Out-of-Core Visualization of Time-Varying Hybrid-Grid Volume Data

Min Shih* University of California, Davis
Yubo Zhang† University of California, Davis
Kwan-Liu Ma‡ University of California, Davis
Jayanarayanan Sitaraman§ University of Wyoming
Dimitri Mavriplis¶ University of Wyoming

ABSTRACT

Traditional computational fluid dynamics (CFD) solvers are usually written for a single gridding paradigm such as structured-Cartesian, structured-body-fitted, or unstructured grids. Each type of mesh paradigms has inherent advantages and disadvantages. Thus, the methods of coupling multiple mesh paradigms have been developed to facilitate the use of different solvers in different part of the computational domain. However, the complex hybrid gridding paradigm poses challenges to rendering calculations for visualizing the data. This paper describes a volume visualization system for time-varying adaptive moving-body CFD datasets, where the grid system consists of unstructured grids near the body surface, coupled with Structured Adaptive Mesh Refinement (SAMR) grid in the off-body domain. We present two approaches to the hybrid-grid volume ray casting: a KD-tree based single-pass algorithm, and a multi-pass algorithm using the depth peeling technique. The system has a three-level memory hierarchy: GPU memory, main memory, and a solid state drive (SSD). Through data caching and prefetching within the memory hierarchy, the latency of time-step swapping can be hidden. Experimental results show that our system allows interactive volume exploration on single-GPU commodity PCs.

1 INTRODUCTION

In scientific computing, physics models are approximated using numerical methods, where the simulation domains are typically discretized using different types of meshes depending on the numerical scheme and the domain shapes. For example, structured (Cartesian/curvilinear) meshes and unstructured (tetrahedral/prismatic/hexahedral) meshes are commonly used in computational fluid dynamics (CFD). Each mesh paradigm has inherent advantages and disadvantages. Cartesian grids are easy to generate, to adapt, and are computationally efficient in general, but they are not suited for resolving boundary layers around complex geometries. On the other hand, structured curvilinear grids and unstructured grids work well for resolving boundary layers, but the generation of curvilinear grids for complex geometries is tedious, and unstructured grids often suffer from less computational efficiency. Therefore, today, the tendency is to couple different mesh paradigms so that different types of mesh are used in different part of the computational domain. HELIOS [18] [20] [26] is one of the computation software packages following this strategy.

In HELIOS datasets, unstructured near-body meshes are used to cover dynamic irregular boundary regions around the moving body, and adaptive Cartesian off-body grids, which follow Structured Adaptive Mesh Refinement [1] scheme, are used to fill the rest. The time-varying hybrid-grid data introduces new challenges to direct volume rendering (DVR) including efficient cell access to overlapped hybrid meshes and optimized data transfer in multi-level memory hierarchy. To our knowledge, there are no published papers addressing the difficulties encountered from this kind of data.

In this paper, we present a volume renderer for the time-varying hybrid-grid HELIOS datasets. Two different volume rendering approaches to the hybrid-grid data are introduced and compared. The first approach is a single-pass method which employs a hybrid KD-tree structure for fast cell location within the ray traversal process. In the second approach, the near-body unstructured grid and the off-body AMR grid are rendered in a multi-pass fashion with a modified depth peeling technique. In addition, we introduce an optimized data streaming strategy for the out-of-core volume rendering of hybrid-grid data to hide the latency while producing animations and to improve the user experience. The renderer allows user to interactively explore the data in both spatial and temporal domain.

2 RELATED WORK

Our hybrid-grid volume rendering technique involves time-varying unstructured-grid data and AMR data. In this section, we review previous work on GPU accelerated unstructured grid and AMR volume rendering, as well as time-varying volume visualization techniques.

AMR Volume Rendering For AMR data, Kähler and Abel presented a single-pass ray-casting technique on GPU [11]. To accelerate data access, a KD-tree approach is adopted in their implementation. Leaf et al. presented a cluster-parallel GPU volume rendering technique for large-scale AMR data [13]. The data volume is divided into convexly-bounded chunks for load-balancing.

Unstructured Grid Volume Rendering The Projected Tetrahedra (PT) algorithm introduced by Shirley and Tuchman [19] is a cell projection approach, which decomposes a projected tetrahedron into triangles in order to benefit from hardware rasterization. The PT algorithm needs a visibility ordering of the cells. A variety of sorting approaches have been developed [4] [16]. However, for rasterization-based GPU techniques, interactivity degrades significantly for large datasets. Also, the proposed sorting methods are hard to use directly on the hybrid-grid data handled by this paper. On the other hand, Weiler et al. presented the first implementation of GPU accelerated volume ray-casting for tetrahedral meshes [24]. To render non-convex tetrahedral meshes on GPU, the depth peeling algorithm is adopted in [25]. Muigg et al. introduced a focus+context visualization technique for unstructured volume data [17], where important regions are maintained as unstructured bricks and other regions are resampled using structured bricks, and the volume is then rendered in a hybrid ray-casting manner.

Time-varying Volume Visualization A survey of time-varying data visualization strategies is given in Ma's paper [15]. Chiang proposed an out-of-core isosurface extraction method for time-varying irregular grids [5]. Bernardon et al. presented an interactive volume renderer for time-varying unstructured-grid data [2].

*e-mail: minshih@ucdavis.edu
†e-mail: ybzhang@ucdavis.edu
‡e-mail: ma@cs.ucdavis.edu
§e-mail: jsitaram@uwyo.edu
¶e-mail: mavripl@uwyo.edu

IEEE Symposium on Large Data Analysis and Visualization 2014
October 9–10, Paris, France
978-1-4799-5215-1/14/$31.00 ©2014 IEEE

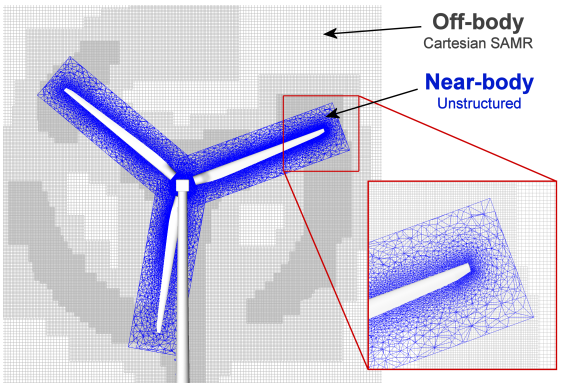


Figure 1: Hybrid grid. Unstructured grids capture boundary layers around the geometry and adaptive Cartesian grids capture far-field flow features.

