# Algorithm for the Removal of Rectangle Containment for Rectangle Spline Generation

Les A. Piegl[1], Parikshit Kulkarni[2] and Khairan Rajab[3]

[1]University of South Florida, lpiegl@gmail.com
[2]Synopsys Inc., Parikshit.Kulkarni@synopsys.com
[3]Najran University, khairanr@gmail.com

## ABSTRACT

Given a set of axis-aligned rectangles in the plane, this paper presents an algorithm for the removal of rectangle containment as well as rectangle enclosures. That is, given a query rectangle $R_q$, this algorithm removes all rectangles $R_k, \ldots, R_l$ from the rectangle set that are contained in $R_q$. It also removes $R_q$ if there is a rectangle $R_j$ that encloses $R_q$. The algorithm, initially implemented for rectangle spline generation using rectangles as building blocks, is fast, easy to implement and maintain, it can be run on parallel machines such as GPUs at no extra cost, but it requires extra memory to store the rectangle indexes for quick access and space partitioning.

**Keywords:** rectangle containment, rectangle splines, spatial data structures, GPUs.

## 1. RECTANGLE SPLINES INTRODUCED

NURBS curves have played a significant role in design and manufacturing since the advent of CAD tools in engineering. As the field matured, they found their ways into more and more diverse applications such as graphics, entertainment, medical modeling and the representation of arbitrary shapes bounded by closed curves. A NURBS curve, defined as [7]

$$C(u) = \sum_{i=0}^{n} P_i N_{i,p}(u)$$

where $P_i$ are the control points and $N_{i,p}(u)$ are the normalize B-splines, is well understood and has a number of desirable properties that account for their popularity. Since this is a curve, it sweeps out a path that has no thickness. In many applications the goal is to create a path with some (variable) thickness, i.e. to create an area. This is easily done with splines:

$$A(u) = \sum_{i=0}^{n} S_i N_{i,p}(u)$$

where $S_i$ represent any shape, e.g. circular discs or arbitrary domains. In this research we are interested in rectangle splines defined by axis aligned rectangles:

$$R(u) = \sum_{i=0}^{n} R_i(C_i, w_i, h_i) N_{i,p}(u)$$

where the control rectangles $R_i$ are defined by $(C_i, w_i, h_i)$, the centers, the widths and heights. Figure 1 shows such a rectangle spline.

The rectangles connected by a dashed polygon form the control rectangles and the spline is swept out by a set of rectangles that change position and width and height as the parameter moves from the start to the end. There are many ways one can generate such a rectangle spline; one is by putting rectangles on the plane and using them as control rectangles, and another is to generate a set of rectangles inside the shape and fitting a rectangle spline to them. While this is a very viable way of generating areas, it turns out that in case of a dense set of rectangles one may end up with a sizeable set of rectangles that are contained in one another. The removal of such containment is highly desirable in applications like graphics, however, it is a must for spline fitting in order to avoid a badly conditioned system of equations.

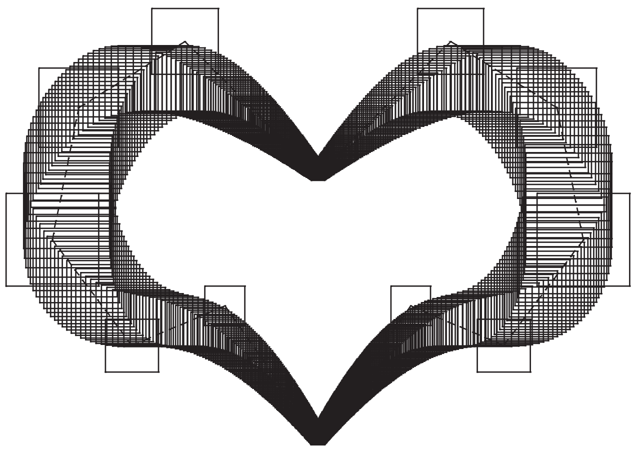

Fig. 1: Rectangle spline with control rectangles and area coverage.

