Eurographics/ IEEE-VGTC Symposium on Visualization 2010
G. Melançon, T. Munzner, and D. Weiskopf
(Guest Editors)

Volume 29 (2010), Number 3

Matrix Trees

Nathan Andrysco and Xavier Tricoche

Department of Computer Science, Purdue University

Abstract

We propose a new data representation for octrees and kd-trees that improves upon memory size and algorithm speed of existing techniques. While pointerless approaches exploit the regular structure of the tree to facilitate efficient data access, their memory footprint becomes prohibitively large as the height of the tree increases. Pointer-based trees require memory consumption proportional to the number of tree nodes, thus exploiting the typical sparsity of large trees. Yet, their traversal is slowed by the need to follow explicit pointers across the different levels. Our solution is a pointerless approach that represents each tree level with its own matrix, as opposed to traditional pointerless trees that use only a single vector. This novel data organization allows us to fully exploit the tree's regular structure and improve the performance of tree operations. By using a sparse matrix data structure we obtain a representation that is suited for sparse and dense trees alike. In particular, it uses less total memory than pointer-based trees even when the data set is extremely sparse. We show how our approach is easily implemented on the GPU and illustrate its performance in typical visualization scenarios.

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.6]: Methodology and Techniques—Graphics data structures and data types; Computer Graphics [I.3.5]: Object hierarchies—

1. Introduction

Data structures such as octrees and kd-trees are ubiquitous in scientific fields that involve spatial data and crucial to the performance of a large number of visualization techniques. Algorithms that manipulate this spatial data can be accelerated when they use these structures. These spatial data structures are also used for compression by grouping similar values together. Practically, octrees and kd-trees are built upon an underlying data structure that is either pointerless or pointer-based. Since the tree in the pointer-based approach needs to be traversed like a linked-list, each node requires some additional memory overhead such as pointers to the child/parent nodes and possibly position information. The main advantage of using the pointer data structure is that the memory required can adapt to the structure of the data.

Pointerless trees are typically implemented using an underlying array and exploit regular structure to derive spatial information and child/parent array locations. Even though pointerless trees require less memory overhead per node, they need to allocate enough space for all possible nodes despite how sparse the tree may actually be. For this reason, pointerless trees do not adapt well to many data sets and are rarely used since they can potentially require much more space than pointer-based trees for the same data set.

In this paper, we present a new underlying representation for octrees and kd-trees that has the algorithmic speed and simplicity of traditional pointerless approaches and uses less memory than pointer based representations. We review these methods in Section 2. This pointerless representation makes use of matrices to store the data, as discussed in Section 3. This allows us to take better advantage of the tree's regular structure than previously proposed methods and results in significant improvements to both algorithm speed and data structure size. In addition, by making use of a sparse matrix data structure (Section 4), we achieve a significant reduction in the memory footprint that outperforms pointerless and pointer-based trees, even when the tree is sparse. We explain this property through a detailed complexity analysis in Section 5. In addition, by keeping memory costs low and avoiding pointers, our method can easily be translated to the GPU, as shown in Section 6. Typical applications of our data structure in visualization applications are proposed in Section 7 to illustrate both its versatility and performance.

2. Prior Work

Octrees and kd-trees are some of the oldest data structures used in computer graphics [Mor66, Ben75], and as such, researchers have developed a wide variety of implementations over the years. Besides being able to classify a tree's implementation as pointerless or pointer-based, we can classify a tree based on its shape and how that shape is recorded. *Full* trees require each non-leaf node to have its full complement of children, 8 for octrees and 2 for kd-trees. This means that the data stored in the tree cannot be compressed using the tree's natural data compression capabilities, and therefore, it can require a significant amount of space. *Branch-on-need* trees [WVG92] subdivide only where tree refinement is needed, which typically results in much smaller trees. A third type of tree, *linear* [Gar82], further minimizes the space required by storing the leaf nodes contiguously in memory. More details on previous octree methods can be found in a survey paper by Knoll [Kno06].

Recent techniques that use octrees and kd-trees to perform ray casting/tracing include [KWPH06, BD02, FS05, WFMS05, HSHH07]. Of note is the recent work done by Crassin et al. [CNLE09] (Gigavoxels) and Hughes and Lim [HL09] (KD-Jump). In Gigavoxels, the authors show impressive performance numbers by converting triangle geometry to octree voxels. Using our proposed method, both their memory requirements and speed numbers would be improved (by removing their use of pointers and using our more efficient tree traversal algorithm). Though KD-Jump is memory efficient in the sense that it is a pointerless representation like ours, their tree construction requires that the number of leaves equals the number of grid voxels. Since our method does not have this restriction, we are able to trim branches to remove or compress data values, which provides memory savings. By the authors' own admission, their method is best suited for isosurfaces on the GPU and using it as a general data structure might be limited.

One valuable use of spatial data structures in computer graphics and visualization is to accelerate point location queries. These queries in large unstructured meshes, especially those meshes created by numerical simulations, can be particularly costly. Langbein et al. [LST03] use kd-trees to help solve this problem. Many papers try to avoid these costly queries in unstructured meshes altogether by simplifying the data. Leven et al. [LCCK02] approximate the unstructured mesh with regular grids to help accelerate volume rendering. Another level-of-detail approach to accelerate the mesh's visualization is done by Callahan et al. [CCSS05]. Song et al. [SCM*09] use the transfer function to cull cells that will not contribute to the final rendered image.

Sparse matrices are defined as matrices that contain a high percentage of zeros compared to non-zero values [GJS76]. This type of matrix often arises in scientific simulations that involve solving partial differential equations. In many modern day applications, the matrices produced are very large and naively representing them with all zero values intact is prohibitively inefficient. A typical solution to this memory problem is to only record the non-zero values and their positions within the array. Bell and Garland [BG08] give a good survey of the more commonly used variations of these sparse data structures and discuss their performance on the GPU.

