UC Davis
IDAV Publications

Title
Fast, Memory-Efficient Cell Location in Unstructured Grids for Visualization

Permalink
https://escholarship.org/uc/item/0vq7q87f

Journal
IEEE Transactions on Computer Graphics and Visualization, 16

Authors
Garth, Christoph
Joy, Ken

Publication Date
2010

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Fast, Memory-Efficient Cell Location in Unstructured Grids for Visualization

Christoph Garth and Kenneth I. Joy, *Member, IEEE*

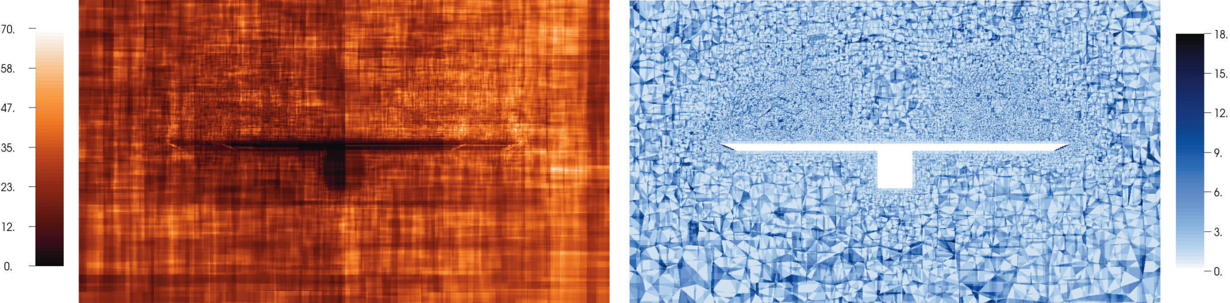


Fig. 1. A visualization of the complexity of unstructured cell location on a plane in a 3D delta wing dataset using the hierarchical scheme developed in this paper. In the left image, the color of each pixel encodes the number of tree leaves overlapping the location point on the plane. The right image maps the number of grid cells that must be tested for point inclusion to locate the correct cell.

Abstract—Applying certain visualization techniques to datasets described on unstructured grids requires the interpolation of variables of interest at arbitrary locations within the dataset's domain of definition. Typical solutions to the problem of finding the grid element enclosing a given interpolation point make use of a variety of spatial subdivision schemes. However, existing solutions are memory-intensive, do not scale well to large grids, or do not work reliably on grids describing complex geometries. In this paper, we propose a data structure and associated construction algorithm for fast cell location in unstructured grids, and apply it to the interpolation problem. Based on the concept of bounding interval hierarchies, the proposed approach is memory-efficient, fast and numerically robust. We examine the performance characteristics of the proposed approach and compare it to existing approaches using a number of benchmark problems related to vector field visualization. Furthermore, we demonstrate that our approach can successfully accommodate large datasets, and discuss application to visualization on both CPUs and GPUs.

Index Terms—Unstructured grids, cell location, interpolation, vector field visualization.

1 Introduction

Among the many different forms of describing the data resulting from scientific simulation, unstructured grids represent one of the most complex and difficult forms. The storage overhead that results from explicitly representing the connectivity of the grid elements or cells is however balanced by the ability to aggressively adapt the spatial resolution of such grids to the complexity of the simulation, resulting in an overall reduction in storage overhead and computational complexity of performing the simulation.

Many visualization algorithms require access to not only the variable values stored at the vertices of such a grid, but apply interpolation to reconstruct continuous fields. This is especially true for vector field visualization, where numerical integration is one of the key techniques to approximate integral curves that model the trajectories of virtual massless particles. This integration procedure is based on the interpolation of a vector field at arbitrary locations inside a dataset's domain of definition. For unstructured grids, this problem is especially hard: for every interpolation point, the cell containing that point must be identified. This *cell location* problem requires the application of spatial data structures that quickly narrow down the range of candidate cells that are then tested for the inclusion of the interpolation point.

• *C. Garth and K. I. Joy are with the Institute of Data Analysis and Visualization at the University of California, Davis, E-mail: {garth|kijoy}@ucdavis.edu.*

Manuscript received 31 March 2010; accepted 1 August 2010; posted online 24 October 2010; mailed on 16 October 2010.
For information on obtaining reprints of this article, please send email to: tvcg@computer.org.

In this context, the challenge for such data structures is twofold: first, they must be able to take into account the greatly varying cell sizes often found in modern adaptive unstructured grids. Second, and more importantly, the memory size of this supporting data structure must not grow beyond a reasonable bound. This latter aspect is especially important when considering the ever-growing sizes of modern simulation datasets. Due to the complexity in both design and implementation of cell location data structures, many interesting methods of vector field visualization are often not designed for or tested against unstructured grids. This especially holds true for GPU-based flow visualization, where the massive computational power available through such devices is applied to provide advanced real-time and interactive visualization tools. Unstructured grids are virtually absent from this innovative field.

In this paper, we present a novel data structure and associated construction scheme for cell location that can accommodate very large and complex unstructured grids and delivers both good performance and memory efficiency. It allows the treatment of unstructured grids with tens to hundreds of millions of unstructured elements of mixed type on workstation-class machines. Furthermore, it is designed for both CPUs and GPUs, and the same basic data structure directly applies to both classes of devices.

The *celltree* data structure we propose (Section 3) is based on the *bounding interval hierarchy* [18, 22] space partitioning scheme, which we supplement with a fast construction algorithm (Section 3.2) using a heuristic to determine good spatial partitions. We examine the performance of our system using a number of benchmarks derived from typical visualization scenarios, and determine the influence of certain parameters of our construction scheme on memory footprint and the cell location speed (Section 5), and provide default parameters that

work well over a wide range of grid sizes and complexities. Moreover, taking advantage of the fact that our data structure can be easily coupled with existing visualization systems, we perform a comparative performance and robustness analysis (Section 6). Finally, we demonstrate how our cell location scheme can be used on GPUs to achieve interactive vector field visualization through massive particle advection (Section 7).

2 Background and Scope

In the following, we assume a basic setting in which an unstructured grid is described in the form of ordered lists of vertices, where cells of different types are defined as a tuple of indices. From the cell type and indices, the topology of the cell is implicitly defined. The cell type also implies an interpolation scheme, or alternatively a set of basis functions, which can be used to translate between (cell-)local and global coordinates. For our purposes, the exact nature of the per-cell interpolant is not pertinent. Global interpolation of variables over the grid domain is then performed by first locating the unique cell that includes the interpolation point, i.e. performing *cell location*, and then interpolating the variable locally within the cell. This setting includes the overwhelming majority of vector-field visualization use cases for datasets produced by unstructured simulation codes. While we concern ourselves with the three-dimensional case here, all considerations in this paper apply equally to two-dimensional datasets.

Among the properties of unstructured grids produced by adaptive simulation codes, the typically highly non-uniform spatial distribution of vertices is one of the complicating factors in cell location. Other problematic aspects derive from the geometric properties of the cells found in such grids: often generated by automated meshing schemes and aggressively adapted to the complexity of the underlying problem, individual cells possess very large aspect ratios or are non-convex (e.g. twisted hexahedra), making the grid complicated to work with from a numerical point of view. Occasionally, numerical oddities such as inverted or overlapping cells resulting from errors in grid generation are encountered in practice.

Performance of cell location data structures is of paramount importance, since variable interpolation is often among the most frequent operations in visualization algorithms. A typical example is integration-based visualization, in which integral curves are constructed by evaluating a vector field repeatedly. While a single integral curve already requires hundreds to tens of thousands evaluations, modern visualization techniques such as integral surfaces [2, 8, 17] or Lagrangian techniques [14, 8] in turn require the computation of thousands to millions of integral curves. Furthermore, data structure construction time and memory overhead should be small, otherwise, a scheme can quickly become infeasible for larger datasets.