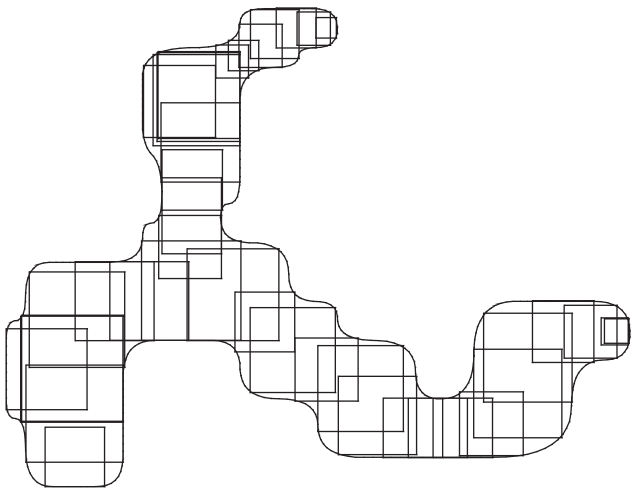

Fig. 2: Typical geometrical shape represented by a boundary and a set of rectangles.

## 2. RECTANGLE CONTAINMENT

Given a set of axis-aligned rectangles $R_0, \ldots, R_n$ in the plane, for each query rectangle $R_q$ we are looking for a set of rectangles $R_k, \ldots, R_l$ so that $R_i \subset R_q, i = k, \ldots, l$. That is, we are looking for all rectangles that are *contained* inside $R_q$. In order to serve a diverse set of applications such as graphical queries, we are also looking for a set of rectangles so that $R_q \subset R_i, i = k, \ldots, l$. That is, all rectangles are sought so that they *enclose* the query rectangle $R_q$. The algorithm presented herein will remove both contained and enclosed rectangles simultaneously.

This problem arises in applications where rectangles are used to describe shapes. Fig. 2 shows such a shape with a small set of rectangles covering a percentage of the free-form area. One way to generate such a set of rectangles is via region growing: select a guiding curve such as the medial axis, then for each discretized point on the medial axis grow a rectangle until it hits the boundary, i.e. until it cannot be grown any further without leaving the boundary or creating a skewed rectangle. Fig. 3 shows the process for one branch of the media axis. On the left a large set of 40 rectangles are produced via region growing creating lots of containments. On the right all containment and enclosures have been removed to generate 15 non-containing rectangles; more than half of the rectangles were unnecessary!

Important applications arise in graphical search using bounding rectangles and a rectangular search window, i.e. what is visible in a particular window, and in clipping applications where a complicated scene needs to be clipped using a rectangular clipping windows and bounding boxes.

Given two rectangles $R_1(x_l^1, x_r^1, y_b^1, y_t^1)$, $R_2(x_l^2, x_r^2, y_b^2, y_t^2)$ and an offset distance $\delta$, rectangle containment is defined as follows:

$$R_1 \subset R_2 : x_l^2 < x_l^1 + \delta \wedge x_r^2 > x_r^1 - \delta \wedge y_b^2 < y_b^1 + \delta \wedge y_t^2 > y_t^1 - \delta$$

The case $R_2 \subset R_1$ is defined similarly. Now given any rectangle pair, there are three outcomes: $R_1 \subset R_2$, or

