Laboratório VISGRAF
Instituto de Matemática Pura e Aplicada

TopQT: A Topological Data-Structure for Quadtrees

Esdras Medeiros, Luiz Velho, Helio Lopes

Technical Report TR-2005-05 Relatório Técnico

October - 2005 - Outubro

TopQT: A Topological Data-Structure for Quadtrees

ESDRAS MEDEIROS[1], LUIZ VELHO[1] HLIO LOPES[2]

[1]IMPA–Instituto de Matemática Pura e Aplicada
{esdras,lvelho}@visgraf.impa.br

[2]PUC-RIO–Pontifícia Universidade Católica do Rio de Janeiro
lopes@mat.puc-rio.br

Abstract. In this paper we propose a topological data structure for quadtrees, TopQT. This data structure is somewhat different from others very common in cell-complexes representations such as the famous half-edge. In the main result of this paper, we show that given a cell in the quadtree, TopQT is able to find all four connected neighbor cells in constant time. We give some pseudo algorithms to explain how to access the surrounding cells. These algorithms may point out to many nice applications in the computer graphics comunnity. We will cite some of them, specially the one that motivated this work.

1 Introduction

Quadtrees are rooted graph trees where every internal node has four children. They were developed by Finkel and Bentley [?] and since then, there have been hundreds of papers dealing with quadtrees. They are powerfull data-structures used in many areas such as computer graphics, image analysis, geographic information systems (GIS) and many others.

Typically, quadtrees are used to answer range queries, but they can be used for many other operations as well. Sometimes they are not used because no sublinear and even sublogarithmic bounds on the query time can be proved. For example, given a specific cell **c**, finding its four connected neighbors, i.e. all cell neighbors sharing the same edges of **c**, is in the worst case $O(\log n)$ (we will call this example as topological neighbors problem (TNP)). In this example, a topological data structure is a nice approach.

In Samet [?] we can find a usefull study on quadtrees and its applications. It describes how to solve (TNP) based only on hierarchical information. In Celes et al. [?] the authors describe a generic topological data structure for hierarchical planar subdivisions (HPS). Although they focus on GIS applications, it can also be used in quadtrees, however, with a high memory cost.

As long as we know there not exists a topological structure for quadtrees, at least, in the same way we will describe. In this paper, we propose hierarchical topological data structure for quadtrees (TopQT) that can, in the same time, solve TNP in $O(1)$ and save a great amount of memory cost if compared with Celes et al. [?]. It is also very flexible which means that the user can navigate through topological entities hierarchically and recover informations efficiently.

1.1 Contributions

As a planar subdivision, quadtrees present self similarities, i.e. all faces are squares with different sizes generated by a regular subdivision process. We projected TopQT taking advantage of this property by combining two key classes of data structure. The first one is the topological, commonly used in cell complexes (e.g. half-edge [?], quad-edge [?], corner-table [?], etc.), and the second one, is the hierarchical (e.g. quadtrees and binary trees).

The main features of TopQT can be summarized in three items:

- Constant time access to the four connected neighbors;
- Hierarchical access through topological entities;
- Simple and easy implementations of topological queries;

1.2 Overview

The outline of this paper is as follows. The next section describe in datails TopQT: discrete entities, the subdivision process and its inverse operation. Section 3 shows how to use it to perform topological queries, partilarly, how to draw neighbors cells. In section 4 we analyse some aspects of TopQT involving memory and complexity. In section 5 we show some powerfull applications that TopQT can be easilly assigned. Finally, in section 6 we give brief conclusion.

2 TopQT description

In this section we will describe the TopQT discrete entities, discuss some implementation details and justify several of its partilarities.

TopQT is basically composed of two entities: a quaternary tree, *qtcell*, and a binary tree, *qtedge*. One may observe that the *vertex* entity is not necessary.

