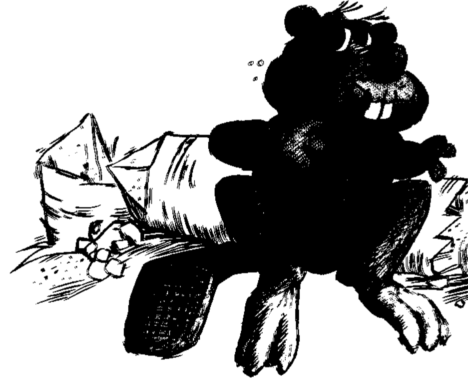
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT

UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT



Fast Hit Detection for Disjoint Rectangles

Dan Field

CS-85-53

December, 1985

Fast Hit Detection for Disjoint Rectangles

Dan Field

Computer Graphics Laboratory
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

ABSTRACT

The problem of determining which (if any) among a static set of n disjoint rectangles contains a query point is shown to take less than $8+\epsilon$ comparisons in the average case for any $\epsilon>0$. The technique is based upon a hashing scheme that requires $O(n+n\alpha^2/\epsilon^2)$ space where α is a function of the aspect ratios of the rectangles. Initial preprocessing to initialize the hash table takes $O(n\log n+n\alpha^2/\epsilon^2)$ time. The data structures and algorithms are practical to implement. The technique is useful for applications in interactive computer graphics.

1. Introduction

A common problem in computer graphics is the determination of which among a number of menu items is currently selected by the user. The menu items are typically enclosed in rectangles and distributed about the display; the user selects a menu item by placing a tracker inside the appropriate rectangle and clicking a button. Disjoint rectangles remove ambiguities in the selection of items. Often the tracker consists of a pictorial "icon" whose shape depends upon which menu item would be selected upon a button click. The software that drives such an menu system must repeatedly sample the position of the tracker and check for containment among the set of menu rectangles. It is important that the detection of both *hits* and *misses* among the set of rectangles be performed as quickly as possible as it is reasonable for the tracker to be positioned outside all menu items at least as often as inside any. An attractive feature of the technique described here is its ability to quickly determine misses.

The straightforward solution to this problem is to perform a linear search among the rectangles. This technique uses $4n$ comparisons in the worst case. An average of $2n$ comparisons for hits are used assuming that any of the n rectangles are equally likely to be selected. $4n$ comparisons are used for queries that miss all of the rectangles. Linear space requirement and implementation ease are principal attractions of linear search.

Binary search techniques as described by Edelsbrunner and Maurer [1] reduce the worst case performance of an algorithm for this problem to $O(\log n)$ comparisons with $O(n \log^2 n)$ space. It is possible to reduce the space requirements to $O(n)$ at the expense of increasing the time requirement to $O(\log^2 n)$. The detection of a miss always requires examination of an entire root to leave path in a search tree. The average number of comparisons to answer a query is $\Omega(\log n)$ if any of the rectangles are equally likely to be selected.

A quadtree is useful for this problem, but suffers from a space requirement that could be on the order of nM, the resolution of the query region [2]. It is possible for some of the queries to be determined quickly, but in the worst case they will take $3\log M$ comparisons to traverse the tree from root to leaf. Typical values for M are 512 and 1024. Average case analysis is particularly difficult for quadtrees; their structure is not stable under small perturbations in the set of rectangles.

The corner-stitched data structure of Ousterhout [3] may be used to represent the set of rectangles using $O(n)$ space and achieving a worst case query time of $O(n)$. Ousterhout claims that for "nice" sets of rectangles, the query time drops to $O(\sqrt{n})$. This data structure has the attractive feature that many queries can take time related to the manhattan distance between the current and previous query. Thus, for queries that are very close to each other, as may be the case for interactive graphics applications, high performance may be possible. No provable average case results are known for this data structure.

