Chapter 4

DECOMPOSITION OF A FIGURE INTO BLOCKS AND HINGES

Using runs, a figure gets converted into a series of one-dimensional structural elements. However, our goal is to recover, somehow, the two-dimensional structure of the figure. To this end, we shall partition the figure into two types of *objects* called *blocks* and *hinges*. Both are built with runs of F.

§4.1 Blocks, Hinges, d-Blocks and Block-Continuations

Any run of F that is k-adjacent to no more than one run above it and/or one run below it will be called a *block run*. Any run which does not qualify as a block run will be called a *hinge*. Formally, let $run[i,t]$ be a run of F; it is a *block run* if and only if

$$conpr[i,t] \leq 1,$$

and

$$consu[i,t] \leq 1; \qquad (4.1)$$

otherwise, it is a *hinge*.

Now, we define a *block* as a union of k-adjacent block runs. Formally, a *block* of F is a union of $1 + w$ runs ($w \geq 0$) on successive rows $i, \ldots, i + w$, say $run[i, t_0], \ldots, run[i + w, t_w]$, such that both the following conditions hold:

(i) For $j = 0, \ldots, w$, $run[i + j, t_j]$ is a block run.

(ii) If $w > 0$, then for $j = 0, \ldots, w - 1$, $run[i + j, t_j]$ is k-adjacent to $run[i + j + 1, t_{j+1}]$.

We can express these two conditions in terms of the run parameters defined in Chapter 3. Upon noting that for any $r > 0$, the rth run on any row—if it exists—has index $r-1$, it readily follows that conditions (i) and (ii) taken together can equivalently be replaced by the following three conditions taken together:

(a) $conpr[i, t_0] \leq 1$ and $consu[i + w, t_w] \leq 1.$ (4.2)

(b) If $w > 0$, then for $u = 0, \ldots, w - 1$,
$lefsu[i + u, t_u] = nrisu[i + u, t_u] - 1 = t_{u+1}.$ (4.3)

(c) If $w > 0$, then for $v = 1, \ldots, w$,
$lefpr[i + v, t_v] = nripr[i + v, t_v] - 1 = t_{v-1}.$ (4.4)

In Figure 4.1, we give an example of a figure (a) subdivided into blocks and hinges (b). One can see that blocks correspond to downstrokes or tails in the figure while hinges serve to join blocks together. It is also apparent that the topological structure of the figure can be derived from the structure of blocks and hinges, and the adjacency relations between them.

In a variety of applications it is justified to impose one further condition in the definition of a block. This additional condition is nothing but a restriction on the extent to which k-adjacent block runs may extend to the right or left of each other.

Let d be a positive integer. Then, a block is called a *d-block* if the following condition holds:

(iii) If $w > 0$, then for $j = 0, \ldots, w - 1$,
$| be[i + j + 1, t_{j+1}] - be[i + j, t_j] | \leq d,$
and
$| en[i + j + 1, t_{j+1}] - en[i + j, t_j] | \leq d.$ (4.5)

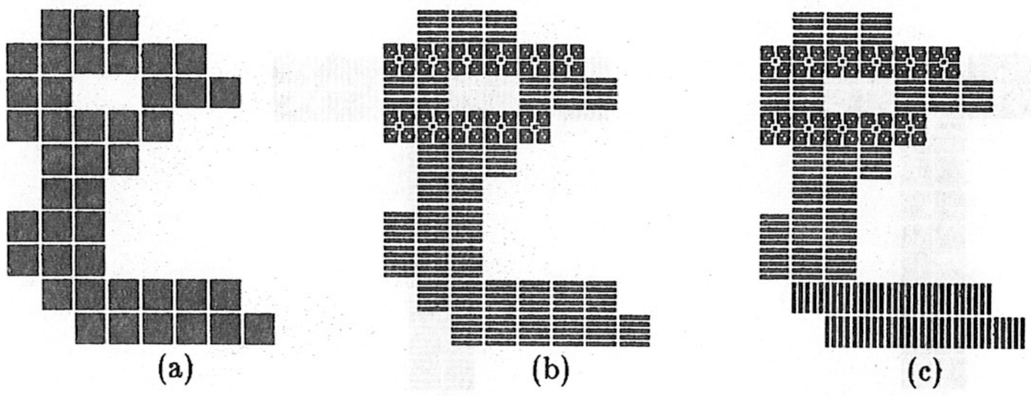

Figure 4.1. Decomposition of a figure (*a*) into hinges ▦ and maximal blocks ≡ (*b*), and maximal 3-blocks, ≡ and ||||| (*c*).

