An Exact Incremental Hidden Surface Removal Algorithm

A.A.M. Kuijk, P.J.W. ten Hagen, V. Akman

Department of Interactive Systems
Centre for Mathematics and Computer Science (CWI)
P.O. Box 4079, 1009 AB Amsterdam
The Netherlands

This paper describes an incremental Hidden Surface Removal Algorithm (HSRA), developed to be embedded in a new architecture for raster graphics described in [1, 7]. The algorithm can be classified as "exact" since it operates in object space, rather than image space. It can be classified as "incremental" because this HSRA is able to support addition, removal and changes on a single object or a group of objects. Thus a firm basis for powerful interaction and animation is established. Due to specially designed data structures for both geometric objects as well as storage of these objects, the hidden surface removal calculation on a complete scene will have the same time complexity as existing algorithms. However, the effort needed for incremental changes is much less than any other known algorithm. The data structures as well as the algorithm are designed to exploit parallelism in computation.

Categories and Subject Descriptors:
C.3 *[Special-Purpose and Application-based Systems]: — real-time systems*
I.3.1 [Computer Graphics]: Hardware Architecture — *raster display devices;*
I.3.3 *[Computer Graphics]:* Picture/Image Generation — *display algorithms;*
I.3.5 *[Computer Graphics]:* Computational Geometry and Object Modeling — *curve, surface, solid, and object representations*
I.3.7 *[Computer Graphics]:* Three-Dimensional Graphics and Realism — *visible line/surface algorithms*

Key Words & Phrases: Raster Graphics, Overlap Separation, Representation, Incremental Operations, VLSI.

1. Introduction

Although the image quality of raster displays from their introduction on always surpassed the quality possible with vector displays, their widespread use in interactive environments only became possible after the introduction of —hardware implemented— techniques that allow fast incremental operations on the pixel representation of the image (bitmap operations). Since in this pixel representation all information about the origin of the particular pixels is lost, bitmap operations are not well-suited to support incremental operations needed on 3D raster graphics workstations. What is needed there is an incremental Hidden Surface Removal Algorithm (HSRA) that stores its results in a more object-oriented form, so that incremental operations can be performed more easily. This object-oriented approach enables a reference from the visible parts of objects in the image to the objects in the higher level models (up to the application model) they originated from. Thus flexible interaction can be established [1].

The incremental HSRA is designed to be embedded in a new architecture for raster graphics workstations [7]. An important aspect of this new architecture is a specially designed representation for surface elements (domains). Both input and output primitives of the HSRA are types (in the sense of "abstract data types") of that specially designed representation. With this representation overlap of two domains can be calculated efficiently. After the HSR, resulting domains are stored in the so called Low Level Display File (LDF) [1, 7], sorted in a specially designed data structure.

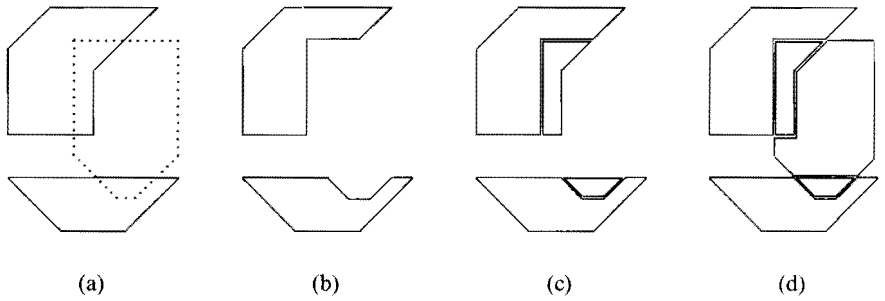


Figure 1: Example of the steps taken upon addition of a new domain: (a) Before addition there are 2 domains in the LDF —these are drawn solid, the domain to be added is drawn dotted—. (b) First the overlapping sections are removed. (c) Next the overlapping sections are added to the LDF as separate domains. (d) Finally, the new domain with overlapping sections removed is added. As a result there are 5 disjunct domains in the LDF after the addition.

The strategy followed is to add the domains making up the 3D model one by one, thereby incrementing the number of domains in the LDF. The first step taken upon addition of a domain is to check for overlap with domains already stored in the LDF. All those domains of the LDF that exhibit overlap with the domain to be added are replaced by a domain with that overlapping section removed (see Figure 1). This is independent of the fact whether the overlapping section would be visible or not. After this, the overlapping sections —removed from the domains already present in the LDF— are added to the LDF as separate domains. Of course, the appearance of these overlapping sections (attributes, colouring) is determined by the domain that is on top. Next, the domain to be added —with all overlapping sections removed— is added to the LDF. In this way, the domains that are present in the LDF will **always** be disjunct. Overlap sections are kept as separated domains, so that removal of objects can be done in arbitrary order.