Assuming that query points are uniformly distributed over a rectangular query region bounding the rectangles, this paper demonstrates how hashing can be used to find the rectangle (if any) containing a query in less than $8+\epsilon$ comparisons in the average case for any $\epsilon>0$. The technique requires $O(n+n\alpha^2/\epsilon^2)$ space. The value of α depends upon the aspect ratios of the rectangles. The pre-processing time taken to initialize the data structure is shown to be $O(n\log n+\max\{n, n\alpha^2/\epsilon^2\})$. The main advantage of this scheme over others is its fast expected behavior and quasi-linear space when $\epsilon=1$. A secondary advantage over the search-tree techniques of [1] is its simplicity of implementation. The data structure and associated query algorithm are robust in the sense that non-

disjoint rectangles and on-the-fly rectangle insertions and deletions can be handled correctly. However, the stated time and query bounds may be exceeded in both of these cases.

Sketch of Technique

Assume that all the rectangles and query points are contained within a rectangular query region. A rectangular grid consisting of cells having size $H \times W$ is placed over the region. It is convenient to increase the size of the query region to $Z_h \times Z_w$ so that every cell in the grid is contained entirely within the region†.

Cells are numbered from 0 in a left-to-right, bottom-to-top fashion. See Fig. 1. Cells intersecting any rectangles are placed into a hash table of size m with collisions resolved via chaining. Chains are ordered by increasing cell number and are terminated by a cell whose number is greater than any cell in the grid. Each cell within the hash table contains a list of intersecting rectangles. See Fig. 2.

The first step in processing a query (x,y) is to determine the cell number p that contains (x,y). Assuming the lower-left corner of the query space has coordinates (0,0), the formula for p is

$$p = \left\lfloor \frac{x}{W} \right\rfloor + \left\lfloor \frac{y}{H} \right\rfloor \left\lceil \frac{Z_w}{W} \right\rceil, \tag{1}$$

where $\left\lceil \frac{Z_w}{W} \right\rceil$ is the number of cells spanning the query region in the horizontal direction. The hash table is searched for cell p by examining the chain emanating from bucket $p \bmod m$. If cell p is not on the chain, the query point cannot be contained within any of the rectangles. If cell p is on the chain, the rectangles intersecting the cell are searched in the straightforward manner for containment of the query.

The success of this technique depends upon the size of the hash table m and the cell dimensions H and W. In Section 2 it is demonstrated that the average number of comparisons to search a chain for a cell is less than 3. Section 3 demonstrates how to choose H and W so that the expected number of comparisons required to search a cell is less than $5+\epsilon$ and that the number of cells in the hash table is $O(n+n\alpha^2/\epsilon^2)$. Section 4 describes the process of initializing the data

†Increasing the query region decreases the probability of containment within a rectangle when the query points are distributed uniformly throughout the region. This problem is remedied in the final section.

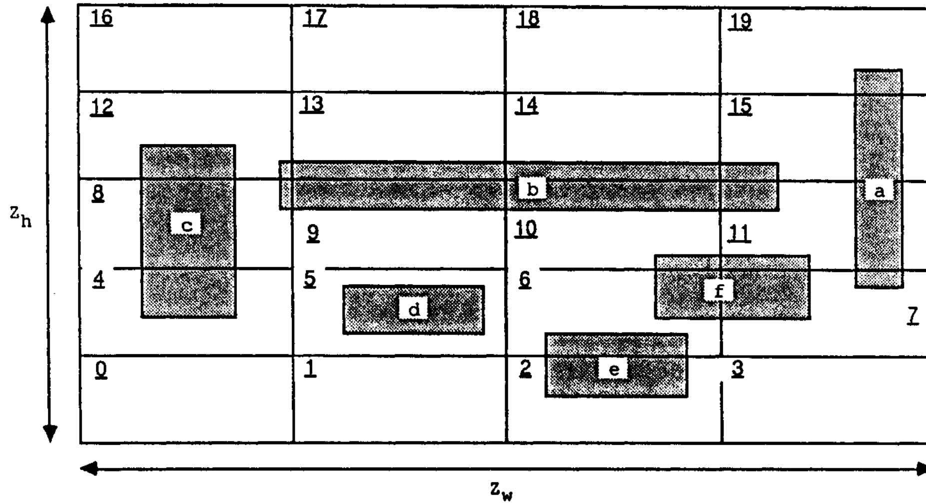


Figure 1. A regular grid of cells is placed over the query region containing rectangles a,b,c,d,e,f. Cells have height H and width W and are numbered from left-to-right, top-to-bottom starting at 0.

structure.

