Commun. Comput. Phys.
doi: 10.4208/cicp.2014.m414

Vol. **17**, No. 5, pp. 1246-1270
May 2015

A Fast and Rigorously Parallel Surface Voxelization Technique for GPU-Accelerated CFD Simulations

C. F. Janßen[1,*], N. Koliha[1] and T. Rung[1]

[1] *Institute for Fluid Dynamics and Ship Theory, Hamburg University of Technology, Schwarzenbergstraße 95 C, 21073 Hamburg, Germany.*

Received 16 January 2014; Accepted (in revised version) 9 October 2014

Abstract. This paper presents a fast surface voxelization technique for the mapping of tessellated triangular surface meshes to uniform and structured grids that provide a basis for CFD simulations with the lattice Boltzmann method (LBM). The core algorithm is optimized for massively parallel execution on graphics processing units (GPUs) and is based on a unique dissection of the inner body shell. This unique definition necessitates a topology based neighbor search as a preprocessing step, but also enables parallel implementation. More specifically, normal vectors of adjacent triangular tessellations are used to construct half-angles that clearly separate the per-triangle regions. For each triangle, the grid nodes inside the axis-aligned bounding box (AABB) are tested for their distance to the triangle in question and for certain well-defined relative angles. The performance of the presented grid generation procedure is superior to the performance of the GPU-accelerated flow field computations per time step which allows efficient fluid-structure interaction simulations, without noticeable performance loss due to the dynamic grid update.

AMS subject classifications: 51P05, 65Z05, 68W10

Key words: Fast Grid Generation, Parallel Cartesian Mapping, Voxelization, GPU, Lattice Boltzmann, ELBE.

1 Introduction

CFD methods rely on the discretization of the continuous governing equations into discrete, finite approximations, on either Lagrangian grids, Eulerian grids, or in meshless formulations. Lattice Boltzmann methods, as addressed in the present publication, discretize the governing equations on an equidistant Eulerian grid. Grid points of a computational domain used for Lattice Boltzmann CFD implementations can be of a variety of

*Corresponding author. *Email addresses:* christian.janssen@tuhh.de (C. Janßen), nils.koliha@gmail.com (N. Koliha), thomas.rung@tuhh.de (T. Rung)

different kinds. Those points representing a fluid particle ensemble are considered *fluid particles*, whereas other points can represent a slip wall, a no-slip wall, an inflow or even moving velocity boundaries. Since the grid itself is static and does not change over time, it is - apart from proper dynamic boundary conditions at the body surface - the changing character of the points through which a moving body's motion is manifested. Fig. 1 shows fluid particles as black circles and solid body particles as red dots. Black circles overlaid with red dots signify solid body particles from previous time steps, where a lighter shade of red indicates solid body particles from iterations further back in time.

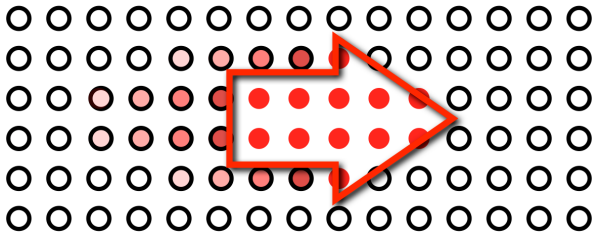

Figure 1: At each time step a moving body (red arrow) has to be mapped to the Cartesian grid. Hues of red indicate the changing grid node domain occupied by the body over time.

The grid generation algorithm that is presented in this paper is designed to be integrated seamlessly into a GPU-based CFD solver, the efficient Lattice Boltzmann environment ELBE [1]. The ELBE solver solves for three-dimensional turbulent free surface flows, including effects of viscosity and turbulent dissipation. Interactions of fluid and structure are considered as well. In transient CFD simulations with fluid-structure interactions, the solid body points have to be updated at each time step. The GPU-based implementation of ELBE allows for very competitive simulation times. Hence, the performance of the grid update algorithm and its implementation is crucial for successful simulations of challenging FSI problems. The main goal of this contribution is to develop a grid generation algorithm, that is

- Real-time capable — to rapidly voxelize the surface of a geometric object into a surface voxel representation in (or near) real time to be included in near-real time CFD simulations.
- Error-free and robust — to minimize the errors and the number of misidentified voxels, a source of potential instabilities in the numerical simulation.
- Efficient and extensible — to be convenient to integrate into a wide range of other CFD applications, e.g. SPH solvers.

After a brief literature review in Section 2, the following sections give further details of the necessary surface mesh preparations, details of the concrete algorithm and a detailed analysis of the performance of the algorithm, including numerical experiments.

2 Related work and key ideas of the proposed algorithm

The current paper describes a method to voxelize triangle meshes for Lattice Boltzmann CFD simulations on Eulerian grids. However, because voxelization is a very important topic for all sorts of different applications, multiple such methods have been proposed in recent years. Many of these even focus on very efficient GPU implementations, as the underlying technique (voxelization) is — by nature — related to rendering purposes. In order to motivate the need for yet another GPU-accelerated grid generation technique, we briefly review the current state-of-the-art in voxelization in the following.

2.1 An attempt of classification

Key characteristics of the proposed voxelization algorithms have to be identified, in order to systematically review the relevant literature. Apart from the purpose of voxelization, which can be but is not limited to rendering, collision detection, or grid generation for numerical solvers, two characteristics can be identified: (i) the voxelization type (surface voxelization vs. solid voxelization), and (ii) the utilized acceleration techniques (software vs. hardware acceleration). This work's main focus is on hardware accelerated surface voxelization.

