Vision, Modeling, and Visualization (2012)
M. Goesele, T. Grosch, B. Preim, H. Theisel, and K. Toennies (Eds.)

Adaptive Treelet Meshes for Efficient Streak-Surface Visualization on the GPU

R. Fuchs[1], B. Schindler[1], R. Carnecky[1], J. Waser[2], Y. Jang[1], and R. Peikert[1]

[1]ETH Zurich, Switzerland, [2]VRVis Vienna, Austria

Abstract

We describe a novel adaptive mesh representation for streak-surfaces. The surface is represented as a mesh of small trees of initial depth zero (treelets). This mesh representation allows for efficient integration, refinement, coarsening and appending of surface patches utilizing the computational capacities of modern GPUs. Integration, refinement, and rendering are strictly separated into effectively parallelizable substeps of the streak-surface integration algorithm. We also describe a sampler framework which unifies the handling of different vector field representations.

1. Introduction

Turbulent fluids are characterized by their nonlinear behavior, resulting in quick mixing of the fluid particles. For material surfaces this means a constant streching and warping of the surface. Accordingly, the discretized representation, which is used to track the movement of the surface, has to be refined many times during integration. Streak-surfaces are a class of especially relevant material surfaces since they are related to time-dependent topological structures and can convey important topological information about the flow [SW10, FBTW10].

The computational model and memory access capabilities of modern GPUs are still not as flexible as that of a CPU. One instance of this problem is handling triangle-mesh connectivity changes on the GPU. Since the number of triangles adjacent to a vertex of a triangle mesh is arbitrary, large differences in the number of computations at individual vertices during parallel processing of the triangle mesh are possible. Therefore it is very often impossible to prove strong guarantees regarding the computational cost of an algorithm. Since triangle meshes are not optimally suited for processing on the GPU, many techniques resort to using unconnected quads or particles for surface representation. However, the lack of connectivity reduces the usefulness of the surface representation when it comes to further processing and analysis (e.g., computing surface curvature or flux). The central objective of this paper is a streak-surface representation which can be refined, coarsened and undergo connectivity changes in parallel on the GPU. To achieve this goal we make the following contributions:

- Streak-surface integration on the GPU featuring arbitrary levels of refinement based on a novel representation as a mesh of linked trees of low depth (treelets).
- The streak-surface integration is decomposed into a number of simple functions which parallelize well. All steps are of linear computational and memory complexity.
- A sampler framework which separates integration from data handling.

The evaluation shows that the algorithm scales linearly with the number of quads in the streak-surface.

2. Related Work

Streak-based visualization techniques are an important link to experimental flow visualization techniques. Krishnan et al. [KGJ09] present a high quality, high precision streak-surface integration algorithm which operates on the CPU and directly modifies a triangle mesh data structure using the flexibility of the CPU. This approach can generate high quality results but would be quite difficult to perform on the GPU efficiently. An approach to circumvent the difficulties of maintaining a triangle-mesh data structure is presented by Schafhitzel [Sch08]. He suggests a point-based approach where individual particles are integrated and then rendered by surface splatting. Similarly, Cuntz et al. [CKSW08] suggest a particle level set technique to compute streak volumes. McLoughlin et al. [MLZ10] present a streak-surface tech-

DOI: 10.2312/PE/VMV/VMV12/119-126

nique based on a mixed quad- and triangle-mesh where special T-junction objects can represent an additional type of connectivity in the mesh. Wiebel et al. [WTS*07] suggest extending streaklines by moving the seed point over time.

Many GPU-based approaches apply a trade-off between memory consumption and speed on one side, and flexibility on the other. Recently, Bürger et al. [BFTW09] present the first surface representation suitable for the GPU. This representation can handle iterated surface refinement and coarsening. It is based on the idea of reserving enough space for the highest resolution and leaving all memory unused where such a high resolution is not required. This allows for a certain degree of refinement at the cost of large proportions of unused memory in all places where the highest refinement level is not required. To be more precise, the memory layout requires $3 \cdot s \cdot m \cdot (2^R - 1)$ floating point variables in memory, where s is the number samples of the seed curve, R is the level of refinement, and m is the number of integration steps. We can see that the memory requirement is independent of which proportion of vertices is actually created for the streak-surface. The exponential coefficient will lead to a large proportion of unused memory when refinement is required only locally. Another trade-off is suggested by von Funck et al. [vFWTS08] who use a triangular mesh of fixed topology. This does not allow for refinement, but uses GPU-memory efficiently. Where the surface curvature becomes too high, transparency is increased. Ultimately, the most complex regions become fully transparent and are not integrated further.

