A General PASCAL Program for Map Overlay of Quadtrees and Related Problems*

F. W. BURTON†‖, V. J. KOLLIAS‡¶ AND J. G. KOLLIAS§

† *Department of Electrical Engineering and Computer Sciences, Box 104, University of Colorado at Denver, 1100 Fourteenth Street, Denver, Colorado 80202, U.S.A.*

‡ *Laboratory of Soils and Agricultural Chemistry, Athens Faculty of Agriculture, Iera Odos 75, Votanicos, Athens, Greece*

§ *Department of Electrical Engineering, Division of Computer Science, National Technical University of Athens, Zografou, 15773 Athens, Greece*

A simple but general map overlay function is defined for quadtrees. It is shown that many common quadtree operations, including* union, intersection, relative complement, masking, copy, complement *and* generalisation *of quadtrees, can all be viewed as special cases of map overlay, depending on how pixel values of the result of an overlay are determined from the corresponding pixels of the maps being overlaid.

It is shown that some operations, such as* union, *can be more efficiently performed by specialised algorithms. A more general overlay function is given, which is optimal (to within a constant factor) with respect to both time and space for all of the above operations. This overlay function requires two functions as parameters. One function determines when the quadtrees being overlaid are simple enough to be processed without further subdivision. The other performs the overlay for simple cases.

Received January 1986, revised April 1986

1. INTRODUCTION

There are many ways of representing a map in a computer.[3] We shall regard a map as an image. A map may be partitioned into a large number of small squares, called pixels. Associated with each pixel is a value identifying the theme or district in which the pixel belongs. A district within a map may correspond to a state-defined area (e.g. county), to a soil type (e.g. Alfisol) or to an elevation range (e.g. 600–700 m). The map may be represented by an array of pixel values. However, the above approach may be impracticable, since the array normally requires a very large amount of storage. For example, the British National Grid is a 1000 kilometre square divided into 10^{12} 1 metre square cells. This problem may be substantially reduced if the quadtree data structure is used.

Consider the map in Fig. 1, where $1, 2, \ldots, 5$ denote district numbers. Fig. 2 is an approximation of Fig. 1.

The map representation in Fig. 2 can be transformed to that of Fig. 3 by applying the following process. The pixel array is divided into four square subarrays. If a subarray is constant (i.e. belongs entirely to a district) the subarray can be represented by a single value. Otherwise, the subarray is again quartered into four small subarrays, in a recursive fashion. The quartering may continue down to the level of individual pixels which of course are constant.

We will restrict our attention to pixel arrays which contain $2^k \times 2^k$ elements for some integer k. (Any array can be embedded in such an array. The cost of using an overly large array as the basis for the structure defined below is usually not significant.)

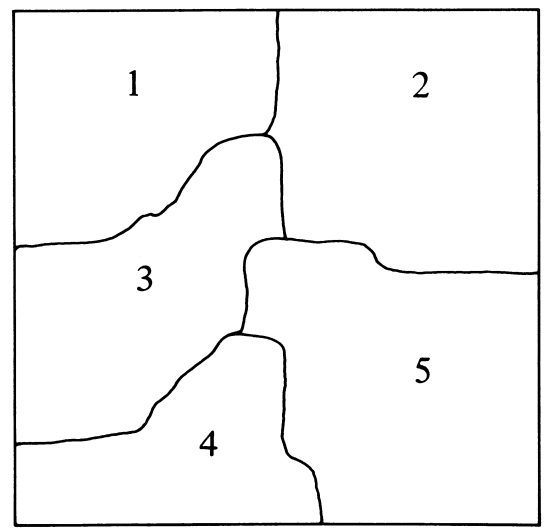

Figure 1. A map.

