Implementing a Node Based Split-Tree Search Algorithm

Vincent C. Betro[1] and Steve L. Karman, Jr.[2]

[1] University of Tennessee at Chattanooga SimCenter, 701 E. MLK Blvd., Chattanooga, TN, 37403 Vincent-Betro@utc.edu
[2] University of Tennessee at Chattanooga SimCenter, 701 E. MLK Blvd., Chattanooga, TN, 37403 Steve-Karman@utc.edu

1 Introduction to Cartesian Hierarchical Meshing

Cartesian hierarchical meshes are becoming a central focus of grid generation research since they are rapid to generate [1], they can refine in pertinent areas to a desired resolution [2], and they provide an excellent basis for mesh adaptation [3]. More importantly, having a Cartesian hierarchical mesh allows for the natural creation of a tree, often an Omni-tree or an Octree, which aids in searching for elements as well as only needing to remesh certain branches for dynamic meshing.

Generating a Cartesian hierarchical mesh requires the creation of a root cell around the geometry to be meshed. Then, through recursive refinement, the root cell volume is discretized by creating successively smaller volumes, often utilizing some method of preserving spacing, such as Riemannian Metric Tensors [4]. Finally, cells outside of the computational domain are turned off, and a body conforming volume mesh, overset mesh, or immersed boundary formulated mesh is created.

The basic building block for a Cartesian hierarchical mesh is a voxel [5]. Each voxel contains the index of its mother voxel, and the root voxel is the only voxel without an initialized mother; it also contains a list of its children, if they exist. A split variable may also be given in the structure (except in the case of an Octree, since refinement is isotropic), wherein the direction of refinement is stored for ease of tree traversal. Also, the physical coordinates of the high and low corner points of the voxel are stored for construction of the physical points after the mesh is generated and for relative tree traversal purposes [6] .

2 Utilizing the Split-Tree

Many Cartesian hierarchical mesh generation schemes use the Octree, which makes it simple to keep track of neighbors and children since there is only one type of refinement allowed (isotropic, seen in row three of Figure 1). However, this limits the user to unit aspect ratio meshes, which create very high node and element counts in the case of meshes with very small spacing needed to resolve the flow. Another option is to use the Omni-tree technique, which allows in a single step each type of refinement seen in Figure 1. While this is very versatile, it requires multiple tests in order to appropriately traverse the tree and it does not preserve tree integrity, which means that the same number of refinements on any given root cell will yield cells of the same volume, since each voxel may be refined in one, two, or three directions in one step. This can be avoided by not adopting children of children while refining the mesh; however, not refining each voxel to the fullest extent possible requires that each voxel be visited at each step of the refinement process, even if it has already been refined.

In Figure 1, the first row shows one directional refinement, which is used to create the split-tree. While this allows for refinement in only one direction, isotropic refinement can still be achieved, just over multiple steps. This method preserves tree integrity and makes tree-traversal and neighbor searching rapid due to the fact that only one direction needs tested to determine node placement and which branch to search based on the cut parameter of the voxel. Since voxels can only be refined in one direction, there is no need to adopt grandchildren and reshuffle parentage as voxels are refined differently by different tensor specifications.

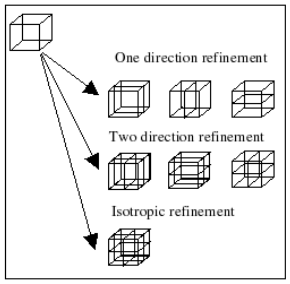

Fig. 1. Subdivision Refinement of a Cartesian Cell in Three Dimensions – Refinement options in three dimensions. The choice used to create the split-tree is one-directional refinement.

3 Split-Tree Neighbor Search Algorithm

In order to effectively apply quality constraints and create distinct nodes, neighbors must be determined. In order to save memory, deal with a constantly changing list of elements, and best utilize the tree, a highly efficient neighbor algorithm was developed. In many applications, an array which stores each of six face neighbors is constructed, which not only is costly in terms of memory but also in terms of computational time spent changing the hash table as the mesh is generated and needing to remap elements. Instead, the choice was made to do a point search tree traversal, thus assuring that the most updated information is available at any given time, there is no need for remapping, and no memory is used for a hash table.

As seen in Figure 2, a point is projected in the direction of the desired neighbor, and then a tree traversal determines in which voxel the point lies. While most neighbor constructs only consider the six face neighbors, seen in the first row of Figure 3, the method devised here is able to find both caddy-corner neighbors and vertex neighbors, as seen in the second row of Figure 3.

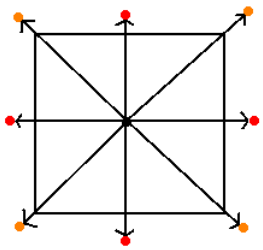

Fig. 2. Neighbor Searching – Neighbor search directions and points for a face of a given voxel.

