EFFICIENT CONSTRUCTION OF FLOW STRUCTURES

Tobias Salzbrunn, Alexander Wiebel and Gerik Scheuermann
Institut für Informatik, Universität Leipzig
PF 100920, D-04009 Leipzig, Germany
email: {salzbrunn|wiebel|scheuermann}@informatik.uni-leipzig.de

ABSTRACT
Visualizing flow structures according to the users' interests provides insight to scientists and engineers. In previous work, a flow structure based on streamline predicates, that examine, whether a streamline has a given property, was defined. Evaluating all streamlines results in characteristic sets grouping all streamlines with similar behavior with respect to a given predicate. Since there are infinitely many streamlines, the algorithm chooses a finite subset for the computation of an approximated discrete version of the characteristic sets. However, even the construction of characteristic sets based on a finite set of streamlines tends to be computationally expensive. Based on a thorough analysis of all processing steps, we present and compare different acceleration approaches. The techniques are based on simplifications that result in characteristic set boundaries deviating from the correct but computational expensive boundaries. We develop measures for objective comparison of the introduced errors. An adaptive refinement approach turns out to be the best compromise between computation time and quality.

KEY WORDS
Scientific and mathematical visualization, flow visualization, feature detection

1 Introduction

Flow visualization is an important topic in scientific visualization. Many of its well-established methods are widely used in science and various engineering disciplines. Especially feature-detection methods like the λ_2-criterion for vortex core regions proposed by Jeong and Hussain [1] or the vortex core line extraction approach introduced by Sujudi and Haimes [2] are extensively used. However, what users really want to understand is the overall behavior of the flow in connection with these features. For this task, Salzbrunn and Scheuermann [3] recently introduced the notion of flow structures based on streamline predicates. The basic idea of this approach is a partition of the flow based on user-defined properties. Predicates on streamlines evaluate whether a streamline has a given property. In a next step, all streamlines fulfilling the predicate are grouped and form the characteristic set of this predicate. This can be done for as many predicates as are needed for a satisfying description of the behavior of interest. A set

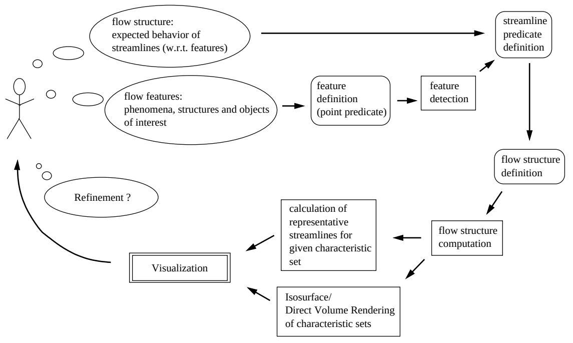

Figure 1. Framework of flow structure methodology (taken from [3]).

of predicates such that every streamline fulfills exactly one predicate defines a partition of the whole flow. Such a partition is called a *flow structure* - a structure tailored to the users needs! We presented a framework explaining how to work with flow structures in [3]. Fig. 1 shows the interplay of features, streamline predicates and flow structures. In most cases, users know which features are important for them. These features influence streamline behavior. So, after the specification and identification of features, the users formulate the behavior of streamlines of interest as streamline predicates.

As vector fields contain infinitely many streamlines a finite subset has to be chosen to be able to compute an approximate discrete representation of the characteristic sets (see Sec. 3). Unfortunately, even this process tends to be computationally expensive for a reasonable resolution if computed in a brute force manner. Hence, a thorough analysis of all processing steps is needed, in order to exploit all acceleration potentials. Based on such an analysis, we present different acceleration approaches and compare them by their time cost and accuracy. We also take their memory demands into account. As basis for the accuracy-based comparison, we define quantities measuring the errors that are introduced by the different strategies.

2 Related Work

One of the acceleration strategies presented throughout this paper produces AMR (adaptive mesh refinement) data. For such data standard isosurface extraction schemes (e.g. [4] or [5]) may produce surfaces with cracks. A scheme designed to prevent such cracks was presented by Weber *et al.* [6]. It can be applied for the AMR data if needed.

