Interactive Computation and Rendering of Finite-Time Lyapunov Exponent Fields

Samer Barakat, Christoph Garth, *Member, IEEE*, and Xavier Tricoche, *Member, IEEE*

Abstract—Finite-time Lyapunov exponent and Lagrangian coherent structures are popular concepts in fluid dynamics for the structural analysis of fluid flows but the associated computational cost remains a major obstacle to their use in visualization. In this paper, we present a novel technique that allows for the coupled computation and visualization of salient flow structures at interactive frame rates. Our approach is built upon a hierarchical representation of the FTLE field, which is adaptively sampled and rendered to meet the need of the current visual setting. The performance of our method allows the user to explore large and complex datasets across scales and to inspect their features at arbitrary resolution. The paper discusses an efficient implementation of this strategy on the graphics hardware and provides results for an analytical flow and several CFD simulation datasets.

Index Terms—Flow Visualization, Vector Field Data, GPU and Multi-core Architectures, Interactive, FTLE, Streaming Data.

1 Introduction

Fluid flows are ubiquitous in science and industry and massive computational resources are invested to study their behavior though Computational Fluid Dynamics (CFD) simulations. The analysis of the resulting data seeks a better understanding of both qualitative and quantitative properties of fluid flows, which has become instrumental in the study of multifaceted physical phenomena. Yet, despite the impressive research activity in flow visualization over the last 20 years [22], [23], [17], creating effective depictions of very large, complex, and unsteady vector fields remains challenging.

Topological [18] and *feature-based* [23] methods specifically attempt to characterize salient structures to offer a high-level representation of the flow. Yet, both approaches have significant shortcomings. The topological technique is sometimes unable to capture essential flow patterns such as vortices. Moreover, its Eulerian viewpoint and lack of Galilean invariance make the interpretation of topological structures problematic in the transient setting. In contrast, feature-based visualization techniques use problem-driven and typically heuristic feature definitions to identify interesting patterns in an application-specific fashion. In this context, a promising third avenue has been emerging in recent years, which leverages the notion of *Lagrangian Coherent Structures* (LCS) to create a portrait of the flow. LCS correspond to ridges of the Finite-Time Lyapunov Exponent (FTLE), a scalar field that characterizes the amount of stretching about the trajectory of a point over a finite time interval given a certain starting time. Following the seminal work by Haller [10], LCS visualization through FTLE has gained significant traction in the fluid dynamics [29], [9], [30], [2], [20] and visualization [6], [26], [7], [31], [27], [14], [28], [35] communities for its ability to automatically identify important flow structures within an objective mathematical framework.

Unfortunately, despite its compelling properties, the FTLE-based approach poses a fundamental computational challenge. Indeed, the analysis of the FTLE field requires to compute the trajectories of massless particles along the flow from a *dense* set of spatial locations. The relative behavior of neighboring particles during the motion is then used to measure the coherence of the underlying transport phenomenon and to determine the presence of embedded *repelling* and *attracting manifolds*. Hence, a prohibitive numerical integration must be carried out from virtually *every* location in the domain. In addition, the Lagrangian nature of the analysis enables the identification of structures whose scale can be significantly smaller than the computational mesh [21].

Despite the contributions made recently to improve the efficiency of FTLE computation and visualization [6], [26], [2], [20], [27], [14], no technique proposed to date achieves interactive performance. Yet, interactivity is essential to turn any visualization approach into an exploratory tool. In the special case of FTLE visualization, interactivity would be especially valuable since it would afford the user the control necessary to navigate the inherently multi-scale complexity of Lagrangian structures. Furthermore it would permit to focus the computation on the flow patterns that are most relevant to the analysis. These observa-

- *S. Barakat and X. Tricoche are with the Department of Computer Science, Purdue University, West Lafayette, IN 47907. E-mail:* {sbarakat|xmt}@purdue.edu
- *C. Garth is with the Computer Science Department, UC Davis, Davis, CA 95616 and the Computer Science Department, University of Kaiserslautern, 67655 Kaiserslautern, Germany E-mail:* garth@cs.uni-kl.de.

tions constitute the motivation of the present work. We introduce an efficient technique for the coupled computation and visualization of FTLE at interactive frame rates. Our approach is built upon a hierarchical representation of the FTLE field, which is adaptively sampled to meet the needs of the current visual setting. By optimizing the use of the computational power available on modern graphics processing units (GPU) we achieve the interactivity that allows the user to explore a dataset across scales and inspect its features at arbitrary resolution.

The contents of this paper are organized as follows. We review previous work in Section 2. We formally define Lagrangian coherent structures and discuss the challenge posed by their visualization before outlining our proposed solution in Section 3. A novel texture-based dynamic octree data structure is introduced in Section 4 followed by a description of our allocation strategy in Section 5. We present the details of our algorithm in Section 6, along with the pseudocode for the key aspects of our GPU implementation. Results and performance assessments are provided in Section 7. Finally we conclude our presentation by discussing possible extensions of this work in Section 8.

2 Related Work

Haller pioneered the idea of characterizing Lagrangian coherent structures in transient flows through the finite-time Lyapunov exponent [10]. This seminal contribution generated a significant interest in FTLE and its applications to the structural analysis of transient flows in the fluid dynamics community, both from a theoretical and from a practical viewpoint. Haller studied the robustness of the structures characterized by FTLE [12] and showed that they remain valid under approximation errors in the velocity field. He also suggested to identify stable and unstable manifolds with ridge lines of the FTLE field. Shadden et al. investigated the theory of FTLE and LCS in 2D [29] and proved that the flux across ridge lines of the FTLE field is small and typically negligible. An extension of LCS to arbitrary dimensions was proposed [30]. These concepts have been applied to the study of turbulent flows [11], [9], [21] and used in the analysis of vortex ring flows [28].

Several approaches have been explored in the visualization literature to analyze the structures exhibited by time-dependent flows. Topological methods have been applied to transient flows in the Eulerian perspective [34], [32], [33]. Flow topology is not Galilean invariant and the choice of a proper reference frame is especially problematic in the transient setting since instantaneous topological features of the velocity field do not reveal the true nature of the flow. Moreover, the corresponding techniques can miss essential flow patterns like vortices. Texture-based representations have been used to visualize time-dependent flows while offering an effective depiction of salient structures, see [16] and references therein. Because of the intrinsic difficulty of defining structures that are both coherent in space and time, each of these methods resorts to an ad-hoc combination of Eulerian and Lagrangian perspectives, thus yielding animations whose physical interpretation can be difficult.

Sadlo and Peikert presented a method for the extraction of LCS through ridge-driven adaptive refinement of the FTLE field [25] and later extended this work to support the tracking of LCS over time [27]. Lipinski and Mohseni proposed a technique for the efficient computation of LCS through a ridge tracking approach that exploits temporal coherence while assessing the approximation error [20]. Neither method is interactive and their mapping to the GPU appears non-trivial. The GPU-based acceleration of FTLE computations for two-dimensional flows was previously considered by Garth et al. [8]. However, their method does not extend to 3D flows due to memory and bandwidth constraints. An adaptive FTLE computation for 3D flows was also described by Garth et al. [6]. Brunton et al. proposed a technique that exploits the similarities between trajectories of a sequence of flow maps over time to speed up the computation [2]. A germane idea was applied by Hlawatsch et al. who introduced a new hierarchical computation scheme for integral curves and described a GPU implementation [13]. Both approaches trade computational complexity for increased memory footprint, which is tied to the spatial and temporal resolution. Investigating fine scales can become prohibitively expensive, especially for time-dependent datasets. In addition, unnecessary computations are performed when finer resolutions are only relevant in a small portion of the volume. Some papers have considered data-driven refinement algorithms that exploit the coherence of particle paths to generate smooth approximations of the flow map [6], [19]. In our approach, we adopt an adaptive view-dependent refinement where computation is performed on the fly while optimizing the use of memory and computation resources.

Crassin et al. discussed the volume rendering of large datasets using multiresolution data representation on the GPU [4]. They proposed corresponding techniques for streaming and high quality filtering. The *Tuvok* architecture [5] also achieves multiresolution volume rendering using an optimized out-of-core, bricked, level of detail data representation. In both cases, the volume is fixed. Hence, their proposed multiresolution structures are static, in contrast to the dynamic octree data structure described in Section 4. We present an algorithm for adaptive optimization of the tree hierarchy based on the visual result, and the memory available on the GPU. Our method permits progressive sampling on the fly for the required resolution rather than paging solutions.