2.1.1 Surface voxelization vs. solid voxelization

The most important feature of voxelization algorithms is the set of voxels that it creates. In case only the surface of objects is voxelized, *surface voxelization*, the algorithmic challenge is to identify as many voxels as possible to obtain a closed surface, but at the same time mark as little voxels as possible to safe computational costs and memory. The fundamentals of surface voxelization techniques have been discussed in e.g. Cohen-Or [2] and Huang [3]. The latter publication also includes a first basic algorithm for the voxelization of polygon meshes. Each mesh triangle is turned into a set of voxels: (i) inside certain well-defined spheres that surround the vertices, (ii) inside three cylinders that surround the edges, and (iii) between two planes defined by an offset h from the base triangle. The topological approach of Laine [4] uses geometric intersection targets that are associated with each voxel to identify those specific voxels that can be considered as constituting the interface between the inside and outside domains. By utilization of differently shaped intersection targets, the connectivity of the generated voxel surface can be changed and a number of traditional voxelization techniques can be reproduced. However, this approach is non-directional and does not allow for an identification of inner nodes. Opposite to surface voxelization, *solid voxelization* methods identify all the interior voxels of a solid body. Solid voxelization is not in the scope of this work, as we're interested in information on the boundary nodes only. For a detailed review on solid voxelization methods, the reader is referred to Liao [5].

2.1.2 Hardware acceleration

While the first publications on surface and solid voxelization were of theoretical nature, a lot of work was also conducted on a proper hardware acceleration of these techniques. Naturally, as many of the applications of such voxelization algorithms emerged from rendering and visualization tasks, graphics hardware was used very early to accelerate the voxelization process. E.g., Karabassi et al. [6] present a fast voxelization algorithm, which is based on the depth buffers (Z-buffer) of the graphics card. Their method can be accelerated by taking advantage of widely available, low-cost hardware. However, it can not reproduce inner cavities as they can not be seen from the depth buffers. Similarly, Fang and Chen [7] presented three solid/surface voxelization algorithms that have been accelerated with dedicated graphics hardware, the SGI Reality Engine. In 2006, Eisemann [8] presented a voxelization technique based on existing rendering steps in the graphics pipeline. The method employs a texture map, to store voxel information in an RGBA buffer and utilizes the accelerator's resources efficiently but is limited to graphics applications where the x,y dimensions are much larger then the z dimension. It also suffers from occasional wholes in the voxelized surface due to its roots in rendering technology. While being very effective, such hardware-accelerated algorithms are optimized for specific graphics cards and lack the desired extensibility to new hardware at the same level of relative performance. Modern GPU programmability allows the same level of efficiency for much more generally applicable approaches.

Schwarz and Seidel [9] present CUDA implementations of data-parallel algorithms for both surface and solid voxelization. They present a new triangle/box overlap test and claim that their method outperforms previous GPU-based approaches by up to one order of magnitude. More recently, Rauwendaal and Bailey [10] presented a voxelization method based on the standard graphics pipeline and implemented it in OpenGL. They claim that their method can be integrated very well in existing OpenGL applications and that it is both robust and efficient. Pantaleoni's [11] blending-based rasterization uses a two staged process of coarse and fine rasterization. Tiles are generated by a triangle/box overlap test and the containing triangles are sorted for faster processing in the fine raster step. The performance of this method is impressive and it represents a state of the art solution for surface voxelization.

2.2 Grid generation in the scope of Lattice Boltzmann solvers

Existing LBM grid generators include serial and parallel implementations of voxelization methods of various kind on both CPU and GPU architectures. However, the employed algorithms and models seem to be quite independent on the research that took place in the field of graphics processing and visualization, mostly due to historical reasons. One of the reasons might be that, up to now, most of the state-of-the-art Lattice-Boltzmann multiphysics solvers still are CPU-based, so that the published voxelization techniques (most of which were directly related to early GPU hardware) were not applicable. Nevertheless, the state-of-the-art in LBM grid generation will be briefly reviewed in the following.

The authors of [12] and [13] describe three distinct methods for the LBM grid generation around complex three-dimensional bodies and apply them for 3D fluid-structure interaction problems with an LBM-CPU solver on octree-type grids: (i) half-plane based algorithms, that are based on the fact that for convex geometries, all edges between arbitrary pairs of vertices are located inside the geometry. One of the main drawbacks of this method consequently is the limitation to convex polyeders [14]; (ii) the solid-angle-algorithm, that sums up the relative angles from the point under consideration to each polyeder vortex. The result indicates if the point is inside or outside the polyeder, see [15]. The method potentially can be extended to 3D [16], but is very inefficient and does not necessarily identify grid points that are exactly located on the polyeder edges; (iii) ray crossing algorithms, that emit a number of rays from the point under consideration and count the ray intersections with the polyeder surface [17]. Odd or even numbers of ray intersections with the surface indicate the inside or the outside, respectively, of a closed body. However, this method suffers from poor scalability for large numbers of triangles. Moreover, additional care has to be taken for the special case that the ray pierces a grid node and multiple triangles are intersected. In this case, the number of ray intersections is meaningless because it is not trivial to determine if the ray changed its inside/outside state at that point, it might have merely touched the surface. Szucki and Suchy's CPU approach on voxelization [18] uses a similar raycasting method, which creates a solid voxelization, essentially being based on the work of Thon [19]. Odd or even numbers of ray intersections with the surface indicate the inside or the outside, respectively, of a closed body. Since their method's performance decreases with a rising number of surface triangles and is of the order of seconds for just 10,000 triangles it is not suitable for real-time lattice updates. Inclan [20] pursued an approach based on a uniform and structured Cartesian grid produced by CartGen[†] from an STL (StereoLithography Interface Specification or Standard Tessellation Language) [21] file. The generated mesh blocks are filled with lattice nodes according to the applied boundary condition at the considered interface. All processing steps are executed subsequently and in serial on the CPU. The final output data is saved as a VTK (Visualization Toolkit) file and used by the LBM solver. All in all, the previously described techniques are all CPU-based and focus on solid voxelization.

