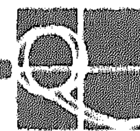


A Generalization of the z-Buffer Algorithm

Kees van Overveld and Huub van de Wetering
Eindhoven University of Technology
PO Box 513; 5600 MB Eindhoven; The Netherlands
e-mail: wsinkvo@win.tue.nl, wstahw@wim.tue.nl

A generalization of the z-buffer algorithm is given that allows for dynamic updates of a screen image without the necessity of re-drawing the entire scene. It is based on the notion of equivalence classes of pixels, that is groups of pixels that are covered by the same set of polygons. A data structure for representing these equivalence classes in an efficient manner is discussed.

1. Introduction

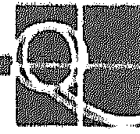
In rendering 3-dimensional objects, one of the issues to be dealt with is the elimination of hidden surfaces. Several techniques exist for this ([1]). The applicability of a given technique depends among other things on the representation of the 3-dimensional object (polygonal or defined by curved surfaces, constructive solid geometry (CSG) or boundary representation) and on the desired quality of the resulting image (transparency, cast shadows, reflections and/or refractions, and so on).

Here we consider only the z-buffer algorithm.

A z-buffer is a two-dimensional array of the same size as the frame buffer and it contains for each pixel the depth-value (for that pixel) of the visible object closest to the viewer. Filling the frame buffer now consists of scan-converting an object and checking the depth-values against the values already in the z-buffer: if the newly computed value is closer to the viewer, a pixel is drawn in the frame buffer and the z-buffer is updated, otherwise nothing happens. In this way the hidden surfaces are not shown in the frame buffer.

The z-buffer algorithm needs no preprocessing time (apart from initializing the z-values to $-\infty$ in all pixels). It is well suited for both polygonal and non-polygonal descriptions. However, it is rather inefficient when it comes to interactive applications: indeed, removing or changing merely one single polygon necessitates re-drawing the entire scene, thus re-examining all polygons.

In this paper, an approach is made to generalize the z-buffer algorithm in order to arrive at an algorithm that is able to deal with changing scenes. The idea is based on the observation that a local change of the scene ideally should



involve a merely local update of the image. This holds for the original z-buffer algorithm in case the change consists of adding an object; in case an object should be removed, however, the information about the part of the scene that underlays this object is missing (since only the information about the nearest part of the scene is maintained), and the emanating 'hole' cannot be filled.

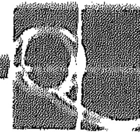
A first approach for modifying the z-buffer algorithm could be to store in every pixel references to all the polygons that cover it; this would necessitate a data structure consisting of a set of polygon references for every pixel. Closer observation of the state of such a data structure when using it to render a typical picture would surely reveal that two neighboring pixels have a large chance of having the same set attached to them: they are so to say equivalent. This equivalence relation (a formal definition will be given in the next section) gives rise to equivalence classes, to be called `e_classes`: groups of pixels having the same set of polygons covering them. Instead of storing the entire set of these polygons at every pixel, merely a pointer to the appropriate `e_classes` suffices. In order to remove one polygon, say P, P should first be scan-converted again in order to establish which `e_classes` are involved, and next merely the pixels in the union of these `e_classes` should be drawn again, thereby removing the references to P from these `e_classes`. On the other hand, adding a polygon is slightly more work than in the original z-buffer algorithm, since the `e_classes` of all pixels being covered by it should be updated. Finally, care should be taken every now and then to remove `e_classes` that are not referenced anymore in order to prohibit excessive waist of memory space (garbage collection). In Section 1, algorithms are given for adding and deleting a polygon and for garbage collection. Although these algorithms are dealing with polygons, they work equally well for every other brand of 3-dimensional graphics primitives that can be rendered into a z-buffer.

Although the concept of coherence, that underlies the notion of the `e_classes`, causes the storage requirements for the set of `e_classes` to be considerably less than the storage that would be needed for naively storing per pixel references to all polygons covering a pixel, the idea of coherence may be used once more to reduce the storage requirements even further. Indeed, many of the equivalent classes will have many polygons in common, and it suffices to store these common parts only once. The essential changes in the algorithms to achieve this will be dealt with in Section 2.

Finally, Section 3 contains some remarks concerning the time-space complexity of the algorithms and discusses other applications of the notion of `e_class` in rendering and object representation.

1.1. Related work

Maintaining a data structure consisting of sets of objects for increasing interaction speed by enabling incremental computation of a hidden surface algorithm are also used in [4] and [5]. In [4] a hashing function is used to delimit the storage needed; this limits the number of objects that can be used. In [5] the object sets are not stored per pixel as in both [4] and in this paper, but per tile



of some tiling of the frame buffer; consequently, it is not immediately possible to find the exact set of polygons that have a pixel in common with a deleted polygon.

In [2] an object space approach is taken as opposed to the image space approach in this paper. Their incremental hidden surface algorithm is based upon a special representation for polygons.

In [6] an item buffer is used which also contains for each pixel a set of objects projecting on that pixel; here these sets are used in a preprocessing step for a ray tracing algorithm.

In [3] a so-called CSG-buffer is used; this buffer contains a per pixel hierarchical representation of a set of objects; the hierarchy is based upon an ordering in depth-priority.

2. Equivalence classes

In this and the following section, the notion of invariants will be used for correctness arguments of algorithms. The notation and the typing conventions of the C-language are applied. For the purpose of notational convenience, a **set** primitive is assumed to be available. In practice, this can be thought of to be implemented as arrays or with dynamic allocation.

For pixels, it is assumed that the coordinates are restricted to the rectangular interval $PX = [1..XMAX][1..YMAX]$. This implies that only those parts of the polygons that fall within this region are considered.

An equivalence class is defined as follows:

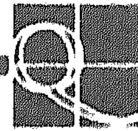
```
                                              (Definition e_class)
typedef struct
{ int n;              /* the number of pixels of this e_class */
  set of polygon PR;           /* the polygons of this e_class*/
} e_class;
                                               (end of definition)
```

The following global variables will occur in the algorithm.

```
e_class EC[NE];
      /* all e_classes; assume NE to be sufficiently large */
int ne ;
    /* 0< ne < NE; points to the first empty location in EC */
int z_buff[PX] ;                              /* the z-buffer */
int e_buff[PX] ; /* index into EC of the e_class of a pixel */
set of polygon PG;        /* the polygons that build a scene */
```

Note that in any practical implementation of this data structure pointers to polygons will occur, rather than the objects themselves. Furthermore, note that array indexing in C starts at zero.