We proceed to survey and discuss previous work under these constraints.

2.1 Cell Location

Cell location typically has two stages: first, a spatial data structure is used to quickly narrow down the number of cell candidates; then, each of these candidate cells is checked for inclusion of the interpolation point. This latter operation typically requires the computation of local coordinates and must be considered expensive. Thus, keeping the number of candidate cells small is a crucial property for any cell location scheme. Early approaches to this problem [16, 21] employed uniform spatial subdivision in the form of octrees. Each leaf of the octree stores the indices of cells whose bounding box overlaps with the leaf extents. Leaves are subdivided until either a maximum depth is reached, or, alternatively, if the number of cell indices falls below an upper bound. Cell location then proceeds by traversing the octree from the root and descending through appropriate children until a leaf is reached, which then contains all the candidate cells. This approach does not work well with non-uniform vertex distributions, requiring either too many levels of subdivision and thus a considerable memory overhead, or does not shrink the candidate cell range down to acceptable levels. An implementation of this technique is the default cell location algorithm in the VTK library, and we compare it experimentally with our approach (cf. 6).

Using kd-trees instead of octrees facilitates non-uniform subdivision, at the cost of generally deeper trees and a storage overhead. Recently, Andrysco and Tricoche [1] presented an efficient storage scheme for kd-trees and octrees, termed *Matrix *Trees*, that addresses the overhead of such data structures by employing an efficient storage scheme based on *compressed sparse row* (CSR) storage of tree levels; essentially, the tree is encoded as a sparse matrix in CSR representation. This alleviates most of the memory overhead of kd-trees, and they are able to perform cell location with reduced time and space complexity, and even demonstrate their scheme on GPUs.

The memory efficiency of both kd-trees and octrees suffers from the duplication of indices of cells that overlap multiple leaves of the tree; the corresponding cell index must be stored in each leaf. For large datasets and deep trees, this can often lead to a significant multiplication of the overall number of indices stored, and storage for these lists can exceed the dataset representation in size. To aggravate this problem, the number of duplications is highly dependent on the geometry and adjacency of the grid cells, and thus essentially unpredictable.

An innovative approach to cell location was given by Langbein et al.[12]. They base their spatial subdivision on the vertices of the unstructured grid to quickly locate a grid vertex close to the interpolation point. Making use of cell adjacency information, an indicent cell is identified and a ray is propagated through the grid towards the interpolation point using cell walking. Through clever storage of the cell-vertex incidence information, the storage overhead can be kept reasonable at the cost of a computational overhead. The authors demonstrate successful application of their scheme to a number of large and complex adaptive unstructured grids with several millions of cells on workstation class machines. However, there are a number of shortcomings to this approach. First, it does not work with unstructured grids that contain T-junctions or consist of multiple independent parts with differing vertex indexing, which is often the case for data obtained from parallel simulations involving decomposed grids. In these cases, the cell-vertex incidence cannot be reliably constructed, or only with significant preprocessing effort. Second, this approach has problems in the vicinity of sharp object boundaries, since the ray from the initial point can exit the grid and then has to be restarted from a different point, which requires complicated logic and renewed traversal of the point-based kd-tree. Finally, the separation of cell inclusion tests from cell face intersection tests can result in numerically ambiguous situations; this is especially problematic for cells with high aspect ratio. Last, the storage overhead incurred by the cell adjacency information is still significant, and its construction requires non-negligible computational effort. This approach is implemented in the FAnToM visualization tool [20], and we directly compare the performance against our approach (see Section 6).

2.2 GPU-based Vector Field Visualization

Many interesting papers have been presented in the recent years that take advantage of the massive computational power of GPUs to deliver interactive visualization of vector fields, using e.g. large amounts of particles [4, 3], integral surfaces [2], or dye advection techniques [19]. However, most of these techniques have been pioneered on uniform data representations; and unstructured grids are rarely used in GPU-based vector field visualization. The notable exception is the work by Schirski et al. [15], who presented a GPU-based scheme for particle tracing over purely tetrahedral grids using cell-adjacency based tetrahedral walking; however, locating the initial cell for a particle is still performed on the CPU, and a large overhead is incurred through the cell adjacency storage. Thus, there is a strong limit for the size of grids on which this scheme can be applied.

The *celltree* scheme we present here is directly applicable to GPU-based vector field visualization algorithms. Our scheme works on arbitrary unstructured grids, and no grid transformation such as tetrahedral decomposition is required to treat large grids on the GPU, resulting in storage advantages and improved robustness. The celltree is not geared towards a specific access pattern such as e.g. the cell walking

used in [15] for particle tracing, and can thus support a wide range of visualization algorithms that rely on unstructured grid cell location.

2.3 Ray Tracing

Spatial subdivision data structures play a very prominent role in the field of ray tracing. There, the focus is on accelerating ray-triangle intersection instead of cell location. However, the two problems share some similar traits. The cell location scheme we present here is quite similar to the *skd-tree* approach presented by Wächter and Keller [18] and Zachmann [22], which among the possible candidate data structures presents a good trade-off of memory-efficiency and speed.

Furthermore, the construction of efficient bounding-interval or bounding-volume hierarchies has been investigated thoroughly. For the specific case of ray tracing, optimal spatial partitions are determined using the so-called *surface area heuristic*, first introduced by Goldsmith and Salmon [9]. This heuristic provides a good model of the expected cost of traversing a single node of the hierarchy with a ray for a given choice partition, and various hierarchy construction algorithms have been proposed based on this. An excellent exposition on this topic, together with a number of theoretical considerations, is provided by Havran [10]. The approach presented here is related to some of these ideas.

2.4 Contribution

The *celltree* scheme we present here is based directly on a spatial decomposition using a *bounding interval hierarchy* based on the cell bounding boxes and offers the following advantages over the previous cell location methods in a visualization context:

- It allows general unstructured grids, consisting of multiple independent parts or with non-manifold structure, and can handle arbitrary cell types. Thus, it accommodates a wide range of dataset types and formats.

- It is numerically robust, in the sense that cell location does not suffer from bad numerical properties of individual cells.

- Our scheme does not optimize for a specific access pattern and can support a wide range of algorithms with good performance. Furthermore, it naturally supports the computational parallelism of multi-core CPUs and many-core GPUs.

- Memory overhead can be flexibly balanced against cell location performance, allowing fast cell location on very large unstructured grids with tens to hundreds of millions of cells on workstation-class machines.

In the following, we describe our data structure and the associated construction and traversal algorithms, and conduct a variety of experiments to demonstrate its viability under the stated goals.

3 The Celltree

In the following, we assume that an unstructured grid is given as a list of cells of size N, where for each cell c_i, $i = 1,\dots,N$ the lower bound cmin_i^d and upper bound cmax_i^d are known for each dimension $d = 0,1,2$ or can be determined. This setting is very general and accommodates unstructured grids consisting of multiple parts, and in addition does not constrain the topology of the grid.

3.1 Basic Structure

The *celltree* we describe here is essentially a *bounding interval hierarchy* (cf. [18, 22]) data structure, with an associated construction scheme. A bounding interval hierarchy is a binary tree of axis-aligned boxes, where for each box, its children's boxes are generated from subdividing the parent box along the *split dimension* d. In this respect, it is similar to the kd-tree, and the difference lies in the fact that the child boxes (in the following called left and right) do not form an exact split of the parent box along d. Consequently, two axis-aligned bounding planes must be stored for each inner node of the tree, thus creating a slightly larger per-node storage requirement than kd-trees, which only need to store one split plane.