Their work focuses on the time-varying scalar fields on static geometry and topology. Wald et al. adopted direct ray-tracing for rendering isosurfaces in time-varying tetrahedral volume data [22]. Gosink et al. introduced a GPU-accelerated query-driven visualization technique for time-varying AMR data [9]. Wang et al. presented a compression technique for rendering time-varying volume data [23], which takes scientists' domain knowledge into account. Hadwiger et al. introduced an interactive volume exploration technique for peta-scale time-varying volume data [10].

3 Hybrid-Grid Data

The HELIOS datasets we deal with in this paper involve aerodynamic flow simulations of time-dependent moving-bodies. The mesh paradigm in HELIOS datasets consists of separate *near-body* and *off-body* grid systems. The near-body grid is an unstructured mixed-element (tetrahedra, prisms, pyramids, and hexahedra) grid, while the off-body grid is an adaptive multi-level Cartesian grid, as shown in Figure 1.

The unstructured near-body grid extends a short distance from the body. The reason for using an unstructured grid in the near-body region is to capture the geometry and viscous boundary layer effects, which are difficult to capture with Cartesian grids alone. To tightly fit complex geometry or separated bodies, the near-body grid can have multiple disconnected blocks overlapping each other. At a short distance from the body, in the regions where the near-body grid overlaps off-body grid, the solution of the near-body grid is interpolated onto the off-body Cartesian grid as Dirichlet boundary conditions.

The off-body grid follows Structured Adaptive Mesh Refinement (SAMR) scheme, where adaptive Cartesian grids capture far-field flow features. The levels are generated from coarsest to finest, where the coarsest level defines the physical extent of the computational domain. For each step of the simulation, the physical quantities and gradients are evaluated at each Cartesian cell, and the cells holding values requiring refinement are marked. These marked cells are clustered into a set of rectangular grid patches, called *subgrids*, forming the new finer level.

4 System Overview

Figure 2 depicts an overview of our system. In regard to our datasets, a single time-step can fit into GPU memory. However, the total size of the whole time sequence is too large, and the dataset has to be stored out-of-core. We place the data on a high speed solid state drive (SSD), which is much faster than traditional hard drives, making the system capable of responding in a short time when user switches between the time-steps. In order to improve the interactivity of the visualization system, it is beneficial if the data preparation latency can be further shortened. To meet this need, we perform a one-time preprocessing to the raw data to minimize the data processing time at runtime. Also, a data caching and prefetching strategy within the memory hierarchy is used in the runtime in order to minimize the data loading latency.

At the runtime, processed data is read from the SSD and loaded into GPU memory for rendering. We implemented two different volume rendering methods: a single-pass ray casting approach and a multi-pass ray casting approach. Both approaches involve two main tasks: unstructured-grid rendering for the near-body grid, and AMR rendering for the off-body grid. Our unstructured-grid renderer adopts the tetrahedral cell walking algorithm, which utilizes the connectivity information of the grid. The AMR renderer implementation is based on the single-pass AMR ray casting algorithm presented by Kähler et al. [11]. The approach employs a KD-tree that partitions the data domain into axis-aligned, non-overlapping blocks of cells of the same resolution level, enabling processing the subgrids in front-to-back order for each view ray. Kähler et al. discussed two different GPU data storage strategies, namely the *texture packing* approach and the *bindless texture* (or *texture object* in CUDA) approach. We also implemented and examined the two approaches for the out-of-core hybrid-grid rendering.

In our ray casting algorithms, one of the critical problems is to find the intersection of the viewing ray and the boundary of the unstructured near-body grid, so that we can initiate the cell walking process. However, the characteristics of the HELIOS hybrid-grid datasets make this task non-trivial. To achieve this, our single-pass method uses a hybrid KD-tree structure, which works for both AMR ray traversal and unstructured-grid cell location. The multi-pass approach, on the other hand, uses a modified depth peeling method to render the near-body grid and off-body grid separately. More details are discussed in Section 6.2.

5 Preprocessing

In order to accelerate time-step swapping in the runtime, we perform a one-time preprocessing to the raw data and store the results on the fast SSD. The preprocessing covers most of time-consuming data preparation tasks. The processed data for each time-step includes following items:

Off-Body Grid Data The off-body grid data consists of a set of AMR subgrids. The data values as well as the information about the subgrids (including the dimensions, layer, and the physical domain represented by the subgrid) are stored.

Near-Body Grid Data The processed near-body grid data includes the vertex list, tetrahedron list, connectivity list, data values, and the gradients.

For the convenience of rendering, the various types of cells (prisms, pyramids, hexahedra) of the original unstructured grid are subdivided into tetrahedral cells. In the cell walking process of the tetrahedral grid ray casting algorithm, the cell-to-cell connectivity information is necessary. We construct the connectivity list where each element stores the indices of the four cells adjacent to the four faces respectively of the corresponding tetrahedral cell. For the boundary cells, the adjacent index of the boundary faces are set to -1, so that we can determine if the viewing ray has left the mesh.

Gradient estimation is necessary for the gradient based Blinn-Phong shading model. Since on-the-fly gradient estimation for unstructured grids is expensive, we perform the per-vertex gradient estimation in the preprocessing stage. A regression based method is used for the gradient estimation. For more details about gradient estimation on unstructured grids, we refer our readers to [6].