3. Matrix Trees

We propose an octree and kd-tree data structure that is built upon matrices to handle the underlying data storage. A motivation for this approach is that completely full quad/octrees and implicit kd-trees look like layered matrices, as can be seen in Figure 1. As a result, we represent each layer of the tree by its own individual 3-dimensional matrix. Each of these matrices are represented as one or more vectors, which is in contrast to traditional pointerless representations that only use a single vector to represent the entire tree. Because we derived data location information from our underlying matrix representations, we do not need to make use of any pointers, which results in an efficient translation to the GPU.

The key observation for memory preservation is that sparse trees can be thought of as having NULL, or zero, values at non-existent nodes. These data sets can be represented using sparse matrix data structures to save space, while maintaining the regular structure of the underlying matrix. As with most data structures, one trades speed for size efficiency. In brief, a random data query with our sparse matrix data structure requires $O(\log_2 m)$ time, where m is the average number of non-zero entries per row. A more detailed description of our proposed sparse matrix data structure is found in section 4. We demonstrate in section 7 that this lookup time is very efficient in practice.

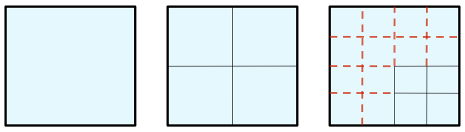


Figure 1: *A quadtree using an underlying sparse matrix representation to fill in any "holes" (dotted red lines).*

3.1. Exploiting Regular Structure

Similar to the pointerless tree approach, using matrices as the underlying data structure allows us to derive information at each node instead of explicitly storing it. Each node is represented by a 4-tuple that includes the height in the tree of the node and the x, y, z integer indices of the matrix at that particular height. This information is not stored at the nodes, but instead is just used for indexing into memory.

Using both the 4-tuple and the regular structure of the matrices, we can derive parent and child index information. Calculating the 4-tuple of the parent involves first subtracting 1 from the height of the child (by convention, the root is located at height 0). Next, each index where a split was made between the parent and child level is divided by 2, which can be efficiently implemented by right-shifting the bits once. Octrees, where a split occurs along all three dimensions at every level, will have all three of the indices divided. Kd-trees are only split along a single dimension per tree level, so which index is divided depends on the split. For example, a split along the x-axis requires the x-index be divided by 2.

Ancestors higher up in the tree can similarly be calculated. For octrees, one simply needs to right-shift the bits the number of times one wishes to traverse up the tree toward the root. Since kd-trees split directions are arbitrary, ancestor calculations are slightly more complicated. A naive implementation involves iteratively doing a single parent calculation until the ancestor is reached. A faster method precalculates the number of splits along each direction for each level in the kd-tree, as demonstrated in [HL09]. The difference in these "sum of split directions" between the child and ancestor indicates the number of shifts for each index.

A child's 4-tuple is similarly calculated given the parent's 4-tuple. The height is incremented and each index along a split direction is multiplied by 2, which can be implemented as a left-shift of bits. Additionally, one needs to add 1 to the indices depending on the location of the child in relation to its parent. As a convention, we add a 1 if the child's index component is in the positive direction.

Like with other previous pointerless tree representations, the 4-tuple also allows derivation of node spatial information by storing only the tree's bounding box information. Just from this information, we can calculate each node's position and size. Similarly, this information can be used to spatially hash directly to a node given a point in space, which is exploited in section 3.2.

3.2. Leaf Finding

Leaf finding is a very common operation used when dealing with octrees and kd-trees. Traditional octrees have time complexity of $O(\log_8 N_{total})$, where N_{total} is the total nodes in the structure. Similarly, kd-trees have a time complexity of $O(\log_2 N_{total})$. Linear trees [Gar82], where only leaf nodes are stored, can be searched in a binary fashion so that they achieve a corresponding leaf finding time of $O(\log_2 L)$ time, where L is the number of leaf nodes. By representing the tree with matrices, we can further improve upon this time to $O(\log_2 h)$, where h is the height of the tree.

Given a point, we calculate the node that contains it at the lowest tree level by using a spatial hash, which is made possible by the regular structure of the matrices. Provided that this leaf node does not exist, we use the fact the the hash eliminates all nodes in the tree except those along the path from the leaf to the root. We can perform a binary search along this path, providing us with a large performance gain

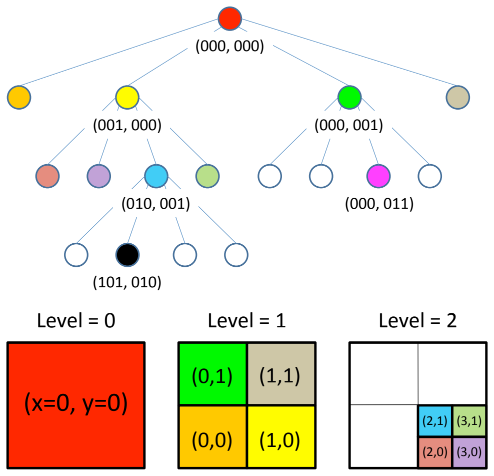


Figure 2: *An example showing child, parent, and neighbor relationships. Binary indices are shown beneath select nodes to illustrate common ancestor computation between black and pink nodes. We first compute black's ancestor at the same level as pink - blue. We then examine the bits between blue and pink to discover that red is the common ancestor.*

over previous methods. As noted, the use of a sparse matrix representation requires additional complexity for locating the node, which is discussed in detail in section 5.2.

3.3. Finding Common Ancestor Node

Some tree traversal methods make use of ancestor information to accelerate our leaf search. In applications where our search point is moved in small increments (i.e ray casting or streamline integration), we may save time by saving the last node used and starting our downward search from the ancestor that contains both this last node and our search point, thereby possibly limiting our search to a small subset of the tree instead of starting at the root each time. KD-Jump [HL09] uses a similar technique whereby the algorithm keeps track of tree depths it needs to revisit as it moves along a ray. Using this information (stored in a bit vector) and the implicit structure of their kd-tree, it can "jump" up the tree to the appropriate ancestor. Our method for computing ancestors does not require any previous traversal knowledge and is not limited by the size of a bit vector.