3 Theory and Method Overview

In the following we provide the definition of finite-time Lyapunov exponent before outlining our approach for its computation and visualization.

3.1 Flow Map and Finite-time Lyapunov Exponent

Let $\mathbf{v} : (I \subset I\!R) \times (D \subset I\!R^3) \rightarrow I\!R^3$ be a smooth time-dependent three-dimensional vector field defined over a spatial domain D and a time interval I describing the velocity of a fluid flow. The corresponding dynamical system describes the motion of massless particles along the flow:

$$\begin{cases} \dot{\mathbf{x}}(t,\, t_0,\, \mathbf{x_0}) &= \mathbf{v}(t,\, \mathbf{x}(t,\, t_0,\, \mathbf{x_0})) \\ \mathbf{x}(t_0,\, t_0,\, \mathbf{x_0}) &= \mathbf{x_0}. \end{cases} \tag{1}$$

The map $\mathbf{x}(\cdot,\, t_0,\, \mathbf{x_0}) : t \mapsto \mathbf{x}(t,\, t_0,\, \mathbf{x_0})$ describes a particle *trajectory*. The map $\mathbf{x_t} := \mathbf{x}(t,\, t_0, \cdot)$ is called *flow map*: $\mathbf{x_t}(\mathbf{x_0})$: it indicates the position reached at time t by a particle released at $\mathbf{x_0}$ at time t_0.

With previous notations, one considers the spatial variations of the flow map $\mathbf{x}_t$, whereby $t = t_0 + \tau$ and τ is finite. The variations of this flow map around a given position $\mathbf{x_0}$ are locally determined by its spatial derivative, the matrix $\mathbf{J_x}(t, t_0, \mathbf{x_0}) := \nabla_{\mathbf{x_0}}\mathbf{x}(t, t_0, \mathbf{x_0})$ at $\mathbf{x_0}$ and the associated (right) Cauchy-Green deformation tensor $\mathbf{C} = \mathbf{J}_\mathbf{x}^T\mathbf{J_x}$. Maximizing the dispersion of particles over all directions around $\mathbf{x_0}$ at t_0 is namely equivalent to evaluating

$$\sigma_\tau(t_0, \mathbf{x_0}) := \sqrt{\lambda_{\max}(\mathbf{C}(t, t_0, \mathbf{x}_0))},$$

where $\lambda_{\max}$ denotes the maximum eigenvalue of the tensor. To obtain the average exponential separation rate, one finally computes the finite-time Lyapunov exponent $\lambda(t, t_0, \mathbf{x_0})$, defined as follows:

$$\lambda(t, t_0, \mathbf{x_0}) = \frac{1}{|\tau|} \log \sigma_\tau(t_0, \mathbf{x_0}).$$

This rate can be evaluated for both forward and backward advection (positive or negative τ).

Practically, the flow map is sampled at a discrete set of locations (typically at the vertices of a raster grid) through the numerical integration of the differential equation 1. FTLE is subsequently evaluated by approximating the spatial derivative $J_\mathbf{x}$ from this discrete information. Note that large values of λ for forward advection correspond to manifolds that have a strong repelling impact on nearby particles, while large FTLE values for backward advection correspond to manifolds that attract nearby particles.

3.2 Challenges and Proposed Solution

The visual exploration of FTLE in practical fluid flow problems poses two sets of challenges. The first one concerns the inherently multi-scale complexity of the patterns exhibited by FTLE in typical practical (*e.g.,* turbulent) flows. This property implies that an extremely fine flow map sampling rate may be required to properly resolve all the features of interest. In particular, this sampling resolution may significantly exceed the resolution of the flow simulation itself, as a consequence of the Lagrangian nature of the processing. Beside the difficulty of assessing this (*a priori* unknown) finest resolution in pre-processing, trying to uniformly compute an FTLE field at that resolution is clearly wasteful. First, the sparse nature of the LCS embedded in the FTLE field means that only a small fraction of the volume is actually going to contain salient surfaces. Second, a high-resolution uniform sampling yields a data volume that typically exceeds the memory of the graphics hardware in the rendering stage. In that regard, it is worth noting that while the method proposed by Sadlo and Peikert [26] offers a means to adaptively refine the flow sampling in the direct vicinity of LCS detected at an initially coarse resolution, it is not interactive and cannot guarantee the detection of those fine scale LCS that elude detection at a low flow sampling rate.

A second significant challenge pertains to the ambiguity of the integration length τ needed to achieve the best characterization of the flow structures. While short integration times fail to resolve fine structures and properly localize major ones, excessive integration times can yield an excessively complex pictures by superimposing multiple temporal scales. Hence, the usefulness of any offline FTLE computation in pre-processing depends on the appropriate choice of this parameter, which may require multiple attempts and thus incur a significant computational overhead.

Our solution to tackle both problems consists in carrying out the FTLE computation and the rendering of the resulting information in a view-dependent and interactive fashion. That way, we give the user flexible control over both integration length and adequate sampling resolution in a setting that is responsive to her interest and exploration behavior. Resolving FTLE structures at a high uniform resolution is intractable due to memory and time constraints. Instead, our method restricts the computation to the visible portion of the flow domain and further it adaptively focuses on the regions that contribute to the rendered image, as determined by the user-defined transfer function. More precisely, the basic idea of our method consists in alternating partial FTLE sampling and ray casting of the resulting field at each frame, following an incremental approach. By intertwining computation and rendering, we leverage the rendering step to inform the following computing step about missing samples that were identified during the volume rendering of the available data. Our method is built upon a hierarchical data representation implemented in texture memory. In contrast to typical view-dependent rendering techniques however, our method requires a dynamic data representation, which disqualifies static GPU data structures. We therefore propose a novel data structure that extends the traditional texture

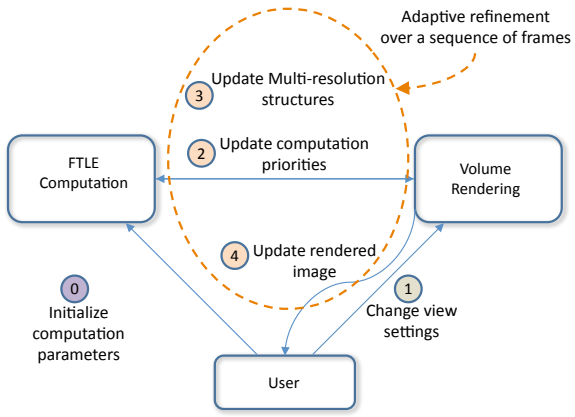


Fig. 1. Overview of the components of our method

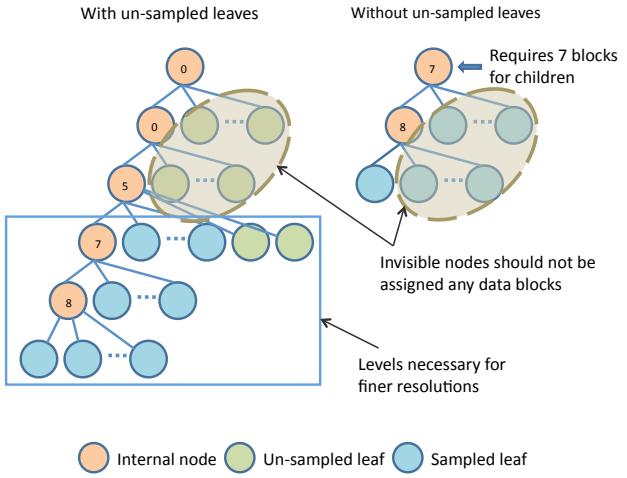


Fig. 2. The need for the virtual leaves

octree by allowing for the dynamic allocation of texture memory to the spatial region that is currently visualized. The allocation policy is controlled by a priority metric that quantifies the importance of a spatial region in the current frame in terms of its FTLE content, size, and visibility.

With these elements, our algorithm proceeds as an incremental process that optimizes the use of available computational and memory resources at each frame. Moreover, we adopt a progressive approach that weighs responsiveness and image quality based on the user interaction. Finally, the entire method has been implemented and optimized on the GPU to document its benefit in a practical setting. We present the main building blocks of our method in Figure 1.

4 Dynamic Hierarchical Data Representation

In the following we describe a novel data structure that supports the dynamic and adaptive refinement of both flow map and FTLE field. This data structure is pointerless and can be implemented very efficiently in texture memory on the graphics hardware.

