

Drawing

This chapter shows how you can draw simple graphics and text inside of windows using QuickDraw, the part of the Macintosh Toolbox that performs graphics operations on the user's screen. All Macintosh applications use QuickDraw indirectly whenever they call other Toolbox managers to create and manage the basic graphic user interface elements (such as windows, controls, and menus). Most applications also call QuickDraw directly to define areas in a window and to draw appropriate graphic elements in those areas. The Venn Diagrammer application, for instance, calls QuickDraw to draw the overlapping circles, the tool icons, and the figure and mood selection icons. It also calls QuickDraw to draw all the text displayed in a window.

This chapter begins with a description of QuickDraw, its basic drawing model, and some of the data structures QuickDraw uses. Then it shows how to

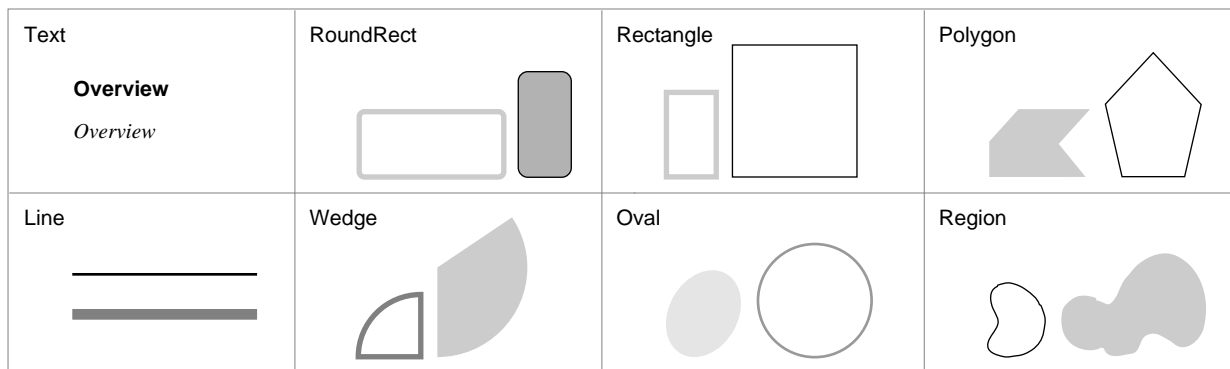
- define and draw simple objects such as lines, rectangles, and circles
- define complex graphic objects by combining simple objects
- outline and fill graphic objects
- draw static (that is, noneditable) text in a window

For a complete description of the drawing capabilities of QuickDraw, see the chapter “QuickDraw Drawing” in *Inside Macintosh: Imaging*. For a complete description of the text capabilities of QuickDraw, see the chapter “QuickDraw Text” in *Inside Macintosh: Text*. To learn how to handle editable text, see the chapter “TextEdit” in *Inside Macintosh: Text*.

About QuickDraw

QuickDraw allows you to draw many types of objects on the Macintosh display screen. Some of these objects are illustrated in Figure 5-1.

Figure 5-1 Samples of QuickDraw's abilities



Drawing

As you can see, you can use QuickDraw to draw

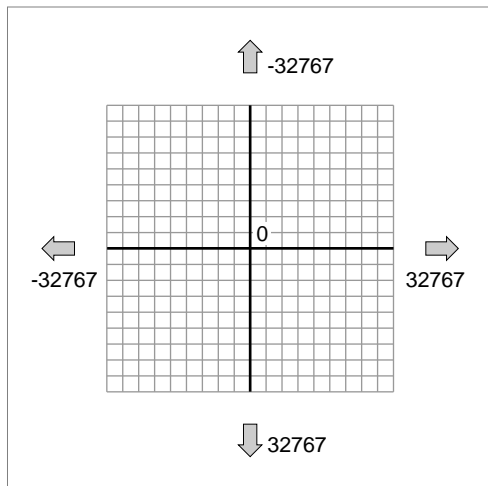
- text characters and strings in a number of fonts, sizes, and styles
- straight lines of any length, width, and pattern
- a variety of simple shapes, including rectangles, rounded-corner rectangles, circles, and ovals
- polygons
- arcs of ovals, or wedge-shaped sections filled with a pattern
- any other arbitrary shape or collection of shapes
- bit images, such as icons, cursors, and patterns

This section explains the basic mathematical model employed by QuickDraw and shows how you can define several of these sorts of objects.

Points

QuickDraw measures location and movement in terms of coordinates on a very large plane. The plane is a two-dimensional grid, with integer coordinates ranging from -32767 to 32767, as illustrated in Figure 5-2.

Figure 5-2 The coordinate plane



The intersection of a horizontal and a vertical grid line marks a *point* on the coordinate plane. Because all coordinates are limited to simple integers, there are 4,294,836,224 unique points in the QuickDraw plane.

