A Survey of Octree Volume Rendering Methods

Aaron Knoll
Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, Utah
knolla@sci.utah.edu

Abstract: Octrees are attractive data structures for rendering of volumes, as they provide simultaneously uniform and hierarchical data encapsulation. They have been successfully applied to compression, simplification, and extraction as well as rendering itself. This paper surveys and compares existing works employing octrees for volume rendering. It focuses specifically on extraction, direct volume rendering, and isosurface ray tracing.

1 Introduction

An octree is a hierarchical binary decomposition of 3-space along its component axes. A conventional octree guarantees regular, non-overlapping node spacing, thus is well-suited as a container for rectilinear scalar field data. How this hierarchical container is used, however, varies with application. For general purposes, octrees allow data to be stored with adaptive levels of resolution, and accessed quickly via a binary hash function. In volume rendering, the octree often mutates to suit the individual technique, commonly delivering occlusion, acceleration, adaptive multiresolution, compression, or a combination of several features. We will examine the role of the octree structure in three different volume rendering systems: extraction and rasterization; direct volume rendering; and interactive ray tracing. As background, we survey several varieties of octree and efficient hashing schemes for their traversal.

2 Octree Structures and Hash Schemes

The first actual octree publication is unclear [Sam90]. Among the first credited works is that of Morton [Mor66] using quadtrees for geographic indexing scheme. The term "quad-tree" was first used by Finkel and Bentley [FB74] for non-uniform point quadtrees. The first definitive octree is credited to Jackins and Tanimoto [JT80], adapting the earlier quadtree work of Hunter and Steiglitz [HS79] to three dimensions.

The principle of recursive eight-fold subdivision of space is common to all octrees. However, several varieties exist depending on the desired application. Different methods exist

to store the structure in memory and provide pointers from children to parents. Implementation of the structure in turn affects the behavior of data access. In general, data is retrieved from an octree via a one-way hash function: given a point in octree space, it seeks the leaf node containing that point. This process is often referred to as *point location*. A related problem is *neighbor finding*: given a node deep in the tree, finding adjacent or nearby nodes. Often, hashing is extended such that a hash code can refer to a region of nodes, or coordinates of one node relative to another.

2.1 Pointer Octree and Hashing

In a traditional octree, a parent node stores pointers to all children. A leaf node could be indicated by a eight null pointers, but most commonly it is given a separate pointerless leaf structure, or represented implicitly within the parent node. While the pointer octree carries a higher storage footprint than pointerless varieties, it is generally simpler to hash as each pointer guarantees the path to the desired child octant. In addition, it is possible to exploit memory tricks so that a node in a pointer octree does not require a full 32-bit (or 64-bit) pointer to each of its children.

Point Location Numerous options exist for performing point location on an octree. The simplest consists of applying comparisons at each parent to determine which child octant the point lies in, until this process reaches a leaf. Comparison operators are expensive, however. Another common scheme is to store a horizontal array containing 0-7 octant indices at each element, in order of increasing depth. Thus, the traversal path is given explicitly in this locational array. The locational array can be built from simple integer coordinates by a process called interleaving proposed by Morton [Mor66]. Array-based keys were popular in early implementations of octrees [Gar82, HS79, JT80].

More recently, Frisken and Perry [FP02] observed that octree and quadtree locational codes could be simply represented by a vector of integer coordinates. Specifically, given a maximum octree depth d_{max} and $x, y, z \in ([0, 2^{d_{max}}] \cap N)^3$, one can hash directly to the correct child node until a leaf is reached. This is accomplished by a bitwise & of each x, y, z component with a 1-bit mask corresponding to the current depth of the hash function. As a result, point location can be performed by simply casting a vector to integer and hashing to the desired node. It is important to note that while Frisken and Perry did not invent the locational code, they were the first to recommend composing the code directly from integer coordinates during hash operation.

Neighbor Finding The problem of finding adjacent neighbors was addressed early on in octree research. Gargantini [Gar82] provides a concise algorithm for finding adjacent nodes in the linear quadtree, using Morton's [Mor66] array-based locational code. Samet [Sam82] proposed a more general scheme for adjacent neighbor-finding, using only an original node and a desired direction, and no knowledge of a locational code. The algorithm was later applied to accelerating ray tracing using a pointer octree [Sam89].

Samet [Sam90] conducts rigorous analysis of the neighbor finding technique, decomposing the algorithm into cases for edge, vertex and aligned neighbors. He concludes that the average cost to find an adjacent neighbor at the same depth is $O(1)$.