Recently, Weinkauf et al. [WT10] presented a derived vector field in which streaklines can be computed by streamline integration. To create the derived vector field streaklines are seeded on a regular grid, sampling the whole 4D time+space domain densely. Based on this representation, streak-surfaces can be constructed using a stream-surface integration algorithm. This saves computational ressources at the cost of additional memory requirements, which is a good trade-off on the CPU. For streak-surface integration on the GPU this approach has several drawbacks: it introduces sampling artifacts, and is therefore especially unsuited for scattered SPH data. Furthermore, it samples the data onto a Cartesian grid, which adds additional data which needs to be stored. Most importantly, when computing a surface on-the-fly a the largest proportion of the derived field are not needed, since it does not contribute to the streak-surface of interest. For a more in-depth overview of the related work we suggest the state of the art reports by McLoughlin et al. [MLP*10] and Pobitzer et al. [PPF*11].

3. Surface Representation

In this paper, the surface is represented by two structures: first, an array of vertices (i.e., particles), which have positions in space and time; second, by a mesh of treelets which stores the topological information about which vertices are

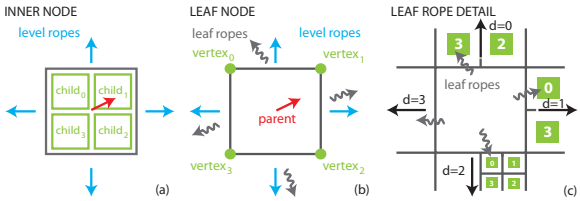

Figure 1: *Data structure representing leaf nodes and inner nodes of the treelet. (a) Inner nodes can store pointers to a parent node, to neighboring nodes on the same refinement level and pointers to four child nodes. (b) Leaf nodes represent quads. Level ropes point to one neighboring quad on each side. (c) There can be more than one neighbor leaf quad in each direction.*

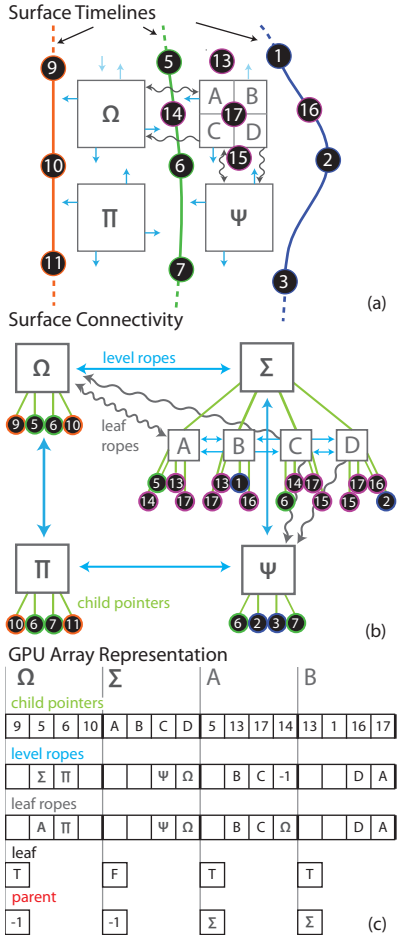

Figure 2: *Treelet mesh. (a) Three timelines (orange, green and blue) of the surface. (b) Hierarchical representation of the surface. Only leaf ropes which differ from level ropes are shown. The leaf node Σ at the vertices 1, 2, 6, 5 is refined once. (c) Representation on the GPU. −1 represents the NULL pointer, empty fields point to parts of the surface which are not shown in the illustration.*

adjacent to each other. A *treelet* is a small tree of linked nodes, which represent different levels of refinement hierarchically. The tree structure allows for straightforward local refinement of an individual node without global modification of the surface connectivity.

Treelet nodes are initially created as leaf nodes, in which case they represent individual quads of the surface mesh. During refinement, a quad is refined into four new quads. In this case, the leaf node becomes an inner node and the four new leaf nodes become its children. Each node in a treelet stores the following connectivity information: four *child pointers*, four *level ropes*, and four *leaf ropes*. The child pointers point either to the quad vertices (if the node is a leaf), or to other nodes (if the node is an inner node). The level ropes point to neighboring nodes on the same refinement level. Leaf ropes are only used in leaf nodes and point to neighboring leaves. This is important to avoid searching for neighboring leaves during updates (e.g., coarsening), as neighboring leafes might lie on different treelet levels. Figure 1 shows the different pointers on a single treelet node.

The indices to the children can either point to vertices (if the node is a leaf) or to other nodes (if the node is an inner node). The level ropes point to neighboring nodes on the same refinement level. Leaf nodes also store indices of neighboring leaves. This is important to avoid searching for neighboring leaves during updates (e.g. coarsening) of the streak-surface. In Figure 2(a), the adaptive streak-surface representation is illustrated in more detail. The seed curve, depicted in orange, is advected twice, creating two additional timelines, depicted in blue and green. The quad at the vertices 1, 2, 6 and 5 has been refined once to create the new vertices 13 to 17. Figure 2(b) shows a different representation of the same surface, where the links between quads are represented better: quads on the same level are linked by pointers providing connectivity to the mesh. We call these pointers level ropes since they connect nodes of the same level in the trees. Corresponding to the child pointers (green) connecting down the tree, there is also a *parent pointer* in the opposite direction. Figure 2(c) shows how the treelets are represented in linear memory on the GPU.