Figure 4.1.(*c*) shows the decomposition of (*a*) into 3-blocks and hinges. Figure 4.2 provides another interesting example where the letter "T", which consists of one block only, (4.2.*a*), gets decomposed in two 3-blocks, (4.2.*b*). Clearly, the two strokes are separated by the additional constraint imposed on *d*-blocks. This example suggests a first reason for using *d*-blocks instead of blocks: Some geometrical features are detected with *d*-blocks, provided a suitable value for *d* is chosen. For additional rationale behind the use of *d*-blocks, see *e.g.*, [2] on vector representation of engineering drawings, and Section 4.4 on data compression.

A block which is not contained in a larger block is called a *maximal block*. One defines similarly a *maximal d-block*. A maximal block (resp. *d*-block) is constructed from a block (resp. *d*-block) by extending it from above and below as long as this is possible.

In practical applications, blocks or *d*-blocks can have any length. However, from an operational point of view, it is frequently convenient to deal with objects of bounded size. For instance, in our implementation, and with the code written in Pascal, blocks and hinges are coded as *records* (of bounded length). With this operational constraint, a block can be coded only if its length does not exceed a given bound. If this constraint is violated, we split that block into smaller ones. The topmost one is considered as a block. The other ones are its continuations. We call them *block-continuations.*

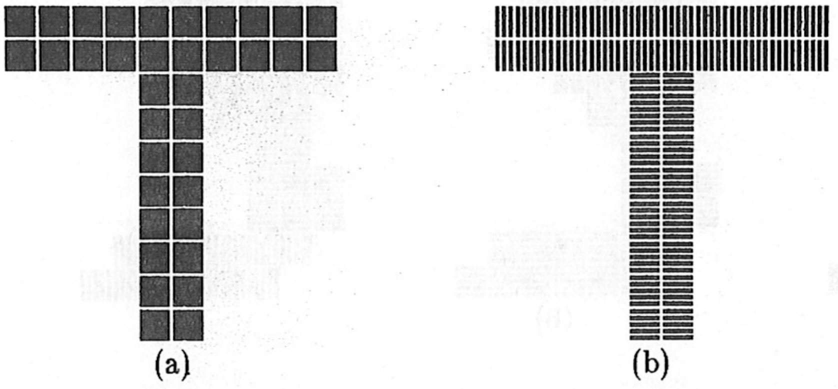

Figure 4.2. T, which is a block, gets decomposed in two 3-blocks

As stated earlier, hinges, blocks and block-continuations are what we call *objects.* To each object we associate a *type*, an integer between 0 and 3:

Objects of type 0 are hinges.

Objects of type 1 are blocks.

Objects of type 2 are block-continuations.

The unique object of type 3 is the empty set $\emptyset$. (4.6)

§4.2 Basic Properties of Blocks and d-Blocks

Let X and X' be two blocks, and Y and Y' two d-blocks. The following properties are straightforward:

(1°) A block run is a block, and a d-block.

(2°) If $X \bigcap X' \neq \emptyset$, then $X \bigcap X'$ is a block.

(3°) If $X \bigcap X' \neq \emptyset$ or X is k-adjacent to X', then $X \bigcup X'$ is a block.

(4°) X is contained in a unique maximal block X^*, which is the union of all blocks X'' satisfying the property of X' in (3°).

(5°) If $Y \bigcap Y' \neq \emptyset$, then $Y \bigcap Y'$ is a d-block.

(6°) If $Y \bigcap Y' \neq \emptyset$, then $Y \bigcup Y'$ is a d-block.

(7°) Y is contained in a unique maximal d-block Y^*, which is the union of all d-blocks Y'' satisfying the property of Y' in (6°).

(8°) The figure F is partitioned in a unique way into hinges and maximal blocks (or maximal d-blocks).

Property (8°) is illustrated in Figure 4.1.(b) and (c).

§4.3 Adjacency Relations Between Objects

Very much in the same way that adjacency relations between runs were coded by the run parameters defined in Section 3.2, adjacency relations between objects (hinges, blocks and block-continuations) will be coded by object parameters which are introduced hereafter.

Let S designate a nonvoid object with topmost run $run[i,a]$, and bottommost run $run[j,b]$. Let

$$lefpr[i,a] = c,$$

$$lefsu[j,b] = d,$$

$$conpr[i,a] = u,$$

$$consu[j,b] = v. \tag{4.7}$$

If $u > 0$, let $run[i-1,c], \ldots, run[i-1,c+u-1]$ belong to the objects $X_0, \ldots, X_{u-1}$ respectively. If $v > 0$, let $run[j+1,d], \ldots, run[j+1,d+v-1]$ belong to the objects $Y_0, \ldots, Y_{v-1}$ respectively. (Note that if S is a block, then $u, v \leq 1$, whereas if S is a block-continuation, then $u = 1$, and $v \leq 1$.) Readily, under these conditions, S is k-adjacent to the objects $X_0, \ldots, X_{u-1}$ above it and the objects $Y_0, \ldots, Y_{v-1}$ below it, and it is k-adjacent to no other object. Presently, our goal is to develop an efficient way of storing that information.