It is clear that both an efficient search algorithm to identify all the domains (stored in the LDF) that overlap and an efficient algorithm to generate the domains with overlapping parts removed are needed. An economical identification of possible overlapping domains can be done by making use of the bounding box information contained in the domain representation which will be described in § 2. Thus the first step is to identify all the domains in the LDF whose bounding box overlap the bounding box of the domain to be added. This search process will be described in § 3. The domains reported by this search process will be fed to the separation function, described in § 4, which checks whether the domains themselves overlap and outputs non-overlapping, separated domains. The remainder of the algorithm is administrative work which is described in § 5; the domains resulting from the separation function have to be relocated or inserted into the data structure of the LDF and pointers to higher level representations have to be updated. A hardware implementation of the algorithm is discussed in § 6.

2. Domain Representation

Before we discuss the functions needed, it is useful to explain some basic ideas of the domain representation itself.

A domain is the geometric entity of an object in 3D space, and is not rasterized. It is one of the two components of a pattern as described in [5, 6]. The other component of a pattern —the colour function— assigns a colour to each point of the domain. Since most of the operations involved in HSR calculations are pure sorting operations [9] it will not be surprising that the domain representation which is designed for efficient HSR, contains the geometrical data in a presorted form, free of all sorts of pathological cases normally encountered.

The representation envisaged consists of a header and a number of scanlines and scanpoints (see Figure 2). In the header, global information about the domain, such as bounding box, plane equation and information about scanlines, scanpoints (location and number of) and administrative information are stored. Scanlines are lines at those y-values of the domain where, while sweeping over a polygon in y-

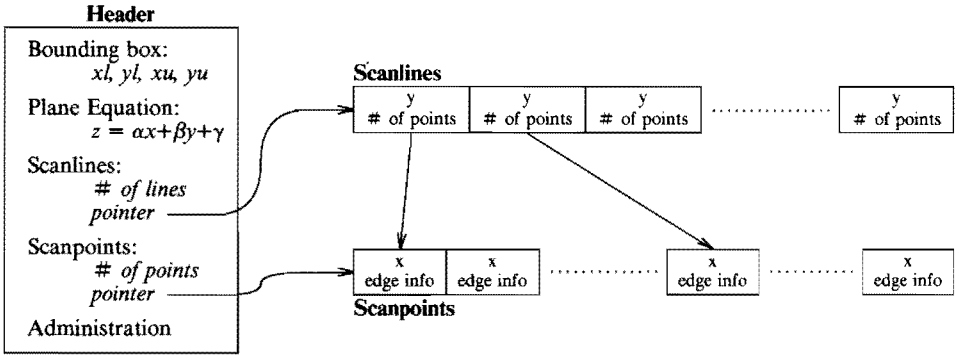
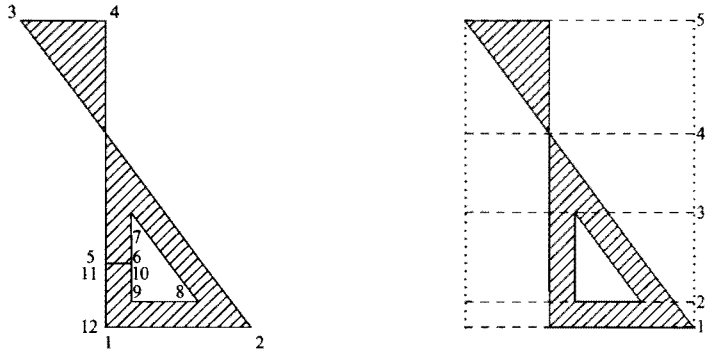


Figure 2: The domain representation contains geometrical data, sorted to enable more efficient algorithms.

direction —like in the scanline algorithm described in [3]— the active edge list would have to be updated (Figure 3). Each scanline has a number of scanpoints which contain information with which the topology of the area between this and the next scanline is unambiguously fixed. At present, a scanpoint contains the x-value where an edge crosses the scanline and the slope of this edge. A future enhancement could be that a scanpoint contains more information about the edge, such as the curvature or appearance control attributes.



Vertex representation Domain representation

Figure 3: Traditional vertex representation versus domain representation. Note that there is no scanline at the "double bridge" between vertices 5,6 and 10,11. On the other hand, there is a scanline at the crossing of the edges between vertices 2,3 and 4,5.

By conversion from traditional boundary representation (a list of vertices) to our domain representation on the one hand all sorts of superfluous information will be removed, but on the other hand information can be added to avoid pathological cases (see Figure 3). With this representation, flat regions in continuous 3D space can be represented without loss of information and without limitations on domain complexity (i.e. connected or disconnected, convex or concave, with holes or islands).