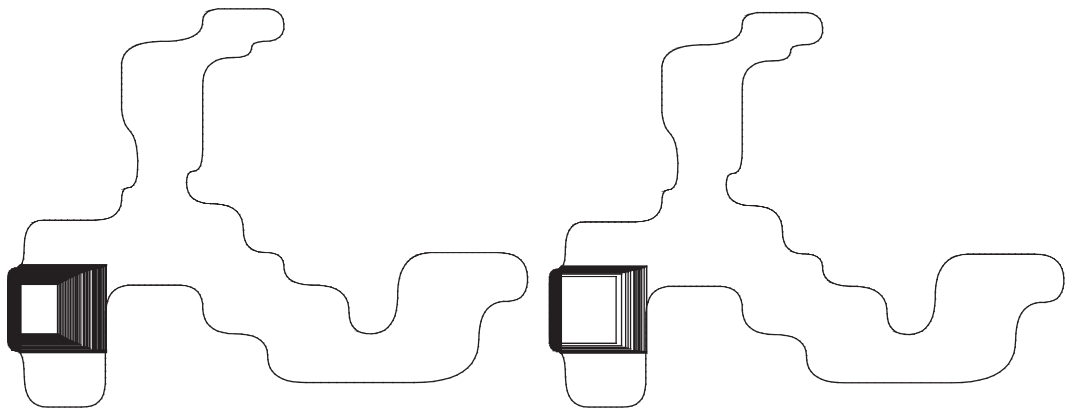
Fig. 3: Rectangles obtained via region growing: original rectangles (left), containment removed (right).

$R_2 \subset R_1$, or $R_1 \subset\supset R_2$, i.e. they mutually contain one another. The mutual containment comes into play when the rectangles are identical or nearly identical. One or the other or both of them can be removed, depending on the needs of the application.

## 3. PRIOR WORK

The literature has a number of fine algorithms for the reporting of rectangle containment and/or enclosures. One of the earliest works is due to Vaishnavi and Wood [8] where the enclosure and containment problem is formulated as a batch range searching problem in 4-D and the algorithm is based on a semi-dynamic range tree data structure. Lee and Preparata [6] improved the space requirement of this algorithm by transferring the problem to the point dominance problem in 4-D and using a quadruply threaded list with a divide-and-conquer algorithm. The rectangle retrieval problem was investigated by Abel and Smith [1] using quadtree decomposition and a locational key for each rectangle. The data structure that is applied with their algorithm is a B-tree. Rectangle enclosures are computed by Bistiolas et al [2] using a multi-layered data structure consisting of a combination of range and priority search trees. Gupta et al [4] reported an algorithm with linear space requirement based on the 4-D dominance reporting method. Improvements of this algorithm are found in Bozanis et al [3] and Lagogiannis et al [5].

## 4. THE ALGORITHM

Given $R_0, \ldots, R_n$ and an offset distance $\delta$, compute $R_0, \ldots, R_m$, $m \le n$ so that no two rectangles in the output set contain one another, i.e. eliminate all containment as well as enclosures.

### 4.1. Data Preparation

The algorithm uses a cell data structure to hold the indexes of rectangles that overlap a given cell. The cell structure is computed as follows:

- Get the boundary box and the x- and y-extent $(x_d, y_d)$ of the rectangle set
- Compute the cell size $size = \alpha\sqrt{\frac{x_d \cdot y_d}{n}}$
- Get the cell resolution $(n_x, n_y) = (\frac{x_d}{size}+1, \frac{y_d}{size}+1)$
- Allocate memory to hold $(n_x, n_y)$ number of cells

Note that at this point we have an empty cell structure. Each cell will be given memory to hold rectangle indexes when the rectangles are pre-processed. Also, the choice of the value of $\alpha$ has a profound effect on the performance of the algorithm (see below).

### 4.2. Pre-processing

Once the cell structure is established, each rectangle is pre-processed into the cell-based registry, i.e. each cell is given a list of indexes whose rectangles overlap the cell. That is

- For each rectangle do:
  - Get the indexes of the cells that cover the rectangle
  - For each cell allocate memory to store the indexes. Keep all indexes in sorted order.

