Journal of Science and Technology ***49****, Number 5 (2011) 69-78*

A SIMPLE ALGORITHM FOR FINDING FAST, EXACTLY TILES INTERSECT WITH POLYGONS

Nam V. Nguyen[1], Nam M. Nguyen[1], Bac H. Le[2]

[1]*Vietbando, Vietnam*

[2]*Faculty of Information Technology, University of Science, VNU-HCM*

Received September 30, 2011

ABSTRACT

In this article, we present a simple algorithm that allows us to find tiles intersect quickly and precisely with a given polygon. This is the one of important tasks in maintaining tile system for Web Map services today. In this study, 75% of tiles can be successfully saved by use of our newly developed method as compared with the conventional Minimum Bounding Rectangle method when applying for administrative regions of Vietnam. In addition, this algorithm is simple and easy to implement.

Keywords. computational geometry, spatial data mining, web map, polygon.

1. INTRODUCTION

The maps we are familiar with are powered by tile sets – collections containing hundreds of thousands of individually rendered images that stitch together to form a larger map view. For instance, Microsoft (www.bing.com/maps/) and Google maps (maps.google.com) are such systems. Tile sets are useful because they allow users to pan and zoom around a map with a web browser, but creating and maintaining a tile set is challenging. Tile generation demands a considerable amount of computing power and can take days depending on the size of the region being rendered. At present, Microsoft's tile system [4] (only for road 2D) supports 24 levels of maps that means we must render and store $4^0 + 4^1 + \ldots + 4^{22} + 4^{23}$ tiles in storage devices as shown in Figure 1.

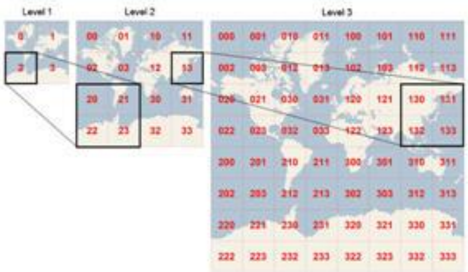

Figure 1. Tile System

(Resource from http://msdn.microsoft.com/en-us/library/cc451900.aspx)

In those systems, space is tessellated into a grid of tiles at each level, and each spatial object is represented by the set of tiles it intersects. When a region updated, we must specify which tiles have to be re-rendered. In principle, at each level, after getting a minimum bounding rectangle (MBR) of the region, we find tiles that intersect with MBR, here so-called T_1^R, and we only render these tiles. This method is simple and easy to implement.

However, as the difference between the bounding rectangle and the polygon is large, too many unexpected tiles (tiles that do not intersect with the polygon) are re-rendered. As you see in Figure 2, the number of the necessary tiles (in light red) is much smaller than that of the unexpected tiles. At the higher zoom level, more unexpected tiles increase. As a result, this algorithm actually is not so effective in practice. Another method should be mentioned here is that we can firstly find limits of row and column of the polygon, then check if the tiles intersect with polygon or not. This method leads to tile set as expected, but the cost for checking is so expensive.

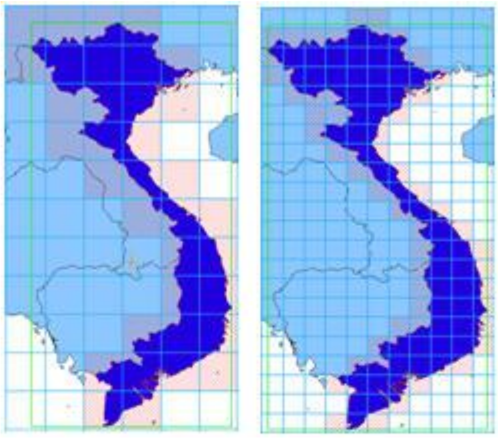

Figure 2. Spatial object is decomposed to a list of tiles at a given grid level

In this study, we solved all these issues by developing an algorithm that allows us to quickly and precisely find the tiles that intersect with a given polygon. Our algorithm uses border(s) of polygon as a guideline to specify tiles on border of polygon, and to encode these tiles in order to find all expected tiles. In addition, this algorithm is simple and easy to implement.

2. PRELIMINARIES

A regular grid *G* is a tessellation of the Euclidean plane by congruent rectangles. Each cell in the grid can be addressed by index (*r*, *c*), and each vertex has coordinates (*c* * *dx*, *r* * *dy*) for some real numbers *dx* and *dy* representing the grid spacing. For this article, we interpret the cell as a tile. Each tile has four sides: *left*, *top*, *right*, *bottom*. Each side is represented by two pairs of coordinates that can be calculated from coordinates of tile.