We implemented an algorithm to generate a domain representation from a traditional polygon representation, based on the sweepline technique. It can be seen as a modified version of the scanline algorithm presented in [3]. Modifications were needed to make the algorithm operate in continuous coordinates, to generate scanlines only at strategic y-values and to remove superfluous information (such as double edges, vertex on a straight edge). Thus the following algorithm emerged[†]:

```
make edgelist sorted on y_bot else on x_bot else on slope
initialize y on lowest y_bot encountered during sort
while ∃ edges in sorted list do {
    move edges with y_bot = y from sorted list into active list sorted on:
                                    x_current
                                    else on slope
                                    else remove redundancy
    generate scanline (y)      /* based on active list */
    remove edges with y_top = y from active list
    y-next = minimum (y_bot of edges in sorted list, y_top of edges in active list)
    update x_current of edges in active list
    while ∃ disordered x_current in active list do {
        y-cross = minimum (y-crossings)
        generate scanline (y-cross)
        exchange edges crossing at y-cross in active list
    }
    y = y-next
}
```

The domain representation will cause the following frequently occuring, time-critical operations in the image generation process to be handled more efficiently:

- Overlap calculation of two domains. This can be done on a slice by slice basis, thereby reducing the time needed from $O(n \cdot m)$ to $O(n+m)$ for domains having n and m scanlines ($\simeq$vertices) respectively.
- Clipping. This can be seen as a special sort of overlap calculation and can be performend in $O(n)$ time. Clipping in y-direction is a trivial operation. Simply

[†] N.B. The algorithm presented here scans from bottom to top.

all scanlines below or above the clipping boundary can be removed from the representation. Only the two scanlines on the clipping boundaries may have to be changed or added. The scanline that may have to be added on the low clip boundary, can be generated easily; only the information of the scanline just under the clip boundary is needed. The terminating scanline to be added on the top clip boundary is empty. Clipping in x-direction is simplified as well, due to the ordering of the scanpoints.

- A point-in-domain test. This test will be an $O(n)$ process; it requires looking for the scanline that describes the slice the point is in.
- Scan conversion. It will be clear that this representation is extremely well-suited for the scan-conversion process.

Note that these improvements in time bounds hold for sequential algorithms. A further reduction of these time bounds can be obtained by paralellising the algorithms. They can be parallelised easily due to the fact that the domain representation enables sub-algorithms that work on slices independently.

3. Searching for Overlapping Bounding Boxes

As mentioned in the introduction, it is needed to find among a set of n rectangle bounding boxes, the subset of bounding boxes that overlap the bounding box of a domain to be inserted. This subset of overlapping bounding boxes indicate possible overlapping domains.

Optimal algorithms to report from a set of n rectangles all overlapping pairs are well-known from computational geometry [2, 8, 10] and need $O(n \log n + k)$ time and $O(n)$ storage worst case, k being the number of overlaps found. These optimal algorithms however, cannot be used here since we are not interested in pairs of overlapping bounding boxes in the LDF itself*. These algorithms —already difficult to implement due to the sophisticated data structures used— also do not lend themselves to be converted to an algorithm reporting the subset of n rectangles overlapping one query rectangle in $O(\log n + k)$ time. Therefore we tried to find a suitable solution (that is, optimal under normal conditions and easy implementable in VLSI) operating in better than $O(n)$ time expected case. We think that the following data structure and algorithm will be such a suitable solution.

* Bounding boxes of domains stored in the LDF may overlap, but the domains themselves are definitely disjunct.

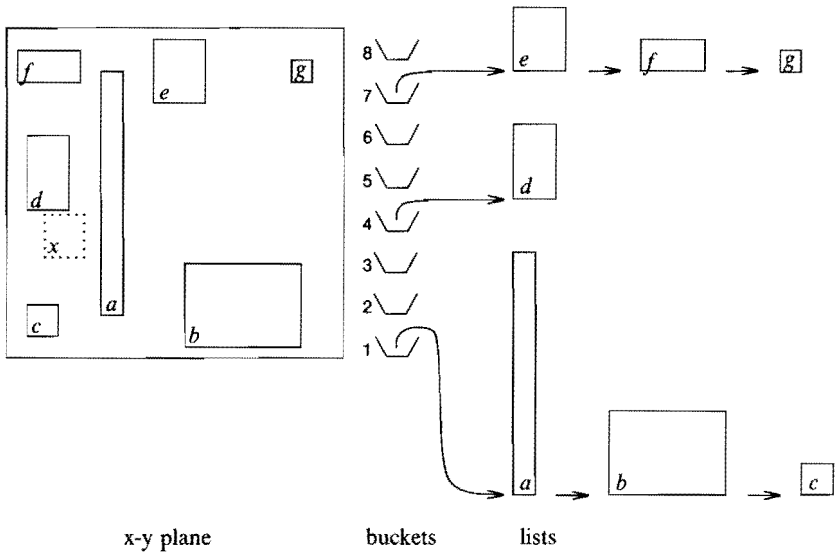


Figure 4: The data structure in which the domains resulting from the HSR are stored. It is designed to avoid unnecessary visiting of domains outside a range of y-values when searching for bounding boxes overlapping a query rectangle. In this example (a search with query rectangle x) buckets 1-4 and accordingly, domains a, b and d will be visited.