Now we will go deep in each one. The *qtcell* data structure is, in fact, the node of an ordinary quadtree. It can be also interpreted as a generic HPS face. It contains three categories of information. The first one stores hierarchical data, namelly, the level of the tree, the parent cell and, finally, its four descendants. These parameters are represented by the identifiers *lv* for level, *par* for parent and *dct[i]* for each descent. The second category of data represents topological information. In our formulation, it is stored by four *qtedge* data structure references. We will describe about them latter. Finally, in the the third category of information, we have a reference to a generic data, *data*, responsible for storing in each cell of the tree the necessary information that depends on the user application (e.g. points, colors, curves, etc). In the following figure bellow one can see a detailed description of *qtcell*.

```
structure qtcell{
int       lv;
qtnode*   par;
qtnode*   dct[4];
qtedge*   e[4];
void*     data;
}
```

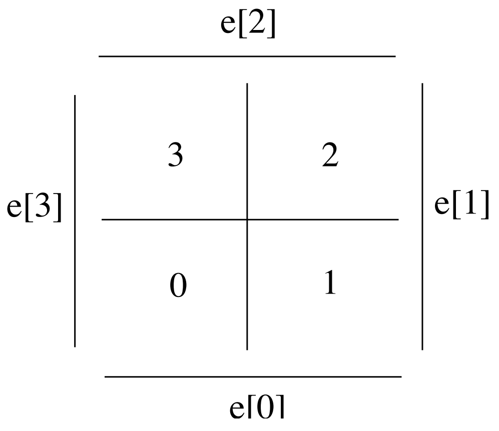


Figure 1: The cell data structure. It is composed of four edges and four subcells.

The other data structure we describe now is *qtedge*. It is a binary tree containing *par* as a parent and *dsc[i]* as descents. Topological data is stored in *nb[i]* parameters. They reference quadtree neighbors (see fig. **??**). The structure *qtedge* is a bridge that links hierarchy coming from an ordinary quadtree and the topology for cell-complexes respresetations. As we will see in section [**?**] it will provide simple queries implementations through the neighbors of one cell.

```
structure qtedge {
qtedge*  par;
qtedge*  dct[2];
qtcell*  nb[2];
}
```

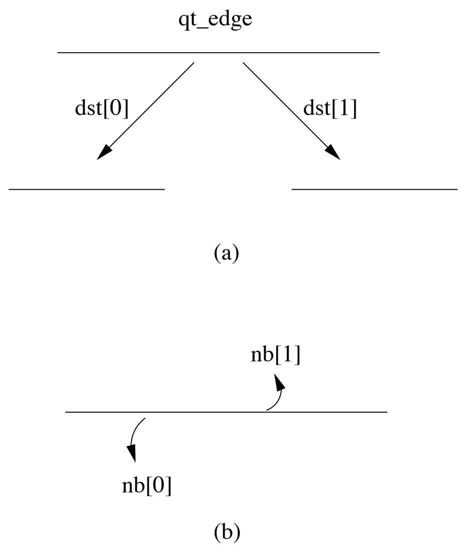


Figure 2: The edge data structure. It is composed of two subedges and two cell neighbors.

We adopt some conventions. The identification we give for each descendant cell follows the usual counter-clockwise orientation. For descendant cells we start numbering from the south-east (0) followed by the south-west (1), north-west (2) and north-east (3). The four edges are numbered in the same way : south (0), west (1), north (2) and east (3).

Conventions in *qtedge* depends on the edge orientation: vertical or horizontal. In the horizontal case the first descent (0) is on the left. The cell neighbor (0) is under the edge. It is always in a level less or equal to the the cell where the edge belongs. The vertical case is similar. Up to a 90^0 rotation, the conventions are the same of the horizontal case.

Unlike half-edges and similars data-structures, note that these conventions allow neighbors cells share the same edge. It brings a great advantage in memory cost.

Constructing TopQT: In each subdivision process we create four cells where each cell has four edges. As we observed before, descent cells are sharing some edges. For example, subdividing a given a cell **c** we have the following equality,

$$c.dsc[0].e[1] = c.dsc[1].e[3].$$

Some edges may be already subdivided. We just update their neigbors by traversing the binary tree until the actual level of the subdivided cell (see fig. **??**).

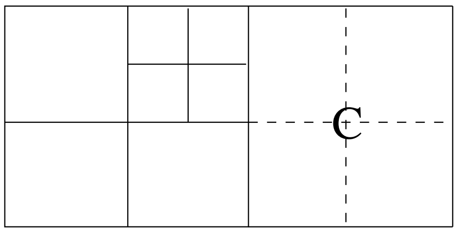


Figure 3: Subdividing a cell may deal with existent edges. In this figure cell C is being subdivided and its left edge is already subdivided.

