CHAPTER FIVE

SEGMENTS

INTRODUCTION

In the preceding chapters we have organized our display as a single picture. In reality, however, the image on the display screen is often composed of several pictures or items of information. A single image may contain several views of an object. It may have a picture of the overall object and a close-up of a particular component. It may also contain information about the object, instructions for the user, and, possibly, error information. As an example, we may wish to display the design plans for a building. The plans may contain structural diagrams, electrical diagrams, plumbing diagrams, and heating diagrams. We might wish to show all this information simultaneously or, at other times, look at the individual elements. As another example, consider an animated display of a spaceship in motion. We might want to show the spaceship at different positions while the background remains fixed, or we might choose to keep the ship centered on the screen and move the background. The transformations of the last chapter tell us how to shift the position of the image, but now we wish to apply them to only a portion of the scene (either the ship or the background, but not both).

We would like to organize our display file to reflect this subpicture structure. We would like to divide our display file into *segments*, each segment corresponding to a component of the overall display. We shall associate a set of *attributes* with each segment. One such attribute is *visibility*. A visible segment will be displayed, but an invisible segment will not be shown. By varying the settings of the visibility attribute, we can build a picture out of the selected subpictures. Consider the building plan applica-

tion. We might, for example, in one display select structural and electrical information for a building by making the first and second segments visible and the third and fourth segments invisible. In another display we could select structural and plumbing information by making only the first and third segments visible. While all display information may be present in the machine, we can selectively show portions of it by designating which segments of the display file are to be interpreted.

Another attribute which can be associated with each segment is an image transformation. This will allow the independent scaling, rotating, and translating of each segment. For the spaceship, we could put the ship in one segment and the background in a different segment. We could then shift either of them by means of an image transformation, while leaving the other unchanged. (See Figure 5-1.)

In this chapter we shall consider segmentation of the display file. We shall learn how to create, close, rename, and delete segments. We shall also consider such display-file attributes as visibility and image transformation.

THE SEGMENT TABLE

We shall begin our discussion by considering some of the information that must be associated with each segment and how this information might be organized. We must give each segment its own unique name so that we can specify it. If we are to change the visibility of a segment, we must have some way to distinguish that segment from all the others. When we refer to a display-file segment, we must know which display-file instructions belong to it. This may be determined by knowing where the display-file instructions for the segment begin and how many of them there are. For each segment, we shall need some way of associating its display-file position information and its attribute information with its name. We need to organize this information so that given the segment name, we can look up or alter its attributes, or given its name, we can interpret the corresponding display-file instructions. In our system we shall do this

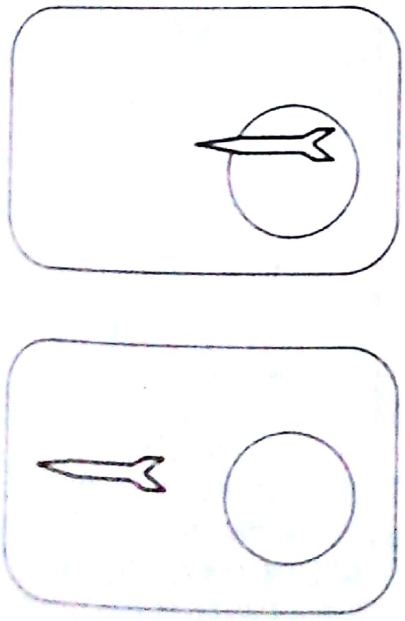

FIGURE 5-1
Transformation of a portion of the display.

by forming a segment table. We shall use a number for the name of the segment. Simple arrays will serve to hold segment properties, and the segment name will be used as the index into these arrays. We shall have one array containing display-file starting locations. A second array will hold segment size information, while a third will indicate visibility, and so on. To find out the size of, say, the third segment, we would look for the third element in the size array. (See Figure 5-2.)

The number 1 will refer to the first named segment of the table, 2, to the second named table entry, and so on. We shall have table entries (arrays) for holding the display-file starting position, for the segment size, and for attributes such as visibility and the image transformation parameters. If we wish not to show segment number 3, we will set the corresponding entry in the visibility array to off. When we show our picture, we shall consult the segment table to determine which segments are visible. For each visible segment, we shall look up its starting point and size and pass this information along to our display-file interpreter. The interpreter will therefore only interpret those segments which are visible.

There are other possible schemes for implementing the segment table, many with substantial advantages over the one which we have chosen here. We have selected this design because it allows simple accessing. It does not require any new data-structuring concepts, and its updating is straightforward.

We would like this extension to our graphics system to be compatible with our previous programs, which did not specify any segments. Perhaps the easiest way to do this is to say that when no segment name is specified, the instructions will be placed in a special "unnamed" segment. Information, such as display-file starting position and segment size, must be stored for the unnamed segment just as it is for named segments. We shall do this by placing a special entry in the segment table for the unnamed

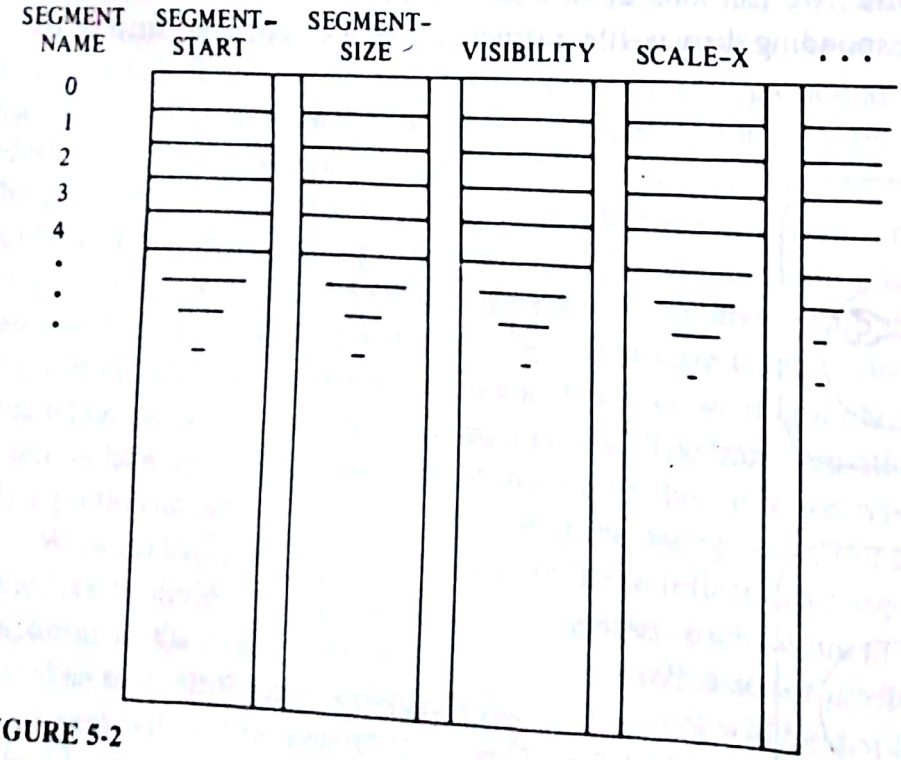

FIGURE 5-2
The segment table.

segment. In this implementation, we have chosen to associate the unnamed segment with table index 0. That is, the segment table arrays will be dimensioned with lower bound 0 instead of 1. Thus SEGMENT-SIZE[1] will still be the number of instructions of the segment with name 1, but SEGMENT-SIZE[0] will be the number of instructions in the unnamed segment.