The algorithm recursively searches the tree by passing a coordinate down child branches until the finest level in which it is contained is reached and then returning the voxel found back up the stack. The coordinate (x, y, or z) that is in the direction of the split of the voxel is tested for containment in the correct child branch by a simple greater than or less than test. Since there is a user defined minimum spacing for refinement, the tolerance applied to this floating point calculation is 1/4th of the minimum spacing, and by simply adding it or subtracting it (depending on which branch is being tested) from the corresponding x, y, or z coordinate of the centroid, one is guaranteed to be in the voxel that actually contains the point. In some situations, this leads to a voxel with children being identified as the neighbor (specifically when a face centroid or mid-edge node is used to search) and other times it returns a

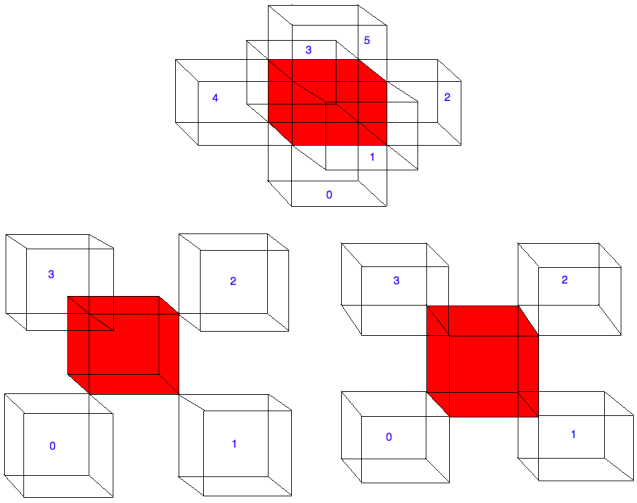

Fig. 3. Types of Neighbors – The red voxel in the center of each of the above diagrams is the voxel which has neighbors being determined. The first row shows the six face neighbors that are normally stored. The second row, left, shows four of the twelve caddy-corner neighbors that exist. The second row, right, has four of the eight vertex neighbors that exist displayed.

voxel without children (always in the case of the corner nodes being used to search).

This algorithm is versatile and will find neighbors despite differing sizes and levels of refinement since it is simply looking for containment. In cases like that seen in Figure 4, one may find a face neighbor with children but actually need to know which voxel at the finest level contains the given point and the orientation of the children. In this case, the algorithm will create four nodes, denoted in red in Figure 4, and filter the results of these searches to discern how many children the neighbor has and in what positions. The nodes are created to be 1/4th of the minimum spacing into the neighbor and 1/4th of the minimum spacing in the given cardinal direction away from the face center node. This results in a vector that is $\frac{\sqrt{2}}{4}$ times the minimum spacing, which is larger than the 1/4th that the neighbor routine uses as a tolerance, thus assuring that we get into the right voxel and get the appropriate neighbor.

4 Conclusions

Having a robust, recursive voxel search algorithm allows for hierarchical Cartesian mesh generation without the need to keep up with neighbor hash tables or use a simple Octree to make neighbor creation automatic. Having a split-tree

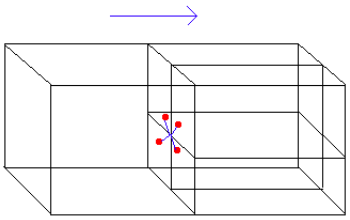

Fig. 4. Finding Children of Neighbors – The algorithm creates four nodes, denoted in red in the above figure, and filters the results of these searches to discern how many children the neighbor has and in what positions.

further speeds the process due to the fact that only one direction needs to be checked to proceed down to the next branch, and having a constant tolerance value due to the lack of general cutting assures correct determination of the voxel in which the point is contained. This algorithm allows the user to find not only face neighbors but caddy-corner and vertex neighbors in a simple fashion, making it very general. Finally, the ability to discern the orientation of neighbor children assures that even if a voxel is found that is not at the finest level, regardless of how many more refinements have been performed on the voxel, the proper children will be fetched.

References

1. Berger, M.J., and Aftosmis, M.J. (1998) Aspects (and Aspect Ratios) of Cartesian Mesh Methods. In: *Proceedings of the 16th International Conference on Numerical Methods in Fluid Dynamics*. Springer, Berlin Heidelberg New York
2. Dawes, W.N. (2006) Towards a fully parallel integrated geometry kernel, mesh generator, flow solver, and post-processor. At: *44th Aerospace Sciences Meeting and Exhibit (January 9-12, 2006)*. AIAA, Reno, NV
3. Lahur, P.R. and Nakamura, Y. (2001) Anisotropic Grid Adaptation. In: *The Japan Society for Aeronautical and Space Sciences Journal, Volume 44, Number 143*, pp. 31-39. JSASS, Tokyo
4. Karman, Jr., S.L., Anderson, W.K., and Sahasrabudhe, M. (2007) Mesh Generation Using Unstructured Computational Meshes and Elliptic Partial Differential Equation Smoothing (2007-0559). *AIAA American Institute of Aeronautics and Astronautics Journal*, 44,6: 1277–1286.
5. Voxel. (2010) Retrieved May 25, 2010, from http://en.wikipedia.org/wiki/Voxel
6. Betro, Vincent C. (2010) Fully Anisotropic Split-Tree Adaptive Refinement Grid Generation with Tetrahedral Mesh Stitching. PhD Dissertation, University of Tennessee at Chattanooga, Chattanooga, TN