In Sec. 7 we define an error rate to compare the different acceleration techniques. As the error is influenced by the shape of the surfaces we additionally define a measure for their complexity. This enables us to interpret the error values in the correct context. A similar but more complex measure was introduced by Bribiesca [7]. His *discrete compactness* is a description of complexity and compactness of surfaces. We found our measure to be simpler and sufficient for our case.

3 Flow Structures

In this section, we shortly review the formalism needed to construct characteristic sets. Let $D \subset \mathbb{R}^3$ be the domain of our steady three-dimensional flow. Let its ***velocity field*** be represented by a Lipschitz continuous map $v : D \rightarrow \mathbb{R}^3,\ x \mapsto v(x)$. A ***streamline*** of v passing through a point $x \in D$ is a map $S_x : I_x \rightarrow \mathbb{R}^3$ where $0 \in I_x \subset \mathbb{R}$ is an interval of maximal extent and $S_x(0) = x$ and $\partial S_x(t)/\partial t = v(S_x(t)),\ \forall t \in I_x$.

A ***streamline predicate*** SP is defined as a Boolean map on the set of streamlines $\mathscr{S}$, i.e.

$$\begin{aligned} SP : \mathscr{S} &\rightarrow \{TRUE, FALSE\}, \\ S &\mapsto SP(S). \end{aligned}$$

The corresponding ***characteristic set*** $A_{SP} \subset D$ is defined as

$$A_{SP} = \bigcup_{S_x \in \mathscr{S},\ SP(S_x) = TRUE} S_x(I_x)$$

To set up the flow structure, we need a grouping mechanism that creates a finite number of groups of streamlines with common properties. Our mechanism assumes a finite set of streamline predicates $\mathscr{G}$ with disjunct characteristic sets, i.e.

$$\mathscr{G} = \{SP_\lambda \mid \lambda \in \Gamma\}, \quad A_{SP_\lambda} \cap A_{SP_\mu} = \emptyset \ \forall \lambda, \mu \in \Gamma,\ \lambda \neq \mu,$$

where Γ is an index set. We define a ***flow structure*** to be the following partition: $D = \bigcup_{SP_\lambda \in \mathscr{G}} A_{SP_\lambda}$.

3.1 Visualization

Isosurfaces or direct volume rendering can be used to visualize the borders of the different characteristic sets. While volume rendering often provides a better overview of the whole partition, isosurfaces, when stored, can serve as basis for further processing steps. Hence, as we need the boundaries of the characteristic sets for our acceleration strategies, isosurfaces will be used throughout this paper.

One should notice that the process chain (Fig. 1) is cyclic: insights gained in the visualization step are used to improve streamline predicates for a next iteration. As we will see later, the amount and the kind of reuse strongly influence the choice of a acceleration strategy.

3.2 Example

A simple example of a streamline predicate is the deviation of the flow from the principal in-flow direction. We discuss this predicate to illustrate the concept of flow structures here, and will use it to compare different acceleration strategies later. We obtain the deviation by integrating the difference between the streamline's tangent vector direction and the main in-flow direction along the streamlines. For the discretization we sample a representative finite subset $\tilde{\mathscr{S}}$ of all streamlines $\mathscr{S}$ using a Cartesian grid for the starting positions. Then we compute the deviation and choose a minimum threshold for the deviation. This defines the streamline predicate:

$$D \,\hat{=}\, (\textit{"Deviation of } \tilde{S} \textit{ from given direction"} > d_{min})$$

The discretized characteristic set is comprised of all voxels visited by a streamline fulfilling the respective streamline predicate. For a formal description of this kind of discretization we refer the reader to [3].