2. Searching for a Cell

The process of searching for cell p in the hash table consists of computing the bucket of the hash table $p \bmod m$ where m is the size of the table. The chain emanating from this bucket is searched linearly for p. As the chain is terminated by a cell numbered larger than any in the query region and is ordered by increasing cell number, the search may terminate once a cell on the chain whose number is larger than p is encountered. To reduce the number of comparisons when moving from one cell to the next during the search, p is compared for equality with cells on the chain only after finding the first cell numbered larger than p. The previous cell on the chain is then checked for equality. With this technique, a search for the kth cell on the chain uses $k+2$ comparisons.

Let the size of the hash table, m, be the number of non-empty cells. It will be shown that the expected time to search for a cell is maximized when every chain contains a single non-empty cell (that is, chains have a length of 2). Therefore, the number of comparisons for a successful search is 3.

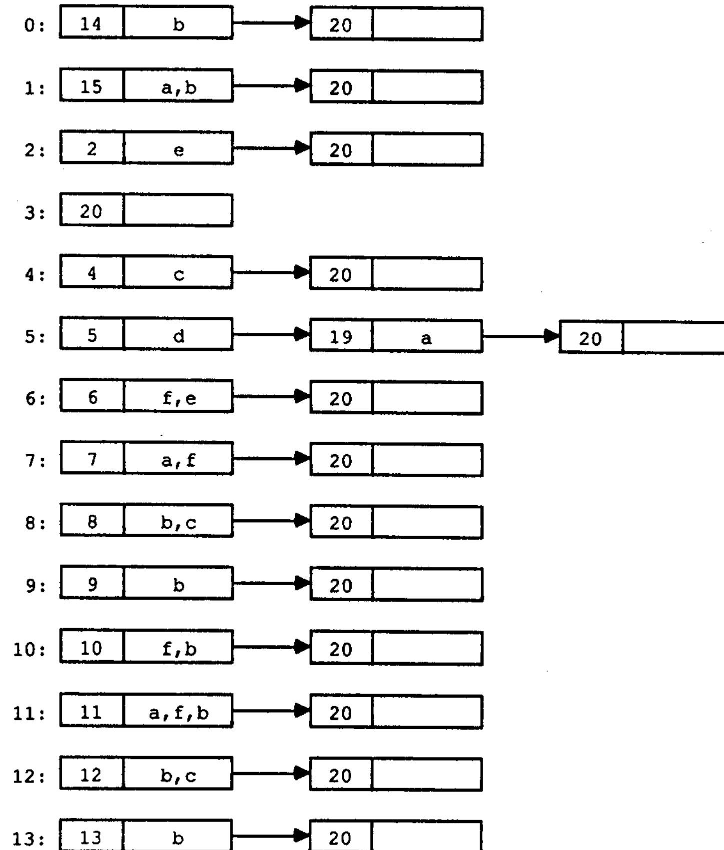


Figure 2. The hash table representing the rectangles and grid structure depicted in Fig. 1. The number of entries in the table, $m=14$, corresponds to the number of non-empty cells. Each cell contains the cell number along with a list of intersecting rectangles. Chains are ordered by cell number and are terminated by a cell numbered larger than any within the query region.

An unsuccessful search uses no more than 1 plus the length of the chain comparisons. When each chain has length 2, the maximum number of comparisons is 3. Therefore, the expected number of comparisons of either search type is less than 3.

Lemma: The expected number of comparisons to search for a cell is maximized

when the chains have length 2.

Proof: Let k_i be the number of non-empty cells that hash to bucket i and l_i be the number of empty cells that hash to bucket i.