A simple polygon *P* is defined as an ordered list of vertices $P = \{v_0, v_1, v_2, \ldots, v_{n-1}\}$. A segment is a pair of two adjacent vertices $s_i = (v_i, v_{i+1})$ and $s_{n-1} = (v_{n-1}, v_0)$. In the case of the polygon *P* has more than one part (generally, one outer ring and some inner rings), we treat this polygon as a set of polygons – each ring is a polygon, so *P* is represented as $P=\{r_0, r_1, r_2, ..., r_{k-2}, r_{k-1}\}$ with r_0 is outer ring, and other r_i's are inner rings, we use term *polygon with hole(s)* for this polygon type, see Figure 3.

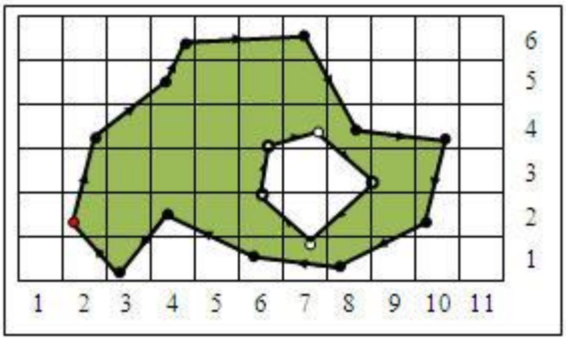

Figure 3. A polygon with hole(s)

Let *G* be a grid of tiles that covers fully a polygon *P*. Our task is to find set of tiles *T* that satisfies the following condition:

$$T=\{t_i : t_i \in G, t_i \cap P \neq \emptyset\}$$

Our approach based on checking if a segment intersects [5] with sides of tile to change tile state. In this algorithm, we need a data structure that allows finding a key quickly and to update state related to that key.

Let *M* be a map structure based on variants of B-Tree index [1, 2, 3]. *M* contains pairs of (*key*, *state*) where *key* is tile index and *state* indicates that number of cut-edge of the tile with *key* is odd or even, value of *state* will change when there is a segment goes out of the tile *key* as shown in Figure 4.

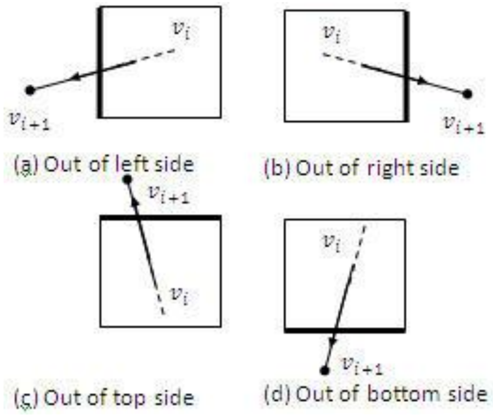

Figure 4. Illustrations for the segment that go out of the tile sides

This algorithm is depicted as following:

Input: Polygon P and grid of tile T

Output: Set of tile T_p intersects with P

$L_P^M \leftarrow \varnothing$

for each ring r_i in P

  $L_P^M \leftarrow M_i = FindBorderTiles(r_i, T)$

$M_P = MergeTiles(L_P^M)$

$T_P = MakeCellRanges(M_P)$

L_P^M is a list of M_i data structures. The *FindBorderTiles* function specifies tiles on the border of a ring and store these tiles in M_i. The *MergeTiles* function merges $M_i(s)$ into an only M_P of the P polygon. The *MakeCellRanges* function bases on M_P to finds all tiles that intersect with the P polygon.

3. ALGORITHM

3.1. Finding border tiles

Let $s_0 = (v_0, v_1)$, in order to find tiles of T that this segment crosses, in the *first step*, we find tile that contains vertex v_0. This tile names the current tile t_c. The segment s_0 is $s_i = (v_i, v_{i+1})$ with $i = 0$. In the *second step*, we check intersection of the s_i segment with sides of the t_c tile. If any side is cut, stop checking other sides of this tile. We don't care coordinates of intersection point, all we need know is which side of tile intersects with segment.

When segment s_i goes out of the current tile t_c, we update state of this tile in M_i data structure and find next tile t_n to continue. Note that tile t_c and tile t_n has one common edge, so we do not perform such "cut-checking" for this common edge for tile t_n because segment s_i goes into but goes out of tile t_n.