We apply the deviation predicate to a dataset from a simulation of the flow around a sphere with a drilled hole in the center. The dominant flow pattern is a group of three ring vortices on the lee side of the sphere. The sphere has a diameter of 200. The underlying unstructured grid contains 2.5 million tetrahedra. For the finite subset $\tilde{\mathscr{S}}$ we use a Cartesian grid in the area $[-250, 250] \times [-125, 125] \times [-125, 125]$ with a spacing of 3.125 (6.25) in all directions as starting positions. This is a set of 1056321 (136161) streamlines that densely fills the space around the ball. We apply the predicate to this dataset with $d_{min} = 0.6$ and $d_{min} = 0.1$. Fig. 3 shows the boundary of the resulting characteristic set A_D for both thresholds for d_{min}. The simple flow structure $\mathscr{G}_{Dev} = \{\ A_D, A_{\bar{D}}\ \}$ is sufficient as a benchmark for the different acceleration approaches.

4 Factors Determining Computation Time

The construction process of the characteristic set of a given streamline predicate is built up of several tasks. For each of the tasks the respective contribution to the overall computation time can be determined.

$$\begin{array}{lcll} \textit{compTime} & = & \left.\begin{array}{c} \textit{streamline integration} \\ \textit{and rasterization} \\ + \\ \textit{streamline evaluation} \end{array}\right\} & \begin{array}{c} \textit{core} \\ \textit{computational} \\ \textit{part} \end{array} \\ & & \quad + & \\ & & \left.\textit{isosurface generation}\ \right\} & \begin{array}{c} \textit{visualization} \\ \textit{part} \end{array} \end{array}$$

Concerning streamline integration there is the known tradeoff between speed and accuracy. The required accuracy depends on the streamline predicate. The effort to rasterize the streamlines depends on the resolution of the discretization of the characteristic sets: an increasing resolution of the grid yields an increasing computation time for the voxelization of a streamline. To evaluate a predicate for a streamline, the streamline needs to be sampled at some points. The computation time for the evaluation depends on the number of samples used for the evaluation and the time needed for processing one sample. The visualization part is the last step in the process chain. It builds upon the finished predicate evaluations and, hence, is independent of acceleration approaches applied in the previous steps. Nevertheless, the creation of the isosurface can consume considerable time. This depends on the number of voxels that have to be examined.

The discussion in the previous paragraph shows that the most promising idea for reducing computation time is to compute and evaluate as few streamlines as possible when creating a characteristic set with a prescribed quality. Acceleration approaches aiming in this direction will be presented in Sec. 6. In the next section we will discuss possible ways to speed up the computation if using all streamlines starting at a regular sampling grid.

5 Using Precomputed Data

All ideas presented in this section are based on an *off-line* computation of data (e.g. streamlines) that can be used in many steps of the cyclic process chain (Fig. 1). Given a sufficient integration scheme accuracy, streamlines need to be computed only once for the whole process chain. They can be stored and reused in different parts of the process chain and for the evaluation of different streamline predicates.

The straight-forward approach for off-line computation is to precompute streamlines starting at positions on a regular grid. Obviously a critical parameter of this approach is the grid resolution. It should be fine enough to represent all structures one wants to capture by a streamline predicate. Data on a grid with step size δ can represent only structures that have at least a size of 2δ. It should be noted that different streamline predicates are able to capture structures of different granularity.

The computation of streamlines for a given set of starting points can be perfectly distributed to different computing nodes. For computing more than one streamline predicate with the same resolution of starting points, it is useful to precompute also the rasterizations of all streamlines and not only the streamlines themselves. For parameterized predicates that use thresholds on scalar quantities (e.g. integral of values like vorticity, deviation,… along streamlines) it is useful to save these quantities as attributes for each streamline. For all but the first threshold a streamline predicate evaluation thus can be reduced to a comparison of the stored attributes and the threshold. This is especially useful for quantities where are meaningful threshold is not known a priori. In such a case, a sweep through a range of threshold values is needed.

If a predicate evaluates true in case of attributes greater than a threshold, the visualization of the resulting characteristic set can be transformed into an isosurface-problem by saving the maximum attribute values of streamlines visiting a voxel for all voxels. Using the threshold as iso-value leads to an approximation of the characteristic set's boundary (with a maximal deviation of the size of one voxel). Sweeping an isosurface through a range of threshold values can be accelerated by common isosurface acceleration techniques like span-space (see e.g. [5]). Saving additional rasterizations and streamline attributes as suggested here increases the needed disc space. However, this increase is negligible as it is far from the storage demands of the streamlines (see Sec. 8).