A condition for the expected number of comparisons to be maximized at bucket i is that all searches for empty cells examine the entire chain of non-empty cells at bucket i. Summed over all l_i empty cells, this results in a total of $(k_i+2)l_i$ comparisons. The number of comparisons to find the jth cell on a chain is $j+2$. Summed over all the successful searches in the chain, $\sum_{j=1}^{k_i}(j+2)$ comparisons will be made.

One consequence of using the division method as a hash function is that k_i+l_i is a constant for any i, regardless of k_i. If k_i were increased by 1, then the total number of comparisons over all searches of the chain would increase by l_i. Suppose that 2 buckets i and j have $k_i+l_i = k_j+l_j$, and that a non-empty cell in bucket j becomes empty and an empty cell in bucket i becomes non-empty. The total number of comparisons over both changes by $l_i-(l_j-1)$. Thus, the number of comparisons over all searches in both buckets may be increased as long as chain i is shorter than j.

Another consequence of the hash function is that k_0+l_0 cells hash to buckets 0 through $r-1$, and k_0+l_0-1 cells hash to buckets r through $m-1$. An argument similar to that of the previous paragraph can be used to show that when the number of cells hashing to one bucket differs from the number hashing to another bucket by 1, then the total search cost over both is maximized when the number of non-empty cells in the chains differ by at most 1. Since there are as many non-empty cells as buckets, they must be distributed evenly across the buckets. Therefore, all chains are of length 2.

Chains of length 2 need no more than 3 comparisons to determine if a cell is contained. Therefore, the following has been proven.

Theorem: The expected number of comparisons to search for any cell in the query region is less than 3.

3. Time and Space Results

3.1. Searching a Cell

The time to search a cell is proportional to the number of intersecting rectangles; as the cell size increases, so does the number of intersecting rectangles and hence the query time. As the cell size decreases, the query time decreases but the number of non-empty cells, and hence space, increases. An ideal balance occurs when each non-empty cell intersects a single rectangle and a rectangle intersects a single cell. This should occur when the rectangles and cells have the same size. It is shown below that the desired time and space bounds are achieved by making the cell dimensions proportional to the average dimensions of the rectangles. Rectangles whose dimensions deviate from the average affect the time and space performance in an adverse manner. The degradation is measured in terms of the variation from the mean of the rectangle aspect ratios. The variations can be made to affect only the space requirements while preserving the time bounds.

Let rectangle i have height h_i and width w_i. The cell dimensions are set to

$$H = \frac{\sum^{n} h_i}{cn} \tag{2}$$

and

$$W = \frac{\sum^{n} w_i}{cn}. \tag{3}$$

The value of c is determined by ϵ. Define

$$g = \frac{4\sum^{n} h_i \sum^{n} w_i}{n\sum^{n} h_i w_i}. \tag{4}$$

Then it will be shown that

$$c = \frac{g+\sqrt{25g^2+4g\epsilon}}{2\epsilon} \tag{5}$$

achieves the desired query performance.

Let m be the number of non-empty cells intersected by the set of rectangles. Then

$$m \leq \sum^{n} \left(\frac{h_i}{H} + 2 \right) \sum^{n} \left(\frac{w_i}{W} + 2 \right) \tag{6}$$

because no rectangle can overlap more than 2 more its height or width in cells and, in the worst case, no more than one rectangle can intersect a single cell. Furthermore,

$$m \geq \frac{1}{HW} \sum^{n} h_i w_i \tag{7}$$

because rectangles are mutually disjoint. The expected number of rectangles that intersect a non-empty cell, B, is the number of unique rectangle-cell intersections divided by the number of non-empty cells. B is bounded from above by the upper bound of m in Eq. (6) divided by the lower bound of m in Eq. (7). Thus,

$$B \leq 1 + \frac{2W \sum^{n} h_i + 2H \sum^{n} w_i + 4nHW}{\sum^{n} h_i w_i}. \tag{8}$$

Substituting for H and W,

$$B \leq 1 + \frac{(c+1)}{c^2} g. \tag{9}$$

Setting the quantity on the right side of Eq. (9) to $1+\epsilon/5$ and solving for c yields Eq. (5).