The invariants that are to be maintained are the following predicates:



P0 : (references in `e_buff`)
The references in the `e_buff` are valid entries in EC.

P1 : (the polygon set of `e_classes`)
The `e_class` associated with a pixel contains precisely those polygons that cover the pixel

P2 : (the number of pixels in a `e_class`)
In every `e_class e`, `e.n` is the number of pixels that refer to `e`

P3 : (the z-buffer algorithm)
In every pixel `px`, `z_buff[px]` is the largest z-value of all polygons covering that pixel

In order to give a formal version of these invariants, the following operator is introduced:

' $\Pi(p)$ ', with p a polygon, is the pixel set onto which p projects.

Using $\Pi()$, the invariants may be written:

P0 : $(\forall_{px \in PX} : 0 \leq \texttt{e_buff}[px] < \texttt{ne})$
P1 : $(\forall_{px \in PX} : \texttt{EC}[\texttt{e_buff}[px]].\texttt{PR} = \{p \in PG | px \in \Pi(p)\})$
P2 : $(\forall_{0 \leq i < ne} : \texttt{EC}[\texttt{i}].\texttt{n} = \#\{px \in PX | \texttt{e_buff}[\texttt{px}] = i\})$
P3 : $(\forall_{px \in PX} : \texttt{z_buff}[\texttt{px}] = \max\{$ 'the z-value of polygon p at px'$| p \in PG \wedge px \in \Pi(p)\})$

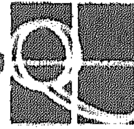
In order to initialize the invariants, all elements of `z_buff` should be set to $-\infty$, the set PG should be empty and the array `EC` should contain the reference to one `e_class`, `EC[0]`. The latter contains an empty set `EC[0].PR`, and `EC[0].n`= $XMAX * YMAX$. All elements of `e_buff` should be set to 0. Furthermore ne should equal 1.

2.1. Polygon addition

Assuming all invariants to hold, a naive version of the algorithm for adding a polygon is given below. In this algorithm the control structure **forscan** implements the scan conversion, including the updating of the z-buffer. The function **union** takes as arguments a set and an element and returns the union of these. The function **lin_search** searches the array EC for the occurrence of an `e_class` with the prescribed set of polygons attached to it and returns its index if so. It returns ne (that is, the first empty location) if no such `e_class` can be found. The algorithm to add a polygon to the scene is as follows:

```
add_polygon(new_pol)     (naive algorithm for adding a polygon)
polygon new_pol;
{
pixel px; int e, new_e; set of polygon PR_test;

forscan (px ∈ Π (new_pol) )
```



```
  {
  e= e_buff[px];
  PR_test= union (EC[e].PR,new_pol);
                                  /* Find e_class containing PR_test */
  new_e= lin_search (EC,PR_test);
                                              /* or create a not yet */
  if (new_e==ne) { EC[new_e]=(0,PR_test); ne=ne+1; }
                                               /* referenced e_class */

  EC[e].n=EC[e].n-1;
  EC[new_e].n = EC[new_e].n + 1;
  e_buff[px]=new_e;
  }
}                                             (end of naive algorithm)
```

The computational expense of this algorithm comes from the function calls to **union** and **lin_search**. Using coherence, both functions may be omitted. Indeed: instead of manipulating the entire set of polygon references for every pixel, it suffices to manipulate the reference to the `e_class`; moreover, for the majority of the pixels examined within the scan conversion of one polygon, this reference will remain the same. Assume that no more than ND different `e_classes` will be encountered in scan converting one polygon, then the `e_classes` encountered thus far may be stored in an array, `enc[ND]` of type `int`. The associated array `new_enc[ND]` serves to record the corresponding new `e_classes`. The integer nd is the number of encountered classes. The relation between `nd`, `enc`, and `new_enc` is given by:

$$\texttt{nd} = \# \{\texttt{e_buff}[\texttt{px}] \mid \texttt{px} \in p'\}$$

$$(\forall_{px \in \Pi(p')} : (\exists_{0 \leq i < nd} : \texttt{e_buff}[\texttt{px}] = \texttt{enc}[\texttt{i}] \;))$$

$$(\forall_{0 \leq i < nd} : \texttt{EC}[\texttt{new_enc}[\texttt{i}]].\texttt{PR} = \texttt{EC}[\texttt{enc}[\texttt{i}]].\texttt{PR} \cup \{\texttt{new_pol}\})$$

Here, p' stands for the part of the polygon `new_pol` already scan converted at a given instance. Observe that instead of having to search the entire array `EC` for the occurrence of the `e_class` with `PR_test` as its set of polygons, it suffices to search the array `enc[]` for an identical `e_class` reference, since all `e_classes` containing `new_pol` are referenced in the corresponding entry of `new_enc[]`.

An improved version of the addition algorithm therefore reads:

```
 add_polygon(new_pol)
                        (Improved version of the addition algorithm)
  polygon new_pol;
  {
  pixel px;
  int e,new_e,enc[ND],new_enc[ND],i,nd;
```

```
nd=0;
forscan (px ∈ Π (new_pol))
{
 e=e_buff[px];

 i=0; while (enc[i] ≠ e and i<nd) i=i+1;       /* Find e_class */
 if (i==nd)              /* or create e_class containing new_pol */
 {
  EC[ne] = (0, union (EC[e].PR,new_pol));
  enc[nd]=e;
  new_enc[nd]=ne;
  nd=nd+1; ne=ne+1;
 }
 new_e = new_enc[i];

 EC[e].n = EC[e].n-1;
 EC[new_e].n = EC[new_e].n+1;
 e_buff[px] = new_e;
}
}                                        (end of improved version)
```

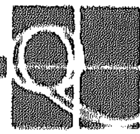
In the above algorithm the linear search loop only searches the array *enc*, which contains at most a number of elements equal to the number of different `e_classes` that occur within the current polygon. The number of times this loop is executed can be reduced by checking the current `e_class` `e_buff[px]` first against the `e_class` of the previous pixel in the scan conversion and assigning in case of equality the new `e_class` of this previous pixel to `new_e`. Using scan line coherence in this way the search loop is only executed for a small fraction of the pixels of a polygon.

