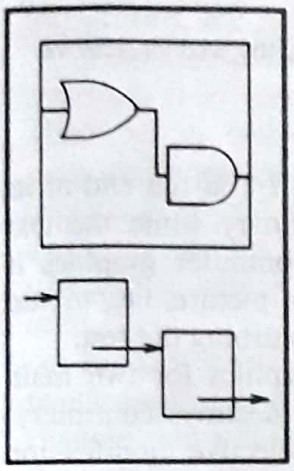


7

SEGMENTED DISPLAY FILES

Dynamically changing displayed pictures appeal to everyone involved in computer graphics—user, programmer, system designer, or hardware expert. They attract because they are so unlike the static pictures we habitually draw on paper, because they suggest all sorts of intriguing uses, and because their creation is in itself such a challenging task. There is ample evidence of this attraction in the way the very first computer-graphics research workers immediately tackled the problems of dynamic computer graphics, ignoring until much later the simpler but equally useful subject of static computer graphics.

The graphics package we have discussed in Chapter 6 is basically unsuited to the display of dynamically changing pictures. One reason must at once be obvious to anyone who has sat in front of a storage-tube display. It is simply not possible to change the displayed picture fast enough to produce a smooth transition from one picture to the next. The lines appear one after the other; there is a short pause, followed by a brilliant green flash, and the picture vanishes; then the lines of the next picture start to appear.

Thus dynamic graphics demands *speed of regeneration* of successive pictures. Before we rush to invent ways of achieving this speed, however, we should consider what else is needed. Applications of computer graphics sometimes require that the complete picture be redrawn at each change, just as described; more frequently, however, only a small part of the picture changes, and the rest remains unchanged. For example, we might like to use a light pen

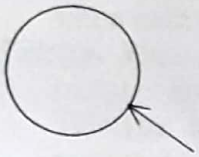


Figure 7-1 Positioning labels, illustrating the need for selective modifications.

or tablet to position the dimension of the hole in Figure 7-1 at the end of its arrow. Here it is essential that the arrow remain stationary while the text moves. The second major requirement for dynamic computer graphics is therefore the ability to make *selective modifications* to the picture, i.e., to add new parts, move them around, and delete them without disturbing the rest.

The storage-tube display is unsuited to dynamic graphics for two main reasons: First, the green flash that accompanies erasure destroys continuity; second, storage tubes cannot provide the full range of selective modification functions needed for dynamic graphics. The three basic functions we require are addition, replacement, and deletion of information. The storage-tube display can provide only the addition function; the other two require erasing the screen and redrawing the entire picture.

As we have seen in Chapter 3, the use of random-scan refreshed CRT displays provides all the characteristics we require for dynamic graphics. The picture is maintained on the screen by repeated regeneration from a stored *display file*, and this display file can be modified selectively and rapidly; no green flash accompanies modifications. Other types of display, such as raster-scan CRT displays and plasma panels, provide what is called *selective-erase* capability; portions of the picture can be deleted by retracing them, i.e., by redrawing them at the opposite intensity level. Since this is not as efficient as display-file modification for performing simple dynamic operations, we shall confine our discussion in this chapter to the random-scan CRT display.

7-1 SEGMENTS

It is easy to form the impression that the selective modification capability is all that is needed to achieve dynamic computer graphics. This impression is mistaken: it is like believing, as many did in the early 1960s, that to acquire a graphics display is to ensure its immediate and profitable use. Programming, as well as hardware, is involved. Before the selective-erase capability can be used, we must solve the question of *what* is erased: how do we provide the programmer with the ability to control which parts of the picture change and which parts stay still?

The first stage in solving this problem is obvious: we provide the programmer with the means to *name* the different parts of the picture. Having

achieved a naming mechanism, we then furnish him with subroutines or functions that effect the desired modifications on the appropriate parts. The display file, provided it resides in addressable memory, possesses the two fundamental attributes we require: we can assign names or labels to sets of instructions, and these instructions are easily modified.

How should we choose our sets of display instructions for naming? One approach is to name each graphic primitive that we add to the display file. There is no reason, however, why these primitive entities should be a particularly appropriate choice; in almost all cases the unit of modification consists of several if not many such entities. So great, indeed, are the potential classes of pictures and of dynamic modifications that we cannot expect a single unit size to suffice. Instead we must let the programmer determine his own units.

In this chapter we shall discuss a simple approach to display-file manipulation. It is by no means the only such scheme, as we shall see in later chapters, but it is a widely accepted approach that is particularly appropriate to the concepts developed in the previous chapter. We shall use the term *segment* for the unit of picture that is named for modification purposes. The segment is a unit of the *display file*; thus the use of segments by the programmer requires him also to recognize the existence of the display file as a stored representation of the displayed image.

The segment is a *logical* unit, not necessarily contiguous either in the display file or on the screen. It is simply a collection of display-file instructions representing graphic primitives that we can manipulate as a single unit. We therefore need functions to perform these manipulations, and a naming scheme so that we can refer to each segment unambiguously.

7-2 FUNCTIONS FOR SEGMENTING THE DISPLAY FILE

In Chapter 6 we raised the notion that primitive graphic functions draw lines and text directly on the screen. We now modify that notion slightly by saying that they deposit display commands in a display-file segment representing the same lines and text. This slight modification to our model of the display process provides the basis for a set of functions for display-file manipulation.