2.3 Proposed algorithm: ePiP, an efficient point in polygon tester

The more recent hardware accelerated solutions reviewed in Section 2.1.2 can be used to generate Lattice Boltzmann grids efficiently. However, a tailor made solution was sought after by the authors of this work to meet the solver's requirements in terms of accuracy and extensibility. Moreover, for a better compatibility with ELBE and other current GPU-accelerated flow solvers, instead of rendering pipelines and/or OpenGL, the nVIDIA CUDA programming model is to be used for the implementation.

[†]http://cartgen.sourceforge.net/

Opposite to most of the previously mentioned voxelization techniques, the requirements for voxelizations that are used to create our Lattice Boltzmann grids are very special. Ideally, only the first layer of inner body nodes is identified, allowing for a proper fulfillment of flow field boundary conditions and indirect addressing techniques, so that only surface voxelization techniques can be applied. Instead of the flow field calculations, the boundary nodes execute specific operations to satisfy the solid-body boundary conditions. If the surface layer is not closed, the surface is not watertight. If the surface layer is too thick, more boundary operations are executed than needed. Moreover, the presented grid generation algorithm could also be run in an inverse mode, identifying the first layer of fluid nodes in the computational domain. With a reformulation of solid-body boundary conditions, this would allow to remove all inner nodes from both the simulation and also the data structure, for maximum savings. All this is only possible with the help of a surface voxelization algorithm. In case that a solid voxelization is required in any intermediate simulation step, this still can be realized by some state-of-the-art flood fill algorithms. In the most-complete hydrodynamic LBM model, the D3Q27 LBM model, every lattice node interacts with its 26 next-neighbors, on equidistant Cartesian grids. Consequently, the voxelizations that are employed to produce the computational grids have to be 26-separating, in order to obtain a watertight surface. Opposite to previously suggested surface voxelization techniques, in addition to the 26-separability, all the identified voxels need to be in the interior of the solid body. Consequently, our new LBM grid generation process can be categorized as a hybrid surface- and solid-voxelization technique, identifying the outmost 26-separated layer of a solid voxelization of the geometric object. In terms of voxelization performance, race conditions and conflicting memory accesses that occur in standard overlap tests have to be avoided. Schwarz and Seidel [9] also identified this problem and used atomic functions and memory buffers in the voxel update. The algorithm that is presented here avoids the problem by construction.

In the scope of this paper, a novel, very efficient and reliable test is presented, that is designed for execution on massively parallel processors such as GPUs. On this kind of hardware, identical instructions are executed on different chunks of data, which is often referred to as SIMD (single instruction, multiple data) concept. The main design choice in the development of a novel algorithm is how to split up the global computational task into small and independent instruction sets that are then processed in parallel by a number of threads. In the proposed ePiP grid generation algorithm, each computational thread is assigned with one triangle. For each triangle a local *region test* within the axis-aligned bounding box of the triangle yields the identification of body points. To uniquely define the region of valid *inner* points for the triangles, it is necessary to construct separation planes between adjacent triangles, i.e. for each edge of any given triangle. With this information it is possible to assign unique parts of the body domain to triangles. As an example, consider the 2D case depicted in Fig. 2. In the two-dimensional domain three-dimensional triangles and planes resemble edges or lines (3D bodies resemble 2D areas). The gray body consists of five corner points and edges. Assuming a symmetrical partition of the domain, at each of these corner points (corner points in 2D correspond to

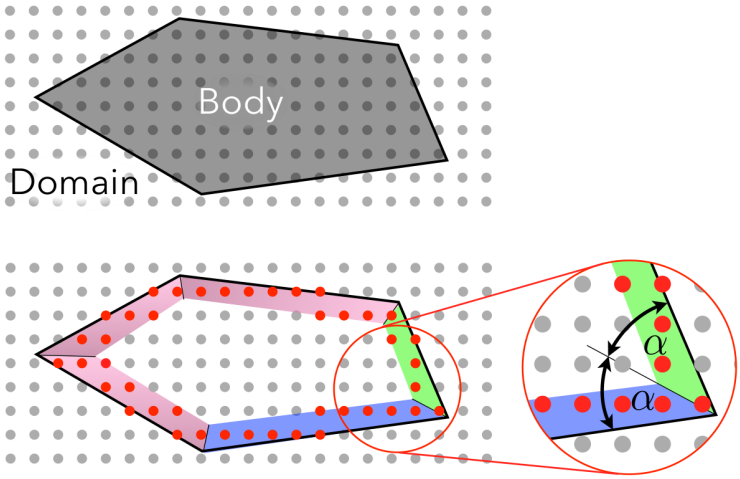

Figure 2: Two-dimensional example for the definition of half-angles and separation planes.

triangle edges in 3D) half-angles α can be defined in such a way that the angle between an edge and the separation plane equals α and the angle between two body edges (triangles) equals 2α. For efficiency reasons and to better apply boundary conditions to the body's points, it is desirable to identify the first inner layer of body points only. In Fig. 2 the image on the lower left shows this first-layer region in blue, green, and pink colors. The detailed magnification of the blue-green corner depicts the half-angle definition. Red dots represent the first layer of solid body grid nodes.