4.1 Modified Texture Octree

Our data structure is based on the idea of the texture octree [15], whereby the octree nodes are saved and interlinked in a texture called *indirection pool*. Our implementation adopts a modified version of the texture octree in which texture blocks rather than individual values are assigned to the leaves [1], [24]. A direct benefit of this approach is that texture caching and interpolation can be leveraged, thus achieving high performance on the GPU. Note that the boundaries of each block are replicated across blocks to ensure the local memory footprint of both interpolation and gradient computation. In this work we improve upon this basic data structure in two ways to support the dynamic modification of the octree at runtime.

First, we introduce the concept of *virtual leaf*. A virtual leaf, in contrast to a *regular* leaf, does not have an assigned 3D texture block. In our terminology, leaves that possess a texture block are called *sampled leaves* since they contain sampled values of the flow map and FTLE field. Virtual leaves allow us to dynamically assign the limited number of 3D blocks available in texture memory to the portion of the tree that directly contributes to the visualization. In particular, as the user zooms into a particular region of the dataset, sampled leaves in coarser levels are continuously evacuated when necessary to free blocks for the finer levels, as explained in the algorithm description in Section 6. Observe, however, that we prevent the evacuation of nodes in the three coarsest levels to ensure the availability of a basic data set overview at all times (*e.g.*, when the user is suddenly zooming out). Refer to Figure 2.

Our second improvement consists in allowing continuous changes to the structure of the texture octree. This necessitates the use of a map, hereafter referred to as *indirection pool map* that keeps track of the free tree nodes. This map associates each octree node in the indirection pool with a value indicating whether that entry is currently used or not. When an octree node is deleted and an entry is freed in the indirection pool, its corresponding entry in the indirection pool map is marked as invalid. Upon sorting the indirection pool map with respect to the valid / invalid tag, free entries in the indirection pool are identified and made available for subsequent processing. Since the performance of the texture octree heavily relies on the caching mechanism offered by the GPU, it is read-only by nature. We overcome this limitation by deferring all per-frame structure modifications to the end of a computation step (further discussed in Section 6.3.1), which alleviates the issues caused by cache invalidation. In addition, the leaves (sampled and virtual) are linked to corresponding entries inside

the priority table data structure whose role is detailed in Section 5.2.

4.2 Block Repository

The block repository stores the 3D texture blocks that are pointed to by the sampled leaves of the octree. Each block contains entries corresponding to forward and backward FTLE computation over a portion of the domain. Updates to this structure can only occur at the granularity of a block since it is allocated in texture memory referenced by the ray casting. Consequently, we use a scratch pad structure that is capable of holding all the updates that can occur during a single computation phase (Section 6.3.1). The volume of these updates is significantly smaller than the repository size. The updates are then copied from the scratch pad structure into their corresponding positions in the repository once the computation phase is completed. This approach permits concurrent execution of computing and rendering, and increase cache performance by deferring updates until the rendering is complete.

5 Optimal Sampling

The interactivity of our visualization technique depends on its ability to automatically and effectively determine what sections of the dataset must be processed (through flow map sampling and FTLE computation) to meet the needs of the current camera setting. Given the strong memory and performance constraints that apply to our method, this selection amounts to an optimization process. Specifically, we introduce a priority function that quantifies the contribution of a region to the visualization from the current viewpoint. The priority value assigned to each leaf allows us to define an optimal octree as the solution that maximizes the overall priority within the given memory constraints. These aspects are detailed next.

5.1 Priority function

The priority function evaluation naturally occurs during the rendering stage since this is where the visual impact of a given block in the final image is determined. The basic objective of the priority function is to provide an objective measure of visual importance.

The image footprint of a block at distance r from the viewpoint scales as $1/r^2$, hence we insert this term in the expression of the priority function. The second factor concerns the effect of the block on the color of the pixels corresponding to its image footprint. Several aspects need to be taken into consideration. The data values in the block may be mapped through the transfer function to a very low opacity, which implies that the block has very little to show and its priority should be low. The same is true if the block is occluded by other blocks. A straightforward technique would be to measure the total contributions of the block to the colors of the rays intersecting it. Thus for every ray one could measure the change in the ray color and opacity after traversing the block. However we are primarily interested in the maximum impact of the block on all crossing rays since it allows us to account for the presence of the sparse salient structures embedded in a block.

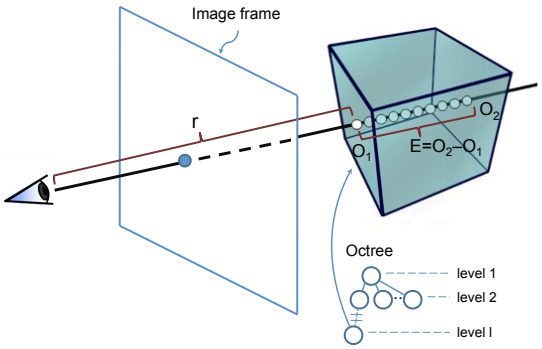


Fig. 3. Priority function terms

Coarse blocks should have a higher priority than finer blocks because their volume and corresponding image footprint are larger. For example, the screen footprint of a block at level l is roughly the quarter of that of a block at level $l+1$. More generally, the priority of a block at level l is inversely proportional to $(2^l)^2$. If a certain level corresponds to a resolution that is sufficient for a good rendering quality we should penalize the priority of finer levels to prevent a split when this level is exceeded and save space for possible new blocks. Assuming that the best current level is m, we therefore divide the priority value by the factor $(2^{(l-m)})^2$. The different factors are illustrated in Figure 3, where $E = O_2 - O_1$ designates the change in opacity caused by the block traversal. Finally we arrive at the following expression for the priority of a block b with maximum color effect E:

$$P(b) = \frac{E}{r^2 * (2^{(l-m)})^2}$$

5.2 Priority Table

The link between octree leaves (both sampled and virtual) and their priority value is established through a *priority table*. This table stores the priorities of the leaves during the ray casting algorithm. When a ray crosses a leaf, we look up the corresponding entry in the priority table and update the priority of the leaf only if the newly computed priority exceeds the original value in the table.

In addition, the priority table is used to keep track of the vacant blocks in the block repository. Practically, each table entry contains three values corresponding to leaf reference, leaf priority, and associated block reference. Hence, the priority table contains as many entries as necessary to link all leaves (sampled and

virtual) to their priorities as well as to all blocks. Virtual leaves are given an invalid block reference while free blocks have table entries that contain an invalid leaf reference. As a result, a sort applied to the block reference field of the priority table entries will reveal the identity and number of free blocks for subsequent processing.

5.3 Optimality criteria

To define an optimal octree refinement, we introduce some convenient notations. First, we call *merge candidate* any internal node whose children are all leaves (sampled or virtual). Such nodes are sometimes referred to as *internal leaves*. The term merge candidate alludes to the fact that in the course of a dynamic modification of the octree, these nodes can be collapsed by merging their children to form a new leaf. The algorithm determining when such a collapse takes place will be discussed in Section 6. Observe that upon being merged such a node can either become a sampled or virtual leaf depending on the availability of sample information in its children. Hence, if a leaf has a virtual child, it will itself be virtual.

- $SL(T)$: set of sampled leaves of an octree T
- $VL(T)$: set of virtual leaves of an octree T
- $L(T)$: set of leaves of an octree T where $L(T) = SL(T) \bigcup VL(T)$
- $P(T)$: (total) priority

$$P(T) = \sum_{l \in SL(T)} P(l)$$

- $\overline{P(T)}$: average priority

$$\overline{P(T)} = \frac{1}{|SL(T)|} \sum_{l \in SL(T)} P(l)$$

- $\mathsf{Var}(T)$: priority variance

$$\mathsf{Var}(T) = \frac{1}{|SL(T)|} \sum_{l \in SL(T)} (P(l) - \overline{P(T)})^2$$

- $M(T)$: set of merge candidates
- N_B: total number of blocks

The optimal tree uses the limited available blocks efficiently such that the memory allocated to the various portions of the tree is commensurate with the priority of the corresponding regions. As stated previously, the priority value provides a quantitative assessment of the contribution of a given region to the visual representation. We define an optimal tree T_{opt} as a tree satisfying two conditions:

1) T_{opt} has maximum priority over all trees $\mathbf{T}$

$$P(T_{opt}) = P_{max} := \max_{T \in \mathbf{T}} P(T)$$

where $\mathbf{T}$ is defined as

$$\mathbf{T} = \{T \mid SL(T) \leq N_B\}$$

This condition guarantees that the data blocks hold data with the highest visual effect.