In our datasets, the topology of the mesh does not change much while the time step changes. Therefore, the tetrahedron list and the connectivity list can be stored in a compact way: we only store the

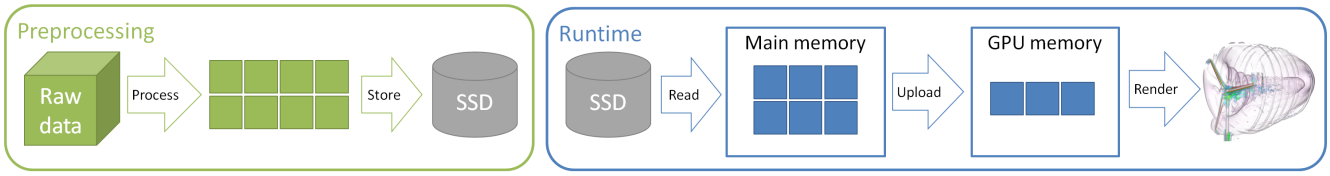


Figure 2: System overview.

complete list of the first time-step, while for the remaining time-steps, only the differences between the step and the first step are stored.

Hybrid KD-Tree The hybrid KD-tree plays an important role in our single-pass ray casting algorithm. The KD-tree must fulfill two main requirements: 1) With the KD-tree, we should be able to perform a front-to-back visit to the AMR subgrids, and 2) given a point within the computational domain, we should be able to efficiently determine if the point lies inside a near-body unstructured grid and locate the grid cell that contains the point.

The KD-tree is constructed with a two-stage process: In the first stage, we follow the strategy suggested in [11] to build a KD-tree which partitions the computational domain covered by the off-body AMR subgrids into non-overlapping blocks of cells of the same resolution level. Algorithm 1 summarizes the procedure. In the second stage, we find the leaf nodes generated in the previous stage in which the domain overlaps the near-body grid, and further decompose these leaf nodes for cell location purpose. Here we use a point-based KD-tree construction scheme: the center points of the near-body grid cells are used to guide the node subdivision. The node containing cell-center points are subdivided recursively until the number of cell-center points is less than a predefined threshold. Through selecting the threshold value, we can control the size of the constructed tree. Whenever the subdivision stops, the resulting leaf node stores the index of one of the cells contained in the node. The KD-nodes are stored in a compact layout: one floating-point value and two 32-bit integer values per node. The floating-point value is used to store the position of the splitting plane. The first integer value is divided into two parts: Two bits are used to indicate the splitting axis (value 0-2) for inner nodes. If the node is a leaf, the value is set to 3. The remaining bits are used to store the ID of the AMR subgrid. The second integer value stores the offset to the children when the node is an inner node or the ID of the tetrahedral cell when the node is a leaf.

Algorithm 1 Hybrid KD-tree construction, stage 1

```
1:  set the domain of the root node of the KD-tree to the bounding
    box of all subgrids on the root level of the AMR hierarchy;
2:  for i ← 0 .. finest level of the AMR do
3:      for each leaf node n in the current KD-tree do
4:          slist ← all the subgrids in level i that overlap node n;
5:          SplitLeaf(n, slist);
6:      end for
7:  end for
8:
9:  function SPLITLEAF(n, slist)
10:     if slist.empty() or slist.count() = 1 then
11:         return;
12:     end if
13:     select an axis-aligned splitting plane that introduces the
    smallest number of intersections with the bounding boxes of
    the subgrids in slist (there should be at least one slab of cells
    on each side);
14:     construct n.leftChild and n.rightChild according to the se-
    lected splitting plane;
15:     llist ← the subgrids in slist that overlap n.leftChild;
16:     rlist ← the subgrids in slist that overlap n.rightChild;
17:     SplitLeaf(n.leftChild, llist);
18:     SplitLeaf(n.rightChild, rlist);
19: end function
```

6 RENDERING

At the runtime, our system loads the preprocessed data into main memory and GPU memory, and performs volume ray casting to generate the image.

6.1 Data Caching and Prefetching

The basic idea of our caching design is to keep as many time-steps in the GPU memory and main memory as possible. The GPU memory and the main memory are treated as level 1 and level 2 cache, respectively, forming a three-level memory hierarchy: GPU memory, main memory, and the SSD. When a time-step is selected by the user, the cache manager checks the GPU memory and the main memory to see if the requested time-step already exists. If the time-step is not in the GPU memory or the main memory, the corresponding data transfer from lower levels to upper levels is then performed.

Since the time-step switching actions are usually taken between adjacent time-steps, the subsequent time-steps can be prefetched beforehand. We keep a priority queue of all time-steps keyed on the prefetching priority according to the current time-step. The closer time-steps to the current time-step have higher priority. For the two time-steps evenly close to the current step, the one after the current step has higher priority since the time sequence is more likely to be player in the forward direction. In order to overlap the rendering time and the data transferring time, a separate cache management thread is used to handle the data swapping. The prefetching takes place when the cache management thread is idle.

6.2 Ray Casting Hybrid-Grid

As aforementioned, the HELIOS datasets contain two different types of grid: unstructured near-body grid and Cartesian AMR off-body grid. The overlap between these grids makes volume rendering a non-trivial task. The unstructured near-body grid always overlays the background Cartesian AMR off-body grid, and the near-body grid itself can have multiple blocks overlapping each other. In order to correctly render the hybrid-grid data, the overlapping regions must be carefully handled. In HELIOS datasets, the transition from the near-body grid to the off-body grid normally occurs at a distance where the size of the near-body grid cells is approximately commensurate with the size of the off-body grid cells. In other words, at the regions that the near-body grid overlaps off-body grid, the resolution of near-body grid is usually higher than or at least about the same as the resolution of the off-body grid. Therefore, our strategy dealing with the overlapping regions is straightforward: we always sample the unstructured near-body grid when it is available.