A note on memory allocation. It is tempting to use a linked data structure to hold the rectangle indexes. Our experience shows that dynamic memory allocation and de-allocation can be quite expensive, slowing down the algorithm with a factor of 1.5–2.5. We had good success with a combination of dynamic and static memory allocation. It works as follows:

- Estimate the memory that is needed for each cell.
- Allocate a static array to hold the indexes.
- If the array becomes too small, reallocate the memory by adding another chunk to it.

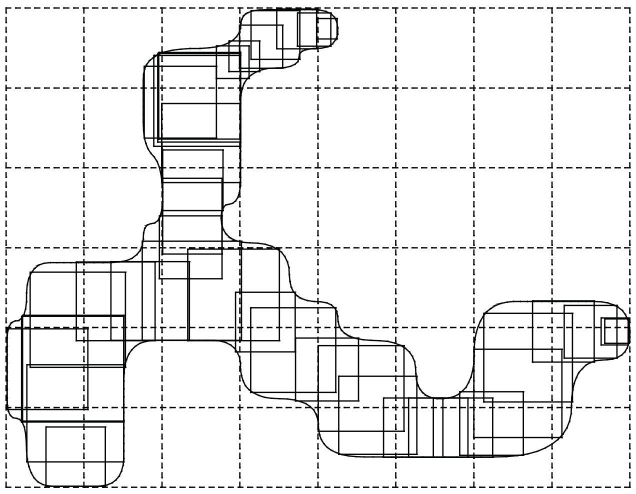

Fig. 4: Cell structure used to register the rectangles.

With some tests and intelligent estimates memory reallocation is done very rarely (at the expense of wasting some of the statically allocated memory which may have been an issue in the early days of computing but is becoming less of a concern).

Fig. 4 shows the cell structure for the rectangles depicted in Fig. 2. There are 144 rectangles to be registered with the cells and after the pre-processing is complete, the number of rectangles per cell is given in the matrix below.

$$\begin{bmatrix} 0 & 4 & 7 & 10 & 1 & 0 & 0 & 0 \\ 0 & 5 & 6 & 4 & 0 & 0 & 0 & 0 \\ 0 & 3 & 5 & 2 & 0 & 0 & 0 & 0 \\ 4 & 7 & 6 & 4 & 1 & 0 & 2 & 5 \\ 6 & 8 & 4 & 5 & 5 & 6 & 5 & 6 \\ 5 & 5 & 0 & 0 & 3 & 6 & 3 & 1 \end{bmatrix}$$

It is evident that most of the non-empty cells contain the indexes of more than one rectangle, more precisely, on average 4 rectangles are registered with a given cell.

### 4.3. Containment Elimination

After the rectangles have been pre-processed, the algorithm is ready to eliminate containment. The algorithm works as follows.

```
for each rectangle $R_q$ do
  find all cells that cover $R_q$
  for each cell do
    for each index list in the cell do
      $R_i \leftarrow$ registered rectangle with the cell
      ContainmentCheck($R_q, R_i, \delta$)
      if $R_i \subset R_q$ eliminate $R_i$ from the cell list and the rectangle data structure
      if $R_q \subset R_i$ break
    end
    if $R_q \subset R_i$ break
  end
  if $R_q \subset R_i$ eliminate $R_q$ from the rectangle data structure
  eliminate $R_q$ from the cell list
end
```

Note that the algorithm eliminates both containment as well as enclosures within the same loop structure. The function **ContainmentCheck** looks for containments both ways as well as mutual containment. Once the query rectangle is found to be contained, the inner loop structure stops and the rectangle is removed both from the rectangle data structure as well as from the cell registry. On the other hand, if the query rectangle contains more than one rectangle, all of them are found and removed from the global as well as the local cell data structure. Because each rectangle is registered will cells that cover the rectangles, the query rectangle has access to all contained rectangles. Note that the data structures used in this algorithm are dynamic, i.e. contained and/or enclosed rectangles are eliminated from rectangle as well as index data structures.

## 5. THEORECTICAL ANALYSIS

In this section we give a theoretical analysis of the different stages of the algorithm as well as a growth analysis, i.e. how the algorithm performs as the number of rectangles increases.

