Simple Rectangular Intersection and Grouping Methods for Specification-Based Image Operation Optimization

Submitted to:

Dr. Le Gruenwald

Submitted by:

Jianting Zhang

November 20, 2000
Index

1 Introduction and Related work ................................................................. 3
  1.1 Specification-based image operation its optimization .......................... 3
  1.2 Region Intersection for optimization .............................................. 4

2 Proposed Methods .................................................................................. 6
  2.1 Theoretical Analysis of benefits using intersection .......................... 6
  2.2 Intersection for two rectangles ...................................................... 8
  2.3 Intersection for N rectangles ........................................................ 8
    2.3.1 Failure of the Incremental Intersection method ......................... 9
    2.3.2 Algorithm for concurrent N-Rectangle Intersection ................... 10
  2.4 Grouping defined regions into non-overlapped region groups ............ 11

3 Implementations of proposed methods ................................................ 14

4 Conclusion and Future work ............................................................... 17

5 Reference ............................................................................................. 19
1 Introduction and Related work

1.1 Specification-based image operation its optimization

Image, as an important data type in multimedia, has been widely used in everyday lives and extensively studied in academic researches. In several fields, such as signal processing, computer vision and multimedia database, more and more operations have been developed to extract information from images (Sonka, 1999). The general schema of image processing is to apply a sequence of operations on the original images one by one.

The idea of only storing base image and operations instead of all the derived images in the process has been proposed and it is generally known as specification. Logical Modeling Langue or LML defines a standard set of operations include DEFINE, MUTATE, MODIFY, COMBINE and MERGE (Speegle, 1998). Discussions on the minimality and independence of these operations can be found at (Brown, 1998).

Since specification defines a set of primitive image operators, like set operators provides opportunities for SQL operations optimization, a natural extension is to optimize image operations based on specifications to further reduce processing time. Essentially, a specification-based optimization problem is to recognize patterns from an operation list and replace them with equivalent patterns. The equivalent patterns should be better than original ones according to different criteria or their combinations, such as storage and time cost. For image processing, usually a shorter operation list on a smaller region means fewer operations to be performed and thus less storage and time are needed. In this study, the primary goal is to replace an operation list with an equivalent and shorter one on reduced regions.
Some primary work on specification optimization based on LML has been done (Zou, 1998). His study addressed several important issues in specification-based image operation optimization, such as characterizing define and change operators, distinguishing between Take-out Optimization and Composition Optimization, static and dynamic optimization, etc. However, there are some disadvantages in his methods:

1. All the studied optimization methods either only works on the same defined region or only works on intersected region. This is a heavy restriction on image operation optimization.

2. Its implementation scans entries in the specification list linearly (from top to bottom). Thus it has to go over all the entries from beginning to current in order to check whether the entries satisfy a certain optimization pattern defined. The time complexity in this case is $O(N^2)$ which is unnecessary.

In this study, we will propose region intersection to solve the first problem and region grouping to solve the second problem. We call the methods “Simple” because they are somehow intuitive and do not complex index (such as R-tree) support.

### 1.2 Region Intersection for optimization

Suppose a sequence of operations $A_i$ works on region $R_1$ and a sequence of operations $B_i$ works on region $R_2$, they have the intersected part $R_{12}$. Since they are overlapped, operations A and B can not be optimized in Zou’s method. An intuitive idea is to intersect these two regions and generate three sub-regions which are $R_1-R_{12}$, $R_{12}$ and $R_2-R_{12}$. Then these three sub-regions can be optimized separately.

Basically there are two ways to achieve this goal. One is to take the original two rectangles as polygons and use map algebra (which has been implemented in most GIS
software) to intersect these two regions. It is clear that the resulted sub-regions are convex polygons. As more and more defined regions are intersected, the shapes of these polygons are hard to predict. Another way is to keep the sub-regions as rectangles. In most cases, this will result more sub-regions. Unlike the convex polygonal intersection which will generate no more than 3 intersected polygons, the intersection of two rectangles will generate at most 7 rectangles. In fig 1 below, two polygonal intersection results 3 polygons, rectangular intersection results 5 rectangles.