We note that it is not possible to label the objects in such a way that the labels of $X_0, \ldots, X_{u-1}$ and $Y_0, \ldots, Y_{v-1}$ form a subrange of integers as in the case of runs. So, one might think of associating to S the arrays $[X_0, \ldots, X_{u-1}]$ and $[Y_0, \ldots, Y_{v-1}]$. However, this is definitely too costly, because, *a priori*, the numbers u and v may be as large as $N/2$. Thus, we shall have to steer a more economical course.

In essence, our approach consists in attaching to S—in actual fact, to every object—a fixed number of ten parameters. The first two are the numbers (u and v) of k-connected objects above or below S. The next four are the labels of leftmost and rightmost X's and Y's k-adjacent to S. The last four serve to recover the left-to-right and right-to-left sequences of labels of X's and Y's. (See [1, p.256] for a very similar approach applied to tree-coding.)

Accordingly, using the notation just introduced, we define first

$$precnnb[S] \doteq u,$$

$$succnnb[S] \doteq v. \tag{4.8}$$

Next, (see Figure 4.3)

$$\begin{aligned} prefi[S] &\doteq X_0 \quad \text{if } u > 0, \\ &\doteq \emptyset \quad \text{or is undefined if } u = 0; \\ prela[S] &\doteq X_{u-1} \quad \text{if } u > 0, \\ &\doteq \emptyset \quad \text{or is undefined if } u = 0; \\ sucfi[S] &\doteq Y_0 \quad \text{if } v > 0, \\ &\doteq \emptyset \quad \text{or is undefined if } v = 0; \\ sucla[S] &\doteq Y_{v-1} \quad \text{if } v > 0, \\ &\doteq \emptyset \quad \text{or is undefined if } v = 0. \end{aligned} \tag{4.9}$$

Finally, if $u > 1$, then for $x = 0, \ldots, u-2$, and $y = 1, \ldots, u-1$, we set

$$preletori[X_x] \doteq X_{x+1},$$

$$preritole[X_y] \doteq X_{y-1}, \tag{4.10}$$

and if $v > 1$, then for $x = 0, \ldots v-2$, and $y = 1, \ldots, v-1$, we set

$$sucletori[Y_x] \doteq Y_{x+1},$$

$$sucritole[Y_y] \doteq Y_{y-1}. \tag{4.11}$$

The prefixes "pre" and "suc" are shorts for "<u>pre</u>ceding" and "<u>suc</u>ceding", while the suffixes "cnnb", "fi", "la", "letori" and "ritole" are shorts for "<u>c</u>o<u>n</u>ection <u>n</u>um<u>b</u>er", "<u>fi</u>rst", "<u>la</u>st", "<u>le</u>ft <u>to</u> <u>ri</u>ght" and "<u>ri</u>ght <u>to</u> <u>le</u>ft".