1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2
1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2
1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2
1	1	1	1	1	1	1	1	2	2	2	2	2	2	2	2
1	1	1	1	1	1	3	3	2	2	2	2	2	2	2	2
1	1	1	1	1	3	3	3	2	2	2	2	2	2	2	2
1	1	1	1	3	3	3	3	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	5	5	5	5	2	2	2	2	2
3	3	3	3	3	3	3	5	5	5	5	5	5	5	5	5
3	3	3	3	3	3	3	5	5	5	5	5	5	5	5	5
3	3	3	3	3	4	4	4	5	5	5	5	5	5	5	5
3	3	3	3	3	3	4	4	5	5	5	5	5	5	5	5
3	3	3	3	4	4	4	4	5	5	5	5	5	5	5	5
4	4	4	4	4	4	4	4	5	5	5	5	5	5	5	5
4	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5
4	4	4	4	4	4	4	4	4	5	5	5	5	5	5	5

Figure 2. Pixel array approximating a map.

* This material is based upon work supported by the National Science Foundation under grant no. E08-8312748. Travel support was provided by NATO under collaborative research grant no. 401/84.

‖ Now at: Department of Computer Science, 3180 MEB, University of Utah, Salt Lake City, Utah 84112, U.S.A.

¶ To whom correspondence should be addressed.

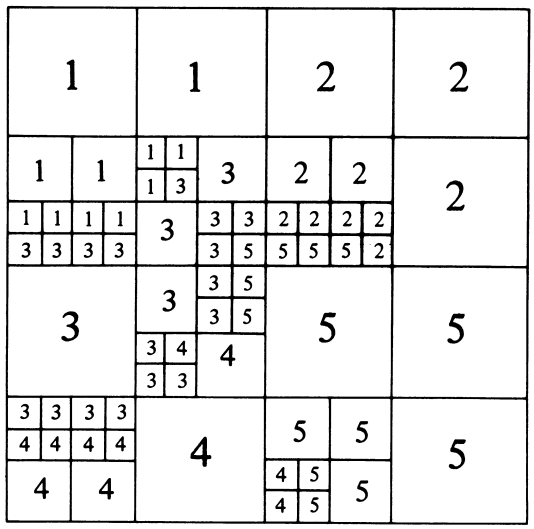

Figure 3. A compact representation for a map.

```
type
  quadtree = ^quadrec;
  quadrant = (nw, ne, sw, se);
  children = array[quadrant] of quadtree;
  quadrec = record
    case leaf: boolean of
      true: (value: integer);
      false: (child: children)
    end;
```

Figure 4. Type *quadtree* and related types.

A quadtree is defined as a recursive Pascal record type as shown in Fig. 4.

A quadtree is a degree-four tree of height k or less. Each node of a quadtree corresponds to a square array or subarray of pixels. If all of the pixels in the subarray have the same value, the node is a leaf (*leaf* is *true*) and the value field contains the common value. Otherwise (*leaf* is *false*) the array is partitioned into four subarrays (nw for northwest, ne for northeast, sw for southwest and se for southeast). Fig. 5 is the quadtree representation of the map in Fig. 1.

We shall call this quadtree data structure an integer quadtree, since the values appearing on it are integers. (We can have quadtrees of any type.) If the values of the pixel array (Fig. 2) are restricted to the values 0 and 1, the pixel array may be used to represent a geographical region, and its corresponding quadtree is a 0–1 quadtree. For the rest of this paper we shall use the term quadtree to mean integer quadtree.

With reference to the quadtree data structure the following remarks will be made. First, Ref. 9 and the references therein provide an excellent account of the many research studies and applications of quadtrees. Secondly, in Ref. 2 we have proved that each thematic map requires for storage a quadtree with $0(n)$ nodes, where n is the number of pixels in the boundary of maps. The fact that, in general, $n \ll 2^k \times 2^k$ guarantees the utility of the structure.

2. THE MAP OVERLAY PROBLEM

In a computerised geographical processing system it is often useful to overlay different types of site data to produce some kind of composite map. For example, given a topographic map, a ground cover map and a hydrological map, it might be desired to overlay them to find areas suitable for building as defined by the criteria that slope should be less than 30%, existing stands of trees should not be disturbed and buildings should not be located on a flood plane.[7] This problem is commonly referred to as the map overlay problem.