The data structure envisaged (Figure 4) consists of a number of buckets, each containing a sorted list of domains. Thus the domains can be stored, sorted on two criteria: 1) The bottom y-value of the domain (y_{bot}) determines the bucket where a domain is to be stored; 2) The top y-value (y_{top}) determines the location of the domain in the linked list of decreasing y_{top}. The intention of this data structure is to be able to quickly address only those domains that have an overlap in y-direction with the query rectangle[‡].

The following algorithm will report all domains having a bounding box overlapping the bounding box of a query domain:

[‡] It seems that for the scan conversion hardware it is better to interchange x and y. For the HSRA such a change is irrelevant.

```
bucket = 1
while bucket range < query-domain.y_top do {
        domain = first-domain                    /* in list of bucket */
        while domain.y_top > query-domain.y_bot do {
                if overlap in x-direction † do {
                        report domain
                }
                domain = next in list            /* if no next, break inner while loop */
        }
        bucket = bucket + 1
}
```

It is obvious that the data structure proposed needs $O(n)$ space, since all domains are stored just once. The time complexity of the algorithm will be $O(n)$, in the worst case all n (already stored) domains have to be checked for overlap.

More interesting however, is the asymptotical behaviour in the expected case. For this, let us assume that we have the total y-space (normalised on 1) covered by m buckets. The average height of the n domains is h_y (n is big and h_y is small) and we assume that the domains are evenly distributed in the y-space. Then the average length of the lists in the buckets will be $\frac{n}{m}$.

Consider a query rectangle of average height h_y and having y_{bot} in the range covered by bucket k, not too close to the y-space limits. In order to check all candidate domains with $y_{bot} \geqslant y_{bot}$ of the query rectangle, all items in buckets k to $k + m \cdot h_y$ have to be visited, resulting in an average number of $n \cdot h_y$ items to visit. Similarly, checking all candidate domains having a $y_{bot} <$ the y_{bot} of the query rectangle, on the average takes visiting $n \cdot h_y$ items. Thus the average number of steps to take in total is $2n \cdot h_y$. Since this is equal to the average number of domains in a slice of height $2h_y$, we obtained an optimal reduction of the search space in y-direction. Unfortunately it is not as simple to obtain a similar reduction of the search space in x-direction simultaneously.

Because it is impossible to continue increasing n, without reducing the average size of the domains (note: they are disjunct), we observe that h_y (the average height of the domains) will be inversely proportional to $\sqrt{n}$. As a result, the search algorithm will have an asymptotical behaviour of $O(\sqrt{n})$, a result that is acceptable in our situation. The above reasoning shows that the behaviour of the algorithm is primarily dependent on the average height of the domains, apparently not on the

† To avoid reporting non-overlapping rectangles while scanning the last bucket for which bucket range $<$ *query-domain.*y_{top}, also overlap in y-direction has to be checked. This is because each bucket contains a range of y_{bot}-values, so that in this last bucket it could be that *domain.*$y_{bot} >$ *query-domain.*y_{top}.

number of buckets. However, in the above analysis the influence of the bucket width on the number of steps has not been taken into account. Unfortunately, this influence is such that an optimal number of buckets is dependent on the particular domain arrangement, so that an adaptive number of buckets would be desirable. This would imply a restructuring whenever we are too far from optimal. This is a variant of the adaptive grid solution as described in [4].

To be able to compare this result with the optimal algorithms mentioned, we estimate the total time spent by the search algorithm when n domains are added one by one, each addition initiating a search for overlap. Then the average number of steps in total will be $2(1 + 2 + \cdots + (n-1)) \cdot h_y = (n^2 - n) \cdot h_y$ *.

Again assuming h_y to be inversely proportional to $\sqrt{n}$, the total time spent by the search algorithm will be $O(n\sqrt{n})$ in the expected case. We conclude that in worst-case situations, our solution may not be optimal, but is easy to implement and it is relatively cheap to perform insert and delete operations on the data structure. On the other hand, optimal algorithms are rather difficult to implement and have less flexible data structures.

4. Overlap Separation

When adding a new domain to the LDF, all overlaps with domains already stored in the LDF have to be separated from it. This operation is performed by the separation function. This separation function has two input and four output domains (some of which may be NULL) as shown in Figure 5. One of the input domains (*Area* 1) is the new domain. The other input domain (*Area* 2) is a domain from the set of domains reported by the search algorithm. Output of the separation function are the four domains *Dif* 1, *Dif* 2 and *Olp* 1, *Olp* 2. *Olp* 1 and *Olp* 2 together form the complete overlap of the two input domains, *Olp* 1 is the part of the overlap where *Area* 1 is nearest to the viewer and *Olp* 2 is the part of the overlap where *Area* 2 is nearest to the viewer. *Dif* 1 and *Dif* 2 are the remaining non-overlapping parts of the two input domains *Area* 1 and *Area* 2 respectively.