To perform a switch from the off-body grid to the near-body grid

at the interface between the two grids, we need to find the intersection of the viewing ray and the boundary of the unstructured near-body grid. Previous unstructured-grid ray casting approaches usually do this by using rasterization of the visible boundary faces of the mesh. However, this procedure becomes a bit complex if the mesh is non-convex since there may be re-entries, i.e. the viewing rays exited from the mesh may re-enter the mesh. To properly handle non-convex meshes, convexification [24] or depth peeling [3] techniques are used to solve the re-entry problem. However, these techniques cannot be directly used for the HELIOS hybrid-grid datasets since there can be overlaps within the unstructured near-body grid, which means that when the viewing ray exits from the near-body grid, it may still be inside another near-body grid block. Our single-pass method and the multi-pass method mainly differ in the way they handle this issue.

6.2.1 Single-Pass Ray Casting

We implemented the single-pass hybrid-grid ray casting algorithm using NVIDIA CUDA. The traversal of the viewing rays is performed by launching a screen-sized CUDA grid, where each thread computes a single ray and outputs the color of a pixel on the screen. At the beginning of the traversal, a ray-box intersection test between the viewing ray and the bounding box of the data domain, that is, the bounding box of the root node of the KD-tree, is applied to obtain the ray segment that intersects the domain box. The threads in which the ray misses the domain box is terminated immediately.

The KD-tree traversal starts at the root node. A top-down traversal is applied to find the foremost leaf node. Although previous GPU KD-tree traversal approaches often make use of stack-less techniques such as *KD-restart* algorithm [8] in order to avoid using shader-unsupported local arrays, our implementation uses standard stack-based KD-tree traversal algorithm as the one introduced in [21] since the limitation of using local arrays is relaxed in modern GPU programming. Our experiments show that KD-restart algorithm does not benefit the rendering in terms of performance.

Once a leaf node is reached, we apply ray marching within the AMR subgrid pointed by the node to evaluate the color integration along the ray segment. This process is the same as the standard GPU ray casting approach for uniform-grid data except that if the cell ID field of the node is non-negative, which means that there are some tetrahedral cells located within the bounding box of the node, we make a cell location query to check if the ray has entered the near-body unstructured grid in each step.

The cell location is performed by the following process: We start at the cell indicated by the leaf node, and shoot a ray from the cell center to the queried point. Making use of cell adjacency information provided by the connectivity list, the cell containing the queried point is reached by cell walking, and the ID of the cell is then returned by the cell location procedure. If the grid boundary is reached during the cell walking, the procedure returns -1, meaning that the point is outside the grid. Note that this procedure is not one hundred percent accurate since the ray may exit from the grid and then enter the grid again at complex grid boundary. Langbein et al. [12] present a method to handle this situation. However, with the HELIOS datasets, even if we miss some boundary parts of the near-body grid, the space can still be filled by the off-body grid, so there will not be serious artifacts due to this problem.

If a non-negative cell ID is returned by the cell location procedure, we immediately stop the ray marching of the AMR subgrid and enter the tetrahedral grid cell walking routine. Weiler et al. [24] give a detailed description of the method to determine the exit point and the exit face given the ray and the cell vertices. Given the exit face, the connectivity list is again used to locate the next cell for each step. The cell walking terminates when the ray hits the grid boundary.

During the traversal within the near-body unstructured grid, the KD-tree traversal status is not updated as the ray marches. So when the ray exits from the near-body grid, we have to update the KD-tree traversal status according to the current position. We do this by examining the ray segments of the stack elements. We keep popping the stack until the top element of the stack has the ray segment containing the current position (the exit point from the near-body grid). The near distance of the top element is then replaced by the current position so that everything before the current point will be skipped in the following traversal.

For the ray integration of both AMR grid and unstructured grid, we apply the pre-integration approach published in [14]. The method extends ordinary pre-integration approach to support high-quality lighting. In AMR grids, the step size is selected according to the voxel size of the subgrid. We found that in coarse subgrids, a low sampling rate (large step size) may cause obvious artifacts due to the huge gradient variation. On the other hand, in fine subgrids, a lower sampling rate (relative to the voxel size) can be used. We set the step sizes for fine subgrids and coarse subgrids respectively so that the sampling rate for fine subgrids is about two samples per voxel and the sampling rate for coarse subgrids is about ten samples per voxel.

The ray traversal algorithm is summarized in Algorithm 2.

6.2.2 Multi-Pass Ray Casting

In this section, an alternative approach to ray casting hybrid grid, a multi-pass method, is presented. Our multi-pass rendering algorithm is based on a technique similar to depth peeling [7]. Depth peeling is a technique originally introduced to handle order-independent transparency. Weiler et al. [25] uses the idea to handle the re-entry problem of ray casting non-convex tetrahedral meshes. We further extend the method to deal with hybrid-grid rendering.

The idea of our multi-pass approach is to perform several traversal cycles on the near-body grid. In each cycle, we traverse the view rays starting from the intersection of the rays with a layer of the front faces of the boundary mesh. Before the first cycle, after the last cycle, and between each two consecutive cycles, AMR ray traversal is performed to evaluate the contribution of the off-body grid within the regions that are not covered by the near-body grid.

A rendering pass starts with rasterizing the list of boundary triangles, labelled with cell indices, of the tetrahedral near-body grid. During the rasterization, besides the standard OpenGL `GL_LESS` depth-test, a front-depth-test is applied. The depth-buffer of the previous pass, which records the depth of the previous layer, is bound as a texture, serving as the front-depth-buffer. In the fragment shader, the incoming sample is tested to see if the depth of the sample is greater than the value of the front-depth-buffer. Any sample that does not pass the test is discarded, so that previous layers are not rendered again. For each pixel, the fragment shader outputs the index of the intersected tetrahedral cell as well as the intersection distance (from the ray origin).