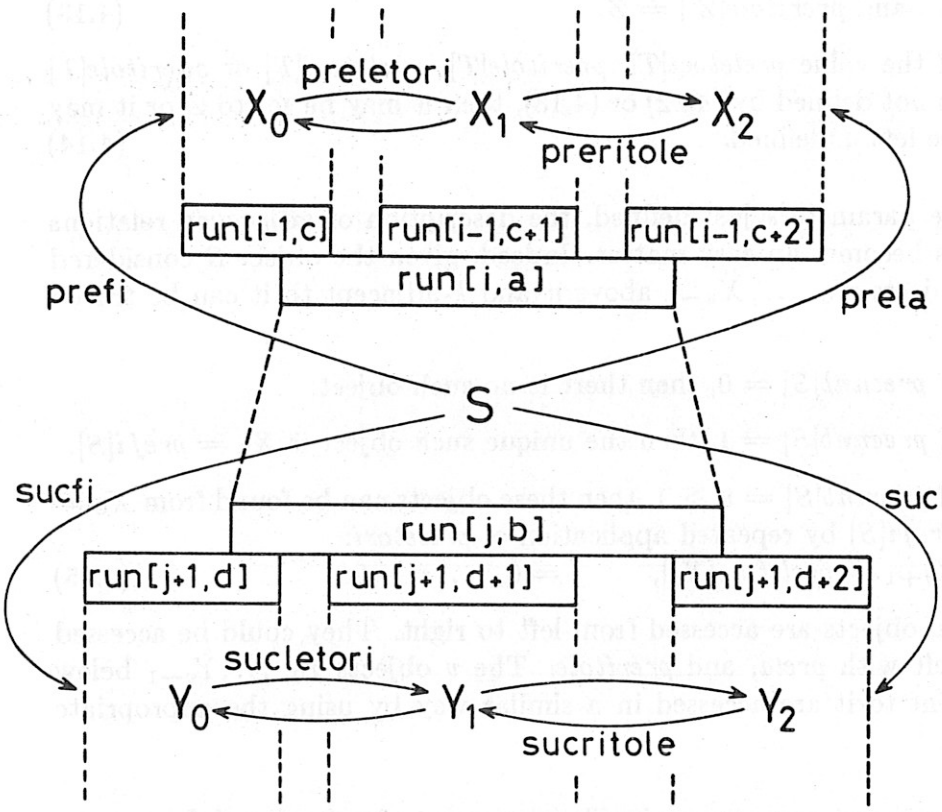

Figure 4.3. Object parameters

So far, the definitions of *preletori*, *preritole*, *sucletori* and *sucritole* depend implicitely on the object S. Obviously, this is not satisfactory, and we need equivalent definitions that do not depend on any particular object.

Let Z and Z' be two nonvoid objects with respective topmost runs $run[i, a]$ and $run[i', a']$, and respective bottommost runs $run[j, b]$ and $run[j', b']$. Then:

> If $i' = i$, $a' = a + 1$ and there is a run on row $i - 1$ which is k-adjacent to both $run[i, a]$ and $run[i, a + 1]$, then $sucletori[Z] \doteq Z'$, and $sucritole[Z'] \doteq Z$. (4.12)
>
> If $j' = j$, $b' = b + 1$ and there is a run on row $j + 1$ which is k-adjacent to both $run[j, b]$ and $run[j, b + 1]$, then $preletori[Z] \doteq$

Z', and $preritole[Z'] \doteq Z$. (4.13)

If the value $preletori[T]$, $preritole[T]$, $sucletori[T]$ or $sucritole[T]$ is not defined by (4.12) or (4.13), then it may be set to $\emptyset$ or it may be left undefined. (4.14)

With the parameters just defined, the description of adjacency relations between objects becomes an easy matter. Indeed, given the object S considered above, the u objects $X_0, \ldots, X_{u-1}$ above it and k-adjacent to it can be traced out as follows:

(*i*) If $precnnb[S] = 0$, then there is no such object.

(*ii*) If $precnnb[S] = 1$, then the unique such object is $X_0 = prefi[S]$.

(*iii*) If $precnnb[S] = u > 1$, then these objects can be found from $X_0 = prefi[S]$ by repeated application of $preletori$:
$X_{j+1} = preletori[X_j], \qquad j = 0, \ldots, u-2.$ (4.15)

In this example, objects are accessed from left to right. They could be accessed from right to left with *prela*, and *preritole*. The v objects $Y_0, \ldots, Y_{v-1}$ below S and k-adjacent to it are accessed in a similar way by using the appropriate parameters.

Our ten parameters convey a significant amount of redundant information. Therefore, depending on the application at hand, both the definition and use of some of these will be left optional, (see Section 5.2). However, when it is intended to take full advantage of the potentialities of the approach, redundancy in the input data enables us to significantly improve the algorithmic efficiency. This is but another occurrence of the ubiquitous time-space tradeoff in algorithmic complexity.

§4.4 Representation of Objects and Adjacency Relations

As we pointed out previously, a careless approach to the extraction of connected components is likely to place overwhelming demand in terms of storage space. Anticipating slightly upon what follows, let us note that our goal is to reserve primary storage for objects belonging to connected components which are not yet completely disclosed; to transfer the corresponding data to some secondary storage once the disclosure of any connected component is completed; and to release the primary storage space in order that it may be re-used in the sequel. Clearly, these considerations reveal much of our motivation for using Pascal—with its dynamic storage allocation—as a programming language. From now on, this choice will reflect more and more on the presentation, but is should remain clear that the implementation of our approach could be seasoned to very different tastes. Incidentally, it will be assumed hereafter that the machine-dependent, Pascal procedure *dispose*—which releases the record to which its argument is a pointer— enables *all* of the corresponding memory to be re-used.

To every object (hinge, block, block-continuation) we associate a *record*, of type "*objrec*" indexed by pointers of type "*link*". Thus we write $link = \uparrow objrec$. (In accordance with Pascal notation, the procedure *new*(p) creates a component of type *objrec* whose name is $p \uparrow$, whereas the procedure *dispose*(p) erases the variable $p \uparrow$ and releases the corresponding memory.)

