Computing 3D Curvature through a Bucket PR Octree

Marcelo Velloso
Dept. of Computer Science
University of Maryland, College Park
mvelloso@umd.edu

April 11, 2012

Abstract

An empirical study has been conducted to determine the optimal bucket size for the bucket PR octree while calculating the vertex distortion of tetrahedral meshes. The motivation is to study the effects of the bucket PR octree data structure. This study implemented the data structure using the Java programming language and its libraries and found that the optimal bucket size was 32. This paper also discusses the extension of this project, the PR-star octree, and the work that was done on a recent paper accepted to the ACM SIGSPATIAL GIS 2011 conference.

1 Introduction

The curvature of shapes plays an important role in understanding the geometry and topology of surfaces. Vertex distortion is a generalization of the notion of concentrated curvature defined for triangulated surfaces and tetrahedral meshes embedded in 4D space and provides a powerful tool for understanding the local geometry and topology of 3-manifolds. Vertex distortion is defined in [2].

The main problem with the calculation of vertex distortion is that for very large tetrahedral meshes the amount of memory used to calculate it becomes an issue. The Java programming language and its library classes are used to test a solution to this problem using a bucket PR octree, the principal focus of this work, in which local topological connectivity of a tetrahedral mesh is obtained through its spatial locality. In contrast to previous topological data structures, which have focused on the adjacencies or incidences of mesh elements, the bucket PR octree is used as a spatial data structure on its embedding space to locally reconstruct the optimal application-dependent topological representation at runtime using the sorted geometry available from this spatial index. The innovative feature of this data structure is in computing topology through space. More specifically, vertex distortion is computed by generating the VT

(Vertex-Tetrahedron) topological relation locally with the amount of work to be done limited by the bucket size of each leaf node in the octree. This implementation allows for the VT relation to only be present upon the calculation of the vertex distortion for each leaf node and removed after the calculation has been completed.

The remainder of this paper is organized as follows: section 2 reviews background notions; section 3 discusses related work; section 4 introduces the bucket PR octree and discusses its properties; section 5 reviews the results of the empirical study; section 6 discusses the PR-star octree and how it compares to the bucket PR octree; and, finally, section 7 discusses the conclusions drawn from this empirical study.

2 Background Notions

The bucket PR octree combines notions from the PR octree, a spatial data structure over point datasets, with those of the indexed representation for tetrahedral meshes.

2.1 The PR Octree

The PR (point region) octree generalizes the PR quadtree defined in [3] to 3D. Each internal (non-leaf) node in an PR octree subdivides the space it represents into eight octants (cubic regions). The internal nodes always have eight children and leaf nodes are either empty or contain at most one vertex and its coordinate values [3]. An example of a PR quadtree is shown in figure 1.

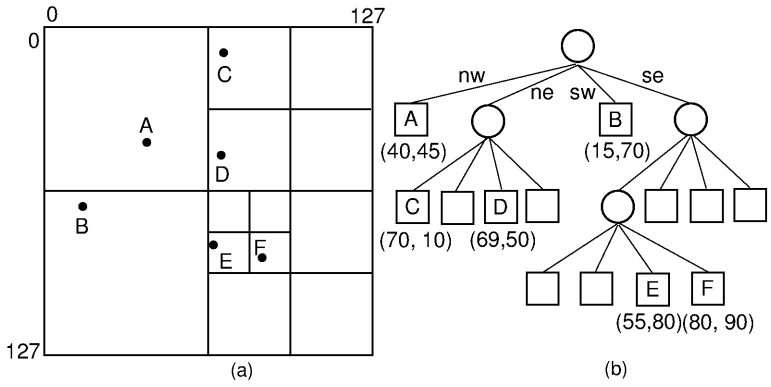

Figure 1: PR quadtree with six vertices

2.2 Indexed Tetrahedral Meshes

An indexed tetrahedral mesh is a common boundary-based data structure for a tetrahedral mesh. It encodes the Tetrahedron-Vertex (TV) relation, which

is the relation among a tetrahedron and its four vertices. It consists of two arrays: the vertices V, which encode the geometry of the mesh in terms of their coordinates in $\mathbb{R}^3$; and the tetrahedra T, which are encoded in terms of the indices in V of their four vertices [1].