The above user scenario does not need streamlines for the computation of the flow structure. This permits to keep the rasterization and streamline attribute data in main memory which results in a further speed-up. Even if streamline data is needed, it is useful to store as many streamlines in main memory as possible. A useful strategy is to experiment with different streamline predicates with low resolution and in-core computation before computing the final high-resolution version of the chosen flow structure using out-of-core computations.

The techniques described in this section are only useful if more than one predicate are to be evaluated. If computing only the characteristic sets for one streamline predicate the additional time for storing/loading streamlines and for computing many unnecessary streamlines (depending on the streamline predicate) is wasted.

6 Acceleration Approaches

In case the user does not want to use precomputed data, calculations can be accelerated by using only a subset of the streamlines used in the brute-force approach. In [3], Salzbrunn and Scheuermann presented different acceleration approaches. As one of these approaches, a skipping technique, can be combined with the adaptive refinement we develop in Sec. 6.1, we recall the basics of this acceleration approach in the following.

For the skipping technique, streamlines starting at voxels already visited by previous streamlines are skipped (”unconditional skipping”). If a streamline with negative evaluation of a predicate passes many starting positions of streamlines with a theoretical positive evaluation, many elements of the characteristic set can be missed. This can happen, if a streamline belongs to a border case of a predicate evaluation where the streamline is evaluated to false only due to small numerical inaccuracies. To handle this problem the condition for skipping a voxel can be changed such that starting positions are only skipped in case a streamline with positive predicate visited the voxel (”positive skipping”). This prioritizes positive predicates in comparison to negative ones. While unconditional skipping clearly is faster, positive skipping produces less errors. This is confirmed by our experiments (see Sec. 8).

Another simple idea for accelerating the computations is to start streamlines on a grid that is coarser than the one used for the rasterization. We refer to this approach

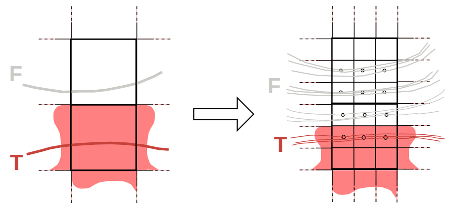

Figure 2. Adaptive refinement of a border voxel: magenta voxels belong to the characteristic set. Letters on the left of the images indicate evaluation of predicate: false and true.

as sparse seeding. This may result in holes in the characteristic sets, because of voxels not visited by any streamlines.

6.1 Adaptive Refinement of Characteristic Sets

We are mainly interested in the borders of the characteristic sets. They indicate where the behavior of the flow (with respect to given streamline predicates) changes. Hence it is useful to compute streamlines that are near the border case of positive or negative predicate evaluation, because they determine the borders of the characteristic sets. In order to get a first approximation of the region and voxels containing the border one can compute the characteristic set with a coarse resolution. The left image in Fig. 2 shows an example of such an border voxel. To compute the characteristic set with increased resolution every border voxel and its neighbor voxel not belonging to the characteristic set are subdivided into 27 sub-voxels[1]. The right image of Fig. 2 shows this subdivision in 2D. From the 54 sub-voxels of the two voxels (border voxel and neighbor) the 18 sub-voxels being closest (w.r.t. center) to the border are selected. A new streamline is started, evaluated and rasterized with new finer resolution for each of these 18 sub-voxels. All other sub-voxels and voxels not belonging to the border are directly assigned to the characteristic set in the new resolution.

Fig. 3 shows the deviations of the characteristic set A_D for all acceleration approaches described in this section.