Many readers will be familiar with the common methods of creating and modifying sequential disk files. We *open* a file before we add data to it, and we *close* the file when we have added the last data item and the file is complete. To change the contents of a file, we open it again, add the new data to replace the old, and close the new file. To get rid of a file, we *delete* it.

The very same operations are ideal for manipulating display-file segments. To create a new segment, we open it and then call graphic primitives to add to the segment the lines and text to be displayed; then we close the segment. The same sequence of operations applied to an existing segment will cause that

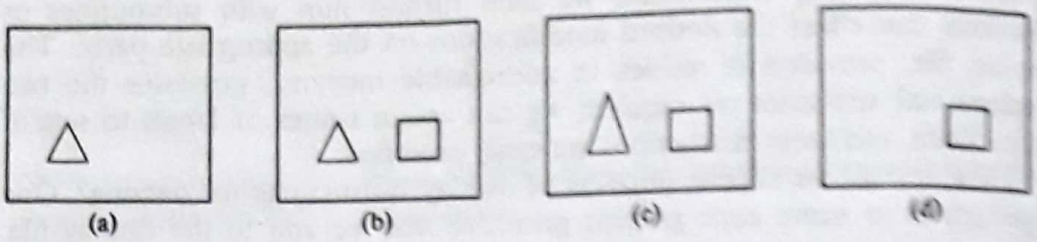


Figure 7-2 Manipulation of triangle and square as separate segments.

segment to be replaced by a new one. To remove a segment from the display file, we delete it. Thus we need only three basic functions:

OpenSegment(*n*)	open a display file segment named *n*
CloseSegment	close the open segment
DeleteSegment(*n*)	remove from the display file the segment named *n*

To illustrate the use of these functions, suppose we wish to display a triangle, as shown in Figure 7-2a. We may use the following statements:

```
InitGraphics;
SetWindow(0, 0, 500, 500);
OpenSegment(t);
MoveTo(100, 100);
LineTo(150, 200); LineTo(200, 100); LineTo(100, 100);
CloseSegment;
```

The display file, if originally empty, now contains the single segment *t* (Figure 7-3a). We can add a square, as shown in Figure 7-2b, as follows:

```
OpenSegment(s);
MoveTo(300, 100);
LineTo(300, 200); LineTo(400, 200); LineTo(400, 100); LineTo(300, 100);
CloseSegment;
```

Now the display file contains the two segments *s* and *t*, as in Figure 7-3b. We can redefine the triangle as follows, to achieve Figure 7-2c:

```
OpenSegment(t);
MoveTo(100, 100);
LineTo(150, 250); LineTo(200, 100); LineTo(100, 100);
CloseSegment;
```

The original segment *t* has been replaced by a new one (Figure 7-3c). Note that we would like the original triangle to remain on the screen until the definition of the new one is complete. We shall encounter this requirement again in Chapter 8 when we discuss double buffering.

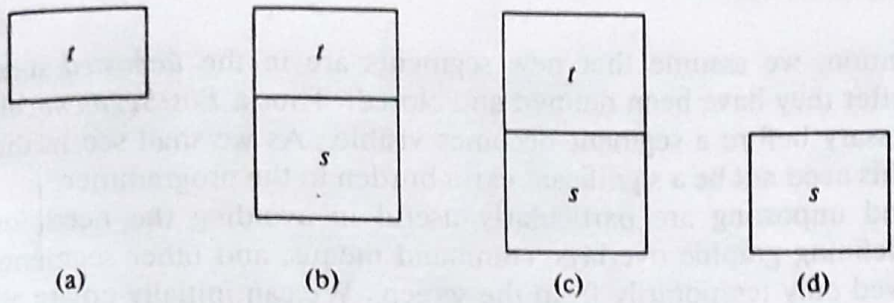


Figure 7-3 Display file segments corresponding to Figure 7-2.

Finally we can erase the triangle (Figure 7-2d):

DeleteSegment (*t*)

The display file now contains only the single segment *s*, as shown in Figure 7-3d.

This is a minimal function set for manipulating segmented display files. The design of the functions shows the influence of the standards of simplicity, completeness, and robustness raised in Chapter 6. Thus there are no more functions than necessary, just enough to do what we want. Two different *OpenSegment* functions could have been provided, one for creating segments and one for replacing existing ones, but this would be capable of misuse — the programmer could try to open a new segment with the same name as an old one.* We could also have required an argument to be passed to *CloseSegment*, to indicate the name of the segment to be closed. This is unnecessary, however, since we can almost always construct display files without opening more than one at a time; thus *CloseSegment* closes the currently open segment and needs no argument.

7-3 POSTING AND UNPOSTING A SEGMENT

We can extend the usefulness of our graphics package considerably by allowing segments to become temporarily invisible. For this we need two additional functions. *Posting* is the action of including a segment in the display refresh cycle; thus posting makes a segment visible. *Unposting* removes the segment from the refresh cycle, so that it is no longer visible. The segment is not destroyed by unposting, and so it can be rendered visible again without redefining it, simply by posting it. The two functions to achieve this effect are

PostSegment (*n*)	add segment *n* to the refresh cycle
UnpostSegment (*n*)	remove segment *n* from the refresh cycle

*File systems generally provide two separate functions here, partly to safeguard against accidentally overwriting a file. Such precautions are not necessary with display-file segments.

