `activeFlag` constant) that indicates whether the event specifies window activation or deactivation.

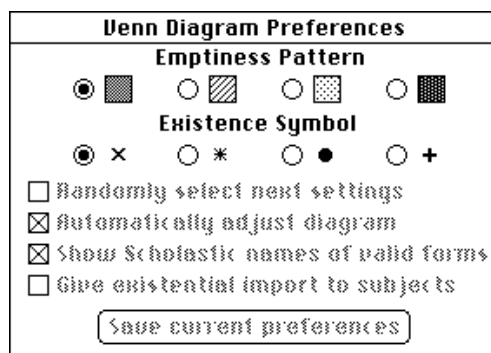
Notice that `DoActivate` does nothing to Venn Diagrammer's document windows, because those windows contain no controls, text, or other items whose visual state might depend on the activation state. For document windows belonging to Venn Diagrammer, the Window Manager handles all the necessary activation and deactivation.

Note

If your application's document windows contain controls (such as scroll bars), your application does need to activate them appropriately. For more information, see the chapter "Control Manager" in *Inside Macintosh: Macintosh Toolbox Essentials*. ♦

However, the Preferences dialog box supported by the Venn Diagrammer application does contain controls, so the `DoActivate` procedure needs to inactivate those controls when the window is deactivated and then reactivate them when the window is activated. The `DoActivate` procedure checks the window's control list and calls the Control Manager procedure `HiliteControl` to perform the necessary activation or deactivation. (The head of the window's control list is stored in the `controlList` field of the window record.) Figure 6-2 shows the Preferences dialog box in its inactive state.

Figure 6-2 An inactive window containing controls



Closing Windows

The user closes a window either by clicking the window's close box (in the upper-left corner of the window) or by choosing the Close command from the File menu. To determine which window to close, you'll proceed in slightly different ways for these two cases. When the user clicks a window's close box, you can get a window pointer for that window by calling the `FindWindow` function in response to the mouse-down event. When the user chooses a menu command, however, you can't do that; instead, you can call the `FrontWindow` function to retrieve a pointer to the frontmost window on the screen.

Note

You'll also want to close any windows that might be on the desktop when the user quits your application. You can do that by repeatedly calling `FrontWindow` until it returns `NIL`. See Listing 9-4 on page 175. ♦

When the user presses the mouse button while the cursor is in the close box, your application should call the `TrackGoAway` function to track mouse movement until the user releases the button, as illustrated in Listing 6-13.

Listing 6-13 Handling clicks in the close box

```
PROCEDURE DoGoAwayBox (myWindow: WindowPtr; mousetloc: Point);
BEGIN
    IF TrackGoAway(myWindow, mousetloc) THEN
        DoCloseWindow(myWindow);
    END;
```

If `TrackGoAway` returns `FALSE`, the user released the button while the cursor was outside the close box, and your application should do nothing. If `TrackGoAway` returns `TRUE`, your application should invoke its own procedure for closing a window.

Listing 6-14 illustrates an application-defined function that closes a window. Notice that the effect of this function varies according to which kind of window it's being asked to close. If the user wants to close a dialog window, `DoCloseWindow` simply hides the window; this strategy leaves the data structures associated with the dialog box in memory, in expectation that the user might open the dialog box again. If the user wants to close a desk accessory window, `DoCloseWindow` calls the Desk Manager routine `CloseDeskAcc` to close that desk accessory.

Listing 6-14 Closing a window

```

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

```

If the window to be closed is a document window, `DoCloseWindow` calls the application-defined procedure `DoCloseDocWindow` defined in Listing 6-15 to deallocate the document record, close the window, and then deallocate the window record.

Listing 6-15 Closing a Venn diagram window

```

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;

```

The `DoCloseDocWindow` procedure retrieves a handle to the document record from the window record. Then it calls `DisposeHandle` to free the memory occupied by the document record. Next `DoCloseDocWindow` closes the window by calling the Window Manager procedure `CloseWindow` and deallocates the window record by calling `DisposePtr`.

Windows

Note

When you create a window, if you allow the Window Manager to allocate memory for the window record (by passing `NIL` as the second parameter to `GetNewWindow`), then you should call the `DisposeWindow` procedure to close the window, instead of calling `CloseWindow` and `DisposePtr`. ♦