SEGMENT CREATION

Let us consider the process of creating or opening a segment. When we create a segment, we are saying that all subsequent LINE, MOVE, TEXT, or POLYGON commands will be members of this segment. We must give the segment a name so that we can identify it. We might, for example, say that this is segment number 3. Then all following MOVE and LINE commands would belong to segment 3. We could then close segment 3 and open another segment, say segment 5. The next MOVE and LINE commands would belong to segment 5.

The first thing we do when we create a segment is check to see whether some other segment is still open. We cannot have two segments open at the same time because we would not know to which segment we should assign the drawing instructions. If there is a segment still open, we have an error. If no segment is currently open, we should check to make sure that we have a valid segment name. If the segment name is correct, check to see whether there already exists a segment under this name. If so, we again have an error. We initialize the items in the segment table under our segment name to indicate that this is a fresh new segment. The first instruction belonging to this segment will be located at the next free storage area in the display file. The current size of the segment is zero since we have not actually entered any instructions into it yet. The attributes are initialized to those of the unnamed segment, which provides the default attribute values. Finally we indicate that there is now a segment open (the one which we just created).

5.1 Algorithm CREATE-SEGMENT(SEGMENT-NAME) User routine to create a named segment

```
Argument   SEGMENT-NAME the segment name
Global     NOW-OPEN the segment currently open
           FREE the index of the next free display-file cell
           SEGMENT-START, SEGMENT-SIZE, VISIBILITY
           ANGLE, SCALE-X, SCALE-Y, TRANSLATE-X, TRANSLATE-Y
           arrays that make up the segment table
Constant   NUMBER-OF-SEGMENTS size of the segment table
BEGIN
  IF NOW-OPEN > 0 THEN RETURN ERROR 'SEGMENT STILL OPEN';
  IF SEGMENT-NAME < 1 OR SEGMENT-NAME > NUMBER-OF-SEGMENTS
  THEN
    RETURN ERROR 'INVALID SEGMENT NAME';
  IF SEGMENT-SIZE[SEGMENT-NAME] > 0 THEN
    RETURN ERROR 'SEGMENT ALREADY EXISTS';
  SEGMENT-START[SEGMENT-NAME] ← FREE;
```

```
    SEGMENT-SIZE[SEGMENT-NAME] ← 0;
    VISIBILITY[SEGMENT-NAME] ← VISIBILITY[0];
    ANGLE[SEGMENT-NAME] ← ANGLE[0];
    SCALE-X[SEGMENT-NAME] ← SCALE-X[0];
    SCALE-Y[SEGMENT-NAME] ← SCALE-Y[0];
    TRANSLATE-X[SEGMENT-NAME] ← TRANSLATE-X[0];
    TRANSLATE-Y[SEGMENT-NAME] ← TRANSLATE-Y[0];
    NOW-OPEN ← SEGMENT-NAME;
    RETURN;
END;
```

CLOSING A SEGMENT

With the segment open we can proceed with the entry of display-file instructions, as we did in previous chapters. Now, however, all commands which are entered are associated with the open segment. Once we have completed the drawing instructions for the segment, we should close it. At this point, all that is necessary in closing the segment is to change the value of the NOW-OPEN variable, which indicates the name of the currently open segment. We must change it to something, so we shall set it to 0 and make the unnamed segment the one which is open. In our algorithm below we do this by placing a 0 in the NOW-OPEN variable. We don't want to have two unnamed segments around because we shall only show one of them and the other would just waste storage. So we delete any unnamed segment instructions which may have been saved. We initialize the unnamed segment to have no instructions, but to be ready to receive them in the next free display-file location.

5.2 Algorithm CLOSE-SEGMENT User routine to close the currently open segment

```
Global      NOW-OPEN the name of the currently open segment
            FREE the index of the next free display-file cell
            SEGMENT-START, SEGMENT-SIZE start and size of the segments
BEGIN
    IF NOW-OPEN = 0 THEN RETURN ERROR 'NO SEGMENT OPEN';
    DELETE-SEGMENT(0);
    SEGMENT-START[0] ← FREE;
    SEGMENT-SIZE[0] ← 0;
    NOW-OPEN ← 0;
    RETURN;
END;
```

DELETING A SEGMENT

The most complex algorithm for this chapter is that for deleting a segment. If a segment is no longer needed, we would like to recover the display-file storage occupied by its instructions. Since these instructions will never again be executed, we would prefer to use this storage for some other segment. At the same time, we would rather not have to destroy and re-form the entire display file. We may be altering some small

portion of an image which is very costly to generate. We would like to be able to delete (and perhaps re-create with modifications) just one segment, while preserving the rest of the display file. The method of doing this depends upon the data structure used for the display file. We have used arrays. Recovery of a block of storage in an array is both simple and straightforward, but it is not as efficient as some other storage techniques. What we shall do is take all display-file instructions entered after the segment which we are deleting was closed, and move them up in the display file so that they lie on top of the deleted segment. Thus we fill in the gap left by the deleted block and recover an equivalent amount of storage at the end of the display file. (See Figure 5-3.)

We begin our segment deletion algorithm by checking to make sure that we have a valid segment name. If the name is correct, we next check that it is not open. Open segments are still in use; if an attempt is made to delete an open segment, we shall treat it as an error. We next check to see whether the size of the segment is 0. A segment with no instructions has nothing to remove from the display file, so no further processing is necessary. We can now shift the instructions in the display file. We wish to place instructions on top of the deleted segment. So our first relocated instruction will be put on top of the first instruction of our deleted segment. We will get the instruction to be moved from the first location beyond the deleted segment. We shall move through the display file, getting instructions and shifting them down until we come to an unused display-file location. When we have completed moving the display-file instructions, we can reset the index of the next free instruction to reflect the recovered storage. Finally, we must adjust not only the display file but also the segment table, since the starting positions of segments created after the segment which was deleted will now have changed. We can do this by scanning the segment table and noting any segments whose starting position lies beyond the starting position of the segment which we deleted. We change the starting positions by subtracting the size of the deleted segment. We can set the size of the segment which we deleted to be 0 to indicate that this segment no longer exists. If we delete a visible segment, a NEW-FRAME action is required. These steps are detailed in the following algorithm:

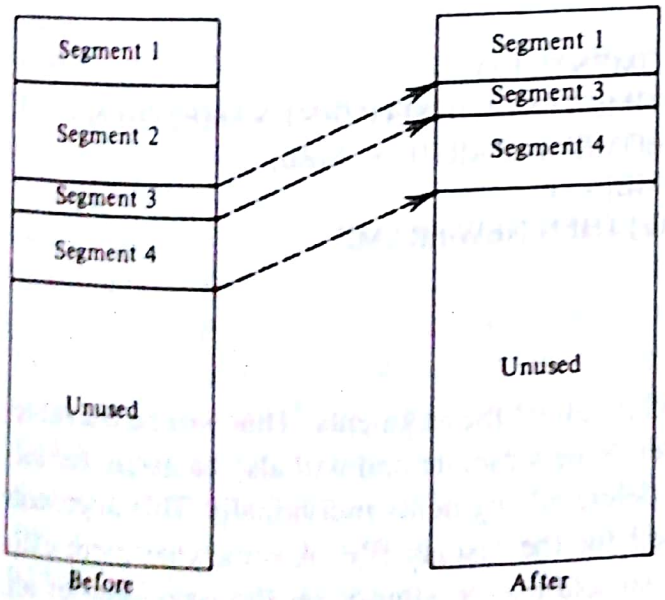

FIGURE 5-3
Deleting segment 2 from the display file.