By convention, we assume that new segments are in the *unposted* state immediately after they have been defined and closed. Thus a *PostSegment* call is always necessary before a segment becomes visible. As we shall see in the next section, this need not be a significant extra burden to the programmer.

Posting and unposting are particularly useful in avoiding the need for repeatedly redefining graphic overlays, command menus, and other segments that are removed only temporarily from the screen. We can initially create all the menu or overlay segments and then switch between them by simply unposting the old segment and posting the new one.

7-4 SEGMENT NAMING SCHEMES

The most common scheme for naming segments is to provide each segment with a *unique integer name*. This is a departure from normal file-system practice, where text strings are more popular as names. Segment names are used only within the program, however, so they are essentially *internal names* unseen by the user of the program. Integers are not only more compact and more easily manipulated than strings but often more convenient as a means of relating segments of the display file to the items of data they represent.

Suppose, for example, we wish to use a graphical display in an air-traffic control center. We might present the air-traffic controller with a display like that shown in Figure 7-4, in which small symbols represent individual aircraft. From signals received in real time, a computer program maintains a data structure containing the identity, airspeed, heading in degrees, position, and altitude of each aircraft; Figure 7-5 shows a suitable list structure. Each aircraft symbol, together with the text labels showing identity, airspeed, and altitude, may be displayed as a single segment, greatly speeding up the task of modifying the picture. An appropriate name for each segment is then the index, in the main list, of the pointer to the vector defining the aircraft in the data structure. In this way we can easily determine the name of the segment representing any

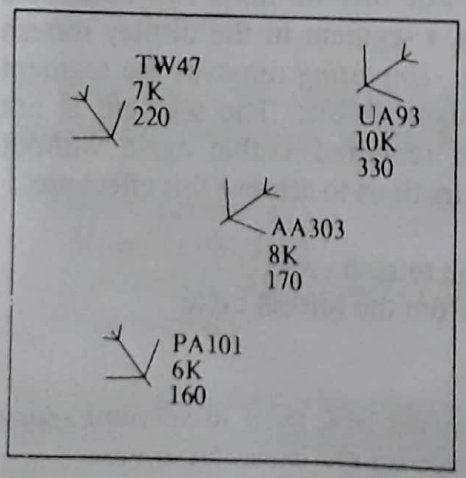


Figure 7-4 An air traffic control display.

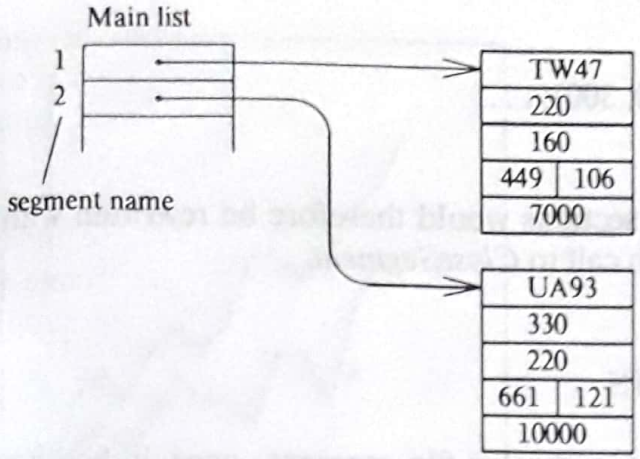


Figure 7-5 List-structure representation of air traffic control information.

particular aircraft in the data structure without using space in the data structure to define it.

7-5 DEFAULT ERROR CONDITIONS

Quite clearly, the programmer can make any one of a number of programming mistakes with the five functions *OpenSegment*, *CloseSegment*, *DeleteSegment*, *PostSegment*, and *UnpostSegment*, by issuing function calls in peculiar orders. For example, he can forget to close a segment, and then try to open another; or he can attempt to delete a nonexistent segment.

There are several solutions to this class of problem. It would obviously be quite simple and satisfactory to generate an error message on each such occasion. Error messages are often superfluous during display-file construction, however. Just the visible evidence on the screen is usually enough to indicate that the display file has been constructed incorrectly. Consequently there is no need to do any more than perform the following default actions:

OpenSegment (*n*)	any currently open segment is closed
CloseSegment	if no segment is currently open, no effect
DeleteSegment (*n*)	any currently open segment is closed; if segment *n* does not exist, no further effect
PostSegment (*n*)	any currently open segment is closed; if segment *n* is nonexistent or already posted, no effect
UnpostSegment (*n*)	any currently open segment is closed; if segment *n* is nonexistent or already unposted, no effect

Not only do these default actions take care of the more common mistakes of omission, but they simplify programming too. It is no longer necessary both to close and to post a segment in order to make it visible; posting alone will suffice. Normal practice, therefore, is to use the following sequence to create a new, visible segment:

```
OpenSegment(s);
MoveTo(200, 200);
LineTo(200, 300); LineTo(300, 300); . . .
PostSegment(s);
```

The examples given in the earlier sections would therefore be rewritten with a call to *PostSegment* in place of each call to *CloseSegment*.