![Fig. 1 Illustration of polygonal and rectangular region intersection](image)

Polygonal intersection has been extensively studied in computational geometry. There are a lot of methods to solve this problem. Most of them require the original polygons to be convex polygons. Although new methods on the intersection of polygons with holes and other non-convex polygons have also been proposed, non-convex polygons intersection is still a theoretical and practical difficult problem.

Generally speaking, the time complexity of intersection of two convex polygons is $O(M*N)$ where $M$ and $N$ are the number of points in the corresponding polygons respectively. Most polygon intersection algorithms are very complex and only few companies provide good commercial software to solve this problem.
On the other hand, rectangular intersection is relatively simple. Two rectangles intersection will result no more than 7 non-overlapped rectangles. There are several advantages of using rectangle instead of polygon to define a region. First, keeping the intersected region in the shape of rectangle is in accordance with mouse-based interactive region definition habit of human operations in image processing. Second, a lot of image operations, such as convolution (COMBINE in LML) which is frequently used in image processing, usually require rectangular region. Third, unlike polygon which has variant length of points, all rectangles have fixed length and can be easily stored in a relational database for further use purposes. However, no work on specifications-based optimization of image operations using rectangular intersection (i.e., deal with overlapped regions) has been reported our best knowledge so far.

In the rest of this report, Section 2 introduces the proposed method. In this section, some theoretical analysis will be performed on how the rectangular intersection can improve system performance of image operation on MODIFY and MERGE operations. Then the method is extended to deal with intersections between N rectangles and two methods namely concurrent and incremental intersection are discussed. After solving the first problem in Zou’s study, this paper then introduces a region grouping method to solve the second problem. Section 3 covers the implementation and simulation result. Finally, section 4 made some remarks and proposed some future work directions.

2 Proposed Methods

2.1 Theoretical Analysis of benefits using intersection

Among the 5 operations defined in LML, COMBINE and MUTATE are not suitable for optimization on intersected rectangles. The reason is that in these operations,
calculating new value of a pixel requires neighborhood pixel values. When the new or old pixels are near the boundary area, they will require information from some other parts of intersected rectangles which is expensive to keep after intersection.

On the contrary, MODIFY and MERGE operations are suitable for optimization on intersected regions. The reason is that they directly works on individual pixel level and do not require neighborhood pixel information. Actually, MODIFY and MERGE (using opaque merge option) are not mutually independent. In both operations, the value of a pixel is changed to a new value either according to the modify rules or corresponding pixel values at the specific geometric location.

For changing the value at individual pixel level, following definitions mentioned above, suppose there are \( K_1 \) operations on region \( R_1 \) can be optimized into \( K_1' \) and \( K_2 \) operations on region \( R_2 \) can be optimized into \( K_2' \). Options on the intersected region can be optimized from \((K_1+K_2)\) to \((K_{12}')\). Rows and columns in \( R_1 \) and \( R_2 \) are \((M_1,N_1)\) and \((M_2,N_2)\) respectively. Rows and columns in the intersected region \( R_{12} \) is \((M_{12},N_{12})\).

Then computation saved on the non-intersection-based optimization is \((M_1*N_1)*(K_1-K_1')+(M_2*N_2)*(K_2-K_2')\). The computation saved on the intersection-based optimization is \((M_1*N_1-M_{12}*N_{12})*(K_1-K_1')+(M_2*N_2-M_{12}*N_{12})*(K_2-K_2')+ M_{12}*N_{12}*(K_1+K_2-K_{12}')\).

The difference between the two is as follows:

\[
\begin{align*}
& [(M_1*N_1-M_{12}*N_{12})*(K_1-K_1')+(M_2*N_2-M_{12}*N_{12})*(K_2-K_2')] + M_{12}*N_{12}*(K_1+K_2-K_{12}') \\
& = M_{12}*N_{12}*(K_1'+K_2'-K_{12}')
\end{align*}
\]

Clearly, the more \( M_{12}, N_{12} \) and \((K_1'+K_2'-K_{12}')\) are, the more computation this method can save. Since the intersected region has longer operation list, usually we can expect \( K_{12}' \) is much less than \( K_1'+K_2' \).
Next I’ll illustrate how to generate all the intersected rectangles for 2 originally involved rectangles and then extend it to the intersections of N rectangles.