Simple, usually recursive, algorithms for map overlay and similar problems are well known.[5, 8] We shall show that a single simple algorithm can solve a number of related problems if it is supplied with appropriate functional parameters. While our examples are in Pascal, the approach is even more suitable for languages, such as Ada, which allow generic or polymorphic functions to be defined.

We shall assume that we have two maps partitioned into districts in two different ways and that each map is structured as a quadtree. Figs 3 and 6 present two such maps. For example, Fig. 3 may correspond to a soil map and Fig. 6 may be a political map. Our aim is to construct the overlay map which is also structured as quadtree and defines a new district for each (soil type, political unit) pair.

For each new district, we must produce a name as a function of the two defining districts. One simple way to do this is to multiply the number of the first district by a factor (say 10), and add the number of the second

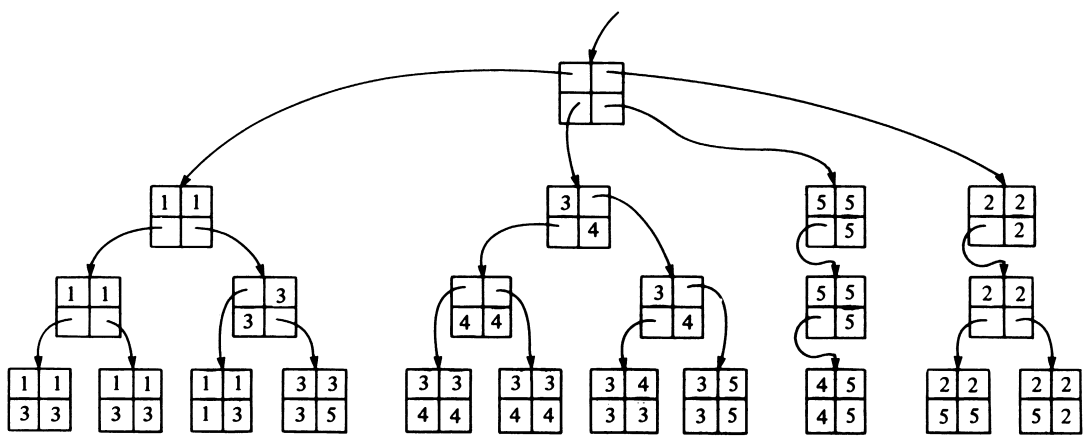

Figure 5. The quadtree representation for a map.

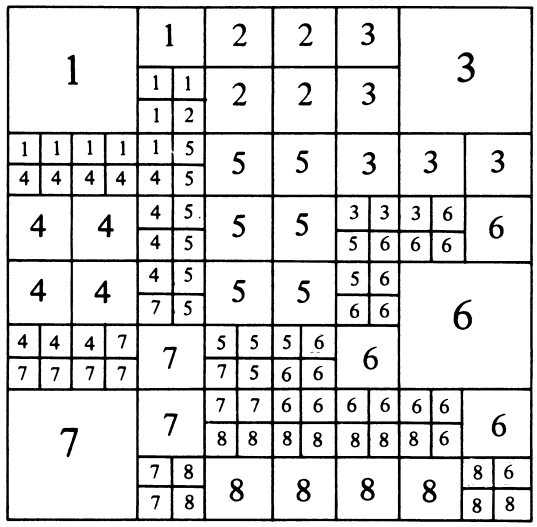

Figure 6. Another map.

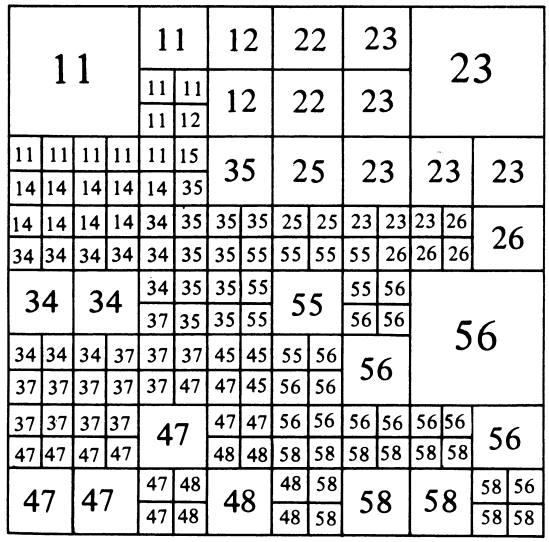

Figure 7. The overlay map for the maps in Figs 3 and 6.

district. In this way the new district with number 23 would be the intersection of (soil) district 2 with (political) district 3. Fig. 7 gives the overlay map for the maps in Figs 3 and 6 by taking the factor as 10.