7-6 APPENDING TO SEGMENTS

It is sometimes inconvenient that a display-file segment, once it has been defined, cannot be incrementally modified but must always be completely rebuilt. Consider, for example, a program to display the results of two concurrent simulations in the form of two graphs plotted on the same axes, as shown in Figure 7-6. Every 2 minutes each simulation generates a fresh data point, and the graphs are extended.

One method of writing such a program would be to create a new segment for each addition made to either graph. This would be extremely inefficient, however, since a complete simulation might involve several hundred data points. On the other hand, any alternate approach, such as the use of a single segment for each graph, could involve reconstructing the entire segment after every step.

In situations like this, a function for appending to segments is useful:

AppendToSegment(*n*)	open segment *n* for additions, leaving its present contents intact

Thus to add a line to a graph represented by segment *g* and leave the segment posted, we might write the following statements:

```
AppendToSegment(g);
MoveTo(lastx, lasty);
LineTo(newx, newy);
PostSegment(g);
```

The *AppendToSegment* function, although often useful in writing application programs, has some troublesome side effects. In view of these problems, which are discussed in the next chapter, the *AppendToSegment* function may involve more trouble than it is worth. In a few cases, like the example given, the lack of such a function will be felt by the application programmer; it is usually possible, however, to solve the problem by careful use of segments. Thus in the simulation example we can use a separate segment for each sequence of *n* data points, where *n* is in the range 20 to 50. We use a separate segment to represent each recent addition to either graph and redraw

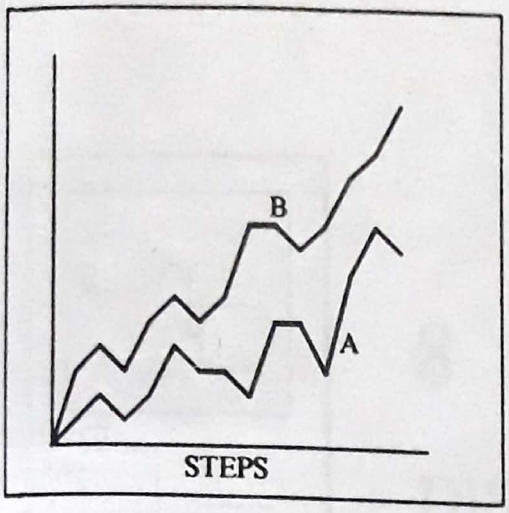


Figure 7-6 Plotting the results of two concurrent simulations.

these additions as a single segment each time the total number of the recent segments reaches *n*. Techniques of this kind are an indication of the complexity that can occasionally be avoided with the provision of an *AppendToSegment* function.

EXERCISES

7-1 What should the default error action be if the application program should call primitive functions without first opening a segment?

7-2 Suppose we wished to be able to manipulate individual display primitives in the same manner as segments: what changes would be necessary to the graphics functions described in Chapters 6 and 7?

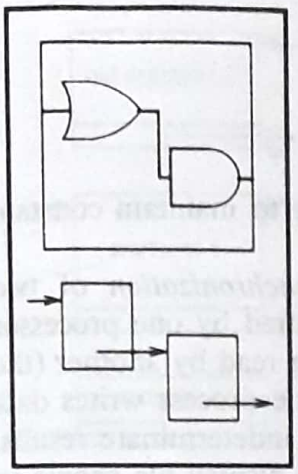
7-3 What would be the advantages and disadvantages of permitting more than one segment to be open at a time?

7-4 Although it is almost universal to use integers as segment names, it is feasible to use other types of data, such as strings and floating-point numbers. Can you think of reasons for using such names?

7-5 Rather than divide the displayed image logically into segments, we could divide it physically, e.g., into small squares. What effect would this have on interactive programming?

7-6 As we have seen, it is possible to provide two functions for opening segments, one to create a new segment, the other to redefine an existing segment. Does such a scheme have advantages? Illustrate your answer with some short program sequences.

7-7 Some graphics packages implement posting and unposting as manipulation of an *attribute* of the segment, i.e. changing its visibility. Can you think of other useful segment attributes?



8

DISPLAY FILE COMPILATION

It is by no means as easy to construct the display-code segments for a refresh display as it is to generate code for a storage-tube terminal. Since the display code must be stored for refresh, memory must be allocated for its storage; the space it consumes must be kept to a minimum, and care must be taken to ensure that it can be refreshed as fast as possible to reduce flicker. Furthermore, we would often like to reconstruct segments repeatedly in order to achieve dynamic picture changes; this suggests that we attempt to construct segments as fast as possible. All in all, the goals in display-file construction are very similar to those in compiling source programs into machine code. This has led to the use of the term *display-file compiler* for the refresh display-code generator. In this chapter we shall consider some of the issues in display-file compilation.

8-1 REFRESH CONCURRENT WITH RECONSTRUCTION

A display that periodically blinks or goes blank for a second or two is very annoying: it prevents the user from concentrating on the task at hand. Therefore we should try to construct our display-file compiler to ensure *uninterrupted refresh*. We should never stop the refresh process, even when we are reconstructing part of the display file, since a blink of even a fraction of a

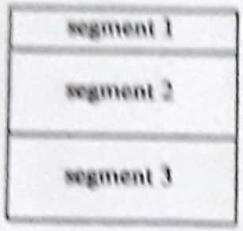


Figure 8-1 A three-segment display file.

second is noticeable to the user. Instead we should try to maintain constant refresh during all display-file modifications.