Next, we perform AMR ray traversal on the off-body grid. The traversal is applied within a ray-segment, represented by the ray origin, ray direction, `tnear`(near-distance), and `tfar`(far-distance). We set `tfar` to the intersection distance got from the near-body boundary mesh rasterization, so that the off-body grid traversal stops at the point that the ray intersects the near-body grid. `tnear` is set to the near-clip distance, usually a small value, in the first pass. In the following passes, `tnear` is set to the distance between the ray origin and the exit point from the near-body grid in the previous pass, so that the ray starts the off-body grid traversal immediately at the point that the ray exits from the near-body grid.

After the off-body traversal, the unstructured-grid traversal on the near-body grid is then performed. Similar to the AMR ray traversal, the unstructured-grid traversal procedure takes a ray-segment as input, where the `tnear` is set to the same value as

Algorithm 2 Single-Pass Ray Casting

```
1: tnear, tfar ← domainBox.clipRaySegment();
2: if tnear > tfar then                                  ▷ ray misses the box
3:     return;
4: end if
5: node ← root;
6: while true do
7:     KD-tree traversal to find the leaf node (updating node, tnear, tfar, and stack);
8:     subgrid ← subgrid[node.subgridId];
9:     cellId ← -1;
10:    while tnear < tfar do
11:        p ← ray.origin + ray.dir * tnear;
12:        if node.cellId ≥ 0 then
13:            cellId ← cellLocation(p, node.cellId);
14:            if cellId ≥ 0 then
15:                break;
16:            end if
17:        end if
18:        sample subgrid at p;
19:        accumulate color and opacity;
20:        tnear ← tnear + sampleStep;
21:    end while
22:    if cellId ≥ 0 then                                ▷ ray hits the near-body grid
23:        while cellId ≥ 0 do
24:            accumulate color and opacity;
25:            tExit ← the distance at which the ray exits from the cell;
26:            exitFace ← the face through which the ray exits from the cell;
27:            cellId ← connectivityList[cellId][exitFace];
28:        end while
29:        while not stack.empty() and stack.top.tfar ≤ tExit do
30:            stack.pop();
31:        end while
32:        if not stack.empty() then
33:            stack.top.near ← tExit;
34:        end if
35:    end if
36:    if stack.empty() then
37:        return;
38:    end if
39:    nodeId, tnear, tfar ← stack.pop();
40: end while
```

which used in the off-body traversal, and `tfar` is set to the infinite value. The cell index and the intersection distance retrieved in the rasterization step is used to determine the starting point of the traversal. Since there can be multiple near-body grid blocks overlapping each other in the hybrid-grid data as shown in Figure 3, the overlapping regions must be carefully handled. The spatial relationship between the current layer and the previous layer can be judged by comparing the `tnear` value with the intersection distance `tIsect`. The case that `tIsect` is greater than `tnear` means the current layer is separated from the previous layer and no special treatment is needed, so we start the cell walking from `tIsect` normally. On the other hand, if `tIsect` is less than `tnear`, it means that the two layers overlap each other and the exit point from the previous layer is inside the current layer. In this case, we perform cell walking but ignore the color/opacity contributions of the cells until the ray reaches `tnear`, so that the overlapping region is not taken into account again. Note that the off-body traversal between two overlapping layers is not performed. This is achieved automatically because in this case the `tnear` would be greater than `tfar` in the AMR traversal procedure. At the end of the near-body

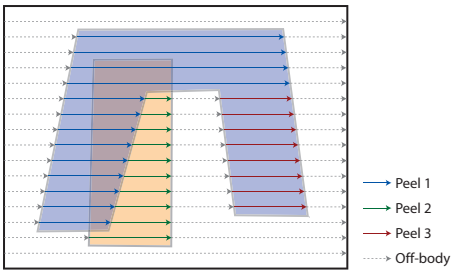


Figure 3: Depth peeling for overlapping near-body grid blocks. The blue rays are traversed in peel 1, while green rays are in peel 2, and red rays are in peel 3. Dashed rays are traversed in the off-body grid.

ray traversal process, the distance between the ray origin and the point where the ray exits from the mesh is written to a buffer for the `tnear` value of the next pass.

The process continues until there is no sample passing the depth-test in the fragment shader, meaning that the layers of the near-body grid are all "peeled-off".

The AMR and the unstructured-grid ray traversal are both implemented in CUDA. The AMR ray traversal kernel implements the single-pass AMR ray casting algorithm presented by Kähler et al. [11]. The algorithm relies on the KD-tree built in the preprocessing stage. But unlike the single-pass approach described in the previous section, we do not use this KD-tree for cell location, so only the AMR part of the KD-tree is needed. The unstructured-grid ray traversal kernel implements a single-pass tetrahedral cell walking algorithm based on the techniques suggested in the paper of Weiler et al. [25]. The same as the single-pass approach presented in the previous section, we apply the pre-integration method to perform high-quality ray integration. The color and opacity integration result of each ray segment of both AMR traversal and unstructured-grid traversal is blended into the same buffer.