7 Measuring Errors

The different accelerations techniques presented in Sec. 6 produce different discrete characteristic sets. We would like to use the characteristic sets produced by the brute force approach as a reference to compare the accuracy of the different acceleration strategies. Therefore, we need a measure for the deviation of characteristic sets. Let A_{SP} and $\tilde{A}_{SP}$ be the characteristic sets computed by usage of all streamlines and by using an acceleration approach respectively. As deviation measure we use the following ratio:

$$error\ rate = \frac{|(\tilde{A}_{SP} \cup A_{SP}) - (\tilde{A}_{SP} \cap A_{SP})|}{|A_{SP}|}$$

The lower the measure the better the discretization of the characteristic set. The error rate depends on the complexity of the surface of the characteristic set. A complex (extreme case: fractal) surface would in most cases result in a greater error rate than a smooth surface. Therefore, we have to take the shape of a characteristic set into account. A measure to describe shape complexity of a characteristic set is the ratio between the number of voxels of the characteristic set's boundary and the number of voxels of the characteristic set. The example characteristic set A_D with $d_{min} = 0.6$ ($d_{min} = 0.1$) has a complexity of 0.72 (0.46).

8 Results

Tables 2, 3, 4, and 5 show computation times and error rates for the two characteristic sets used as benchmarks. The figures are presented for two different voxel sizes. The computations were carried out on an AMD Opteron 224 (1.8 GHz) with 8 GB main memory. The figures show that "positive skipping" has the best error rates (especially for $d_{min} = 0.1$) but also a time cost near that of the "brute-force" approach. "Unconditional skipping" has very low time cost, but nearly unacceptable error rates, especially in case of high complexity characteristic sets. Even exploiting the short computing time to compute a higher resolution "uncond. skip hi-res" (voxel size 1.5625) does not improve accuracy enough to be able to compete with that of the other approaches. A good compromise between time cost and accuracy are "sparse seeding" and "adaptive". While "sparse seeding" has lower computing times but higher error rates, especially for low resolutions, "adaptive" has good error rates at cost of higher computing time. Combining both approaches improves accuracy to a level above that of the respective single methods and yields the best balance between time and accuracy.

Tables 2 and 4 additionally contain figures for the time cost of the strategies using precomputed data[2]. Clearly, using precomputed data cuts time cost dramatically. Of course, to generate the data, one has to invest the time of the "brute-force" approach once. Parallelization, i.e. distributing the computation to different computing nodes, can accelerate the generation. The shortest computing times can be achieved in the special but important case of parameterized streamline predicates (see Sec. 5). Their rasterizations (r) and attributes (a) are directly accessible from main-memory (in-core) or have to be load from hard-disk (out-of-core). All other streamline predicates need the streamline information (s) and additionally the rasterization for further acceleration. Table 1 summarizes the memory consumption for different resolutions.

[1]A subdivision into 8 sub-voxels does not allow the reuse of already computed streamlines, since the old voxel center is not a voxel center of any new sub-voxel.

[2]For A_D with $d_{min} = 0.1$ the values do not change significantly. We, thus, omit the corresponding figures. The same is true for "incore(s+r)" for the higher resolution due to main memory limitations.