### 5.1. Calibration

As noted above, the choice of $\alpha$, the size of a cell, has a profound effect on the performance of the method. Also, the size of the static array for each cell determines the amount of memory reallocations and hence the performance. Setting these parameters is done in two steps: the first step is calibration, i.e. for a particular application one has to run the algorithm for a small set of values to find out which provides the best performance. The second step is knowledge base building, i.e. for each type of data set and for each amount of data the appropriate parameters are stored (a simple look-up table would do).

It is our opinion that the one-size-fits-all type algorithms may not perform well in the presence of a large variety of data. Algorithms should be endowed with freely chosen parameters to accommodate many different rectangle configurations. As the knowledge base expands, the algorithm becomes smarter and is able to perform well even if the data set changes quite a bit. This is a biologically-inspired computing paradigm where algorithms are allowed to learn and to adapt; the longer they are in use, the smarter they get and the better the performance.

### 5.2. Theoretical Analysis for a Given *n*

The theoretical performance of the algorithm is broken down into the various components as follows:

- *Data preparation.* Only bounding box and simple numerical computations are done at this stage and this is achieved in $O(n)$ time.
- *Pre-processing.* Each rectangle is registered with all cells that cover this rectangle. If on average each cell list has $k$ indexes, then this part is achieved in $O(n \log k)$ time as the indexes are kept in sorted order. The size of the index arrays is a small percentage of the total number of rectangles, usually around 5-15%
- *Elimination.* The performance of this stage depends largely on how the rectangles are arranged. We identified three vastly different cases:
  - *Trivial arrangement.* In this case the rectangles are mutually contained, i.e. nearly identical, and the algorithm removes all of them (minus the query rectangle) in one step. For $n-1$ rectangles the cost is $(n-1)k \log k$, where $k$ is the average length of the index array for each cell which is kept in sorted order (it takes $O(\log k)$ time to find the index and $O(k)$ to shift the elements).
  - *Well distributed.* If for each query rectangle the density around this rectangle is the same, then for each query one has to check about the same number of rectangles for containment, i.e. containment check is done in $O(n)$ time. Once a contained rectangle is found, its index has to be removed from the cell registry which requires $mk \log k$ operations, where $m$ is the number of rectangles to be removed.
  - *Poorly distributed.* One can create an arrangement where the rectangles are nearly mutually contained, i.e. for each query rectangle the algorithm has to check all other rectangles for containment. This is clearly an $O(n^2)$ process and an additional $mk \log k$ cost is added to update the cell registry.

It is clear from the above analysis that the algorithm's performance is very dependent on how the rectangles are distributed in space. An empirical analysis is necessary to assess the value of the method to any particular application.

### 5.3. Theoretical Growth Rate as a Function of *n* and the Density

To establish a theoretical growth rate, one has to consider two issues: the growth of the number of rectangles and the change of the density for each query rectangle. The performance depends greatly on how the rectangles are placed as the number of rectangles increases. We consider two cases:

- The rectangles are placed while keeping the density constant, i.e. rectangles are added to the part of the plane where no or few rectangles are present. This is done by tiling the plane with shapes similar to the one shown in Fig. 2 so that there is no overlap between any two shapes. Since the density is constant, i.e. for each query rectangle the number of containment checks is the same, the growth rate is $O(n)$.
- The rectangles are placed into a confined space so that as more of them are squeezed in, the density increases. In this case not only the number of query rectangles increases but the number of containment checks increases as well, e.g. for twice the number of rectangles there may be two times more candidate rectangles in the vicinity of each query rectangle requiring four times more comparisons. In this case the performance is clearly exponential.

The major contributing factor to the growth rate is density; as more and more rectangles are packed into the same space, the performance suffers a great deal (as we demonstrate in the practical analysis section).