5.3 Algorithm DELETE-SEGMENT(SEGMENT-NAME) User routine to delete a segment

```
Argument   SEGMENT-NAME the segment name
Global     NOW-OPEN the currently open segment
           FREE the index of the next free display-file cell
           DF-OP, DF-X, DF-Y the display-file arrays
           SEGMENT-START, SEGMENT-SIZE, VISIBILITY part of the segment
           table arrays
Constant   NUMBER-OF-SEGMENTS the size of the segment table
Local      GET the location of an instruction to be moved
           PUT the location to which an instruction should be moved
           SIZE the size of the deleted segment
           I a variable for stepping through the segment table

BEGIN
  IF SEGMENT-NAME < 0 OR SEGMENT-NAME > NUMBER-OF-SEGMENTS
  THEN
    RETURN ERROR 'INVALID SEGMENT NAME';
  IF SEGMENT-NAME = NOW-OPEN AND SEGMENT-NAME ≠ 0 THEN
    RETURN ERROR 'SEGMENT STILL OPEN';
  IF SEGMENT-SIZE[SEGMENT-NAME] = 0 THEN RETURN;
  PUT ← SEGMENT-START[SEGMENT-NAME];
  SIZE ← SEGMENT-SIZE[SEGMENT-NAME];
  GET ← PUT + SIZE;
  shift the display-file elements
  WHILE GET < FREE DO
    BEGIN
      DF-OP[PUT] ← DF-OP[GET];
      DF-X[PUT] ← DF-X[GET];
      DF-Y[PUT] ← DF-Y[GET];
      PUT ← PUT + 1;
      GET ← GET + 1;
    END;
  recover the deleted storage
  FREE ← PUT;
  update the segment table
  FOR I = 0 TO NUMBER-OF-SEGMENTS DO
    IF SEGMENT-START[I] > SEGMENT-START[SEGMENT-NAME] THEN
      SEGMENT-START[I] ← SEGMENT-START[I] − SIZE;
  SEGMENT-SIZE[SEGMENT-NAME] ← 0;
  IF VISIBILITY[SEGMENT-NAME] THEN NEW-FRAME;
  RETURN;
END;
```

We also shall have a routine to delete all of the segments. This will be useful to the user should he wish to start a completely new picture and will also be useful for initialization. One way of doing this is to delete all segments individually. This approach is independent of the data structure used for the display file. A somewhat more efficient approach for our particular array structure is to simply set the size value of all

segments to 0 and initialize the free cell index FREE to be the first cell in the display file. We also set all starting positions to 1 so that after initialization there will not be any garbage in these locations which might upset the DELETE-SEGMENT routine.

with Daigram.

5.4 Algorithm DELETE-ALL-SEGMENTS User routine to delete all segments

```
Global      NOW-OPEN the segment currently open
            FREE the index of the next available display-file cell
            SEGMENT-SIZE the segment size array
            SEGMENT-START the segment starting index array
Constant    NUMBER-OF-SEGMENTS the size of the segment table
Local       I a variable for stepping through the segment table
BEGIN
  FOR I = 0 TO NUMBER-OF-SEGMENTS DO
    BEGIN
      SEGMENT-START[I] ← 1;
      SEGMENT-SIZE[I] ← 0;
    END;
  NOW-OPEN ← 0;
  FREE ← 1;
  NEW-FRAME;
  RETURN;
END;
```

RENAMING A SEGMENT

Another routine which is easy to implement and often useful is renaming a segment. As an example of how it might be used, consider a display device with an independent display processor. The display processor is continuously reading the display file and showing its current contents. (We would not need a MAKE-PICTURE-CURRENT routine for such a device because the picture is always current.) Now suppose we wish to use this device to show an animated character moving on the display. This would be done by presenting a sequence of images, each with a slightly different drawing of the character. Assume we have a segment for the character. Then, for each new image, we could delete the segment, re-create it with the altered character, and show the result. The problem with this is that during the time after the first image is deleted but before the second image is completed, only a partially completed character can be seen. Since we may begin working on the next image as soon as the last one is completed, we may in fact be continually looking at only partially completed characters. To avoid this problem we should not delete a segment until a replacement for it is completed. This means that both segments must exist in the display file at the same time. We do this by building the new invisible image under some temporary segment name. When it is completed, we can delete the original image, make the replacement image visible, and rename the new segment to become the old segment. These steps can be repeated to achieve apparent motion. The idea of maintaining two images, one to show and one to build or alter, is called *double buffering*. The renaming is carried out by our RENAME-SEGMENT algorithm. The algorithm checks that the segment names are

valid and that they are not still open. It also checks against using the name of an already existing segment. If these conditions are met, the segment table entries for the old name are copied into the new name position and the size of the old segment is set back to 0.

5.5 Algorithm RENAME-SEGMENT(SEGMENT-NAME-OLD, SEGMENT-NAME-NEW) User routine to rename SEGMENT-NAME-OLD to be SEGMENT-NAME-NEW

```
Arguments   SEGMENT-NAME-OLD old name of segment
            SEGMENT-NAME-NEW new name of segment
Global      SEGMENT-START, SEGMENT-SIZE, VISIBILITY, ANGLE, SCALE-X,
            SCALE-Y, TRANSLATE-X, TRANSLATE-Y the segment table arrays
Constant    NUMBER-OF-SEGMENTS the size of the segment table
BEGIN
  IF SEGMENT-NAME-OLD < 1 OR SEGMENT-NAME-NEW < 1 OR
    SEGMENT-NAME-OLD > NUMBER-OF-SEGMENTS OR
    SEGMENT-NAME-NEW > NUMBER-OF-SEGMENTS THEN
      RETURN ERROR 'INVALID SEGMENT NAME';
  IF SEGMENT-NAME-OLD = NOW-OPEN OR
    SEGMENT-NAME-NEW = NOW-OPEN THEN
    RETURN ERROR 'SEGMENT STILL OPEN';
  IF SEGMENT-SIZE[SEGMENT-NAME-NEW] ≠ 0 THEN
    RETURN ERROR 'SEGMENT ALREADY EXISTS';
  copy the old segment table entry into the new position
  SEGMENT-START[SEGMENT-NAME-NEW]
    ← SEGMENT-START[SEGMENT-NAME-OLD];
  SEGMENT-SIZE[SEGMENT-NAME-NEW]
    ← SEGMENT-SIZE[SEGMENT-NAME-OLD];
  VISIBILITY[SEGMENT-NAME-NEW] ← VISIBILITY[SEGMENT-NAME-OLD];
  ANGLE[SEGMENT-NAME-NEW] ← ANGLE[SEGMENT-NAME-OLD];
  SCALE-X[SEGMENT-NAME-NEW] ← SCALE-X[SEGMENT-NAME-OLD];
  SCALE-Y[SEGMENT-NAME-NEW] ← SCALE-Y[SEGMENT-NAME-OLD];
  TRANSLATE-X[SEGMENT-NAME-NEW]
    ← TRANSLATE-X[SEGMENT-NAME-OLD];
  TRANSLATE-Y[SEGMENT-NAME-NEW]
    ← TRANSLATE-Y[SEGMENT-NAME-OLD];
  delete the old segment
  SEGMENT-SIZE[SEGMENT-NAME-OLD] ← 0;
  RETURN;
END;
```