### 2.2 Intersection for two rectangles

There are several situations for the intersections of two rectangles. The two rectangles could be disjoint, intersect and containing one in the other. The method should be generic to deal with all of these situations. As we know, in each directions (X and Y), there are at most 4 points for two rectangles, thus the possible combinations are (4-1)*(4-1)=9. However, at least two of them are not valid. The proposed method is as follows:

1. Retrieve all the x coordinators and y coordinators from the four points in the two rectangles into two arrays (x and y respectively). Denote the elements in the two arrays as x1,x2,x3,x4,y1,y2,y3,y4.
2. If (x3>x2) or (x1>x4) or (y3>y2) or (y1>y4) return the input two rectangles.
3. Sort the two arrays in ascending order.
4. Check each of the following 9 candidate rectangles and add the successful one to result list.
   - (x1,y1), (x2,y2), drop it if (x1,y1) is not a lower-left corner of either of the original rectangles
   - (x1,y2), (x2,y3)
   - (x1,y3), (x2,y4), drop it if (x1,y4) is not an upper-left corner of either of the original rectangles
   - (x2,y1), (x3,y2)
   - (x2,y2), (x3,y3)
   - (x2,y3), (x3,y4)
   - (x3,y1), (x4,y2), drop it if (x1,y1) is not a lower-right corner of either of the original rectangles
   - (x3,y2), (x4,y3)
   - (x3,y3), (x4,y4), drop it if (x4,y4) is not an upper-right corner of either of the original rectangles

### 2.3 Intersection for N rectangles

To extend the intersection of two rectangles to N rectangles, it is intuitive to see that there are two options. One is to perform two rectangles intersection first and then
incrementally perform intersections for a new rectangle with all the intersected rectangles (incremental intersection). Another way is to intersect all N rectangles at the same time (concurrent intersection). Some observations are as follows which might help understanding the method being discussed better:

1. Intersecting of two rectangles will result no more than 7 rectangles.
2. Intersecting of N rectangles will result no more than \((2N-1)(2N-1)\) rectangles.
3. Intersecting of a rectangle with existing N rectangles will add no more than \(8N\) rectangles. \(^1\)

In this section I’ll present a case to show that the intuitive incremental intersection method is not always correct and present some analysis on concurrent intersection.

### 2.3.1 Failure of the Incremental Intersection method

The intuitive Incremental Intersection method is as follows:

1. Select the first two rectangles and intersect them, put the intersected rectangles in the result list.
2. For each of the rest rectangles \(r_1\), do
3.  For each of the rectangle in the result list \(r_2\), do
4.      If \(r_1\) intersects with \(r_2\) then
5.         Remove \(r_2\) from result list
6.         Add the intersected rectangles of \(r_1\) and \(r_2\) to result list.
7.      End if
8.  End for
9. End For

However, in fig 2, suppose rectangle 1 and rectangle 2 are previously resulted intersected rectangle list. Obviously 1 and 2 are non-overlapped. Suppose we want to intersect rectangle 3 with them. Following the above method, one of the resulted

\[1 \text{ Proof: } (2*(N+1)-1)*(2*(N+1)-1)-(2N-1)*(2N-1))=8N\]
rectangles when intersecting rectangle 1 with 3 will be rectangle (b+c) and one of the resulted rectangles when intersecting rectangle 2 with 3 will be rectangle (a+b), thus rectangle b is double-counted. This case demonstrates that this method is not always correct.

To correct error, it requires checking all the newly intersected rectangles with the next rectangle to be intersected. For example, in the above case, after the intersecting of rectangle 1 and 3, we need to check all the resulted rectangles in the intersection (including rectangle (b+c)) with rectangle 3 in the next loop. By intersecting rectangle (b+c) with rectangle 3 we can get the correct result. However, this will dramatically increase its complexity. Thus this method makes itself uninteresting to us.