Output domains *Olp* 1, *Olp* 2 and *Dif* 2 will replace *Area* 2 in the LDF, while *Dif* 1 (the reduced input domain) will be input domain *Area* 1 for the next separation process with as second input domain *Area* 2, the next domain from the set reported by the search algorithm. This process will be repeated until all domains from the set reported by the search algorithm have been handled, each time possibly causing a reduction of the new domain. Finally, this multiple-reduced new domain can be stored in the LDF (see also Figure 8).

Note that an input as well as an output domain of the separation function

* In case overlaps are reported, domains are split up, so that due to the increased number of domains in the LDF this result would have to be corrected with a factor $(1 + \frac{k}{n})$, k being the number of overlaps reported. This results in $(n + k) \cdot (n - 1) \cdot h_y$ steps to take. However, assuming k to be $O(n)$, this correction factor will be $O(1)$.

may consist of disconnected areas and —depending on overlap— some output domains may turn out to be empty. Due to the domain representation, the separation function is insensitive to object complexity (e.g. polygons with holes, self-crossing boundaries).

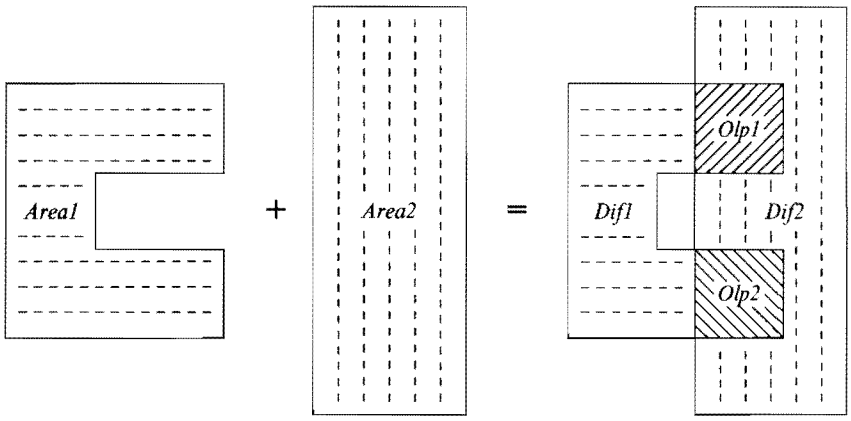


Figure 5: The principle of the separation function —the basic function of the incremental HSRA. Two input domains *Area* 1 and *Area* 2 result in four output domains. Two output domains *Dif* 1 and *Dif* 2 are copies of the input domains with overlap removed. Output domains *Olp* 1 and *Olp* 2 are those parts of the overlap for which *Area* 1 and *Area* 2 respectively are nearest to the viewpoint.

5. Administrative Aspects

Domains resulting from the separation function will have to be inserted into the LDF. Their location in the LDF data structure is fully determined by their geometric properties. Therefore in principle it is always possible to use knowledge of these geometric properties when looking for a domain in the LDF data structure. However, this may not be the most economical way to look for domains. Input domains for the HSRA are stored in the Medium Display File (MDF) [1], not sorted upon their geometric properties. So, in order to be able to find in the MDF the so-called "parent" domain that a "child" domain in the LDF originated from, an administration of the relations of the domains from these two levels of display files has to be maintained.

This is done as follows. Each parent domain from the MDF is augmented with a list of pointers to all child domains in the LDF that emerged from it[‡], marked with one of the following labels: *original, hiding* or *hidden* (Figure 6). A parent can

[‡]This list of pointers could in principle be of size $O(n^2)$, but in practice (i.e. CAD applications) it will be of a limited size. It can always be forced to be of limited size because for interaction support it does not always make sense to maintain a full overlap administration.

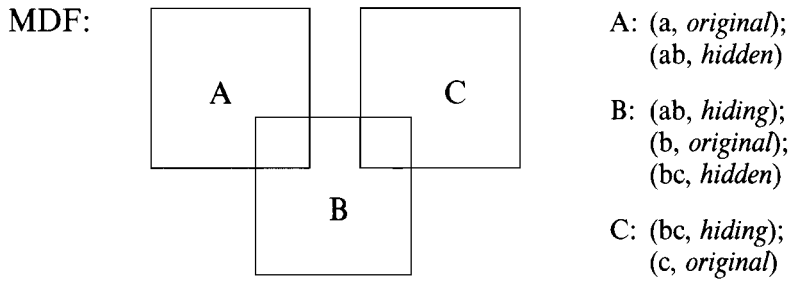


Figure 6: Administration information of domains stored in the MDF. Each domain has a list of its children in the LDF and marked whether these are *original, hiding* or *hidden.* Domains A and B share child ab and domains B and C share child bc.