While neighbor-finding is demonstratably less complex than point location from the root, Samet's algorithms for pointer octrees without locational codes consist of case decompositions that could be highly optimized. Frisken and Perry noted that the "reflection" function for adjacent neighbors, implemented by Samet via a switch statement, can be accomplished by a binary exclusive "or" of the current location and the direction in which we seek a neighbor. [FP02]. Then, the neighbor itself can be retrieved by recursing up to the deepest common ancestor, and performing point location to find the target neighbor node.

Region Finding Another common goal in octree and quadtree hashing is a mechanism for recovering all nodes in a certain region; or more simply determining if a given node is within a region. With locational code schemes [Mor66, Gar82, FP02], a region can be described by coverage of implicit parent nodes [Gar82]. This is done by marking lower bits in the locational code with an appropriately high number, indicating that all children form part of that region. Alternately, with a coordinate system similar to that of Frisken and Perry [FP02], regions can more intuitively be described by a minimum and maximum pair of vectors.

Region finding has proven useful in image processing applications; thus far its application to volume rendering on octrees has been limited. However, it could conceivably be used when individual sections of an octree volume require attention, particularly when the region is not perfectly aligned with a parent block in the octree.

2.2 Octree Varieties

Other breeds of octree exist beside conventional pointer octrees, and are suited to various applications. Full descriptions of most octree varieties are given by Samet [Sam90]. For our purposes, we will only examine octree structures that are of interest in volume rendering.

Full Octree In a full octree, it is possible to compute the address of any node based on its location in the tree, and thus pointers are unnecessary. Clearly, however, a full octree is undesirable in most cases where data could be compressed or consolidated; the added space obviates most gains from not storing pointers. A common application of a full octree would be one where the encapsulated data is non-homogenous and universally significant, and where we make use of a coarser-resolution representation in interior nodes of the tree.

Linear Octree The linear octree is a variety of pointerless octree in which only leaf nodes are stored, and allocated contiguously in memory. This method was originally pro-

posed by Gargantini for quadtrees [Gar82]. Linear octrees make use of an interleaved base-8 code similar to that proposed by Morton for traditional octrees [Mor66]. The important difference lies in storage: rather than connect the leaf nodes via interior nodes, the leaf nodes are sorted by locational code and then laid out sequentially in memory. Then, rather than matching the Morton code segment with the correct child at each depth of the octree, point location consists of a binary search on the sorted array of leaves. This binary search performs node lookup in $O(log_2(L))$ in a tree with L leaf nodes; as opposed to $O(log_8(N))$ complexity for a pointer octree with N total nodes. Except in the case of highly vertical trees with few leaf nodes relative to interior nodes, the linear octree is generally slower to hash. The major advantage is that it requires storage of neither pointers nor interior nodes. In applications where storage is of the utmost importance, and we only care about leaf nodes, the linear octree is an attractive structure.

Thus far, no one has directly applied the linear octree to volume rendering. However, it is an intriguing structure in it potential compression abilities. As compression is one of the main goals of adaptive octree methods on volume data, it is worth mentioning this structure.

Branch-on-need Octree Wilhelms and Van Gelder [WV92] proposee an incomplete pointer octree that subdivides space non-uniformly, the goal being to build an octree with the fewest-possible empty subtrees within interior nodes. In this way, the ratio of nodes to data points (scalars) is minimized, saving space. As it subdivides space non-uniformly, the BONO may require multiple parent nodes at deeper regions of the tree where a traditional octree would only require one. This potentially causes added traversal steps. In addition, the non-uniform nature of subdivision precludes the use of a pure coordinate system (e.g. Frisken and Perry [FP02]) as a direct hashing scheme.

3 Octrees in Volume Rendering

3.1 Extraction

Early interactive volume rendering commonly employed isosurface extraction via marching cubes [LC87] or a similar variant; paired with z-buffer rasterization of the resulting mesh. Before the widespread availability of GPU's, the goal was to use the octree to simplify the extraction process. In general, the purpose of the octree was to provide a structure with a small memory footprint that could encapsulate cells of a volume, and from which a mesh could be extracted.

Wilhelms and Van Gelder Perhaps the first application of an octree in isosurface extraction was by Wilhelms and Van Gelder [WV92]. Their system employed the aforementioned branch-on-need octree (BONO) to minimize the number of nodes stored relative to cells in the extraction phase. The main downside of this choice, sacrifice in speed of cell location, was a non-issue: the algorithm was bound by extraction, not octree structure traversal.