2) T_{opt} has the minimum variance among the trees with maximum priority.

$$\mathsf{Var}(T_{opt}) = \min_{T \in \mathbf{T},\, P(T) = P_{max}} \mathsf{Var}(T)$$

The smallest variance ensures the smoothness of the image. Given two trees of equivalent priority, it favors a more regular assignment of blocks to priorities: the memory space per priority unit is closer to the average across the different blocks.

6 Algorithm

We now describe the basic algorithmic strategy of our method as well as its practical implementation. The latter takes the form of a succession of computational steps designed to progressively increase the priority of the tree and rendering steps that display the information currently available. These various aspects are discussed next.

6.1 Heuristic

To meet the optimality conditions outlined in Section 5, we apply the A*–search algorithm [3]. Specifically, we find the optimal tree by first defining a heuristic lower bound on the number of operations needed to reach this optimum. The two conditions mentioned above are separate: the second condition implies a constrained optimization, once the first condition is satisfied. Hence we can use two different heuristics, which once combined define the underestimated total number of needed operations. Equivalently, we can carry out the search based on the first condition followed by a search based on the second condition. The heuristic function for the first condition is:

$$h_1(t) = \{v \mid v \in VL \wedge \exists\, s \in SL \; s.t. \; P(v) > P(s)\}$$

In other words, the number of operations needed is at least the number of virtual leaves that can replace sampled leaves and get transformed to normal sampled leaves.

For the second condition we apply the following heuristic:

$$h_2(t) = \{s \mid s \in SL \wedge \exists\, m \in M \; s.t. \; P(s) > P(m)\},$$

whereby SL is the split candidate list (since split candidates are leaves by definition) while M is the merge candidate list (refer to Section 6.2). $h_2(t)$ is an underestimate of the number of operations needed to satisfy constraint 2) because trading each merge node $m \in M$ with priority $P(m)$ with a split node $s \in SL$ with priority $P(s)$ such that $P(m) < P(s)$ will reduce the overall priority variance of the tree. Indeed, assuming that the priority of both nodes remains unchanged after this exchange

(see discussion below), the variance is decreased since $\mathsf{Var}(\{s_1,..,s_8,m\}) < \mathsf{Var}(\{s,m_1,..,m_8\})$, with $\forall i\, P(s_i) = P(s)/8$ and $\forall i\, P(m_i) = P(m)/8$.

The description above relies on the assumption that the priority of the merge candidate is equivalent to the sum of the priorities of its children. However, the definition of the priority given in Section 5.1 does not strictly guarantee this invariance. To explain why this assumption is justified it is necessary to consider the makeup of the priority function. First, the difference in levels between a node and its children implies that the parent will have a screen footprint roughly four times larger than any of its children when a merge occurs. Second, since a ray crosses two children on average as it traverses the corresponding parent node, we can assume that the effect on the opacity of a ray color crossing the parent will be twice that of a ray crossing any child. Thus, given that the eight children have close priorities, the parent will have a priority that is almost eight times larger than the average child priority. At coarse resolutions, siblings will have close priorities and difference between merge and split candidates in priority will be significant which makes our assumption true. At finer resolutions, the decision will not be as accurate. However, since re-evaluation occurs in each rendering phase and therefore after only one operation is performed per node (merge or split), and because the priority difference between merge and split is insignificant, we make sure that the effect is visually unnoticeable.

6.2 Iterative Solution

Assuming that an upper bound k is known on the number of operations that can be performed per frame we can restrict our considerations to an equivalent number of merge and split candidates to determine which node transformations will actually take place. Note that the creation of any sampled leaf during split requires to perform trajectory integration for both forward and backward FTLE, a computationally intensive operation dominating the whole computation phase. We refer to this number k as *consideration zone* and discuss it further in Section 6.3.1. Practically, a *collective list* is used to hold k split candidates (*i.e.* leaves) having highest priorities and k merge candidates (Section 5.3) having lowest priorities. This list is then used to decide which sampling operations must be performed next.

To further illustrate this process, let us now describe the operations performed over a sequence of frames in order to approach the target optimal tree. As stated previously, the operations should result in a monotonic decrease of the heuristic measure h_1 and h_2 mentioned previously. The first heuristic can be targeted by the following two operations:

1) **Convert a node from virtual leaf to sampled leaf.** This happens when the virtual node priority exceeds other nodes priorities in the collective list and when the redistribution of the frames in this list grants the needed empty data block to the node.
2) **Convert a node from sampled leaf to virtual leaf.** The new virtual leaf will retain the same priority as the original sampled leaf. However, if the view settings changed virtual leaves priorities are set to a high fixed value if it is hit by a ray in the new settings. This will force a split to happen and consequently will permit the re-evaluation of the actual priority of this leaf under the new camera setting.

For the second heuristic to monotonically decrease, we perform the merge and split operations after the redistribution of data blocks with preference to high priorities. These operations occur over a sequence of frames until no merge candidate has a priority less than any leaf.

In addition we do not allow a split unless the node has a priority larger than a certain limit. Otherwise splits would continue until all blocks are consumed, which would slow down subsequent frames when free blocks become necessary. We also force a merge when the priority of the internal node is considerably less than the same limit mentioned for splitting even if the required blocks are available. This allows us to free blocks when possible. This can be thought of as amortizing the efforts of merging over frames by doing some merges ahead of time if they are anticipated.

6.3 Implementation

Practically, the operations performed at each frame of the visualization are organized in a computation phase (during which flow map sampling and FTLE computation are performed) and a rendering phase.

6.3.1 Computation Phase

During the computation phase, we use parallel radix sort on the priority table in order to separate the entries corresponding to free blocks from the rest of the entries in addition to finding the highest priority nodes. After separation, we can easily find the number of free data blocks by counting. It is necessary to correct the map from the octree nodes to the priority table entries after the sort. We apply the same technique to the indirection pool map (previously discussed in Section 4) to separate empty and non-empty indirection grids. Again we need to correct the references from the internal nodes to the indirection pool map entries.

Before deciding what merge and split operations need to be performed in the computation phase, we check all sampled leaves for possible evacuation. If a sampled leaf is not visible (its priority has the lowest initialization value after the rendering is complete)

and has a level coarser than the current finest level by more than 3 then we convert it to a virtual leaf and free its block. The condition on the level is to avoid deleting samples that might be needed in the close future due to sudden zoom out or rotation. This is the step that makes sure blocks are saved to finer resolutions when the coarser resolutions in the tree are not visible. After this operation, we need to repeat the sorting operations mentioned above and to find the new number of free blocks.

In order to find the merge candidates, we search all internal nodes having only leaves as children, and we add them to the merge candidates lists. Each merge candidate is coupled with the number of blocks freed if the merge is performed. In addition, their priorities are computed as the sum of their children priorities. An exception is when the node has a level less than four; in this case the node is exempted from the merge. We then perform a sort to find the merge candidates of lowest priorities.

The consideration zone parameter k (previously mentioned in Section 6.2) is chosen according to the degree of parallelism available by the architecture and according to the execution time of a single operation. After identifying k, we add the k highest priority split candidates (found using the priority table) and the k lowest priority merge candidates (found using the merge candidate list) to the collective map and we sort it according to priorities. We sum the blocks that will be released when merging the k lowest priority merge candidates. We add this number to the number of free nodes to get the number of permissible blocks. We then reassign all permissible blocks to the entries of the list in decreasing order of priority until the blocks are exhausted. If a split candidate was included in the selected entries, it is marked for splitting. Similarly, if a merge candidate was not included in this priority-based selection, it is marked for merging.

Merge is performed by first updating the data structures. This is followed by copying data from children blocks to parent block. The copy operations are performed only if none of the children is a virtual leaf. After finding the starting index of the free blocks in the priority table, we assign offsets to leaves marked for splitting according to their position in the collective list.

We perform the split operations in 3 stages:

1) Update the data structures.
2) Compute the flow map for the new points. This operation is performed in a loop by streaming the vector maps, and performing the integrations corresponding to the time frame of these maps.
3) Perform FTLE computations for all the points. For every point we find the flow map gradient using the neighboring grid points. Replication of the block boundaries is necessary to find the neighbors.

