2
Programming in the Simple Raster Graphics Package (SRGP)

Andries van Dam
and David F. Sklar

In Chapter 1, we saw that vector and raster displays are two substantially different hardware technologies for creating images on the screen. Raster displays are now the dominant hardware technology, because they support several features that are essential to the majority of modern applications. First, raster displays can fill areas with a uniform color or a repeated pattern in two or more colors; vector displays can, at best, only simulate filled areas with closely spaced sequences of parallel vectors. Second, raster displays store images in a way that allows manipulation at a fine level: individual pixels can be read or written, and arbitrary portions of the image can be copied or moved.

The first graphics package we discuss, SRGP (Simple Raster Graphics Package), is a device-independent graphics package that exploits raster capabilities. SRGP's repertoire of primitives (lines, rectangles, circles and ellipses, and text strings) is similar to that of the popular Macintosh QuickDraw raster package and that of the Xlib package of the X Window System. Its interaction-handling features, on the other hand, are a subset of those of SPHIGS, the higher-level graphics package for displaying 3D primitives (covered in Chapter 7). SPHIGS (Simple PHIGS) is a simplified dialect of the standard PHIGS graphics package (Programmer's Hierarchical Interactive Graphics System) designed for both raster and vector hardware. Although SRGP and SPHIGS were written specifically for this text, they are also very much in the spirit of mainstream graphics packages, and most of what you will learn here is immediately applicable to commercial packages. In this book, we introduce both packages; for a more complete description, you should consult the reference manuals distributed with the software packages.

We start our discussion of SRGP by examining the operations that applications perform in order to draw on the screen: the specification of primitives and of the attributes that affect

their image. (Since graphics printers display information essentially as raster displays do, we need not concern ourselves with them until we look more closely at hardware in Chapter 4.) Next we learn how to make applications interactive using SRGP's input procedures. Then we cover the utility of pixel manipulation, available only in raster displays. We conclude by discussing some limitations of integer raster graphics packages such as SRGP.

Although our discussion of SRGP assumes that it controls the entire screen, the package has been designed to run in window environments (see Chapter 10), in which case it controls the interior of a window as though it were a virtual screen. The application programmer therefore does not need to be concerned about the details of running under control of a window manager.

2.1 DRAWING WITH SRGP

2.1.1 Specification of Graphics Primitives

Drawing in integer raster graphics packages such as SRGP is like plotting graphs on graph paper with a very fine grid. The grid varies from 80 to 120 points per inch on conventional displays to 300 or more on high-resolution displays. The higher the resolution, the better the appearance of fine detail. Figure 2.1 shows a display screen (or the surface of a printer's paper or film) ruled in SRGP's integer Cartesian coordinate system. Note that pixels in SRGP lie at the intersection of grid lines.

The origin (0, 0) is at the bottom left of the screen; positive x increases toward the right and positive y increases toward the top. The pixel at the upper-right corner is (width–1, height–1), where width and height are the device-dependent dimensions of the screen.

On graph paper, we can draw a continuous line between two points located anywhere on the paper; on raster displays, however, we can draw lines only between grid points, and the line must be approximated by intensifying the grid-point pixels lying on it or nearest to it. Similarly, solid figures such as filled polygons or circles are created by intensifying the pixels in their interiors and on their boundaries. Since specifying each pixel of a line or closed figure would be far too onerous, graphics packages let the programmer specify primitives such as lines and polygons via their vertices; the package then fills in the details using scan-conversion algorithms, discussed in Chapter 3.

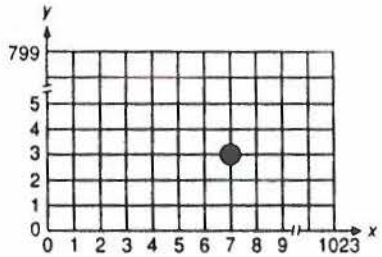

Fig. 2.1 Cartesian coordinate system of a screen 1024 pixels wide by 800 pixels high. Pixel (7, 3) is shown.

SRGP supports a basic collection of primitives: lines, polygons, circles and ellipses, and text.[1] To specify a primitive, the application sends the coordinates defining the primitive's shape to the appropriate SRGP primitive-generator procedure. It is legal for a specified point to lie outside the screen's bounded rectangular area; of course, only those portions of a primitive that lie inside the screen bounds will be visible.

Lines and polylines. The following SRGP procedure draws a line from ($x1$, $y1$) to ($x2$, $y2$):

```
procedure SRGP_lineCoord (x1, y1, x2, y2 : integer);[2]
```

Thus, to plot a line from (0, 0) to (100, 300), we simply call

```
SRGP_lineCoord (0, 0, 100, 300);
```

Because it is often more natural to think in terms of endpoints rather than of individual x and y coordinates, SRGP provides an alternate line-drawing procedure:

```
procedure SRGP_line (pt1, pt2 : point);
```

Here "point" is a defined type, a record of two integers holding the point's x and y values:

```
type
  point = record
    x, y : integer
  end;
```

A sequence of lines connecting successive vertices is called a *polyline*. Although polylines can be created by repeated calls to the line-drawing procedures, SRGP includes them as a special case. There are two polyline procedures, analogous to the coordinate and point forms of the line-drawing procedures. These take arrays as parameters:

```
procedure SRGP_polyLineCoord (
    vertexCount : integer; xArray, yArray : vertexCoordinateList);
procedure SRGP_polyLine (vertexCount : integer; vertices : vertexList);
```

where "vertexCoordinateList" and "vertexList" are types defined by the SRGP package—arrays of integers and points, respectively.

The first parameter in both of these polyline calls tells SRGP how many vertices to expect. In the first call, the second and third parameters are integer arrays of paired x and y values, and the polyline is drawn from vertex (*xArray*[0], *yArray*[0]), to vertex (*xArray*[1], *yArray*[1]), to vertex (*xArray*[2], *yArray*[2]), and so on. This form is convenient, for instance, when plotting data on a standard set of axes, where *xArray* is a predetermined set

---

[1] Specialized procedures that draw a single pixel or an array of pixels are described in the SRGP reference manual.

[2] We use Pascal with the following typesetting conventions. Pascal keywords and built-in types are in boldface and user-defined types are in normal face. Symbolic constants are in uppercase type, and variables are italicized. Comments are in braces, and pseudocode is italicized. For brevity, declarations of constants and variables are omitted when obvious.

of values of the independent variable and *yArray* is the set of data being computed or input by the user. As an example, let us plot the output of an economic analysis program that computes month-by-month trade figures and stores them in the 12-entry integer data array *balanceOfTrade*. We will start our plot at (200, 200). To be able to see the differences between successive points, we will graph them 10 pixels apart on the x axis. Thus, we will create an integer array, *months*, to represent the 12 months, and will set the entries to the desired x values, 200, 210, . . ., 310. Similarly, we must increment each value in the data array by 200 to put the 12 y coordinates in the right place. Then, the graph in Fig. 2.2 is plotted with the following code:

```
{Plot the axes}
SRGP_lineCoord (50, 200, 350, 200);
SRGP_lineCoord (200, 50, 200, 350);

{Plot the data}
SRGP_polyLineCoord (12, months, balanceOfTrade);
```

We can use the second polyline form to draw shapes by specifying pairs of x and y values together as points, passing an array of such points to SRGP. We create the bowtie in Fig. 2.3 by calling

```
SRGP_polyLine (7, bowtieArray);
```

The table in Fig. 2.3 shows how *bowtieArray* was defined.

Markers and polymarkers. It is often convenient to place *markers* (e.g., dots, asterisks, or circles) at the data points on graphs. SRGP therefore offers companions to the line and polyline procedures. The following procedures will create a marker symbol centered at (x, y):

```
procedure SRGP_markerCoord (x, y : integer);
procedure SRGP_marker (pt : point);
```

The marker's style and size can be changed as well, as explained in Section 2.1.2. To create

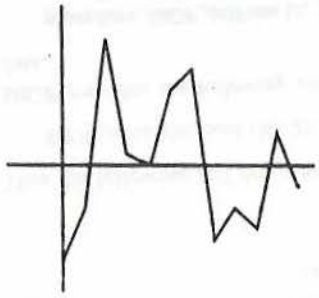

Fig. 2.2 Graphing a data array.