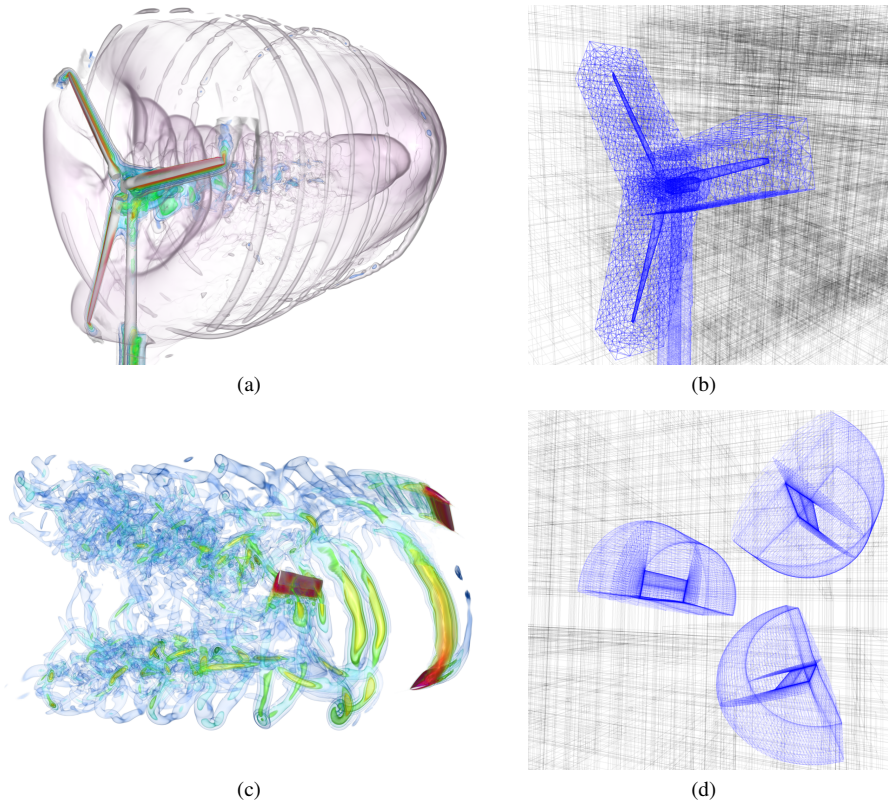
Algorithm 3 gives a summary of the multi-pass method.

7 Results and Discussion

We tested our implementation on an NVIDIA GeForce GTX Titan graphics card with 6 GB of video memory, installed on a PC with an Intel Core i7 3.5 GHz processor and 32 GB of main memory. Performance and memory requirement tests are performed on two datasets. The first dataset is a 264-step simulation of a wind turbine (henceforth referred to as Large dataset). The second dataset is a relatively small dataset with a 289-step simulation of a wind turbine with vertical blades (henceforth referred to as Small dataset). Example images are shown in Figure 4. Table 1 lists the characteristics of a typical time-step of the datasets.

The one-time preprocessing is performed on the test datasets. Table 2 lists the size of the preprocessed data as well as the processing time. For the grid part of the near-body grid, only the first time-step is stored completely, while in the following steps we only keep the difference of the step to the first step.

The runtime performance is evaluated by benchmarking the data transferring time and the rendering time. Rendering time is tested on a viewport size of 1024×1024 pixels. We compared the two rendering methods, the single-pass and the multi-pass approach, proposed in this paper. The results show that the multi-pass approach is faster than the single-pass approach, suggesting that the cost of processing the depth peeling and launching kernel many times may be lower than the overhead introduced by the deep KD-tree traversal for cell location in the single-pass approach. Also the long kernel code of the single-pass ray traversal may consume too many registers, causing low warp occupancy and reducing latency hiding ability in the CUDA kernel. Furthermore, in the single-pass



(a) (b) (c) (d)

Figure 4: Rendering examples of the datasets. (a) shows the velocity magnitude of the Large dataset. (c) shows the vorticity magnitude of the Small dataset. In both images, high-value regions are colored in red, while blue color indicates the low-value regions. (b) and (d) show the hybrid-grid layout of the datasets.

Table 1: Dataset characteristics.

			Off-body grid			Near-body grid			
Dataset	raw data size	#steps	#levels	#grids	#cells	#tetras	#prisms	#pyramids	#hexas
Large	456GB	264	6	2704	54M	3.2M	2.6M	29K	0
Small	164GB	289	6	765	17M	0	0	0	196K

Table 2: Processed data size and the processing time for the preprocessing of a single time-step.

	Off-body grid	Near-body grid			KD-tree		
Dataset	solution	grid(first step)	grid	solution	#nodes	size	processing time
Large	208MB	367MB	22MB	29MB	3.9M	45MB	38.0s
Small	68MB	38MB	2MB	3MB	0.3M	4MB	8.1s

approach, the near-body grid and the off-body grid are handled in the same kernel at the same time. The large memory footprint may work against the caching performance in the kernel.

The packing approach and the bindless texture approach for AMR rendering are examined. Although these two approaches have been discussed in [11], we applied the comparison again because our system considers not only the rendering performance but also the data loading and preparation time. Our results show that, as shown in Table 3, there is no significant difference in rendering time between the packing approach and the texture object approach for the AMR rendering. The packing approach uses more memory because fragmentation is introduced when the differently sized subgrids are packed into the texture memory pool. Since we perform the packing at the moment the subgrids are loaded into the main memory, the data loading+processing time from SSD to the main memory of the packing approach is longer than the texture object approach. However, the texture object approach spends more time on the GPU data loading stage due to the frequent allocation of the GPU texture memory and the creation of texture objects. In addition, we found that with the texture object approach, sometimes the GPU out-of-memory exception occurs even when the total texture size does not exceed the GPU memory size. A possible reason is that the large amount of small textures causes fragmentation in the GPU memory. So overall, we found that the texture object approach

Algorithm 3 Multi-Pass Ray Casting

```
1: clearBuffers();
2: tCurrent ← EPSILON;
3: tSceneMax ← INFINITE;
4: layer ← 0;
5: while true do
6:     entryCell, tIsect, depthBuffer[1-layer%2] ← peelALayer(depthBuffer[layer%2]);
7:     if no sample passed then
8:         break;
9:     end if
10:    rayCastingAMR(tCurrent, tIsect);
11:    rayCastingTetrahedral(tCurrent, tSceneMax, entryCell, tIsect);
12:    layer ← layer + 1;
13: end while
14: rayCastingAMR(tCurrent, tSceneMax);
15:
16: function RAYCASTINGAMR(tnear, tfar)
17:    accumulate the color and opacity along the ray from tnear to tfar;
18: end function
19:
20: function RAYCASTINGTETRAHEDRAL(tnear, tfar, entryCell, tIsect)
21:    if tnear < tIsect then
22:        tnear ← tIsect;
23:    else
24:        skip the cells between tIsect and tnear;
25:    end if
26:    accumulate the color and opacity along the ray from tnear until the ray reaches tfar or exits from the mesh;
27:    tCurrent ← the exit distance;
28: end function
```

does not benefit from the fewer memory usage.