3 Surface mesh preparations

The proposed grid generation algorithm is based on tessellated surface mesh information. Before the actual grid generation algorithm is triggered, several preliminaries are required. In order to construct the separation planes that are needed later in the grid generation process, the provided triangular surface mesh is preprocessed and information on the triangle connectivity is extracted from the given vertex information.

3.1 Topology based neighbor search for tessellated surfaces

In order to generate half-angles for the triangle edges, it is necessary to find each triangle's neighbors. For any given closed and tessellated surface without bifurcations there exist three neighboring triangles to every triangle. In the example shown in Fig. 3 the blue triangle t_1 has three light blue neighbors $\mathsf{n}(i,\mathsf{t}_1)$ with $i=1,2,3$. The gray triangles $\mathsf{n}(j,\mathsf{n}(i,\mathsf{t}_1))$ are the respective neighbors thereof, where $j=1,2,3$, $i=1,2,3$, and $\mathsf{n}(3,\mathsf{n}(i,\mathsf{t}_1))=\mathsf{t}_1$. With the blue triangle being the considered triangle, the light blue ones are first degree neighbors and the gray triangles are second degree neighbors. The out-

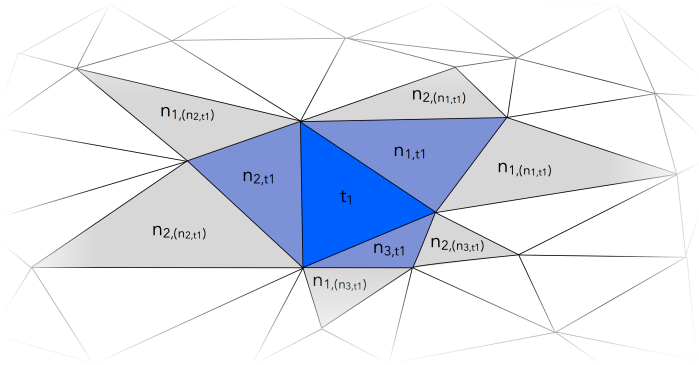

Figure 3: A triangle and its neighbors on a closed and tessellated surface.

lined surrounding triangles are neighbors of order three or higher. Fig. 3 represents all the information a neighbor search operation should produce.

Arbitrary surfaces can be described with the widely used STL file format, containing a list of triangle corner point locations. Repetitively, triangles are defined by stating their vertex coordinates as three-tuples. For closed surfaces, this representation redundantly stores vertices that are shared by multiple triangles. This not only introduces possible gaps between triangles resulting from round off errors but also requires more memory than actually necessary to store the well defined point positions and surface topology. From the information stored in the STL file, a list of all occurring points and their respective triangle IDs is generated. Sorting this list in a $x-y-z$ hierarchy allows the identification of point clusters. E.g. a cluster of five identical points indicates five triangles sharing this point. The point clusters are used to derive the triangle connectivity necessary for the definition of separation planes.

Consider the unit cube with its eight cornering points shown in Fig. 4. Each face of the cube is divided into two triangles, generating twelve triangles with a total of 36 non-unique points. After generating the list $\widetilde{\mathcal{L}}$ from the STL information and $\widetilde{\mathcal{L}}_{\text{points}}$ by sorting it in space, the eight unique points are identified from the point clusters. The `newID` is counting through the unique points, the `longID` is the point's original ID and the `shortID` stores the cluster's first point's original ID, i.e. its `longID`. Extracting the unique points of $\widetilde{\mathcal{L}}_{\text{points}}$ yields the list $\mathcal{L}$. The arrays A_p, and A_l are used for the neighbor determination algorithm. They reference points in $\mathcal{L}$ with their `newID`, can access point clusters through the `shortID` and derive the triangle ID from a point's `longID`. The neighbor determination process is executed by identifying all triangles that share a given unique point. This means that the determination of the neighbors of a given triangle only involves the triangles that are part of the point clusters that result from the vertices of the given triangle. A triangle is then determined as the given triangle's neighbor if they have two points in common. Note that the computational expense per triangle is independent of the overall number of triangles and scales solely with the number of triangles connected to the given