VISIBILITY

We have talked about the property of visibility. Each segment is given a visibility attribute. The segment's visibility is stored in an array as part of the segment table. We are therefore able to look up the value of each segment's visibility. By checking this array we can determine whether or not the segment should be displayed. We should give the user some method of changing the value of a segment's visibility, so that he can choose to show or not to show the segment. The following algorithm does this, while freeing

the user from any concern about global variables and segment table representation. If the visibility is being turned off, then a new-frame action is needed. (See Figure 5-4.)

5.6 Algorithm SET-VISIBILITY(SEGMENT-NAME, ON-OFF) User routine to set the visibility attribute

```
Arguments   SEGMENT-NAME the name of the segment
            ON-OFF the new visibility setting
Global      VISIBILITY the array of visibility flags
Constant    NUMBER-OF-SEGMENTS the size of the segment table
BEGIN
   IF SEGMENT-NAME < 1 OR SEGMENT-NAME > NUMBER-OF-SEGMENTS
   THEN
      RETURN ERROR 'INVALID SEGMENT NAME';
   VISIBILITY[SEGMENT-NAME] ← ON-OFF;
   IF NOT ON-OFF THEN NEW-FRAME;
   RETURN;
END;
```

IMAGE TRANSFORMATION

An image transformation is carried out on the contents of the display file. If we think of the display file as containing the picture which we have constructed, then the image

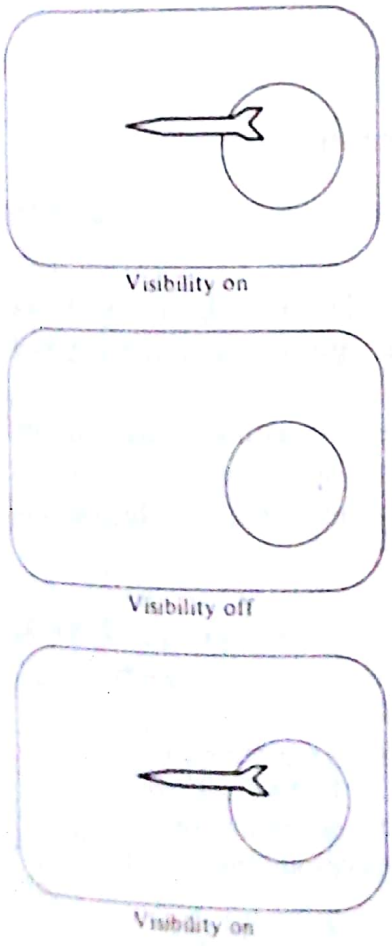

FIGURE 5-4
Changing the visibility attribute.

transformation provides some variety as to how that picture is displayed. This transformation may be supported by the hardware which reads the display file and generates the image. In the following chapters we shall see how other transformations may be used in the picture-creation process.

We would like to give each segment its own image transformation attributes. Let us now consider how this might be done. As we saw in the last chapter, our image transformation can be specified by five numbers: x and y scale factors, a rotation angle, and x and y translation amounts. These are, then, five more attributes to be saved for each segment. In Chapter 4 we used global variables to hold the image transformation parameters. Now, however, we shall use arrays so that individual parameters may be stored for each of the display-file segments. We shall have an array for each type of parameter as part of the segment table. Of course, the user must be able to set the image transformation parameters. Translation may be set by the following algorithm:

5.7 Algorithm SET-IMAGE-TRANSLATION(SEGMENT-NAME, TX, TY) User routine to set the image translation for a segment

```
Arguments   SEGMENT-NAME the segment being transformed
            TX, TY the translation parameters
Global      TRANSLATE-X, TRANSLATE-Y segment translation parameter table
Constant    NUMBER-OF-SEGMENTS the size of the segment table
BEGIN
   IF SEGMENT-NAME < 1 OR SEGMENT-NAME > NUMBER-OF-SEGMENTS
   THEN
      RETURN ERROR 'INVALID SEGMENT NAME';
   TRANSLATE-X[SEGMENT-NAME] ← TX;
   TRANSLATE-Y[SEGMENT-NAME] ← TY;
   IF VISIBILITY[SEGMENT-NAME] THEN NEW-FRAME;
   RETURN;
END;
```

The above algorithm saves translation amounts for the SEGMENT-NAME segment. Notice that a new-frame action is called only if the segment which is being modified happens to be visible.

A graphics system may not have individual routines for setting image scale and rotation. This is because almost every time a scale or rotation is performed, a translation adjustment is also needed. Our system will provide a single routine which sets all of the image's transformation parameters. (See Figure 5-5.)

5.8 Algorithm SET-IMAGE-TRANSFORMATION(SEGMENT-NAME, SX, SY, A, TX, TY) User routine to set the image transformation parameters of a segment

```
Arguments   SEGMENT-NAME the segment being transformed
            SX, SY, A, TX, TY the new image transformation parameters
Global      VISIBILITY, SCALE-X, SCALE-Y, ANGLE, TRANSLATE-X,
            TRANSLATE-Y arrays for attribute part of the segment table
Constant    NUMBER-OF-SEGMENTS the size of the segment table
```

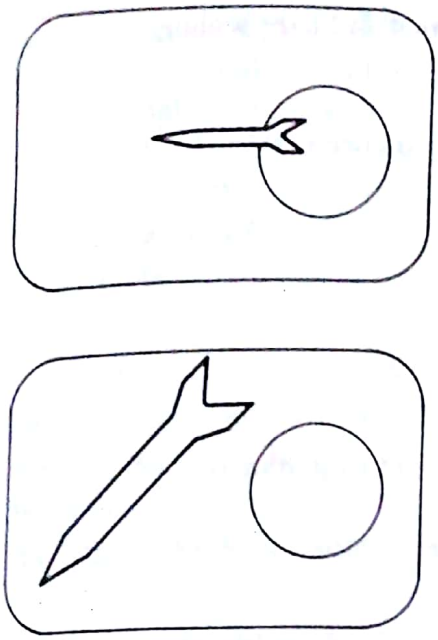

FIGURE 5-5
Changing the image transformation.

```
BEGIN
   IF SEGMENT-NAME < 1 OR SEGMENT-NAME > NUMBER-OF-SEGMENTS
   THEN
      RETURN ERROR 'INVALID SEGMENT NAME';
   SCALE-X[SEGMENT-NAME] ← SX;
   SCALE-Y[SEGMENT-NAME] ← SY;
   ANGLE[SEGMENT-NAME] ← A;
   TRANSLATE-X[SEGMENT-NAME] ← TX;
   TRANSLATE-Y[SEGMENT-NAME] ← TY;
   IF VISIBILITY[SEGMENT-NAME] THEN NEW-FRAME;
   RETURN;
END;
```

REVISING PREVIOUS TRANSFORMATION ROUTINES

We can modify the algorithms of Chapter 4 that set the image parameters to be compatible with the segmented display file by making them correspond to the unnamed segment.