After we have completed merge and split operations, we reassign entries in the priority table for all leaves. The following pseudocode illustrates the computation phase sequence:

1) `Sort priority table and indirection pool map`
2) `Find number of free blocks and needed blocks through reductions`
3) `Test sampled leaves for possible conversion to virtual leaves and convert when appropriate`
4) `Repeat sorting and find new number of free blocks`
5) `Construct merge candidate list and sort`
6) `Merge k highest priority split candidates and k lowest priority merge candidates in collective list of size 2k`
7) `Sort collective list`
8) `Find number of blocks used by merge candidates in the collective list and add it to number of free blocks`
9) `Redistribute this number on collective list`
10) `Perform merge and split operations and corresponding trajectory integration in forward and backward directions`
11) `Copy memory to update the block repository and rebind the textures, and re-initialize priority table for upcoming rendering`

6.3.2 Rendering Phase

The rendering phase is a conventional ray casting algorithm where we assign rendering to threads on a per ray basis. When a ray hits the volume the texture octree is traversed to identify the leaf from which to sample the FTLE value. To speed up the rendering, we hold references from children to parents in order to climb to the closest ancestor containing the new point. Normal top down traversal is then resumed from this common ancestor node. When the ray exits a block, we compute the difference in opacity due to the block and we evaluate the priority function. The result is then saved in the priority table if it is higher than the original priority value of the leaf in the table.

To enhance the responsiveness of our visualization, we perform the computation phase every few frames (*e.g.*, in every third or fourth frame) whenever the interaction is high. Since we do not allow the merge of the first (coarser) three levels, the renderer is guaranteed to have enough data to produce an image that shows the overall structure at all times. Hence, when the user starts interacting with the system the frame rate will increase due to the computation phase skipping. Once the interaction stops the frame rate will drop to provide an accurate image in the shortest time. Another way to enhance responsiveness is to decrease the sampling rate by a factor of 6 (value chosen empirically) while interactions are heavy. The sampling rate across blocks is constant and hence is

variable along the ray. This is a consequence of blocks having different resolutions corresponding to leaves at various depths in the octree. The sampling distance along the ray is half the distance between samples in the block. This satisfies the required Nyquist rate necessary to benefit from the computed samples.

7 Results and Discussion

We have implemented our method on the GPU using NVIDIA CUDA. Both phases, computation and rendering, are broken into small parallel tasks to ensure maximum code path coherency among threads and take advantage of the parallelism. We have benchmarked our algorithm on a Intel Core2 Extreme QX9650 (12MB,3.0GHz) machine, whereby the GPU kernels are executed on NVIDIA GeForce GTX 280 (30 multiprocessors, 30720 threads, 1GB device memory).

The selected block size is 16^3. This size is chosen as a compromise between the total number of blocks needed to represent the different levels of refinements and the efficiency of the ray casting. A smaller block size will increase the number of blocks traversed by a single ray and consequently the number of octree queries. Using a texture to hold the data is necessary because close rays are kept on the same core through thread assignment to rays. Updates are copied from the scratch pad (Section 4.2) after every computation phase. We also rebind the indirection pool texture to its memory structure at the same time to commit changes made to the octree structure.

The consideration zone parameter is selected according to the compute capability of the device and the current load. As operations are assigned per core, this number should be a multiple of the number of available cores (30 for the NVIDIA GTX 280 used in our implementation). Since only one operation fits on each core, this parameter should be a multiple of 30 to keep all cores busy. This is equivalent to manipulating $8k$ data blocks (each operation involves 8 children) during the split and merge. This k value however can be changed dynamically to increase the efficiency. For example, if the user is changing the viewpoint we can pick a small k to favor higher frame rates. In contrast, if the view is stable we can select a higher k value and compute more data per frame. We used a consideration zone of 30 for our tests. This amounts to around 20 million integration steps per computation phase. The lower threshold on priority is selected empirically to be equivalent to the priority of a block that is barely visible. This parameter is fixed across all datasets.

The benchmark cases considered in the following consist of the standard analytical ABC (Arnold-Beltrami-Childress) flow and of three unsteady CFD datasets described in Table 1. The ABC flow is a canonical test case, exhibiting complex turbulent structures [10]. In the PLATE dataset, fast and slow fluid layers mix after passing a thin plate; the shear induced by the differing velocities creates strong vortices. The JET1 and JET2 datasets describe a fluid flow injected into a medium at rest using high-speed jets. Here, initially large LCS quickly break down into very small turbulent structures. Characterizing these structures requires a high resolution in the computation and depiction of the FTLE field. Hence, the analysis of these datasets greatly benefits from our approach.

TABLE 1
Dimensions of the different datasets

Dataset	PLATE	JET1	JET2
Size	384 x 96 x 96	128 x 128 x 128	128 x 256 x 128

In order to measure the performance of our method we compute the average frame rate during a (conic) helix camera movement around the volume. The camera starts at a position such that the volume fills around 25% of the screen area and approaches the volume gradually until the volume is totally filling the screen. During this helix movement, the camera rotates around the volume several times with a speed corresponding roughly to a user exploration. The helix path allows us to test the ability of our method to accommodate a continuously changing viewpoint. Computations need to be performed continuously in a timely fashion in order to provide a smooth view around the volume. A usual interaction behavior will typically contain pauses, which allows computations to proceed without view changes. As a result the frame rate and responsiveness of the system increases due to a reduced computational load. Thus, our test cases can be interpreted as more demanding than a realistic application scenario.

We keep threads of spatial proximity on the same core to maximize chances for cache hits and hence decrease the time needed for texture fetches. Table 2 (a) presents the average frame rates for the helix movement described above when the early ray termination is turned off, and when it is turned on. The opacity transfer function is a smooth approximation of the step function. The viewport used is 1024×768 and the computations are performed every three frames, as described in Section 6.3.2. The advection takes place in space and time. We use as many time steps as can fit in memory to reduce the need for streaming. The early ray termination helps increase the frame rates in general but to various degrees. The frame rates speedup for the PLATE dataset is more substantial than the one achieved for the other datasets. This is due to the fact that the PLATE dataset contains strong (opaque) structures on the exterior, a configuration that best leverages the early ray termination.

For the advection of particles we use a Runge–Kutta 2 (RK2) scheme with a small step size (around 0.5% of the volume maximum side length). This scheme

TABLE 2
Impact of rendering modes (a), integration method (b), transfer function (c), and viewport size (d) on frame rates (fps)

		JET1	JET2	PLATE
(a)	Computation + Rendering	19	15	24
	+ Early Ray Termination	22	18	30
(b)	Euler	25	20	34
	Runge–Kutta 2	22	18	30
	Runge–Kutta 4	19	16	23
(c)	Smooth Step	22	18	30
	Constant Transparency	21	18	28
(d)	400×300	40	37	39
	640×480	31	28	38
	1024×768	22	19	30

works best with the GPU architecture because it requires a small number of registers hence allowing for a larger number of concurrent threads. In our tests, this method outperformed higher-order numerical methods with larger step size. In Table 2(b), we show the effect of using different integration methods on the performance while using the same step size. In Figure 6(c-d), we show the FTLE field for the ABC dataset with a time dependent term computed by our method with RK2 (c) in side by side comparison with that computed on the CPU at fixed resolution (512^3) with the Dormand–Prince (DOPRI) integration scheme (d). All major structures are similar in both views.

While slight discrepancies can be spotted, they are in fact primarily due to the limited (fixed) resolution used for the CPU computation (a limit imposed by memory constraints in the subsequent rendering stage). Indeed, we validated our results at high resolution by comparing the images obtained on GPU and CPU with the same integration scheme (RK2), whereby we computed only the visible portion of the volume at high resolution on the CPU. We then extracted 2D ridges from the resulting images and measured the corresponding average CPU/GPU distance (using a simple gradient-descent approach). That distance was found to be under a pixel width in all tested cases, thus confirming the accuracy of our GPU results.

The opacity of a block affects its priority as described in Section 5.1. Hence the transfer function can affect the amount of computation performed. We used a 1D transfer function in our experiments though multidimensional transfer functions are possible. We compared the frame rates achieved with a smooth step opacity function and with a constant low opacity function covering the same range of values. The constant opacity is used to reveal the internal structures of the volume, thus increasing the computation load. Table 2(c) shows a comparison between the frame rates achieved with each transfer function. We notice a slight decrease in the case of uniform transparency, which is expected since less computation is required in outside blocks while more is necessary in inside blocks. This balanced effect led to a limited change in the frame rate. However, we observe that for a high zooming factor and with a transfer function consisting of a set of spikes the frame rate can drop to 7 fps. This is due to the fact that more computations are required to identify the optimal tree in which visible blocks are distributed to a wide range of resolutions. Similarly, the cost of rendering is increased, too. When the ray traverses blocks of significantly different resolutions, the corresponding thread needs to move up and down the tree for every change in resolution crossing a large number of nodes in each case (c.f. Section 6.3.2). This implies significantly more accesses to the octree nodes by the threads.