## 6. PRACTICAL ANALYSIS

In this section we present some numerical data to gain a deeper insight into the working of the algorithm. We chose to test the method on random rectangles, Fig. 5 right, confined to a certain space, i.e. both the number of rectangles as well the density have been increased. Also because they are chosen randomly, there may not be consistency as in shape filling rectangles shown in Fig. 5 on the left.

Table 1 shows data for space as well as time requirements tested on an off-the-shelf laptop with

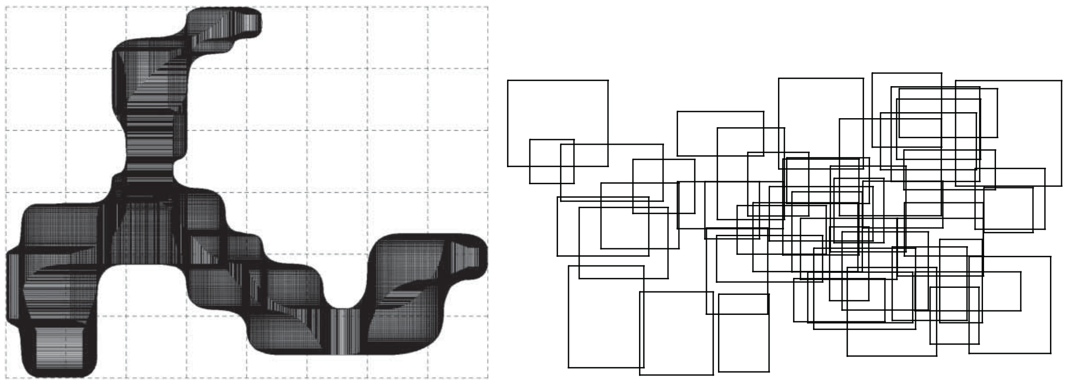
Fig. 5: Rectangle configurations: shape filling rectangles (left), random selection (right).

8 GB of RAM, and choosing $\alpha = 10$ for all tests. The notations are as follows:

- $n$ is the number of input rectangles
- $m$ is the number of output rectangles, i.e. the rectangles after containment elimination
- $n_x \times n_y$ is the cell resolution in x- and y-directions
- $k$ is the average number of indexes per cell
- $t_P$ denotes the time to take to prepare the data
- $t_E$ denotes elimination time
- $t_T$ represents total processing time

| n | m | $n_x \times n_y$ | k | $t_P$ | $t_E$ | $t_T$ |
|---|---|---|---|---|---|---|
| 1,000 | 592 | 5 ×3 | 179 | 0.005 | 0.031 | 0.036 |
| 2,000 | 958 | 6 ×4 | 279 | 0.003 | 0.06 | 0.063 |
| 4,000 | 1,515 | 9 ×5 | 398 | 0.01 | 0.141 | 0.151 |
| 8,000 | 2,416 | 12 ×7 | 616 | 0.023 | 0.428 | 0.451 |
| 16,000 | 3,830 | 17 ×10 | 955 | 0.05 | 1.056 | 1.106 |
| 32,000 | 5,888 | 24 ×14 | 1,574 | 0.094 | 3.66 | 3.754 |
| 64,000 | 8,909 | 33 ×20 | 2,709 | 0.325 | 14.875 | 15.2 |
| 128,000 | 13,440 | 47 ×28 | 4,831 | 1.164 | 71.359 | 72.523 |

Tab. 1: Timing results for random rectangles.

A number of notes are in order:

- The removal rate is quite high, e.g. for 16,000 rectangles 76% of the input rectangles are removed, whereas for 128,000 the removal rate is 89%.
- The algorithm requires quite a bit of memory to hold the indexes of rectangles, e.g. for 16,000 rectangles the memory requirement is about 6% of the total number of rectangles per non-empty cell, and for 128,000 this drops to about 4%.
- For each query rectangle all rectangles that may be contained inside this rectangle are readily available as their indexes are registered with the cells that cover the query rectangle. No search is necessary.
- The growth rate of the algorithm is very good up until about 16,000 rectangles (nearly linear). Then, because of the increased density, the performance deteriorates rapidly.
- The preparation time is negligible: 5% for 16,000 rectangles and only about 2% for 128,000.