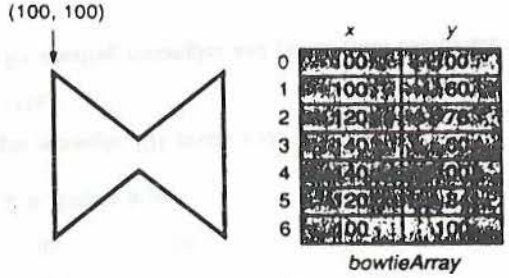

Fig. 2.3 Drawing a polyline.

a sequence of identical markers at a set of points, we call either of

```
procedure SRGP_polyMarkerCoord (
    vertexCount : integer; xArray, yArray : vertexCoordinateList);
procedure SRGP_polyMarker (vertexCount : integer; vertices : vertexList);
```

Thus, the following additional call will add markers to the graph of Fig. 2.2 to produce Fig. 2.4.

```
SRGP_polyMarkerCoord (12, months, balanceOfTrade);
```

Polygons and rectangles. To draw an outline polygon, we can either specify a polyline that closes on itself by making the first and last vertices identical (as we did to draw the bowtie in Fig. 2.3), or we can use the following specialized SRGP call:

```
procedure SRGP_polygon (vertexCount : integer; vertices : vertexList);
```

This call automatically closes the figure by drawing a line from the last vertex to the first. To draw the bowtie in Fig. 2.3 as a polygon, we use the following call, where *bowtieArray* is now an array of only six points:

```
SRGP_polygon (6, bowtieArray);
```

Any rectangle can be specified as a polygon having four vertices, but an upright rectangle (one whose edges are parallel to the screen's edges) can also be specified with the SRGP "rectangle" primitive using only two vertices (the lower-left and the upper-right corners).

```
procedure SRGP_rectangleCoord (leftX, bottomY, rightX, topY : integer);
procedure SRGP_rectanglePt (bottomLeft, topRight : point);
procedure SRGP_rectangle (rect : rectangle);
```

The "rectangle" record stores the bottom-left and top-right corners:

```
type
  rectangle = record
    bottomLeft, topRight : point
  end;
```

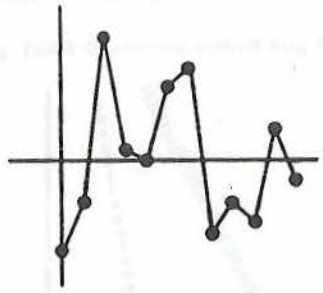

Fig. 2.4 Graphing the data array using markers.

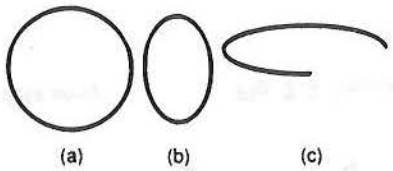

Fig. 2.5 Ellipse arcs.

Thus the following call draws an upright rectangle 101 pixels wide and 151 pixels high:

```
SRGP_rectangleCoord (50, 25, 150, 175);
```

SRGP provides the following utilities for creating rectangles and points from coordinate data.

```
procedure SRGP_defPoint (x, y : integer; var pt : point);
procedure SRGP_defRectangle (
    leftX, bottomY, rightX, topY : integer; var rect : rectangle);
```

Our example rectangle could thus have been drawn by

```
SRGP_defRectangle (50, 25, 150, 175, rect);
SRGP_rectangle (rect);
```

Circles and ellipses. Figure 2.5 shows circular and elliptical arcs drawn by SRGP. Since circles are a special case of ellipses, we use the term *ellipse arc* for all these forms, whether circular or elliptical, closed or partial arcs. SRGP can draw only standard ellipses, those whose major and minor axes are parallel to the coordinate axes.

Although there are many mathematically equivalent methods for specifying ellipse arcs, it is convenient for the programmer to specify arcs via the upright rectangles in which they are inscribed (see Fig. 2.6); these upright rectangles are called *bounding boxes* or *extents*.

The width and height of the extent determine the shape of the ellipse. Whether or not the arc is closed depends on a pair of angles that specify where the arc starts and ends. For convenience, each angle is measured in *rectangular degrees* that run counterclockwise, with 0° corresponding to the positive portion of the x axis, 90° to the positive portion of the y

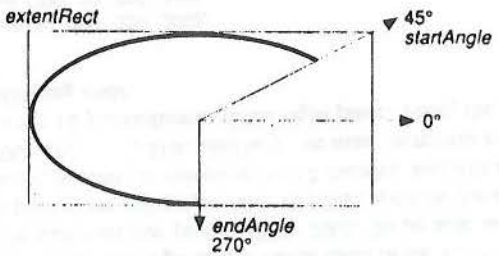

Fig. 2.6 Specifying ellipse arcs.

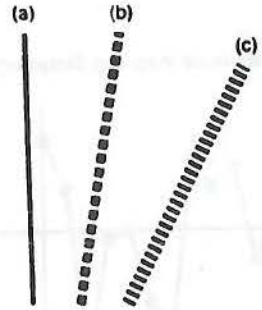

Fig. 2.7 Lines of various widths and styles.

axis, and 45° to the "diagonal" extending from the origin to the top-right corner of the rectangle. Clearly, only if the extent is a square are rectangular degrees equivalent to circular degrees.

The general ellipse procedure is

```
procedure SRGP_ellipseArc (extentRect : rectangle; startAngle, endAngle : real);
```

2.1.2 Attributes

Line style and line width. The appearance of a primitive can be controlled by specification of its *attributes*.[3] The SRGP attributes that apply to lines, polylines, polygons, rectangles, and ellipse arcs are *line style*, *line width*, *color*, and *pen style*.

Attributes are set *modally*; that is, they are global state variables that retain their values until they are changed explicitly. Primitives are drawn with the attributes in effect at the time the primitives are specified; therefore, changing an attribute's value in no way affects previously created primitives—it affects only those that are specified after the change in attribute value. Modal attributes are convenient because they spare programmers from having to specify a long parameter list of attributes for each primitive, since there may be dozens of different attributes in a production system.

Line style and line width are set by calls to

```
procedure SRGP_setLineStyle (value : CONTINUOUS / DASHED / DOTTED / ... );[4]
procedure SRGP_setLineWidth (value : integer);
```

The width of a line is measured in screen units—that is, in pixels. Each attribute has a default: line style is CONTINUOUS, and width is 1. Figure 2.7 shows lines in a variety of widths and styles; the code that generated the figure is shown in Fig. 2.8.

---

[3] The descriptions here of SRGP's attributes often lack fine detail, particularly on interactions between different attributes. The detail is omitted because the exact effect of an attribute is a function of its implementation, and, for performance reasons, different implementations are used on different systems; for these details, consult the implementation-specific reference manuals.

[4] Here and in the following text, we use a shorthand notation. In SRGP, these symbolic constants are actually values of an enumerated data type "lineStyle."

```
SRGP_setLineWidth (5);
SRGP_lineCoord (55, 5, 55, 295);        [Line a]

SRGP_setLineStyle (DASHED);
SRGP_setLineWidth (10);
SRGP_lineCoord (105, 5, 155, 295);      [Line b]

SRGP_setLineWidth (15);
SRGP_setLineStyle (DOTTED);
SRGP_lineCoord (155, 5, 285, 255);      [Line c]
```

Fig. 2.8 Code used to generate Fig. 2.7.

We can think of the line style as a bit mask used to write pixels selectively as the primitive is scan-converted by SRGP. A zero in the mask indicates that this pixel should not be written and thus preserves the original value of this pixel in the frame buffer. One can think of this pixel of the line as transparent, in that it lets the pixel "underneath" show through. CONTINUOUS thus corresponds to the string of all 1s, and DASHED to the string 1111001111001111 . . ., the dash being twice as long as the transparent interdash segments.

Each attribute has a default; for example, the default for line style is CONTINUOUS, that for line width is 1, and so on. In the early code examples, we did not set the line style for the first line we drew; thus, we made use of the line-style default. In practice, however, making assumptions about the current state of attributes is not safe, and in the code examples that follow we set attributes explicitly in each procedure, so as to make the procedures modular and thus to facilitate debugging and maintenance. In Section 2.1.4, we see that it is even safer for the programmer to save and restore attributes explicitly for each procedure.

Attributes that can be set for the marker primitive are

procedure SRGP_setMarkerSize (*value* : **integer**);
procedure SRGP_setMarkerStyle (*value* : MARKER_CIRCLE / MARKER_SQUARE / ...);

Marker size specifies the length in pixels of the sides of the square extent of each marker. The complete set of marker styles is presented in the reference manual; the circle style is the default shown in Fig. 2.4.

Color. Each of the attributes presented so far affects only some of the SRGP primitives, but the integer-valued color attribute affects all primitives. Obviously, the color attribute's meaning is heavily dependent on the underlying hardware; the two color values found on every system are 0 and 1. On bilevel systems, these colors' appearances are easy to predict—color-1 pixels are black and color-0 pixels are white for black-on-white devices, green is 1 and black is 0 for green-on-black devices, and so on.