- Update state of tile
  State updating depends on which side is cut. We classify two groups: horizontal side (top and bottom) and vertical side (left, right). Each group has properly updating method. Here, we imply *EVEN* $\leftrightarrow$ true and *ODD* $\leftrightarrow$ false.
  With horizontal side, if this tile is available in M_i structure, nothing to do. Otherwise, we insert $(key, state) = (t_c, EVEN)$ into M_i.
  With vertical side,we have two following cases:
  - Cut-edge is left side of tile
    If this tile is available in M_i structure, state of this tile changes the following: $EVEN \rightarrow ODD$ and $ODD \rightarrow EVEN$. Otherwise, we insert $(key, state) = (t_c, ODD)$ into M_i.
  - Cut-edge is right side of tile

If this tile isn't available in M_i structure, we insert $(key, state) = (t_c, EVEN)$ into M_i. Otherwise, we find tile that is right of this tile to update.

- Finding the next tile

The next tile is specified by which side of the current tile the segment s_i goes out of. Let r_c and c_c are row index and column index of t_c correspondingly, the next tile is identified the following formula, see cases in Figure 4:

- $(a): r_n \leftarrow r_c, c_n \leftarrow c_c - 1$
- $(b): r_n \leftarrow r_c, c_n \leftarrow c_c + 1$
- $(c): r_n \leftarrow r_c - 1, c_n \leftarrow c_c$
- $(d): r_n \leftarrow r_c + 1, c_n \leftarrow c_c$

The next tile t_n becomes the current tile $t_c (t_c \leftarrow t_n)$. In Figure 5, the dot-line indicates the guideline for seeking border tiles.

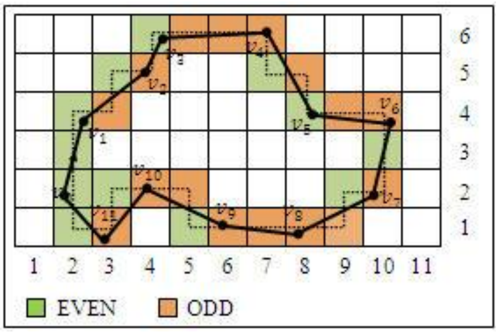

Figure 5. Moving on grid of tiles and border tiles

We repeat the second step until the vertex v_{i+1} of the segment s_i is inside any tile. Go to next segment $s_{i+1} = (v_{i+1}, v_{i+2})$ and apply the second step for this new segment. In the case both vertices of considering segment is contained in the same tile, we fire this segment and go to next segment s_{i+1}. Notes that with new segment we must check all sides of the current tile.

The finding process stops when all the segment of the polygon is considered. We have border tiles of ring and cut-edge state of these tiles.

By this searching method, number of odd cut-edge state in a column is always even. This feature is basis for finding all tiles that intersect with the *P* polygon.

3.2. Merging border tiles

The below procedure isn't applied for simple polygon that has no hole. Applying the *FindBorderTiles* function for each ring: $H_i = FindBorderTiles(r_i, T)$. Border tiles of polygon P is calculated as follows:

$$H = H_0 \cup H_1 \cup H_2 \ldots H_{k-2} \cup H_{k-1}$$

Supposing that polygon in Figure 5 has an inner ring as shown in Figure 6. The new polygon is shown in Figure 7.

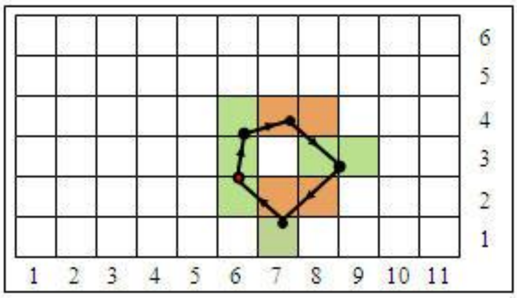

Figure 6. Border tiles of inner ring

The merge method is depicted the following:

$H \leftarrow H_0$

$foreach\ t_k.key \in H_i\{$

$\quad if\,(t_k.key \in H)\{$

$\qquad \text{state=!}(t_k^H.state \wedge t_k^{H_i}.state)$

$\qquad \text{H} \leftarrow (key, state)$

$\quad \}\text{else}\{$

$\qquad \text{H} \leftarrow (t_k.key, t_k.state)$

$\}$

The symbol ^ is XOR operator in Boolean algebra. Figure 8 shows encoded tiles of column 7 of polygon in Figure 7.

