Optimizing an SPT-Tree for Information Visualization

Connor Gramazio[*]
Department of Computer Science
Brown University & Tufts University

Remco Chang[†]
Department of Computer Science
Tufts University

ABSTRACT

Despite the extensive work done in the scientific visualization community on the creation and optimization of spatial data structures, there has been little adaptation of these structures in visual analytics and information visualization. In this work we present how we modify a space-partioning time (SPT) tree – a structure normally used in direct-volume rendering – for geospatial-temporal visualizations. We also present optimization techniques to improve the traversal speed of our structure through locational codes and bitwise comparisons. Finally, we present the results of an experiment that quantitatively evaluates our modified SPT tree with and without our optimizations. Our results indicate that retrieval was nearly three times faster when using our optimizations, and are consistent across multiple trials. Our finding could have implications in using our modified SPT tree in large-scale geospatial temporal visual analytics software.

1 INTRODUCTION

In recent years the visual analytics community has made great advances in optimizing data storage for tabular data. Perhaps the most notable contribution is from Polaris[4], which helped introduce visual analytics to online analytical processing. Yet other types of popular data, like geospatial-temporal data, have received little attention. Furthermore, prior work in other fields is seldomly incorporated into visual analytics research, depriving the community of valuable resources. In this work we show how we modified a space-partioning time (SPT) tree[1] – a structure from the scientific visualization community used in direct-volume rendering – to match how geospatial-temporal data is used in visual analytics. An illustration of our structure is shown in Figure 1. The original SPT tree first traverses down a binary time tree where the root is the whole time span of the data and the leaves are individual time steps. At each node the tree stores spatial data in a complete octree associated with that time span where each octree leaf represents a voxel. We also discuss several optimizations we have made to traversing the structure to improve search speed. Unlike data used in direct-volume rendering, data in the visual analytics and information visualization communities often produce incomplete trees due to the distribution of data points. However, navigation through incomplete trees can become slow when using quadtrees and other spatial, hierarchal data structures, as it is not possible to perform simple jumps into a cell's memory location. Instead algorithms must traverse down the entire structure. Our optimizations mitigate the cost of the required traversal while still using an incomplete tree to conserve space.

Our work's immediate purpose is to make working with geospatial-temporal data more attractive for visualization researchers and developers by providing a hierarchal data structure that is both efficient and in accordance with how the visual analytics community uses data. We show how concepts from other fields of visualization can be effective in visual analytics software development by example. We also optimize around common interactions with spatial-temporal data to reduce retrieval time for frequently used tasks, though we have left user testing for future work.

[*]e-mail: cgrama01@cs.tufts.edu
[†]e-mail:remco@cs.tufts.edu

2 THE STRUCTURE

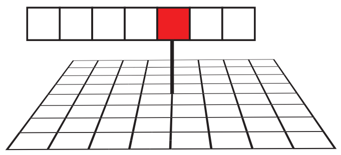

Figure 1: Our modified SPT tree

Given the extensive work in scientific visualization and graphics on optimizing hierarchal, spatial structures like quadtrees, we propose a hierarchal-based data model. But if we are to switch to a hierarchal model, a new schema for representing geospatial-temporal data must be considered. In existing relational storage methods it is most common to represent time as an extra dimension in a data cube. However, as shown by Shen et al.[3], treating time as a third dimension of a hierarchal structure can sharply decrease its resolution, which degrades a hierarchal structure's efficiency.

Despite decreased performance, thinking of time as a third dimension is often times more intuitive. To help developers, we looked for hierarchal structures in the scientific visualization community that maintained this unified abstraction in their interfaces, yet did not suffer a drop in performance by coupling them as described by Shen et al. in their implementation. After a survey of existing structures, the SPT tree was the structure that closest fit these two constraints.

2.1 Temporal and spatial substructures

The original SPT tree used a binary tree as its temporal indexing structure, with leafs representing single time steps and internal nodes representing increasingly large durations of time with the root spanning the whole time frame. Instead of using a time tree we use a hash map. It is common in geospatial-temporal software for users to scrub along a timeline, or index into specific points in time, and hash maps are ideal for this type of indexing. We felt it was more important in building a scalable data structure for fast individual time step indexing rather than adding support for quick access to time spans, which the user may never even use.

Our spatial substructure remains the same as the SPT tree. In our implementation we used a quadtree, however the structure supports any number of dimensions.

3 OPTIMIZATIONS

While the primary emphasis of our work is on optimizing retrieval speed, our structure does save space. Because time steps are likely to be unique, the hash map we use for temporal indexing can be shrunk in size, assuming an appropriate hash is used, as there should be few if any collisions. We also see savings in the spatial substructure because we do not construct complete trees like the original SPT tree.

We base our indexing optimizations on work done by Frisken and Perry[2]. Their work describes a way to efficiently traverse quadtrees and higher dimensional structures through bit comparisons. In this work we focus on searching for points, however Frisken and Perry also provide optimizations for region search and moving to adjacent nodes in the tree.

3.1 Frisken and Perry optimizations

The optimizations Frisken and Perry describe in their work rely on locational codes represented by bit strings. These bit strings are generated by bit shifting normalized values as shown in Listing 1. Every bit represents a branching decision for one level of the spatial tree. In a quadtree if both x and y locational codes are zero, then the algorithm will traverse to the top left child. If x is one and y is zero, then the algorithm will traverse to the top right child. The other two traversal decisions are made in similar fashion. These comparisons and branching decisions use almost exclusively bit comparisons, rather than using if statements or other flow control techniques, thereby lowering the constant for traversal.