3 Related Work

Hierarchical spatial indices for points in 3D euclidean space are provided by PR octrees and MX octrees. The MX (matrix) octree is organized in the same way as the region octree defined in [3]. What differentiates this data structure from the region octree is that the structure subdivides the 3D spatial region until each leaf node represents a $1 \times 1 \times 1$ space; these spaces are either empty or contain a data point and its coordinates [3].

Some data structures have been proposed for spatial indexing of polygonal maps, including graphs, planar triangle meshes and tetrahedral meshes [1]. The PM (polygonal map) quadtree family represents an attempt to overcome some problems associated with edge quadtrees defined in [3]. It is the quadtrees that are discussed in this section but this knowledge can be extended to 3D spatial regions as well in the form of PM octrees.

PM quadtrees are PR quadtrees that store vertices and edges. There are three variants to the PM quadtree that are discussed below [3]:

- The PM_1 quadtree has the following rules:
    1. At most, one vertex can lie in a region represented by a quadtree leaf node.
    2. If a quadtree leaf node's region contains a vertex, then it can contain no edge that does not include that vertex.
    3. If a quadtree leaf node's region contains no vertices, then it can contain, at most, one edge.
- The PM_2 quadtree has the following rules:
    1. At most, one vertex can lie in a region represented by a quadtree leaf node.
    2. If a quadtree leaf node's region contains a vertex, then it can contain no edge that does not include that vertex.
    3. If a quadtree leaf node's region contains no vertices, then it can contain only edges that meet at a common vertex exterior to the region.
- The PM_3 quadtree has the following rules:
    1. At most, one vertex can lie in a region represented by a quadtree leaf node.
    2. There are no restrictions as to how many edges can be kept in each leaf node.

4 Bucket PR Octree

This section introduces the bucket PR octree as a spatio-topological data structure for a tetrahedral mesh.

4.1 Formal Definition

The bucket PR octree combines an indexed tetrahedral mesh representation (section 2.2) with an augmented PR octree (section 2.1) that also indexes the set of tetrahedra in the VT relation of its indexed vertices. Thus, the bucket PR octree over a tetrahedral mesh is represented using three entities:

- An array of vertices V, encoding the geometry of the mesh.
- An array of tetrahedra T; each tetrahedron in T is encoded in terms of the indices of its four vertices within V (e.g. the TV relation).
- An augmented PR octree N, whose leaf nodes index the set of vertices within its domain and the set of all tetrahedra incident in these vertices.

4.2 Implementation Details

Four data structures are used to implement the bucket PR octree: a 2D global array of double (floating point) values for the vertices in the tetrahedral mesh, a 2D global array of integer values to represent the tetrahedra in the mesh, a 1D global array of boolean values to represent the vertices on the boundary of the tetrahedral mesh and the augmented PR octree.

The global array of vertices has the following information stored for each vertex: x, y and z coordinates and the scalar field value used in the vertex distortion calculation. The global array of tetrahedra has four integer values stored for each tetrahedron that represents the vertices making up each tetrahedron. Those integers are indices into the global array of vertices so that the vertices are only stored by one data structure. The global array of boolean values has an entry for each vertex (the vertex index) and it stores a true or false value if the vertex is on the boundary of the mesh or not.