Prunning a branch: Deleting is a node is quite easy because it is precisely a recursion of the inverse subdivsion process. Shared edges by others cells are not deleted.

Memory cost: Memory used by the TopQT, is clearly dependent on the number of cell nodes.

3 Accessing Neighbors

One of the most important utilities of TopQT is that it provides simple accessibility to the neighbors cells of a given node. In this section, we show how to achieve this potentiality through some pseudo algorithms.

The aim of our examples is to make a simple drawing in the display to show the accessed neighbors. However, the user can easily adaptate them to perform other operations.

We developed a function called *draw-four-neighbors* that receives a reference to a quadtree *q* and an integer number *lv*. As the name suggests, it draws the neighbors of the four internal edges. The latter parameter is optional. Its purpose is to restrict the hierarchical access by drawing only the cells with levels less or equal than *lv*. For example, if *lv* is the depth of the quadtree then *draw-four-neighbors* draws all surround leaves of the quadtree.

The core of this function is an auxiliary procedure called *draw-neighbors*. It analyses each internal edge by traversing its tree and drawing the corresponding neighbors (see algorithm **??**).

Algorithm 1 draw-four-neighbors(q, lv)

```
draw-neighbors(q→ $e[0]$, 0, lv);
draw-neighbors(q→ $e[1]$, 1, lv);
draw-neighbors(q→ $e[2]$, 1, lv);
draw-neighbors(q→ $e[3]$, 0, lv);
draw(q);
```

The *draw-neighbors* procedureid presented in algorithm **??**. It receives a reference to an edge *e*, the "side", the neighbor *sd* (assuming 0 or 1) and the maximal depth of the neighbor *lv*. As one can see, it is recursive.

Algorithm 2 draw-neighbors(e, sd, lv)

```
if e→dst[0]!=NULL and level(dst[0]) $<$ lv then
    draw-neighbors(e→dst[0], sd, lv);
    draw-neighbors(e→dst[1], sd, lv);
else
    if e→nb[sd]!=NULL then
        draw(e→nb[sd]);
    end if
end if
```

To complete this section we will show how to access the full connected neighbors of a cell. The lack of a vertex entity does not allow a direct access to them because we fall in a singularity, that is, the juntion of to edges. These singularities are present in the four corners of the cell and most of time their neighbors cannot be accessed by the cell edges. For simplicity we will call these cell neighbors as south-east (SW), south-west (SE), north-east (NE) and north-west (NW) (see fi.g **??**).

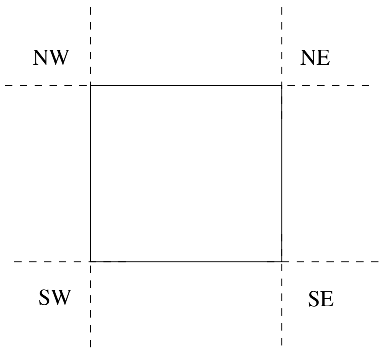


Figure 4: The cell corner neighbors.

There are many possible approaches to recover them. A complication arises when the neighbor of a cell corner may not exist. This situation happens when the neighbor cell level that the corner belongs is less than the level of the actual cell (see fig **??**).

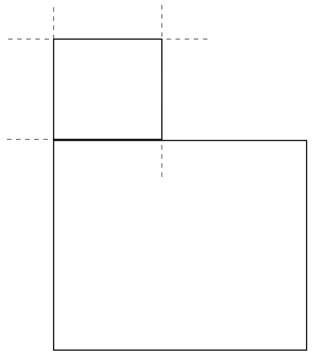


Figure 5: Dealing with non existence corner neighbor. In this figure the south-east corner of the upper cell does not exist or it is overlaped by another.

In our implementation ([**?**]) we use the geometric information to deal with the above situation. For example, if we wish to draw the SE corner neighbor, we must analyse its existence. This is performed by a geomettric test. Let $(a.x, a.y)$ and $(b.x, b.y)$ represent the south-west and the north-east corner positions of the cells, respectivelly. Then, the *if* statement condition of the the algorithm [**?**] is one possible test. It analyses correctly the level and the geometrics coincidences of the corners of the cells involved.