Given two separate node 4-tuples, we exploit the regular structure of the matrices to compute the common ancestor. We first need to put the two nodes at equal depth, which is done by computing the ancestor of the lower node so that this ancestor is at the same height as the other node (see section 3.1). We now examine the difference in bits between the two nodes' indices by XOR-ing them. The position of the most significant bit, calculated using a $O(\log_2 n)$ algorithm (n is the number of bits in the index), that differs indicates how many more levels we need to traverse up the tree to find the first common ancestor. See Figure 2 for a brief example. We note that the cost of $O(\log_2 n)$ might be considered high, but the cost to lookup in the sparse matrix data structure might be even higher. This penalty can also be lessened using GPU functionality, such as CUDA's *__clz* function.

3.4. Neighbor Finding

Samet [Sam90] shows that the average cost of finding a neighbor of the same depth for pointer-based octrees is O(1). Though two nodes may border each other spatially, these nodes may actually be located far away within the tree's representation. This worst case scenario results in a much more costly leaf finding operation. In contrast, pointerless octrees will always take O(1) time by just adding/subtracting to the node's indices. Our pointerless representation is analogous.

Doing the basic calculation from the previous paragraph results in a neighboring node at the same depth in the tree. However, this node may not be a leaf or may not exist at all. Using a method similar to the one used in leaf finding (section 3.2), we can efficiently search along a path to find the desired leaf. If the node exists, we search down the tree. Otherwise, we search up towards the root. We limit our binary search by setting one end point of the search to be the depth neighbor node from the initial calculation. If the node exists, we can further limit the search by setting the other end point to the common ancestor node between the original node and its neighbor at the same depth (section 3.3).

3.5. Corner Centric Data

Storing data at positions other than the node's center often helps simplify the implementation of quadtree, octree, or kd-tree based algorithms. Storing data at corners is particularly useful as it simplifies algorithms that require interpolation [KT09], with recent applications in animating fluids [BGOS06, LGF04] and ray casting [HL09]. However, this ease-of-use comes at a cost for previous tree data structures. Since nodes share common corner values with both its neighbors and ancestors, one needs to map the corners to another data structure so that the data is not duplicated. The mapping incurs additional storage and programming difficulty, which would be desirable to avoid.

Given a leaf level node matrix of size $n_x * n_y * n_z$, we construct an additional matrix of size $(n_x + 1) * (n_y + 1) * (n_z + 1)$ to represent our data points at corners. Because the nodes have a dual relationship with the corner points we can access corner data from any node without any additional pointer overhead. To calculate the data index of our corner matrix, one takes the given node's indices and multiplies by n_{leaf}/n_i, where n_{leaf} and n_i are the number of cells along a particular direction of the leaf level and i^{th} matrix respectively. Analogous to computing a child, we add 1 if the relative direction to the corner is positive.

An added advantage of this approach is that we can traverse through blocks of data once we reach a leaf with minimal extra effort. This block traversal approach has become common in recent applications [CNLE09, HL09] due to relatively poor memory performance on GPUs that results from traversing the full tree structure [AL09]. These data blocks introduce extra memory overhead due to the additional pointers needed to reference them and may also require data duplication at the blocks' borders to help facilitate interpolation. Our corner data representation eliminates the need for pointers (since addresses are implicit) and data duplication (since all data is contained within a single matrix).

Though the idea is not further explored in this paper, we note that edge or face data could be recorded in a similar fashion by exploiting the regular structure of the matrices.

4. Sparse Matrix Implementation

We use the compressed sparse row (CSR) format for data storage when it is more memory efficient than using a full matrix (see section 5.1 for further analysis). For completeness, we briefly summarize the basic properties of this format. Please refer to [BD01] for more details.

Given a 2D sparse matrix with dimensions n^2, we construct two for indexing and one to store data. The first index vector, with length $n+1$, stores the row indices and is used for an O(1) lookup into the next index vector. This second index vector, which stores column index information, has length $O(nm)$, where m is the average number of non-zero values (or nodes) per row. This vector is sorted by row to allow a binary search of the m values of interest, $O(\log_2 m)$. The result of this search enables a O(1) lookup into the data vector (of size nm), giving an overall lookup time of $O(\log_2 m)$. Random insertions take $O(n)$ time.

Provided a 3D matrix of size n^3, we project two of the dimensions into the first index vector (i.e. row and column), giving it a length of n^2. This maintains the $O(\log_2 m)$ lookup time complexity, but increases our first index's memory to $O(n^2)$. Consequently, the insertion time is $O(n^2)$. An alternative approach would be to introduce a third index vector, but this incurs another $O(\log_2 m)$ lookup. We believe the $O(n^2)$ memory requirement is a fair tradeoff since it is not significant compared to $O(n^3)$ for a full matrix. The $O(n^2)$ insertion time can be prohibitive when building the tree, but we suggest using the approach in [Bri08] where a faster insertion/slower data access scheme is first used and then quickly converted to our desired representation.

To help reduce the cost when interpolating data, we suggest fetching blocks of data instead of doing each fetch individually, which requires 8 $O(\log_2 m)$ operations for trilinear interpolation. Since indices and data along one of the dimensions are stored in contiguous memory, we can retrieve the next piece of data by looking at the next element within the vector, reducing our total cost in half. Neighboring rows in the matrix tend to have a similar construction (i.e. number of nodes, relative location of nodes). Using this insight, we can use our initial O($\log_2 m$) result as a guess when performing additional searches. This has the potential to reduce our entire block fetch operation to a single O($\log_2 m$) fetch.