5.9 Algorithm TRANSLATE(TX, TY) (Upgrade of algorithm 4.6) Setting the translation parameters for the unnamed segment

Argument TX, TY the user translation specification
Global TRANSLATE-X, TRANSLATE-Y arrays for translation part of the segment table

```
BEGIN
   TRANSLATE-X[0]← TX;
   TRANSLATE-Y[0]← TY;
   NEW-FRAME;
   RETURN;
END;
```

5.10 Algorithm SCALE(SX, SY) (Upgrade of algorithm 4.7) Image scaling transformation
Arguments SX, SY the scaling parameters
Global SCALE-X, SCALE-Y the segment scaling parameter tables
BEGIN
SCALE-X[0]← SX;
SCALE-Y[0] ← SY;
NEW-FRAME;
RETURN;
END;

5.11 Algorithm ROTATE(A) (Upgrade of algorithm 4.8) Image rotation
Argument A the angle of rotation
Global ANGLE the segment-angle parameter table
BEGIN
ANGLE[0]← A;
NEW-FRAME;
RETURN;
END;

The algorithm for building the complete transformation matrix should now specify which segment's image transformation parameters should be used.

5.12 Algorithm BUILD-TRANSFORMATION (SEGMENT-NAME) (Upgrade of algorithm 4.14) Build the image transformation matrix
Argument SEGMENT-NAME the segment which we are transforming
Global SCALE-X, SCALE-Y, ANGLE, TRANSLATE-X, TRANSLATE-Y arrays for attribute part of the segment table
IMAGE-XFORM a 3 × 2 array containing the image transformation
INVERSE-IMAGE-XFORM a 3 × 2 array for the inverse of the image transformation
BEGIN
IF SEGMENT-NAME < 0 OR SEGMENT-NAME > NUMBER-OF-SEGMENTS THEN
RETURN ERROR 'INVALID SEGMENT NAME';
IDENTITY-MATRIX(IMAGE-XFORM);
MULTIPLY-IN-SCALE(SCALE-X[SEGMENT-NAME],SCALE-Y[SEGMENT-NAME],IMAGE-XFORM);
MULTIPLY-IN-ROTATION(ANGLE[SEGMENT-NAME], IMAGE-XFORM);
MULTIPLY-IN-TRANSLATION(TRANSLATE-X[SEGMENT-NAME], TRANSLATE-Y[SEGMENT-NAME], IMAGE-XFORM);
IDENTITY-MATRIX(INVERSE-IMAGE-XFORM);
MULTIPLY-IN-TRANSLATION(− WIDTH-START, − HEIGHT-START, INVERSE-IMAGE-XFORM);
MULTIPLY-IN-SCALE(1 / WIDTH, 1 / HEIGHT, INVERSE-IMAGE-XFORM);
MULTIPLY-IN-TRANSLATION(− TRANSLATE-X[SEGMENT-NAME], − TRANSLATE-Y[SEGMENT-NAME],INVERSE-IMAGE-XFORM);
MULTIPLY-IN-ROTATION(− ANGLE[SEGMENT-NAME],INVERSE-IMAGE-XFORM);

```
    MULTIPLY-IN-SCALE(1/SCALE-X[SEGMENT-NAME],1/SCALE-Y[SEGMENT-
      NAME],INVERSE-IMAGE-XFORM);
    MULTIPLY-IN-SCALE(WIDTH, HEIGHT, INVERSE-IMAGE-XFORM);
    MULTIPLY-IN-TRANSLATION(WIDTH-START,HEIGHT-START,
      INVERSE-IMAGE-XFORM);
    RETURN;
END;
```

We shall also need an initialization routine for this chapter. At the start of processing, all segments should be empty. A call on DELETE-ALL-SEGMENTS accomplishes this. The unnamed segment should always be visible, and this attribute can be initialized here. At the start of the program, no named segments should be open, so the NOW-OPEN variable is initialized to 0.

5.13 Algorithm INITIALIZE-5

```
Global      VISIBILITY the segment visibility table
            NOW-OPEN the currently open segment
BEGIN
    INITIALIZE-4;
    DELETE-ALL-SEGMENTS;
    VISIBILITY[0] ← TRUE;
    NOW-OPEN ← 0;
    RETURN;
END;
```

SAVING AND SHOWING SEGMENTS

So far we have given a lot of algorithms for creating and storing information about segments in a segment table. We still have to attach this segment structure to the routines for saving and for showing display-file instructions. (See Figure 5-6.)

The first routine we must alter is PUT-POINT. We must add to this routine a statement which increments the size of the segment currently open every time a new instruction is added to a display file.

5.14 Algorithm PUT-POINT(OP, X, Y) Extension of algorithm 2.1 to include updating the segment table

```
Arguments   OP, X, Y a display-file instruction
Global      NOW-OPEN the segment currently open
            SEGMENT-SIZE the segment size array
            DF-OP, DF-X, DF-Y the three display-file arrays
            FREE the position of the next free display-file cell
BEGIN
    SEGMENT-SIZE[NOW-OPEN] ← SEGMENT-SIZE[NOW-OPEN] + 1;
    IF FREE > DFSIZE THEN RETURN ERROR 'DISPLAY FILE FULL';
    DF-OP[FREE] ← OP;
    DF-X[FREE] ← X;
    DF-Y[FREE] ← Y;
```