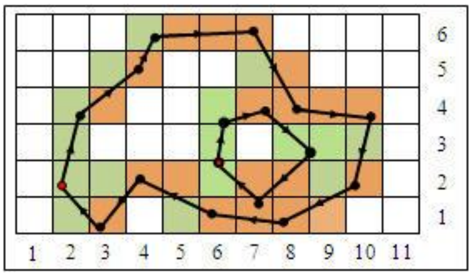

Figure 7. Encoding compound polygon-two rings

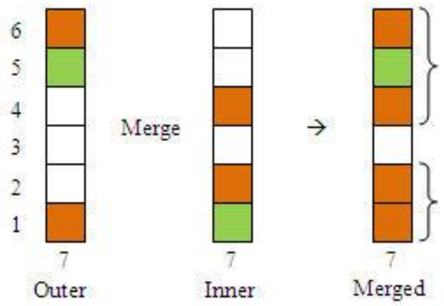

Figure 8. Border tiles of two rings

3.3. Finding all tiles that intersect with polygon

Let C_L and C_U be left most column and rightmost column of grid that intersect with the polygon P. T_P^c is all tiles of column c that intersect with the polygon P. To save storage space, we manage these tiles by list of ranges. Each column has a list of ranges. All tiles that intersect with the polygon P can be defined as the following:

$$T_p = T_P^{C_L} \cup T_P^{C_L+1} \cup T_P^{C_L+2} \cup ... \cup T_P^{C_U}$$

Our algorithm is depicted as follows:

$T_p \leftarrow \varnothing$

$A_p \leftarrow M_p$

Sort A_p order by the column and row

foreach c in $(\mathrm{C}_L : C_U)${

$T_P^c \leftarrow FindTileRanges(c)$

$\mathrm{T}_P \leftarrow T_P \cup T_P^c$

}

We create an array of border tiles A_P from M_P, then sort this array A_P in increasing order by column and row. After sorting, tiles in a column is grouped together and separated into some continuous tile blocks – see in Figure 9, we have two blocks in column 7:{1:2} and {4:6}. Basing on column value of each item in the array A_P, we can specify lower index I_l^c and upper index I_u^c of each column c in A_P - they are 0 and 3 for column 2 in Figure 9.

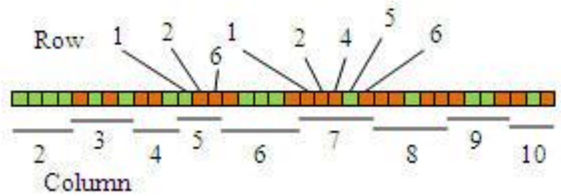

Figure 9. Sorted border tiles array

The $FindTileRanges$ function finds and merges blocks in the column c to create tile ranges. State of block determines adjacent blocks whether they can be merged to make bigger blocks or not. Block's state depends on number of ODD tiles N_{ODD}. If N_{ODD} is odd then block state is ODD. Otherwise, it is $EVEN$. Pairs of adjacent ODD blocks is merged into bigger blocks. State of new block is $EVEN$. Notes that, when two ODD blocks merged, new block can overlay one or some $EVEN$ blocks. Such $EVEN$ blocks should be deleted. After merging, there are only $EVEN$ blocks in the column and these blocks contain tiles that we want to find. These blocks are used to create ranges for this column.

After the sketch of the $FindTileRanges$ function, it's time to go into more details. The algorithm scans tiles t_k^c from I_l to $I_u (k \in [I_l, I_u])$ to compute state of blocks and to merge blocks as soon as possible. Let R_i^c is i^{th} range of the column c.