The bucket PR octree is implemented as a separate class file from the previous three data structures. This was done to distinguish the global data structures from the local computations. The class file contains a private class for the variables and methods of the octree node within the more general octree class. Each ocnode, as the private class is called, has double values to represent the minimum and maximum x, y and z coordinates of the region it represents. Each ocnode also has an array of indices into the global vertex array where the maximum length is the bucket size of each node, an array of indices into the global tetrahedra data structure and a potential array of eight ocnode children if the ocnode is not a leaf node. If the ocnode is not a leaf node then it will only have an array of eight ocnode children, the other variables will be null. The array of indices into the tetrahedra array represents the set of tetrahedra that

are incident to the vertices represented by the array of vertex indices in each ocnode. Finally, each ocnode also has a hash map in which the keys of the hash map are vertex indices and the values of the hash map are lists of tetrahedron indices incident to that vertex index. In other words, each hash map represents the VT (Vertex-Tetrahedra) relation and is only constructed locally when the vertex distortion is computed. This design decision was made to save memory so that the VT relation would not linger in memory unless it was needed since it is not too expensive in terms of time to compute the VT relation when the vertex distortion calculation needs to be done for a particular ocnode.

5 Experimental Results

Experiments were run on the following tetrahedral meshes: Cube1 (27 vertices and 40 tetrahedra), Cube2 (24 vertices and 30 tetrahedra), SuperPhoenix (2,896 vertices and 12,936 tetrahedra), Fighter (13,832 vertices and 70,125 tetrahedra), BuckyballSmall (35,937 vertices and 163,840 tetrahedra), BluntFin (40,948 vertices and 222,414 tetrahedra), F117 (48,518 vertices and 240,122 tetrahedra), Torso (168,930 vertices and 1,082,723 tetrahedra), SF2_C (378,747 vertices and 2,067,739 tetrahedra), Bucky2 (262,144 vertices and 1,250,235 tetrahedra), Plasma (274,625 vertices and 1,310,720 tetrahedra) and Post (108,300 vertices and 616,050 tetrahedra). Graphing all the results would make the graphs difficult to interpret so a set of four meshes representative of the entire space covered by the experiments was used: Cube1, BluntFin, Torso and SF2_C. A note of interest is that the smallest bucket size (number of vertices stored in each ocnode) the graphs show results for is 2 because the BluntFin tetrahedral mesh has two vertices that are identical so infinite recursion is unavoidable for a bucket size of 1.

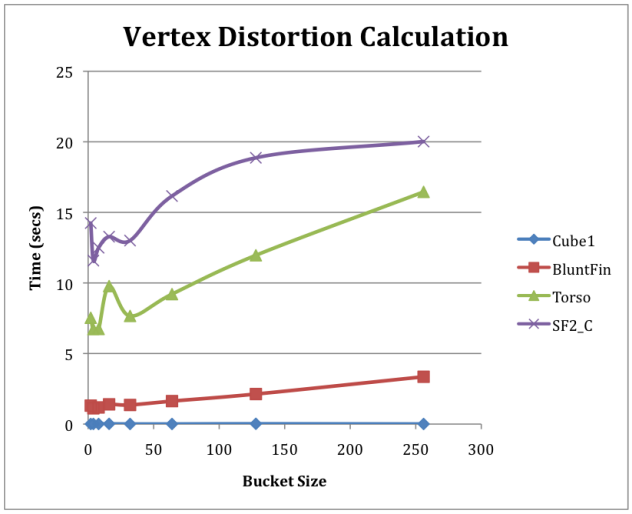

Figure 2: The time elapsed while calculating vertex distortion for each mesh

The graph in figure 2 shows a general trend that the greater the bucket size the longer it takes to compute the vertex distortion using the bucket PR octree data structure. However, it also shows a trend that if the bucket size is too small for meshes that are very big the performance is not good. The above graph shows the optimal bucket size to be 4 to minimize the time it takes to compute vertex distortion for every vertex in the mesh. Bucket sizes of 8 and 32 also seem to perform well.