For a few thousand rectangles the performance of the algorithm is very good, it exhibits a nearly linear characteristic. The only disadvantage is space requirement. If this is an issue, the method may not be viable. However, if shapes are processed one at a time, then the memory usage is a non-issue. It turns out, however, that the extra storage requirement and the cell data structure allow easy parallelization, which is the subject of the next section.

## 7. PARALLEL IMPLEMENTATION

The proliferation of GPUs (Graphics Processing Unit) into everyday computing opened up new opportunities for algorithm design. GPUs are used not only for graphics processing but also for other computing tasks such as cutter path generation. The trick is to transform the problem into a graphics problem, solve it with a GPU and then transform the result back to the original domain.

The cell structure provides a natural way to removing rectangle containment in parallel. The method works as follows:

- Create a stand-alone version of the rectangle eliminator and load it into each processor.
- Prepare the data in the main processor as before, i.e. build the cell structure and bin each rectangle into a set of cells.
- Group the cells into a set of sub-groups, e.g. 1-by-1 or 2-by-2, and for each group do:
  - Collect all rectangles registered with the group.
  - Pass these rectangles to individual processors with an access to the global cell index and rectangle data structures.

- Have the processors eliminate containment and update the global cell and rectangle data structures.

The issue that needs a bit of care is how to grant access to the processors in case many of them want to update the global (master) index and the rectangle data structure. The main processor holds the global cell and rectangle data structures, whereas individual sub-processors receive only a subset of the rectangles, e.g. for 8,000 input rectangles each processors gets on average about 600 rectangles, and build their own local cell structures for local elimination. Once a rectangle is found to be eliminated, its index is removed from the local as well as the master cell structure (to which the processor has access) and the rectangle is eliminated from both the local as well as the global rectangle arrays.

Table 2 shows timing results for a 1-by-1 subarray of cells, i.e. each cell is passed onto individual processors.

| n | $n_x \times n_y$ | k | $\infty$ −core | 64 −core | 128 −core |
|---|---|---|---|---|---|
| 1,000 | 5 ×3 | 179 | 0.0020 | 0.0020 | 0.0020 |
| 2,000 | 6 ×4 | 279 | 0.0025 | 0.0025 | 0.0025 |
| 4,000 | 9 ×5 | 398 | 0.0048 | 0.0048 | 0.0048 |
| 8,000 | 12 ×7 | 616 | 0.0099 | 0.0198 | 0.0099 |
| 16,000 | 17 ×10 | 955 | 0.0171 | 0.0515 | 0.0343 |
| 32,000 | 24 ×14 | 1,574 | 0.0294 | 0.1796 | 0.0883 |
| 64,000 | 33 ×20 | 2,709 | 0.0643 | 0.7074 | 0.3858 |
| 128,000 | 47 ×28 | 4,831 | 0.1376 | 2.8914 | 1.5145 |

Tab. 2: Timing results obtained via a GPU.

The data is interpreted as follows:

- The $\infty - core$ means that each cell receives its own processor, i.e. there is no shortage of processor. For example, for 8,000 rectangles there are $12 \times 7 = 84$ processors, whereas for 128,000 rectangles there are 1,316 processors ($47 \times 28$) at our disposal.
- The timing results are average processing times since each processor receives different number of rectangles.
- Assuming no shortage in processors, the rectangle elimination problem can be solved in linear time, as the timing results so indicate. In fact, with a 128-core GPU the problem is tackled in linear time up to 8,000 points.
- Once we run out of processors, the data has to be run in batches, e.g. for 8,000 points on a 64-core machine the rectangles are processed in 2 batches, in the first batch 64 cells are processed and in the second the remaining 20 are handled ($12 \times 7 = 84$ require 64 plus 20 processors).
- For a larger number of rectangles the elimination remains to be exponential after some number of rectangles, although the timing results are much better than for single processors (for 128,000 rectangles 1.5 seconds on a 128-core processor vs. 72 seconds on a single processor!).