The amount of velocity data streamed from the CPU-side cache to the device is a critical factor in the overall performance of the computation phase. We compare 4 different cases. In the first case the flow is steady. No streaming is necessary and the cache performance is optimal since only one texture is held in memory. Now, let N be the maximum number of steps that can reside in memory simultaneously. The second case corresponds to an unsteady flow where the time steps used for both forward and backward advections can fit in memory simultaneously (N time steps). This case does not require streaming but will lead to decreased cache coherence compared to the previous case since different textures are accessed at different times. The third case corresponds to an unsteady flow for which the number of steps used for each advection direction can fit in memory. Hence, in this case we need to stream once per integration direction (N time steps for each direction). The last case corresponds to an unsteady flow for which the number of time steps needed for every direction corresponds to twice as many steps as the previous case ($2N$ time steps for each direction). Hence, we need to stream twice for every direction. In our implementation we leverage page locked memory to hold the datasets in the RAM. This leads to faster transfers using DMA. In addition, we use asynchronous streaming for the different components of the velocity data. The latest versions of NVIDIA hardware (*e.g.*, NVIDIA Fermi) should be able to interleave integrations with streaming. This is not possible with our current configuration because the cost of interrupting the integration kernel more frequently with a ping pong like technique is significant and cannot properly hide the streaming latency on our hardware. The number of time steps used for each case is summarized in Table 3(a). The resulting computation times are shown in Table 3(b) (total computation time and its percentage out of total time) and the resulting frame rates are presented in Table 3(c). Again, we are using the same viewport size

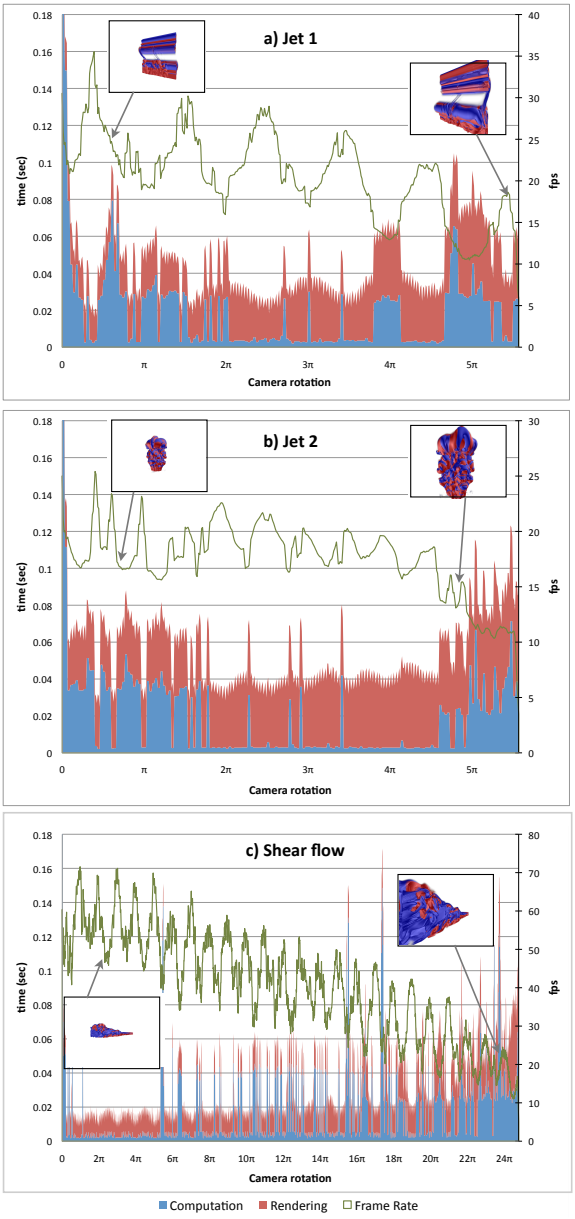


Fig. 4. Detailed behavior of the three datasets for the unsteady case without streaming.

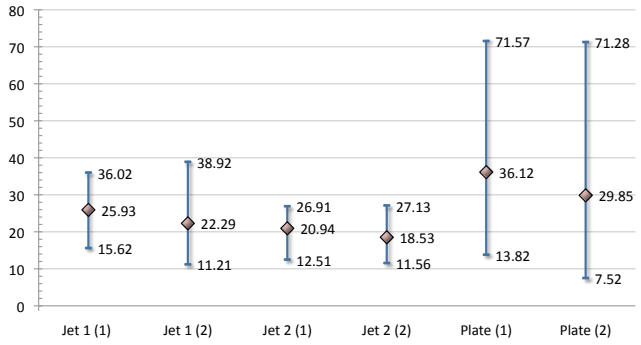


Fig. 5. Frame rates for the steady (labeled 1) and unsteady (labeled 2) cases without streaming

(1024×768) and camera path discussed previously. For the unsteady test cases with streaming, we augment the number of vector fields through interpolation in between original time steps. This ensures that the computations we perform and the obtained structures are strictly identical and hence the comparison reveals differences due to the streaming alone.

For the first 2 cases, we notice that the computation phase constitutes only a small percentage of the whole test duration. This is expected for two reasons. First, we only do computations that are necessary using an optimized strategy. Second, our rendering is more costly than the rendering of a typical 3D texture because our rendering includes the traversal of a hierarchical structure (octree) in addition to priority computations and corresponding memory writes. The priority for any block is set to the highest value computed by any ray. Computation, rendering times, and frame rates in the course of the animation for the unsteady case without streaming are shown in Figure 4. The computation phases appear as spikes since they are executed only every three frames. It can also be noticed that the computation time is low in certain ranges while high in subsequent ones. This can be explained by the fact that block resolutions remain sufficient for a while until a higher resolution is found necessary as the object approaches the camera. Low frequency oscillations with increasing times correspond to complete rotations around the object from successively shorter distances to the camera.

The frame rates achieved for the first two cases are shown in Figure 5. Notice that the resulting volumes for these cases are not strictly identical owing to transient variations but they remain similar because we use very close time steps. Hence, the difference reflects primarily the cache performance decrease when using a set of textures in the unsteady case compared to a single texture in the steady case. The percentage of the computation phase time increases significantly when we switch to the streaming cases. This is due to the limited bandwidth between RAM and device.

As we described previously, the percentage of rendering time relative to the whole animation time is high in the absence of streaming. To determine how this ratio scales with different viewport sizes, we repeat the second case with viewport sizes of 400×300 and 640×480 in addition to the 1024×768 viewport we used along our tests so far. The results are summarized in Table 2(d). Every viewport has 2.56 times more pixels than the directly following smaller viewport. The frame rate increases by only about 50 to 60% moving from the 1024×768 viewport to the 640×480 viewport. This is explained by the fact that the computation ratio increases as the viewport gets smaller. Also, as the viewport gets smaller, the cache coherence between ray texture accesses decreases. As we explained in Section 6.3.2, the sampling rate increases by a factor of 6 while the user is not changing

TABLE 3
(a) Number of time steps used in every case. (b) Effect of streaming on computation time and (c) frame rates.

	a			b			c		
	JET1	**JET2**	**PLATE**	**JET1** (s./%)	**JET2** (s./%)	**PLATE** (s./%)	**JET1**	**JET2**	**PLATE**
steady	1	1	1	5.41/27.3	6.24/25.5	24.61/39.2	25.93	20.94	36.12
unsteady + resident	11	6	6	9.29/40.11	9.48/34.06	38.73/50.58	22.29	18.58	29.85
unsteady + 2 x streaming	21	11	11	45.17/67.04	41.95/60.59	167.7/76.74	7.43	7.22	10.15
unsteady + 4 x streaming	41	21	21	82.37/79.1	77.68/74.69	295.79/88.86	4.84	4.83	6.72

the visual setting to achieve very high quality renderings. The difference between sampling rates is almost invisible unless a spiky opacity transfer function is used. We notice however that the frame rate decreases by a factor of around only 3 when the sampling rate is increased by 6. We explain that by the increased cache coherence when the sampling distance is shorter. In general, this reduced frame rate is not a concern for interactivity since the sampling rate automatically adjusts to the behavior of the user.