4.1. Improved CSR Size and Speed

We observe that if we only wish to know whether a node exists or not, as might be the case when data is only stored in the corner matrix, we do not require the CSR's data vector. When there are more "non-zero" values (i.e. existing nodes) in the matrix, we can use a *complementary* representation that keeps track of "zeros" (non-existent nodes). Using this representation when the matrix has more "non-zero" values then "zeros" results in saved space (since fewer indices will need to be recorded) and decreased lookup time (since the time is dependent on the size of the index vector). This format has most likely not been used in other applications since these applications' sparse matrices tend not to come close to 50% sparsity. For our application, we encounter cases when this complementary format is desirable.

Even with node-centric data, this complementary CSR approach can still be beneficial. Of course, we would still use the regular CSR when it is more memory efficient to do so. When performing a lookup, we will still calculate a position within the index vector. If we design our search to return the position in the vector where it would occur had it been in the vector (i.e. right before the first value greater than our search value), i_s, we can calculate the number of existing nodes that occur before our search value (remember the index vector contains non-existing nodes) and use this to index into the data vector i_d. Assume that we are projecting rows and columns to the vector with size O(n^2), we have: $i_d = n_z * n_{xy} + z - i_s$, where n_z is dimension along the z-axis, n_{xy} is the projected position of a given row/column tuple into the second (depth) index vector, and z is the depth value we are searching for. By removing the one-to-one index/data mapping restriction of the traditional CSR approach, we can easily facilitate other storage schemes, such as ones that store separate data types at leaf and non-leaf nodes (e.g. [KWPH06, VT01]).

5. Complexity

For clarity, let us first introduce some convenient notations.

h	$=$	Height of the octree
n_i	$=$	2^i (Max. nodes along an axis at height i)
m_i	$=$	Mean nodes along any axis at height i
m_i/n_i	$=$	Sparsity ratio along any axis at height i
N_{total}	$=$	$\sum_{i=1}^{h} n_i^3$ (Total nodes in completely full tree)
M	$=$	$\sum_{i=1}^{h} m_i n_i^2$ (Total nodes in sparse tree)
s_d	$=$	Bytes needed to store data at a node
s_{idx}	$=$	Bytes needed to store an index (e.g. int)
s_{ptr}	$=$	Number of bytes needed for a pointer

Observe that because n and m are the same along any axis, the sparsity ratio (m/n) and total number of nodes (M) will be the same regardless of the axis we examine. For simplicity, we will do our analysis on octrees since each matrix at a given level has the same dimensions, regardless of the axis we examine. Since kd-trees split directions are arbitrary, this property does not apply to them. In general kd-trees are about 3 times the height of octrees, and because of the increased tree height, kd-trees will benefit even more from our size and traversal speed improvements.

5.1. Size

The sparse matrix representation (section 4) uses space that is dependent on the sparsity of the octree. We distinguish between the integer element size of the two index vectors, row/column and depth. If our data set can be encompassed by a 1024^3 matrix, or an octree of height 10, our depth index can be represented as a 16-bit integer. The row/column index projection requires the row and column indices to share the full 32-bits, limiting the octree height to 16. At each level i, the row/column index structure requires $n_i^2 s_{idx_1}$ bytes. The amount of space required for the data vector is proportional to m (i.e. the number of nodes per dimension). In total it is $m_i n_i^2 s_d$ bytes. The depth vector is dependent on our chosen CSR representation (section 4.1), providing us with $\min(m_i, n_i - m_i) n_i^2 s_{idx_2}$ bytes.

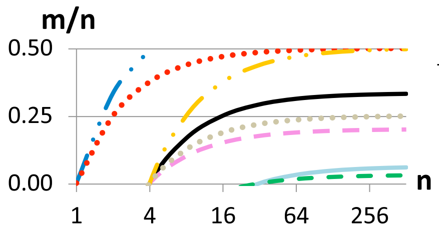
A traditional pointerless octree is analogous to matrix octrees where each level uses a regular, full matrix as its underlying representation. Since room is allocated for all nodes, regardless of their existence, the total space required is $N_{total} s_d$. Or on a single level of the octree the space required is $n_i^3 s_d$. To distinguish between the data on the sparse matrix side of the equation and the full matrix side, we annotate the variables with s and f respectively. This is useful if we wish to only track where nodes are in the octree, as we can omit the data vector entirely from the CSR side. Due to the sparse matrix data structure requiring additional space to help with indexing its non-zero entries, there will actually be situations where it requires more size than the full matrix in certain situations. We can calculate when the two matrix representations will be equal by comparing their respective size equations:

$$n_i^3 s_{d_f} = n_i^2 s_{idx_1} + m_i n_i^2 s_{d_s} + \min(m_i, n_i - m_i) n_i^2 s_{idx_2}$$

$$\frac{m_i}{n_i} = \begin{cases} \dfrac{s_{d_f} - \frac{s_{idx_1}}{s_n}}{s_{idx_2} + s_{d_s}} & \text{for} \quad \frac{m_i}{n_i} \leq 0.5 \\[2ex] \dfrac{\frac{s_{idx_1}}{s_n} - s_{d_f} + s_{idx_2}}{s_{idx_2} - s_{d_s}} & \text{for} \quad \frac{m_i}{n_i} > 0.5 \end{cases}$$

Within our program, we can dynamically determine the matrix type to use to represent the nodes/corners based on the equation above. A sparsity ratio less than the ratio when matrix sizes are equal means we will save on space using CSR. Even if we are only tracking nodes within the octree

Color	s_d	s_{idx_2}	$n \longrightarrow \infty$
━━━	1	2	1/3
- - - -	1	4	1/5
—·-·—	4	2	1
········	4	4	1/2



Color	s_d	s_{idx_2}	$n \longrightarrow \infty$
━━━	1/8	2	1/32
- - - -	1/8	4	1/16
—·-·—	1	2	1/2
········	1	4	1/4

Figure 3: *Graphs of the equation in section 5.1 ($s_{idx_1} = 4$). The left table shows when the nodes contain data, the right table is when they do not. If our matrix dimensions, n, and sparsity ratio, m/n, fall into the area below the line that matches the size parameters from the tables, we will save space using the sparse matrix representation.*

and do not care about storing data at the actual nodes, the full matrix representation still needs to store some form of data to indicate whether a node exists or not. In Figure 3 we show the result when the nodes are represented using *char* (1 byte per node) and as a more space efficient, but slightly more computationally expensive bit-vector (1/8 byte per node). This graph also shows that with low values of n (those levels near the root), it is typically more beneficial to use the full matrix.