Theorem: The size of a cell can be tuned so that the expected number of comparisons to search a non-empty cell for a rectangle containing the query point is less than $5+\epsilon$.

Proof: From the above argument, the expected number of rectangles in a non-empty cell can be made to be $1+\epsilon/5$. Four comparisons are used to test each rectangle in the cell for containment of the query point; a single comparison is used to check for termination of the list containing rectangles in the cell.

3.2. Space Requirements

The data structure consists of 3 components: the hash table consisting of chain pointers, chain nodes consisting of non-empty cells, and lists of rectangles contained in chain nodes. The size of the first two components is $O(m)$. The quantity of Eq. (6) is an upper bound on the number of rectangles contained in all of the cells and is larger than m because non-empty cells contain at least one rectangle.

Let C be the amount of storage required by the data structure. Substituting values for H, W, and c in Eq. (6), one obtains

$$C \leq 4nd\left[\frac{5g}{2\epsilon}+\left(\frac{25g^2}{4\epsilon^2}+\frac{5g}{\epsilon}\right)^{1/2}+1\right]\left[\frac{5g}{\epsilon}+1\right] \tag{10}$$

where $d>0$ is a constant taking into account overhead for pointers and the like. If $\epsilon = O(g)$ then

$$C = O(n). \tag{11}$$

Otherwise,

$$C = O\left(\frac{ng^2}{\epsilon^2}\right). \tag{12}$$

It can be shown that $4\max_i\{h_i,w_i\}$ is an upper bound on g. A less pessimistic bound that provides some intuition on the behavior of g is given below. Let

$$\rho_i = \min\{h_i,w_i\}/\max\{h_i,w_i\} \tag{13}$$

be the aspect ratio of rectangle i and

$$d_i = \max\{h_i,w_i\}. \tag{14}$$

Then $\rho_i d_i^2 = h_i w_i$. Let $\rho = \min_i\{\rho_i\}$. The quantity ρ represents the aspect ratio of the rectangle that deviates most greatly from a square. Note that $\rho \leq 1$. Then

$$g \leq \frac{4\sum^n h_i \sum^n w_i}{\rho n^2 \mathrm{E}(d^2)}, \tag{15}$$

where $\mathrm{E}(d)$ is the expected value of the d_i. Finally, let

$$\beta = \min\{H,W\}/\max\{H,W\}. \tag{16}$$

In words, β is the aspect ratio of the cells which, in some sense, represents the average of the n rectangles. Note that $\beta \leq 1$. Then,

$$\begin{aligned} \sum^n h_i \sum^n w_i &= \beta\max\{H^2,W^2\} \\ &\leq \beta n^2 \mathrm{E}(d)^2. \end{aligned} \tag{17}$$

Since $\mathrm{E}(d)^2 \leq \mathrm{E}(d^2)$,

$$g \leq \frac{4\beta}{\rho} = \alpha \tag{18}$$

In words, α is the aspect ratio of the "average" rectangle divided by the aspect ratio of the rectangle that deviates the most from a square. The value of α is bounded from above by $1/\rho$; for rectangles of bounded aspect ratio α could be considered a constant. β moderates the value of $1/\rho$, in the sense that as the set of rectangles approaches a uniform aspect ratio, α approaches 1.

The following has been proven:

Theorem: The storage required for the data structure is $O(n+n\alpha^2/\epsilon^2)$.

4. Pre-processing Time

A naive method of initializing the data structure is, for each rectangle, update or insert the intersecting cell in the hash table to include the new rectangle. In the worst case, every cell found to intersect the rectangle might not yet be in the hash table and have to be placed at the end of a chain consisting of all cells intersected so far. This results in an algorithm quadratic in the size of the data structure. A more clever algorithm visits each non-empty cell once and in an order that allows it to be inserted at the head of a chain. This results in an algorithm that is linear in the size of the data structure once the appropriate cell ordering has been found.

As the chains are ordered by increasing cell number, an efficient sequence in which to consider the non-empty cells is by decreasing cell number. Inserting a new cell in the hash table is a constant time operation since it would always be placed at the beginning of a chain.