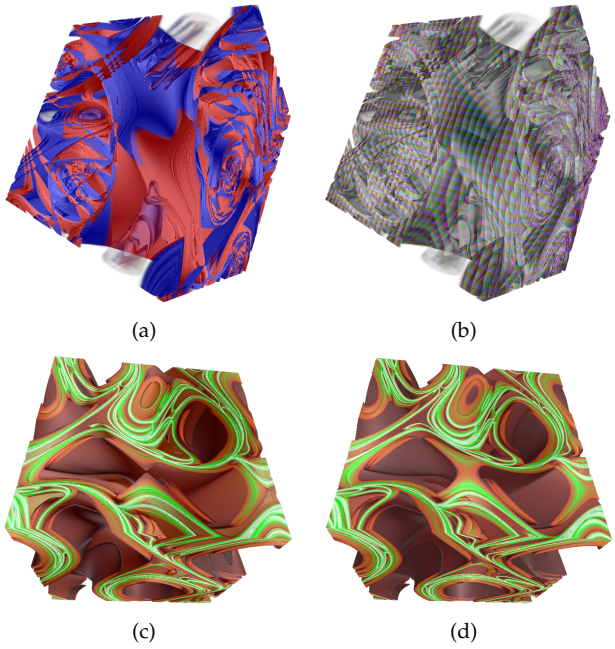


Fig. 6. (a) General view for the FTLE field of the ABC (*Arnold-Beltrami-Childress*) flow dataset (b) Corresponding block visualization for the same view. (c) Forward FTLE field using RK2 (d) Forward FTLE at fixed resolution computed on the CPU using DOPRI.

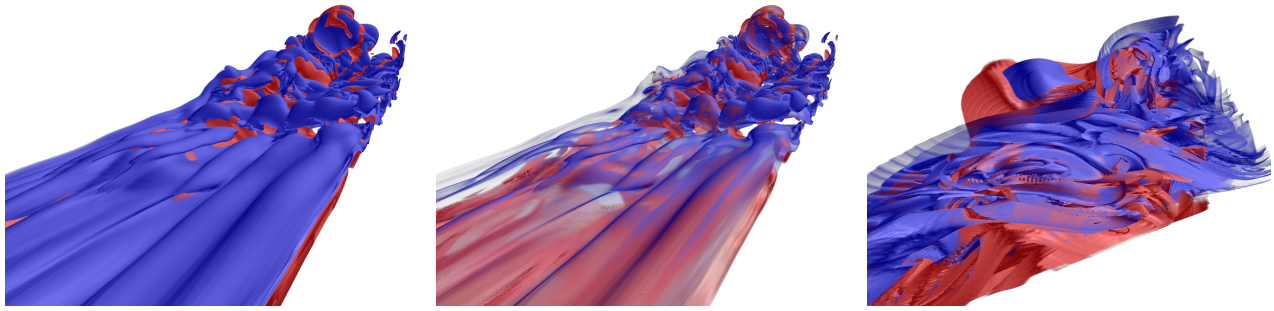
Figure 6(a-b) presents a high quality rendering of the ABC flow dataset along with the corresponding block subdivision (the local parameterization of each block is mapped to the RGB unit cube). Figure 7 shows two different opacity transfer functions applied to the PLATE dataset (step function vs. opacity peaks). In both cases the boundaries between blocks at different resolutions are invisible. Figure 8 documents the zooming capability for a clipped section of the JET2 dataset. Intermediate zooming steps are shown. Notice that as we zoom in we cross certain structures to reveal the smaller-scale ones located behind. Finally, Figure 9 illustrates the zooming applied to two different views of the JET1 dataset.

8 Conclusion

We have presented a novel and highly efficient approach for the visual exploration of flow structures at arbitrary resolutions through the interactive computation and rendering of the FTLE field in transient fluid flows. Our solution consists in intertwining computation and rendering stages. Specifically, our method restricts the costly numerical advection of particles associated with the FTLE computation to those regions of the domain whose mapping through the chosen transfer function contributes to the rendered image. The ray casting in the rendering stage is used to determine the optical properties of each block in an adaptive subdivision of the domain. Those properties are mapped to a priority value quantifying the visual impact of each block to rank their respective compute priority. This algorithm is enabled by a novel dynamic octree encoding introduced in this paper, which we use to constantly reallocate limited memory resources to the most relevant sections of the FTLE volume.

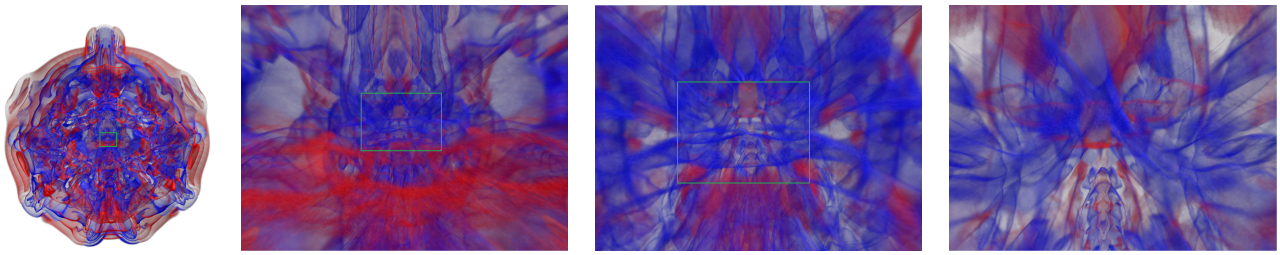
We have described a very efficient implementation of this approach on the graphics hardware, which fully leverages the massive compute power of modern GPUs. Finally, we have presented experimental results obtained by our system on 3 CFD datasets and a canonical analytical flow. Our numbers document the high performance of our method across a range of demanding scenarios. With its interactivity and its ability to efficiently navigate across spatial scales, our technique offers for the first time a visualization framework in which the powerful concept of FTLE can be used as a data exploration tool.

An interesting avenue for future work is to extend the current algorithm to allow for the explicit geometric characterization of LCS as ridges of the FTLE field. Ridge extraction is an involved procedure whose interactive computation on an adaptive spatial subdivision appears quite challenging. We also wish to explore improved streaming approaches and a better collaboration between CPU and GPU to permit the interactive exploration of time-dependent LCS in large-



(a) (b) (c)

Fig. 7. (a), (b), and (c) show different transfer functions applied to the FTLE field of the PLATE dataset.



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

Fig. 8. Overview of the FTLE field for JET2 with a cutting plane and different zooming steps.

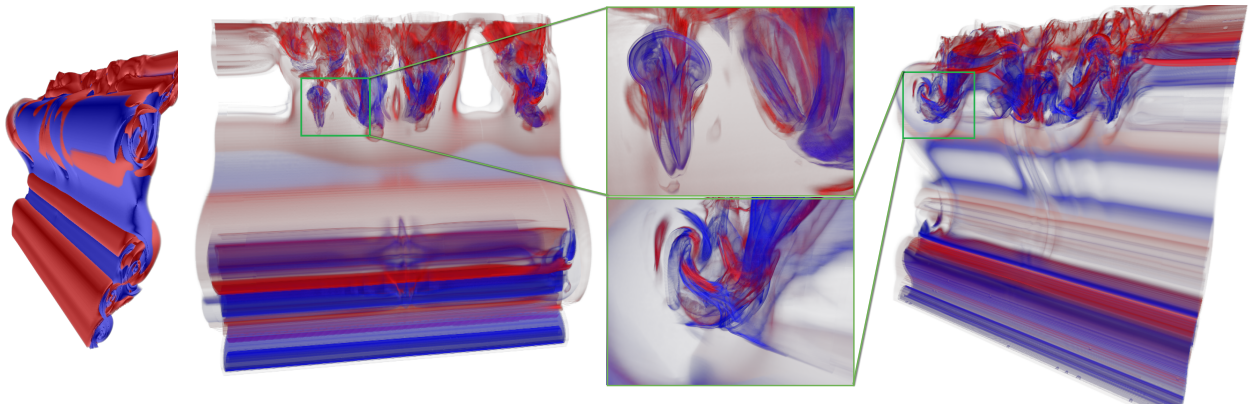


Fig. 9. Overview of the FTLE field for JET1 with two different zooming scenarios.

scale transient CFD simulations. This latter aspect will also raise some interesting questions pertaining to user interaction given the multidimensional (space, scale, time) nature of the exploration space.