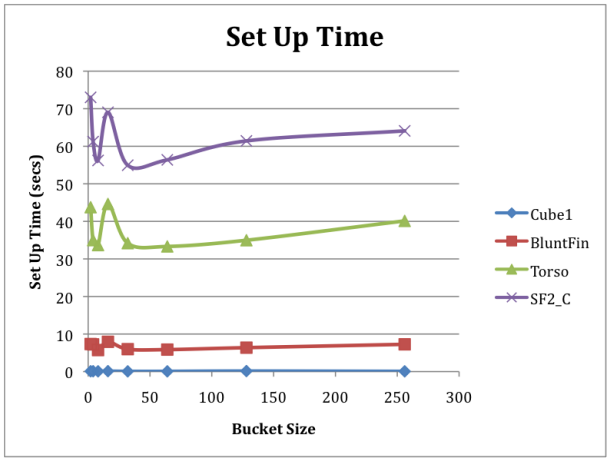

Figure 3: The time elapsed while setting up the bucket PR octree for each mesh

The results in figure 3 show a similar result to the vertex distortion time

results. If the bucket size is too small (smaller than 4) it takes longer to set up the bucket PR octree data structure because of the amount of splits and recursive calls that get made due to the depth of the tree. One interesting oddity in this graph is the bad performance of bucket size 16. The performance seems to be a bit of an outlier and theoretically it does not make sense why that would be the case. The optimal bucket size for set up time is either 8 or 32 with a bucket size of 4 also performing well.

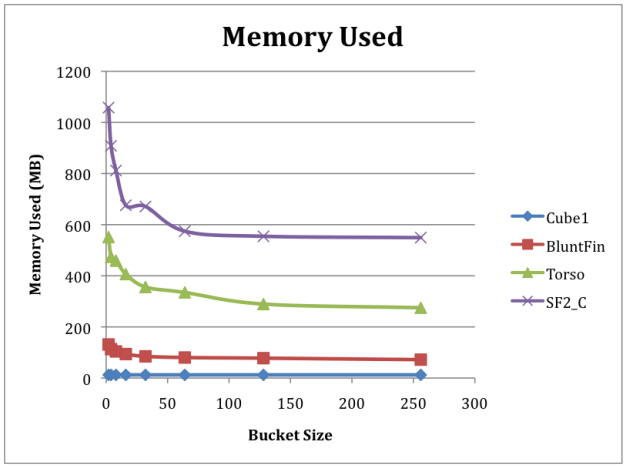

Figure 4: Memory used to set up the bucket PR octree for each mesh

The results in figure 4 have a clear downward trend where the smaller the bucket size is the more memory is used to store the bucket PR octree. This is because more nodes need to be created and maintained for smaller bucket sizes. The downward trend seems to level out at a bucket size of 32 or 64 so these two values seem to be the optimal sizes for memory efficiency; for bucket sizes greater than 64 there are diminishing returns in memory efficiency.

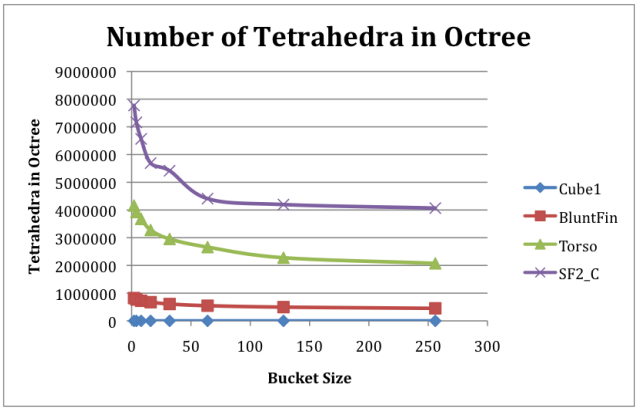

Figure 5: Number of tetrahedra in bucket PR octree for each mesh

The graph in figure 5 shows that when the bucket size increases there are less tetrahedra stored in the bucket PR octree. Therefore, the optimal bucket size is 256 but there seems to be diminishing returns after a bucket size of 64. The principal reason for this is that when the bucket size increases there are less leaf nodes for the incident tetrahedra to spread out among.