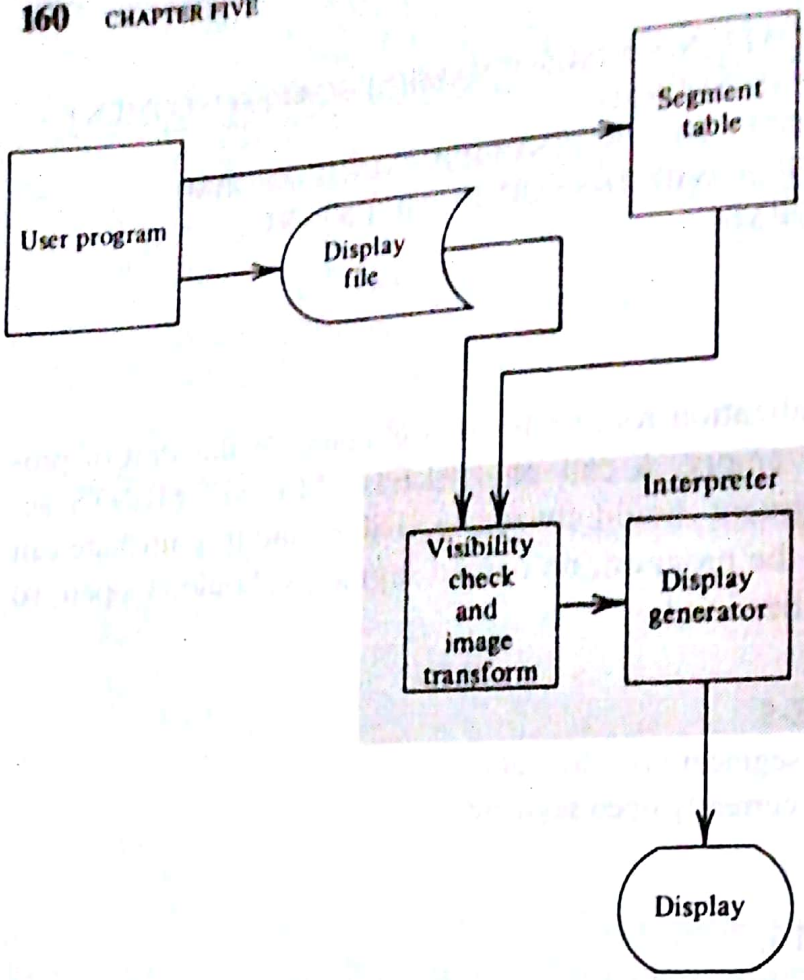

FIGURE 5-6
Picture generation with segments.

```
    FREE ← FREE + 1;
    RETURN;
END;
```

The second routine we must change is MAKE-PICTURE-CURRENT. This routine contained an instruction to display the entire display file, beginning with instruction 1 and ending with index FREE − 1. We replace this statement by a loop which steps through the segment table, examining each segment to see if its size is greater than 0 and if it is also visible. If both of these conditions are met, then its image transformation matrix is formed and the segment is interpreted. The segment table tells us where to begin and how many instructions to interpret. We pass this information to our INTERPRET routine as its arguments.(See Figure 5-7.)

5.15 Algorithm MAKE-PICTURE-CURRENT (Revision of algorithm 4.10)

```
Global      SEGMENT-START, SEGMENT-SIZE, VISIBILITY the segment table
            ERASE-FLAG a flag indicating that the display should be erased
Local       I a variable for stepping through the segment table
Constant    NUMBER-OF-SEGMENTS the size of the segment table
BEGIN
   IF ERASE-FLAG THEN
      BEGIN
         ERASE;
         ERASE-FLAG ← FALSE;
      END;
```

```
    FOR I = 0 TO NUMBER-OF-SEGMENTS DO
      IF SEGMENT-SIZE[I] ≠ 0 AND VISIBILITY[I] THEN
        BEGIN
          BUILD-TRANSFORMATION(I);
          INTERPRET(SEGMENT-START[I], SEGMENT-SIZE[I]);
        END;
    DISPLAY;
    DELETE-SEGMENT(0);
    RETURN;
END;
```

OTHER DISPLAY-FILE STRUCTURES

The necessary operations on the display file are insertion, when we construct a drawing; selection, when we interpret and display; and deletion, when we are finished with a segment. There are many possible data structures which might be used. We have chosen a simple one, the array. While insertion, selection, and deletion are easy for an array, deletion may not be very efficient. If we wish to remove an instruction at the beginning of the display file, we must move all succeeding instructions. If the display file is large, this could mean a lot of processing to recover only a small amount of storage. One alternative data structure which might be used is the *linked list*. (See Figure 5-8.)

In a linked list the instructions are not stored in order; rather a new field is added to the instruction. This field, called the link or pointer, gives the location of the next instruction. We step through the instructions by following the chain of links. The instruction cells which have not yet been used are also linked to form a *list of available space*.

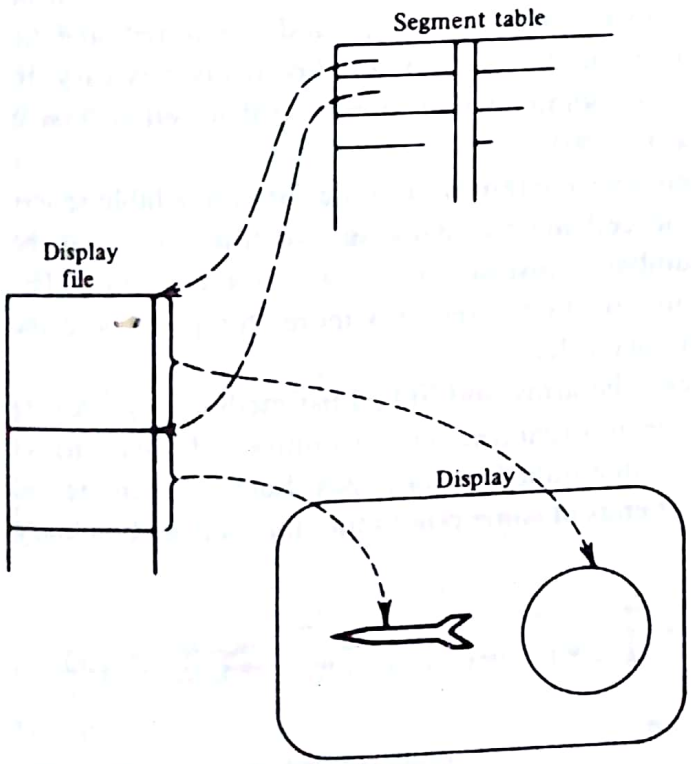

FIGURE 5-7
The segment table indicates the portions of the display file used to construct the picture.