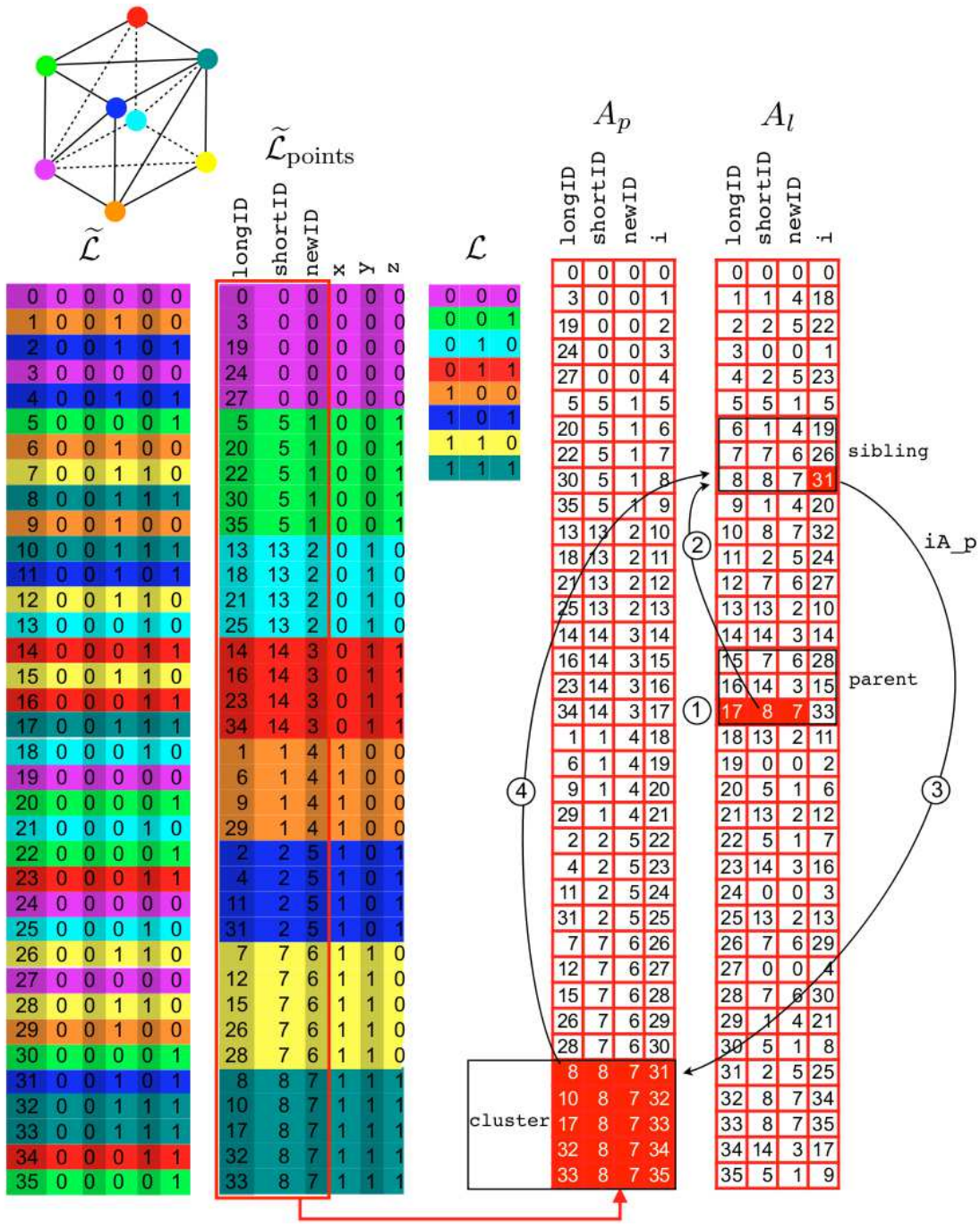

Figure 4: Essential data structure elements for the surface mesh preparation, exemplified by a unit cube. Sorting lists ($\widetilde{\mathcal{L}}$ & $\widetilde{\mathcal{L}}_{\text{points}}$) of vertices spatially allows the extraction of unique points ($\mathcal{L}$) and identification of point clusters, i.e. triangles that share the same point. The information is stored in arrays (A_p & A_l) for faster unordered access.

triangle's vertices. The process for a given triangle (`parent`) can be summarized as:

1. For each of the `parent`'s vertices, calculate the `longID` from the `parent`'s sequential ID $\{0\cdots n_{\text{Triangles}}\}$ and the local vertex number $\{0\cdots 2\}$.
2. Access A_l at the entry determined above (`longID`), acquiring the `shortID` (sequential ID of the first point in the cluster) of the vertex.
3. Enter A_l at the `shortID` to get the sequential ID `i` of the first point in the cluster.
4. Sequentially loop over all the points of the cluster and use their `longID` to derive the sequential ID $\{0\cdots n_{\text{Triangles}}\}$ of the triangle (`sibling`) they belong to.
5. Determine the number of points that the `sibling` and the `parent` share by comparing their vertices' respective `newID`s.
6. If a `sibling` shares two points with the `parent`, set it as a neighbor of the `parent`.

For the sake of an example, consider `iTriangle = 5` and `iVertex = 2` yielding `longID = 3*5+2 = 17` and thus acquiring `shortID = 8` and `newID = 7` from the A_l array (step 1 in Fig. 4). The `shortID = 8`th element in the A_l array (step 2) provides the `i = 31` value of the point cluster's first element (step 3). Its `longID` is `8` resulting in `jTriangle = 2` (step 4). This sibling has vertices with `longID = jTriangle*3 + {0, 1, 2} = {6, 7, 8}` and `newID = {4, 6, 7}`. The parent triangle has vertices with `longID = {15, 16, 17}` and `newID = {6, 3, 7}`. Parent and sibling triangle share the two unique points `6` and `7` (see identical `newID`s), which makes them neighbors. Since the shared points are along the parent's third edge the value `jTriangle = 2` is stored in the parents third neighbor entry. This example shows the determination of just one of the three neighbors.

These mesh preparations only have to be carried out once, before the actual grid generation takes place.

3.2 Mesh refinement

Since each computational thread operates on the assigned triangle's bounding box, the size of the triangle is directly affecting the execution time of each thread. Smaller triangles yield smaller bounding boxes resulting in a shorter execution time. *Size* in this context was chosen to be judged by the triangle's longest edge, since this attribute predominantly dictates the bounding box's dimensions. Moving bodies could rotate in such a way that the longest edge is aligned in the least preferable 45° angle with respect to the grid axes. It is therefore desirable to divide the tessellated surface into smaller, i.e. shorter longest edge, and consequently more triangles. To this end a mesh refinement procedure has been developed.