5.1.1. Pointer Octrees

Pointer octrees can be implemented in a variety of ways. For our analysis we will assume a very space-efficient one. Node spatial information can be derived from the octree's bounding box and its path from the root. To help facilitate this process, each node will have a pointer to its parent so that a random node can derive its position without any additional information. This parent is also useful for other tasks such as finding common ancestor nodes or neighbor finding. Knoll et al. [KWPH06] demonstrated that one can use data spatial locality to represent children using a 32-bit pointer and offsets. This is further simplified to a single pointer for our analysis with a NULL child pointer indicating a leaf. In total, each node requires $2s_{ptr} + s_d$ bytes, or $8 + s_d$ on a 32-bit architecture. If there are m nodes per 2D slice (n^2), the total memory requirement for one octree level is $m_i n_i^2(8+s_d)$. Comparing to full matrices:

$$\begin{array}{rcl} m_i n_i^2(8+s_d) & > & n_i^3 s_d \\ \frac{m_i}{n_i} & > & \frac{s_d}{8+s_d} \end{array}$$

Letting $s_d = 4$ shows that the pointer octree is more memory efficient only when $\frac{m_i}{n_i} < \frac{1}{3}$. The cost of the pointers at each node quickly adds up, something that is free with our representation. When the pointer-based tree is more memory efficient than the full matrix, we propose a sparse matrix data structure as the underlying data representation (Figure 3). Assume the largest memory configuration for the sparse matrix representation ($s_{idx_1} = s_{idx_2} = 4$):

$$m_i n_i^2(8+s_d) \quad > \quad 4n_i^2 + m_i n_i^2(4+s_d)$$

Solving for m, we find that the sparse matrix representation is more memory efficient when $m > 1$ (i.e. there is on average 1 node per 2D slice of the octree), which is typical of most data sets. Memory requirements comparisons for actual data sets can be found in section 7.

5.2. Time

The new leaf finding algorithm presented in section 3.2 assumes O(1) time to access a node within a matrix representation. This assumption holds true when using the basic, full matrix representation. Here we will analyze the time complexity of leaf finding when CSR is used. For reference, the time for a CSR node lookup is $\log_2 m_i$ at level i (section 4). Assume that each level has the same average sparsity. If there is an average of m nodes along each matrix dimension at the leaf level h, then there are $m/8$ nodes at $h-1$, $m/64$ nodes at $h-2$, etc.

Assume the worst case binary search scenario for the leaf finding algorithm, i.e. we encounter the levels where there are the most number of nodes. The search begins with a hash to a node at the leaf level. In the worst case the node does not exist so we perform a binary search, first looking at the matrix located at level $\frac{h}{2}$. This is followed by searching at $\frac{3h}{4}$, then $\frac{7h}{8}$, etc. (i.e. the matrices with the most number of nodes along the binary search path) and continues until $h-1$ is reached. The number of nodes along an axis at level i is equal to $\frac{m}{2^{\frac{3h}{2^i}}}$, where $1 \le i \le h$.

$$\begin{array}{rcl} \log_2 m + \sum_{x=1}^{\log_2 h} \log_2 \frac{m}{2^{\frac{3h}{2^x}}} & = & \log_2 m + \sum_{x=1}^{\log_2 h} (\log_2 m - \frac{3h}{2^x}) \\ & \ge & \log_2 h \, \log_2 m - 3h \\ & \Rightarrow & O(\log_2 h \, \log_2 m) \end{array}$$

Given the same worst case scenario as above, the time taken to find a leaf by linearly searching from the root will be $h-1$. The most costly lookup time we can incur for CSR happens when $m = n/2$ since if the octree was completely full (i.e. $m = n$) we would use the complementary CSR representation, which results in a O(1) lookup. From this we can simplify $\log_2 m$ to $h-1$ since $n = 2^h$.

$$(h-1)\log_2 h - 3h < h - 1, \text{for } h \le 18$$

When $h \leq 18$ the matrix octree is faster than the pointer octree even with the assumptions made. In practice an octree of height 18 is sufficient to encompass most data sets, having a leaf level matrix of size $(2^{18})^3$. If this example octree had sparsity of 50% and each node was represented as 1 byte, it would require 8 TB to encode only the leaf nodes. Data sets are usually very sparse at this level of refinement so the assumption of $m = n/2$ is not a realistic approximation. For best case scenarios where the leaf is found near the root, the pointer octree will have better performance due to our method being a bottom-up approach combined with a binary search. But for real-world data sets, the matrix octree will outperform pointer octrees as shown in the results section 7.