Many other problems may be regarded as variants of the map overlay problem. For example, we might want to produce a 0–1 map, where a 1 indicates that the (soil type, political unit) pair satisfies some particular condition. In this case, in certain portions of the map, the leaves of the result of the overlay operation may be at a higher level than the leaves of the arguments.

In general, we want to be able to give the map overlay function an arbitrary function expressing how district pairs should be mapped to new district names. This function should be an argument to the map overlay function.

For example, we might pass the map overlay function the function *combine*, defined by

```
function combine (x, y: integer): integer;
begin
  combine := 10*x+y
end:
```

to perform the first type of overlay. On the other hand, if we want a 0–1, map where 1 indicates soil type 2 and political unit 3, we might pass the overlay function the function *select* defined by

```
function select (x, y: integer): integer;
begin
  if x = 2 and y = 3 then select := 1
  else select := 0
end;
```

In this second case, if the maps in Figs 3 and 6 are overlaid, all the pixels outside the northwest quadrant have value 0 and can therefore be represented by three leaves.

Before presenting the overlay function, we will introduce two useful quadtree manipulation functions. These are given in Fig. 8.

Given a quadtree, we will often want the quadtree representing a particular quadrant of the original. If the original is not a leaf, we select the appropriate subtree. Otherwise, the quadrant may be represented by the same leaf as the original. Function 'quarter' in Fig. 8 performs this simple computation.

In many cases, in performing an overlay we will produce a quadtree consisting of four identical leaves. These can be combined into a single leaf by the function *reduce* given in Fig. 8. Since storage management varies between Pascal implementations, we note the location where the *quadrecs* representing the original leaves should be returned to free storage if appropriate. We will assume

```
function quarter(a: quadtree; q: quadrant): quadtree;
begin
  if a^.leaf then
    quarter := a
  else
    quarter := a^.child[q]
end;

procedure reduce(var a: quadtree);
var
  v: integer;
  same: boolean;
  q: quadrant;
begin  if a^.child[nw]^.leaf then
    begin
      v := a^.child[nw]^.value;
      same := true;
      for q := ne to se do
        with a^.child[q]^ do
          if leaf then
            same := same and (value = v)
          else
            same := false;
      if same then
        begin
          {Free children if required by storage
                                          management}
          a^.leaf := true;
          a^.value := v;
        end
    end
end;
```

Figure 8. Useful quadtree manipulation functions.

that arguments passed to the overlay function never have nodes with four identical leaves as children, and ensure that the result of the operation is similarly well behaved.

The map *overlay* function is given in Fig. 9. Notice that *reduce* is called just before the function returns its result. This ensures that sets of identical leaves are combined at each level before higher-level nodes in the same part of the tree are considered.

```
function overlay(function f: integer;
    a, b: quadtree): quadtree;
var
  result: quadtree;
  q: quadrant;
begin
  new(result);
  result^.leaf := a^.leaf and b^.leaf;
  if result^.leaf then
    result^.value := f(a^.value, b^.value)
  else
    begin
      for q := nw to se do
        result^.child[q] := overlay (f,
            quarter(a, q), quarter(b, q));
      reduce (result)
    end;
  overlay := result
end;
```

Figure 9. A general map overlay function.

The function application

$$overlay(combine, a, b)$$

returns an overlay map computed from the two maps a and b. For example, given the quadtree representation of Figs 3 and 6 the quadtree representation of Fig. 7 is produced. Similarly,

$$overlay(select, a, b)$$

constructs from the quadtrees of Figs 3 and 6 a 0–1 quadtree having the value 1 only to the nodes which satisfy the selection criterion (i.e. soil type = 2 and political unit = 3).