2.2. Polygon deletion

Next an algorithm is given for deleting a polygon, say `pol`. It consists of three phases:

(1) First, the polygon `pol` is scan converted once more into the e-buffer. While doing so, a set of equivalence classes is recorded that are covered by `pol`. The numbers of these equivalence classes are stored in an array, the array `e_adapted[NA]`. The integer na equals the number of equivalence classes stored in `e_adapted`. Furthermore, the part of the z-buffer covered by `pol` is initialized to $-\infty$.

(2) Next, during the second phase, the polygons that are contained in these equivalence classes are recorded in the array `re_draw[NR]`, and the polygon `pol` is removed from the `e_classes` in `e_adapted[]`. The integer `nr` equals the number of polygons recorded in `re_draw`. The constants `NA` and `NR` are assumed to be large enough; in practical implementations, dynamical allocation techniques are useful here. Since recording the



equivalence classes covered by a given polygon does not involve lengthy linearly searching EC, a construction as in the 'improved version of the addition algorithm' appears to be unnecessary. On the other hand, to avoid storing e_class numbers more than once, linear searches in the array e_adapted[] of encountered e_classes will be necessary.

(3) Finally, in the third phase, the polygons in the array re_draw[] are drawn again while checking the z-coordinates against the z-buffer.

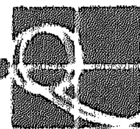
The algorithm looks as follows:

```
delete_polygon(pol)          (Algorithm for deleting a polygon)
polygon pol;
{
pixel px;
int prev,cur,e_adapted[NA],i,j,na,nr;
polygon p,re_draw[NR];

na=0; prev=ne;                                   /* PHASE ONE */
forscan (px ∈ Π (pol))
{
z_buf[px]= - ∞;
 cur=e_buff[px];
 if (cur ≠ prev)
 {
 i=0; while (e_adapted[i] ≠ cur and i<na) i=i+1;
 if (i==na) e_adapted[na]=cur; na=na+1;
 prev=cur;
 }
}

nr=0;                                            /* PHASE TWO */
for (i=0;i<na;i=i+1)
{
 'remove pol from EC[e_adapted[i]].PR'
 for (p ∈ EC[e_adapted[i]].PR)
 {
 j=0; while (j<nr and re_draw[j] ≠ p) j=j+1;
 if (j==nr)  re_draw[nr]=p; nr=nr+1;
 }
}

for (j=0;j<nr;j=j+1)                             /* PHASE THREE */
{
 'redraw polygon re_draw[j], using the z-buffer, but don't
 alter the e_classes '
 }
```



```
}                          (end of algorithm for deleting a polygon)
```

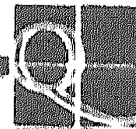
Despite the occurrence of three nested loops in phase two, the above algorithm is not unreasonably inefficient, since it only searches among (1) the number of `e_classes` covered by *pol*, (2) the number of polygons in one of these `e_classes`, and (3) the array of polygons to be re-drawn, `re_draw[]`. If all of these three sets exist of typically 10 or less items, then the total number of operations for removing one polygon is of the order of 1000. This is comparable to the mere scan converting of an average polygon.

2.3. Garbage collection

The two algorithms for adding and deleting a polygon, respectively, may result in large numbers of `e_classes`. Moreover, both during adding and deleting, `e_classes` may be formed that are not needed for the functioning of the algorithm. During the addition of a polygon, the value of the n-attribute of the original `e_class` decreases. When this reaches zero, that `e_class` is not referenced anymore by any pixel and ought to be removed from `EC[]`. These `e_classes` will be referred to as garbage of the first kind. On the other hand, during the deletion of a polygon, two `e_classes` that originally had different sets PR might become to have the same PR, and they therefore should be merged to become the same e_class. Spurious `e_classes` in this sense will be called garbage of the second kind. With respect to the garbage of the first kind, it appears at first sight that on-the-fly garbage collection might apply here: indeed, at the instance an `e_class`, say at location `k` in `EC[]`, gets `EC[k].n=0`, all references to `k` in `e_buff` have vanished, and the location `EC[k]` may be taken by the `e_class` `EC[ne-1]`, thus freeing the latter location and decrementing ne by one. There is a problem, however, in updating the `e_buffer` to accommodate this, since `e_classes` have no references to the pixel sets referencing to them, and a complete scan through all pixels seems necessary to replace all references to $ne - 1$ by references to k. Therefore on-the-fly garbage collection is not advisable. Instead, sophisticated lazy evaluation techniques might be employed. A simple garbage collector, however, which should be invoked whenever ne reaches a certain threshold, is given below.

```
garbage_collect()                  (Garbage collector algorithm)
{
int replace[NE],r,n;
pixel px;

n=0;
while (n<ne)
{
 if (EC[n].n ≠0)
 {
  r = r +1; replace[n]=r; EC[r]=EC[n];
```



```
  }
  n = n + 1;
  }
  for (px ∈ PX ) e_buff[px]=replace[e_buff[px]];
  }                             (end of garbage collector algorithm)
```

After the first loop in this algorithm the following predicates hold. (EC' is the array EC before execution of the loop)

$$(\forall_{0 \leq i < ne} : \mathtt{EC}'[\mathtt{j}] \neq 0 \rightarrow \mathtt{EC}[\mathtt{replace}[\mathtt{i}]] = \mathtt{EC}'[\mathtt{i}])$$

$$(\forall_{0 \leq i < ne} : \mathtt{replace}[\mathtt{i}] = \# \{0 \leq < i | \mathtt{EC}'[\mathtt{j}].\mathtt{n} \neq 0\})$$

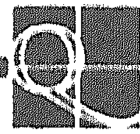
These two predicates indicate that the elements in the array EC with EC′[i].n≠ 0 are removed and that the remaining entries are put in consecutive entries of EC. The way in which the entries are changed is stored in the array *replace*; this array is used in the second loop to replace all old e_class pointers in the e_buff.