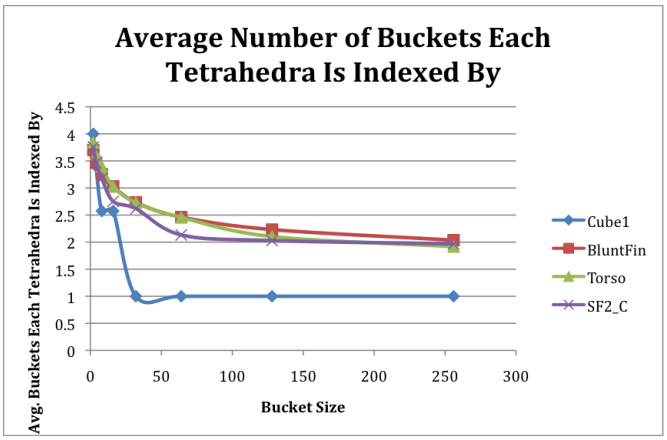

Figure 6: Average number of buckets each tetrahedra is indexed by for each mesh

The graph in figure 6 shows the average number of buckets each tetrahedron is indexed by. If all four vertices of the tetrahedron appear in different nodes of the octree then four buckets index that tetrahedron. The results above show that when the bucket size is really small the average numbers approach 4, the worst case and when the bucket size is large the average numbers approach 1, the best case. It is important to note that, save for the Cube1 mesh, the other

meshes seem to approach 2 as the bucket size increases but if greater bucket sizes were tested this number would theoretically continue to approach 1 and not 2. The optimal bucket size is 256 but after a bucket size of 64 there are diminishing returns to increasing the bucket size.

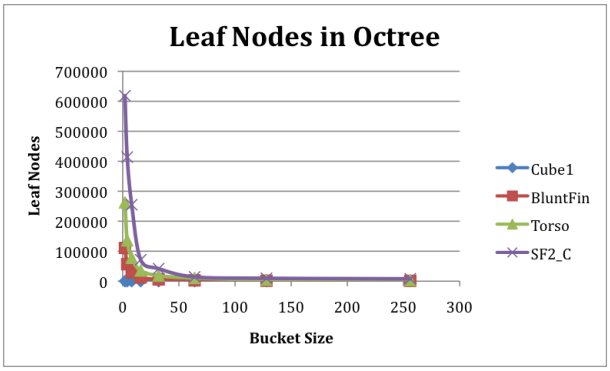

Figure 7: Amount of leaf nodes in the bucket PR octree for each mesh

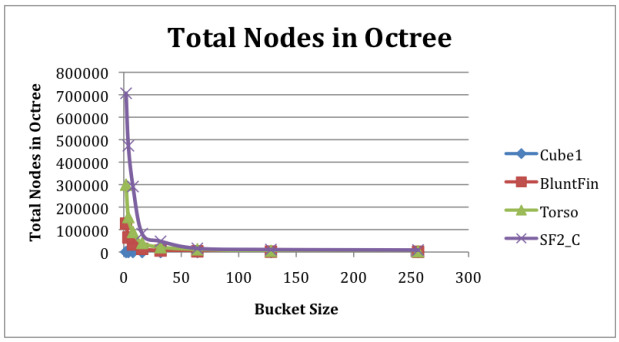

Figure 8: Amount of total nodes in the bucket PR octree for each mesh

The final set of results, shown in figure 7 and figure 8, involves the amount of leaf nodes and total nodes in the bucket PR octree. As the two graphs above show, with a smaller bucket size more leaf nodes and total nodes are created in the data structure. By the same logic, with a greater bucket size less leaf nodes and total nodes are created in the data structure. The optimal bucket size is 256 but at a size of 32 there are diminishing returns to increases in bucket size.

6 PR-Star Octree

The PR-star octree is an extension of the work done in this paper. It is described in more detail in a paper accepted to the ACM SIGSPATIAL GIS 2011

conference [1].

The PR-star octree keeps some aspects of the bucket PR octree but also makes some changes to the data structure. The similarities between the two data structures are listed below:

- An array of vertices V, encoding the geometry of the tetrahedral mesh.
- An array of tetrahedra T, where each tetrahedron is encoded by the indices of its four vertices within V.
- An augmented PR octree N, whose leaf nodes index the set of vertices within its domain and the set of all tetrahedra incident in these vertices.