The integer color attribute does not specify a color directly; rather, it is an index into SRGP's *color table*, each entry of which defines a color or gray-scale value in a manner that the SRGP programmer does not need to know about. There are 2^d entries in the color table, where d is the *depth* (number of bits stored for each pixel) of the frame buffer. On bilevel implementations, the color table is hardwired; on most color implementations, however,

SRGP allows the application to modify the table. Some of the many uses for the indirectness provided by color tables are explored in Chapters 4, 17, and 21.

There are two methods that applications can use to specify colors. An application for which machine independence is important should use the integers 0 and 1 directly; it will then run on all bilevel and color displays. If the application assumes color support or is written for a particular display device, then it can use the implementation-dependent *color names* supported by SRGP. These names are symbolic constants that show where certain standard colors have been placed within the default color table for that display device. For instance, a black-on-white implementation provides the two color names COLOR_BLACK (1) and COLOR_WHITE (0); we use these two values in the sample code fragments in this chapter. Note that color names are not useful to applications that modify the color table.

We select a color by calling

procedure SRGP_setColor (*colorIndex* : **integer**);

2.1.3 Filled Primitives and Their Attributes

Primitives that enclose areas (the so-called *area-defining* primitives) can be drawn in two ways: *outlined* or *filled*. The procedures described in the previous section generate the former style: closed outlines with unfilled interiors. SRGP's filled versions of area-defining primitives draw the interior pixels with no outline. Figure 2.9 shows SRGP's repertoire of filled primitives, including the filled ellipse arc, or *pie slice*.

Note that SRGP does not draw a contrasting outline, such as a 1-pixel-thick solid boundary, around the interior; applications wanting such an outline must draw it explicitly. There is also a subtle issue of whether pixels on the border of an area-defining primitive should actually be drawn or whether only pixels that lie strictly in the interior should. This problem is discussed in detail in Sections 3.5 and 3.6.

To generate a filled polygon, we use SRGP_fillPolygon or SRGP_fillPolygonCoord, with the same parameter lists used in the unfilled versions of these calls. We define the other area-filling primitives in the same way, by prefixing "fill" to their names. Since polygons

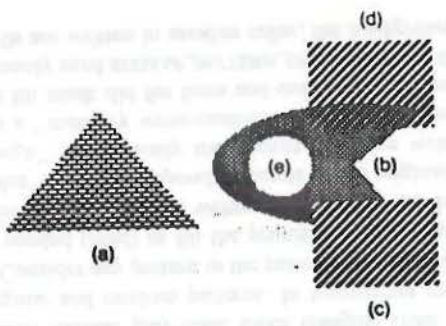

Fig. 2.9 Filled primitives. (a-c) Bitmap pattern opaque. (d) Bitmap pattern transparent. (e) Solid.

may be concave or even self-intersecting, we need a rule for specifying what regions are interior and thus should be filled, and what regions are exterior. SRGP polygons follow the *odd-parity* rule. To determine whether a region lies inside or outside a given polygon, choose as a test point any point inside the particular region. Next, choose a ray that starts at the test point and extends infinitely in any direction, and that does not pass through any vertices. If this ray intersects the polygon outline an odd number of times, the region is considered to be interior (see Fig. 2.10).

SRGP does not actually perform this test for each pixel while drawing; rather, it uses the optimized polygon scan-conversion techniques described in Chapter 3, in which the odd-parity rule is efficiently applied to an entire row of adjacent pixels that lie either inside or outside. Also, the odd-parity ray-intersection test is used in a process called *pick correlation* to determine the object a user is selecting with the cursor, as described in Chapter 7.

Fill style and fill pattern for areas. The fill-style attribute can be used to control the appearance of a filled primitive's interior in four different ways, using

```
procedure SRGP_setFillStyle (
    mode: SOLID / BITMAP_PATTERN_OPAQUE / BITMAP_PATTERN_TRANSPARENT /
          PIXMAP_PATTERN);
```

The first option, SOLID, produces a primitive uniformly filled with the current value of the color attribute (Fig. 2.9e, with color set to COLOR_WHITE). The second two options, BITMAP_PATTERN_OPAQUE and BITMAP_PATTERN_TRANSPARENT, fill primitives with a regular, nonsolid pattern, the former rewriting all pixels underneath in either the current color, or another color (Fig. 2.9c), the latter rewriting some pixels underneath the primitive in the current color, but letting others show through (Fig. 2.9d). The last option, PIXMAP_PATTERN, writes patterns containing an arbitrary number of colors, always in opaque mode.

Bitmap fill patterns are bitmap arrays of 1s and 0s chosen from a table of available

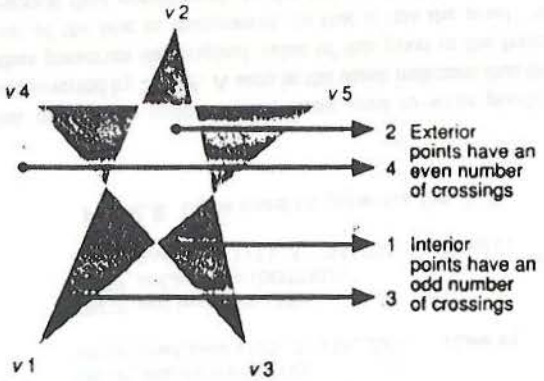

Fig. 2.10 Odd-parity rule for determining interior of a polygon.

patterns by specifying

```
procedure SRGP_setFillBitmapPattern (patternIndex : integer);
```

Each entry in the pattern table stores a unique pattern; the ones provided with SRGP, shown in the reference manual, include gray-scale tones (ranging from nearly black to nearly white) and various regular and random patterns. In transparent mode, these patterns are generated as follows. Consider any pattern in the pattern table as a small bitmap—say, 8 by 8—to be repeated as needed (*tiled*) to fill the primitive. On a bilevel system, the current color (in effect, the *foreground* color) is written where there are 1s in the pattern; where there are 0s—the "holes"—the corresponding pixels of the original image are not written, and thus "show through" the partially transparent primitive written on top. Thus, the bitmap pattern acts as a "memory write-enable mask" for patterns in transparent mode, much as the line-style bit mask did for lines and outline primitives.

In the more commonly used BITMAP_PATTERN_OPAQUE mode, the 1s are written in the current color, but the 0s are written in another color, the *background color*, previously set by

```
procedure SRGP_setBackgroundColor (colorIndex : integer);
```

On bilevel displays, each bitmap pattern in OPAQUE mode can generate only two distinctive fill patterns. For example, a bitmap pattern of mostly 1s can be used on a black-and-white display to generate a dark-gray fill pattern if the current color is set to black (and the background to white), and a light-gray fill pattern if the current color is set to white (and the background to black). On a color display, any combination of a foreground and a background color may be used for a variety of two-tone effects. A typical application on a bilevel display always sets the background color whenever it sets the foreground color, since opaque bitmap patterns are not visible if the two are equal; an application could create a SetColor procedure to set the background color automatically to contrast with the foreground whenever the foreground color is set explicitly.

Figure 2.9 was created by the code fragment shown in Fig. 2.11. The advantage of

```
SRGP_setFillStyle (BITMAP_PATTERN_OPAQUE);
SRGP_setFillBitmapPattern (BRICK_BIT_PATTERN);          {Brick pattern}
SRGP_fillPolygon (3, triangle_coords);                  {a}

SRGP_setFillBitmapPattern (MEDIUM_GRAY_BIT_PATTERN);    {50 percent gray}
SRGP_fillEllipseArc (ellipseArcRect, 60.0, 290.0);      {b}

SRGP_setFillBitmapPattern (DIAGONAL_BIT_PATTERN);
SRGP_fillRectangle (opaqueFilledRect);                  {c}

SRGP_setFillStyle (BITMAP_PATTERN_TRANSPARENT);
SRGP_fillRectangle (transparentFilledRect);             {d}

SRGP_setFillStyle (SOLID);
SRGP_setColor (COLOR_WHITE);
SRGP_fillEllipse (circleRect);                          {e}
```

Fig. 2.11 Code used to generate Fig. 2.9.

having two-tone bitmap patterns is that the colors are not specified explicitly, but rather are determined by the color attributes in effect, and thus can be generated in any color combination. The disadvantage, and the reason that SRGP also supports pixmap patterns, is that only two colors can be generated. Often, we would like to fill an area of a display with multiple colors, in an explicitly specified pattern. In the same way that a bitmap pattern is a small bitmap used to tile the primitive, a small pixmap can be used to tile the primitive, where the pixmap is a pattern array of color-table indices. Since each pixel is explicitly set in the pixmap, there is no concept of holes, and therefore there is no distinction between transparent and opaque filling modes. To fill an area with a color pattern, we select a fill style of PIXMAP_PATTERN and use the corresponding pixmap pattern-selection procedure:

```
procedure SRGP_setFillPixmapPattern (patternIndex : integer);
```

Since both bitmap and pixmap patterns generate pixels with color values that are indices into the current color table, the appearance of filled primitives changes if the programmer modifies the color-table entries. The SRGP reference manual discusses how to change or add to both the bitmap and pixmap pattern tables. Also, although SRGP provides default entries in the bitmap pattern table, it does not give a default pixmap pattern table, since there is an indefinite number of color pixmap patterns that might be useful.