The overlay algorithm is optimal to within a constant factor with respect to both time and space if the function f is arbitrary with no known characteristics. It is clearly necessary to inspect all nodes of both trees in order to compute the result of a map overlay. The time required by this algorithm is clearly bounded above and below by a constant times the combined sizes of the arguments. (We never go to a lower level of recursion unless at least one of the arguments has lower-level nodes to be considered.) As for space, apart from the space required by the result, storage requirements are proportional to the depth of recursion. (This includes the storage required for up to three leaf nodes at each level of recursion, which will later be combined into a single leaf when the fourth node is produced and found to be an identical leaf.)

3. OTHER USAGES OF THE ALGORITHM

We shall now demonstrate how the procedure *overlay* may produce other useful map operations by merely changing the function *combine*.

3.1. Masking

Suppose that we have a quadtree a, representing a map. Suppose further that we want to produce another quadtree which corresponds to some part of the map. Assume that the part of interest has been expressed by the 0–1 quadtree b. It is easy to see that the call

$$overlay(mask, a, b)$$

produces the desired operation provided that we have defined the following function:

```
function mask(x: integer, y: zo): integer;
begin
  mask := x*y
end;
```

In the above function we assume that the data type *zo* (for zero-one) has been defined globally with the statement

$$zo = 0..1;$$

3.2. Intersection, Union and Difference

The following three functions apply only to 0–1 quadtrees, which may represent geographical regions. Given two such quadtrees a and b it is useful to be able to compute the intersection, union and difference of the regions. We can do this by passing *overlay* the appropriate function out of the following:

```
function intersection(x, y: zo): zo;
begin
  intersection := x*y {Same as mask}
end;

function union(x, y: zo): zo;
begin
  union := x+y-x*y {Pascal has no max function}
end;

function difference(x, y: zo): zo;
begin
  difference := x+y-2*x*y
end;
```

3.3. Single-Argument Operations

There are many single-argument quadtree operations, such as *copy* and *complement*. Of course these can be performed using *overlay* if the second argument is ignored. For example, to compute the complement of a 0–1 quadtree a we write:

$$overlay(complement, a, a)$$

where

```
function complement(x, y; zo): zo;
begin
  complement := 1-x
end;
```

The code for *copy* is similar.

Another useful variant of the map overlay problem is the map generalisation problem. If a pixel of the single quadtree passed to the generalisation function has value v, then the value of the corresponding pixel of the result must have the value $F(v)$ for some function F. The

function F may be a many-to-one function. Map generalisation may be used to reverse a map overlay. For example given

```
function reverse(x, y: integer): integer;
begin
  reverse := x mod 10
end;
```

the expression

overlay(*reverse*, *a*, *a*)

will produce the quadtree of Fig. 6 if a is the quadtree of Fig. 7. We can generate the quadtree of Fig. 3 from that of Fig. 7 by

overlay(*remainder*, *a*, *a*);

where

```
function remainder(x, y: integer): integer;
begin
  remainder := x div 10
end:
```

4. A MORE GENERAL OVERLAY ALGORITHM

In Section 2 we presented a general overlay function which was optimal to within a constant factor, with respect to both time and space, if f is an arbitrary function with no known properties. However, the functions *union* and *intersection* have properties which can lead to improved efficiency (e.g. The intersection of the empty set with anything is the empty set). If we change the problem statement slightly, further savings are possible.

The *overlay* function of Section 2 always generates a new quadtree as a result. However, if subtrees can be shared between various quadtrees, substantial savings may result in some cases. For example, if a is an arbitrary quadtree, the intersection of this quadtree with a quadtree consisting of a single leaf (which is also the root) having value 1 is a tree identical to a. If we are allowed to return a pointer to the root of a, rather than a copy of a, this operation can be performed in constant time and space. Otherwise, the time and space required to produce a copy of a is proportional to the size of a.