Drawing

You can store the coordinates of a point into a Pascal variable of type `Point`, defined by QuickDraw as a record of two integers:

```

TYPE
    VHSelect = (v,h);

    Point =
    RECORD
        CASE INTEGER OF
            0: (v:   Integer;      {vertical coordinate}
               h:   Integer);     {horizontal coordinate}
            1: (vh:  ARRAY[VHSelect] OF Integer);
        END;

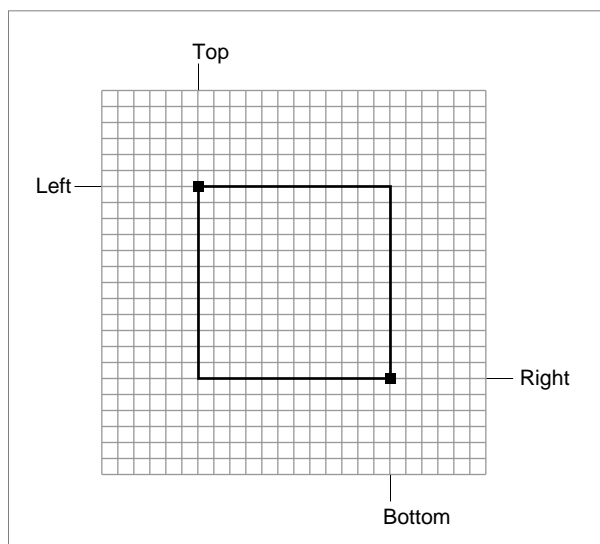
```

The variant part of this record lets you access the vertical and horizontal coordinates of a point either individually or as an array. This book will always use the first way of specifying the coordinates. So, for example, the vertical coordinate of the variable `myPoint` is accessed as `myPoint.v`.

Rectangles

Any two points can define the upper-left and lower-right corners of a *rectangle* on the coordinate plane, as shown in Figure 5-3.

Figure 5-3 A rectangle



Drawing

You can describe a rectangle using a data structure of type `Rect`, which consists of four integers or two points.

```

TYPE Rect =
  RECORD
    CASE INTEGER OF
      0: (top:      Integer;      {top coordinate}
         left:     Integer;      {left coordinate}
         bottom:   Integer;      {bottom coordinate}
         right:    Integer);     {right coordinate}
      1: (topLeft: Point;        {upper-left point}
         botRight: Point);       {lower-right point}
    END;

```

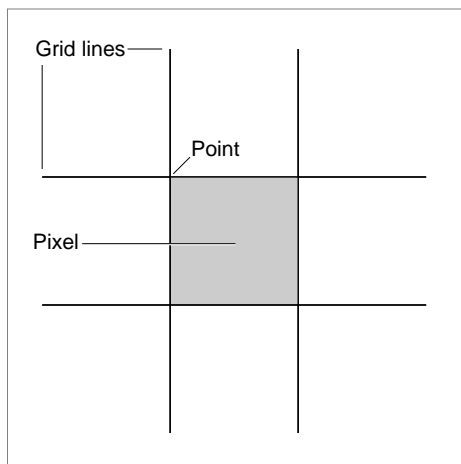
Once again, the record variant allows you to access a variable of type `Rect` either as four boundary coordinates or as two diagonally opposite corner points. This book will always use the first way of specifying a rectangle. So, for example, the top coordinate of the variable `myRect` is accessed as `myRect.top`.

Note

If the bottom coordinate of a rectangle is less than or equal to the top coordinate, or if the right coordinate is less than or equal to the left coordinate, the rectangle is treated as an empty rectangle (that is, one that has no area). ♦

A *pixel* is a physical dot on the screen and corresponds to a rectangle in the QuickDraw coordinate plane that has sides one coordinate long, as shown in Figure 5-4. (This, of course, is the smallest possible rectangle.)

Figure 5-4 Pixels and rectangles



Drawing

You can think of a pixel as corresponding to the point at the top left of the rectangle. There are many more points in the QuickDraw coordinate plane than there are pixels on the screen. As a result, you'll associate small parts of the coordinate plane with areas on the screen. In general, you don't need to worry about where in that large coordinate plane you're working, because QuickDraw always forces you to work with a particular graphics port, which has its own local coordinate system. (A graphics port is a complete drawing environment that defines where and how graphics operations will take place; see page 92 for more information on graphics ports.)

To draw a line, you can simply move to the desired starting point of the line and draw to the desired end. For example, to draw a line in the current graphics port from point (100,150) to the point (200,250), you could do this:

```
MoveTo(100, 150);
LineTo(200, 250);
```

To draw a rectangle, you need to proceed in a slightly different manner. You first need to define the rectangle in the coordinate plane and then perform some graphical operation on the rectangle. Here's an example:

```
SetRect(myRect, 100, 200, 300, 400);
FrameRect(myRect);
```

These two lines of code define a rectangle and then frame it (that is, draw its outline). Instead of just drawing the rectangle's outline, you could also fill the rectangle with the current pattern (by calling `PaintRect`) or with some other pattern (by calling `FillRect`).

Note

Coordinates are passed to `SetRect` in the order left, top, right, bottom (which is different from the order in the `Rect` data type). The word *litterbug* is a useful mnemonic; it contains the letters l, t, r, and b in the correct order. ♦

QuickDraw does not contain data types that describe circles or ovals. Instead, you draw an oval by defining a rectangle and then asking QuickDraw to draw the oval that fits inside of the rectangle. The oval is completely enclosed within the rectangle, and never includes any pixels lying outside the boundary. If the rectangle is a square, then the oval is a circle.