Pen pattern for outlines. The advantages of patterning are not restricted to the use of this technique in area-defining primitives; patterning can also be used to affect the appearance of lines and outline primitives, via the *pen-style* attribute. Using the line-width, line-style, and pen-style attributes, it is possible, for example, to create a 5-pixel-thick, dot-dashed ellipse whose thick dashes are patterned. Examples of solid and dashed thick lines with various patterns in transparent and opaque mode and their interactions with previously drawn primitives are shown in Fig. 2.12; the code that generated the image is in Fig. 2.13. The use of a pen pattern for extremely narrow lines (1 or 2 pixels wide) is not recommended, because the pattern is not discernible in such cases.

The interaction between line style and pen style is simple: 0s in the line-style mask

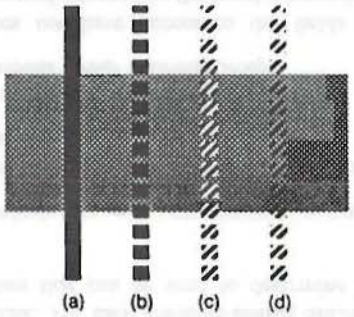

Fig. 2.12 Interaction between pen style and line style. (a) Continuous solid. (b) Dashed solid. (c) Dashed bitmap pattern opaque. (d) Dashed bitmap pattern transparent.


```
{We show only the drawing of the lines, not the background rectangle.
 We draw the lines in order from left to right.}

SRGP_setLineWidth (15);                    {Thick lines show the interaction better.}

SRGP_setLineStyle (CONTINUOUS);
SRGP_setPenStyle (SOLID);
SRGP_line (pta1, pta2);                    {a: Solid, continuous}

SRGP_setLineStyle (DASHED);
SRGP_line (ptb1, ptb2);                    {b: Solid, dashed}

SRGP_setPenBitmapPattern (DIAGONAL_BIT_PATTERN);
SRGP_setPenStyle (BITMAP_PATTERN_OPAQUE);
SRGP_line (ptc1, ptc2);                    {c: Dashed, bitmap pattern opaque}

SRGP_setPenStyle (BITMAP_PATTERN_TRANSPARENT);
SRGP_line (ptd1, ptd2);                    {d: Dashed, bitmap pattern transparent}
```

Fig. 2.13 Code used to generate Fig. 2.12.

fully protect the pixels on which they fall, so the pen.style influences only those pixels for which the line-style mask is 1.

Pen style is selected with the same four options and the same patterns as fill style. The same bitmap and pixmap pattern tables are also used, but separate indices are maintained so that resetting a pen style's pattern index will not affect the fill style's pattern index.

```
procedure SRGP_setPenStyle (mode: SOLID / BITMAP_PATTERN_OPAQUE / ... );
procedure SRGP_setPenBitmapPattern (patternIndex : integer);
procedure SRGP_setPenPixmapPattern (patternIndex : integer);
```

Application screen background. We have defined "background color" as the color of the 0 bits in bitmap patterns used in opaque mode, but the term *background* is used in another, unrelated way. Typically, the user expects the screen to display primitives on some uniform *application screen background pattern* that covers an opaque window or the entire screen. The application screen background pattern is often solid color 0, since SRGP initializes the screen to that color upon initialization. However, the background pattern is sometimes nonsolid, or solid of some other color; in these cases, the application is responsible for setting up the application screen background by drawing a full-screen rectangle of the desired pattern, before drawing any other primitives.

A common technique to "erase" primitives is to redraw them in the application screen background pattern, rather than redrawing the entire image each time a primitive is deleted. However, this "quick and dirty" updating technique yields a damaged image when the erased primitive overlaps with other primitives. For example, assume that the screen background pattern in Fig. 2.9 is solid white and that we erase the rectangle marked (c) by redrawing it using solid COLOR_WHITE. This would leave a white gap in the filled ellipse arc (b) underneath. "Damage repair" involves going back to the application database and respecifying primitives (see Exercise 2.9).

2.1.4 Saving and Restoring Attributes

As you can see, SRGP supports a variety of attributes for its various primitives. Individual attributes can be saved for later restoration; this feature is especially useful in designing application procedures that perform their functions without side effects—that is, without affecting the global attribute state. For each attribute-setting SRGP procedure, there is a corresponding inquiry procedure that can be used to determine the current value; for example,

```
procedure SRGP_inquireLineStyle (var value : CONTINUOUS / DASHED /...);
```

For convenience, SRGP allows the inquiry and restoration of the entire set of attributes—called the *attribute group*—via

```
procedure SRGP_inquireAttributes (var group : attributeGroup);
procedure SRGP_restoreAttributes (group : attributeGroup);
```

The application program does not have access to the fields of the SRGP-defined "attributeGroup" record; the record returned by the inquiry procedure can be used only for later restoration.

2.1.5 Text

Specifying and implementing text drawing is always complex in a graphics package, because of the large number of options and attributes text can have. Among these are the style or *font* of the characters (Times Roman, Helvetica, Clarinda, etc.), their appearance ("Roman," **bold**, *italic*, underlined, etc.), their size (typically measured in *points*[5]) and widths, the intercharacter spacing, the spacing between consecutive lines, the angle at which characters are drawn (horizontal, vertical, or at a specified angle), and so on.

The most rudimentary facility, typically found in simple hardware and software, is fixed-width, monospace character spacing, in which all characters occupy the same width, and the spacing between them is constant. At the other end of the spectrum, proportional spacing varies both the width of characters and the spacing between them to make the text as legible and aesthetically pleasing as possible. Books, magazines, and newspapers all use proportional spacing, as do most raster graphics displays and laser printers. SRGP provides in-between functionality: Text is horizontally aligned, character widths vary, but space between characters is constant. With this simple form of proportional spacing, the application can annotate graphics diagrams, interact with the user via textual menus and fill-in forms, and even implement simple word processors. Text-intensive applications, however, such as desktop-publishing programs for high-quality documents, need specialized packages that offer more control over text specification and attributes than does SRGP. PostScript [ADOB87] offers many such advanced features and has become an industry standard for describing text and other primitives with a large variety of options and attributes.

[5]A point is a unit commonly used in the publishing industry; it is equal to approximately 1/72 inch.

Text is generated by a call to

```
procedure SRGP_text (origin : point; text : string);
```

The location of a text primitive is controlled by specification of its *origin*, also known as its *anchor point*. The *x* coordinate of the origin marks the left edge of the first character, and the *y* coordinate specifies where the baseline of the string should appear. (The *baseline* is the hypothetical line on which characters rest, as shown in the textual menu buttons of Fig. 2.14. Some characters, such as "y" and "q," have a tail, called the *descender*, that goes below the baseline.)

A text primitive's appearance is determined by only two attributes, the current color and the font, which is an index into an implementation-dependent table of fonts in various sizes and styles:

```
procedure SRGP_setFont (value : integer);
```

Each character in a font is defined as a rectangular bitmap, and SRGP draws a character by filling a rectangle using the character's bitmap as a pattern, in bitmap-pattern-transparent mode. The 1s in the bitmap define the character's interior, and the 0s specify the surrounding space and gaps such as the hole in "o." (Some more sophisticated packages define characters in pixmaps, allowing a character's interior to be patterned.)