An algorithm for merging e_classes that have the same set PR (garbage collection of the second kind) is much more laborious. Indeed: finding these sets with less than $O(ne * ne)$ complexity would imply them to be ordered in some sense, which in turn would imply to have an ordering on the elements within the set e.PR of every e_class e. Since there are some intricacies involved with implementing the set type in an efficient manner with respect to the latter ordering, no attempt is done here to give an algorithm for garbage collection of the second kind. Instead, in the next section some alternatives for the representation of the e_classes are presented that allow for a more straightforward garbage collection of both first and second kind.

3. Efficient representations of equivalence classes

The representation of equivalence classes using the definition of e_class as presented in Section 1 might seem to be the most natural, it certainly is not the most efficient. Indeed, with respect to storage requirements it is noted that many of the elements of EC[NE] have numerous polygons in common. With respect to computational requirements, maintaining the set variable PR puts a significant burden on storage allocation computations.

To overcome this, a simple data structure that avoids the need of maintaining sets of polygons and at the same time saves considerable storage is presented. It is based on the notion of a tree of equivalence classes. Every e_class is a node in the tree. In contrast with the original data structure, however, only one polygon is stored in such a node, instead of all polygons of the class. The one polygon that is stored is the polygon that distinguishes this e_class from the e_class it stemmed from (the parent e_class). In additon to this one polygon, therefore, a node contains a reference to its parent e_class. Moreover, to facilitate garbage collection, a node also contains the number of pixels referencing it and the number of its sons. In the algorithms considered here, no recursive searching of the tree is needed, thus for simplicity it may be assumed that it is



still represented in linear form in the array EC[NE]. This yields for this type of e_class, to be referred to as e_t_class (*e*quivalence *t*ree *class*):

```
                                        (Definition e_t_class)
typedef struct
{ polygon pol;
    /* polygon distinguishing this e_t_class from its parent */
  int n,                 /* number of pixels of the e_t_class */
      m,                 /* number of sons of this e_t_class */
      p;             /* index in EC[] of the parent e_t_class */
} e_t_class;
                                             (end of definition)
```

An example of such an e_t_class tree is depicted in Figure 2a. Here, the three polygons A, B, and C of Figure 1 are added in that order, and scan conversion takes place from top to bottom and from left to right. The e_t_classes are added to the array EC in their order of creation. Every node is labeled with its index into EC[], the name of its distinguishing polygon, its number of sons, and its parent e_t_classes, respectively. In Figure 1 the numbers in parentheses within the different parts of the polygons indicate the distinct e_t_classes.

Note that an e_class maps one-to-one to an e_t_class, so the number of e_t_classes equals the number of e_classes. No other changes with respect to the global variables occur. The name e_buff[] will still be used for compatibility (although e_t_buff would be more appropriate). The contents of the set PR of an e_class corresponds with all polygons on the root path of the associated e_t_class. The root of the tree is EC[0], with initially EC[0].n=$XMAX * YMAX$, EC[0].m=0, EC[0].p=0 (the root node is its own parent), EC[0].pol=no_pol (=a non-occurring polygon).

3.1. Polygon addition

The improved version of add_pol given in the previous section needs only a slight adjustment, which is given below.

```
                              (adaption to add_pol for e_t_classes)
...
i=0; while (enc[i] ≠ e and i <nd) i=i+1;
if (i==nd)
{
 EC[ne] = (new_pol,0,0,e);                        /* new e_t_class */
 EC[e].m = EC[e].m + 1;                         /* add son to EC[e] */
```