So that every cell is considered only once, it is important that every rectangle be known that intersects a cell under consideration. To accomplish this, the rectangles are first sorted in decreasing cell number order by the cell containing the upper-right corner of the rectangle and placed in a list L. Moving from the top of the query region to the bottom, rows of cells are examined for intersecting rectangles in a sweep-line fashion. Cells within a particular row are examined from right to left.

In addition to L, another list of rectangles R is maintained that intersects the current row of cells. Rectangles on R are in decreasing order of the x coordinate of their right edges. See Fig. 3. It is a straightforward matter to scan R and produce all the rectangles that intersect the cells in the row in decreasing cell number order. The details are omitted. Once all the rectangles in a cell have

been determined, the cell is placed on the start of the chain emanating from the bucket whose address is the hashed cell number. Over a single row, this process takes time linear in the number of rectangle-cell intersections in the row. Over all the rows, this process takes time linear in the number of cell-rectangle intersections in the entire query space.

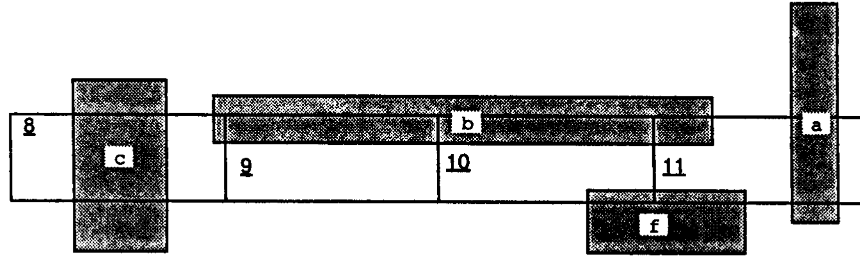


Figure 3. The list R contains the names of the rectangles intersecting a row of cells and is ordered from right-to-left by the right edges of the intersecting rectangles. For the row depicted above, $R{=}a,f,b,c$. The list L contains the names of all the rectangles in the query region and is ordered from right-to-left and top-to-bottom by the rectangle upper-right corners. $L{=}a,b,c,f,e,d$ for the situation depicted in Fig. 1.

Once the cells in the current row have been placed in the hash table, a new list R' is created for the row below. Some rectangles from the current row are no longer present and additional rectangles need to be inserted. R' is constructed from R, in place, using a merge-like pass over R and L. See Fig. 4. Each rectangle is considered for inclusion in or deletion from a row list R over all intersecting cells. Over all the merge steps, this takes time linear in the number of cell-rectangle intersections in the entire query space.

Theorem: The hash table can be initialized in $O(n\log n + n\alpha^2/\epsilon^2)$ time.
Proof: The hash table is first initialized so that every chain contains a single cell whose number is greater than any cell in the query region. This takes time linear in the space of the data structure. The rectangles are sorted in $O(n\log n)$ time. The number of cell-rectangle intersections can be determined by considering every non-empty cell to intersect one rectangle only. An upper bound on this number is the quantity on the right side of Eq. (6). It was shown in Section 3

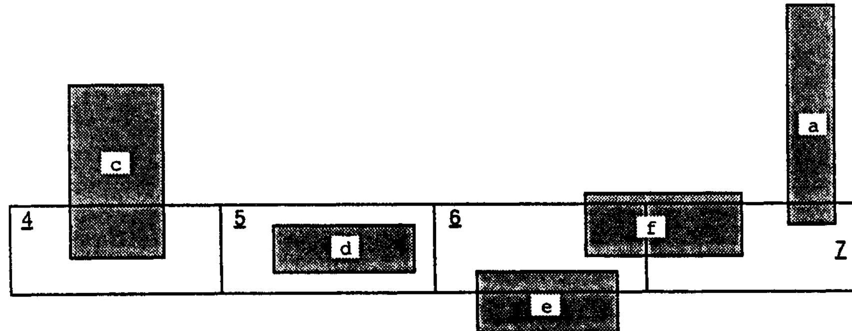


Figure 4. The list R' of rectangles intersecting the current row of cells depicted above is obtained from a merge-like operation on lists L and R from the previous row depicted in Fig. 3. $R'=a,f,e,d,c$.

that this quantity is $O(n+n\alpha^2/\epsilon^2)$.