Formatting text. Because SRGP implementations offer a restricted repertoire of fonts and sizes, and because implementations on different hardware rarely offer equivalent repertoires, an application has limited control over the height and width of text strings. Since text-extent information is needed in order to produce well-balanced compositions (for instance, to center a text string within a rectangular frame), SRGP provides the following procedure for querying the extent of a given string using the current value of the font attribute:

```
procedure SRGP_inquireTextExtent (
    text : string; var width, height, descent : integer);
```

Although SRGP does not support bitmap opaque mode for writing characters, such a mode can be simulated easily. As an example, the procedure in Fig. 2.15 shows how extent

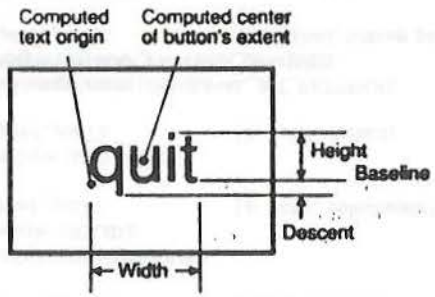

Fig. 2.14 Dimensions of text centered within a rectangular button and points computed from these dimensions for centering purposes.

```
procedure MakeQuitButton (buttonRect : rectangle);
var
    centerOfButton, textOrigin : point;
    width, height, descent : integer;
begin
    SRGP_setFillStyle (SOLID);
    SRGP_setColor (COLOR_WHITE);
    SRGP_fillRectangle (buttonRect);
    SRGP_setColor (COLOR_BLACK);
    SRGP_setLineWidth (2);
    SRGP_Rectangle (buttonRect);

    SRGP_inquireTextExtent ('quit', width, height, descent);

    centerOfButton.x := (buttonRect.bottomLeft.x + buttonRect.topRight.x) div 2;
    centerOfButton.y := (buttonRect.bottomLeft.y + buttonRect.topRight.y) div 2;

    textOrigin.x := centerOfButton.x – (width div 2);
    textOrigin.y := centerOfButton.y – (height div 2);

    SRGP_setColor (COLOR_BLACK);
    SRGP_text (textOrigin, 'quit')
end;
```

Fig. 2.15 Code used to create Fig. 2.14.

information and text-specific attributes can be used to produce black text, in the current font, centered within a white enclosing rectangle, as shown in Fig. 2.14. The procedure first creates the background button rectangle of the specified size, with a separate border, and then centers the text within it. Exercise 2.10 is a variation on this theme.

2.2 BASIC INTERACTION HANDLING

Now that we know how to draw basic shapes and text, the next step is to learn how to write interactive programs that communicate effectively with the user, using input devices such as the keyboard and the mouse. First, we look at general guidelines for making effective and pleasant-to-use interactive programs; then, we discuss the fundamental notion of logical (abstract) input devices. Finally, we look at SRGP's mechanisms for dealing with various aspects of interaction handling.

2.2.1 Human Factors

The designer of an interactive program must deal with many matters that do not arise in a noninteractive, batch program. These are the so-called *human factors* of a program, such as its interaction style (often called "look and feel") and its ease of learning and of use, and they are as important as its functional completeness and correctness. Techniques for user–computer interaction that exhibit good human factors are studied in more detail in Chapters 8 and 9. The guidelines discussed there include these:

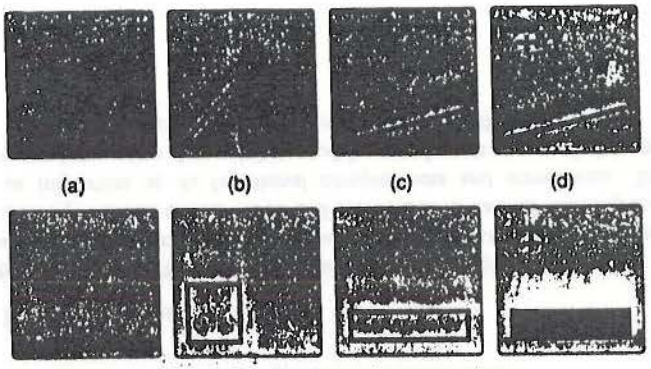

Fig. 2.23 Rubber-echo scenarios.

reset to the center of the screen whenever the locator is deactivated. Unless the programmer explicitly resets it, the measure (and feedback position, if the echo is active) is initialized to that same position when the device is reactivated. At any time, whether the device is active or inactive, the programmer can reset the locator's measure (the *position* portion, not the fields concerning the buttons) using

```
procedure SRGP_setLocatorMeasure (position : point);
```

Resetting the measure while the locator is inactive has no immediate effect on the screen, but resetting it while the locator is active changes the echo (if any) accordingly. Thus, if the program wants the cursor to appear initially at a position other than the center when the locator is activated, a call to SRGP_setLocatorMeasure with that initial position must precede the call to SRGP_setInputMode. This technique is commonly used to achieve continuity of cursor position: The last measure before the locator was deactivated is stored, and the cursor is returned to that position when it is reactivated.

Keyboard attributes and measure control. Unlike the locator, whose echo is positioned to reflect movements of a physical device, there is no obvious screen position for a keyboard device's echo. The position is thus an attribute (with an implementation-specific default value) of the keyboard device that can be set via

```
procedure SRGP_setKeyboardEchoOrigin (origin : point);
```

The default measure for the keyboard is automatically reset to the null string when the keyboard is deactivated. Setting the measure explicitly to a nonnull initial value just before activating the keyboard is a convenient way to present a default input string (displayed by SRGP as soon as echoing begins) that the user can accept as is or modify before pressing the Return key, thereby minimizing typing. The keyboard's measure is set via

```
procedure SRGP_setKeyboardMeasure (measure : string);
```

2.3 RASTER GRAPHICS FEATURES

By now, we have introduced most of the features of SRGP. This section discusses the remaining facilities that take particular advantage of raster hardware, especially the ability

to save and restore pieces of the screen as they are overlaid by other images, such as windows or temporary menus. Such image manipulations are done under control of window- and menu-manager application programs. We also introduce offscreen bitmaps (called *canvases*) for storing windows and menus, and we discuss the use of clipping rectangles.

2.3.1 Canvases

The best way to make complex icons or menus appear and disappear quickly is to create them once in memory and then to copy them onto the screen as needed. Raster graphics packages do this by generating the primitives in invisible, offscreen bitmaps or pixmaps of the requisite size, called *canvases* in SRGP, and then copying the canvases to and from display memory. This technique is, in effect, a type of buffering. Moving blocks of pixels back and forth is faster, in general, than is regenerating the information, given the existence of the fast SRGP_copyPixel operation that we shall discuss soon.

An SRGP *canvas* is a data structure that stores an image as a 2D array of pixels. It also stores some control information concerning the size and attributes of the image. Each canvas represents its image in its own Cartesian coordinate system, which is identical to that of the screen shown in Fig. 2.1; in fact, the screen is itself a canvas, special solely in that it is the only canvas that is displayed. To make an image stored in an off-screen canvas visible, the application must copy it onto the screen canvas. Beforehand, the portion of the screen image where the new image—for example, a menu—will appear can be saved by copying the pixels in that region to an offscreen canvas. When the menu selection has taken place, the screen image is restored by copying back these pixels.

At any given time, there is one *currently active* canvas: the canvas into which new primitives are drawn and to which new attribute settings apply. This canvas may be the screen canvas (the default we have been using) or an offscreen canvas. The coordinates passed to the primitive procedures are expressed in terms of the local coordinate space of the currently active canvas. Each canvas also has its own complete set of SRGP attributes, which affect all drawing on that canvas and are set to the standard default values when the canvas is created. Calls to attribute-setting procedures modify only the attributes in the currently active canvas. It is convenient to think of a canvas as a virtual screen of program-specified dimensions, having its own associated pixmap, coordinate system, and attribute group. These properties of the canvas are sometimes called the *state* or *context* of the canvas.

When SRGP is initialized, the *screen canvas* is automatically created and made active. All our programs thus far have generated primitives into only that canvas. It is the only canvas visible on the screen, and its ID is SCREEN_CANVAS, an SRGP constant. A new offscreen canvas is created by calling the following procedure, which returns the ID allocated for the new canvas:

```
procedure SRGP_createCanvas (width, height : integer; var canvasID : integer);
```

Like the screen, the new canvas's local coordinate system origin (0, 0) is at the bottom-left corner and the top-right corner is at (*width*–1, *height*–1). A 1 by 1 canvas is therefore defined by width and height of 1, and its bottom-left and top-right corners are both (0, 0)! This is consistent with our treatment of pixels as being at grid intersections: The single pixel in a 1 by 1 canvas is at (0, 0).

A newly created canvas is automatically made active and its pixels are initialized to color 0 (as is also done for the screen canvas before any primitives are displayed). Once a canvas is created, its size cannot be changed. Also, the programmer cannot control the number of bits per pixel in a canvas, since SRGP uses as many bits per pixel as the hardware allows. The attributes of a canvas are kept as part of its "local" state information; thus, the program does not need to save the currently active canvas's attributes explicitly before creating a new active canvas.