The Wilhelms and Van Gelder system is best known as the first application of a min/max tree for acceleration. While not explicity mentioned in their paper, they make use of a dual relationship between voxels of the volume and cells from which surface points are extracted. Specifically, a cell is composed of eight voxels, and always indexed by a voxel at a single corner. In the octree, voxels are grouped into eights at the maximum depth level. The min/max tree, then, is built by examining 27 voxels: the original voxels, and their neighbors that constitute the far boundaries of the cells.

During extraction, the marching algorithm need only examine octree nodes when the desired isovalue falls in between the minimum and maximum pair. These nodes can be quickly identified by traversing the octree from top-down as a min/max tree.

Livnat and Hansen Wilhelms and Van Gelder only effectively used their octree to accelerate the extraction process. However, in real-time adaptive surface reconstruction, the same octree structure can be used to order visibility based on the user-specified camera position. This was exploited by Livnat and Hansen. [LH98]. In addition to using the octree to prune empty subtrees (hence, nodes containing no surface), Livnat and Hansen perform visibility culling on nodes that are occluded by previously-traversed nodes closer to the camera. Thus, the octree is used not only to speed extraction, but rasterization as well.

Westermann et al. While the technique of Wilhlems and Van Gelder allowed large regions of 3D scalar data to be systematically ignored, their extraction technique essentially processed leaves at the deepest level of the octree. Westermann et al. [WKE99] recognized that speedups could be gained by exploiting the multiresolution nature of the octree structure; namely by adaptively deciding to extract the mesh from fewer, coarser-resolution nodes corresponding to the same region. This permits higher-resolution volumes to be processed at real-time rates with appropriate level of detail.

The main problem with adaptive surface reconstruction is that cell corner values may vary at junctions in the octree. Such junctions occur when a finer-resolution group of cells is adjacent to a coarser cell. When surfaces are extracted from adjacent cells of different resolution, they are not connected at junctions and therefore cracks appear. A major insight in Westermann et al.'s work is how to patch the resulting meshes to guarantee continuity.

For adaptive multiresolution, the authors built upon previous view-dependent extraction methods (e.g. Livnat et al. [LH98]). They proposed a single pass of the octree, building an "oracle" structure that determines the level of detail, hence the depth of the octree to be traversed, based on view-dependent and geometric criterion. Specifically, they implement one "oracle" for predicting maximum surface curvature within each leaf node. Regions of low curvature are represented with fewer polygons, hence allow shallower traversal of the multiresolution octree. Regions of high curvature merit traversal to the finest level available. In addition to curvature, the octree traversal depth is determined by a "focus point" oracle based on distance to the camera view position.

Velasco and Torres Thus far, extraction-based techniques employing octrees used the structure for indexing existing volume data, and accelerating extraction or rasterization.

The advantage to this technique is that cells can be reconstructed from the original grid data without no costly neighbor-finding [WV92]. The disadvantage is that the full original 3D scalar array must be stored in memory, in addition to a separate octree structure.

Velasco and Torres [VT01] propose a format called a *Cells Octree*, which enumerates several types of distinct octree nodes. Internal nodes contain pointers to eight children. "Single-cell leaf" nodes consist of a single cell, bordered by eight scalar values. Finally, "eight-cell" leaf nodes contain eight cells at the maximum depth of the octree, rolled into their parent node. The cells are represented entirely at every level of the octree, and no neighbor-finding is required to reconstruct cells from voxels. Unfortunately, this technique falls short of the compression achievable by storing only single-scalar voxels within an octree. Moreover, the largest volume tested has dimensions 128^3; which is sufficiently small that no compression would be necessary to store even a full octree, in addition to the original uncompressed volume, at the time of its publication.

Worse still, Velasco and Torres propose modifying the original scalar data to smooth values across cells, and thus guarantee continuous surfaces. This approach differs markedly from that of Westermann et al. [WKE99]; from a visualization standpoint it is arguably wrong as the source data is being modified to create a smooth surface. However, one could equally argue that any piecewise linear mesh is itself an inexact representation of an interpolating isosurface, no matter how well refined. In practice, the results of Velasco and Torres appear acceptable, and theirs is the first published application of an octree towards visualization of compressed structured data.