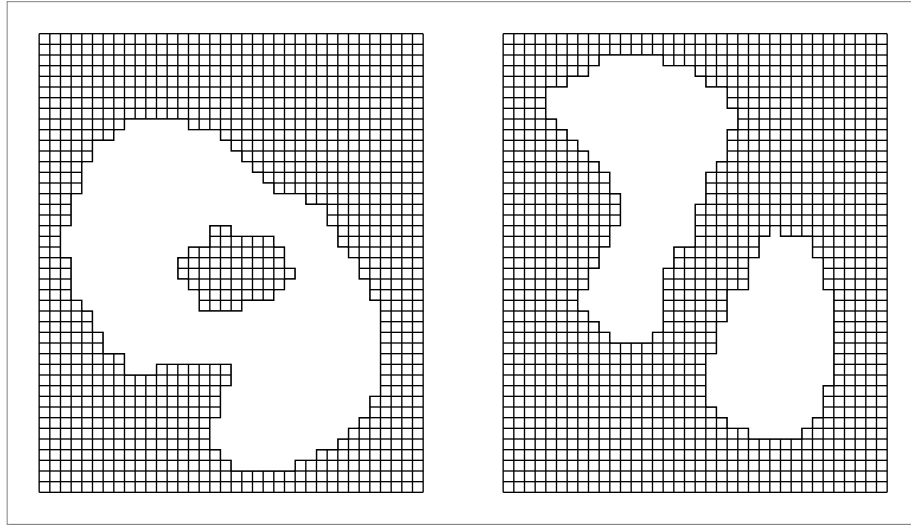
Regions

One of QuickDraw's most powerful capabilities is the ability to work with *regions* of arbitrary size, shape, and complexity. You define a region by drawing its boundary with QuickDraw operations. The boundary can be any set of lines and shapes (even including other regions) forming one or more closed loops. A region can be concave or convex, can consist of one connected area or many separate ones, and can even have holes in the

Drawing

middle. In Figure 5-5, the region on the left has a hole in it, and the region on the right consists of two disjoint areas.

Figure 5-5 Two regions



QuickDraw describes a region using a data structure of type `Region`. This structure contains two fixed-length fields followed by a variable-length field.

```

TYPE Region =
  RECORD
    rgnSize:      Integer;      {size in bytes}
    rgnBBox:      Rect;         {enclosing rectangle}
    {more data if not rectangular}
  END;

  RgnPtr        = ^Region;
  RgnHandle     = ^RgnPtr;
  
```

The `rgnSize` field contains the size, in bytes, of the region variable. The `rgnBBox` field contains a rectangle that completely encloses the region. In general, however, you'll treat the `Region` data structure like a "black box"; you shouldn't need to read the two named fields except in special circumstances.

Drawing

The Venn Diagrammer application uses a number of regions to pick out the areas defined by the overlapping circles. See “Drawing Shapes” beginning on page 94 for details.

Bit Images

Points, rectangles, and regions are mathematical models—data types that QuickDraw uses for defining areas on the screen—but they can also be graphic elements that actually appear on the screen. A rectangle, for example, can mathematically define a particular visible area, but it can also be an object to be framed, painted, or filled. QuickDraw also defines a number of other graphic elements, including icons, bitmaps, patterns, and other bit images, that have only a direct graphic interpretation. An icon, for instance, defines an image not by mapping an abstract mathematical representation onto the screen pixels but by directly indicating which pixels in a given area are to be black and which are to be white.

IMPORTANT

The discussion in this section applies only to black-and-white bit images, which are the simplest cases. For complete information on color bit images (such as color icons), see *Inside Macintosh: Imaging*. ▲

The Macintosh user interface uses bit images extensively, so QuickDraw contains a number of additional data types describing such direct entities and routines to draw them. The Venn Diagrammer application uses two kinds of bit images: bitmaps and patterns.

A **bitmap** is a data structure that defines a physical bit image in terms of the coordinate plane. A bitmap has three parts: a pointer to a rectangular collection of bits, the row width of that rectangular collection, and a boundary rectangle that gives the bitmap both its dimensions and a coordinate system.

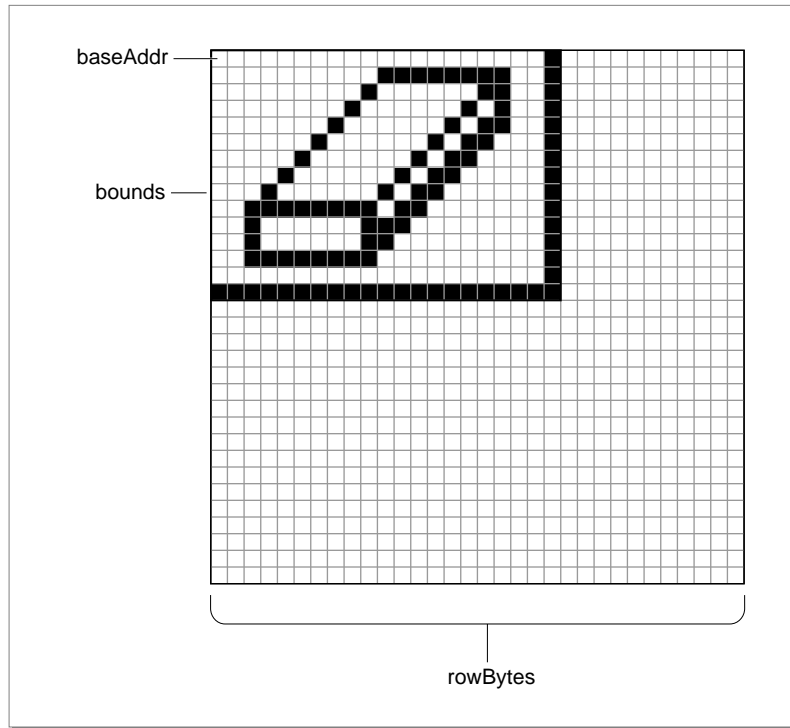
The structure of a bitmap is defined by the `BitMap` data type:

```
TYPE BitMap =
    RECORD
        baseAddr:    Ptr;           {pointer to bit image}
        rowBytes:    Integer;       {row width}
        bounds:      Rect;          {boundary rectangle}
    END;
```