The application selects a previously created canvas to be the currently active canvas via

```
procedure SRGP_useCanvas (canvasID : integer);
```

A canvas being activated in no way implies that that canvas is made visible; an image in an offscreen canvas must be copied onto the screen canvas (using the SGRP_copyPixel procedure described shortly) in order to be seen.

Canvases are deleted by the following procedure, which may not be used to delete the screen canvas or the currently active canvas.

```
procedure SRGP_deleteCanvas (canvasID : integer);
```

The following procedures allow inquiry of the size of a canvas; one returns the rectangle which defines the canvas coordinate system (the bottom-left point always being (0, 0)), and the other returns the width and height as separate quantities.

```
procedure SRGP_inquireCanvasExtent (canvasID : integer; var extent : rectangle);
procedure SRGP_inquireCanvasSize (
    canvasID : integer; var width, height : integer);
```

Let us examine the way canvases can be used for the implementation of PerformPulldownMenuInteraction, the procedure called by the high-level interaction handler presented in Fig. 2.22 and Section 2.2.6. The procedure is implemented by the pseudocode of Fig. 2.24, and its sequence of actions is illustrated in Fig. 2.25. Each menu has a unique

```
function PerformPulldownMenuInteraction (menuID : integer) : integer;
{The saving/copying of rectangular regions of canvases is described in Section 2.3.3.}
begin
    highlight the menu header in the menu bar;
    menuBodyScreenExtent := screen-area rectangle at which menu body should appear;
    save the current pixels of the menuBodyScreenExtent in a temporary canvas;
        {See Fig. 2.25a.}
    copy menu body image from body canvas to menuBodyScreenExtent;
        {See Fig. 2.25b and Pascal code in Fig. 2.28.}
    wait for button-up signalling the user made a selection, then get locator measure;
    copy saved image from temporary canvas back to menuBodyScreenExtent;
        {See Fig. 2.25c}
    if GEOM_pointInRect (measureOfLocator.position, menuBodyScreenExtent) then
        calculate and return index of chosen item, using y coord of measure position
    else
        return 0
end;
```

Fig. 2.24 Pseudocode for PerformPulldownMenuInteraction.

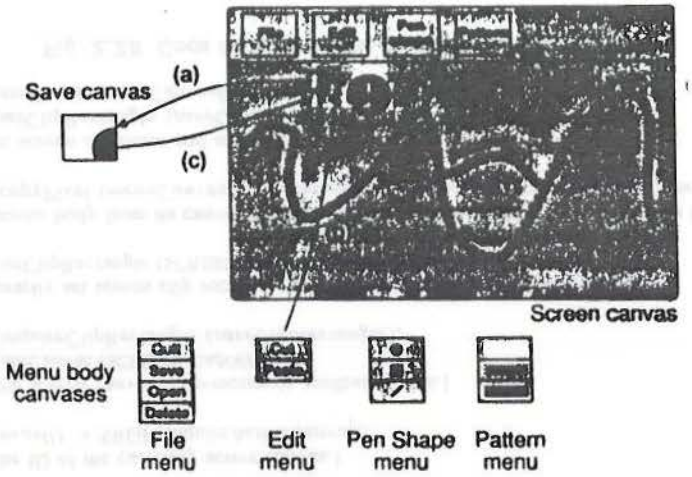

Fig. 2.25 Saving and restoring area covered by menu body.

ID (returned by the CorrelateMenuBar function) that can be used to locate a database record containing the following information about the appearance of the menu body:

- The ID of the canvas storing the menu's body
- The rectangular area (called *menuBodyScreenExtent* in the pseudocode), specified in screen-canvas coordinates, in which the menu's body should appear when the user pulls down the menu by clicking in its header

2.3.2 Clipping Rectangles

Often, it is desirable to restrict the effect of graphics primitives to a subregion of the active canvas, to protect other portions of the canvas. To facilitate this, SRGP maintains a *clip rectangle* attribute. All primitives are clipped to the boundaries of this rectangle; that is, primitives (or portions of primitives) lying outside the clip rectangle are not drawn. Like any attribute, the clip rectangle can be changed at any time, and its most recent setting is stored with the canvas's attribute group. The default clipping rectangle (what we have used so far) is the full canvas; it can be changed to be smaller than the canvas, but it cannot extend beyond the canvas boundaries. The relevant set and inquiry calls for the clip rectangle are

```
procedure SRGP_setClipRectangle (clipRect : rectangle);
procedure SRGP_inquireClipRectangle (var clipRect : integer);
```

A painting application like that presented in Section 2.2.4 would use the clip rectangle to restrict the placement of paint to the drawing region of the screen, ensuring that the surrounding menu areas are not damaged. Although SRGP offers only a single upright rectangle clipping boundary, some more sophisticated software such as POSTSCRIPT offer multiple, arbitrarily shaped clipping regions.

2.3.3 The SRGP_copyPixel Operation

The powerful SRGP_copyPixel command is a typical raster command that is often called bitBlt (bit block transfer) or pixBlt (pixel Blt) when implemented directly in hardware; it first became available in microcode on the pioneering ALTO bitmap workstation at Xerox Palo Alto Research Center in the early 1970s [INGA81]. This command is used to copy an array of pixels from a rectangular region of a canvas, the *source* region, to a *destination* region in the currently active canvas (see Fig. 2.26). The SRGP facility provides only restricted functionality in that the destination rectangle must be of the same size as the source. In more powerful versions, the source can be copied to a destination region of a different size, being automatically scaled to fit (see Chapter 19). Also, additional features may be available, such as *masks* to selectively shield desired source or destination pixels from copying (see Chapter 19), and *halftone patterns* that can be used to "screen" (i.e., shade) the destination region.

SRGP_copyPixel can copy between any two canvases and is specified as follows:

```
procedure SRGP_copyPixel (
    sourceCanvasID : integer; sourceRect : rectangle; destCorner : point);
```

The *sourceRect* specifies the source region in an arbitrary canvas, and *destCorner* specifies the bottom-left corner of the destination rectangle inside the currently active canvas, each in their own coordinate systems. The copy operation is subject to the same clip rectangle that prevents primitives from generating pixels into protected regions of a canvas. Thus, the region into which pixels are ultimately copied is the intersection of the extent of the destination canvas, the destination region, and the clip rectangle, shown as the striped region in Fig. 2.27.

To show the use of copyPixel in handling pull-down menus, let us implement the fourth statement of pseudocode—"copy menu body image"—from the PerformPulldownMenuInteraction function (Fig. 2.24). In the third statement of the pseudocode, we saved in an offscreen canvas the screen region where the menu body is to go; now, we wish to copy the menu body to the screen.

The Pascal code is shown in Fig. 2.28. We must be sure to distinguish between the two rectangles that are of identical size but that are expressed in different coordinate systems.

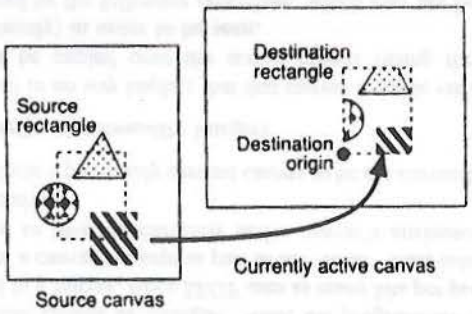

Fig. 2.26 SRGP_copyPixel.

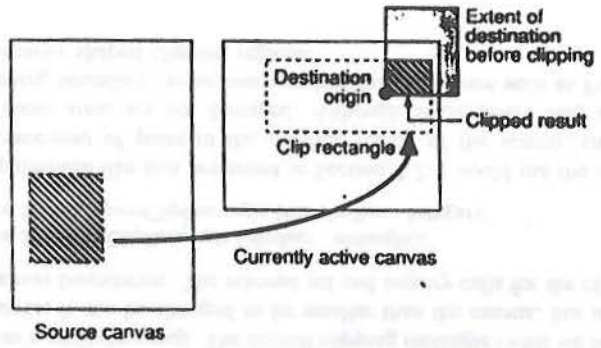

Fig. 2.27 Clipping during copyPixel.