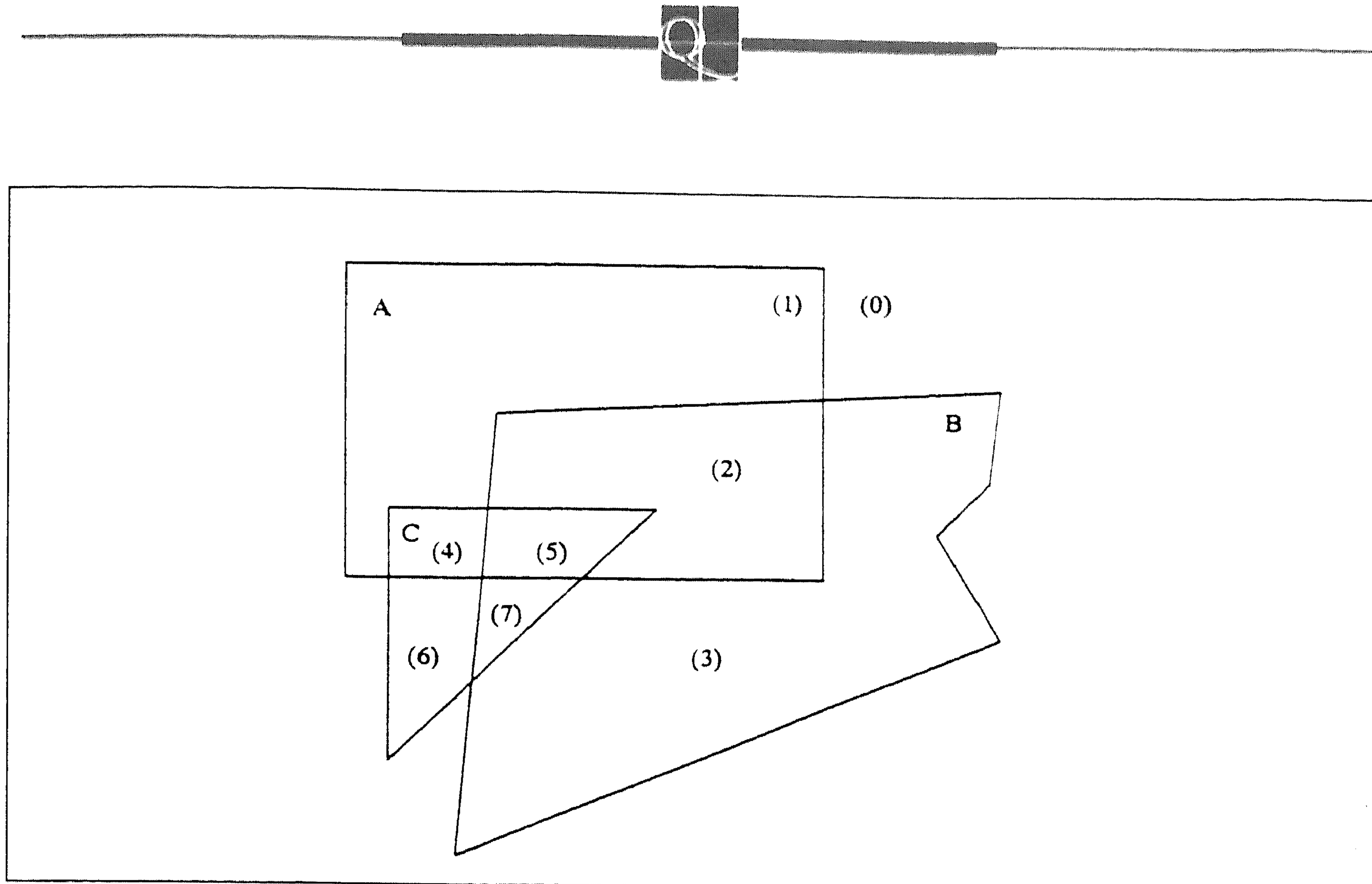


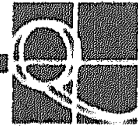
FIGURE 1.

```
enc[nd]=e;
  new_enc[nd]=ne;
  nd=nd+1; ne=ne+1;
  }
  ...                                          (end of adaption)
```

3.2. Polygon deletion

Next, the algorithm for deleting a polygon using the `e_t_classes` is given. It strongly resembles the former version. Merely phase two is organised slightly different: instead of searching the set PR of an `e_class`, the root path of an `e_t_class` has to be searched. A relevant observation is that a given polygon occurs at most once in every root path. It is necessary, therefore, to traverse the root path in order to find the polygon to be removed.

```
nr=0;             (PHASE TWO of delete_polygon for e_t_classes)
for (i=0;i<na;i=i+1)
{
 eti=e_adapted[i];
 while (eti ≠ 0)      /* while eti differs from the root node */
  {
   p=EC[eti].pol
   if (pol ≠ p and p ≠ no_pol)
   {
    j=0; while (j<nr and re_draw[j] ≠ p)j=j+1;
    if (j==nr) re_draw[nr]=p; nr=nr+1;
   }
```



```
   else EC[eti].pol= no_pol;

   eti=EC[eti].p;
  }
 }                                                  (end of PHASE TWO)
```

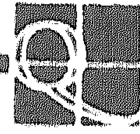
3.3. Garbage collection

Observe that the above deletion algorithm has left some equivalence classes with `no_pol` as their associated polygon. Moreover, note that, unlike the `e_class` representation, in the case of the `e_t_class` representation garbage collection only should take place in the case of the deletion of a polygon. It is a task for the garbage collector (of the first kind) to remove the nodes that correspond with deleted polygons. Moreover, when doing first kind garbage collection on `e_t_classes`, it is not allowed to remove nodes k that just have `EC[k].n=0`; indeed: nodes may exist that have no pixels referring to them but that lay on the root path of other nodes that are referred to. For this reason, nodes k may only be removed if `EC[k].n=0` **and** `EC[k].m=0`. Care should be taken therefore to decrease `EC[].m` whenever appropriate. The new first kind garbage collector is given below and consists of two passes:

(1) In the first pass nodes with `no_pol` as distinguishing polygon are removed. This is done by checking for every node whether or not its parent has a `no_pol`; if so the reference to the parent is replaced by a reference to its grandparent and the registration of the number of sons is adapted. For reasons of uniformity we consider the root to be its own parent (and, hence, grandparent).

(2) In the second pass nodes with `EC[k].n=0` $\wedge$ `EC[k].m=0` are removed similarly as in the garbage collector algorithm given in the previous section.

```
garbage_collect()
              (Garbage collector algorithm using e_t_classes)
{
 int replace[NE],n; pixel px;
 e_t_class son, par, gra;
 n=0;                                          /* First pass */
 while (n<ne)
 {
  son=EC[n]; par=EC[son.p]; gra=EC[par.p];
  if (par.pol==no_pol)
  {
   par.m=par.m-1;                        /* parent looses a son */
   son.p=par.p; gra.m=gra.m+1;      /* grandparent is new parent*/
  }
  n=n+1;
 }
```



```
n=1;                                                   /* Second pass */
while (n<ne)
{
 if (EC[n].n ≠ 0 or EC[n].m ≠ 0 )
 {
  r = r +1; replace[n]=r; EC[r]=EC[n];
 }
 n = n + 1;
}
for (i=0;i<ne;i=i+1) EC[i].p=replace[EC[i].p];
for (px ∈ PX ) e_buff[px]=replace[e_buff[px]];
}        (end of garbage collector algorithm using e_t_classes)
```

The effect of the garbage collector is depicted graphically in Figure 2.

The elements of the tuples (i;P;m;p) indicate the index in `EC[]`, the polygon, the number of sons, and the index in `EC[]` of the parent node, respectively; the polygon names and class indices refer to Figure 1.

Figure 2a: original situation; B has to be removed;
Figure 2b: situation after `delete_polygon(B)`; 'X' indicates '`no_pol`';
Figure 2c: situation after first pass;
Figure 2d: situation after the second pass and garbage collection of the second kind.