The GPU memory layout is as follows: we represent vertices as structures of four `float` values to store the position in space and the integration time (aka. lifetime) of a particle. Fetching four float values at the same time is implemented very efficiently on current GPUs. All vertices are stored in a consecutive array on the GPU and new vertices are added to the end of the array using an implementation similar to `std::vector`. The nodes are represented by multiple arrays storing child indices, parent indices, level ropes and leaf ropes separately. This way we can fetch the properties of a quad independently. Since the properties of leaf nodes and inner nodes can be stored in the same data structure they all reside in a common linear array in GPU memory. On current GPU architectures each node requires 13 bytes of memory and each vertex requires 16 bytes of memory.

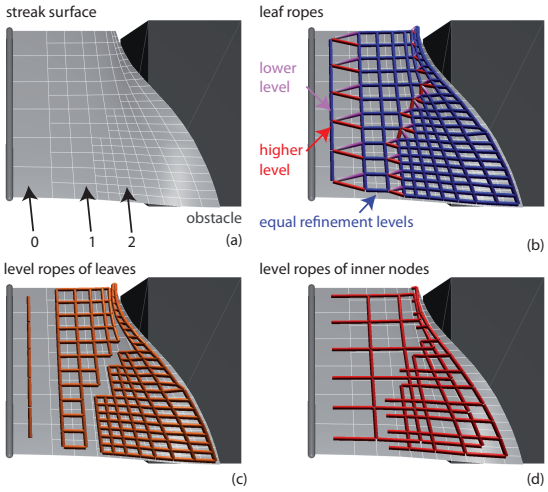

Figure 3: *Ropes and refinement. (a) The surface is refined twice. We use a coarse mesh for illustration purposes. (b) Leaf ropes to neighbors of the same refinement level are shown in blue. Uni-directional leaf ropes pointing upwards are shown in magenta at their starting point. Bi-directional leaf ropes between neighbors of different levels are shown in red. (c) Level ropes between leaf nodes. (d) Level ropes between non-leaf nodes.*

Figure 3 shows a concrete example of a refined surface and the ropes providing connectivity in the surface. In Figure 3(b), we can see the leaf ropes connecting the leaf nodes of the surface. The blue lines represent leaf ropes connecting nodes at the same level. Red lines represent leaf ropes which connect leaf nodes of different refinement levels bi-directionally. The magenta ropes point upwards unidirectionally from nodes in a deeper level of refinement towards larger nodes of smaller refinement level.

4. Algorithm

In this section we describe the streak-surface integration algorithm. We first explain the steps of the algorithm in general terms. The following subsections provide additional detail on how each step modifies the treelet mesh.

Overview The algorithm starts with an initialization phase during which the seed curve is advected once and the first row of quads is set up in memory. After that, the algorithm proceeds in five major steps: integration, refinement, coarsening, compaction and rendering. See Figure 4 for an illustration of these steps:

0. During initialization the first row of quads is created.
1. Integration moves all vertices one step and creates one row of leaf nodes.