3.2 Direct Volume Rendering

An alternative to rendering a mesh is direct volume rendering (e.g. Levoy [Lev90]), which integrates rays intersecting a volume. While this is slow in ray tracing, it is effective on current GPUs by accumulating gradients across sequential cutting planes of the volume stored as 2D textures. This no longer restricts the viewer to rendering an isosurface, although choosing a singular color map yields a surface approximation if desired.

GPU volume rendering is quite fast; easily permitting medium-sized volumes (up to 512^3) to be viewed at interactive rates. The main problem is that larger volumes are absolutely limited by GPU memory. To bypass this, it is necessary to store the volume out-of-core in CPU main memory, and page it into GPU memory.

Boada et al. The notion of rendering octree-compressed data was first applied by Boada et al. [BNS01], with an important caveat: scalars are not compressed into a pure octree, but rather an octree is built around bricks of a certain size. From these bricks, "cuts" of the octree are reconstructed into textures, and then sent to the GPU for volume rendering. The octree lookup process was a bottleneck in their implementation; they report sub-interactive frame rates for medium-sized (256^3) volumes. Admittedly, however, available GPU memory at time of publication was a fraction of this size.

Ruijters and Vilanova More recently, a similar technique was applied by Ruijters and Vilanova [RV06] with stunning results. Here, the authors do not use an octree for data paging at all; instead they use the octree to speed up rendering of bricks by skipping empty space. The octree itself is stored in main memory and used in an out-of-core preprocess phase, where the desired bricks are decomposed into visible regions (non-empty octree nodes) and those in turn are mapped to triangular cuts of the volume.

The technique of Ruijters and Vilanova demonstrates that GPU volume rendering can scale to large data. However, it still requires that that data remain uncompressed in main memory, and rendered on the GPU using a paging scheme.

3.3 Ray Tracing

Ray casting involves traversing the path of a ray from an origin pixel in our frame buffer to a point on some surface in space. Then, one evaluates a shading model to color that pixel. Ray tracing is more computationally costly than rasterization, however the relative cost decreases as scene complexity rises, as is the case for large volumes. Moreover, recent advances in parallel or coherent ray tracing [PSL+99, WSBW01] allow for interactive rendering rates on moderately powerful CPU hardware. Nonetheless, interactive ray tracing is generally limited to isosurfacing, due to the high complexity of direct volume rendering.

Ray tracing does entail certain advantages, however. As a purely CPU technique, it has access to full system memory and more sophisticated use of branching and caching than a GPU. As such, it can process a pure octree structure without need of a proxy. Moreover, isosurface ray tracing as proposed by Parker et al. [PSL+98] promises topologically correct surfaces within each cell. Although the global surface is only g_0-continuous at cell borders, it is guaranteed to be "correct" with respect to the original data and a forward-differencing stencil.

Before 1995, octrees were popular spatial acceleration structures for general-purpose polygonal ray tracing [Gla84, Sam89, Sun91, GA93]. Afterwards, hierarchical grids and subsequently kd-trees came into favor due to their faster traversal and better adaptability to irregular geometry. Nonetheless, the octree remains potentially well-suited to volumes, whose voxels are uniformly spaced and non-overlapping.

Knoll et al. The authors combine the min/max acceleration functions of the Wilhelms and Van Gelder [WV92] octree with a compressed format similar to the "cells octree" of Velasco and Torres [VT01]. Instead of containing cells as was the choice of the latter authors, Knoll et al. [KWPH06] build the octree directly around voxels. This requires a fast neighbor-finding scheme to retrieve the values of cell corners when ray tracing; for this the authors borrow (and improve upon) the algorithm of Frisken and Perry [FP02]. Ray traversal is performed on the min/max octree structure, which is the same structure encapsulating the voxel data. For this, the authors employ a traversal similar to that proposed by Gargantini and Atkinson [GA93].

Even employing non-coherent single-ray traversal techniques, Knoll et al. achieve interactive frame rates on multicore architectures. Performance is underwhelming compared

to that of GPU volume renderers; however it allows extremely large and time-variant data to be rendered that otherwise would be difficult to accomodate even with a GPU paging scheme. Octree volumes as proposed by Knoll et al. generally allow volumes to be losslessly compressed into 10-30% their original size, even though they are ordinary pointer octrees and optimized more for fast traversal than maximum compression.

4 Conclusion

In conclusion, octrees are useful for a variety of purposes in volume rendering. In extraction and direct volume rendering they traditionally serve the purposes of acceleration and adaptive multi-resolution representation. For interactive ray tracing applications, it is possible to employ a pure octree volume structure and render extremely large volume data in compressed format. Future applications of octree volume rendering could attempt to combine the pure octree volume with GPU rendering approaches, using out-of-core methods.