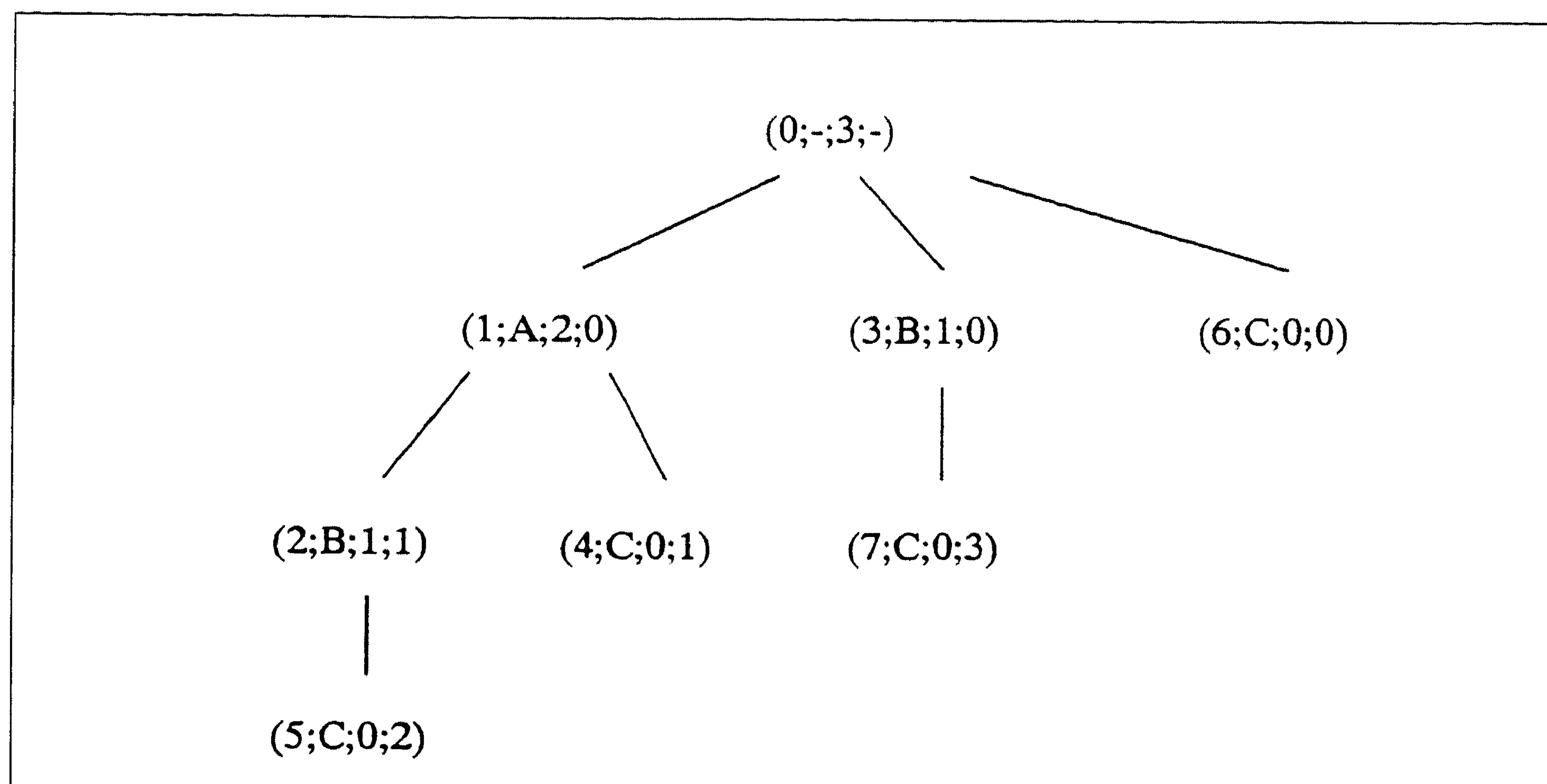
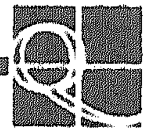