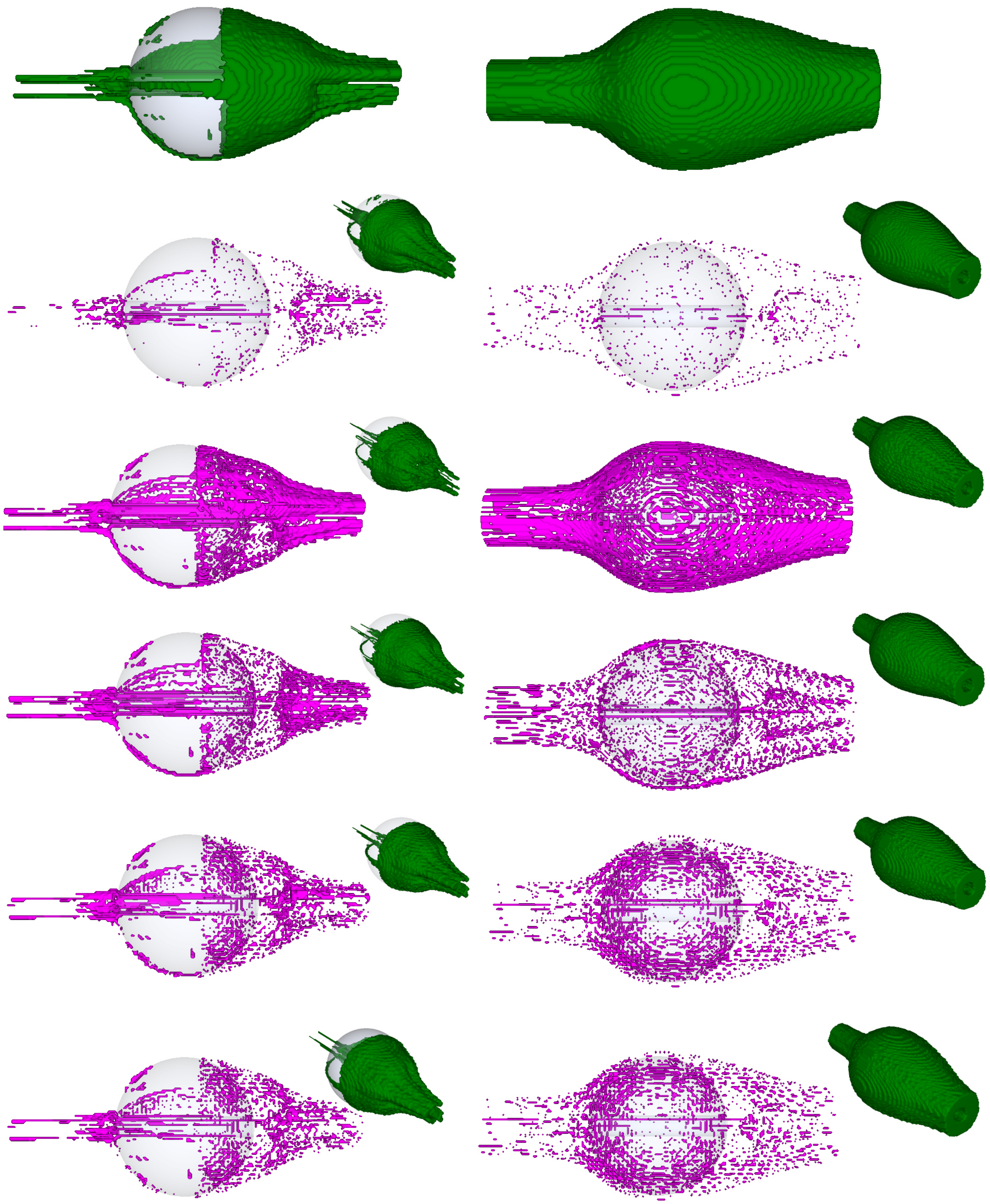

Figure 3. Comparison of different acceleration strategies for two exemplary characteristic sets ($d_{min} = 0.6$ (left) and $d_{min} = 0.1$ (right)) with voxel size 3.125. The upper row shows the original characteristic sets. The differences resulting from application of the different acceleration strategies are shown in the rows below. The small images in the upper right of the differences show the boundary of the characteristic set obtained with acceleration. The used acceleration strategies are, from second row to bottom: positive skipping, unconditional skipping, sparse seeding, adaptive refinement, and a combination of the last two approaches.

Resolution	Streamline Data	Rastered	Attributes
12.5	190 MB	2.8 MB	142 KB
6.25	1.4 GB	41 MB	1.1 MB
3.125	11 GB	616 MB	8.1 MB

Table 1. Memory needed to store streamline, raster and attribute data for different resolutions.

Obviously the acceleration still does not yield interactive cycling rates in the process chain. However, the *remaining* computation time is only a fraction of the original simulation effort, and can therefore go almost unnoticed.

9 Conclusion

In this paper, we examined the time cost of all parts involved in the process chain for computing a flow structure. Based on this analysis we investigated different acceleration approaches for different user scenarios. The adaptive refinement approach in combination with the sparse seeding turned out to be the best choice as it provides substantial acceleration while not reducing quality too much.

Acknowledgments

The authors would like to thank Markus Rütten from DLR for providing the ball dataset. This work was partly supported by DFG grant SCHE 663/3-7.

References

[1] J. Jeong and F. Hussain, On the Identification of a Vortex, *Journal of Fluid Mech.*, 285, 1995, 69-94.