have only one child marked *original.* This child represents that part (or whole) of the parent domain that is not overlapped by and is not overlapping any of the other MDF domains. Children representing that part of the parent that is not overlapped by any other MDF domain, but overlap at least one domain are marked *hiding.* Children representing that part of the parent that is overlapped by at least one MDF domain are marked *hidden.*

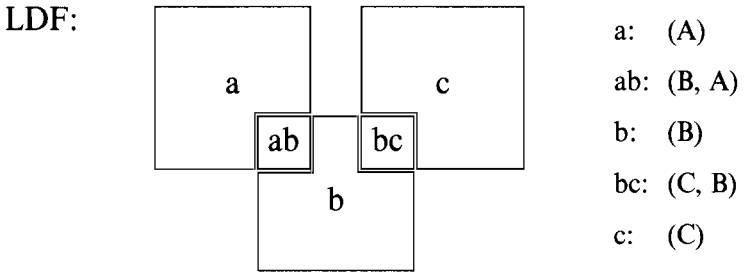


Figure 7: Administration information of domains stored in the LDF. Each domain has a list of its parent domains in the MDF, sorted on visibility.

Each child domain in the LDF in turn is augmented with a list of pointers to the parent domains in the MDF it originated from (Figure 7), ordered by visibility (i.e. distance to the viewer).

Figure 6 and Figure 7 show the situation for three overlapping domains. Of these three domains, A is overlapped by B, which in turn is overlapped by C.

Adding or removing domains will be initiated in the MDF. Consequently, upon these changes, an update of the domains stored in the LDF as well as an update of the administration has to be done.

5.1. Adding a Domain

With the discussion of the separation function in § 4 we explained how a parent domain that is added to the MDF will generate the disjunct child domains to be installed in the LDF. Here we show how to update the administration of relations as needed after such a separation process.

```
if Area 1 from MDF do {          /* first separation process, no children yet */
      Dif 1.parent-list = (Area 1)
      Olp 1.parent-list = (Area 1, Area 2.parent-list)
      Olp 2.parent-list = merge of Area 1 and Area 2.parent-list

      Area 1.child-list = (Dif 1, original); (Olp 1, hiding); (Olp 2, hidden)
}
else do {                        /* Area 1 is reduced before ⇒ is not parent */
      Dif 1.parent-list = (Area 1.parent-list)
      Olp 1.parent-list = (parent-of-Area 1, Area 2.parent-list)
      Olp 2.parent-list = merge of parent-of-Area 1 and Area 2.parent-list

      replace parent-of-Area 1.child-list item (X, original)
                  by (Dif 1, original); (Olp 1, hiding); (Olp 2, hidden)
}

Dif 2.parent-list = Area 2.parent-list

∀ parents-of-Area 2 do {
      replace child-list item (Area 2, x)
                  by (Dif 2, x); (Olp 1, hidden); (Olp 2, x)
}
```

5.2. Removing a Domain

One aspect of the claimed incremental behaviour of this HSRA is that it does not only support addition of domains, but it supports removal of domains as well. Due to the splitting of domains into parts that are original and parts that hide others and the administration of relations between the parent domains from the MFD and the child domains from the LDF, removal of domains can be done without having to fear for an avalanche of domains to be recomputed. Since up to now the main attention has been focussed on addition of domains, we will first address some aspects of removing domains.

When removing a domain from the MDF, its child domain marked *original* has to be removed from the LDF and the administration concerning all children left has to be updated. Although this would cover the necessary steps to produce a correct image, it is clear that in this way a lot of unnecessary split-up domains in the LDF will arise from subsequent removal operations. After a removal operation,

child domains having an identical list of parent domains, can be "glued" together. These glue operations are the inverse of the separation operations to be performed upon addition of the domain just removed. Their computational complexity however, is much less. By making use of the knowledge that domains to be glued will "fit", such glue operations can be optimized.

Upon removal of *Domain*, the following steps to update the administration have to be taken.

```
∀ children in Domain.child-list do {
    if child.marking == original do {
        remove child from LDF
    }
    if child.marking == hiding do {
        remove Domain from child.parent-list*
        if one parent left do {
            change parent's marking of this child to original
        }
        else {
            change first parent's marking of this child to hiding
        }
    }
    if child.marking == hidden do {
        remove Domain from child.parent-list‡
        if one parent left do {
            change parent's marking of this child to original
        }
    }
}
```

6. Hardware Aspects: Parallelism

Now that we have discussed the different functions needed for the HSRA, we can raise the question: How to implement this most effectively? For this it is useful to have a look at the functional model shown in Figure 8. From this model we can see that the most time critical part of the HSRA will be the separation function. This function will be used repeatedly and has to perform the most computationally intensive task: the separation of two overlapping domains into disjunct non-overlapping domains. It is clear that this function is an obvious candidate for a

* Since for this child, *Domain* is the first parent in the list and thus determines its appearance, removal of *Domain* causes a change in the appearance of *child* (this will now be determined by the next parent in the list).