5. Discussion

Recall from Section 2 that the original query region has been expanded to $Z_h \times Z_w$ so that all cells fit within the region. By choosing a suitable value for c in Eqs. (2) and (3), the grid can be made to fit exactly within the original query region. Note that c need not be more than twice its original value, so that the number of non-empty cells, and hence space requirements, are not increased by more than a factor of 4.

The analyses of time and space requirements is rather pessimistic. On one hand, the query time for a cell given in Eq. (9) assumes that exactly one rectangle intersects any non-empty cell; on the other hand it also assumes that rectangles are packed in as tightly as possible with no gaps between. Alone, either is possible. Together, unless all rectangles have the same dimensions as a cell, they are not. The estimate of the time is used to determine c which ensures the desired query time ϵ. If the estimate were more accurate, c could be decreased resulting in a more realistic bound on the space requirements.

There is no need for both W and H to have the same form. A reasonable approach is to solve the time estimate for H in terms of W and ϵ, then minimize the equation representing the space estimate with respect to W. Replacing W in the time estimate and back-solving yields H. Attempts to do so have resulted in

complex formulas that appear to lack intuitive meaning.

The technique described performs well for a special case of the general hit detection problem. The data structure and algorithms can be extended to handle multi-dimensional rectangles, as well as the non-disjoint and dynamic cases. However, the time and space results of the previous sections do not follow and it appears that analysis for these extensions is difficult.

5.1. Dimensions 3 and Above

For k dimensions, determination of c becomes a matter of solving a kth order polynomial in c. The resulting equation expressing the space requirement appears to become too complex to obtain a meaningful upper bound. A reasonable conjecture is that the space bound might be on the order of $n\alpha'/\epsilon^k$ where α' is some function of the aspect ratios of the rectangular polyhedra.

5.2. Non-disjoint Rectangles

The data structure and associated algorithms correctly handle non-disjoint rectangles. However, the analysis of Section 3 relies heavily upon the fact that they are disjoint. In the worst case, all the rectangles could be identical and intersect only a portion of exactly 4 cells. It is reasonable to expect that extra time must be taken to report all the intersections of a query point. However, queries in these 4 cells take $O(n)$ time independent of the query point. A possible approach to this problem would be to incorporate the percentage of area that rectangles overlap into the cell size determination.

5.3. Dynamic Sets of Rectangles

The dimensions of the rectangles are necessary to determine the appropriate cell size. It is straightforward to insert and delete rectangles from the data structure. Such operations may cause either or both of the time and space bounds to be exceeded. With a fixed cell size, the bound on the average query time will not be exceeded if the average rectangle dimension increases. However, the space bound could be exceeded. Conversely, the bound on space will not be exceeded if the average rectangle dimension decreases. However, the average query time bound may be exceeded. One would not expect either figure to be affected radically with small changes in the average rectangle, especially considering the pessimistic nature of the preceding analysis. If the number of rectangles in a cell exceeds a predefined maximum due to insertions, it is possible to split the cells in half in the x or y direction and perform a rehash as described in [4]. The average performance of the query algorithm could then be guaranteed.

References

[1] EDELSBRUNNER, H. AND MAURER, H.A. On the intersection of orthogonal objects. *Inf. Proc. Let. 13*, 4,5 (End 1981), 177-181.

[2] DYER, C.R. The space efficiency of quadtrees. *Comp. Gr. and Im. Proc. 19*, 4 (Aug. 1982), 335-348.

[3] OUSTERHOUT, J.K. Corner stitching: A data-structuring technique for VLSI layout tools. *IEEE Trans. on CAD CAD-3*, 1 (Jan. 1984), 87-103.

[4] AHO, A.V., HOPCROFT, J.E., AND ULLMAN, J.D. *Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, Mass., 1975.