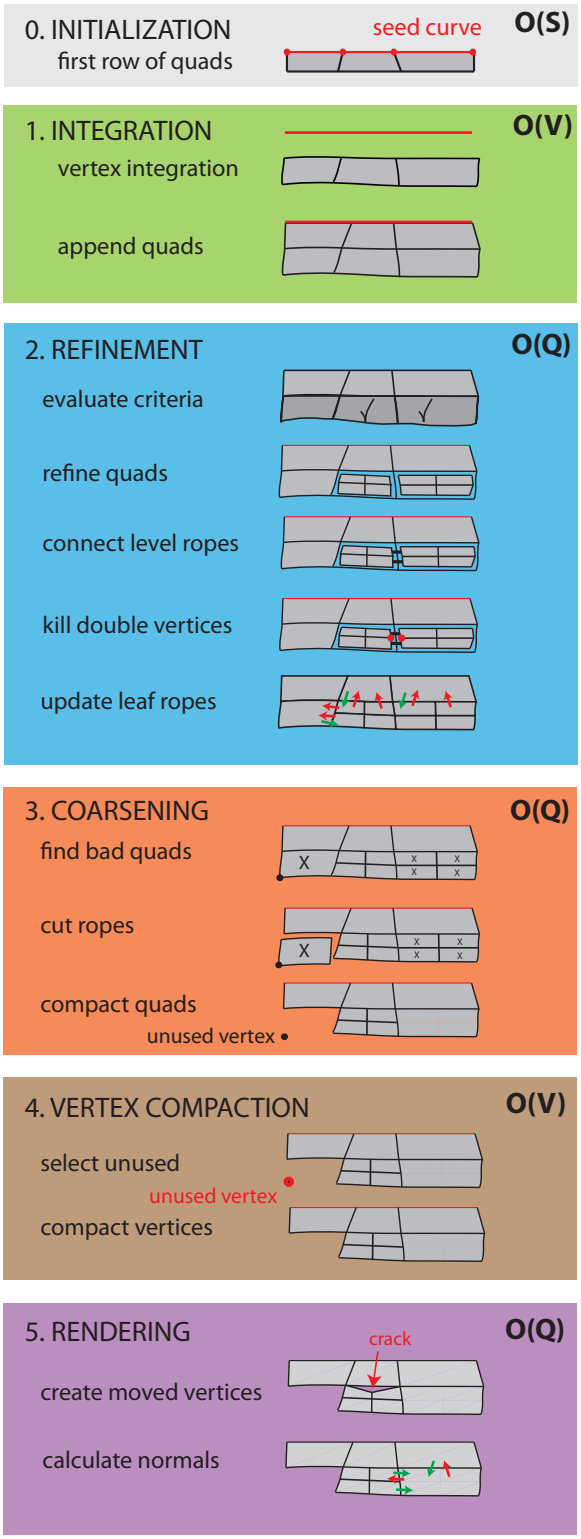

Figure 4: *Overview. For each step we note the computational complexity of the operation in Landau notation. Q is the number of quads, V the number of vertices, and S the number of seed points.*

2. Refinement creates new nodes by subdividing leaf nodes which have become too big or where the surface has high curvature. Where new nodes are created the leaf ropes and level ropes are updated.
3. Coarsening removes spurious quads or quads which have moved outside of the data domain. In this process all ropes linking to these quads are removed. Afterwards, the invalid quads are removed from memory by compaction. Spurious quads are small quads at places of low mesh curvature, where the notion of small depends on user-specified thresholds.
4. Vertex compaction removes unused vertices and updates the pointers into the vertices array at the leaf quads.
5. To prepare the streak-surface for rendering, cracks are removed from the mesh by creating a new set of vertices by projecting t-nodes onto the edge of the adjacent leaf node. To create a compelling rendering, normals can be computed based on the treelet representation on the GPU.

In the following we will describe all operations in more detail and explain why they all run in linear time. Since the number of vertices is strictly bounded by four times the number of quads, we could in principle describe the complexity in terms of the number of quads Q. However, during steps where only vertices are involved, we will state the complexity of operations in relation to the number of vertices V.

Integration During a single integration step all vertices are advected one time increment through the flow. Vertices which leave the data domain are flagged as invalid so that they can be removed at a later point in the algorithm. In the current framework, we use a fixed step size fourth order Runge-Kutta scheme (RK4) for advection. After the advection step, a new row of nodes is appended to the surface. Since we have to advect all vertices, we can conclude that the integration step is of linear complexity in the number of vertices.

Refinement For refinement, we compute criteria as suggested in the literature [BFTW09,KGJ09,MLZ10] based on: the maximal side length, the area of a node, and on an approximation of the local curvature. This can be done locally in constant time per leaf node, since we can use the leaf ropes from the previous step to find neighboring leaf quads.

In the next step, we create four new quads and five new vertices for each quad which is marked for refinement. Four vertices are created at the edge midpoints of the parent quad and one at the centroid by interpolation. See also Figure 2 for an illustration where vertices 13 to 17 result from a refinement step. In this step the ropes between the four new quads and their parent index are set. What cannot be set are the ropes to the adjacent quads, since the refinement operates in parallel and has to be executed independently for each quad. Since the refinement has to be done independently of the neighboring quads to avoid race conditions and to provide optimal performance it is possible that two neighboring

quads are refined concurrently. In this case, two vertices are created at the same position. After the local refinement is finished, we connect the newly created leaves. This is done by following the level ropes of the parents. With the level ropes connected in the previous step we can find these double vertices and set the vertex indices to the smaller one of the two.

One advantage of using treelets is that this refinement strategy can create individual samples on timelines which were previously not available. In comparison the approach of Burger et al. [BFTW09], requires memory for the full timeline for refinement. This is an important advantage, since local stretching at one point should not lead to non-local additional memory requirements. Since all refinement can be done locally using direct pointers it is of linear complexity in the number of quads Q.

Coarsening The coarsening step builds on the computation of the prefix sum for compaction of the vertex and quad arrays. Harris et al. [HSO07] present a fast way to compute a prefix sum in parallel using CUDA using an additional array. The prefix sum is an operation on an array in which each element in the result list is the sum of the elements in the input list up to its index. Given an array of values $[a_1, a_2, \dots a_n]$ the result is the array $[a_1, a_1 + a_2, \dots, a_1 + \dots + a_n]$. The prefix sum can be computed in parallel by adding values of increasing distance, distributing the additions equally to all processing units.

For coarsening, we evaluate coarsening criteria for all leaf quads. Quads which have become too small and quads outside the data domain are flagged for removal in an auxiliary array of integers. There are two ways for a quad to be actually removed from the array: either all its three siblings are also flagged for removal or it is already a root quad. The compaction of the arrays describing the quads is based on the prefix sum of the flag array. After compaction all pointers (children, parents, leaf ropes, and level ropes) are updated. Since all operations which are performed during coarsening run in linear time, this step is of linear complexity in the number of quads as a whole.