2.3.2 Algorithm for concurrent N-Rectangle Intersection

The concurrent intersection is also an extension of intersection between two rectangles. The method is as follows:

1. Extract x and y coordinates of the points in the N rectangles into two arrays with size of 2N.
2. Sort these two arrays in ascending order.
3. For each of elements of i in X array from 0 to 2*N-1
4.       For each of elements of j in Y array from 0 to 2*N-1
5.           Build a rectangle (tempRect) with the following 4 coordinates
6.           (X[i],Y[j], X[i+1],y[j+1])
7.           For each of rectangles of Rect[k] in original rectangle list
8.               If tempRect is within Rect[k] then
9.                   Add Rect[k] to result rectangle list
10.              End if
11.           End For k
12.       End For j
13.    End For i
14.  End For k
Clearly, the complexity of this method in the worst case is $4 \times O(N \times (2N-1) \times (2N-1))$ comparisons and in the average case is $4 \times (N/2) \times (2N-1) \times (2N-1)$. As far as complexity is concern, it is inferior to incremental intersection. Although the complexity of worst case in incremental intersection is about 1/3 more than concurrent intersection\(^2\), the newly added rectangles in each 2-rectangle intersection are far less than 8*N in practice and thus incremental intersection is more efficient. In fact, it is often the case that the number of addition is a constant number around 5-10, thus the complexity of incremental intersection could be further reduced to $O(N \times N)$.\(^3\) This is the reason why we prefer incremental intersection to concurrent intersection. However, since incremental intersection is not always correct, we will have to stick to concurrent intersection if no further action is performed. Fortunately, when the regions defined are clustered, we can first perform region grouping before intersection to reduce time complexity which will be discussed below.

### 2.4 Grouping defined regions into non-overlapped region groups

When N is big and the N involved rectangles are unevenly distributed, it is possible to use a pre-processing method to break the big N into several small $N_i$ to reduce complexity. The idea is to scan all the defined regions and divide them into several non-

---

\(^2\) $X_{i+1} = X_i + 8i \Rightarrow X_i = 4i \times (i-1), S = \sum x_i = 4 \times N \times (N \times N - 1)/3 = 1/3 \times (2N^2 \times 2N - 4)$

Since it needs 4 comparisons in each intersection of two rectangles, the total number of comparisons are $4/3 \times (2N^2 \times (2N - 2))$

\(^3\) $X_{i+1} = X_i + A \Rightarrow X_i = A \times (i - 1), S = \sum x_i = A \times N \times (N - 1)/2$
overlapped region groups. Then we can use the intersection method to intersect rectangles in each region group.

The preprocessing method is as follows:

1. Set up a vector to record non-overlapped region groups.
2. Set up a vector of vector to record rectangles in all non-overlapped region groups.
3. Set up a vector to record all the rectangles of the first region group whose range intersects with r.
4. For each region rectangle r to be added
5.       Set up a boolean array (markDelete) with the size of number of region groups and set all the initial values in the array to false.
6. Set up a boolean variable isTheFirst and set its initial value to false
7. For each region group rectangle
8.       Get the region group rectangle as Range
9.       If (r intersects with Range) then
10.          Get the vector of rectangles in the region group
11.          If isTheFirst is true then
12.              Set isTheFirst to false
13.              Record the region group number (N_i)
14.              Set Range as the union of Range and r
15.              Add r to the current region group
16.              Set the current group to firstGroup.
17.          Else
18.              Set the range of current region group as the union of range of N_i region group and r
19.          For each rectangles in the current region group
20.              Add the rectangle to the N_i region group
21.          End for
22.          Set current element in the markDelete to true
23.       End If
24.       End if
25.     End for
26.     Set up a vector to record resulted ranges of region groups
27.     Set up a vector of vector to record resulted rectangles in corresponding region groups.
28.     For each region group
29.         If current element in markDelete is false
30.         Add the range of current region group to new region group vector
31. Add the vector of rectangles in the current region group to new region group rectangle vector.
32. If isTheFirst is false
33. Add r to the new region group vector
34. New a vector, add r to the vector and then add the vector to the vector of the new vector of rectangles in each region group.
35. Replace old vector of region group with the new one.
36. Replace old vector of vector of rectangles in each region group with new one.
37. End for

By dividing the original N rectangles into M region groups (suppose each region group has $N_i$ rectangles), then we can reduce the worst complexity of concurrent intersection method from $O(N*(2N-1)*(2N-1))$ to $O(\sum_{i=0}^{M-1} N_i*(2N_i-1)*(2N_i-1))$, Where i is from 0 to M-1. This reduce of computation will be very significant when the defined regions are clustered.