The problem thus raised is a classic one of *synchronization* of two concurrent processes. As the display file is being modified by one processor (the CPU executing the application program), it is being read by another (the display processor). This kind of operation, in which one process writes data while another process reads them, in general produces indeterminate results. For example, we might try to change segment 3 of the display file shown in Figure 8-1, simply by overwriting it with the new display code. Halfway through, the display processor might catch up with the display-file compiler, executing first an instruction from the new display segment and then one from the old segment. At best, the picture might appear momentarily to break up in a strange fashion; at worst, the display processor will be directed to the wrong part of memory, perhaps destroying the contents of certain vital memory locations.

We use the term *corruption* of the display file whenever the display file fails to represent a valid sequence of instructions for the display processor. A simple way to avoid corruption is to use *double buffering*. We construct the new segment in an unused area of memory, leaving the old version in the display file. When the new version is closed and posted, it takes the place of the old version and the vacated space is free for future use. Double buffering obviously creates discontinuities in the display file, which may eventually look something

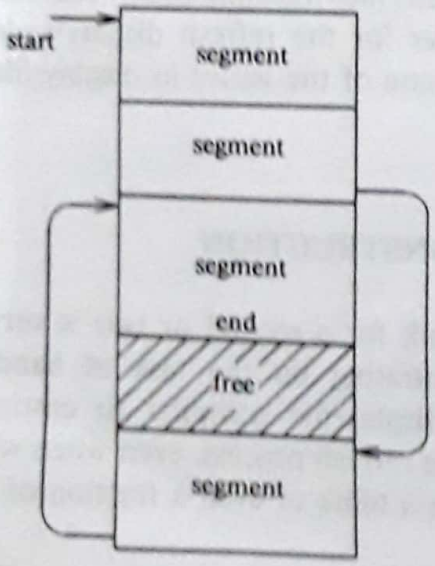


Figure 8-2 Discontinuities in the display file caused by double buffering.

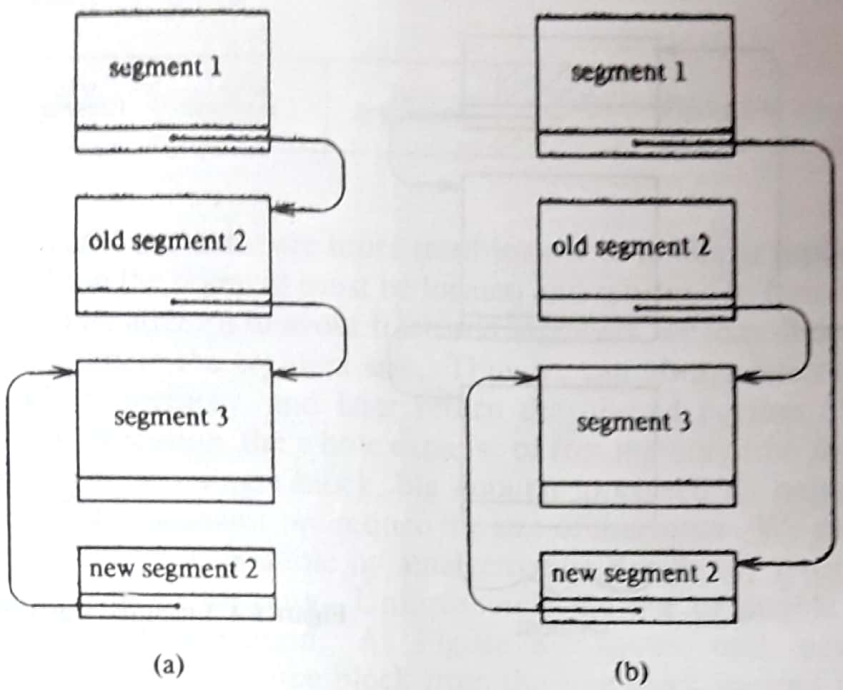


Figure 8-3 Achieving a smooth transition in double buffering.

like Figure 8-2. Here each segment is followed either by the next consecutive segment or by a pointer (or jump instruction) to it. For generality, it is best if all segments terminate with a jump instruction that may simply address a segment starting at the next consecutive memory location.

There is a critical moment in the double-buffering operation when the new and old segments are exchanged. This must be done in the sequence shown in Figure 8-3. The new segment should be complete, including its final jump instruction pointing to the next segment in the display file, as shown in Figure 8-3a. Then the jump instruction in the preceding segment can be switched to point to the new segment, as in Figure 8-3b. In this way a smooth transition is ensured and corruption of the display file is prevented.

8-2 FREE STORAGE ALLOCATION

As we have just seen, the display-file compiler at frequent intervals needs blocks of unused memory in which to construct new segments. Just as frequently it discards blocks of memory for which it has no further use. A *free-storage allocation system* is therefore needed to supply blocks of free memory and to receive blocks that are vacated.