With shared substructures, care must be taken never to modify or free a shared node. Reference counts[6] may be used to detect shared nodes if necessary. If a scan-mark garbage collector is used, the user need only avoid modifying quadtrees which might be shared.

To take advantage of the possibility for shared subtrees the overlay function must become a little more complicated.

With problems such as computing the intersection or union of 0–1 maps, the recursive descent can stop as soon as either argument is a leaf. With masking, it is necessary to descend until the second argument is a leaf. Finally, with overlays in general, it may be necessary to descend until both arguments become leaves. Hence we need a more general method for stopping the recursive descent than the test

$a^{\wedge}.leaf$ **and** $b^{\wedge}.leaf$

used in Fig. 9. In addition, when descent is halted, it may be necessary for the function f to return a quadtree, rather than just a leaf value. With these changes in mind *overlay* may be generalised as shown in Fig. 10.

```
function overlay2(function p: boolean;
    function f; integer; a, b: quadtree): quadtree;
var
  result: quadtree;
  q: quadrant;
begin
  if p(a, b) then
    overlay2 := f(a, b)
  else
    begin
      new(result); result^.leaf := false;
      for q := nw to se do
      result^.child[q] := overlay2(p, f,
          (quarter(a, q), quarter(b, q));
      reduce(result);
      overlay2 := result
    end
end;
```

Figure 10. An overlay function that produces shared subtrees.

5. EXAMPLES OF OVERLAY WITH SHARING

In the case of a map overlay such as the one which produces Fig. 7 from Figs 3 and 6, no sharing is possible and no advantage is gained by using *overlay2*. However, we can still perform this computation by

overlay2 (*bothsimple*, *combine2*, *a*, *b*)

with

```
function bothsimple(a, b: quadtree): boolean;
begin
  bothsimple := a^.leaf and b^.leaf
end;
```

and

```
function combine2(a, b: quadtree): quadtree;
var result: quadtree;
begin
  new(result);
  result^.leaf := true;
  result^.value := a^.value*10+b^.value;
  combine2 := result
end;
```

where a and b are quadtrees representing the maps of Figs 3 and 6.

Let us consider some examples where use of *overlay2* may be of advantage.

5.1. Union and Similar Problems

We can compute the union of two 0–1 quadtrees by

overlay2(*eithersimple*, *union2*, *a*, *b*)

where

```
function eithersimple(a, b: quadtree): boolean;
begin
  eithersimple := a^.leaf or b^.leaf
end;
```

and

```
function union2(a, b: quadtree): quadtree;
begin
  if a^.leaf then
    begin
      if a^.value = 1 then union2 := b
      else union2 := a
    end
  else {b^.leaf must be true}
    begin
      if b^.value = 1 then union2 := b
      else union2 := a
    end
end;
```

Computation of intersections, differences and so forth is similar. It is clear from the examples in Section 4, where one argument to an intersection computation is arbitrarily large and the other is a leaf, that in general the use of *overlay2* (and hence the generation of shared subtrees) may reduce the time and space requirements from those proportional to the size of the large argument to a constant. Let us consider another situation.

In Ref. 2 it is shown that the size of a quadtree representing a region is on average proportional to the number of boundary pixels. If the resolution of an image is doubled, then so is the size of the boundary, in most cases. If a map is defined as an $n \times n$ pixel array, then the size of a quadtree, and the time required to compute the intersection of two quadtrees using *overlay* tends to be $0(n)$ as n is increased and the image is held constant. On the other hand, with *overlay2*, it is necessary to consider only those parts of an image which contain parts of the boundaries of both arguments. There tend to be a fixed number of those on each level. Since the maximal depth of a quadtree is $\log_2 n$ the time and space requirements tend to be $0(\log_2 n)$. This is a substantial saving. (It is interesting to note that it is possible to generate a result of $0(n)$ size in $0(\log_2 n)$ time in these cases, since shared subtrees are not examined.)