Note that this method is actually a simplified version of spatial index techniques such as R-Tree. Unlike R-trees which have multiple spatial levels, this method only have two levels. When N is moderately large, this method might out-perform R-trees since no node split is needed which is very costly. Another advantage of this method is its straightforwardness and simplicity. The method here is not to compare with spatial index techniques, rather it provides an improvement to Zou’s study which requires looking from beginning entry to the very latest entry when a new entry is being processed in order to decide whether two operations are working on the same or intersected region. By using this method, system only need to the region of the entry to region groups instead of all the defined regions and thus improvement is achieved.
3 Implementations of proposed methods

Java programming language is used in this study. For the purpose of completeness, although only define, modify and merge are involved in this study, all the five operators in LML are implemented using Java Advanced Imaging (JAI) class package. A Java class named LMLImage is defined to represent a defined region and all operations on this region. Fig 3 lists the major functions in these two classes.

```java
class LMLImage
{
    BufferedImage opImage=null; //data buffer
    Rectangle r=null; //region defined
    Point p=null; //original point
    int mergeType=-1; //merge type
    Vector operations; //operation list

    LMLImage(); //constructor
    BufferedImage getBufferedImage(); //return processed image
    void addOp(String op); //add an operation to the operation list.
    void process(boolean restart); //call to instantiation
    void defineOp(BufferedImage origin, int definedRegion[][]); //define operator
    void modifyOp(byte[][] modifyArray); //modify operator
    void mutateOp(float[][] mutateArray); //mutate operator
    void combineOp(int[][] combineArray); //combine operator
    void mergeOp(int mergeType, int x, int y); //merge operator
}
```

Figure 3 A Java class for LML instantiation

The attributes inside this class are: A BufferedImage to store the resulted image when instantiation is called; A Rectangle to store the range of defined region; A Point is defined to store the original point when doing merge operation; An integer is defined to store the merge type and finally a Vector is defined to store all the operations on this region. Five operations which are corresponding to the 5 operations defined in LML are implemented and are ready to be called inside or outside the LMLImage class. In this
implementation, there are two ways to use this class. The first way is that each operation on the image is instantiated right after the corresponding operation is called. The second is to store the operations in the operation vector. When function “process“ is explicitly called, the system will process the whole operation vector and return the final result. The second way gives the system opportunity to perform operation optimization. Fig. 4 shows the result of instantiation based on the operation list.

Fig 4. Result of LML instantiation

The two major methods discussed in this report, namely concurrent rectangular intersection and region grouping are also implemented as java classes. For demonstrate purpose, all these two classes are implemented as the subclass of Panel, both the original
rectangles and grouped regions/intersected rectangles are shown visually as the processes are going to help user better understand how the methods works. Fig. 5 shows the simulated result of concurrent rectangular intersection, five rectangles involved are (100,10,100,100), (60,60,100,100), (60,180,100,100), (100,180,100,100) and (150,250,50,50) respectively. Fig. 6 shows simulated result of region grouping, 10 involved rectangles are randomly generated with center at (rand() *50, rand()*50) and (rand() *50+100, rand()*50+100) in turn and width and height of (rand() *50, rand()*50) and (rand() *50+100, rand()*50+100) in turn.

![Simulated Result of Concurrent Rectangular Intersection](image)

Fig. 5  Simulated Result of Concurrent Rectangular Intersection.
4 Conclusion and Future work

The contributions of this study are as follows:

- Theoretically proved that rectangular intersection saves total operations need. Formulas are given on how many options are saved with non-intersection-based optimization, intersection-based optimization and their difference.

- Investigation on multiple rectangular intersection problem and their complexity are discussed although preliminary.

- Providing a case showing that incremental intersection is not always correct.

- Propose a simple method for region grouping without using spatial index to reduce concurrent intersection complexity problem.

- Implement the proposed methods as well as LML instantiation using Java programming language.

The proposed methods work well on large defined regions where their overlapped portion is usually also big. Among the 5 LML basic operations, MODIFY and MERGE
are most suitable for the proposed method while COMBINE and MUTATE are not so good.

There are two major remaining issues on this work. One is how to optimize MODIY operation at pixel level and the other is how spatial index can help region grouping and rectangle intersection when the number of defined regions are extremely large.
5 Reference