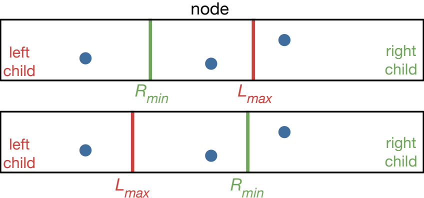


Fig. 2. Different cases when traversing the bounding interval hierarchy. Points left of $L_{\max}$ need to traverse the left subtree, points right of $R_{\min}$ need to traverse the right subtree. In the top sketch, left and right child nodes overlap and both sides must be traversed. At the bottom, traversal stops since the left and right nodes bound a volume of empty space.

We apply this concept to construct a hierarchy through the following procedure:

1. Initialize the tree as a single leaf containing an indexed list of all cells in the dataset.

2. For each leaf node, select a split dimension d to form a disjoint partition of the cell indices contained in it into left and right lists L and R.

3. Attach two children to this node, each containing the corresponding list, and compute the corresponding bounding planes

$$L_{\max} = \max_{i\in L}\{\text{cmax}_i^d\} \qquad \text{and} \qquad R_{\min} = \min_{i\in R}\{\text{cmin}_i^d\}.$$

4. Repeat from 2. until no more leaves can be split.

When compared to construction algorithms for octrees and kd-trees, it is apparent that the lists of cells in left and right children are disjoint; thus such partitioning can be performed in place, since the overall list of cell indices in all leaves is a constant. For kd-trees and octrees, indices for cells that overlap the splitting plane(s) must be stored in both children, creating a multiplication of overall indices stored that can lead to significant overhead. The bounding interval hierarchy avoids this, and thus the size of the resulting data structure is much less dependent on the specific form of a given unstructured grid.

In this setting, to perform cell location, the hierarchy is traversed from the root. At every node encountered, the interpolation point x is compared against both bounding planes. If $x_d \leq L_{\max}$, the left subtree must be traversed, and correspondingly, the right subtree must be traversed if $x_d \geq R_{\min}$. When the traversal reaches a leaf node, the leaf's index list indicates candidate cells that must be checked for point inclusion. If the point is contained in one of the cells, the traversal is terminated and the corresponding index returned. If no cell contains x, the traversal proceeds to other leaves. If no cell is found to contain the point, it is not contained within the domain described by the grid cells. The different cases encountered during the traversal are illustrated in Figure 2.

We note that the traversal of the bounding interval hierarchy is more complicated than that of a tree with exact splits, since both subtrees must be traversed in the case where $L_{\max} \geq x \geq R_{\min}$. Thus recursive traversal or a stack-based approach must be employed. While this appears as a disadvantage on first glance, the traversal can be optimized heuristically (see also Section 4). Furthermore, it can be terminated early in the case where x is contained in neither child node, i.e. $L_{\max} < x < R_{\min}$. We have found this form of empty-space skipping to be beneficial in the vicinity of grid boundaries, where holes in the grid are quickly identified.

As is typical for spatial hierarchies, we note here that if successive interpolations are performed in close spatial proximity, the tree traversal benefits strongly from caching effects since the hierarchy traversal takes a similar path. While one can optimize specifically for specific forms of traversal by including ancestor pointers into the tree, the resulting memory overhead and necessary state (i.e. information where

to restart traversal) make this option unappealing. Furthermore, the stateful nature of restarted traversal complicates parallel interpolation significantly, which is an important consideration e.g. in the context of GPU applications (see Section 7).

We next turn to the question of how to partition the cell list during step 2. of the above algorithm.

3.2 Construction

Choosing a good partition of a node's cell index list I is a crucial step in achieving good performance. Assuming that the cost of a cell-point inclusion test is a unit constant, then the cost of a leaf node for cell location is

$$C_I = N_I,$$

where N_I denotes the number of indices stored in I. If I is partitioned into left and right lists L and R with corresponding children, then the cost changes to

$$C_I = P(L) \cdot N_L + P(R) \cdot N_R + C_{\text{trav}}.$$

where $P(L)$ and $P(R)$ denote the respective probabilities that the interpolation point is contained in L and R, and that descending one level of the hierarchy costs C_{trav}. Under the assumption that point queries are uniformly distributed in space, this can be rewritten as

$$C_I = vol(L) \cdot N_L + vol(R) \cdot N_R + C_{\text{trav}}. \quad (1)$$

Through recursive substitution, one can construct a global cost function C and construct the bounding interval hierarchy such that this function is minimized; however, such global minimization essentially amounts to brute force search which must be considered infeasible. Instead, we choose to neglect C_{trav} and minimize Equation 1 locally upon splitting a leaf node. For our bounding interval hierarchy, it is easily computed that the child volumes $vol(L)$ and $vol(R)$ are proportional to $L_{\max}$ and $R_{\min}$, hence we locally minimize the cost function

$$C = L^d_{\max} \cdot N_L - R^d_{\min} \cdot N_R \quad (2)$$

for each potential split dimension $d = 0, 1, 2$.

Before we consider the practical aspects of cost minimization, we will briefly discuss two common choices for splitting a leaf node in light of Equation 1. The *split-middle* approach divides the cell indices such that cells in L and R are left and right of the center of I's bounding box in the corresponding dimension. While this approximately balances the volume terms, it might result in a strong imbalance of N_L and N_R, such that the overall cost per split node is not minimal and very unbalanced trees can appear in the presence of strong cell size variations. Conversely, the *split-median* approach sorts the cells along the split dimension, and then divides I into two equally sized halves L and R. This ensures a good balance of N_L and N_R and thus balanced trees with minimal depth, but can lead to extensive traversal if $L_{\max} \gg R_{\min}$ since many leaf nodes overlap. Thus, while split-middle and split-median are algorithmically very straightforward, they are not good choices for the general case of highly-adaptive unstructured grids.

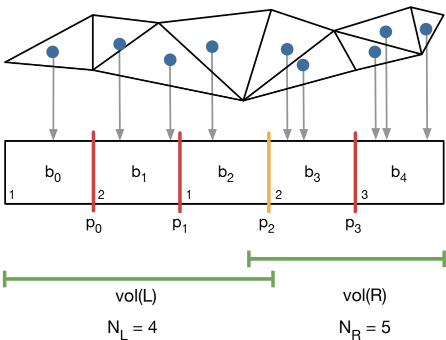


Fig. 3. The fast bucketed split finding algorithm for $n_b = 5$: Cells are classified into buckets according to their bounding box centers. Then the plane p_i is chosen that minimizes $vol(L) \cdot N_L + vol(R) \cdot N_R$. In the shown example, p_2 is the optimal plane. $vol(L)$, $vol(R)$, N_L and N_R can be directly evaluated from accumulating the counters and bounding boxes of the buckets.

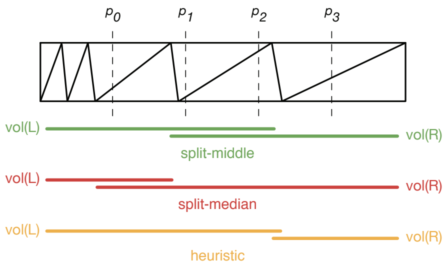


Fig. 4. Split-middle divides the cell into L and R list depending on the position of their center relative to the middle plane p_2, whereas split-median groups the cells in sorted order such the numbers in L and R are equal. The heuristic approach evaluates the cost function to determine the best split plane, here p_2 (with $n_b = 5$). The lines indicate the extent of the cells contained in L and R for each approach, and it is apparent the heuristic approach results in the least overlap – at the cost of balance – for this specific example.