Solved the existence, in the next step, we must access the desired cell parameterized by the level. We just traverse the upper branch of left edge of the cell located under the edge $e[0]$.

The others corner neighbors are drawed in a similar way.

Algorithm 3 draw-south-west-neighbor(q, lv)

```
$e_0$ = e[0];
if ($e_0$ →nb[0] != NULL) and (q.bx == ($e_0$ →nb[0]).bx) then
    $e_1$= $e_0$ →nb[0]→e[1];
    while $e_1$ →dst[1] != NULL do
        $e_1$= $e_1$ →dst[1];
    end while
    draw($e_1$ →nb[1]);
end if
```

In figure **??** we have an illustration the output of these routines.

Complexity: The *draw-four-neighbors* is clearly linear and output sensitive, that is, it depends on the number of drawed cells. Then, each returned cell has, ina avarage, a constant time access. Observe that the complexity of *draw-south-west-neighbor* procedure and similiars, is proportional to the difference of the levels involved.

4 Applications

We now mention some potential applications that TopQT can be directly applied.

The first application is the approximation of implicit curves [**?**]. In the end of the quadtree construction we must return the poligonization, i.e, the order of the edges in the curve. This can be easily achieved by looking for the zeros of the funtion traversing the edges trees (see fig. **??**).

We can find another nice application computing the multilevel partition of the unity in the plane (MPU) [**?**]. In this beautifull work of Yutacaka et al. use MPU to approximate point surfaces using distance functions. They frequently access cells neighbors to find wheight funtions and TopQT simplify it. It can also save a lot of computation.

The last application we can cite is the one that inspired us for this work. The Restricted Ball-Privoting Algorithm [**?**] builds a triangulation level by level with an adaptive radius. It frequently builds a uniform matrix in the triangulation of each level. We can avoid this by using TopQT prunnig procedures. As the ball-pivoting step is local, which reduces drastically the overall computation, TopQT solves this trade-off by using its topological entities operations (see fig. **??**).

5 Discussion

In this paper we presented a topological approach for quadtrees that privides an easy and fast accessibility to neighbors cells of a given cell. We presented its discrete entities (*qtcell* and *qtedge*) and showed implementations of neighbors accessibility.

Basically, our work differs from Celes et l. [**?**] in the special treatment we give for quadtrees: TopQT compress, in a minimal framework, all necessary data which allows navigation through quadtree cells efficiently.

Like every ordinary quadtrees, TopQT is highly exepensive in memory cost if it is used in balanced quadtrees. However, there are extremal examples such as the circle where a TopQT and quadtrees are better than uniforme subdivision approachs.

In a future work we plan to generalize TopQT for heigher dimensions. We know that it is a very difficult problem because we must take care about memory costs.

Acknowledgements

This research has been developed in the VISGRAF Laboratory at IMPA with. VISGRAF is sponsored by CNPq, FAPERJ, FINEP and IBM Brasil.

References

[1] R. A. Finkel and J. L. Bentley, *Quad trees: a data structure for retrieval on composite keys*, Acta Inform, 4:1–9, 1974.

[2] H. Samet,*Applications of Spatial Data Structures*, Addison-Wesley, Reading, MA, 1990.

[3] W. C. Filho, L. H. Figueiredo, M. Gattass, P. C. Carvalho , A topological data structure for hierarchical planar subdivisions, Technical Report CS-95-53, University of Waterloo.

[4] Baumgart, B. *A polyhedron representation for computer vision.*, In National Computer Conference, AFIPS Conf. Proc., 589–596, 1975.