The depth of the hybrid KD-tree can be adjusted in the construction time by selecting the maximum number of tetrahedral cells allowed in a single leaf node. We tested different construction settings, resulting the trees with different sizes. Figure 5 shows the runtime performance with these trees for the Large dataset using the single-pass approach. It is clear that the data loading time decreases as the tree gets shallower. However, the rendering time shows a U-shape in the figure. The two ends, the deepest tree and the shallowest tree, perform worse than the median setting. Apparently, the deep trees need more steps during the traversal, which may introduce more cost. On the other hand, shallow trees allow more cells residing in a leaf node, giving an inaccurate starting point for cell location, and therefore more steps are needed in the cell walking process for locating the queried point.

Although it seems that the multi-pass approach totally outperforms the single-pass approach, the single-pass method may be beneficial in some cases. For example, since the single-pass approach allows simultaneous access to both the near-body grid and the off-body grid, a sophisticated sampling scheme may be used to eliminate the visually undesired seams at the interface between the near-body grid and the off-body grid.

8 Conclusion and Future Work

We presented a volume renderer for time-varying hybrid-grid data which couples near-body unstructured grid and off-body Cartesian AMR grid. Two different approaches to hybrid-grid volume rendering are introduced and compared. In both methods, the difficulties introduced by the hybrid, overlapping meshing paradigm are addressed. Experiments show that in general, the multi-pass method outperforms the single-pass method. With the optimal settings, our renderer can achieve interactive or near-interactive performance for the hybrid-grid data on a single machine. In addition, although not mentioned in the results section, the caching and prefetching mechanism we proposed greatly improves the user experience by overlaying the data transfer time and the rendering time.

Due to the inconsistency between the different solutions from different solvers used for the near-body and the off-body grids, there are obvious seams on the rendered images at the interface between the two grid systems with our current implementation. This problem could be solved by sampling from both grids and blending the sampled values by a carefully chosen weighting at the interface regions.

The data size of a single time-step is limited by the GPU memory size in our current design. To handle large hybrid meshes, the renderer presented in this work could also be adapted to GPU clusters.

Table 3: Runtime performance (all in milliseconds). The single-pass method (SPM) is compared with the multi-pass method (MPM). The texture object approach (TO) and the packing approach (PK) for AMR rendering are also compared.

			Large		Small	
			SPM	MPM	SPM	MPM
SSD to main memory	off-body	TO	180	180	53	53
		PK	360	360	103	103
	near-body		233	233	25	25
	KD-tree		33	0	4	0
	total	TO	446	413	82	78
		PK	626	593	132	128
Main memory to GPU	off-body	TO	819	819	205	205
		PK	173	173	36	36
	near-body		96	96	18	18
	KD-tree		16	0	2	0
	total	TO	931	915	225	223
		PK	285	269	56	54
Rendering	TO		648	436	581	239
	PK		648	419	579	224

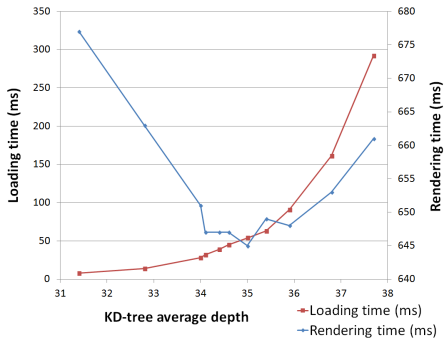


Figure 5: Data loading time (red line) and rendering time (blue line) to the KD-tree depth.

Acknowledgements

This research has been sponsored in part by the National Science Foundation through grants DRL-1323214, IIS-1255237, and CCF-0938114, and Department of Energy through grants DE-FC02-06ER25777, DE-CS0005334, and DE-FC02-12ER26072 with program managers Lucy Nowell and Ceren Susut-Bennett.

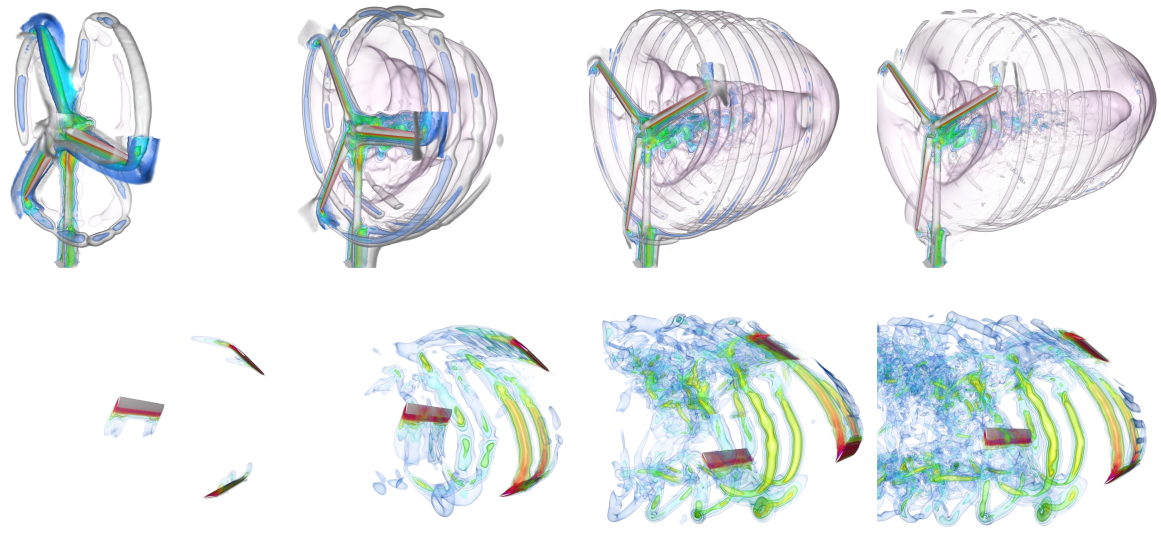


Figure 6: Selected time-steps of the two datasets. Top: the Large dataset. Bottom: the Small dataset.