FIGURE 2a



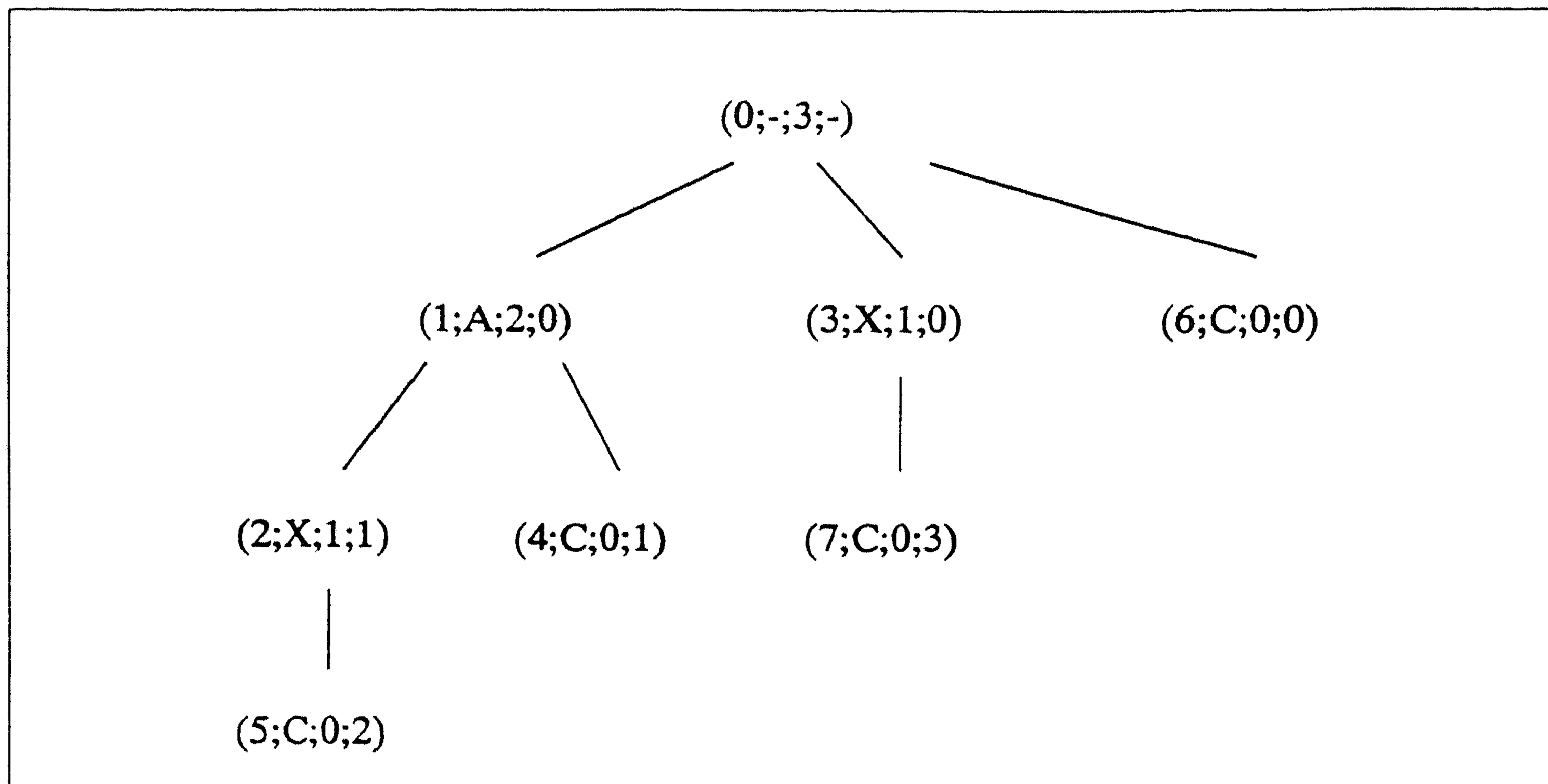


FIGURE 2b

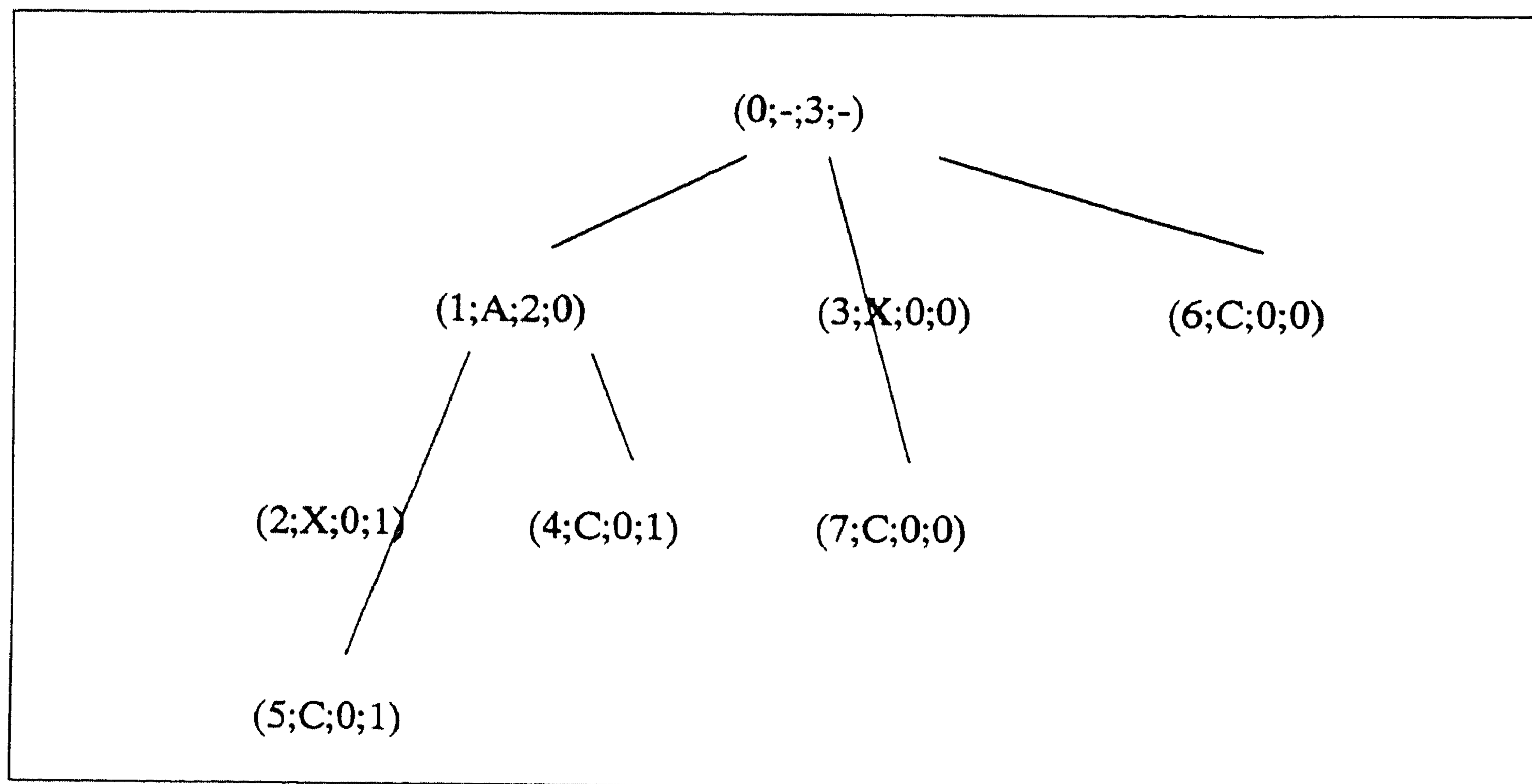
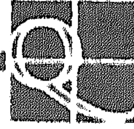


FIGURE 2c

Garbage collection of the second kind using `e_t_classes` may be achieved in a straightforward manner. Indeed, those `e_t_classes` `k` that have the same `EC[k].p` and the same `EC[k].pol` may be merged. Thus merely sorting the array `EC[]` on these keys suffices to find all `e_t_classes` that may be merged. (A consideration might be to maintain the order of the `e_t_classes` sorted in