A triangle splitting approach has been chosen. Each triangle with an edge exceeding a set margin is cut in two, generating two new triangles. The blue triangle in Fig. 5 has an edge exceeding the margin and is cut in two. Since hanging nodes would result

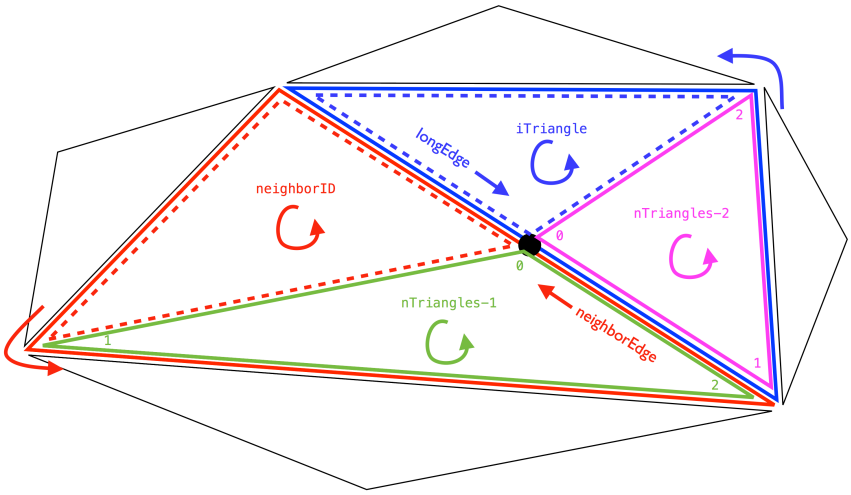

Figure 5: Mesh refinement applied to two neighboring triangles.

in undefined neighbor assignment, the (red) triangle that is sharing the blue triangle's longest edge is also split.

A new node is generated in the middle of the splitting edge and appended to the list of vertices. The blue and the red triangle are considered to have changed in size and remain at their position in the triangle list while the two new triangles are appended to it. Depending on the edge along which the triangles are halved, the neighbor IDs have to be updated accordingly. All eight triangles in Fig. 5 require neighbor updates because a single triangle was supposed to be split.

4 Details of the ePiP grid generation algorithm

After these initial preparations, the actual grid generation process is triggered. In the following section, the details of the proposed grid generation algorithm are discussed. A grid point is considered part of the inner body shell if

1. it is located on the inside of the triangle as defined by its unit normal vector $\tilde{\boldsymbol{n}}$ and its perpendicular distance s to the triangle plane is less than the first layer thickness δ, which is a function of the triangle orientation with respect to the grid, and

2. it is on the inside of all triangle separation planes to the (at most) three adjacent triangles defined by the bisection vectors $\boldsymbol{b}_i$.

4.1 Determination of first layer thickness δ and distance check

For each triangle a specific layer thickness has to be computed so that all the points that will be identified as *inner* points have at least one neighbor outside the triangle, i.e. each

inner point really is the *first* point on the inside. This layer thickness is derived as follows: Consider a cube of grid points (8 points in 3D, 4 points in 2D) and the triangle in question arranged in such a way that one of the cube's vertices is located on the triangle plane. The cosine of the angle φ in-between the triangle normal vector $\boldsymbol{n}=(n_x n_y n_z)^T$, $\|\boldsymbol{n}\|=1$ and the diagonal $\boldsymbol{z}=(\pm\Delta\,\pm\Delta\,\pm\Delta)^T$ of the cube that is oriented positively in the same quadrant as the normal is

$$\cos\varphi=\frac{<\boldsymbol{z},\boldsymbol{n}>}{\sqrt{3}\Delta}=\frac{\pm\Delta n_x\pm\Delta n_y\pm\Delta n_z}{\sqrt{3}\Delta}. \tag{4.1}$$

All sub sums of the inner product are positive since the diagonal is oriented towards the same quadrant as the normal vector. This yields

$$\cos\varphi=\frac{|n_x|+|n_y|+|n_z|}{\sqrt{3}}. \tag{4.2}$$

The layer thickness δ is the projection of the diagonal to the normal, thus:

$$\delta=<\boldsymbol{z},\boldsymbol{n}>=\sqrt{3}\Delta\cos\varphi=\Delta\big(|n_x|+|n_y|+|n_z|\big). \tag{4.3}$$

In Fig. 6, the two-dimensional case is illustrated. In analogy to the three-dimensional, more general case, an equivalent 2D layer thickness of $\delta_{2\mathrm{D}}=\Delta\big(|n_x|+|n_y|\big)$ is found.

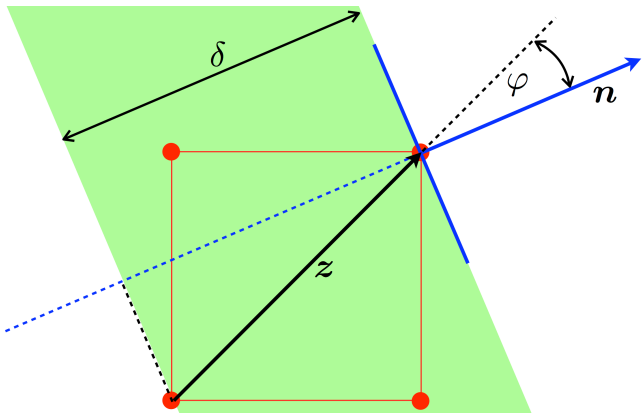

Figure 6: Determination of the layer thickness in 2D.

The perpendicular distance of a grid point to the triangle plane now is computed via the negative inner product of a vector originating at one of the triangle's vertices and pointing at the grid point in question and the unit normal vector. The negative inner product yields the norm of the projection of the point vector onto the inverted normal and therefore the desired distance:

$$s=-<\boldsymbol{v}_4,\boldsymbol{n}>=-<\boldsymbol{v}_5,\boldsymbol{n}>=-<\boldsymbol{v}_6,\boldsymbol{n}>, \tag{4.4}$$

where v_{3+i} with $i=1,2,3$ are defined as vectors originating from $\boldsymbol{P}_i$ and pointing at the grid point in question, $\boldsymbol{P}_4$:

$$\boldsymbol{v}_{3+i}=\boldsymbol{P}_4-\boldsymbol{P}_i\,. \tag{4.5}$$

The perpendicular distance s has to be computed for every point within a triangle's bounding box. Points that satisfy the condition

$$s\in\left[0:\delta\right) \tag{4.6}$$

are identified and further investigated for their validity.