A display-file compiler makes somewhat unusual demands on a free storage allocation system. Three requirements are of particular importance:

1. The *amount* of memory required is unknown at the time of allocation; free

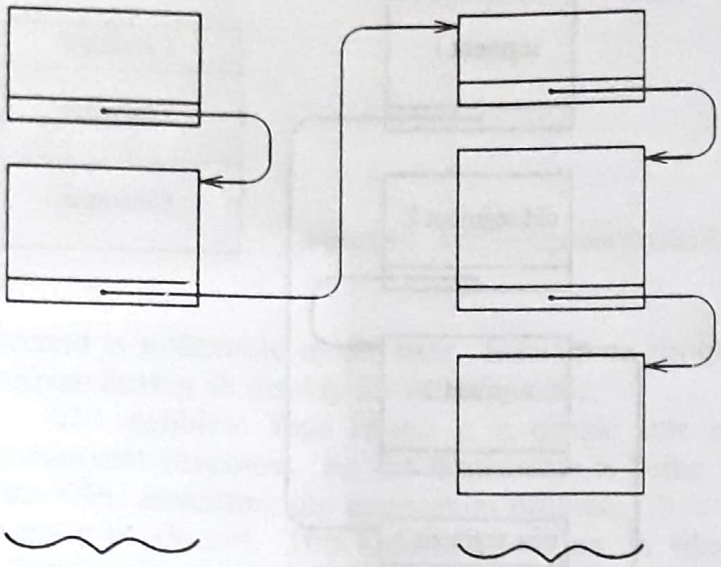


Figure 8-4 Fractured segments.

storage is needed at the moment we open a segment, but the amount needed is not known until the segment is closed.

2. *Speed of allocation* is important since any sizable delay adversely affects the program's response.
3. *Blocks cannot immediately be reused* after they become free. Here again, we encounter a concurrency problem. If we reuse a block immediately, the display processor may still be executing instructions within it, and if we overwrite these instructions, we may corrupt the display file.

Although we do not know how much memory we need when a new segment is opened, it is always possible to make an estimate. In particular, when a segment is replaced, the new segment may be assumed to be approximately the same size as the old one. If we overestimate, we can simply return the unused memory to free storage. The situation caused by underestimating is less simple. We must fetch another block of free storage to accommodate the rest of the segment and include a jump instruction at the end of the first block pointing to the second block. This creates a *fractured segment*, which, as shown in Figure 8-4, may even contain three or more separate blocks.

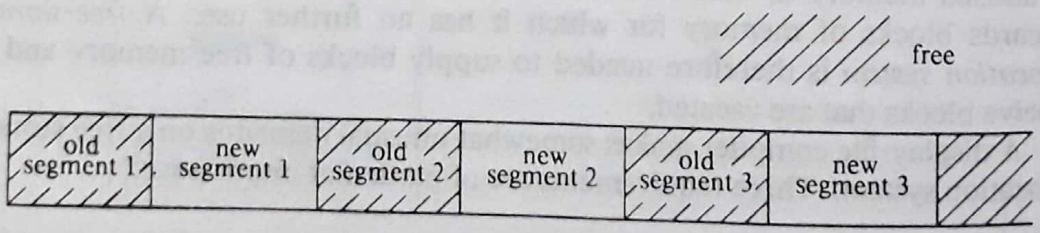


Figure 8-5 Fragmenting effect on free storage of double buffering.

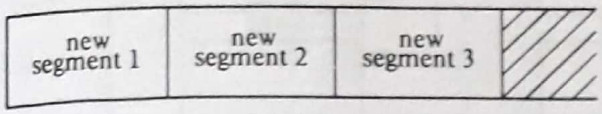


Figure 8-6 Amalgamating free blocks by garbage collection.

Fractured segments are more troublesome to delete or replace since each of the blocks in the segment must be located and returned to free storage.

In an attempt to avoid fractured segments, we may choose instead always to overestimate the segment size. Thus we can always fetch the *largest* available block of memory, and later return the unused portion. When the program begins execution, the whole expanse of free memory (the *free-storage area*) is in the form of a single block, big enough to exceed all requests. Each time we construct a segment, we reduce the size of this block. We can try to maintain as large a block as possible by amalgamating it with any returned blocks that are contiguous in memory. Unfortunately the use of double buffering tends to prevent amalgamation. As Figure 8-5 shows, each new segment usually separates the main free block from the free block vacated by the old segment. Thus the main block becomes gradually smaller, until it is eventually too small to accommodate even one segment. Meanwhile many noncontiguous free blocks are scattered throughout the free storage area. We call this *fragmentation* of the free-storage area.

Fragmentation can be cured in one of two ways: by allowing segments to become fractured, hence permitting the use of fragments of any size; or by collecting all the fragments together into a single free block, by means of a process called *garbage collection*. To perform garbage collection we move all the blocks of display file toward one end of the free storage area, thus filling in all the unused fragments and creating a single free block at the other end (Figure 8-6). When each block is moved, all jump instructions or pointers to the block must be adjusted. Garbage collection may take a second or more to carry out; it is difficult to perform without stopping the display, which then blinks noticeably.