Vertex Compaction In this step we remove vertices which are no longer needed. There are two reasons for a vertex to qualify for removal: either it was created in parallel with another vertex at the same location during refinement, or the nodes it belonged to were deleted in the coarsening step. To find unused vertices, we iterate over all quads and deselect all vertices which are encountered. The vertices which remain flagged can be removed using the same strategy as described in the previous subsection. Since the vertices move, it is important to also update the pointers to the vertices in the leaf quads.

Rendering The rendering step consists of three substeps: we project copies of the vertices onto the edge of the neighboring quad, if the neighboring quad is on a higher level using a second vertex buffer. This can be decided locally based on the leaf ropes of the neighboring quads. The projection removes cracks in the surface which can appear at places where nodes of different level of refinement are located adjacently. This means that the rendered vertices are not necessarily at the same positions as those of the treelet mesh. In case this deviation becomes too large, the quad will be refined in the next iteration, so that only small corrections are required. Vertex normals are computed in parallel by averaging the normals of the surrounding leaf quads. During rendering, quads are tessellated to simplify shading. Both normal computation and hole filling run in linear time and can be performed in parallel per quad.

	ABC	SC	TC	SPH
vertices ($\times 10^6$)	2.4	1.9	1.0	1.8
nodes ($\times 10^6$)	2.6	2.5	1.0	2.3
max. refinement	17	10	15	12
memory (MB)	257	240	88	188

Table 1: *Memory consumption overview. (SC=square cylinder, TC=turbulent cylinder)*

5. Evaluation

We perform tests on a synthetic case, two data sets sampled on a Cartesian grid and SPH data. We select seeding positions which result in interesting surfaces and set the integration time-step size small enough to obtain stable results. Table 1 gives an overview of the memory consumption. All measurements were performed on a 2.8 GHz Core i7 CPU and a GeForce GTX 470 GPU.

ABC flow The ABC flow is an unstable solution of Euler's equation, displaying high-frequency instabilities under perturbation [Hal05]. This example is known to exhibit strong mixing of the fluid. Depending on the specified lifetime, the required level of refinement can reach very high levels. For the evaluation we set the lifetime of a vertex to 3 units. The seed curve is a straight line starting at $(0,0,0)$ and ending at $(1,2,\pi)$. Figure 5(a+b) shows the behavior of the algorithm for the ABC flow for these settings. The number of items in the node and vertex arrays is represented by the vertical axis on the left. The axis on the right shows the maximum number of refinements in the surface. Coarsening, refinement, and integration are all linear in the number of quads in the surface. It is interesting to note that the expensive part of the coarsening seems to be the removal of quads at the front of the streak-surface where the nodes become invalid when the vertices have reached the end of their lifetime. This step would be a candidate for further optimizations in the implementation, but the important point is that all steps are of linear complexity.

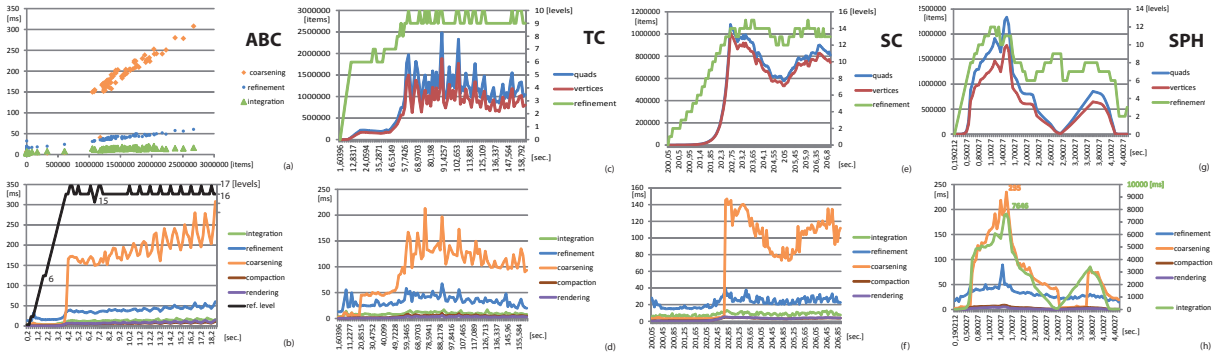

Figure 5: *Performance measurements. (a+b) ABC flow (c+d) Turbulent Cylinder (e+f) Square Cylinder (g+h) SPH dam break.*