‡ Here *Domain* is not the first parent in the list, so removal of *Domain* will not change the appearance of *child* (assuming that we are not dealing with transparent objects).

parallelised approach. The model also shows that the search and administration module cannot simultaneously have access to the LDF, so these functions can be performed by one processor.

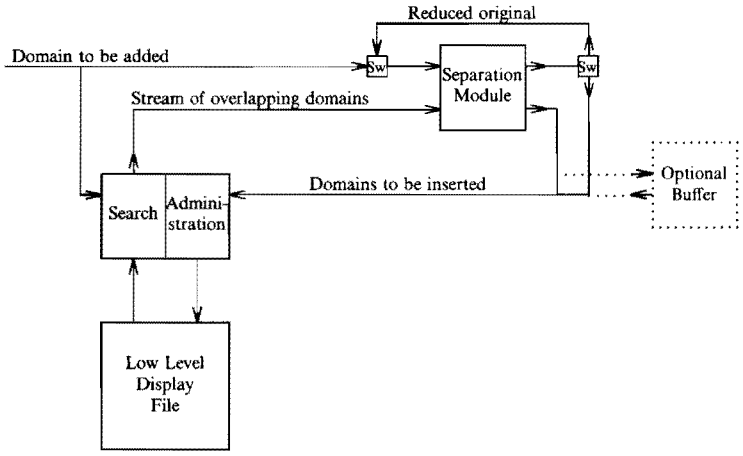


Figure 8: Functional model of adding a domain. The switches (Sw) create a loop as long as the Search Module reports overlapping domains.

The design of the domain representation, as discussed before, is aimed at an effective support of parallelism in computation in general. This effective support is in particular true for computation of the overlap sections as performed by the separation function. Since each scanline in the representation contains all the information of the slice above it, the overlap sections for all scanlines can be computed individually for each slice of the domain. Thus a parallel computation of overlap, followed by a merging process is possible. This is parallelism on a low level and will speed up the execution time of the separation process. We will not discuss this level of parallelism in detail but concentrate on parallelism on a higher level.

The HSRA prescribes that each domain from the LDF having overlap should cause a reduction of the domain to be added. This reduction so far was described as being done by sequential separation operations, thereby causing an execution time $O(k)$ in measure of the average time (τ_{sep}) a separation operation needs, k being the number of overlapping domains. These separation operations however, can also be performed in parallel if a module is added that sums all the overlap sections and removes this sum from the original. Such a reduction adder can realise a time bound $O(\log k)$. Note that this is in terms of the average time one summation operation takes (τ_{add}). Since a summation operation is less complex than a separation operation, this average summation time is smaller than the average

separation time. As a result by parallelising the separation module the total execution time of the separations can be reduced from $\tau_{sep} \cdot k$ to $\tau_{sep} + \tau_{add} \cdot \log k$.

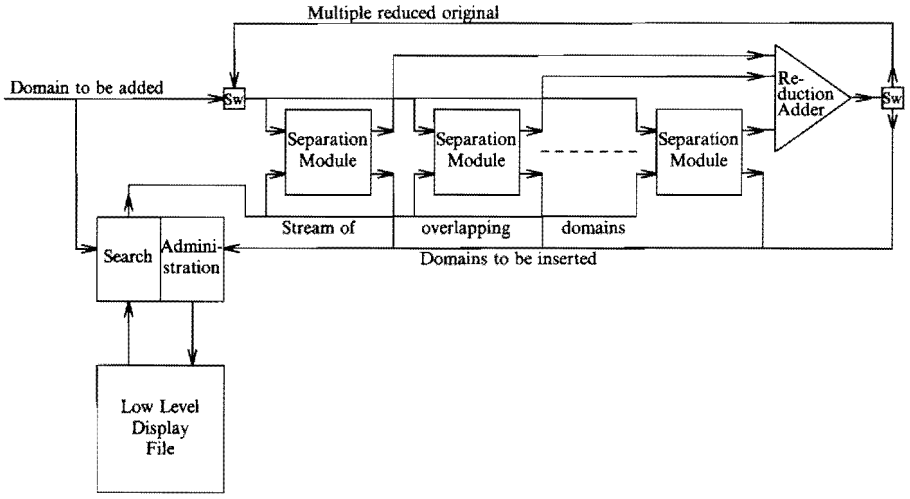


Figure 9: Parallelising the Separation Module. This requires an additional module (Reduction Adder) that sums the necessary reductions of the original, as found by all individual Separation Modules.