The record corresponding to an object has a fixed part, and a variant part depending on the object's type. The discriminant component (tag field) for the variant part is the type variable ty : 0. .3, see (4.6). In terms of the variables defined thus far, the declaration of such a record is as follows:

```
objrec= RECORD
          precnnb, succnnb: 0..maxnbr;      {see p. 34}
          prefi, prela, sucfi, sucla,
          preletori, preritole, sucletori, sucritole: link;
          CASE ty : t03 OF                  {t03 = 0..3}
          0: ( hro: 1..mm2;                 {mm2 = m-2}
               hbe, hen: 1..nm2 );          {nm2 = n-2}
          1: ( fr: 1..mm2;
               b, e: 1..nm2;
               bll: 0..blen;                {blenm1 = blen-1}
               blbedif, blendif: ARRAY[0..blenm1] OF -d..d );
          2: ( ctl: 1..clen;
               ctbedif, ctendif: ARRAY[0..clenm1] OF -d..d );
          3: ()                             {clenm1 = clen-1}
        END;
```

Declaration of an object record

As can be seen in this piece of code, the fixed part of the record merely contains (optionally):

- ▸ Both the numbers *precnnb*$[S]$ and *succnnb*$[S]$, where S is the object coded by the record.
- ▸ Pointers of type *link*, corresponding to *prefi, prela, preletori, preritole*, etc. (In the case that one of these functions has its valuation set to $\emptyset$, or left undefined, we set the corresponding pointer to *NIL*.
- ▸ Not shown here are some additional parameters which will be defined and entered at a later stage.

In the variant part of the record we store the number of runs that make up the object, and their respective rows, beginnings, and ends.

Recall that a hinge is a *type* 0 object, ($ty := 0$), and consists of a single run, say $run[i,t]$. Then, using an obvious notation, the entries in the record declaration, namely, $hro : 1\,..\,M-2$, and $hbe, hen : 1\,..\,N-2$, are set to $hro := i$, $hbe := be[i,t]$, and $hen := en[i,t]$.

Blocks—in actual fact *d*-blocks—are objects of *type* 1, ($ty := 1$). Given a block consisting of $1+w$ runs, we will call w the *length* of that block. Let *blen* designate the chosen maximal block length. *d*-blocks are coded differentially. Suppose that the *d*-block consists of $1+w$ runs, ($w \leq blen$). For $j = 0, \ldots, w-1$ we define

$$blbedif[j] \doteq be[i+j+1, t_{j+1}] - be[i+j, t_j],$$

and

$$blendif[j] \doteq en[i+j+1, t_{j+1}] - en[i+j, t_j]. \tag{4.16}$$

Then the entries in the object's record are: $fr := i$ (fr is a short for first row), $b := be[i, t_0]$, $e := en[i, t_0]$, and $bll := w$ (bll is a short for block length). $blbedif[j]$, and $blendif[j]$ are determined by (4.16) if $j < w$, and can be left undefined otherwise.

We define the length of a block-continuation as the number of runs that it contains. (Note that the definitions of length do not coincide for blocks and block-continuations.) In the case of block-continuations which are *type* 2 objects ($ty := 2$), we use the term *clen* to denote the chosen maximal length. In view of what precedes, the interpretation of the other entries is straightforward. Therefore, it is omitted.

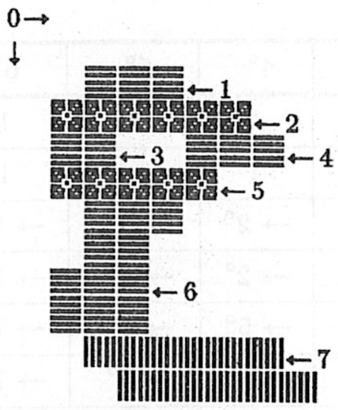

Figure 4.4. The same as Figure 4.1.c.

$ty = 0$			
$n°$	*hro*	*hbe*	*hen*
2°	2	1	6
5°	4	1	5

$ty = 1$										
$n°$	*fr*	*b*	*e*	*bll*	*blbedif*			*blendif*		
1°	1	2	4	0	*	*	*	*	*	*
3°	3	1	2	0	*	*	*	*	*	*
4°	3	5	7	0	*	*	*	*	*	*
6°	5	2	4	3	0	−1	0	−1	0	−1