Other cell grouping can also be tested. In fact, it is advisable to add the cell grouping to the list of parameters that the algorithm has to entertain customization for specific input types (in addition to the cell size and the static array size).

## 8. COMPARISON

In comparing our method to previously published results we find several differences. First, all report worst case performances which may not be applicable in the majority of practical cases. Second, there are no implementation considerations which could be an issue from a software engineering standpoint, e.g. maintenance. Third, there are no practical timing results broken down to the components of the algorithm. For example, divide-and-conquer is a very fine method, however, the division and the merge steps may be quite intricate resulting in complicated implementations. Fourth, most methods use a fairly complicated data structure. Implementing, maintaining and porting such data structures may be a challenge. In the world of software systems simplicity is king, even if it comes at a price of some extra memory requirement. Fifth, no parallel implementation is given in any of the prior work even though the problem admits easy parallelization and the proliferation of GPU into everyday computing requires such consideration. Sixth, most data structures used are static. Our method is dynamic and allows the insertion and deletion of rectangles (dropping a rectangles into the cell structure is a trivial exercise). Finally, prior works give no considerations to tolerances. This is an important issue as many of the applications are in the floating point domain and run into cases of mutual enclosures or containments that need to be handled carefully.

## 9. CONCLUSIONS

We presented an algorithm for the elimination of rectangle containments as well as enclosures with a method that operates in 2-D and uses a simple cell-based data structure to pre-process the rectangles for fast elimination. The advantages of the algorithm are: it is very simple to implement, it requires almost no maintenance, has free parameters that allow customizing the method to specific data types, it performs quite fast up to about 10,000 rectangles and it is easily parallelizable. The one shortcoming the algorithm has is that it requires some extra memory, although the only data that needs to be stored are the indexes of the rectangles.

## ACKNOWLEDGEMENTS

This work was supported in part by Synopsys, Inc. and Najran University. All opinions, findings and conclusions are those of the authors and do not necessarily reflect the funding agencies or the institutions the authors are affiliated with.

## REFERENCES

[1] Abel, D. J.; Smith, J. L.: A data structure and algorithm based on a linear key for a rectangle retrieval problem, Computer Vision, Graphics and Image Processing, 24(1), 1983, 1–13.

[2] Bistiolas, V.; Sofotassios, D.; Tsakalidis, A.: Computing rectangle enclosures, Computational Geometry: Theory and Applications, 2(6), 1993, 303–308.

[3] Bozanis, P.; Kitsios, N.; Makris, C.; Tsakalidis, A.: The space-optimal version of a known rectangle enclosure reporting algorithm, Information Processing Letters, 61(1), 1997, 37–41.

[4] Gupta, P.; Janardan, R.; Smid, M.; Dasgupta, B.: The rectangle enclosure and point-dominance problems, International Journal of Computational Geometry and Applications, 7(5), 1997, 437–457.

[5] Lagogiannis, G.; Makris, C.; Tsakalidis, A.: A new algorithm for rectangle enclosure reporting, Information Processing Letters, 72(5–6), 1999, 177–182.

[6] Lee, D. T.; Preparata, F. P.: An improved algorithm for the rectangle enclosure problem, Journal of Algorithms, 3(3), 1982, 218–224.

[7] Piegl, L.; Tiller, W.: The NURBS Book, Springer-Verlag, 1997.

[8] Vaishnavi, V.; Wood, D.: Data structures for the rectangle containment and enclosure problems, Computer Graphics and Image Processing, 13(4), 1980, 372–384.