4.2 Construction of separation planes

Consider a tessellated closed body topology where each triangle t has the vertices $\boldsymbol{P}_1$, $\boldsymbol{P}_2$, and $\boldsymbol{P}_3$. These points are arranged in such a way that the edges $\boldsymbol{v}_1=\boldsymbol{P}_2-\boldsymbol{P}_1$, $\boldsymbol{v}_2=\boldsymbol{P}_3-\boldsymbol{P}_2$, $\boldsymbol{v}_3=\boldsymbol{P}_1-\boldsymbol{P}_3$ define an outward pointing normal vector $\tilde{\boldsymbol{n}}=\boldsymbol{v}_1\times\boldsymbol{v}_2$. A separation plane is constructed for each triangle edge that is shared with an adjacent neighbor. The unit normal vectors of the focus triangle and its considered neighbor are $\boldsymbol{n}$ and $\boldsymbol{n}_{N,i}$, respectively, where $i=1,2,3$ indicates the edge along which the neighbor with unit normal $\boldsymbol{n}_{N,i}$ occurs. The most straightforward approach would be to construct a symmetrical separation plane $\mathcal{S}_{h,i}$ from the average of the adjacent triangles' unit normals. Such a plane is defined by the shared triangle edge $\boldsymbol{v}_i$ and the half-vector

$$\boldsymbol{h}_i=-\boldsymbol{n}-\boldsymbol{n}_{N,i}\,. \tag{4.7}$$

This approach resembles the half-angle definition for α introduced above, see the left panel in Fig. 7 for the 2D equivalent. Due to the varying layer thickness of the neighboring triangles, this approach is not able to cover the entire inner domain. A triangular prism at the innermost corner of the triangle domains is not tested for either of the neighbors. This can be avoided by introducing the bisection plane $\mathcal{S}_{b,i}$ constructed from the shared triangle edge $\boldsymbol{v}_i$ and a vector $\boldsymbol{b}_i$ that is proportional to a vector originating from the edge and pointing towards the intersection line of the two layer thickness planes while being perpendicular to $\boldsymbol{v}_i$. The layer thickness adjusted bisection approach is depicted in the right panel of Fig. 7. The bisecting vector $\boldsymbol{b}_i$ is defined as

$$\boldsymbol{b}_i\propto\widetilde{\boldsymbol{b}}_i=\frac{\delta_{N,i}}{\sin\mu}\,\boldsymbol{n}^{\perp}+\frac{\delta}{\sin\mu}\,\boldsymbol{n}_{N,i}^{\perp}, \tag{4.8}$$

where δ and $\delta_{N,i}$ are the layer thicknesses of the focus and the neighbor triangle, respectively, and $\boldsymbol{n}^{\perp}=\boldsymbol{n}\times\boldsymbol{v}_i$ and $\boldsymbol{n}_{N,i}^{\perp}=\boldsymbol{v}_i\times\boldsymbol{n}_{N,i}$ are vectors perpendicular to the triangles' unit normals and their shared edge (see Fig. 8).

The computation of the sine function can be avoided by multiplying equation (4.8) with $\sin\mu$ since only the signed direction of $\boldsymbol{b}_i$ is of interest for the construction of $\mathcal{S}_{b,i}$.

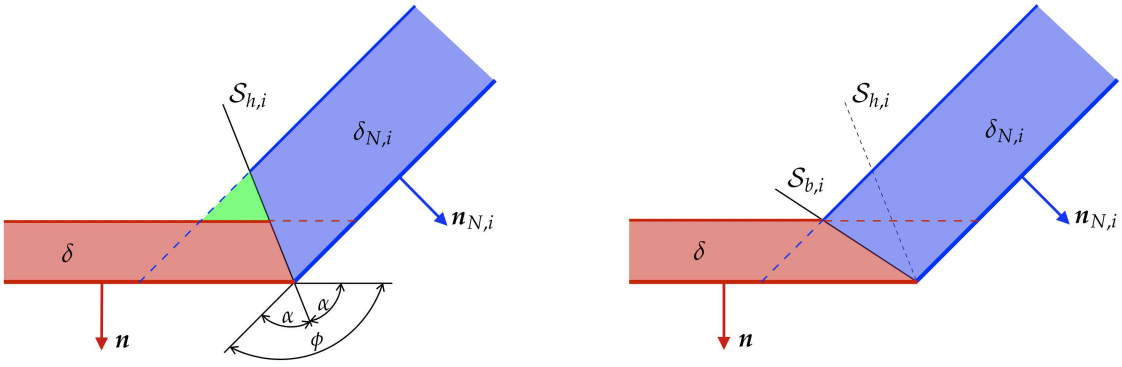

Figure 7: Definition of an unsymmetrical separation plane $\mathcal{S}_{b,i}$ (right). This allows the consideration of grid points lying outside (green region) the triangle domains defined by the half-separation plane $\mathcal{S}_{h,i}$ (left).