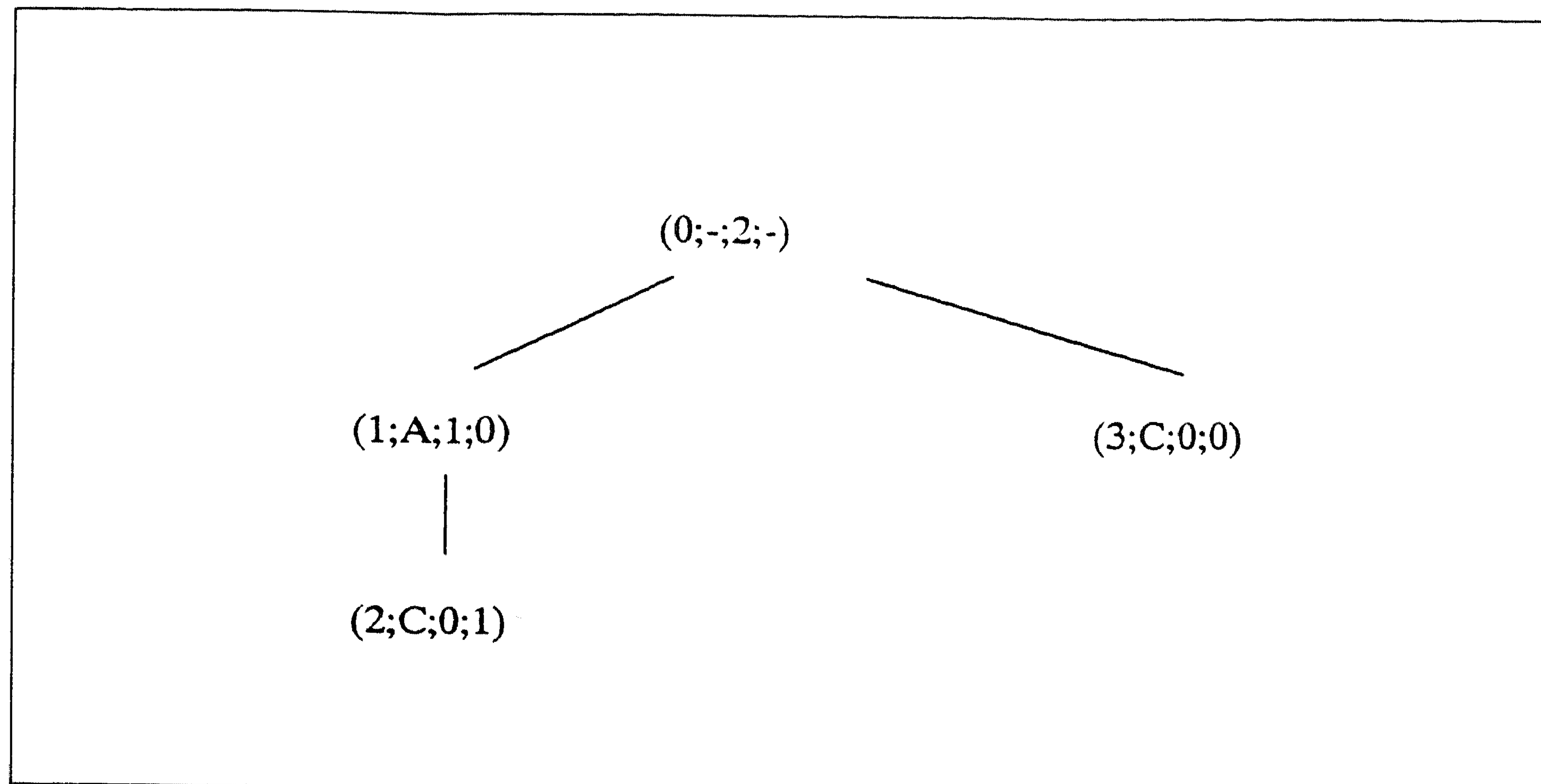


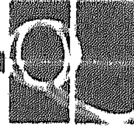
FIGURE 2d

the above sense in order to achieve on-the-fly garbage collection of the second kind at the expense of a somewhat lower average efficiency).

5. DISCUSSION AND CONCLUSIONS

Several applications exist where varying amounts of attributes are needed to be available for large amounts of objects. An example is the z-buffer algorithm: the objects are the pixels, and the attributes that should be available (in order for the z-buffer algorithm to cope with removing polygons) are the polygons that cover every pixel. Another example may be found in the field of geometric modelling where a model may consist of several sub-objects, that may be structured hierarchically. Ideally, of every point of the entire model the question should be answered to precisely which (sub-)objects that point belongs. A standard solution, i.e. maintaining dynamically created lists for every object (pixel, point), is not feasible due to the large computational overhead and the tremendous memory requirements. In both cases, there is a fair chance for coherency to exist; hence, the number of different combinations of attribute sets is likely by far less than the number of objects. A promising alternative then seems to be to associate precisely one pointer to every object that indexes into a set (array) of equivalence classes, where every equivalence class represents one unique combination of attributes. In this paper the idea of the application of such equivalence classes has been worked out for the case of the z-buffer algorithm. Algorithms have been presented for adding and deleting objects to a z-buffer that involve a mere local update of the scene. Moreover, several versions of the representation of the equivalence classes have been discussed.

The worst-case estimate of the number of equivalence classes (and with that,



the efficiency of the garbage collection algorithm and the needed storage requirements) is rather unrealistic. A theoretical upper bound for the number of equivalence classes given n polygons is 2^n, whereas in practical cases, the number of equivalence classes will be of the order of $n * m$ where m is the average depth complexity of the scene. This makes it very hard to give a general advice as to which representation to use. Concerning the time-space complexity of the adding and deleting algorithms a similar remark holds: the worst case performance could be very bad, but the real complexities are governed by the average depth complexity of the scene (this is related to the depth of the tree representation of the `e_t_classes` and the size of the sets PR in the `e_class` representation).

REFERENCES

1. J. FOLEY, A. VAN DAM, S. FEINER and S. HUGHES (1990). *Computer Graphics: Principles and Practice.* Addison-Wesley.
2. A.A.M. KUIJK, P.J.W. TEN HAGEN and V. AKMAN (1988). An exact incremental hidden surface removal algorithm. *Report CS-R8818*, CWI, Amsterdam.
3. FREDERIK W. JANSEN (1991). Depth-Order Point Classification Techniques for CSG Display Algorithms. *ACM transactions on Graphics*, Vol. 10, No. 1, pp 40-70.
4. DETLEF KRÖMKER (1987). Looking at Workstation Architectures from the Viewpoint of Interaction. *Advances in Computer Graphics Hardware I*, pp 27-37.
5. MEL SLATER, ALLAN DAVISON and MARK SMITH (1989). Liberation From Rectangles: a Tiling Method for Dynamic Modification of Objects on Raster Displays. *Computers & Graphics*, Vol. 13, No. 1, pp 83-89.
6. H. WEGHORST, G. HOOPER and D.P GREENBERG (1984). Improved Computational Methods for Ray Tracing. *ACM Transactions on Graphics*, Vol. 3, No. 1, pp 52-69.