Practically, it is simplifying to postulate a split plane p and then sort cell indices into L and R depending on whether the cell center (or alternatively, its minimum or maximum) is located above or below the split plane. The location of p that minimizes Equation 2 is then determined per dimension, and the (p,d) pair with overall minimal cost is used to perform the split. While the candidate split planes can be narrowed down to the union over all minima and maxima of the bounding boxes of cells contained in I, the resulting list is of size N_I in the general case. Evaluating every one of these split plane locations is computationally intensive, and thus finding a good split plane using this approach is not feasible

Instead, we propose a fast algorithm based on sorting I into a small set of n_b equidistant buckets that span the entire bounding box of I along a dimension d. We traverse the set I exactly once, and classify each cell into a bucket based on the center of its bounding box. Specifically, the bucket index b for the cell with index i is

$$b = \left\lceil n_b \cdot \frac{\text{cmin}^d_i + \text{cmax}^d_i}{I^d_{\min} + I^d_{\max}} \right\rceil - 1,$$

where $I^d_{\min}$ and $I^d_{\max}$ denote the bounds of all cells contained in I. Each bucket records a count and bounding box of the cells that are inserted into it. Then, we evaluate the cost function for the $n_b - 1$ split

$$p^d_j = I^d_{\min} + \frac{j+1}{n_b}\left(I^d_{\max} - I^d_{\min}\right),$$

planes separating the buckets and choose the one that minimizes Equation 2. The effort involved in evaluating C is minimal since it counters and bounding boxes are accumulated over the small ranges of buckets left and right of the p_i. Figure 3 illustrates this construction. Conveniently, this evaluation can be performed simultaneously for $d = 0, 1, 2$ with only a single traversal of I by simply keeping three sets of buckets, one set per dimension.

On occasion, this procedure will not yield a viable splitting plane. For example, in the rare case where all cells fall into one bucket in all dimensions, all cells would be assigned to L and R would be empty. In such cases, we fall back to split-median, ordering the cells by bounding box center. We have observed this case in our tests to occur about once in ten thousand leaf splits, and most frequently in splitting leaves with very few cells. Note that the case $n_b = 2$ corresponds to split-middle exactly. Naturally, one wonders how to choose a value for n_b. We have examined this question empirically in Section 5.

When considering the above scheme, the question arises why the above scheme produces better trees than split-median and split-middle in the general case, since Equation 1 is only evaluated in a small number of fixed locations. In general, we find that optimizing Equation 1 results in minimal overlap between the children of a node; thus, while the resulting trees can be unbalanced, the overall number of leaf nodes whose cells must be tested is much smaller. Figure 4 illustrates this argument on an example configuration.

To allow a degree of control the final size of the celltree, we impose a maximum leaf size s_{max}. Leaves whose size falls below this threshold are not split further. We have found this parameter to be much more effective that prescribing a maximum tree depth, as the latter is too closely tied to the assumption that trees are approximately balanced. Again, we investigate the influence of this parameter in Section 5.

Finally, we wish to point out that since both construction and traversal of the celltree rely solely on relational operations and no arithmetic is involved, the resulting system is very robust and does not suffer from numerical problems in the presence of badly shaped cells.

4 Implementation

Our implementation is a straightforward realization of the concepts described in the previous section. To allow for more space efficiency, we have imposed the following design limitations, aimed at using our cell location scheme on a typical workstation.

4.1 Data Layout and Construction

The bounding interval hierarchy underlying the celltree is a straightforward binary tree, stored in linear array. Each tree node consists of 12 bytes and has the following memory layout (in C notation):

```
struct celltree_node {
  unsigned int dim:   2;
  unsigned int child: 30;

  union {
    struct { float Lmax, Rmin; }        node;
    struct { unsigned int start, size } leaf;
  }
};
```

Since the split dimension is constrained to three-dimensions $(0,1,2)$, the value 3 is used in `dim` to indicate a leaf. We further employ the convention that both children of a node are consecutive in memory, such that only one child index into the tree array must be stored. In the case of an inner node, `lmax` and `rmin` denote the corresponding bounding planes of the children, whereas a leaf node stores a starting index and a size into an array of cell indices, which indicate the cells that belong to the leaf. Note that we intentionally choose single-precision floating point storage for the bounding planes due to decreased storage requirements for the tree nodes. We have verified the absence of numerical issues in this respect on the datasets described below (Section 5.1), and have found virtually no difference between using single or double precision.

Construction of the above structure is straightforward using the algorithm outlined in Section 3.2 and recursive splitting of children. New nodes that result from splitting a leaf are appended to the tree array. Growing this array represents the only memory allocation during the construction phase.

The cell index list is partitioned into two disjoint subarrays for left and right child, and splitting the children can be performed independently. Furthermore, most of the computation time spent goes into scanning the list of cells to determine the best split. This is an ideal scenario for parallel processing, and we perform celltree builds using multithreading to take advantage of multi-core and SMP architectures. Leaf nodes that must be split are recorded in a queue, and multiple threads fetch leaf nodes from this queue, analyze and split the corresponding leaf, and, if further splitting is required, put the children back into the queue. While queue access and tree array modification must be synchronized among the threads such that the structure remains consistent, we have observed only little contention among the threads. This parallel build can be significantly faster than a sequential build (typically 3 - 3.5$\times$ using 4 threads). We expect further improvements by additionally parallelizing the partitioning step for large index list, but have not implemented this in our system.

There is one caveat, however: since multiple threads append to the tree array, the ordering of nodes is no longer deterministic. We address this problem by resorting the tree array after construction such that all nodes at the same tree level are consecutive in the array, and nodes at lower levels precede those at higher levels. This resorting step does not play a significant role in the overall build time. Moreover, we have found that even for sequential, recursive builds, this resorting is slightly advantageous with regard to the overall performance of cell location due to a more balanced memory access pattern.

We employ one further optional optimization. Since the split finding from Section 3.2 heavily relies on the bounding boxes of the cells in the grid, we have found it beneficial to precompute and cache these bounding boxes so that they must not be recomputed repeatedly as a cell is examined during successive splits. We observe build time improvements of $3-4\times$ using cached bounding boxes, however, the memory overhead incurred is non-negligible and may play a role when dealing with very large unstructured grids. Thus, this feature is optional in our implementation.

4.2 Traversal

Traversal of the bounding interval hierarchy is performed using a straightforward stack based implementation, and always starts at the root node. In cases where both subtrees of an internal node must be traversed, one child index is put on the stack and the other child's subtree is traversed. If the traversal ends in a leaf and the point is not contained in any of the leaf's cells, traversal is resumed at the node on top of the stack. However, instead of always traversing either subtree first (e.g. left always before right), we first traverse the subtree whose bounding plane has the larger distance to the sought point. This results in less overall traversal, and the correct cell is identified more quickly.

4.3 Point-in-Cell Test and Interpolation

When traversal reaches a leaf node, the query point is tested for inclusion in the candidate cells contained in the leaf. To determine whether a point is located inside a given cell, the global coordinates of the query point are typically transformed into a cell-local coordinate system where the inclusion test is simplified. For a general treatise of local coordinates and interpolation over different cell types, we refer the reader to the finite element literature, e.g. [5].

Our implementation contains corresponding mapping routines for a number of different cell types commonly found in unstructured grids, such as linear tetrahedra, tri-linear hexahedra, pyramids, and triangular prisms (sometimes called *wedges*). With the exception of tetrahedra for which local coordinates can be found by inverting a 3×3 linear system of equations, we use Newton iteration for numerical robustness and accuracy, in similarity to the VTK library [16]. Since the celltree data structure does not rely on the specific nature of the point-in-cell test, it can accommodate arbitrary cell types, and our implementation could be easily extended to support e.g. higher-order elements or other interpolation schemes that rely on the definition of cells.