Drawing

Figure 5-6 shows how these three pieces of information define a particular bitmap.

Figure 5-6 A bitmap



The `baseAddr` field is a pointer to the beginning of the bit image in memory. The `rowBytes` field is the row width, in bytes. (Both `baseAddr` and `rowBytes` must contain even values.) The `bounds` field is the bitmap's bounding rectangle. See "Drawing Bit Images" beginning on page 99 for a description of how to display a bitmap.

Ports and Windows

All drawing takes place in a controlled drawing environment known as a *graphics port*. The graphics port defines a number of drawing parameters, such as the current drawing location, the current font and size used for drawing characters, and so forth. In general, you can think of a graphics port as the window within which you're currently drawing.

A graphics port is defined by the `GrafPort` data structure.

```

TYPE GrafPort =
    RECORD
        device:      Integer;      {device-specific information}
        portBits:    BitMap;       {GrafPort's bit map}
        portRect:    Rect;         {GrafPort's rectangle}
    
```

Drawing

```

visRgn:      RgnHandle;  {visible region}
clipRgn:     RgnHandle;  {clipping region}
bkPat:       Pattern;    {background pattern}
fillPat:     Pattern;    {fill pattern}
pnLoc:       Point;      {pen location}
pnSize:      Point;      {pen size}
pnMode:      Integer;    {pen's transfer mode}
pnPat:       Pattern;    {pen pattern}
pnVis:       Integer;    {pen visibility}
txFont:      Integer;    {font number for text}
txFace:      Style;      {text's character style}
txMode:      Integer;    {text's transfer mode}
txSize:      Integer;    {font size for text}
spExtra:     Fixed;      {extra space}
fgColor:     LongInt;    {foreground color}
bkColor:     LongInt;    {background color}
colrBit:     Integer;    {color bit}
patStretch:  Integer;    {used internally}
picSave:     Handle;     {picture being saved}
rgnSave:     Handle;     {region being saved}
polySave:    Handle;     {polygon being saved}
grafProcs:   QDProcsPtr; {low-level drawing routines}
END;

GrafPtr = ^GrafPort;

```

The fields of a `GrafPort` data structure are maintained by QuickDraw, and you should never write directly into those fields. You can, and often must, read the fields of a `GrafPort` structure. For example, it's often useful to read the `portRect` field of a variable of type `GrafPort`, because it gives the rectangle around the content area of a window. (That information was used in Listing 1-1 on page 3 to center a text string.)

QuickDraw always performs drawing operations on the current graphics port. As a result, you should explicitly set the graphics port before doing any drawing. A safe strategy is to save and later restore the original graphics port upon entry to any routine that affects the screen. Listing 5-1 shows an example.

Listing 5-1 Saving and restoring the current graphics port

```

PROCEDURE DrawInPort(thePort: GrafPtr);
VAR
    origPort: GrafPtr;
BEGIN
    GetPort(origPort);

```

Drawing

```

SetPort(thePort);

{Do your drawing (erasing, etc.) here.}

SetPort(origPort);
END;

```

Notice that QuickDraw uses the `GrafPtr` data type to refer to graphics ports. For historical reasons, the `GrafPort` data structure is one of the few objects in the Macintosh system software that's referred to by a pointer rather than a handle.

Drawing Shapes

As you've seen, you can draw circles by calling `FrameOval`. The Venn Diagrammer application uses code like this to draw the outlines of the five circles:

```

FOR count := 1 TO 5 DO
    FrameOval(gGeometry^^.circleRects[count]);

```

The rectangles defining the circles are stored in an array of rectangles that is one of the fields of an application-defined data structure of type `MyGeometryRec`. Venn Diagrammer allocates just one of these records when the application first starts up. The global variable `gGeometry` is a handle to that record.

```

VAR
    gGeometry: MyGeometryHnd;           {handle to a geometry record}

```

Listing 5-2 shows part of the structure of this record.

Listing 5-2 The structure of a record describing a document window's geometry

```

TYPE MyGeometryRec =
    RECORD
        circleRects:  ARRAY[1..5] OF Rect;           {squares for the 5 circles}
        circleRgns:   ARRAY[1..5] OF RgnHandle;       {regions for the 5 circles}
        premiseRgns:  ARRAY[1..8] OF RgnHandle;       {regions for premises}
        concRgns:     ARRAY[1..4] OF RgnHandle;       {regions for conclusion}
        {other fields omitted}
    END;
MyGeometryPtr = ^MyGeometryRec;
MyGeometryHnd = ^MyGeometryPtr;

```

Drawing

This record contains all the information needed to perform graphics operations on the Venn diagram in a document window. The fields are initialized at application launch time by the application-defined routine `DoInitGeometry`, shown in Listing 5-3.