A simple approach to free-storage allocation is the use of *fixed-size blocks*. At the start we divide the entire free-storage area into blocks of, say, 32 words each. Segments are then constructed by chaining these blocks together. Any unused space at the end of the last block of a segment remains empty, rather than being returned to free storage. This method is extremely simple but somewhat wasteful because of these unused words. Furthermore, the use of fixed-size blocks is not always feasible, since free storage may be needed by other parts of the application program in blocks of other sizes. In such cases, the use is recommended of the *boundary-tag* allocation scheme described by Knuth [268].

Whichever allocation system we use, we must take care when blocks are released to delay their reuse until they are no longer being read by the display processor. The vacated blocks should be placed on a temporary free-storage list that is added to the main list o ly after the end of the refresh cycle has been

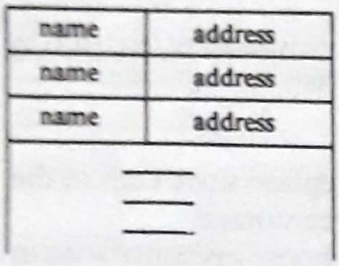


Figure 8-7 Use of a vector as a name table.

reached. If the main list is completely exhausted, we may cause the program to wait until completion of the refresh cycle makes the temporary list available.

It is of course possible to exhaust all sources of free storage. This creates an error condition that should be handled gracefully by the graphics system. The currently open segment should be closed and all further graphic primitive-function calls should be ignored. For the benefit of application programmers who wish to handle this situation by deleting other segments or allocating more memory to free storage, the *Inquire* function should provide information about the amount of free storage available, and if free storage should be exhausted during compilation of a segment, the *Inquire* function should provide the name of the segment so that it can later be regenerated. A well-designed application program will inform the user when memory becomes scarce, advising him what steps to take.

8-3 DISPLAY-FILE STRUCTURE

Before we can implement segmenting functions, we must design the format of the display file. Display processors are frequently designed with little thought for ease of programming; sometimes it is very difficult to implement the set of functions for manipulating display-file segments. This is particularly true if the display processor possesses neither jump instructions nor a display subroutine

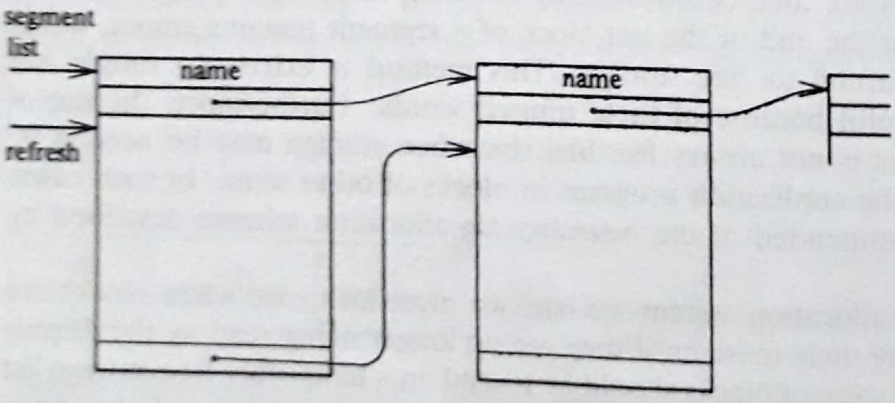


Figure 8-8 Names stored in segments, with linked-list pointers and jump instructions.

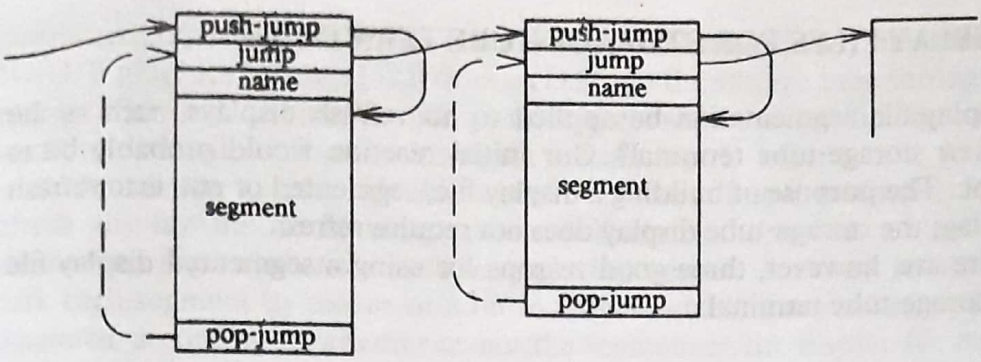


Figure 8-9 Use of push-jump and pop-jump instructions to achieve single-pointer segment linkage.

mechanism. The following discussion of display-file organization will show how useful these two hardware features can be.

An essential part of the display file structure is the *name table,* which allows us to determine a segment's address in memory from its name. The name table may be stored in a vector, as shown in Figure 8-7, with two fields per entry, one for the name, the other for the segment address. A third field may be added containing the length of each segment to assist in garbage collection. A linear table like this can be searched rapidly for a given name and can be used in conjunction with most kinds of display processor. Its main drawback is that its size determines the maximum number of segments permitted in the display file unless provision is made to enlarge it when it fills up.