5 Experiments

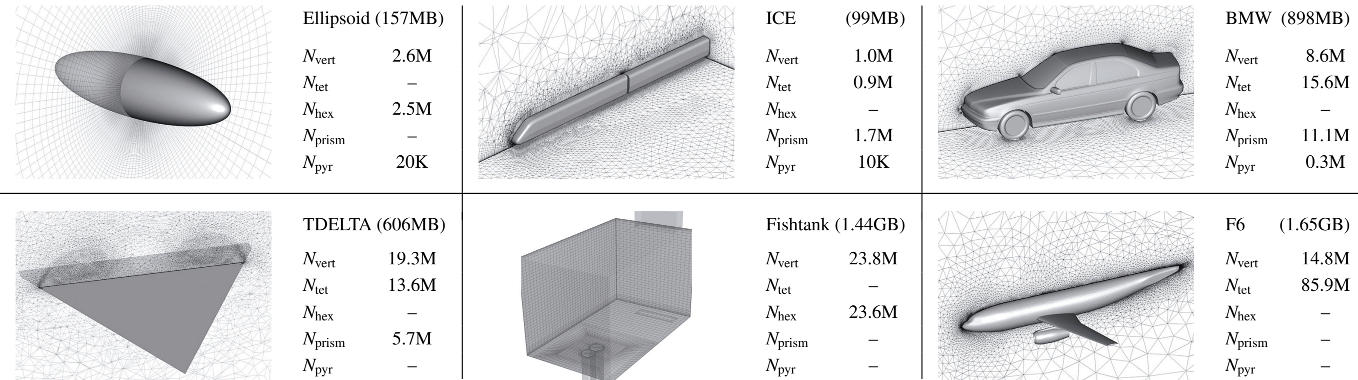
5.1 Datasets

To cover a broad range of unstructured datasets with different properties, we make use of six datasets:

Ellipsoid This smaller dataset models the flow around an ellipsoid; the underlying grid is pseudo-unstructured in the sense that it derived from an originally structured grid over spherical/elliptical coordinates. We have included this case due to its strong variation in cells size but otherwise quite uniform distribution of cells.

ICE and BMW These datasets describe the flow of wind around a high-speed train and a car, respectively. They are of interest to us

Table 1. An overview of the datasets and grid statistics used in the experimental evaluation of the proposed cell location scheme.



	Ellipsoid (157MB)	ICE (99MB)	BMW (898MB)
N_{vert}	2.6M	1.0M	8.6M
N_{tet}	–	0.9M	15.6M
N_{hex}	2.5M	–	–
N_{prism}	–	1.7M	11.1M
N_{pyr}	20K	10K	0.3M

	TDELTA (606MB)	Fishtank (1.44GB)	F6 (1.65GB)
N_{vert}	19.3M	23.8M	14.8M
N_{tet}	13.6M	–	85.9M
N_{hex}	–	23.6M	–
N_{prism}	5.7M	–	–
N_{pyr}	–	–	–

because the contain large boundary layers composed of very thin prism cells, which poses a challenge for cell location schemes.

TDELTA Here, the flow of wind around a delta-shaped wing is modeled. The sharp wing edges constitute a strong concave hole in the grid, and there is a very finely resolved boundary layer above the wing discretized using prisms. Furthermore, the grid is strongly adapted to resolve two large vortical systems above the wing.

Fishtank This dataset studies the turbulent mixing of hot and cold air in a box-shaped domain. This dataset, simulated using a spectral element code [7], is unusual since it represents the union of about 67,000 hexahedral subdomains, each in turn discretized by a 9^3 rectangular structured grid with non-uniform spacing. Each subdomain carries its own vertex enumeration, thus there is no apparent connectivity between the cells of neighboring subdomains. We are especially interested in this grid due to its almost structured nature.

F6 This purely tetrahedral grid is very large and models the flow around an airplane. It shows a strong gradient in cell size as the body of the plane is approached.

Table 1 provides a number of statistics for each dataset. We next describe a number of benchmarks that we use to evaluate and compare the proposed cell location scheme.

5.2 Benchmarks

To evaluate and compare the performance our cell location scheme with real-world visualization applications in mind, we employ the following four benchmarks that represent a mix of typical visualization scenarios that require interpolation:

Random We interpolate one million points that are uniformly distributed over the entire grid domain. The points are pregenerated, and all points lie inside the grid.

Plane We interpolate points on a regular two-dimensional grid. Grid sizes vary per dataset, but the grid is chosen such that it intersects with challenging regions of the dataset's grid, such as e.g. finely discretized boundary layers and grid boundaries.

Volume Similar to Plane, but using a three-dimensional grid.

Streamlines We integrate a number of streamlines using an adaptive integration scheme (DOPRI5, [13]). The number of streamlines is dataset dependent but measures in the thousands. Streamlines are seeded to traverse challenging regions (e.g. flow around embedded bodies), and integration time is chosen sufficiently long that the streamlines traverse a significant portion of the grid.

Note that instead of considering cell location performance in isolation, we have opted to focus on the overall interpolation performance that our approach enables, since the computationally intensive part of unstructured grid interpolation is contained within the computation of local coordinates and the point-in-cell test. Furthermore, this allows us to conduct the data-dependent streamline benchmark which crucially relies on interpolation. For each dataset, we have used the proposed method to perform interpolation of the 3-component velocity field variable.

Instead of examining absolute running times for each of the above benchmarks, we instead count the number of interpolations n_{eval} per benchmark and compute the interpolation rate for each benchmark. In some experiments, we present an average interpolation rate, which is computed by dividing the total number of interpolations for all benchmarks by the sum of the running times. All benchmarks were performed on a workstation with an Intel Core i7 2.66 GHz quad-core processor, equipped with 12GB of RAM.

5.3 Effect of Build Parameters

As promised in Section 3.2, we examine the influence of the build parameters n_b and $s_{\max}$ on the speed of cell location and the time required to build the data structure. Figure 6 illustrates the results for both benchmarks and build times over all datasets as n_b is varied.

It is apparent that n_b has a strong influence on both results. It is interesting to observe that while for some datasets the influence is quite strong, for others it is more diminished. We suppose that this is strongly correlated with inherent symmetries within a dataset that are exploited by a specific choice of n_b. To better understand the reason for these performance differences, we also show the average number of cells traversed per interpolation. Again, as we expect, this graph is strongly correlated with overall interpolation performance. Tree memory usage size was virtually unchanged across all tests and is thus not shown.

Note that the case $n_b = 2$ corresponds exactly to the split-middle approach described above. In all datasets except the Fishtank, which has extremely regular cell distribution, we observe a marked increase for bucket numbers greater than two, of up to 33% for the F6 dataset, which is the largest dataset we have included here. For comparison purposes, we have also included figures for the split-median approach (denoted with "M" in Figure 6). This case is interesting since it generates balanced trees of minimal depth. Here, we find that the cell and node averages for strongly adaptive grids (ICE, BMW, F6) are literally off the charts, with corresponding bad interpolation rates re-

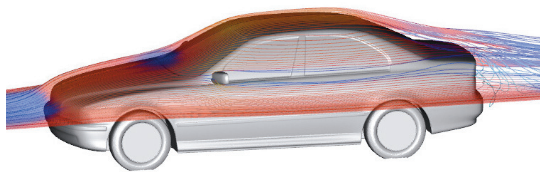


Fig. 5. A subset of streamlines from the corresponding benchmark in the BMW dataset.