Acknowledgments

This work was supported by an Intel PhD Fellowship, 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), and the National Science Foundation under grant IIS-0916289.

References

[1] I. Boada, I. Navazo, and R. Scopigno. Multiresolution volume visualization with a texture-based octree. *The Visual Computer*, 17(3):185–197, May 2001.

[2] S. L. Brunton and C. W. Rowley. Fast computation of finite-time lyapunov exponent fields for unsteady flows. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 20(1):017503, 2010.

[3] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms, 2nd Edition*. The MIT Press, 2001.

[4] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. Gigavoxels: ray-guided streaming for efficient and detailed voxel rendering. In *I3D '09: Proceedings of the 2009 symposium on Interactive 3D graphics and games*, pages 15–22, New York, NY, USA, 2009. ACM.

[5] T. Fogal and J. Kruger. Tuvok, an architecture for large scale volume rendering. In M. Dogget, S. Laine, and W. Hunt, editors, *Proceedings of the 15th International Workshop on Vision, Modeling, and Visualization*, pages 57–66, June 2010.

[6] C. Garth, F. Gerhardt, X. Tricoche, and H. Hagen. Efficient computation and visualization of coherent structures in fluid flow applications. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1464–1471, 2007.

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

[8] C. Garth, G.-S. Li, X. Tricoche, and C. D. Hansen. Visualization of coherent structures in transient 2d flows. In *Topology-based Methods in Visualization II, Mathematics + Visualization*, pages 1–14. Springer, 2009.

[9] M. Green, C. Rowley, and G. Haller. Detection of lagrangian coherent structures in 3d turbulence. *J. of Fluid Mech.*, 572:111–120, 2007.

[10] G. Haller. Distinguished material surfaces and coherent structures in three-dimensional flows. *Physica D*, 149:248–277, 2001.

[11] G. Haller. Lagrangian structures and the rate of strain in a partition of two-dimensional turbulence. *Physics of Fluids*, 13(11):33653385, 2001.

[12] G. Haller. Lagrangian coherent structures from approximate velocity data. *Physics of Fluids*, 14(6):1851–1861, june 2002.

[13] M. Hlawatsch, F. Sadlo, and D. Weiskopf. Hierarchical line integration. *Visualization and Computer Graphics, IEEE Transactions on*, 17(8):1148 –1163, aug. 2011.

[14] J. Kasten, C. Petz, I. Hotz, B. Noack, and H.-C. Hege. Localized finite-time lyapunov exponent for unsteady flow analysis. In M. Magnor, B. Rosenhahn, and H. Theisel, editors, *Vision, Modeling, Visualization*, pages 265–276, 2009.

[15] J. Kniss, A. Lefohn, R. Strzodka, S. Sengupta, and J. D. Owens. Octree textures on graphics hardware. In *ACM SIGGRAPH 2005 Sketches*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[16] B. Laramee, J. van Wijk, B. Jobard, and H. Hauser. ISA and IBFVS: Image space based visualization of flow on surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 10(6):637–648, nov 2004.

[17] R. Laramee, H. Hauser, H. Doleisch, B. Vrolijk, F. Post, and D. Weiskopf. The state of the art in visualization: Dense and texture-based techniques. *Computer Graphics Forum*, 23(2):143–161, 2004.

[18] R. S. Laramee, H. Hauser, L. Zhao, and F. H. Post. Topology-based flow visualization, the state of the art. In Mathematics and Visualization, editors, *Topology-Based Methods in Visualization II*, pages 1–19. Springer-Verlag, 2007.

[19] F. Lekien and S. D. Ross. The computation of finite-time lyapunov exponents on unstructured meshes and for non-euclidean manifolds. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 20(1):017505, 2010.

[20] D. Lipinski and K. Mohseni. A ridge tracking algorithm and error estimate for efficient computation of lagrangian coherent structures. *Chaos: An Interdisciplinary Journal of Nonlinear Science*, 20(1):017504, 2010.

[21] M. Mathur, G. Haller, T. Peacock, J. Ruppert-Felsot, and H. Swinney. Uncovering the lagrangian skeleton of turbulence. *Phys. Rev. Lett.*, 98:144502, 2007.

[22] F. Post, B. Vrolijk, H. Hauser, R. Laramee, and H. Doleisch. Feature extraction and visualization of flow fields. In *State-of-the-Art Proceedings of EUROGRAPHICS 2002 (EG 2002)*, pages 69–100, Sept 2002.

[23] F. Post, B. Vrolijk, H. Hauser, R. Laramee, and H. Doleisch. The state of the art in flow visualization: Feature extraction and tracking. *Computer Graphics Forum*, 22(4):775–792, 2003.

[24] D. Ruijters and A. Vilanova. Optimizing gpu volume rendering. In *Thirteenth International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (Winter School on Computer Graphics)*, pages 9–16, Feb. 2006.

[25] F. Sadlo and R. Peikert. Efficient visualization of lagrangian coherent structures by filtered amr ridge extraction. *IEEE Trans. Vis. Comput. Graph.*, 13(6):1456–1463, 2007.

[26] F. Sadlo and R. Peikert. Visualizing lagrangian coherent structured and comparison to vector field topology. pages 15–29. In Topology-Based Methods in Visualization, Proceedings of the 2007 Workshop, 2007.

[27] F. Sadlo and A. Rigazzi. Time-dependent visualization of lagrangian coherent structures by grid advection. pages 151–165. Springer Berlin Heidelberg, 2008.

[28] S. Shadden, J. Dabiri, and J. Marsden. Lagrangian analysis of fluid transport in empirical vortex ring flows. *Physics of Fluids*, 18:047105, 2006.

[29] S. Shadden, F. Lekien, and J. Marsden. Definition and properties of lagrangian coherent structures from finit-time lyapunov exponents in two-dimensional aperiodic flows. *Physica D*, 212:271–304, 2005.

[30] S. Shadden, F. Lekien, and J. Marsden. Lagrangian coherent structures in n-dimensional systems. *Journal of Mathematical Physics*, 48(065404), 2007.

[31] B. Soni, D. Thompson, and R. Machiraju. Visualizing particle/flow structure interactions in the small bronchial tubes. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1412–1427, 2008.

[32] H. Theisel and H.-P. Seidel. Feature flow fields. In *Proceedings of Joint Eurographics - IEEE TCVG Symposium on Visualization (VisSym '03)*, pages 141–148, 2003.

[33] H. Theisel, T. Weinkauf, H.-C. Hege, and H.-P. Seidel. Topological methods for 2d time-dependent vector fields based on streamlines and path lines. *IEEE Transactions on Visualization and Computer Graphics*, 11(4):383–394, 2005.

[34] X. Tricoche, T. Wischgoll, G. Scheuermannn, and H. Hagen. Topology tracking for the visualization of time-dependent two-dimensional flows. *Computer and Graphics*, 26:249–257, 2002.

[35] G. A. Voth, G. Haller, and J. P. Gollub. Experimental measurements of stretching fields in fluid mixing. *Physical Review Letters*, 88(25 Pt 1):254501, 2002.

Samer Barakat is a Ph.D. student in Computer Science at Purdue University. He received a MSc in computer engineering from Alexandria University, Egypt in 2007 and a MSc in computer science from Purdue University in 2011. His research interests include interactive visual analysis of ultrascale flow phenomena on parallel architectures, extraction and visualization of 3D structures in volumetric medical and engineering datasets, and query-driven visualization.

Christoph Garth received a Ph.D. (Dr. rer. nat.) in Computer Science from the University of Kaiserslautern in 2007 and spent his postdoctoral time as a researcher with the Institute for Data Analysis and Visualization at the University of California, Davis. He is currently an assistant professor in Computer Science at the University of Kaiserslautern. His research focuses on scientific visualization, analysis of vector and tensor fields, topological methods, query-driven visualization, and parallel/scalable algorithms for visualization.

Xavier Tricoche is an Assistant Professor of Computer Science at Purdue University. He was previously with the Scientific Computing and Imaging Institute at the University of Utah. He received a MSc in Computer Science from ENSIMAG ('98), a MSc in Applied Mathematics from Grenoble University ('98), in France, and a PhD in Computer Science ('02) from the University of Kaiserslautern, Germany. His research focuses on the scalable structural and visual analysis of multivariate datasets. His work has found recent applications in fluid dynamics, solid mechanics, high-energy physics, bioengineering, and medical image analysis.