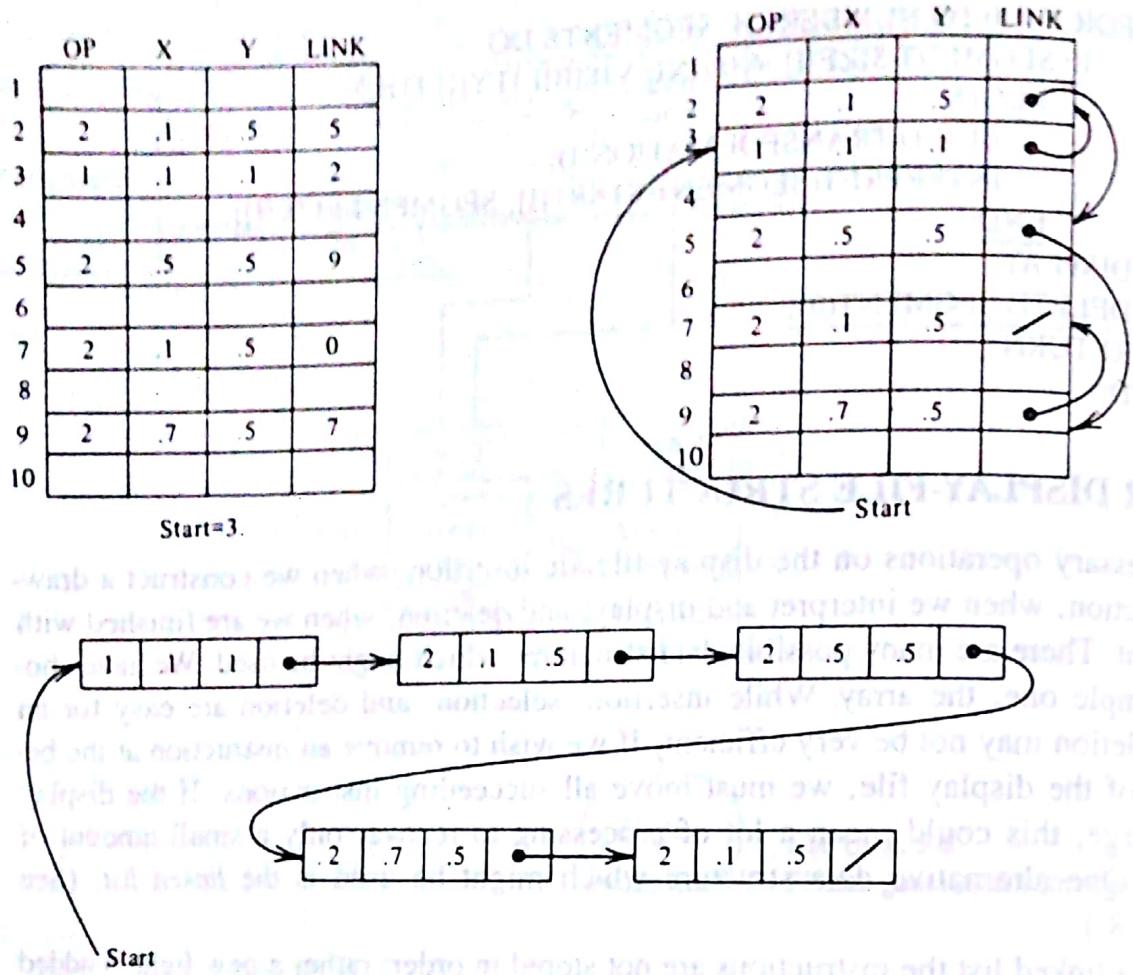

FIGURE 5-8
The display file as a linked list (drawn three different ways).

When a new instruction is added to a display file, a cell is obtained from the list of available space, the correct instruction operation code and operands are stored, and the cell is linked to the display-file list. Deletion of cells from a linked list is very easy. To remove a cell, we need only change the pointer which points to that cell so that it points to the succeeding cell. (See Figure 5-9.)

We could also link the cell which we have removed to the list of available space. In the linked list scheme, deleting the cell means changing two links. This can be much more efficient than moving a number of instructions, as we did for the array. The disadvantages of the linked list scheme are that it requires more storage to hold the links and that it is costly to locate arbitrary cells.

A third scheme, which is between the array and linked list methods, is a *paging scheme*. In this method the display file is organized into a number of small arrays called pages. The pages are linked to form a linked list of pages. Each segment begins at the beginning of a page. If a segment ends at some point other than a page boundary,

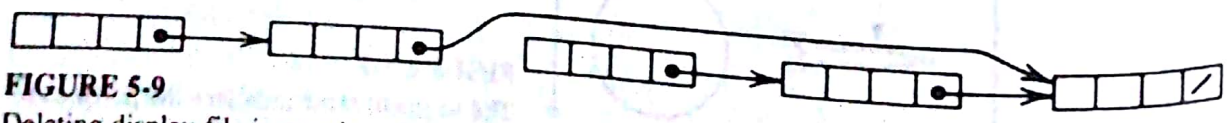

FIGURE 5-9
Deleting display-file instructions from a linked list.

then the remainder of that page is not used. In this scheme, display-file instructions can be accessed within a page just as they were accessed in an array. When the end of a page is reached, a link is followed to find the next page. (See Figure 5-10.)

By grouping the instructions into pages, we have reduced the number of links involved, yet we are still able to delete a segment by simply altering links. A list of unused or available pages provides a source of new pages when the display file is extended, and deleted pages may be returned to this list for reuse. Some disadvantages of this scheme are that storage is lost at the end of the page if a segment does not completely fill it and that accessing is a bit more complex.

In the above discussion, once a display-file segment has been closed, it can no longer be altered. There is no way to replace display-file instructions, and once closed, the segment cannot even be extended. While we have built this restriction into our system, other systems might allow such operations (although some questions may arise, for instance, whether modification commands should follow the current attributes, such as line style and character spacing, or those that were in effect in the original segment at the point of modification). When editing of the display file is allowed, a linked structure may be much more natural than our array scheme. The extension of our array scheme is considered in Programming Problem 5-12.

There are many other possible storage schemes besides the three we have mentioned. We have tried to isolate access to the display file in the routines GET-POINT and PUT-POINT, although the segment deletion routine also depends on the display-file structure. We have organized our program in this way so that different display-file data structures might be employed with a minimum of alteration to the algorithms.

SOME RASTER TECHNIQUES

In the graphics package we are developing, the only method available for altering an image is to change the display file or image transformation and reinterpret the entire picture. This method works well for storage tube and vector refresh displays, but may prove inefficient for raster displays, where clearing and recomputing the pixel values for the entire frame buffer can be costly. The principle behind fast modification of a raster display is to change as little as possible. Special techniques have been developed for raster displays which allow altering a portion of the display while leaving the re-

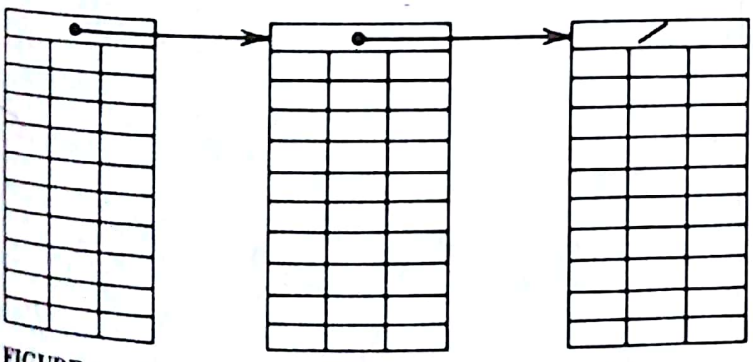

FIGURE 5-10
Linked pages of display-file instructions.

mainder of the frame buffer intact. For example, suppose we wished to make one segment invisible while maintaining the current status on all remaining segments. Instead of clearing the frame buffer and redrawing the entire picture, we might just redraw the segment which we wish to make invisible, only in this drawing use a background value for the setting of each pixel. This in effect erases all lines which were previously drawn by the segment. The segment would be removed without clearing the entire display. This technique could leave gaps in lines belonging to other segments if the segments shared pixels with the invisible segment (for example, points where a line from the invisible segment crosses a line from the visible segment). This damage could be repaired by reinterpreting the segments which are still visible. Again this can be done without first clearing the frame buffer. (See Figure 5-11.)