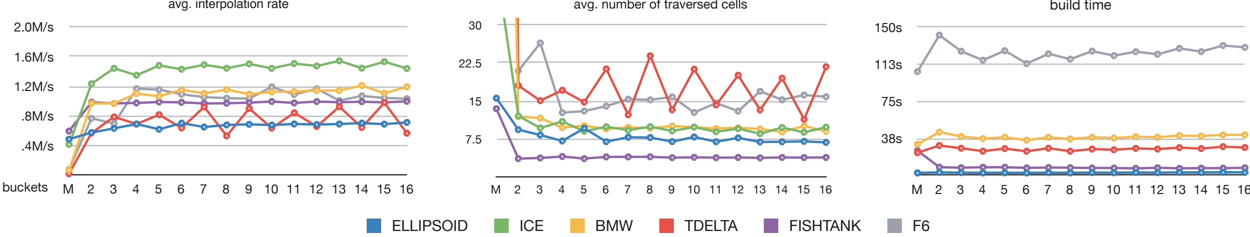


Fig. 6. An illustration of the impact of the number of buckets used in celltree construction on performance and build time.

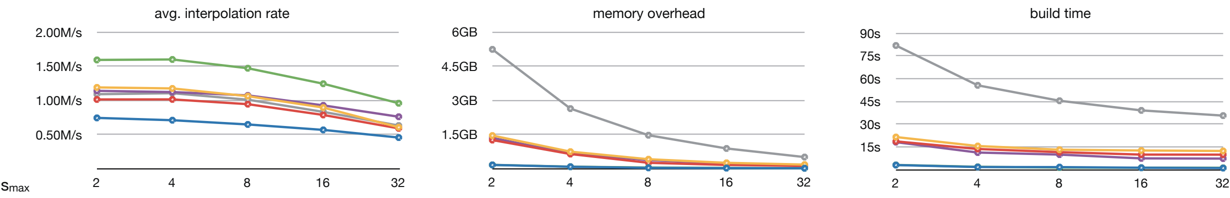


Fig. 7. The maximum leaf size parameter allows a flexible trade-off between interpolation speed and memory overhead. While the overhead drops sharply as the leaf sizes are increased, the interpolation performance degrades gradually.

sulting. Regarding build time, split-median is the clear winner, which is of little surprise since this approach requires the least computational effort. However, we note that it also offers the worst performance; this indicates that tree depth is not a good quality indicator for cell location performance. Otherwise, build performance is not affected strongly by n_b. Overall, we observe good performance values at acceptable build times for $n_b = 5$. We have accepted this value as a default in our construction algorithm and have used it in all successive tests.

Figure 7 depicts the effect of the maximum leaf size $s_{\max}$ on performance, memory size and build time. The figures have ideal shape: while the memory size and build time strongly decrease when doubling $s_{\max}$, performance drops off very gradually, and the performance penalty that is paid for large values of $s_{\max}$ is not overwhelming. This result seems promising for the treatment of larger datasets than the one we have used here, since the overhead incurred by the celltree can be reduced without too much loss of performance. Since we are slightly biased towards treating large datasets, we choose $s_{\max} = 8$ as a default value for our implementation, and all further benchmarks were performed with this value. Figure 1 provides a graphical illustration of the celltree and the cell location statistics discussed here, based on the above default values.

6 Comparative Evaluation

In this section, we compare our cell location schemes against two others. Integrating the celltree data structure into existing visualization frameworks is straightforward; the celltree construction algorithm requires only a method for obtaining the bounding boxes of cells. Note that we make use of the two systems' native routines for all grid and variable access as well as point-in-cell and interpolation routines. This explains the difference in performance and memory usage for identical datasets when comparing Tables 2 and 3.

6.1 VTK

The Visualization Toolkit (VTK, [16]) is a widely used general-purpose framework that allows the rapid development and prototyping of visualization applications, but is mature enough for production applications. Several visualization tools and other frameworks (e.g. VisIt [6] and ParaView [11]) derive from it and make use of its internal pipeline architecture. VTK comes equipped with built in cell location capabilities for unstructured grids, encapsulated in the *vtkCellLocator* class. Internally, it uses an octree-based subdivision approach. In order to test the comparative efficiency of our scheme, we have implemented a *vtkCellTree* class that serves as a drop-in replacement for *vtkCellLocator*.

In addition, recent versions of VTK contain a more modern cell locator in the *vtkModifiedBSPTree* class that implements a modified kd-tree where cells overlapping the split plane are stored in an additional “middle” leaf. While this class is primarily aimed at ray-casting applications, we have nevertheless included it into our benchmarks. To ensure a fair comparison, *vtkCellTree* makes use of VTK's own cell interpolation and inclusion test routines. It is only the candidate search that is performed using the celltree data structure. Cell tree builds were performed serially, and all three classes were measured based on their default parameters.

The results of our tests, using the benchmarks described in Section 5.2, are summarized in Table 2. Some benchmarks did not complete within reasonable time (10 minutes), thus some timings do not exist. For this reason we choose not to aggregate the performance of the individual benchmarks. We find that in most benchmarks, we generally outperform the faster of *vtkModifiedBSPTree* and *vtkCellLocator* by a significant margin, while at the same time being more memory-efficient by a factor of 3–5. For the TDELTA, Fishtank and F6 datasets, the cell tree was the only locator to complete all benchmarks.

The good comparative performance of vtkCellLocator on the “Random” benchmark is explained by the fact that the corresponding interpolation points are uniformly distributed across the grid domain, and thus only rarely fall into complex grid regions. In all other benchmarks, we observe a strong advantage of our scheme in the presence of strongly adaptive grids.

6.2 FAnToM

The FAnToM visualization tool [20], developed at the University of Leipzig, implements the innovative approach described by Langbein et al. [12]. While not widely in use, its handling of large unstructured grids is considered state-of-the-art, thus it is an ideal candidate for comparison here to ensure the practical relevance of our results. Similar to the VTK comparison above, cell location functionality is encapsulated in a dedicated class (*FCellLocator*), for which we have again provided a drop-in replacement (*FCellTree*) that replaces only the cell candidate search with our approach. The benchmarks were run for both locators, with the same restrictions as above (serial celltree build, default parameters), and the results are documented in Table 3.

The default *FCellLocator* class already offers good performance for smaller grids (Ellipsoid, ICE); however, the celltree possesses a strong advantage in memory overhead. It is interesting to note that while the performance numbers for streamline integration are very similar, the celltree has significantly better performance in the “Plane” and

Table 2. Results of the comparison of the celltree data structure against VTK's built-in locators.

Dataset	Locator	Build time	Memory Overhead	Random	Plane	Volume	Streamlines
Ellipsoid	*vtkCellTree*	3.34s	22MB (8%)	4.32s	1.91s	11.58s	1.70s
	vtkCellLocator	0.61s	90MB (34%)	7.71s	75.68s	559.98	–
	vtkModifiedBSPTree	6.72s	236MB (91%)	30.37s	1.90s	116.33s	46.39s
ICE	*vtkCellTree*	3.27s	23MB (13%)	2.51s	1.68s	6.42s	2.66s
	vtkCellLocator	0.53s	115MB (65%)	5.95s	89.93s	57.65s	211.45s
	vtkModifiedBSPTree	7.65s	246MB (140%)	16.90s	13.43s	69.67s	43.57s
BMW	*vtkCellTree*	40.25s	229MB (14%)	2.34s	3.57s	8.94s	1.97s
	vtkCellLocator	5.01s	921MB (56%)	5.15s	–	–	–
	vtkModifiedBSPTree	303.23s	2476MB (151%)	–	–	–	–
TDELTA	*vtkCellTree*	28.08s	165MB (14%)	1.66s	4.65s	9.48s	25.33s
	vtkCellLocator	4.2s	880MB (79%)	3.99s	–	–	–
	vtkModifiedBSPTree	77.57s	1770MB (159%)	61.86s	–	–	–
Fishtank	*vtkCellTree*	27.59s	196MB (8%)	3.52s	3.79s	6.85s	27.11s
	vtkCellLocator	6.16s	851MB (35%)	7.74s	12.04s	40.04s	28.75s
	vtkModifiedBSPTree	199.41s	2162MB (91%)	–	–	–	–
F6	*vtkCellTree*	130.19s	743MB (16%)	1.59s	8.96s	17.63s	9.60s
	vtkCellLocator	22.40s	5426MB (124.54%)	5.80s	–	–	–
	vtkModifiedBSPTree	–	–	–	–	–	–

Table 3. Results of the comparison of the celltree data structure against FAnToM's built-in locator.