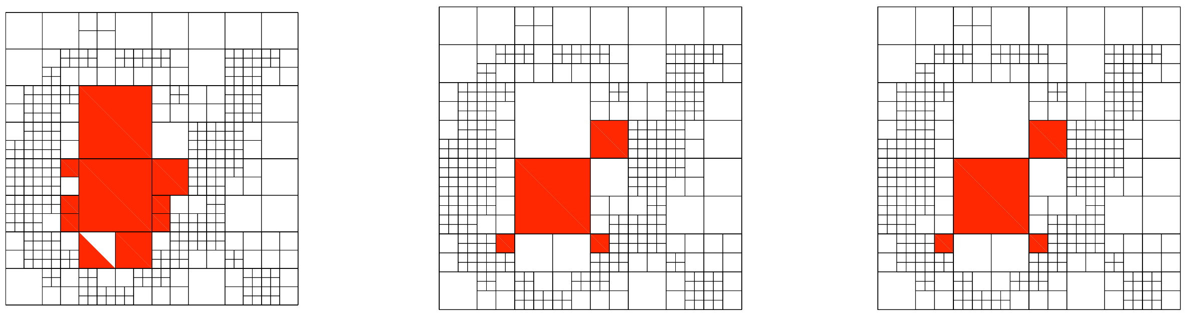
[5] L. J. Guibas and J. Stolfi. *Primitives for the Manipulation of General Subdivisions and the Computation of Vorono Diagrams*, ACM Transactions on Graphics 4(2):74–123, April 1985.

[6] J. Rossignac, A. Safonova, and A. Szymczak. *3D compression made simple: Edgebreaker on a Corner Table*, Shape Modeling International Conference, 278–283, Genoa, Italy May 2001.

[7] Y. Ohtake , A. Belyaev , M. Alexa , G. Turk, and H-P. Seidel. *Multi-level Partition of Unity Implicits*, Proc. SIGGRAPH 2003, 22(3), 463–470, 2003.

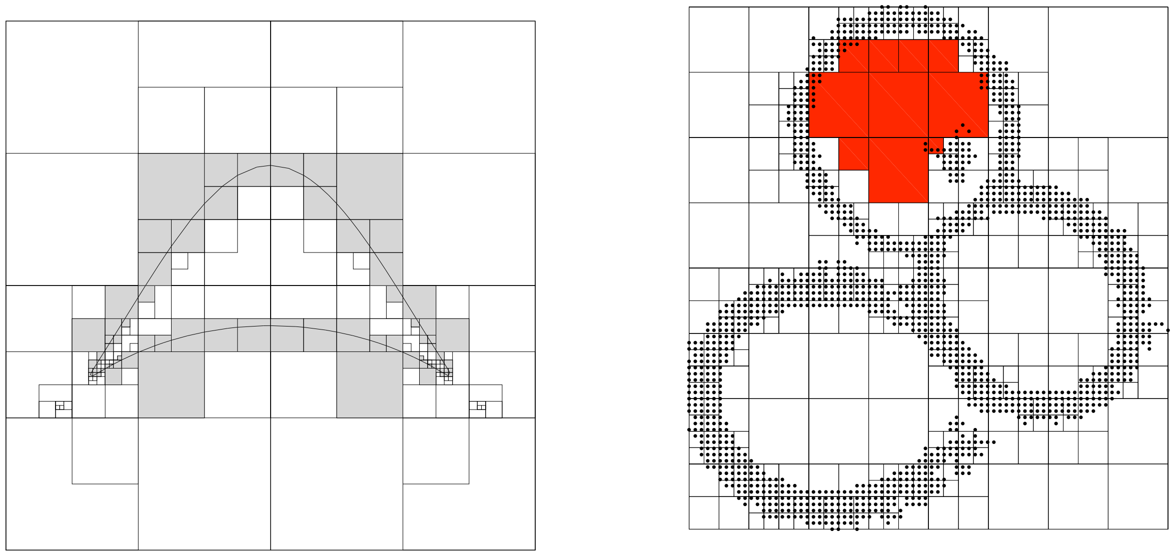
[8] E. Medeiros, L. Velho, H. Lopes. *Restricted BPA: Applying Ball-Pivoting on the Plane*, Proc. SIBGRAPI 2004, IEEE, 2004.

[9] L. H. Figuiredo, H. Lopes, J. B. Oliveira. *Robust adaptive approximation of implicit curves*, Proc. SIBGRAPI 2001, IEEE, 2001.



(a) (b) (c)

Figure 6: (a) four edge neighbors (b) four corner neghbors (c) full neighbors.



(a) (b)

Figure 7: (a) application to approximation of implicit curves enumerating its edges and (b) application to the Restricted BPA intthe local pivoting.