The first rectangle, which we call *menuBodyExtent* in the code, is simply the extent of the menu body's canvas in its own coordinate system. This extent is used as the source rectangle in the SRGP_copyPixel operation that puts the menu on the screen. The *menuBodyScreenExtent* is a rectangle of the same size that specifies in screen coordinates the position in which the menu body should appear; that extent's bottom-left corner is horizontally aligned with the left side of the menu header, and its top-right corner abuts the bottom of the menu bar. (Figure 2.25 symbolizes the Edit menu's screen extent as a dotted outline, and its body extent as a solid outline.) The *menuBodyScreenExtent*'s bottom-left point is used to specify the destination for the SRGP_copyPixel that copies the menu body. It is also the source rectangle for the initial save of the screen area to be overlaid by the menu body and the destination of the final restore.

```
{This code fragment copies a menu-body image onto screen,
 at the screen position stored in the body's record.}

{Save the ID of the currently active canvas.}
saveCanvasID = SRGP_inquireActiveCanvas;

{Save the screen canvas' clip-rectangle attribute value.}
SRGP_useCanvas (SCREEN_CANVAS);
SRGP_inquireClipRectangle (saveClipRectangle);

{Temporarily set screen clip rectangle to allow writing to all of the screen.}
SRGP_setClipRectangle (SCREEN_EXTENT);

{Copy menu body from its canvas to its proper area below the header in the menu bar.}
SRGP_copyPixel (menuCanvasID, menuBodyExtent, menuBodyScreenExtent.lowerLeft);

{Restore screen attributes and active canvas.}
SRGP_setClipRectangle (saveClipRectangle);
SRGP_useCanvas (saveCanvasID);
```

Fig. 2.28 Code for copying the menu body to the screen.

Notice that the application's state is saved and restored to eliminate side effects. We set the screen clip rectangle to SCREEN_EXTENT before copying; alternatively, we could set it to the exact *menuBodyScreenExtent*.

2.3.4 Write Mode or RasterOp

SRGP_copyPixel can do more than just move an array of pixels from a source region to a destination. It can also execute a logical (bitwise) operation between each corresponding pair of pixels in the source and destination regions, then place the result in the destination region. This operation can be symbolized as

$$D \leftarrow S \text{ op } D$$

where **op**, frequently called the *RasterOp* or *write mode*, consists in general of the 16 Boolean operators. Only the most common of these—**replace**, **or**, **xor**, and **and**—are supported by SRGP; these are shown for a 1-bit-per-pixel image in Fig. 2.29.

Write mode affects not only SRGP_copyPixel, but also any new primitives written onto a canvas. As each pixel (either of a source rectangle of a SRGP_copyPixel or of a primitive) is stored in its memory location, either it is written in destructive **replace** mode or its value is logically combined with the previously stored value of the pixel. (This bitwise combination of source and destination values is similar to the way a CPU's hardware performs arithmetic or logical operations on the contents of a memory location during a read–modify–write memory cycle.) Although **replace** is by far the most common mode, **xor** is quite useful for generating dynamic objects, such as cursors and rubberband echoes, as we discuss shortly.

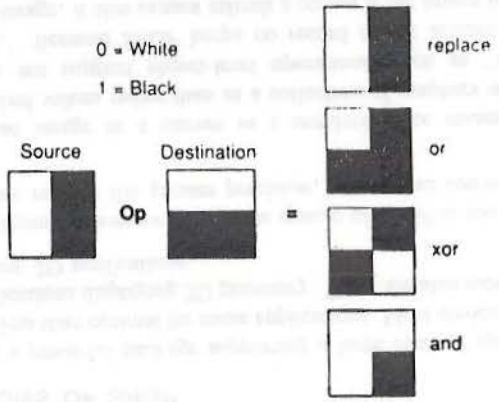

Fig. 2.29 Write modes for combining source and destination pixels.

We set the write-mode attribute with:

```
procedure SRGP_setWriteMode (
    mode : WRITE_REPLACE / WRITE_XOR / WRITE_OR / WRITE_AND);
```

Since all primitives are generated according to the current write mode, the SRGP programmer must be sure to set this mode explicitly and not to rely on the default setting of WRITE_REPLACE.

To see how RasterOp works, we look at how the package actually stores and manipulates pixels; this is the only place where hardware and implementation considerations intrude on the abstract view of raster graphics that we have maintained so far.

RasterOps are performed on the pixel values, which are indices into the color table, not on the hardware color specifications stored as entries in the color table. Thus, for a bilevel, 1-bit-per-pixel system, the RasterOp is done on two indices of 1 bit each. For an 8-bit-per-pixel color system, the RasterOp is done as a bitwise logical operation on two 8-bit indices.

Although the interpretation of the four basic operations on 1-bit-per-pixel monochrome images shown in Fig. 2.29 is natural enough, the results of all but **replace** mode are not nearly so natural for n-bit-per-pixel images ($n > 1$), since a bitwise logical operation on the source and destination indices yields a third index whose color value may be wholly unrelated to the source and destination colors.

The **replace** mode involves writing over what is already on the screen (or canvas). This destructive write operation is the normal mode for drawing primitives, and is customarily used to move and pop windows. It can also be used to "erase" old primitives by drawing over them in the application screen background pattern.

The **or** mode on bilevel displays makes a nondestructive addition to what is already on the canvas. With color 0 as white background and color 1 as black foreground, **or**ing a gray fill pattern onto a white background changes the underlying bits to show the gray pattern. But **or**ing the gray pattern over a black area has no effect on the screen. Thus, **or**ing a light-gray paint swath over a polygon filled with a brick pattern merely fills in the bricks with the brush pattern; it does not erase the black edges of the bricks, as **replace** mode would. Painting is often done in **or** mode for this reason (see Exercise 2.7).

The **xor** mode on bilevel displays inverts a destination region. For example, to highlight a button selected by the user, we set **xor** mode and generate a filled rectangle primitive with color 1, thereby toggling all pixels of the button: 0 **xor** 1 = 1, 1 **xor** 1 = 0. To restore the button's original status, we simply leave **xor** mode, set and draw the rectangle a second time, thereby toggling the bits back to their original state. This technique is also used internally by SRGP to provide the locator's rubber-line and rubber-rectangle echo modes (see Exercise 2.4).

On many bilevel graphics displays, the **xor** technique is used by the underlying hardware (or in some cases software) to display the locator's cursor image in a nondestructive manner. There are some disadvantages to this simple technique; when the cursor is on top of a background with a fine pattern that is almost 50 percent black and 50 percent white, it is possible for the cursor to be only barely noticeable. Therefore, many

bilevel displays and most color displays use **replace** mode for the cursor echo; this technique complicates the echo hardware or software (see Exercise 2.5).

The **and** mode can be used, for example, to reset pixels selectively in the destination region to color 0.

2.4 LIMITATIONS OF SRGP

Although SRGP is a powerful package supporting a large class of applications, inherent limitations make it less than optimal for some applications. Most obviously, SRGP provides no support for applications displaying 3D geometry. There are also more subtle limitations that affect even many 2D applications:

- The machine-dependent integer coordinate system of SRGP is too inflexible for those applications that require the greater precision, range, and convenience of floating-point.
- SRGP stores an image in a canvas in a semantics-free manner as a matrix of unconnected pixel values rather than as a collection of graphics objects (primitives), and thus does not support object-level operations, such as "delete," "move," "change color." Because SRGP keeps no record of the actions that produced the current screen image, it also cannot refresh a screen if the image is damaged by other software, nor can it re–scan-convert the primitives to produce an image for display on a device with a different resolution.

2.4.1 Application Coordinate Systems

In the previous chapter, we introduced the notion that, for most applications, drawings are only a means to an end, and that the primary role of the application database is to support such processes as analysis, simulation, verification, and manufacturing. The database must therefore store geometric information using the range and precision required by these processes, independent of the coordinate system and resolution of the display device. For example, a VLSI CAD/CAM program may need to represent circuits that are 1 to 2 centimeters (cm) long at a precision of half a micron, whereas an astronomy program may need a range of 1 to 10^9 light-years with a precision of a million miles. For maximum flexibility and range, many applications use floating-point *world coordinates* for storing geometry in their database.

Such an application could do the mapping from world to device coordinates itself; however, considering the complexity of this mapping (which we shall discuss in Chapter 6), it is convenient to use a graphics package that accepts primitives specified in world coordinates and maps them to the display device in a machine-independent manner. The recent availability of inexpensive floating-point chips offering roughly the performance of integer arithmetic has significantly reduced the time penalty associated with the use of floating-point—the flexibility makes it well worth its cost to the applications that need it.