Another frame buffer technique can sometimes be used for efficient translation of an image. The image may be moved from one portion of the screen to a different portion of the screen by simply copying those pixels involved in the image from one position in the frame buffer to their new position. If the image is confined to a box, then only those pixels within that box need be copied. Pixels outside can be left unchanged. This could be much more efficient than setting all pixels in the frame buffer to their background value and recomputing the pixel settings for the translated image. (See Figure 5-12.)

There is an operation called a *RasterOp* or *bit block-transfer (BITBLT)* which can be quite useful in working with raster displays. The idea is to perform logic operations on bit-addressed blocks of memory. The BITBLT works on subarrays of the frame buffer. It performs simple operations on subarrays (such as turning all pixels on or off, shifting all values by a row or a column, and copying values from another subarray); and for one-bit-per-pixel frame buffers, it forms the logical AND, OR, or EXCLUSIVE-OR of pixel values in two subarrays. For example, we might use a BITBLT to copy a rectangular area of the display to some other area on the screen. Or we might use the BITBLT to clear a portion of the display. We have already introduced

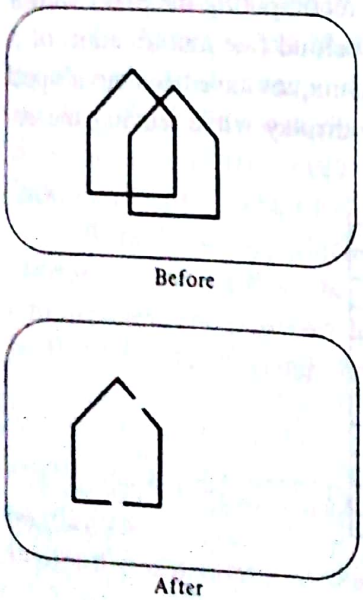

FIGURE 5-11
Removing an image by redrawing with the background.

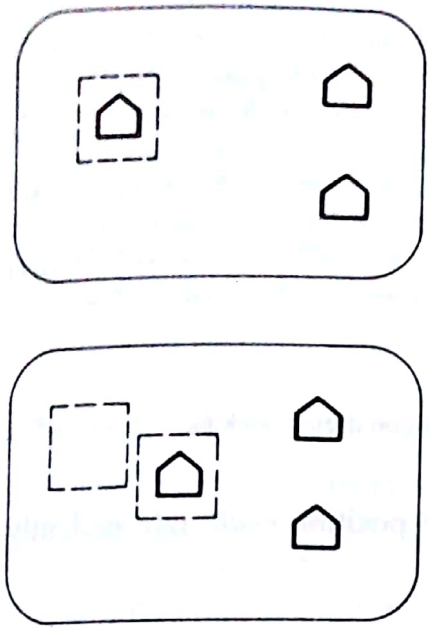

FIGURE 5-12
Alter only the pixels contained by the box.

this idea in connection with character generation. Pixel values for a character may be copied into a subarray of the frame buffer from some fixed template. BITBLT operations may be implemented in hardware so that they are very fast. (See Figure 5-13.)

AN APPLICATION

Let us suppose that a computer graphics display system is to be used to aid in the docking of a large ship. The relative position of the ship and the dock (measured by the ship's sensors) is to be presented graphically by showing an image of the ship and dock as they might be seen from above. How might such a program be written in our system? We can use our LINE and POLYGON commands to generate images for the dock and for the ship. If the instructions for these images are placed in different display-file

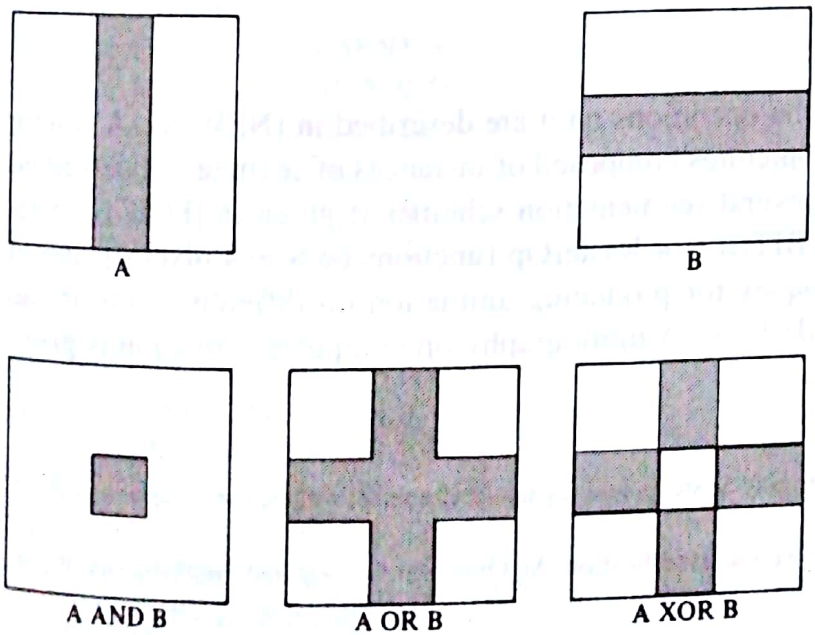

FIGURE 5-13
Logical operations on scenes.

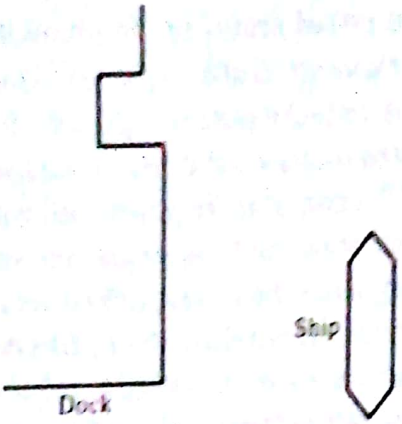

FIGURE 5-14
Graphical display of ship's position during docking.

segments, then we can use the image transformation to position each independently. (See Figure 5-14.)

```
CREATE-SEGMENT(1);
draw the ship
CLOSE-SEGMENT;
CREATE-SEGMENT(2);
draw the dock
CLOSE-SEGMENT;
```

Now we need a loop to repeatedly update the ship's position on the display. This loop will obtain the position from the ship's sensors and determine the rotation and translation parameters required. These parameters are used in the image transformation to place the ship and dock in their correct positions on the display. Of course, if the ship's position has not altered since the last check, then no updating is necessary. We can extend this program to handle docking at several ports-of-call by entering, in separate segments, the shape of the dock at each port. We then make the dock at the current port-of-call visible, while all the others are made invisible.

FURTHER READING

A segmented display file and the operations on it are described in [NEW74]. A system which builds pictures from subpictures composed of instances of segments is described in [JOS84]. An overview of several segmentation schemes is given in [FOL76]. Special hardware which supports BITBLT or RasterOp functions on 8×8 pixel squares is described in [SPR83]. Techniques for producing animation on different hardware architectures are described in [BAE79]. A bibliography on computer animation is given in [THA85].

[BAE79] Baecker, "Digital Video Display Systems and Dynamic Graphics," *Computer Graphics*, vol. 13, no. 2, pp. 48–56 (1979).

[FOL76] Foley, J. D., "Picture Naming and Modification: An Overview," *Computer Graphics*, vol. 10, no. 1, pp. 49–53 (1976).