REFERENCES

[1] M. J. Berger and J. E. Oliger. Adaptive mesh refinement for hyperbolic partial differential equations. *Journal of Computaional Physics*, 53(3):484–512, 1984.

[2] F. F. Bernardon, S. P. Callahan, J. a. L. D. Comba, and C. T. Silva. Interactive volume rendering of unstructured grids with time-varying scalar fields. In *Proceedings of the Eurographics Conference on Parallel Graphics and Visualization 2006*.

[3] F. F. Bernardon, C. A. Pagot, J. L. D. Comba, and C. T. Silva. Gpu-based tiled ray casting using depth peeling. Technical report, 2004.

[4] S. Callahan, M. Ikits, J. Comba, and C. Silva. Hardware-assisted visibility ordering for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 11(3):285–295, 2005.

[5] Y.-J. Chiang. Out-of-core isosurface extraction of time-varying fields over irregular grids. In *IEEE Visualization*, pages 217–224, 2003.

[6] C. Correa, R. Hero, and K.-L. Ma. A comparison of gradient estimation methods for volume rendering on unstructured meshes. *IEEE Transactions on Visualization and Computer Graphics*, 17(3):305–319, March 2011.

[7] C. Everitt. Interactive order-independent transparency. Technical report, 2001.

[8] T. Foley and J. Sugerman. Kd-tree acceleration structures for a gpu raytracer. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '05, pages 15–22, New York, NY, USA, 2005. ACM.

[9] L. J. Gosink, J. C. Anderson, E. W. Bethel, and K. I. Joy. Query-driven visualization of time-varying adaptive mesh refinement data. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1715–1722, Nov. 2008.

[10] M. Hadwiger, J. Beyer, W.-K. Jeong, and H. Pfister. Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2285–2294, Dec 2012.

[11] R. Kähler and T. Abel. Single-pass gpu-raycasting for structured adaptive mesh refinement data. *CoRR*, abs/1212.3333, 2012.

[12] M. Langbein, G. Scheuermann, and X. Tricoche. An efficient point location method for visualization in large unstructured grids. In *VMV*, pages 27–35, 2003.

[13] N. Leaf, V. Vishwanath, J. A. Insley, M. Hereld, M. E. Papka, and K.-L. Ma. Efficient parallel volume rendering of large-scale adaptive mesh refinement data. In *LDAV*, pages 35–42, 2013.

[14] E. B. Lum, B. Wilson, and K.-L. Ma. High-quality lighting and efficient pre-integration for volume rendering. In *Proceedings of the Joint Eurographics - IEEE TCVG Conference on Visualization 2004*.

[15] K.-L. Ma. Visualizing time-varying volume data. *Computing in Science Engineering*, 5(2):34–42, Mar 2003.

[16] A. Maximo, R. Marroquim, and R. Farias. Hardware-assisted projected tetrahedra. In *Proceedings of the Eurographics / IEEE - VGTC Conference on Visualization 2010*.

[17] P. Muigg, M. Hadwiger, H. Doleisch, and H. Hauser. Scalable hybrid unstructured and structured grid raycasting. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1592–1599, Nov 2007.

[18] V. Sankaran, A. Wissink, A. Datta, J. Sitaraman, M. Potsdam, B. Jayaraman, A. Katz, S. Kamkar, B. Roget, D. Mavriplis, H. Saberi, W.-B. Chen, W. Johnson, and R. Strawn. Overview of the helios version 2.0 computational platform for rotorcraft simulations. In *49th AIAA Aerospace Sciences Conference*. AIAA Paper 2011-1105, 2011.

[19] P. Shirley and A. Tuchman. A polygonal approximation to direct scalar volume rendering. In *Proceedings of the 1990 Workshop on Volume Visualization*, VVS '90, New York, NY, USA, 1990. ACM.

[20] J. Sitaraman, A. Wissink, V. Sankaran, B. Jayaraman, A. Datta, Z. Yang, D. Mavriplis, H. Saberi, M. Potsdam, D. O'Brien, R. Cheng, N. Hariharan, and R. Strawn. Application of the helios computational platform to rotorcraft flowfields. In *48th AIAA Aerospace Sciences Conference*. AIAA Paper 2010-1230, 2010.

[21] I. Wald. *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Saarland University, 2004.

[22] I. Wald, H. Friedrich, A. Knoll, and C. D. Hansen. Interactive isosurface ray tracing of time-varying tetrahedral volumes. *IEEE Trans. Vis. Comput. Graph.*, 13(6):1727–1734, 2007.

[23] C. Wang, H. Yu, and K.-L. Ma. Application-driven compression for visualizing large-scale time-varying data. *Computer Graphics and Applications, IEEE*, 30(1):59–69, Jan 2010.

[24] M. Weiler, M. Kraus, M. Merz, and T. Ertl. Hardware-based ray casting for tetrahedral meshes. In *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, pages 44–, Washington, DC, USA, 2003. IEEE Computer Society.

[25] M. Weiler, P. N. Mallon, M. Kraus, and T. Ertl. Texture-encoded tetrahedral strips. In *Proceedings of the 2004 IEEE Symposium on Volume Visualization and Graphics*, VV '04, pages 71–78, Washington, DC, USA, 2004. IEEE Computer Society.

[26] A. Wissink, J. Sitaraman, V. Sankaran, D. Mavriplis, and T. Pulliam. A multi-code python-based infrastructure for overset cfd with adaptive cartesian grids. In *46th AIAA Aerospace Sciences Conference*. AIAA Paper 2008-927, 2008.