Dataset	Locator	Build time	Memory Overhead	Random	Plane	Volume	Streamlines
Ellipsoid	*FCellTree*	5.19s	22MB (18%)	24.08s	2.28s	29.88s	0.83s
	FCellLocator	6.41s	150MB (76%)	21.17s	3.25s	30.14s	0.88s
ICE	*FCellTree*	4.93s	23MB (6%)	4.01s	1.93s	9.10s	3.22s
	FCellLocator	3.06s	88MB (85%)	11.87	37.81s	26.62s	4.32s
BMW	*FCellTree*	57.84s	229MB (10%)	7.91s	4.91s	21.36s	4.06s
	FCellLocator	24.56s	770MB (122%)	12.48s	51.88s	91.08s	4.01s
TDELTA	*FCellTree*	40.03	165MB (11%)	5.25s	7.33s	24.50s	5.11s
	FCellLocator	15.2s	483MB (82%)	11.84s	290.65s	–	6.33s
Fishtank	*FCellTree*	43.28s	196MB (5%)	14.37s	5.19s	24.89s	6.93s
	FCellLocator	–	–	–	–	–	–
F6	*FCellTree*	144.92s	734MB (56%)	2.55s	7.59s	25.72s	3.93s
	FCellLocator	40.66s	1633MB (81%)	12.04s	119.99s	132.23s	4.36s

"Volume" benchmarks that stress interpolation in complicated grid regions. Here, the increased robustness of our scheme is especially apparent. Note that the Fishtank dataset could not be benchmarked with the default locator since it does not have consistent cell adjacency (cf. Section 5.1).

7 GPU Interpolation

The celltree data structure, consisting of essentially two linear arrays containing the tree nodes and cell indices, directly allows the use of arbitrary unstructured grid interpolation on modern GPUs. To this purpose, we have implemented a prototype system based on the CUDA programming language, which uses essentially identical celltree traversal code to the CPU version. In the following, we discuss a number of possible minor modifications of the overall data structure and share some observations that affect the performance of the interpolation procedure on GPUs.

7.1 Cell Inlining

While GPUs attain impressive memory transfer rates, they are only achievable for specific memory access patterns. General unstructured grid representations contain a number of indirections that typically do not work well with GPU memory architectures. For example, cells are represented as a list of $(start, type)$ pairs, and to obtain the locations of all vertices of a cell requires indexing into a cell index array first, and the resulting indices must be used to fetch the actual coordinates. The celltree data structure adds an additional level of indirection, since candidate cell indices in the leaves must be traced to $(start, type)$ pairs.

To remove two levels of indirection, we propose the following storage layout as an ideal unstructured grid representation in conjunction with the celltree data structure on GPUs. Starting with the observation that currently available GPU configurations do not possess enough storage to handle unstructured grids approaching one billion vertices, we make the assumption that 30 bits are sufficient to represent a vertex index. Further noting that the minimum number of indices in any cell is 4 (tetrahedron), we replace each cell index in the leaf list by the full representation of the cell, consisting of all indices, and encoding the cell type in the most significant two bits of the first vertex. Correspondingly, the start indices in the celltree leaf nodes are adjusted to reflect the different starting offset in the cell leaf lists. Upon reaching a leaf during celltree traversal, the first vertex index is read from the leaf list, and the cell type is computed from the two most significant bits. Then, a corresponding number of further indices is read from the list, until the number of indices for the indicated cell type is complete.

This approach yields a more compact representation of the combination of grid and locator data structure, and removes two memory indirections at the lowest level of the interpolation procedure. While the number of possible cell types is reduced to four in this approach, it is readily generalized to include more cell types by encoding the type in the unused bits of multiple indices. Table 4 provides an overview of the memory savings achieved by cell inlining.

7.2 Tree Traversal

Celltree traversal is accomplished very similar to the CPU implementation, with the only difference being a fixed-size stack with 64 entries. We estimate that this stack size is sufficient to treat unstructured grids up to current-generation GPU memory capacity. The stack resides in local memory, which causes some performance concern, however, we observe that the stack is rarely read from in the datasets and cases we tested, and thus the performance impact is negligible.

7.3 Cell Interpolation

In our tests, we employ identical routines to the CPU case to achieve cell interpolation and inclusion tests. However, the high arithmetic intensity of the Newton iteration generates high register pressure and precludes large thread block sizes. Thus, if speed is desired over accuracy and strongly nonlinear cells are not present, we propose to decompose cells on-the-fly into tetrahedra, which can be interpolated with much less effort.

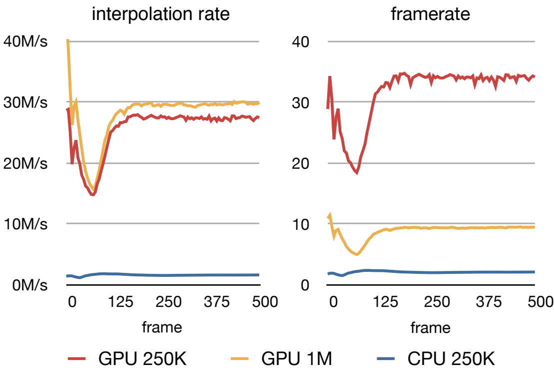


Fig. 8. Interpolation and frame rates for the GPU benchmark.

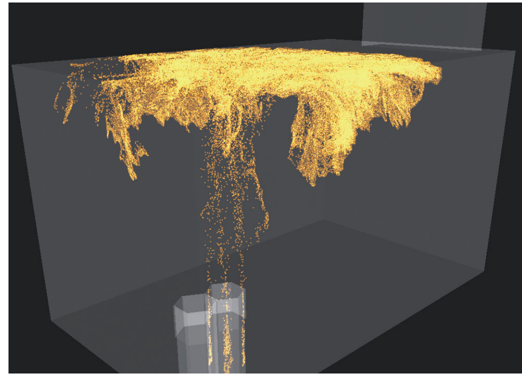


Fig. 9. Interactive advection of 1 million particles on the GPU in an unstructured grid with 23.6 million hexahedra.

7.4 Examples

In order to demonstrate the suitability of our scheme for interactive vector field visualization over unstructured grids, we implemented a particle advection scheme based on a third-order Runge-Kutta method similar to previous approaches on regular grids [4, 3] and applied it to visualize particle movement for the datasets described in Section 5.1.

Figure 8 shows the measured frame rates and corresponding interpolation rates for an advection of 250K and 1M particles over 500 frames in the Fishtank dataset (illustrated in Figure 9). This dataset is ideal for testing, since coherently moving particles seeded from the inlet quickly diverge and traverse different parts of the dataset, thus reducing the amount of coherence. The test was run on an NVidia GeForce 285GTX with 2GB VRAM. The total GPU memory consumption for the combined celltree and grid representation as described above is 1377MB, which is equivalent to an overhead of 8% when compared to the original dataset size (cf. Table 1). The interpolation rate, i.e. the number of interpolations performed per second, is around 26-30 million interpolations per second, after a dip that results from the particles traversing a more highly resolved region of the grid. Overall, we observe that the initially close particle locations translate into coherent memory access since the hierarchy traversal only diverges near the bottom. This effect is reduced after the particles spread out. Since the overhead for rendering the particles is small, the interpolation rate roughly translates to frames per second by taking into account that three interpolations are performed per particle per frame. For comparison, the figure also includes the performance achieved by our CPU implementation running an identical test with 250K particles. Overall, we note an average speedup of 16.5× for the GPU implementation in this example.