Frisken and Perry's work was written in C, so we have made several changes to their approach in our implementation to better fit C++ idioms. Our traversal algorithm can be found in Listing 1. Note that nextNextLevel and the if/else statement can be safely eliminated, however the function will not be warning-free.

Listing 1: Optimized point search for a quadtree

```
vector<DataElt*> QuadNode ::
 findPoint (float x, float y) {
  QuadNode* cell = getSmallestNode(x, y);
  vector<DataElt*> vec;
  DataElt* data;
  for (int i = 0;  i < cell->getNumElts(); i++) {
      data = cell->getDataElt(i);
      if (data != NULL && data->getx() == x
          && data->gety() == y) {
           vec.push_back(data);
      }
  }
  return vec;
}
QuadNode* QuadNode :: getSmallestNode
 (float x, float y) {
  QuadNode* cell = this;
  int nextLevel = rootLevel - 1;
  unsigned int xLocCode = (unsigned int)
     (x * (1 << rootLevel));
  unsigned int yLocCode = (unsigned int)
     (y * (1 << rootLevel));
  while (cell->isLeaf() == false) {
    int nextNextLevel = nextLevel - 1;
    unsigned int childBranchBit = 1 << (nextLevel);
    unsigned int xChild =
      ((xLoc & childBranchBit) >> nextLevel);
    unsigned int yChild;
    if (nextNextLevel < 0) {
      yChild = (yLoc & childBranchBit) << 1;
    } else {
      yChild = ((yLoc & childBranchBit) >>nextNextLevel);
    }
    unsigned int childIndex = xChild+yChild;
    cell = cell->getChild(childIndex);
    nextLevel--;
  }
  return cell;
}
```

4 Methods and Results

To test the efficiency of our structure with and without optimizations we generated random sets of coordinates, populated trees, and then searched for a set point. All testing was performed on 15-inch Early 2008 MacBook Pro with a 2.4GHz Intel Core 2 Duo CPU and 4GB 667 MHz DDR2 SDRAM. We first tested sets of 100, 500, 1000, and 5000 elements with each set size undergoing 10,000 trials. We then tested our structure on sets of 200,000 elements with 1,000 trials. Because indexing through time is free in comparison to indexing through space, we only tested spatial retrieval. Our results can be found in Table 1 and in Figure 2. On average, in each size category, we achieved between a near tripling in performance. Through testing we also discovered that our retrieval using a 200,000 element tree using our optimized algorithm was faster than retrieving from a 100 element tree using the unoptimized traversal, suggesting that the optimizations provide better opportunities for scalability.

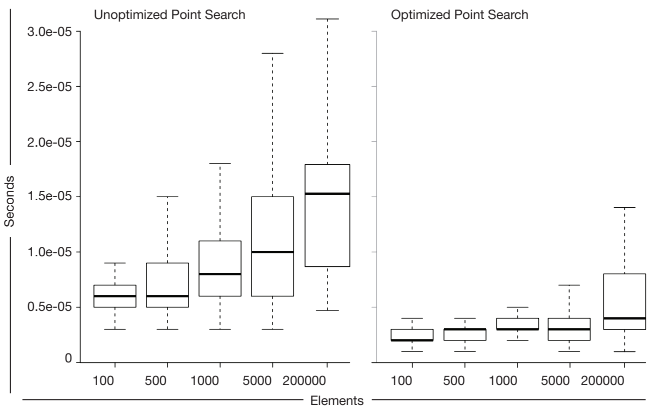

Figure 2: Boxplots of each set of trial results for point search.

Elements	Unopt. Avg.	Unopt. S.D.	Opt. Avg.	Opt. S.D.
100	6.795e-06	3.055e-06	2.608e-06	1.230e-06
500	7.537e-06	4.336e-06	2.827e-06	1.585e-06
1,000	9.264e-06	4.507e-06	3.421e-06	1.626e-06
5,000	1.108e-05	5.538e-06	3.994e-06	2.357e-06
200,000	1.493e-05	6.868e-06	5.094e-06	2.924e-06

Table 1: Search Times (Average and Standard Deviation in Seconds)

5 Conclusion

We have shown a set of adaptations to the SPT tree that made it appropriate to use with geospatial-temporal data and optimizations that caused an almost three times speed up for spatial traversal. We have also shown by example that it is possible to take a structure in a related field and adapt it to help visual analytics software accommodate more types of data. Immediately accessible future work involves further fine-tuning our optimizations and testing against relational databases in real applications.

References

[1] Zhiyan Du, Yi-Jen Chiang, Han-Wei Shen."Out-of-Core Volume Rendering for Time-Varying Fields Using a Space-Partitioning Time (SPT) Tree". In the proc. of IEEE's *Pacific Visualization Symposium*, 2009.

[2] Sarah F. Frisken, Ronald N. Perry. "Simple and Efficient Traversal Methods for Quadtrees and Octrees". In *The Journal of Graphics Tools*, 2002.

[3] Han-Wei Shen, Ling-Jen Chiang, Kwan-Liu Ma."A Fast Volume Rendering Algorithm for Time-Varying Fields Using a Time-Space Partitioning (TSP) Tree". In the proc. of IEEE's *Visualization*, 1999.

[4] Chris Stolte, Diane Tang, Pat Hanrahan. "Polaris: A System for Query, Analysis, and Visualization of Multidimensional Relational Databases". IEEE's *Transactions on Visualization and Computer Graphics*, Vol. 8, No. 1, January-March 2002.