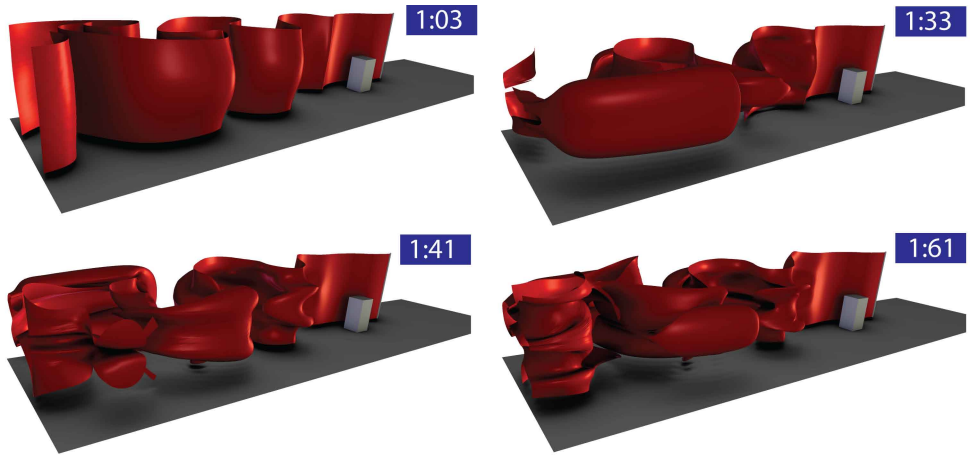

Figure 6: *Streak-surface in the square cylinder data set. The obstacle is shown at 1/3 of its actual height. (1:03) In the beginning the turbulence is low. (1:33-1:61) Over time turbulence increases, the surface starts bulging and stretching.*

Square cylinder The square cylinder data set is the result of a DNS simulation of the three-dimensional flow around a rectangular cuboid between parallel walls at Reynolds number 220 [CSBA05]. The data is sampled onto a uniform Cartesian grid with a resolution of $192 \times 64 \times 48$ and a temporal resolution of 102 steps. The seed curve is from $(-3, 0.5, 0.25)$ to $(-3, 0.5, 4.5)$. Figure 6 shows results for four time steps. The first image shows how the surface splits into two parts when it is partially advected outside of the domain. The obstacle is displayed at $1/3$ of its actual height to occlude the streak-surface less. In the second timestep we can see the bulging and folding effects on the streak-surface as turbulence increases. The third and fourth snapshot show the development of high-curvature regions and small features which require higher refinement levels. Figure 5 (c+d) presents measurements for the square cylinder data set. The square cylinder data set contains the tamest velocity field of all four examples and for the selected seed curve a refinement level of 10 is sufficient.

Turbulent Cylinder The turbulent cylinder data set contains a simulation of flow around a wall-mounted finite cylinder at Reynolds number 200. The turbulent cylinder simulation data is courtesy of Frederich [Fre10]. It captures the motion of large coherent structures behind the cylinder. Important features are: the separation of flow above the cylinder and large recirculating regions originating behind the cylinder. Figure 7 shows separation and mixing of the flow above the obstacle. The seed curve is from $(-0.75, -1, -1)$ to $(-0.75, 1, -1)$. We can see how the large coherent structures emanating from the obstacle pull the streak-surface down, creating twisted structures in the geometry. The spiralling motion in the flow requires very high levels of refinement depending on the lifetime of the particles which constitute the streak-surface. Once a part of the streak-surface is caught inside one of the vortices it can undergo twisting motion for prolonged periods of time. This is where the flexible refinement capabilities of the presented treelet representation is most important. In Figure 5(c+d) we show performance numbers for the turbulent cylinder case.

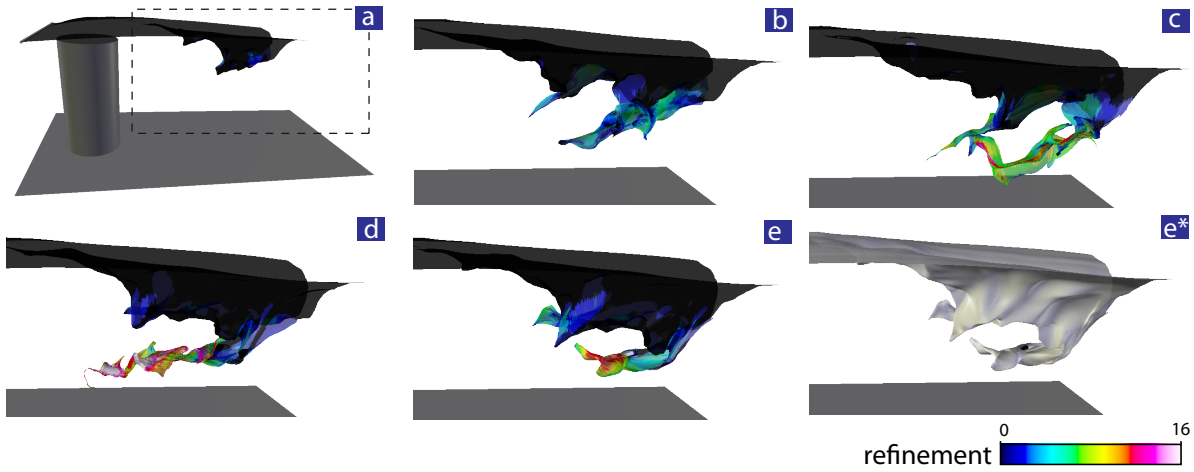

Figure 7: *Turbulent Cylinder. The seed is placed above the obstacle. (a) An early timestep for overview. Dashed line shows viewport of the remaining images. (b) The coherent structures developing behind the cylinder pull the streak-surface down in a twisting motion. (c-e) The spiralling motion of the coherent structures in the flow twists the streak-surface considerably. (e*) Illustration of the smooth shape of the twisted streak-surface.*