6. GPU Implementation

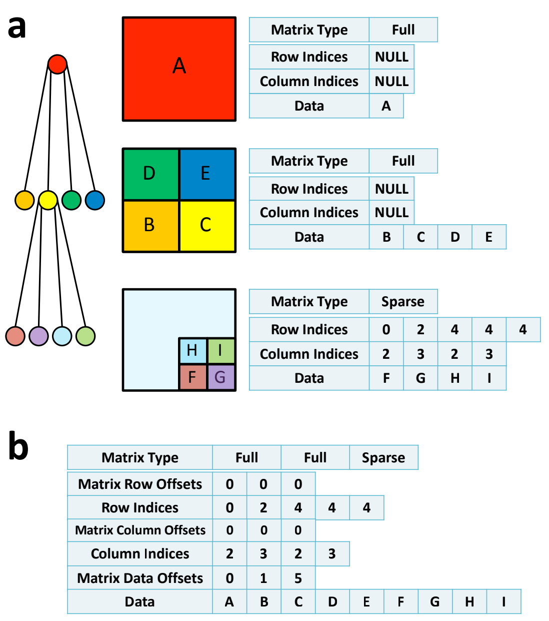
Implementing the matrix trees on the GPU is straightforward. Both the data and the indices are stored as vectors on the CPU, so the GPU implementation just requires us to copy the vectors to a texture. We stack all vectors of similar function together so that only one texture is needed per vector functionality. For nodes, there are three vectors that need to be copied to the GPU - data, row/column indices, and depth indices. To distinguish between different levels of the tree and different matrix representations, we also pass index offset and matrix type identifiers. These vectors are small - they have length that is the same size as the height of the tree. We also need to pass a small amount of positional information, namely the tree's bounding box. Additional data such as node size or number of nodes per level is inherited from our matrix representation. It may be wise to precalculate these values to speedup various computations. The exact same steps should be taken to copy corner data to the GPU. It is advised that the corner values are stored as a single matrix to better exploit the texture's memory cache and so that blocks of data can be retrieved in a quicker fashion.

7. Applications

The work presented in this paper is not application specific, but rather a versatile building block that a broad range of areas can benefit. We have chosen two applications that require both a space efficient and quick data structure to accelerate their applications on large data sets. Specifically, we show that our method is efficient when performing ray and point location queries, two basic operations used in a broad range of volume rendering applications. All examples are performed on Windows Vista (32-bit) with an Intel Xeon X5460 (3.16 GHz) processor and NVIDIA GeForce GTX 280 GPU. Our GPU code is implemented using CUDA.

7.1. Ray Caster

We require the dimensions of the tree's underlying matrices to be a power of 2, but space is not wasted due to our underlying sparse matrix representation. Values are stored at



Matrix Type	Full				
Row Indices	NULL				
Column Indices	NULL				
Data	A				

Matrix Type	Full				
Row Indices	NULL				
Column Indices	NULL				
Data	B	C	D	E	

Matrix Type	Sparse				
Row Indices	0	2	4	4	4
Column Indices	2	3	2	3	
Data	F	G	H	I	

Matrix Type	Full		Full		Sparse				
Matrix Row Offsets	0	0	0						
Row Indices	0	2	4	4	4				
Matrix Column Offsets	0	0	0						
Column Indices	2	3	2	3					
Matrix Data Offsets	0	1	5						
Data	A	B	C	D	E	F	G	H	I

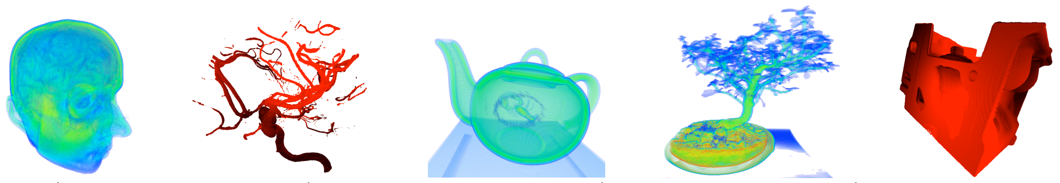
Figure 4: *a) A quadtree with its underlying matrix types and vectors. b) When translating to the GPU, we concatenate the vectors from the CPU and create auxiliary helper vectors.*

the node corners to simplify the interpolation process. Comparatively, a kd-tree will have a height of about 3 times that of its octree counterpart. To further accelerate isosurface extraction, we calculate min/max values for each node. The leaf nodes' min/max values are calculated from the corner data, while non-leaves use their children's min/max. Table 5 shows the memory requirements for different tree representations. Pointer-based trees' size numbers are calculated using the space efficient representation described in section 5.1.1. We conservatively estimate that each leaf node can map each of its corners to the data using only 1 byte / corner.

7.1.1. Traversing a Tree along a Ray

Traversing a spatial tree falls into one of two categories, stack-based and stackless. Unfortunately, stack-based approaches are not appropriate for GPUs due to high memory demand. The authors of [FS05] realized this bottleneck and devised two stackless approaches, KD-Restart and KD-Backtrack. Though they were developed for kd-trees, they have been applied to octrees such as in [CNLE09] (KD-Restart). In [HL09], a new scheme named KD-Jump is proposed. Though it claims impressive numbers, their method manipulates bits in a manner that poses restrictions to octree height and adds additional computation complications.

We demonstrate improvements to state-of-the-art methods by using our improved tree operations in section 3. We be-



	Grid		Properties for Both Trees			Octree			KD-Tree		
Data Set	Data	Dim	Corners	Data	Leaves	NLN	MAT	PTR	NLN	MAT	PTR
Head	27.25	256x256x109	20.43	7.73	19.20	2.96	16.70	31.55	20.22	23.733	44.72
Aneurism	64.0	256x256x256	4.35	1.65	2.31	0.76	3.39	4.12	3.94	5.09	6.54
Teapot	44.5	256x256x178	13.30	5.04	11.54	2.01	10.75	19.15	12.97	15.39	27.50
Bonsai	64.0	256x256x256	36.28	13.73	27.85	5.55	26.95	46.74	33.79	38.12	68.28
Engine	32.0	256x256x128	16.67	6.31	14.93	2.43	13.46	24.63	16.23	19.18	35.16

Figure 5: *The data elements are 4-byte floats to facilitate interpolation on the GPU. Size information for min/max values is not included. Tree properties are expressed in 100,000s of elements, data in MBs. Abbreviations: Non-leaf nodes (NLN), matrix based tree data structure overhead (MAT), pointer-based overhead (PTR).*