$ty = 2$			
$n°$	*ctl*	*ctbedif*	*ctenbedif*
7°	2	1 1 *	4 1 *

Table 4.1.a. Objrec records, variant part

n°	1°	2°	3°	4°	5°	6°	7°
precnnb	0	1	1	1	2	1	1
succnnb	1	2	1	1	1	1	0
prefi	*NIL*	$\to 1^\circ$	$\to 2^\circ$	$\to 2^\circ$	$\to 3^\circ$	$\to 5^\circ$	$\to 6^\circ$
prela	*NIL*	$\to 1^\circ$	$\to 2^\circ$	$\to 2^\circ$	$\to 4^\circ$	$\to 5^\circ$	$\to 6^\circ$
sucfi	$\to 2^\circ$	$\to 3^\circ$	$\to 5^\circ$	$\to 5^\circ$	$\to 6^\circ$	$\to 7^\circ$	*NIL*
sucla	$\to 2^\circ$	$\to 4^\circ$	$\to 5^\circ$	$\to 5^\circ$	$\to 6^\circ$	$\to 7^\circ$	*NIL*
preletori	*NIL*	*NIL*	$\to 4^\circ$	*NIL*	*NIL*	*NIL*	*NIL*
preritole	*NIL*	*NIL*	*NIL*	$\to 3^\circ$	*NIL*	*NIL*	*NIL*
sucletori	*NIL*	*NIL*	$\to 4^\circ$	*NIL*	*NIL*	*NIL*	*NIL*
preritole	*NIL*	*NIL*	*NIL*	$\to 3^\circ$	*NIL*	*NIL*	*NIL*
ty	1	0	1	1	0	1	2

Table 4.1.b. Objrec records, fixed part

There is one particular advantage in using d-blocks (and d-block-continuations) instead of blocks: The numbers in (4.16) can be coded in $\log_2(2d+1)$ bits each. Now, if we take $d = 2^{e-1} - 1$ with $e > 0$, these numbers can be coded in e bits. Thus, the corresponding arrays may be stored economically in the form of *packed arrays*.

In Figure 4.4, we show a decomposition of Figure 4.1.a into objects, with $blen = clen = d = 3$. We give a label to each object and display the corresponding records in Table 4.1.

§4.5 Real-Time Construction of Object Records

Presently, we wish to examine the processing involved by the construction of object records in real-time, and with a single raster scan of the figure. To clarify matters, this part of the program is described in two stages. The first stage introduces the data structure used to acquire the necessary information at the run and row level, and formulates the algorithm in terms of basic processing steps. In the second stage we shall be concerned with creating or updating object records.

4.5.1 Data structure and basic processing steps

Let us first decompose the processing as shown in the following algorithm:

```
BEGIN
initialization
FOR i:=1 TO M-2 DO
     processonrow;
     IF i<M-2 THEN transitiontothenextrow
     END
END;
```

Basic processing steps

At any time during the execution of this program, we assume that the following is known;

(*i*) The records corresponding to the objects of the figure in the state they were left at the end of the processing of row $i-1$.

(*ii*) The runs of row $i-1$, their parameters, and the name of the record of the object to which each of them belongs.

(*iii*) The runs of row i and their parameters.

Assumption (*i*) is quite natural: it is the basis of the iteration process. We have seen in Section 3.3 that assumption (*iii*) can be satisfied in real-time. Our problem, then, is merely to meet the requirements of assumption (*ii*). To this end, we shall presently introduce two more types of auxiliary records. The first one, of type "*rowrec*" is associated with a given row. Its major data subfield is an array of records of the second type, namely, "*runrec*" each of which is associated with a given run in the row. We have the following declarations:

```
rowrec= RECORD

         nbr: 0..maxnbr;
         runpar: ARRAY[0..max1] OF runrec
         END;                          {max1 = maxnbr-1}

runrec= RECORD
         objpoin: link;
         objty: 0..3;
         CASE objty : t03 OF
         0: (rri: 0..max1);
         1: (rbe, ren: 1..nm1);        {nm1 = N-1}
         2,3: ()
         END;
```

Declarations of rowrec and runrec records

In the above declarations, "*maxnbr*" (maximum number of runs on a row) is an upper bound for the number $nbrun[j]$ of runs on row j; "*par*" and "*rec*" stand for parameters and record. Given a row v, we can now proceed with the (self explanatory) assignment of the following values:

$$nbr := nbrun[v].$$

If $j \le nbrun[v] - 1$, then

$$runpar[j].objpoin := z,$$

where z is a pointer to the record corresponding to the object containing $run[v,j]$ (see next section for the determination of z); if, in addition, $run[v,j]$ is a hinge, then

$$runpar[j].objty := 0,$$
$$runpar[j].rri := nrisu[v,j] - 1,$$

while if $run[v,j]$ is a block run, then

$$runpar[j].objty := 1,$$
$$runpar[j].rbe := be[v,j],$$
$$runpar[j].ren := en[v,j].$$

Note that here the case $objty = 1$ corresponds to objects of both types 1 and 2. On the other hand, if $j \geq nbrun[v]$, then:

$$runpar[j]\,.\,objpoin := NIL,$$
$$runpar[j]\,.\,objty := 3.$$

For initialization purposes, we need a variable of type *rowrec* corresponding to a row of white pels. We call it "*emptyrow*". In *emptyrow*, we have: $nbr := 0$; and $runpar[j] := blank$; $(j := 0, \ldots, maxnbr - 1;)$ where *blank* is the variable of type *runrec* defined by $blank\,.\,objpoin := NIL$; and $blank\,.\,objty := 3$.