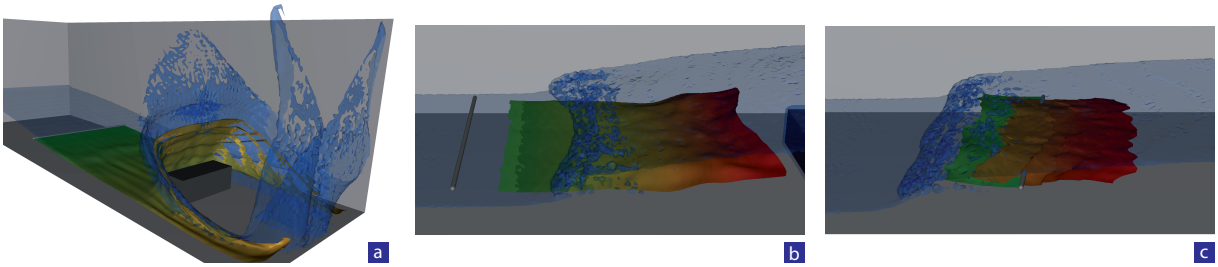

Figure 8: *SPH steak-surface. (a) When the water hits the obstacle, the particles diverge rapidly and both the water surface and the streak-surface develops holes. (b) As the water rushes back, the seed curve is no longer submerged and no new quads are generated. (c) Later the seed curve is inside the fluid again and a second surface patch is advected.*

Dam Break Data Set This data set is an SPH simulation of a breaking dam [KFV*05]. It has 87 time steps, each with 670,000 fluid particles using a *cubic spline* kernel. The advantage of SPH simulations is that they provide a relatively good approximation of the behavior of free surface flows, even though SPH does not provide guaranteed error bounds [Mon94]. This means that, even though this cannot happen in a perfectly correct simulation, it is possible that the free surface touches the streak-surface, creating holes or cutting the surface in multiple parts. Even though this is in contradiction with the Navier-Stokes equations, this is the behavior the data represents. The seed curve is from $(1.175, 0.1, 0.12)$ to $(1.175, 0.9, 0.12)$. Figure 8(a) shows that reconstruction can lead to very small fluid regions. In Figure 8(b) we can see a wave rolling back towards the seed curve. As the water level falls below the position of the seed curve (Figure 8(c)), there are no new particles released and the streak-surface disconnects from the seed curve. In Figure 8(d) we see how a second streak-surface patch is released from the seed curve as the wave front passes. Figure 9 illustrates the problem of noise reconstruction from SPH data. The quality of the reconstructed vector field decreases rapidly as the kernel sum goes to zero and at highly turbulent time steps it is possible to get very noisy results from the reconstruction. Figure 9(a) shows the appearance of very thin water volumes. In Figure 9(b) we can see a few slivery surface elements which result from very high velocity magnitudes in the reconstruction. Figure 5 (g+h) shows an evaluation of the performance for SPH data. The bulk of computation time is spent with vertex integration, since the reconstruction of velocity values is very expensive (Figure 5(h), green line, right axis). The first spike in surface complexity happens when the surface gains a lot of elements during collision of the water with the obstacle and the subsequent splashing motion. The second spike happens when the water hits the wall on the other side of the domain.

6. Conclusion

In this paper we focus on the efficient and flexible integration of streak-surfaces on the GPU. The surface representation is applicable to other types of problems, which require

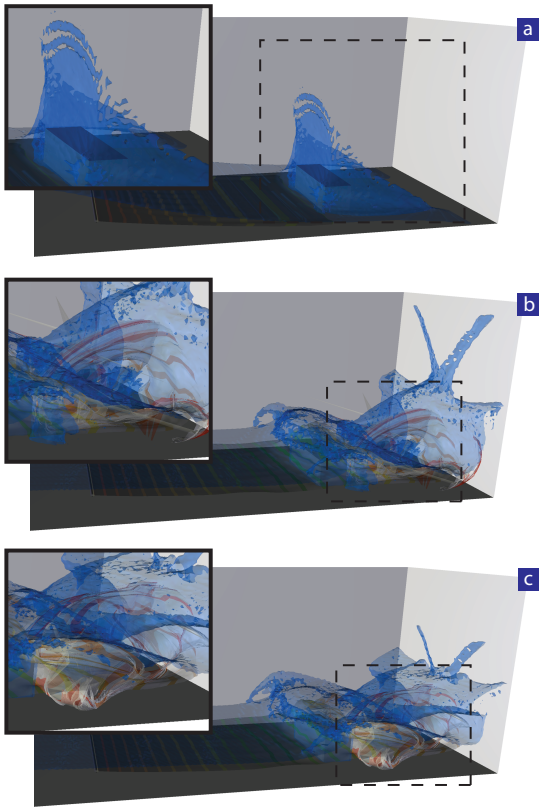

Figure 9: *Three timesteps of the breaking dam. The surface is twisted considerably and high refinement levels (17) are required.*