Listing 5-3 Initializing the geometry record

```
PROCEDURE DoInitGeometry;
BEGIN
    {Allocate the memory needed to hold the diagram's geometry.}
    gGeometry := MyGeometryHnd(NewHandleClear(sizeof(MyGeometryRec)));

    IF gGeometry = NIL THEN                                {make sure we have the memory}
        DoBadError(eNotEnoughMemory);                    {see Listing 9-5 on page 178}

    {Set up the rectangles that define the circles.}
    FOR count := 1 TO 5 DO
        gGeometry^.circleRects[count] := MyGetIndCircleRect(count);

    {Set up the regions that the circles define.}
    DoSetupCircleRegions;

    {Set up the overlapping regions within the circles.}
    DoSetupOverlapRegions;
END;
```

The `DoInitGeometry` procedure allocates a geometry record and calls other application-defined routines to initialize the fields of that record. First, it calls `MyGetIndCircleRect` to determine the rectangle bounding each of the five circles.

Note

The `MyGetIndCircleRect` function is not defined in this book. You could define such a function in many ways. You could determine in advance where in the window the five rectangles should be and then hard-code that information in constants. Alternatively, you could calculate desirable positions dynamically at run time. The Venn Diagrammer application uses the first method, for speed. ♦

Then `DoInitGeometry` calls two other application-defined routines to set up a number of regions in the window. The first, `DoSetupCircleRegions`, defined in Listing 5-4, creates regions corresponding to the area inside each of the five circles. These regions are used in turn by the `DoSetupOverlapRegions` procedure to calculate the regions of intersection.

Drawing

Listing 5-4 Defining circular regions

```

PROCEDURE DoSetupCircleRegions;
VAR
    count: Integer;
BEGIN
    FOR count := 1 TO 5 DO
        BEGIN
            gGeometry^.circleRgns[count] := NewRgn;
            OpenRgn;
            FrameOval(gGeometry^.circleRects[count]);
            CloseRgn(gGeometry^.circleRgns[count]);
        END;
    END;
END;

```

You create a new region by calling the `NewRgn` function, which allocates storage in your application heap for a structure of type `Region` and returns a handle (of type `RgnHandle`) to that region. The newly created region is empty. To add to the region, you call the `OpenRgn` procedure and then draw the outline of the area you want enclosed by the region. As you can see, `DoSetupCircleRegions` indicates the desired area by calling the `FrameOval` procedure on a circle's defining rectangle. When you're done drawing that outline, you call the `CloseRgn` procedure, passing it a handle to the region to close.

If you simply want to create a region that's empty, you can call `NewRgn`, `OpenRgn`, and `CloseRgn` without doing any drawing.

```

myRegion := NewRgn;           {create an empty region}
OpenRgn;
CloseRgn(myRegion);

```

The `DoSetupOverlapRegions` procedure, defined in Listing 5-5, uses the circular regions defined by `DoSetupCircleRegions` to define the regions corresponding to the areas defined by the overlapping circles.

Listing 5-5 Defining noncircular regions

```

PROCEDURE DoSetupOverlapRegions;
VAR
    myRegion: RgnHandle;      {a scratch region}
    count: Integer;
BEGIN
    FOR count := 1 TO 8 DO    {create new, empty regions}
        BEGIN
            gGeometry^.premiseRgns[count] := NewRgn;

```

Drawing

```

        OpenRgn;
        CloseRgn(gGeometry^.premiseRgns[count]);
    END;

myRegion := NewRgn;           {create a scratch region}
OpenRgn;
CloseRgn(myRegion);

{Calculate the overlap regions in the premises diagram.}
HLock(Handle(gGeometry));     {lock the handle}
WITH gGeometry^^ DO
    BEGIN
        DiffRgn(circleRgns[1], circleRgns[2], myRegion);
        DiffRgn(myRegion, circleRgns[3], premiseRgns[1]);

        SectRgn(circleRgns[1], circleRgns[2], myRegion);
        DiffRgn(myRegion, circleRgns[3], premiseRgns[2]);

        DiffRgn(circleRgns[2], circleRgns[1], myRegion);
        DiffRgn(myRegion, circleRgns[3], premiseRgns[3]);

        SectRgn(circleRgns[1], circleRgns[3], myRegion);
        DiffRgn(myRegion, circleRgns[2], premiseRgns[4]);

        SectRgn(circleRgns[1], circleRgns[2], myRegion);
        SectRgn(myRegion, circleRgns[3], premiseRgns[5]);

        SectRgn(circleRgns[2], circleRgns[3], myRegion);
        DiffRgn(myRegion, circleRgns[1], premiseRgns[6]);

        DiffRgn(circleRgns[3], circleRgns[1], myRegion);
        DiffRgn(myRegion, circleRgns[2], premiseRgns[7]);
    END;

    HUnlock(Handle(gGeometry)); {unlock the handle}
    DisposeRgn(myRegion);      {dispose scratch region}
END;

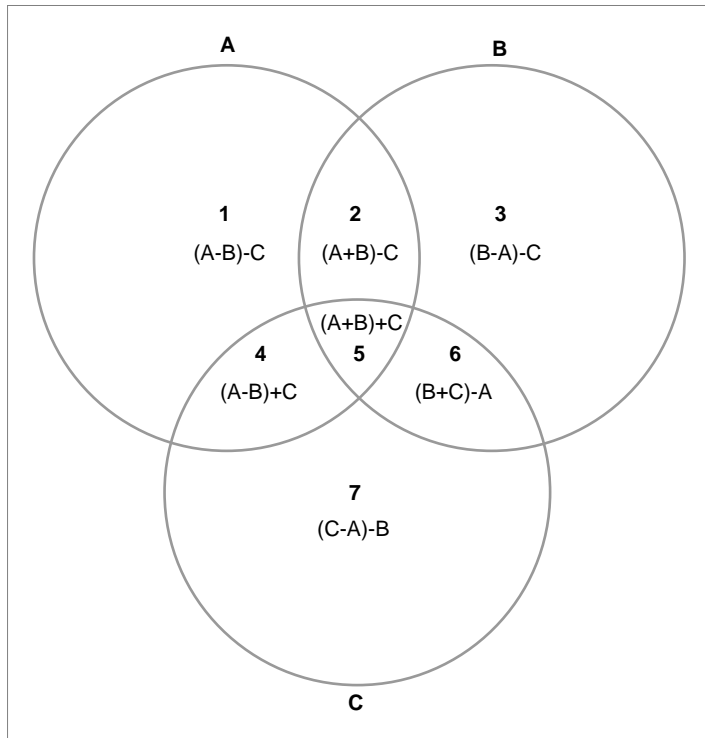
```