During a standard iteration, we need two variables of type *rowrec* for rows $v = i - 1$, and $v = i$, (see assumptions (i) and (ii) above). We call them "*precrow*" and "*thisrow*" respectively.

In terms of these variables, the "initialization" and the "transition to next row" in our basic program read as follows:

```
BEGIN                {Initialization}
precrow:=emptyrow;
thisrow:=emptyrow
END;

BEGIN                {transition to next row}
precrow:=thisrow;
thisrow:=emptyrow
END;
```

Initialization and transition to next row

It is easy to see that these steps meet the real-time requirements. The initialization step is, clearly, no problem. The transition to next row requires a time proportional to *maxnbr*. However, that time can be made proportional to $nbrun[i] + max(nbrun[i-1], nbrun[i])$ with the following improved program:

```
BEGIN
xm:=thisrow.nbr;
xn:=max(precrow.nbr, xm);
FOR j:=0 TO xn-1 DO
     precrow.runpar[j]:=thisrow.runpar[j];
FOR j:=0 TO xm-1 DO
     thisrow.runpar[j]:=blank;
thisrow.nbr:=0
END;
```

Improved transition to next row

4.5.2 Creating and updating object records

Let us now turn to the stage "process on row i" in our basic program. As one could expect, this is a crucial stage.

The operations performed at this stage essentially consist in creating new object records whenever this is required, and updating existing records in accordance with the run configuration prevailing on rows i and $i \pm 1$. All these operations are combined in a unique procedure called "*allocate*" which, depending upon the current configuration, may activate a number of auxiliary subroutines. Ten of these serve to create and update the data structure defined so far. Some more subroutines will be introduced in the following chapter. Let us however have a look at *allocate* as it would read at this stage. (See Appendix C for the exhaustive code of procedure *allocate*, and Appendix D for the code of auxiliary subroutines.)

```
PROCEDURE ALLOCATE(u : tOmaxnbr);    {tOmaxnbr = 0..maxnbr}
VAR p,z : link; wrec : runrec;
BEGIN
thisrow.nbr:=u+1; wrec:=precrow.runpar[lefpr];
z:=wrec.objpoin;
IF (conpr=1) AND (consu<=1) AND (wrec.objty=1)
     AND (abs(be-wrec.rbe)<=d) {for d-blocks only}
     AND (abs(en-wrec.ren)<=d) {id}          {condition 1}
  THEN
```

```
    IF (z↑.ty=1) AND (z↑.bll<blen)                 {condition 2}
      THEN
        BEGIN
        thisrowobjty1(z); blockenlarge(z↑,wrec);
        conbelow(z)
        END                                        {statement 1}
      ELSE
        IF (z↑.ty=2) AND (z↑.ctl<clen)             {condition 3}
          THEN
            BEGIN
            thisrowobjty1(z); continuationenlarge(z↑,wrec);
            conbelow(z)                            {statement 2}
            END
          ELSE
            BEGIN
            newobject; thisrowobjty1(p);
            newcontinuation(p↑,wrec); conbelow(p)
            END                                    {statement 3}
  ELSE
    BEGIN
    newobject;                                     {statement 4}
    IF (conpr<=1) AND (consu<=1)                   {condition 4}
      THEN
        BEGIN
        thisrowobjty1(p); newblock(p↑); conbelow(p)
        END                                        {statement 5}
      ELSE
        BEGIN
        thisrowobjty0(p); newhinge(p↑); conbelow(p)
        END                                        {statement 6}
    END
 END{allocate};
```

The procedure *allocate*

From the mnemonics used as subroutine's names, the reader may have guessed much of the operational significance of procedure *allocate*. Let us, however, examine it in some detail.

In the first place, when $conpr[i,u] \neq 0$, the connections between a given run, say $run[i,u]$, and the objects on preceding rows are established via $run[i-1,$

lefpr[*i*, *u*]] which is the first run not to the left of *run*[*i*, *u*] on row $i-1$. The *runrec* record associated with that run is *precrow*.*runpar*[*lefpr*[*i*, *u*]], and that run belongs to the object *precrow*.*runpar*[*lefpr*[*i*, *u*]].*objpoin* ↑. Hence, the assignments of variables *wrec* and *z* at the beginning of the procedure.