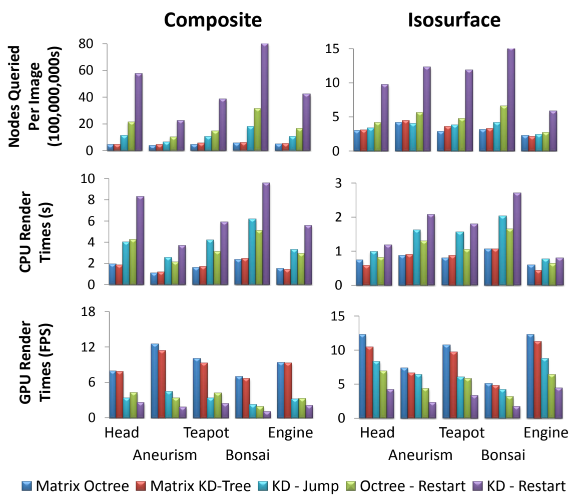
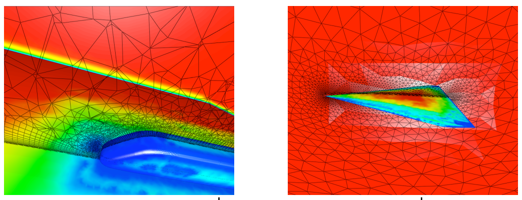


Figure 6: *Graphs showing ray cast render times.*

gin the search with our leaf finding operation (section 3.2). If a node is found that is not located at the leaf level, we keep traversing the tree to skip the empty space until a valid leaf node is found. Empty space skipping involves moving the ray over portions of the tree that do not contain any values that are of interest to our ray caster. When a node of non-interest is encountered, we traverse to the next neighbor along the ray by using the ray's intersection point with the node and the method described in 3.4. We keep performing iterations of this empty space skipping algorithm until a valid leaf node is found. When performing isosurface extraction, we use the min/max values to guide the leaf and neighbor searches by treating nodes as non-existent if the isovalue is outside the range of the node's min/max value.

Table 6 shows performance numbers for various tree traversal methods on a 800x800 pixel image. Our method requires fewer node queries, and as such, is faster than previous traversal schemes. We show speedups of over 3x than the next closest method in some examples, with everyone of our tests outperforming the previous methods on both the CPU and GPU. Our binary search along the tree's height does a very good job of negating the undesirable property of the kd-tree having 3 times the depth of the octree, with the end result of having similar render times for both tree types (unlike with octree/kd-restart). Our results were obtained using a more general approach than those used in [HL09] or [CNLE09], which use application specific optimizations to get speed gains on the GPU. We note that these optimizations can be applied to our method as well, but our focus is on our general applicability of the data structure.

7.2. Unstructured Mesh Visualization



	High Speed Train	Delta Wing
Tetrahedra	931,230	3,853,502
Prisms	1,738,863	2,419,140
Pyramids	14,737	0
Vertices	1,068,701	1,889,283
Size in Memory (MB)	66.520	135.791

Figure 7: *The geometry for the high speed train (left) and delta wing (right). A cut plane of the 3D cells is shown along with surface triangle sizes mapped to a logarithmic BGR color scheme (blue indicates smaller, red larger).*

	Octree						KD-Tree					
	Queries		CPU		GPU		Querries		CPU		GPU	
	Cells	Nodes	Leaf	Total	Leaf	Total	Cells	Nodes	Leaf	Total	Leaf	Total
Train	4.884	1.031	0.561	1.591	0.031	0.952	4.730	1.180	0.827	1.903	0.047	0.858
Wing	366.609	1.149	0.546	10.688	0.168	7.82	17.291	1.341	0.782	6.558	0.157	4.682

Figure 8: *Data is the average of 1,000,000 random point queries. Tree traversal times and total overall times (in μs) are shown.*

On the CPU, we represent the unstructured mesh as a series of vectors. The first vector represents the locations of the mesh's vertices. Another vector is used to store each cell. The only data needed to represent each cell are index pointers to each of the cell's vertices. Additional data such as bounding box/sphere information is not stored to keep memory costs low. Though by pre-computing this information, we can achieve a decent speed up when searching for cells within the mesh. We make the different cell types polymorphic so that all of them can be stored in a single vector. The cell types used in our examples are tetrahedra, pyramids, and three-sided prisms. There is no guarantee that the pyramid or prism is convex. Any additional vertex data, such as velocities, is stored in its own vector.

The transformation of the data to the GPU is straightforward for the vertex data. For cells, we concatenate all the index pointers and store them as a single vector. We assure that all cell types are grouped together, so that we only need a very small vector that stores the offsets for each cell type.

Many visualization approaches require some kind of point location. This is easily done in structured grids, but can be a major performance bottleneck in unstructured ones. To help accelerate point location, we use our matrix trees where we store at each node a vector containing the cells that intersect the node. We need to make a decision when constructing the tree. The numbers provided are for the high speed train data set using an octree. One method is to always duplicate cells that overlap more than one leaf, i.e. the cell intersects or encompasses the leaf. We perform this intersection test by testing each cell face with each node face. Though this method performs best for random point queries (about 5.3 cells tested per query), the duplicate data balloons up the octree size to over 475 MB, which prevents us from transferring it to the GPU. The other extreme is to never duplicate cells that overlap multiple nodes and instead push the cell to the parent. This results in a small tree (11.5 MB), but gives terrible performance (7,718 cells tested per query). Taking a hybrid approach keeps both memory and time costs low. We allow cells to be duplicated, but only at the lower levels of the tree. For this example, our resulting octree is only about 40 MB and requires 8.8 cells to be examined per query.

The tree's translation to the GPU becomes a little more complicated with the addition of a cell index vector at each node. Our translation is done by creating two vectors of integers in texture memory. The first vector is a concatenation of all the cell index vectors located at each node. The other vector stores the node offsets into the first vector.

8. Conclusions and Future Work

We have shown that the matrix representation uses less space than traditional pointer or pointerless trees, allows for faster algorithms, and makes it easier to implement certain tree operations. The only added complexity of our approach is the possible difficulty in implementing the sparse matrix representation. Though our method is currently limited to static trees and split planes down the center of nodes, we believe our method can be extended to allow for general, dynamic implicit trees on the GPU with arbitrary split planes.