The `DoSetupOverlapRegions` procedure is remarkably straightforward. It initializes the regions in the premises diagram and also creates a temporary scratch region. Then it calculates the seven regions of overlap in that diagram by calling `SectRgn` and `DiffRgn` on the circular regions defined in Listing 5-4. The `SectRgn` procedure takes the intersection of two regions and places it into a third region. The `DiffRgn` procedure takes the portion of the first region that is outside the second region and places it into the

Drawing

third region. Figure 5-7 shows how the overlap regions are defined by taking intersections and unions of the three circles.

Figure 5-7 Calculating the overlap regions of a Venn diagram

**Note**

The definition of `DoSetupOverlapRegions` given in Listing 5-5 is not complete. It omits calculations of the conclusion regions and of the fields omitted from the `MyGeometryRec` data structure defined in Listing 5-2. ♦

Now that the Venn Diagrammer application has defined the various regions in the Venn diagram, it's easy to draw in those regions. For instance, to shade the very center of the diagram, you could call the `FillRgn` procedure, as follows:

```
FillRgn(gGeometry^^.premiseRgns[5], gEmptyPats[gEmptyIndex]^^);
```

This fills the specified region with the current emptiness pattern.

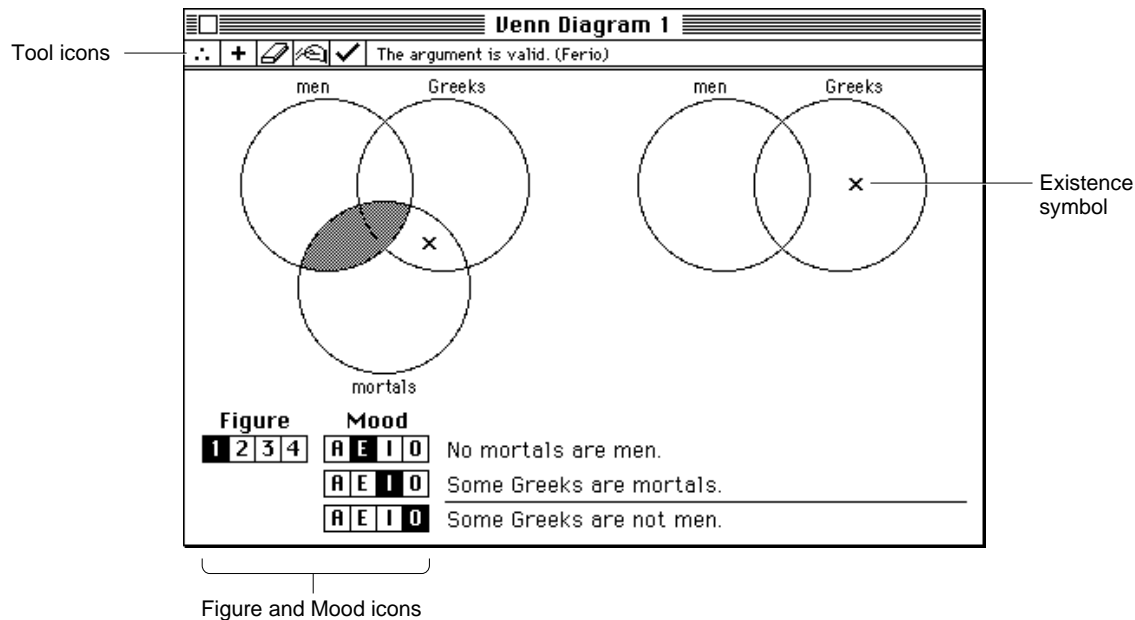
Drawing Bit Images

The Venn Diagrammer application uses bit images to draw several parts of a document window, including

- the tool symbols at the top of a document window
- the figure and mood symbols at the bottom of a window
- the existence symbol within the Venn diagram itself

Figure 5-8 shows the location of these items.