It is not economical to have more separation modules than number of overlapping domains. On the other hand, the number of overlapping domains reported by the search module is variable. When it is decided to have a fixed number of modules, it has to be possible to use the multiple reduced original domain again for further separation processes. This to be able to handle "overflow" caused by exceptional overlap situations. This overflow route is no different from the loop we had before in the functional model of Figure 8. The resulting parallelised version of the functional model is shown in Figure 9. If the number of separation modules is n, the total execution time of the separation processes obtained will be $\left\lceil \frac{k}{n} \right\rceil \cdot (\tau_{sep} + \tau_{add} \cdot \log n)$.

In this parallelised version we see that one search and administration module has to serve several separation modules. This can be improved by also parallelising the search algorithm. This is very well supported by the data structure of the LDF. For this no additional modules are needed, simply assigning one or more buckets of this structure to individual search and administration modules will suffice (see Figure 10). If more buckets are assigned to one search and administration module, for reasons of load balancing, neighbouring buckets should be assigned to alternating modules.

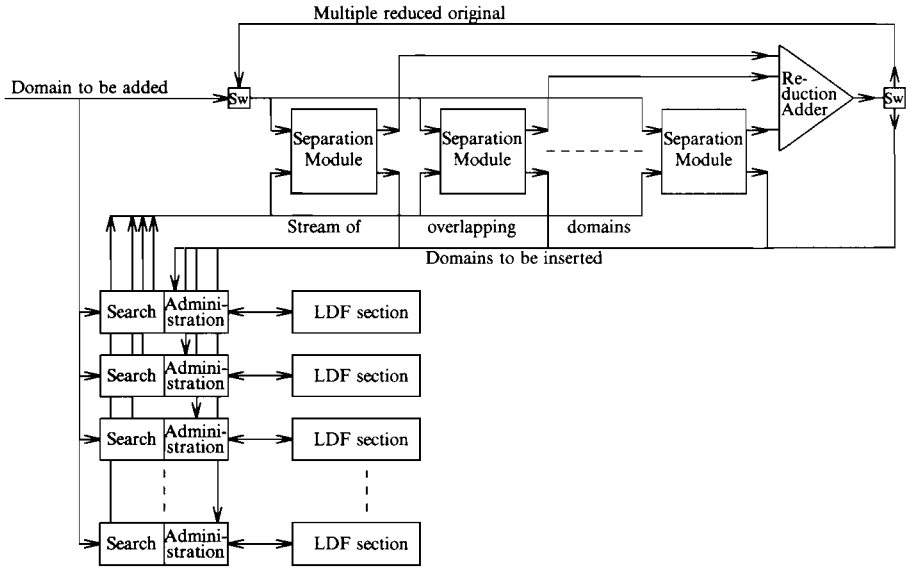


Figure 10: Parallelising both the Separation Module and the Search and Administration Module. The data structure of the LDF can easily be subdivided, each section being served by its individual Search and Administration Module.

Currently an implementation of the HSRA is in progress on a sequential machine. In the near future, we will implement a coarse grain parallel version of the HSRA (all modules on separate general purpose hardware). Later on, a fine grain parallel implementation of some of the modules (in particular the separation module) will be made on general purpose fine grain parallel hardware and perhaps on custom hardware.

References

1. V. Akman, P.J.W. ten Hagen, and A.A.M. Kuijk, "A Vector-like Architecture for Raster Graphics," *these proceedings* (1988).
2. J.L. Bentley and D. Wood, "An Optimal Worst Case Algorithm for Reporting Intersections of Rectangles," *IEEE Transactions on Computers* **C29**(7) (1980).
3. J. Foley and A. van Dam, *Fundamentals of Interactive Computer Graphics,* Addison-Wesley, Reading, Mass. (1982).
4. W.R. Franklin and V. Akman, "Adaptive Grid for Polyhedral Visibility in Object Space: An Implementation," RUU-CS-86-4, University of Utrecht, The Netherlands (1986).
5. P.J.W. ten Hagen and T. Trienekens, "Pattern Representation," Report CS-R8602, Center for Mathematics and Computer Science, Amsterdam (Jan. 1986).
6. P.J.W. ten Hagen, M.M. de Ruiter, and C.G. Trienekens, "Raster Graphics Facilities (RGF)," preliminary report, Center for Mathematics and Computer Science, Amsterdam (1986).
7. P.J.W. ten Hagen, A.A.M. Kuijk, and T. Trienekens, "Display Architecture for VLSI-based Graphics Workstations," in *Advances in Computer Graphics Hardware I*, ed. W. Straßer, Springer-Verlag (1987).
8. H.G. Mairson and J. Stolfi, "Reporting and Counting Intersections Between Two Sets of Line Segments," in *Theoretical Foundations of Computer Graphics and CAD*, ed. R.A. Earnshaw, Springer-Verlag, Heidelberg (1988).
9. I.E. Sutherland, R.F. Sproull, and R.A. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms," *Computing Surveys* **6**(1) (1974).
10. D. Wood, "An Isothetic View of Computational Geometry," in *Computational Geometry*, ed. G.T. Toussaint, Elsevier Science Publishers B.V.(North-Holland) (1985).