An alternate scheme is to store the segment name at the head of the segment itself, as shown in Figure 8-8. Space for names is then exhausted only with the exhaustion of free storage itself. We can locate any given segment by stepping along the linked list of segments. To speed up this search, we should include in the head of the segment a pointer to the next segment; this pointer is essentially a copy in a more easily located place of the jump instruction that terminates the segment.

Figure 8-9 shows a modification of this design, using push-jump and pop-jump instructions. The first instruction in each segment is a push-jump to the fourth instruction, followed by a jump to the next segment and the segment name. The push-jump directs the display over the following two words to the first word of display code. The pop-jump that terminates the segment returns the display processor to the jump instruction, which in turn directs it to the next segment. By eliminating the duplication of pointers, this scheme simplifies garbage collection and makes the display-file compiler more robust. Posting and unposting are very simple. A segment is unposted by copying into the first word the jump instruction contained in the second; it is posted by overwriting the contents of the first word with a push-jump to the fourth. This scheme has some further advantages when used with a light pen, which we shall explore in Chapter 13.

8-4 DISPLAY FILES FOR STORAGE-TUBE TERMINALS

Can display-file segmentation be applied to nonrefresh displays, such as the direct-view storage-tube terminal? Our initial reaction would probably be to guess not. The purpose of building a display file, segmented or not, is to refresh the display; the storage-tube display does not require refresh.

There are, however, three good reasons for using a segmented display file with a storage-tube terminal:

1. *To allow display regeneration without recomputation.* We often need to regenerate the display after a minor deletion. This could involve re-executing the parts of the program that generated the display in the first place. Thus the picture must be recomputed, retransformed and reclipped and the display code regenerated. None of this is necessary if we save the display code each time it is generated.
2. *To reduce the frequency of display regeneration.* If part of the picture is deleted, the display must be regenerated to show the effect. On the other hand, if more lines are added to the picture, no regeneration is necessary. It is tiresome for the programmer to have to remember when regeneration is required and when it is not. The method described below uses a segmented display file to ensure that the display is regenerated only when necessary.
3. *To achieve compatibility with refresh-display packages.* It is most convenient if we can run the same program unchanged on both refresh and nonrefresh displays. The addition of one function, the *Update* function, described below, makes this possible.

The display file for a storage-tube terminal consists of a number of separate segments, each containing the actual display codes to be transmitted to the terminal. As with refresh displays, a free-storage system is necessary, and we also need a name table. The methods described above for refresh displays are quite adequate. The problem of concurrency is no longer present, however, so double buffering and delayed reuse may be ignored.

Whenever a segment is closed or deleted, this display file is modified accordingly. No attempt is made to change the display, however. If we were to update the display automatically after every change, certain operations would become excruciatingly slow. For example, in order to delete five segments the user would have to wait through five complete regenerations of the display. So instead we ask the programmer to indicate when to update the screen, by calling the following function:

Update update the display to reflect the current display file contents

Thus calling *Update* after making a batch of changes in the display file will cause the appropriate segments to be transmitted to the terminal. If no segment has been redefined, deleted, or unposted, only newly posted segments are

transmitted; otherwise the screen is erased and the entire display file is sent (the GINO/F graphics package [522] draws a cross on the storage tube through each segment deleted). Typically the programmer will arrange for the *Update* function to be called before each request for user input.

Although the display file for a storage-tube terminal is simpler than a refresh display file, it requires one embellishment, a means of indicating whether any segment has changed since the last update. A simple solution is to mark each segment by means of a bit in its name-table entry either *painted* or *unpainted*, according to whether or not the segment in the display file matches its current representation on the screen. Thus immediately after a display-file segment has been defined and closed, it is marked *unpainted;* after it has been transmitted to the display, it becomes *painted;* immediately after redefining or deleting the segment becomes *unpainted* again. The *Update* function can determine whether to regenerate the display by the following algorithm:

1. If any segments are painted but unposted, or if any segment has been deleted since the most recent *Update*, erase the screen and transmit to the display all posted segments; otherwise transmit just those segments which are posted but unpainted.
2. Mark all posted segments painted and all unposted segments unpainted.

When this algorithm is employed, the effect of calling *PostSegment* and *UnpostSegment* is merely to change the setting of the segment's posted indicator. The segment's painted indicator is not changed until *Update* is called.

An issue has been raised in this section that will surface again in Part Six, namely *device independence* in graphics systems. By this is meant a system that allows programs to be used unchanged with different input and output devices. We can achieve a small but useful degree of device independence by always employing a segmented display file, whether or not we are using a refresh display, and by including an *Update* function that updates the nonrefresh display but has no effect on a refresh display. As mentioned earlier, this offers two other advantages in the case of the storage display, namely reduction in computation and fewer complete regenerations. One thing is lost, namely the ability to write huge numbers of lines onto the screen without using correspondingly huge display files. Solutions to this problem exist and will be discussed in Chapter 27.

EXERCISES

8-1 Suggest mechanisms for posting and unposting segments in the schemes illustrated in Figures 8-7 and 8-8; assume that no push-jumps are to be used.

8-2 How does the fracturing of segments affect the schemes described above?

8-3 Discuss ways in which display-processor design can simplify or complicate the task of the display-file compiler. For example, how would one cope with multiword instructions, or make use