Table 4 shows the sizes of the GPU celltree structure and the velocity variable. With the exception of F6, for which we choose $s_{\max} = 16$, all datasets fit into GPU memory with the default parameters and show performance comparable to the Fishtank. Overall, we find that GPU performance characteristics are very similar to those for CPUs (cf. Section 5). Please refer to the accompanying video for an illustration of these examples.

We wish to emphasize the fact that our implementation should be taken as a proof of concept. Since we obtained good results even without optimizing for the GPU (or even a specific GPU architecture), we expect that performance can be improved significantly. Since coherent memory access patterns are a crucial factor in this setting, an investigation of hierarchy memory layout and grid vertex ordering is planned for future work; however, the chosen example of particle tracing illustrates that coherent interpolation patterns cannot be expected for the general case. We finally note that the routines for point-in-cell testing are arithmetically complex and account for 90% of the register and instruction count in our implementation and present additional optimization potential. We attribute the good performance of the present implementation to the fact that our data structure is designed to use these routines only a small number of times per interpolation procedure.

8 Conclusion

We have presented a novel cell location scheme, the *celltree*, with associated construction and traversal algorithms. The proposed scheme is fast, robust, flexible and memory-efficient, and can be used for interpolation over unstructured grids with good performance. Due to the flexibility and ease of use of the celltree data structure, we were able to devise drop-in implementations for two established and widely used visualization systems, VTK and FAnToM, and found overall performance and robustness increased while memory overhead was reduced. Furthermore, we have discussed and demonstrated a fully featured prototype GPU implementation.

Porting the celltree data structure to the CUDA environment for GPU computing was straightforward, and despite not performing any low-level GPU-specific optimizations, we obtain good performance that should help enable the use of general unstructured grids for GPU-based visualization applications. We are currently looking into further implementations targeted at the OpenGL shading language, to enable unstructured interpolation directly from the rendering phase of visualization applications. We are also interested in exploring our ideas in the context of the OpenCL framework. Furthermore, we are investigating the possibility of extending our approach to support parallel, distributed interpolation over decomposed unstructured grids on clusters.

Acknowledgments

The authors wish to thank Paul Fischer and Aleksandr Obabko, Argonne National Laboratory, and Markus Rütten, DLR Göttingen, for supplying the benchmark data sets, and Hank Childs, Xavier Tricoche and Max Langbein for insightful discussion. We are also grateful to our colleagues at the Institute for Data Analysis and Visualization and to Xavier Tricoche at Purdue University for discussion and feedback. This work was supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. Department of Energy under Contract No. DE-FC02-06ER25780 through the Scientific Discovery through Advanced Computing (SciDAC) programs Visualization and Analytics Center for Enabling Technologies (VACET).

Table 4. Memory sizes for the inlined celltree data structure used for GPU interpolation of the test datasets.

Dataset	Grid + Velocity	Inlined Celltree + Velocity
Ellipsoid	156MB	149MB
ICE	99MB	92MB
BMW	898MB	821MB
TDELTA	606MB	553MB
Fishtank	1446MB	1377MB
F6	2138MB	1848MB (with $s_{\max} = 12$)

REFERENCES

[1] N. Andrysco and X. Tricoche. Matrix *trees. *Computer Graphics Forum*, 29(3), 2010 (to appear).

[2] K. Bürger, F. Ferstl, H. Theisel, and R. Westermann. Interactive streak surface visualization on the gpu. *IEEE Transactions on Visualization and Computer Graphics*, 15:1259–1266, 2009.

[3] K. Bürger, P. Kondratieva, J. Krüger, and R. Westermann. Importance-driven particle techniques for flow visualization. In *Proceedings of IEEE VGTC Pacific Visualization Symposium*, 2008.

[4] K. Bürger, J. Schneider, P. Kondratieva, J. Krüger, and R. Westermann. Interactive visual exploration of instationary 3D-flows. In *Eurographics/IEEE VGTC Symposium on Visualization (EuroVis), to appear*, 2007.

[5] G. F. Carey and J. T. Oden. *Finite Elements: A Second Course.* Prentice-Hall, Englewood Cliffs, NJ, 1983.

[6] H. Childs, E. S. Brugger, K. S. Bonnell, J. S. Meredith, M. Miller, B. J. Whitlock, and N. Max. A contract-based system for large data visualization. In *Proceedings of IEEE Visualization 2005*, pages 190–198, 2005.

[7] P. Fischer, J. Lottes, D. Pointer, and A. Siegel. Petascale algorithms for reactor hydrodynamics. *Journal of Physics: Conference Series*, 125:1–5, 2008.

[8] C. Garth, H. Krishnan, X. Tricoche, T. Tricoche, and K. I. Joy. Generation of accurate integral surfaces in time-dependent vector fields. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1404–1411, 2008.

[9] J. Goldsmith and J. Salmon. Automatic creation of object hierarchies for ray tracing. *IEEE Computer Graphics and Applications*, 7(5):14–20, 1987.

[10] V. Havran. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University, Prague, 2001.

[11] A. Henderson. *ParaView Guide. A Parallel Visualization Application.* Kitware Inc., 2007.

[12] M. Langbein, G. Scheuermann, and X. Tricoche. An efficient point location method for visualization in large unstructured grids. In *Proceedings of Vision, Modeling, Visualization*, 2003.

[13] P. J. Prince and J. R. Dormand. High order embedded runge-kutta formulae. *Journal of Computational and Applied Mathematics*, 7(1), 1981.

[14] F. Sadlo and R. Peikert. Visualizing lagrangian coherent structures and comparison to vector field topology. In *Topology-Based Methods in Visualization II*, Mathematics and Visualization, pages 15–29. Springer, Berlin Heidelberg, 2007.

[15] M. Schirski, C. Bischof, and T. Kuhlen. Interactive Particle Tracing on Tetrahedral Grids Using the GPU. In *Proceedings of Vision, Modeling, and Visualization (VMV) 2006*, pages 153–160, 2006.

[16] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*. Prentice Hall, 1997.

[17] W. von Funck, T. Weinkauf, H. Theisel, and H.-P. Seidel. Smoke surfaces: An interactive flow visualization technique inspired by real-world flow experiments. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1396–1403, 2008.

[18] C. Wächter and A. Keller. Instant ray tracing: The bounding interval hierarchy. In *Proceedings of the 17th Eurographics Symposium on Rendering*, 2006.

[19] D. Weiskopf and T. Ertl. GPU-Based 3D Texture Advection for the Visualization of Unsteady Flow Fields. In *In WSCG 2004 Conference Proceedings, Short Papers*, pages 259–266, 2004.

[20] A. Wiebel, C. Garth, M. Hlawitschka, T. Wischgoll, and G. Scheuermann. Fantom - lessons learned from design, implementation, administration, and use of a visualization system for over 10 years. In *In Refactoring Visualization from Experience (ReVisE) 2009, co-located with IEEE Visualization*, 2009.

[21] J. Wilhelms and A. van Gelder. Octrees for faster isosurface generation. *ACM Transactions of Graphics*, 11(3):201–227, 1992.

[22] G. Zachmann. Minimal hierarchical collision detection. In *Proc. ACM Symposium on Virtual Reality Software and Technology (VRST)*, pages 121–128, Hong Kong, China, Nov.11–13 2002.