Figure 5-8 Bit images in a document window



The standard way to draw a bit image is to read into memory the appropriate bit data and then call the `CopyBits` routine to move the data into the desired position in the destination window. The Venn Diagrammer application stores the bit data in resources of type 'ICON'. Then it calls its own application-defined routine `DoPlotIcon` to move the appropriate portion of the icon into a document window. Notice that none of the bit images in a document window is actually as large as an icon (which is 32 pixels by 32 pixels). Venn Diagrammer uses this strategy because ResEdit provides a simple way to create and edit 'ICON' resources.

Drawing

When Venn Diagrammer starts up, it reads the necessary icon resources into memory using the code in Listing 5-6.

Listing 5-6 Reading 'ICON' resources into memory

```
{Get handles to tool icons.}
FOR count := 1 TO kNumTools DO
    gToolsIcons[count] := GetResource('ICON', kToolsIconStart + (count - 1));

{Get handles to available existence-indicating icons.}
FOR count := 1 TO 4 DO
    gExistIcons[count] := GetResource('ICON', kExistID + (count - 1));

{Get handles to mood icons.}
FOR count := 1 TO 4 DO
    gMoodIcons[count] := GetResource('ICON', kMoodIconStart + (count - 1));

{Get handles to figure icons.}
FOR count := 1 TO 4 DO
    gFigureIcons[count] := GetResource('ICON', kFigIconStart + (count - 1));
```

As you can see, the icons in each group are given contiguous resource IDs in the resource file. The handles to each icon are stored in the appropriate array, accessed by global variables.

IMPORTANT

As always, you should make certain that none of the returned handles has the value NIL. For brevity, this check is not shown in Listing 5-6. ▲

To draw the tools area of a window, for example, Venn Diagrammer uses the code shown in Listing 5-7.

Listing 5-7 Drawing the tools area of a document window

```
{Redraw the tool 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;
```

Drawing

This code fragment calls the application-defined routine `DoPlotIcon` to draw the appropriate portion of the icon in the specified rectangle. The `DoPlotIcon` procedure is defined in Listing 5-8.

Listing 5-8 Drawing a portion of an icon

```
PROCEDURE DoPlotIcon (myRect: Rect; myIcon: Handle; myWindow: WindowPtr;
                     myMode: Integer);

    VAR
        myBitMap:      BitMap;
BEGIN
    myBitMap.baseAddr := myIcon^;
    myBitMap.rowBytes := 4;
    myBitMap.bounds := myRect;
    CopyBits(myBitMap, myWindow^.portBits, myRect, myRect, myMode, NIL);
END;
```

The `DoPlotIcon` procedure plots a portion of an icon by defining a bitmap that includes the desired portion of the icon. (The desired portion of the icon is specified by the `myRect` parameter.) Then `DoPlotIcon` calls the QuickDraw routine `CopyBits` to copy the appropriate bits from their location in memory to the desired location in the specified window.

The `CopyBits` procedure transfers a bit image between two existing bit maps. In this case, the two bitmaps are the bitmapped portion of the icon and the bits in the destination window (which are specified by the `portBits` field of the window's graphics port; see Listing 6-1 on page 112 for details). The `myRect` parameter specifies the rectangle to copy; it's passed to `DoPlotIcon` from the calling routine so that `DoPlotIcon` can be used to plot different parts of the source icon. Finally, `DoPlotIcon` is passed a *transfer mode*, which indicates how the bits are to be drawn in the existing bit image of the destination rectangle. The constant `srcCopy` is passed in Listing 5-7 to indicate that the source bitmap is to overwrite the destination bitmap.

Drawing Text

In addition to the many routines it provides for defining and drawing both simple and complex graphic elements, QuickDraw also provides support for drawing text. You can use QuickDraw to draw characters, words, or other textual elements at any desired size and in any available font. It might seem odd that QuickDraw handles these operations, until you realize that text, like graphics, permeates the Macintosh user interface. Windows, menus, and some controls (for instance, buttons) have titles, which are essential to the user's understanding and manipulation of the application. As a result, it makes sense to treat text fundamentally as a graphic object and to assign basic

Drawing

text-drawing responsibilities to QuickDraw, which manages all graphics within the Macintosh system software.

Although QuickDraw is ultimately responsible for drawing text on the screen, you might need to use other Toolbox managers for other text-handling needs. For example, if you want the user to be able to input and edit some small amount of text, you can use TextEdit. TextEdit provides basic text-editing capabilities, such as cutting, copying, pasting, and entering words and characters. TextEdit calls QuickDraw to display the editable text. Similarly, if your application allows the user to display text in a variety of fonts, you might need to use the Font Manager. The Font Manager supports QuickDraw by providing the character bitmaps it needs to draw text in a specified font, size, and style. For a complete description of TextEdit and the Font Manager, see *Inside Macintosh: Text*.

The Venn Diagrammer application has very minimal text-handling requirements. It does not support any text entry or editing by the user. Instead, it obtains all the text it needs from resources stored in its resource fork. As a result, the Venn Diagrammer application can use basic QuickDraw text-drawing routines to display its text. For example, the Venn Diagrammer application draws the message in a window's status area by calling the application-defined routine `DoStatusMesg`, defined in Listing 5-9.