First step: The lower bound of the R_0^c range is $t_{I_l}^c.row$. Initial state of range R_0^c is $t_{I_l}^c.state$. *Second step*: Moving to higher index to find the upper bound of range R_i^c, block's state will change depending on next tile's state.

```
T_P^c ← ∅
k ← I_l
while (k < I_u){
    lower ← t_k^c.row
    state ← t_k^c.state
    k ← k+1
    while((k < I_u ) && 
            ((t_{k-1}^c.row = t_k^c.row − 1) || (state=ODD))){
                if (t_k^c.state = ODD){
                    if( state=ODD){
                        state ← EVEN
                    }else{
                        state ← ODD
                    }
                    k ← k+1
                }
                upper ← t_{k-1}^c.row
                T_P^c.Add(CreateNewRange(lower, upper))
    }
```

The tiles that have $EVEN$ state don't change state of block. If the expression $(t_{k-1}^c.row < t_k^c.row - 1)$ is satisfied, it means that the tile t_k^c belongs to the other block, the current state will decide whether this range can expand or not. We have two following cases:

- The parameter $state$ is $EVEN$: $t_{k-1}^c.row$ becomes the upper bound of the R_i^c range. Finding for the R_i^c range finishes. Starting a new range R_{i+1}^c with the lower bound is $t_k^c.row$. Go to *second step*.

– The parameter $state$ is ODD: this range will contain next block. That means the tile t_k^c and tiles that are located between these two blocks belong to this range R_i^c. Go to *second step*.

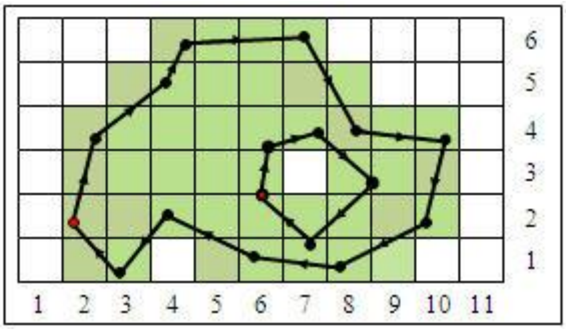

Figure 10. All green tiles intersect with polygon

Applying this algorithm for all columns in range $[C_L, C_U]$, we have expected tiles, figure 10.

4. EXPERIMENT

The experiments were done one core of an Intel® Core™ 2 Duo CPU E6750 @2.66GHz, 2.67 GHz, 2GB main memory. The program was compiler by the Visual Studio C++ 2008 compiler using optimization level 3.

We deal with the world borders data that was obtained from http://mappinghacks.com/data/. We executed our algorithm for Vietnam's border at some levels of tile system (see table 1). The results from *Minimum Boundary Rectangle* method is shown in column *MBR* and ones from our algorithm are shown in column *Extracted*. Calculating time of our algorithm is in column *Time*.

Table 1. Results of algorithm for Vietnam's border at some levels of tile system

Level	MBR	Extracted	Time(ms)
18	57,404,600	14,589,034	62
19	229,596,880	58,295,914	125
20	918,387,520	233,061,707	265
21	3,673,464,728	932,002,989	547
22	14,693,601,405	3,727,524,561	1,140
23	58,773,890,609	14,909,123,888	2,375
24	235,093,843,800	59,634,549,289	4,891

In table 1, it is clearly seen that our algorithm saved up to 75% of tiles as compared with the *MBR* method.The problems merging map tiles is solved completely for two polygon types: simple polygon and polygon with a hole(s). Thus, the algorithm can expand for other simpler geometry shapes such as point and polyline is very easy.

4. CONCLUSION

In this paper, we present a simple algorithm that allows us to find tiles intersect quickly and precisely with a given polygon. The implementation of this algorithm is very easy. All these features make the proposed algorithm very attractive to practical implementation. As shown in the results of the test on the simulated data source, it showed that the algorithm can be applied to maintain consistence of the tile system when their data source changed. The algorithm can apply not only for polygon but also for other geometry types such as polyline, multi-polyline and multi-polygon.

REFERENCES

1. T. Johnson and D. Shasha - The performance of current B-tree algorithms, ACM Transactions on Database Systems, 18(1):51-101, 1993.
2. T. Johnson and D. Shasha - Utilization of Btrees with inserts, deletes and modifies. In Proceedings of the 8th Symposium on Principles of Database Systems, pages 235-246. ACM, 1989.
3. Sai Wu, Dawei Jiang, Beng Chin Ooi and Kun-Lung Wu - Efficient B-tree Based Indexing for Cloud Data Processing. In Proceedings of the VLDB Endowment **3** (1) (2010).
4. http://www.microimages.com/documentation/cplates/76BingStructure.pdf
5. M. de Berg, M. van Kreveld, M. Overmars and O. Cheong - Computational Geometry Algorithms and Applications Third Edition, Springer, 2008.

Corresponding author:

Nam V. Nguyen,

Vietbando, HCMC, Vietnam.

Email: *nguyenvinhnam@vietbando.vn*