local adaptivity on the GPU as well, such as morphing or surface tracking. The additional connection within and between treelets allow the algorithm to perform all operations in parallel and locally. There are two main advantages of the presented surface representation: first, it allows for efficient refinement and coarsening in parallel. Second, it can represent surfaces which contain holes and break into multiple patches. As presented the technique does not support non-isotropic refinement and global remeshing operations. Future work in this direction would improve the treelet approach tremendously. Another open question is the active and efficient prevention of self-intersections of the streak-surface during integration. Another issue is data streaming from CPU to GPU memory during integration; using data sets where at least 3 timesteps fit into GPU memory we avoided this problem.

References

[BFTW09] BÜRGER K., FERSTL F., THEISEL H., WESTERMANN R.: Interactive streak surface visualization on the GPU. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1259–1266. 2, 4, 5

[CKSW08] CUNTZ N., KOLB A., STRZODKA R., WEISKOPF D.: Particle level set advection for the interactive visualization of unsteady 3D flow. *Computer Graphics Forum 27*, 3 (2008), 719–726. 1

[CSBA05] CAMARRI S., SALVETTI M., BUFFONI M., A.IOLLO: Simulation of the three dimensional flow around a square cylinder between parallel walls at moderate Reynolds numbers. In *Proceedings of the XVII Congresso di Meccanica Teorica ed Applicata* (2005). 6

[FBTW10] FERSTL F., BÜRGER K., THEISEL H., WESTERMANN R.: Interactive separating streak surfaces. *IEEE Transactions on Visualization and Computer Graphics 16*, 6 (2010), 1569–1577. 1

[Fre10] FREDERICH O.: *Numerische Simulation und Analyse turbulenter Strömungen am Beispiel der Umströmung eines Zylinderstumpfes mit Endscheibe*. PhD thesis, TU Berlin, 2010. 6

[Hal05] HALLER G.: An objective definition of a vortex. *Journal of Fluid Mechanics 525* (2005), 1–26. 5

[HSO07] HARRIS M., SENGUPTA S., OWENS J. D.: Parallel prefix sum (scan) with CUDA. In *GPU Gems 3*, Nguyen H., (Ed.). 2007, ch. 39, pp. 851–876. 5

[KFV*05] KLEEFSMAN K. M. T., FEKKEN G., VELDMAN A. E. P., IWANOWSKI B., BUCHER B.: A volume-of-fluid based simulation method for wave impact problems. *Journal of Computational Physics 206*, 1 (2005), 363–393. 7

[KGJ09] KRISHNAN H., GARTH C., JOY K.: Time and streak surfaces for flow visualization in large time-varying data sets. *IEEE Transactions on Visualization and Computer Graphics 15*, 6 (2009), 1267–1274. 1, 4

[MLP*10] MCLOUGHLIN T., LARAMEE R. S., PEIKERT R., POST F. H., CHEN M.: Over two decades of integration-based, geometric flow visualization. *Computer Graphics Forum 29*, 6 (2010), 1807–1829. 2

[MLZ10] MCLOUGHLIN T., LARAMEE R. S., ZHANG E.: Constructing streak surfaces for 3D unsteady vector fields. In *Proceedings of the Spring Conference on Computer Graphics 2010* (2010). 1, 4

[Mon94] MONAGHAN J. J.: Simulating free surface flows with SPH. *Journal of Computational Physics 110*, 2 (1994), 399–406. 7

[PPF*11] POBITZER A., PEIKERT R., FUCHS R., SCHINDLER B., KUHN A., THEISEL H., MATKOVIC K., HAUSER H.: The state of the art in topology-based visualization of unsteady flow. *Computer Graphics Forum (accepted for publication)* (2011). 2

[Sch08] SCHAFHITZEL T.: *Particle tracing methods for visualization and computer graphics*. PhD thesis, University Stuttgart, 2008. 1

[SW10] SADLO F., WEISKOPF D.: Time-dependent 2-D vector field topology: An approach inspired by lagrangian coherent structures. *Computer Graphics Forum 29*, 1 (2010), 88–100. 1

[vFWTS08] VON FUNCK W., WEINKAUF T., THEISEL H., SEIDEL H.-P.: Smoke surfaces: An interactive flow visualization technique inspired by real-world flow experiments. *IEEE Transactions on Visualization and Computer Graphics 14*, 6 (2008), 1396–1403. 2

[WT10] WEINKAUF T., THEISEL H.: Streak lines as tangent curves of a derived vector field. *IEEE Transactions on Visualization and Computer Graphics 16*, 6 (2010), 1225–1234. 2

[WTS*07] WIEBEL A., TRICOCHE X., SCHNEIDER D., JÄNICKE H., SCHEUERMANN G.: Generalized streak lines: Analysis and visualization of boundary induced vortices. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (2007), 1735–1742. 2