In the second place, the body of *allocate* is structured around four IF ... THEN ... ELSE decompositions. Condition 1 is satisfied if *run*[*i*, *u*] and *run*[*i* − 1, *lefpr*] form together a *d*-block. If, in addition, object *z* ↑ is a block whose current length is less than *blen* (condition 2), then statement 1 which activates three subroutines is executed. This has the effect of (a) assigning the appropriate values in the run record *thisrow*.*runpar*[*u*] (subroutine *thisrowobjty*1); (b) updating the parameters *bll*, *blbe*(*en*)*dif* in the variant part of object record *z* ↑ (subroutine *blockenlarge*); and (c) updating the connectivity parameters *succnnb*, *sucfi*(*la*), *preletori*(*ritole*) of *z* ↑ (subroutine *conbelow*). As a matter of illustration, the subroutine *blockenlarge* reads as follows:

```
PROCEDURE blockenlarge(VAR t: objrec; VAR w: runrec);
BEGIN
WITH t DO
  BEGIN
  blbedif[bll]:=be-w.rbe;
  blendif[bll]:=en-w.ren;
  bll:=bll+1
  END
END{blockenlarge};
```

The procedure *blockenlarge*

If condition 2 is not satisfied and if condition 3 is, in which case object *z* ↑ is a block-continuation of current length less than *clen*, then the very same operations are performed in terms of a block-continuation instead of a block (statement 2), else a new block-continuation is created (statement 3). In statement 3, we are confronted with the first occurrence of the subroutine *newobject* which, in this case, is followed by the subroutine *newcontinuation*. The reason for this apparent duplication of effort is quite simple: Subroutine *newobject* is a fairly general subroutine whose role is to create a new object record whenever this is required, to fill the fixed part of that record in every possible configuration, and, if necessary, to update the records of the preceding objects accordingly. This procedure is thus invoked again whenever a new hinge and the beginning of a new block are encountered. In the case of statement 3 the role of subroutine *newcontinuation* is then to fill the appropriate variant part of the object record just created.

At this point, the remainder of procedure *allocate* should be nearly self-explanatory. If the requirements in condition 1 are not met, $run[i, u]$ is, in any event, the first run of a new object. Hence statement 4. If condition 4 is satisfied, that new object is in fact a new block, else it is a new hinge. Hence, statements 5 and 6.

Now, the procedure *allocate* could be activated in two slightly different ways. On the first hand, it could be invoked, during the processing of row i, as soon as the parameters of $run[i, u]$ are acquired. In this case, the algorithm operates in real-time at the run level. On the other hand we might defer the calls of *allocate* until all the runs of row i have been acquired. The algorithm would gains in modularity and still operate in real-time, but at the row level. The reader may have inferred from our last comments in Chapter 3, that it is the first of these option which has been implemented.

4.5.3 Comments

Presently, we wish to point out a number of possible, minor simplifications, and to comment on the time and space complexity of procedure *allocate*.

Condition 1 involves two tests which apply to d-blocks only. Clearly, if one uses blocks instead of d-blocks, these conditions should be deleted without further ado. (The same remark applies in various places in the program.)

If the procedure was implemented in a language admitting dynamic arrays and records, then we could handle blocks of arbitrary length, and there would be no need for block-continuations. Readily, Conditions 2 and 3 and Statements 2 and 3 would disappear altogether, as well as the procedures *continuationenlarge* and *newcontinuation*.

If we were to take $blen = clen$, it would be advisable to use the same data structure for type 1 and 2 objects. Both procedures *blockenlarge* and *continuationenlarge* could be merged in a unique procedure. The branching due to Condition 3 would disappear as well as the test "$z \uparrow .ty = 1$" in Condition 2.

Let us now turn our attention to the time-complexity of procedure *allocate*. For what concerns adjacency relations, it is proportional to $conpr[i, u]$. The time-complexity of auxiliary operations is dominated by that of procedure *endof* which, in turn, is bounded from above by $max(blen, clen)$. Procedure *endof* is called only once (in procedure *newobject*) for each block or block-continuation. Therefore, as

$$conpr[i, u] \leq \lceil (en[i, u] - be[i, u])/2 \rceil + 1,$$

allocate operates in real-time at the run level.

At the end of the scan of the current row, the stage `transitiontothenextrow` is also linear in the number of pixels of the row. Therefore, the whole process is also real-time at the row level.

For what concerns memory requirements, they merely consists in *objrec* and *rowrec* records. The number of *objrec* records is the number of objects under consideration. The *rowrec* records *thisrow* and *precrow* have a size which is proportional to N as was the case for the arrays used to store the 3-row window.

REFERENCES

[1] P. Grogono, *Programming in Pascal*, Reading, Ma.:Addison-Wesley, 1978.

[2] K. Ramachandran, "Coding method for vector representation of engineering drawings", *Proc. IEEE* vol. 68, pp. 813–817, July 1980.