Sometimes it is desirable to perform operations such as 0–1 quadtree union such that the resulting quadtree contains no shared subtree. (This is useful if quadtree modifications are expected.) The solution given in Section 2 is not quite optimal, since it does not recognise that the union of a set with the universal set is always the universal set. An optimal (to within a constant) solution is given by

overlay2(*unionsimple*, *union3*, *a*, *b*)

where

```
function unionsimple(a, b: quadtree): boolean;
  function universal(a: quadtree): boolean;
  begin
    if a^.leaf then universal := a^.value = 1
    else universal := false
  end;
begin
  unionsimple := (a^.leaf and b^.leaf)
                 or universal(a)
                 or universal(b)
end;

function union3(a, b: quadtree): quadtree:
var result: quadtree;
begin
  new(result);
  result^.leaf := true;
  if a^.leaf and b^.leaf then
    result^.value := a^.value + b^.value
                     - a^.value*b^.value
  else {either a or b is the universal set}
    result^.value := 1;
    union3 := result
end;
```

5.2. Masking

The masking operation can also benefit from use of *overlay2*. To perform masking we compute

overlay2(*rightsimple*, *mask2*, *a*, *b*)

where

```
function rightsimple(a, b: quadtree): boolean;
begin
  rightsimple := b^.leaf
end;
```

and

```
function mask2(a, b: quadtree): quadtree;
begin
  if b^.value = 1 then
    mask2 := a
  else
    mask2 := b
end;
```

The time required for masking with sharing is clearly proportional to the size of the mask.

6. CONCLUSION

We have seen that a single quadtree overlay function is able to perform a number of common quadtree operations including union, intersection, difference, masking, copy, complement and map generalisation. A slightly more complicated variation of this function allows us to take full advantage of the special characteristics of each of these operations and to perform the computations in a manner which is optimal to within a constant factor, with respect to both time and space.

The approach is motivated by the functional style of programming,[4] where higher-order functions may encapsulate complex control structures. (Functional programming also suggests other ways to write simple and efficient programs.[1]) In a functional language (or even in a language such as Ada that supports generic functions) it is possible to create specific functions such as *intersection* as results of higher-order functions (or as instances of generic functions). For example, we could compute

intersection := *overlay2*(*eithersimple*, *intersection2*)

and then use the function intersection in expressions such as

intersection(*intersection*(*a*, *b*), *c*).

This saves having to include the functions *eithersimple* and *intersection* in every intersection computation. (Performing functional programming in Pascal is rather like doing structured programming in Fortran IV. It is

possible, but the result is not as pretty as might be desired.)

We note that, while we have given an optimal algorithm for computing intersections of pairs of quadtrees, our solution is not optimal if the intersection of three or more quadtrees is desired. See Ref. 1 for further discussion of this problem.

Acknowledgement

We would like to thank the referee for his helpful comments.

REFERENCES

1. F. W. Burton and M. M. Huntbach, Lazy evaluation of geometric objects. *IEEE Comput. Graph. Appl.*, **4**(1), 28–33 (1984).
2. F. W. Burton, V. J. Kollias and J. G. Kollias, Expected and worst case storage requirements for quadtrees. *Pattern Recognition Letters* **3**(2), 131–137 (1985).
3. G. Dutton (ed.), *First International Advanced Study Symposium on Topological Data Structures for Geographic Information Systems* (8 volumes). Harvard Papers on Geographic Information Systems (1979).
4. P. Henderson, *Functional Programming. Applications and Implementation.* Prentice-Hall, Englewood Cliffs, New Jersey (1980).
5. G. M. Hunter and K. Steiglitz, Operations on images using quad trees, *IEEE Trans. Pattern Analysis and Machine Intelligence*, PAMI-**1** (2), 145–153 (1979).
6. D. E. Knuth, *The Art of Computer Programming.* Vol. 1: *Fundamental Algorithms.* Adison Wesley, Reading, Mass. (1968).
7. W. G. Mitchel, *Computer Aided Architectural Design.* Petrocelli/Charter, New York (1977).
8. M. A. Oliver and N. E. Wiseman, Operations on quadtree encoded images. *The Computer Journal* **26**(1), 83–91 (1983).
9. H. Samet, The quadtree and related hierarchical data structures. *Computing Surveys* **16**(2), 187–260 (1984).