We demonstrated that previous state-of-the-art ray casting methods can benefit from our data structure. Combining our representation with the optimizations from these methods will allow for incredible performance in visualization and general graphics applications. Further performance improvements can be obtained by exploring sparse matrix representations that are optimized for computer graphics applications, such as the work done in [LH06].

Using our trees, we were able to represent entire unstructured mesh data sets on the GPU with little memory overhead. We plan to further enhance the representations of these data sets to improve mesh access times on the GPU. Additionally, there are very large unstructured meshes that are too big to fit into most modern day desktop's main memory using current mesh representations. We would like to explore new data structures that will allow us to stream the mesh information onto the GPU for interactive analysis.

Acknowledgements

The datasets used in the ray casting examples are courtesy of `volvis.org`. The CFD datasets were provided by Markus Rütten, DLR Göttigen, Germany. This work was supported in part by a gift by Intel Corportation.

References

[AL09] AILA T., LAINE S.: Understanding the efficiency of ray traversal on gpus. In *Proc. High-Performance Graphics 2009* (2009).

[BD01] BANK R. E., DOUGLAS C. C.: Sparse matrix multiplication package (smmp), 2001.

[BD02] BENSON D., DAVIS J.: Octree textures. *ACM Trans. Graph. 21*, 3 (2002), 785–790.

[Ben75] BENTLEY J. L.: Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9 (1975), 509–517.

[BG08] BELL N., GARLAND M.: *Efficient Sparse Matrix-Vector Multiplication on CUDA*. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, Dec. 2008.

[BGOS06] BARGTEIL A. W., GOKTEKIN T. G., O'BRIEN J. F., STRAIN J. A.: A semi-lagrangian contouring method for fluid simulation. *ACM Transactions on Graphics 25*, 1 (2006).

[Bri08] BRIDSON R.: *Fluid Simulation for Computer Graphics*. A K Peters, Ltd., Wellesley, MA, USA, 2008.

[CCSS05] CALLAHAN S., COMBA J., SHIRLEY P., SILVA C.: Interactive rendering of large unstructured grids using dynamic level-of-detail. In *Visualization, 2005. VIS 05. IEEE* (Oct. 2005), pp. 199–206.

[CNLE09] CRASSIN C., NEYRET F., LEFEBVRE S., EISEMANN E.: Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2009), ACM, pp. 15–22.

[FS05] FOLEY T., SUGERMAN J.: Kd-tree acceleration structures for a gpu raytracer. In *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (New York, NY, USA, 2005), ACM, pp. 15–22.

[Gar82] GARGANTINI I.: An effective way to represent quadtrees. *Commun. ACM 25*, 12 (1982), 905–910.

[GJS76] GIBBS N., JR. W. P., STOCKMEYER P.: An algorithm for reducing the bandwidth and profile of a sparse matrix. *SIAM Journal of Numerical Analysis 13*, 2 (1976), 236–250.

[HL09] HUGHES D. M., LIM I. S.: Kd-jump: a path-preserving stackless traversal for faster isosurface raytracing on gpus. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1555–1562.

[HSHH07] HORN D. R., SUGERMAN J., HOUSTON M., HANRAHAN P.: Interactive k-d tree gpu raytracing. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2007), ACM, pp. 167–174.

[Kno06] KNOLL A.: A survey of octree volume rendering techniques. In *GI Lecture Notes in Informatics: Proceedings of 1st IRTG Workshop* (June 2006).

[KT09] KIM B., TSIOTRAS P.: Image segmentation on cell-center sampled quadtree and octree grids. In *SPIE Electronic Imaging / Wavelet Applications in Industrial Processing VI* (2009).

[KWPH06] KNOLL A., WALD I., PARKER S., HANSEN C.: Interactive isosurface ray tracing of large octree volumes. In *Interactive Ray Tracing 2006, IEEE Symposium on* (Sept. 2006), pp. 115–124.

[LCCK02] LEVEN J., CORSO J., COHEN J., KUMAR S.: Interactive visualization of unstructured grids using hierarchical 3d textures. In *VVS '02: Proceedings of the 2002 IEEE symposium on Volume visualization and graphics* (Piscataway, NJ, USA, 2002), IEEE Press, pp. 37–44.

[LGF04] LOSASSO F., GIBOU F., FEDKIW R.: Simulating water and smoke with an octree data structure. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers* (New York, NY, USA, 2004), ACM, pp. 457–462.

[LH06] LEFEBVRE S., HOPPE H.: Perfect spatial hashing. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers* (New York, NY, USA, 2006), ACM, pp. 579–588.

[LST03] LANGBEIN M., SCHEUERMANN G., TRICOCHE X.: An efficient point location method for visualization in large unstructured grids. In *VMV* (2003), pp. 27–35.

[Mor66] MORTON G.: *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. Tech. rep., IBM Ltd., 1966.

[Sam90] SAMET H.: *The design and analysis of spatial data structures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[SCM*09] SONG Y., CHEN W., MACIEJEWSKI R., GAITHER K. P., EBERT D. S.: Bivariate transfer functions on unstructured grids. In *Computer Graphics Forum* (June 2009), IEEE Press.

[VT01] VELASCO F., TORRES J. C.: Cell octrees: A new data structure for volume modeling and visualization. In *VMV '01: Proceedings of the Vision Modeling and Visualization Conference 2001* (2001), Aka GmbH, pp. 151–158.

[WFMS05] WALD I., FRIEDRICH H., MARMITT G., SEIDEL H.-P.: Faster isosurface ray tracing using implicit kd-trees. *IEEE Transactions on Visualization and Computer Graphics 11*, 5 (2005), 562–572. Member-Slusallek, Philipp.

[WVG92] WILHELMS J., VAN GELDER A.: Octrees for faster isosurface generation. *ACM Trans. Graph. 11*, 3 (1992), 201–227.