There are some differences between the two data structures. They are listed below:

- The PR-star octree includes an additional step to reindex the array of vertices V and the array of tetrahedra T in order to exploit the spatial locality of the vertices once the octree has been properly set up.
- The bucket PR octree has a 1D array of boolean values that encodes which vertices are on the boundary of the tetrahedral mesh.
- The PR-star octree stores the range of vertex indices represented by each node while the bucket PR octree stores the range of x, y and z coordinates represented by each node.

7 Concluding Remarks

A summary of the experimental results found in the previous section shows that for vertex calculation time and set up time the optimal bucket sizes are 4, 8, or 32; for the memory used the optimal bucket sizes are 32, 64, 128 and 256; for the number of tetrahedra in the octree and average number of buckets each tetrahedra is indexed by the optimal bucket sizes are 64, 128 and 256; and for the leaf nodes and total nodes in the octree the optimal bucket sizes are 32, 64, 128 and 256. The best performing bucket size throughout all these tests seems to be 32. A bucket size of 32 achieves just the right balance of memory efficiency and vertex distortion calculation time.

The PR-star octree, as well as the bucket PR octree, was designed to be more memory efficient than the standard PR octree. The observation here is that many queries in typical GIS applications have spatial locality and the two data structures above exploit this spatial locality by only having topoplogical relations computed when they are needed at runtime. This amortizes the cost of constructing these relations over multiple accesses while processing each node. The PR-star octree paper demonstrates the advantages of the PR-star octree representation in several typical GIS applications, including the detection of boundaries, computation of local curvature estimates and mesh simplification [1].

Appendix - File Organization

This section discusses the organization of files in the project and briefly goes over the source code. All the source code appears in the src folder of the Eclipse project folder. There are four packages associated with the source code: data_structures, exceptions, main and test. The data_structures package contains the code used to implement the bucket PR octree. The exceptions package contains code for an exception that gets thrown when a vertex is out of bounds. The main package contains the code that stores the global vertex, tetrahedra and boolean arrays. Finally, the test package contains code that runs a series of Junit tests to test the project. TetrahedralMeshAnalysis.java is a test file that generates a series of .txt files in the analysis_results folder. The .txt files are file names according to the bucket size and run the full series of tests shown in the graphs above for all the tetrahedral meshes in the project folder. TetrahedralMeshTest.java is a test file that tests some methods in TetrahedralMesh.java such as the method for rounding the min and max values of the mesh to the nearest power of 2. Finally, VertexDistortionTest.java is a test file that tests the correct function of the vertex distortion calculation. The sample_vertex_distortions folder has the correct output of vertex distortion for some meshes. The distortion_results folder is where VertexDistortionTest.java stores its vertex calculation results. The graphs folder is where all the excel files of the graphs used in the paper are stored. All the tetrahedral meshes used in this project are stored in the main project folder as files that end with the suffix ts. To run the Junit tests using Eclipse simply open any of the source files in the test package and right-click then select to run as a Junit test.

Acknowledgements

A very big thanks to Kenny Weiss for all his advice over email, for the help with debugging questions, for the sample code provided with the vertex distortion calculation algorithm and for the tetrahedral meshes provided. A big thanks to professor Leila De Floriani for all her advice over email and for her feedback to make this paper a success.

References

[1] Kenneth Weiss, Riccardo Fellegara, Leila De Floriani and Marcelo Velloso. The PR-Star Octree: A Spatio-Topological Data Structure for Tetrahedral Meshes. In *Proceedings of the ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, Chicago, IL, November 2011. Association for Computing Machinery.

[2] Mohammed Mostefa Mesmoudi, Leila De Floriani and Umberto Port. Discrete Distortion in Triangulated 3-Manifolds. *Eurographics Symposium on Geometry Processing*, 27(5):1333–1340, 2008.

[3] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann Publishers, 2006.