Listing 5-9 Retrieving a status message from a resource

```
PROCEDURE DoStatusMesg (myWindow: WindowPtr; myMessageID: Integer);
    VAR
        myText:      Str255;
BEGIN
    GetIndString(myText, rVennD, myMessageID);
    DoStatusText(myWindow, myText);
END;
```

As you can see, the `DoStatusMesg` routine takes two parameters, a window pointer specifying the window whose status area is to be filled in and an integer specifying the index into an 'STR#' resource. Then `DoStatusMesg` retrieves the appropriate message text and calls the application-defined procedure `DoStatusText` to print the message in the window.

Venn Diagrammer calls `DoStatusMesg` whenever it needs to display a message in the status area. For instance, when the user wants to determine if a syllogism is valid or not, Venn Diagrammer checks the syllogism's validity and then executes the code in Listing 5-10.

Drawing

Listing 5-10 Informing the user of an argument's validity or invalidity

```

IF valid THEN
    BEGIN
        IF gShowNames THEN          {show names of valid syllogisms?}
            BEGIN
                GetIndString(myMesg, rVennD, eArgIsValid);
                DoGetName(myWindow, myName);
                myMesg := concat(myMesg, ' (' , myName, ')');
                DoStatusText(myWindow, myMesg);
            END
        ELSE
            DoStatusMesg(myWindow, eArgIsValid);
        END
    ELSE
        DoStatusMesg(myWindow, eArgNotValid);
    END

```

This code fragment illustrates why the Venn Diagrammer application defines two different routines, `DoStatusMesg` and `DoStatusText`. The first, `DoStatusMesg`, retrieves the desired message text from a resource and calls the second, `DoStatusText`, to display it on the screen. The application also calls `DoStatusText` at other times, for instance, when it needs to add something to the resource-based message string. In the example shown in Listing 5-10, the application needs to get the name of the valid syllogism, if the user has indicated that this should be done.

The `DoStatusText` procedure is defined in Listing 5-11. Its job is to display the text passed as a parameter in the status area of the specified window.

Listing 5-11 Displaying a status message

```

PROCEDURE DoStatusText (myWindow: WindowPtr; myText: Str255);
    VAR
        myRect:      Rect;
        origSize:    Integer;
        origFont:    Integer;
        myHandle:    MyDocRecHnd;
    CONST
        kSlop = 4;
        kSize = 9;
        kFont = applFont;
    BEGIN
        IF myWindow <> NIL THEN
            BEGIN
                SetPort(myWindow);
                origSize := myWindow^.txSize;      {remember original size and font}

```

Drawing

```

origFont := myWindow^.txFont;
TextSize(kSize);                      {set desired size and font}
TextFont(kFont);

SetRect(myRect, kToolWd * kNumTools, 0,
        myWindow^.portRect.right, kToolHt);
EraseRect(myRect);
IF length(myText) > 0 THEN
    BEGIN
        MoveTo(myRect.left + kSlop, myRect.bottom - kSlop);
        DrawString(myText);
    END;

TextSize(origSize);                  {restore original size and font}
TextFont(origFont);

{Remember the last message printed in this window.}
myHandle := MyDocRecHnd(GetWRefCon(myWindow));
myHandle^.statusText := myText;
END;
END;

```

The `DoStatusText` procedure first remembers the graphics port's existing font and size, so that it can change and then later restore those values. Then `DoStatusText` sets the desired font and size of the status message by calling the QuickDraw routines `TextFont` and `TextSize`. You should always use these routines—instead of changing the fields of the `grafPort` record—whenever you want to change a graphics port's font and size.

IMPORTANT

Although you should never *change* the fields of a graphics port directly, you sometimes need to *read* those fields directly. In Listing 5-11, the original font and size are determined by reading the appropriate fields (`txFont` and `txSize`) of the graphics port record. This is necessary because QuickDraw doesn't provide routines to read that information from a graphics port record. ▲

Once it's set the desired font and size, the `DoStatusText` procedure calls `SetRect` to define the rectangle into which the text is to be drawn. Then, `DoStatusText` erases that rectangle by calling `EraseRect`. If the string to be displayed consists of at least one character, `DoStatusText` moves to the appropriate spot in the status area and calls the QuickDraw routine `DrawString`, which draws the specified string at the current drawing location in the window.

Drawing

Finally, `DoStatusText` restores the graphics port's original font and size, and then copies the string just drawn into the `statusText` field of the window's document record. The Venn Diagrammer application needs to remember each window's latest status message so that it can redraw the message whenever necessary (for example, if the message is covered up by another window and then later revealed).

Venn Diagrammer uses similar techniques for all other text drawing it requires. Remember that this application supports only static text (that is, text that cannot be edited) stored in the application's resource fork. To allow the user to enter and edit some text, you need to use more powerful text-handling tools. See *Inside Macintosh: Text* for information about using system software services like the Font Manager and TextEdit to handle editable text. See *Inside Macintosh: Files* for information on storing text and other data in files. Finally, see the chapter "Dialog Manager" in *Inside Macintosh: Macintosh Toolbox Essentials* for information on handling text entry and editing in a dialog box.