[2] D. Sujudi and R. Haimes, Identification of Swirling Flow in 3-D Vector Fields, In *12th AIAA CFD Conference*, San Diego CA, June 1995.

[3] T. Salzbrunn and G. Scheuermann, Streamline Predicates, *IEEE Transactions on Visualization and Computer Graphics*, 12(6), 2006, 1601-1612.

[4] W. E. Lorensen and H. E. Cline, Marching Cubes: A High Resolution 3D Surface Construction Algorithm *Computer Graphics*, 21(4), 1987, 163-169.

[5] H.-W. Shen, C. Hansen, Y. Livnat, C. Johnson. Isosurfacing in Span Space with Utmost Efficiency. In *IEEE Visualization '96*, 1996, 287-294.

[6] G. Weber, O. Kreylos, T. Ligocki, J. Shalf, H. Hagen, B. Hamann, K. Joy, Extraction of Crack-Free Isosurfaces from Adaptive Mesh Refinement Data, *Approximation and Geometrical Methods for Scientific Visualization*, 2003, 19-40.

[7] E. Bribiesca. A measure of compactness for 3-D shapes, *Comput. Math. Applicat.*, 40:1275-1284, 2000.

Method	Computing Time	Error Rate
in-core (r+a)	1.27 [s]	-
in-core (s+r)	18.67 [s]	-
out-of-core (r+a)	1.78 [s]	-
out-of-core (s+r)	46.59 [s]	-
brute force	22.67 [min]	-
positive skipping	22.43 [min]	1715 (20.71%)
uncond. skipping	0.95 [min]	2531 (30.56%)
uncond. skip hi-res	5.62 [min]	1948 (23.53%)
sparse seeding	3.05 [min]	1943 (23.46%)
adaptive	8.53 [min]	1025 (12.38%)
combined	8.70 [min]	840 (10.14%)

Table 2. Comparison of different approaches for the construction of $\mathscr{G}_{Dev}$ with $d_{min} = 0.6$, voxel size 6.25, and complexity = 0.72.

Method	Computing Time	Error Rate
brute force	22.67 [min]	-
positive skipping	22.62 [min]	173 (0.74%)
uncond. skipping	1.0 [min]	3318 (14.24%)
uncond. skip hi-res	5.43 [min]	2741 (11.77%)
sparse seeding	3.05 [min]	2133 (9.16%)
adaptive	12.18 [min]	1362 (5.84%)
combined	12.35 [min]	1307 (5.61%)

Table 3. Comparison of different approaches for the construction of $\mathscr{G}_{Dev}$ with $d_{min} = 0.1$, voxel size 6.25, and complexity = 0.46.

Method	Computing Time	Error Rate
in-core (r+a)	9.6 [s]	-
out-of-core (r+a)	17.88 [s]	-
out-of-core (s+r)	5.73 [min]	-
brute force	3.09 [h]	-
positive skipping	3.02 [h]	1560 (2.88%)
uncond. skipping	0.09 [h]	12756 (23.56%)
uncond. skip hi-res	0.54 [h]	9387 (17.34%)
sparse seeding	0.38 [h]	5950 (10.99%)
adaptive	0.75 [h]	5042 (9.31%)
adaptive (2 It.)	0.50 [h]	7120 (13.15%)
combined	0.77 [h]	4128 (7.62%)

Table 4. Comparison of different approaches for the construction of $\mathscr{G}_{Dev}$ with $d_{min} = 0.6$, voxel size 3.125, and complexity = 0.72

Method	Computing Time	Error Rate
brute force	3.09 [h]	-
positive skipping	2.99 [h]	1110 (0.69%)
uncond. skipping	0.09 [h]	15427 (9.67%)
uncond. skip hi-res	0.62 [h]	11797 (7.39%)
sparse seeding	0.38 [h]	5751 (3.60%)
adaptive	0.97 [h]	5617 (3.52%)
adaptive (2 It.)	0.82 [h]	9306 (5.83%)
combined	0.99 [h]	5299 (3.32%)

Table 5. Comparison of different approaches for the construction of $\mathscr{G}_{Dev}$ with $d_{min} = 0.1$, voxel size 3.125, and complexity = 0.46.