References

[BNS01] Imma Boada, Isable Navazo, and Roberto Scopigno. Multiresolution Volume Visualization with a Texture-Based Octree. *The Visual Computer*, 17(3), 2001.

[FB74] R.A. Finkel and J.L Bentley. Quad trees: A Data Structure for Retrieval on Composite Keys. *Acta Informatica*, 4(1):1–9, 1974.

[FP02] Sarah F. Frisken and Ronald N. Perry. Simple and Efficient Traversal Methods for Quadtrees and Octrees. *Journal of Graphics Tools*, 7(3), 2002.

[GA93] Irene Gargantini and H.H. Atkinson. Ray Tracing an Octree: Numerical Evaluation of the First Interaction . *Computer Graphics Forum*, 12(4):199–210, 1993.

[Gar82] Irene Gargantini. An Effective Way to Represent Quadtrees. *Communications of the ACM*, 25(12):905–910, 1982.

[Gla84] Andrew S. Glassner. Space Subdivision For Fast Ray Tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, 1984.

[HS79] G.M. Hunter and K. Steiglitz. Operations on Images Using Quad Trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1(2):145–153, 1979.

[JT80] C.L. Jackins and S.L Tanimoto. Oct-trees and their use in representing three-dimensional objects. *Computer Graphics and Image Processing*, 14(3):249–270, 1980.

[KWPH06] Aaron Knoll, Ingo Wald, Steven Parker, and Charles Hansen. Interactive Isosurface Ray Tracing of Large Octree Volumes. Technical Report UUCS-2006-026, University of Utah, School of Computing, 2006.

[LC87] William E. Lorensen and Harvey E. Cline. Marching Cubes: A High Resolution 3D Surface Construction Algorithm. *Computer Graphics (Proceedings of ACM SIGGRAPH)*, 21(4):163–169, 1987.

[Lev90] Marc Levoy. Efficient Ray Tracing for Volume Data. *ACM Transactions on Graphics*, 9(3):245–261, July 1990.

[LH98] Yarden Livnat and Charles D. Hansen. View Dependent Isosurface Extraction. In *Proceedings of IEEE Visualization '98*, pages 175–180. IEEE Computer Society, October 1998.

[Mor66] G.M. Morton. A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing. *IBM Ltd.*, 1966.

[PSL+98] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan. Interactive Ray Tracing for Isosurface Rendering. In *IEEE Visualization*, pages 233–238, October 1998.

[PSL+99] Steven Parker, Peter Shirley, Yarden Livnat, Charles Hansen, and Peter-Pike Sloan. Interactive Ray Tracing. In *Proceedings of Interactive 3D Graphics*, pages 119–126, 1999.

[RV06] Daniel Ruijters and Anna Vilanova. Optimizing GPU Volume Rendering. *Winter School of Computer Graphics, Pilzen*, 2006.

[Sam82] Hanan Samet. Neighbor finding techniques for images represented by quadtrees. *Computer Graphics and Image Processing*, 18(1):35–57, 1982.

[Sam89] Hanan Samet. Implementing Ray Tracing with Octrees and Neighbor Finding. *Computers and Graphics*, 13(4):445–60, 1989.

[Sam90] Hanan Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Publishing Company, 1990.

[Sun91] Kelvin Sung. A DDA Octree Traversal Algorithm for Ray Tracing. In Werner Purgathofer, editor, *Eurographics '91*, pages 73–85. North-Holland, September 1991.

[VT01] Francisco Velasco and Juan Carlos Torres. Cell Octree: A New Data Structure for Volume Modeling and Visualization. *VI Fall Workshop on Vision, Modeling and Visualization*, pages 665–672, 2001.

[WKE99] Rüdiger Westermann, Leif Kobbelt, and Tom Ertl. Real-time Exploration of Regular Volume Data by Adaptive Reconstruction of Iso-Surfaces. *The Visual Computer*, 15(2):100–111, 1999.

[WSBW01] Ingo Wald, Philipp Slusallek, Carsten Benthin, and Markus Wagner. Interactive Rendering with Coherent Ray Tracing. *Computer Graphics Forum*, 20(3):153–164, 2001. (Proceedings of Eurographics).

[WV92] J Wilhelms and A Van Gelder. Octrees for faster isosurface generation. *ACM Transactions on Graphics*, 11(3):201–227, July 1992.