For 2D graphics, the most common software that provides floating-point coordinates is Adobe's PostScript (see Chapter 19), used both as the standard page-description language for driving hardcopy printers and (in an extension called Display PostScript) as the graphics

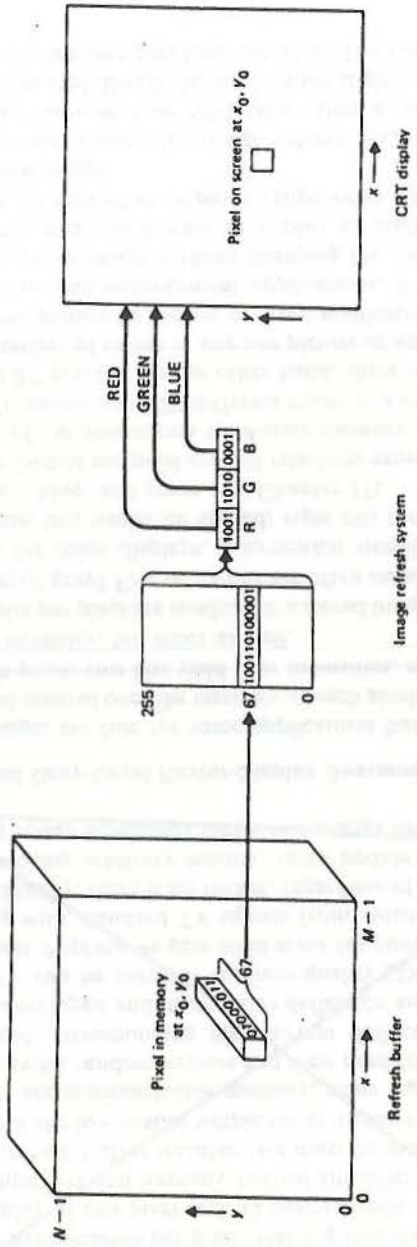

Fig. 3.37 Video look-up table. A pixel with value 67 is shown.

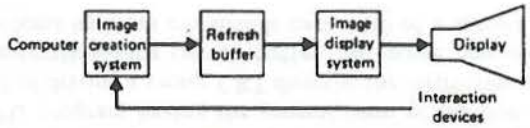

Fig. 3.38 Complete raster-display system.

A raster-display instruction set usually includes points, lines, conic sections, solid areas, and text. Various attributes of these output primitives can be controlled, such as color or intensity, line style (solid, dashed, dotted, etc.), and text spacing, orientation, font, and size. Coordinates are usually given in the coordinate system of the refresh buffer itself: if the buffer is 512 × 512, then the coordinates range from 0 to 511 in each dimension.

The image creation system is also typically able to accept images which already exist in pixel form, such as images of real objects or images of synthetic objects that have already been scan converted. Often a rectangular area of the image storage can be moved around in the buffer. The image creation system can usually load the video look-up table, start and stop image refresh, and deal with interaction devices in much the same way as vector displays.

While there are certainly nontrivial differences between raster and vector displays (discussed further in Chapters 10 and 12), our discussion in the next few chapters considers them as essentially equivalent from the user's and application programmer's points of view. Thus for the time being we will not deal with the ability of a raster display to show solid areas.

EXERCISES

3.1 Modify the line-drawing algorithm in Fig. 3.22 to draw lines with all slopes. Implement the algorithm and test it, observing the visual results for lines with different slopes.

3.2 Design a DPU instruction set in which each opcode (move, point, line) includes both an x and y coordinate, perhaps using multiple words. Do this for a 16-bit and 24-bit instruction length.

3.3 Extend the DPU instruction set from Section 3.3.6 to include two line styles (such as solid and dotted) and four intensity levels. Do this in two different ways: (i) include the style and intensity with each line, point, or text string display instruction; (ii) design instruction(s) to load style and intensity registers. The register values affect all following output primitives until the values are changed.

3.4 Make a list of the advantages and disadvantages of random refresh displays, random DVST displays, and raster refresh displays.

3.5 In some raster systems the image storage is part of the image creation system's memory address space, while in others the image storage and the image creation system's memory are separate. Describe the possible advantages and disadvantages of each arrangement.

resolution 1280 × 1024 image must be displayed at about 30 microseconds per scan line, or about 25 nanoseconds per pixel. Halving this pixel time for repeat-field display requires display of one pixel each 12 nanoseconds. These fast times mean that the deflection amplifiers and intensity control amplifier must have very high bandwidths, and the refresh buffer transfer rate must be increased.

The reason for the low cost of simple raster displays should now be evident: the basic components are semiconductor memory, some logic, a scan generator, and a TV monitor. We avoid random vectors and their need for fast, linear, accurate vector generators and corresponding high-current deflection amplifier technology. Indeed, the well-developed and inexpensive deflection and beam control technology of commercial TV can be used for medium-quality (256 × 256) resolution. Additionally, with raster displays we gain solid areas for *continuous* grey scale or color, and video mixing with standard TV signals from video cameras, video recorders, and video disks. Finally, there is no flicker, regardless of picture complexity. Except in applications needing arbitrary motion, rapid update, and very high resolution, raster technology today dominates vector technology in price/performance ratio.

3.5.1 Color and Grey-Level Raster-Display Systems

Two-intensity images are fine for some applications but grossly unsatisfactory for others. Additional control over the intensity of each pixel is obtained by storing multiple bits for each pixel: two bits yield four intensities, etc. The bits can be used to control not only intensity, but color as well.

How many bits per pixel are needed for a stored image to be perceived as having continuous shades of grey? Five or six bits are often enough, but up to eight bits can be needed. Thus for color displays, a somewhat simplified analysis suggests that three times as many bits would be needed; eight bits for each of the three additive primary colors red, blue, and green (see Chapter 17).

Systems with 24 bits per pixel are still relatively expensive, despite the dramatic decreases in cost of random-access solid-state memory. Furthermore, many color applications don't require up to 2^{24} different colors in a single picture (which typically has only 2^{14} to 2^{20} pixels). On the other hand, there is frequent need for both a relatively small number of colors in any one picture or application and the ability to change colors from picture to picture or from application to application. Also, in many image analysis and enhancement applications, it is desirable to change the visual appearance of an image without changing the underlying data defining the image; for example, one might want to display all pixels with values below some threshold as black, to expand an intensity range, or to create a pseudo-color display of a monochromatic image.

For these various reasons the image refresh system of raster displays often includes a so-called *video look-up table* (also called a *color table* or *color map*). A pixel's value is not routed directly to the intensity digital-to-analog converter, but is instead used as an index into this look-up table. The table entry's value is used to

control the intensity or color on the CRT. A pixel value of 67 would cause the contents of table location 67 to be accessed and used to control the CRT beam. This look-up operation is done for each pixel on each display cycle, so the table must be accessible quickly; for a 512 × 512 image, about 100 nanoseconds is available to process each pixel. The associated computer must be able to load and change the look-up table on program command. The look-up table has as many entries as there are pixel values.

We can diagram systems with n bits per pixel and a look-up table w bits wide as shown in Fig. 3.37. For a monochromatic CRT, 2^w intensity levels are therefore defined. With color, the w bits are typically divided into three equal groups, one for each of the red, blue, and green electron guns of the shadow-mask CRT. Sometimes other color representations with more intuitive appeal (such as intensity and chrominance) are used in the application program and are stored in the refresh buffer. This representation is then converted into red, green, and blue control signals by a fixed-content intensity/chrominance to red/green/blue look-up table. These and other color representations are discussed further in Chapter 17.

3.5.2 Image Creation

How is an image created in the first place? The images of real objects come directly or indirectly from a scanning device of some sort: film scanner, TV scanner, ultrasound scanner, etc. Here, however, we concentrate instead on the creation of synthetic images: images of objects which exist as abstract collections of lines, points, curves, areas, etc. in the computer's memory. This is the usual domain of interactive computer graphics.

There is a fundamental mismatch between the two-dimensional array of pixel values used to drive a raster system and the line, point, and area representation of objects stored and manipulated by the application program. We first saw this mismatch in our discussion of printers as hardcopy raster-scan devices (Section 3.1.1). The process of converting a line, point, and area representation to the pixel array of the image storage is called *scan conversion*. Figure 3.22 is a simple scan-conversion algorithm. Other algorithms for scan-converting lines, as well as areas and circles, are discussed in Chapter 11. For now it is sufficient to say that the algorithms exist and must be executed each time some or all of the displayed image changes. Scan conversion can therefore be a major bottleneck in updating the picture.

Because the scan-conversion algorithms are universally needed in raster-scan systems for interactive graphics, they are often incorporated into the raster-display system as another functional unit, the *image creation system* shown in Fig. 3.38. The entire raster system now loosely corresponds to a random-display DPU. The system accepts a DPU program having the general form of the one discussed in Section 3.3.6. Instead of driving a vector CRT directly, the instructions are converted into a simpler representation—the refresh buffer. Of course, the image creation system need not reprocess its input commands each 1/30 of a second.