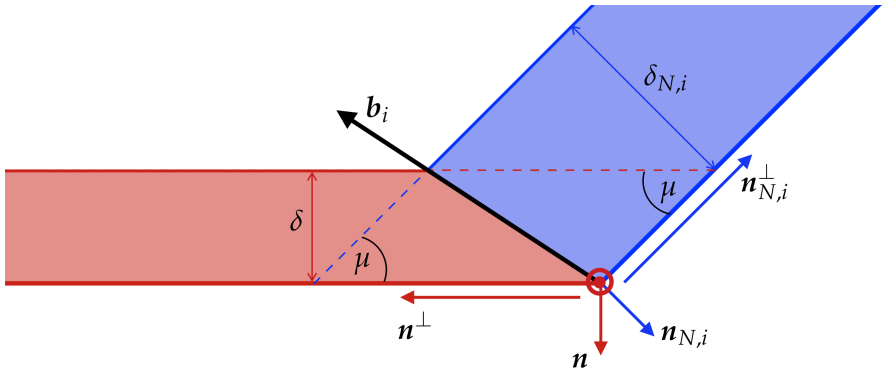

Figure 8: Construction of the unsymmetrical bisection vector $\boldsymbol{b}_i$ from the normal vectors and the shared triangle edge.

For total opening angles ϕ of more than 180°, i.e. concave surface regions, the bisecting vector has to be inverted. The inner product of the edge vector $\boldsymbol{v}_i$ and the vector resulting from the normals' cross product can be used to define the sign of $\boldsymbol{b}_i$:

$$\boldsymbol{b}_i = < \boldsymbol{n} \times \boldsymbol{n}_{N,i}\,,\, \boldsymbol{v}_i > \cdot \left[\delta_{N,i} (\boldsymbol{n} \times \boldsymbol{v}_i) + \delta (\boldsymbol{v}_i \times \boldsymbol{n}_{N,i}) \right]. \tag{4.9}$$

The bisecting vector's norm is vanishing for $\boldsymbol{n} = \boldsymbol{n}_{N,i}$. In such cases, one possibility is to fall back to the half-vector $\boldsymbol{h}_i$ which actually is the exact solution for this scenario. The separation planes have to be constructed for every edge whenever the surface's orientation relative to the grid changes. For transient CFD simulations of moving bodies this is the case at every time step.

The grid point $\boldsymbol{P}_3$ is set as a boundary node if

$$< \boldsymbol{b}_i \times \boldsymbol{v}_{3+i}\,,\, \boldsymbol{v}_i > \geq 0 \quad \forall\ i \in \{1,2,3\} \tag{4.10}$$

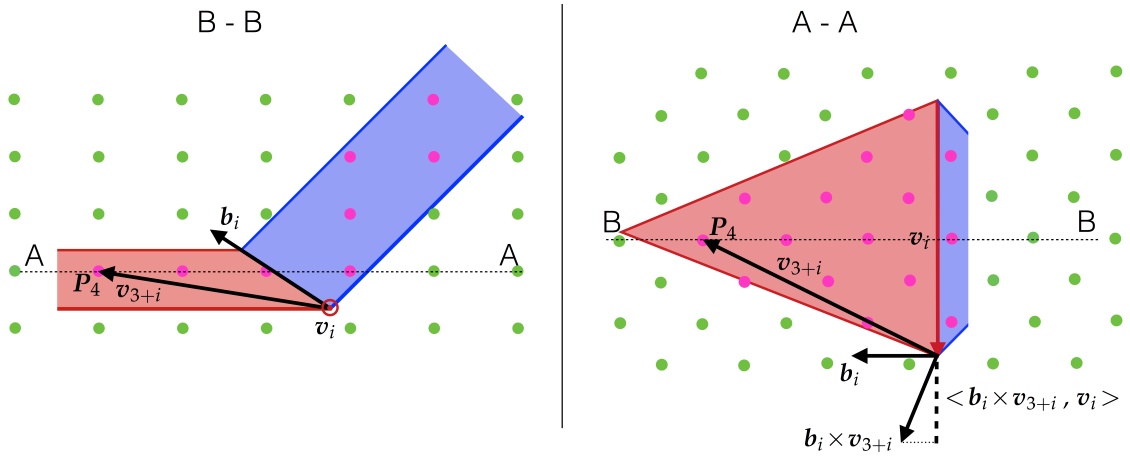

Figure 9: Point classification scheme.

holds true. Fig. 9 depicts the geometrical relations found for the focus triangle and its neighbor: a point is considered part of the domain if it is within each of the triangle's separation planes. Each plane is defined by the bisection vector $\boldsymbol{b}_i$. The point is considered inside of separation plane $\mathcal{S}_{b,i}$ if the cross product's projection of $\boldsymbol{b}_i$ and $\boldsymbol{v}_{3+i}$ onto the edge vector $\boldsymbol{v}_i$ is positive.

For the case that two neighboring triangles almost coincide, i.e. ϕ is very small, all the involved vectors $\boldsymbol{b}_i$, $\boldsymbol{v}_{3+i}$, and $\boldsymbol{v}_i$ will still be numerically identical (except for the sign in case of $\boldsymbol{v}_i$) for the two triangles. Thus, if one of the two triangles yields a positive value for $<\boldsymbol{b}_i \times \boldsymbol{v}_{3+i}, \boldsymbol{v}_i>$ the other triangle will get a negative result, even for very small values of $\boldsymbol{b}_i \times \boldsymbol{v}_{3+i}$.

5 GPU implementation and performance

The main aim of the proposed voxelization algorithm is to find a surface voxelization corresponding to a tessellated mesh and a rectilinear grid of points in a highly parallelized manner with very few operations necessary. The presented surface voxelization algorithm has been implemented in C++ and CUDA for parallel execution on NVIDIA GPUs. Both single and double precision computations are supported. The CUDA philosophy is based on a combined architecture of a host and one or multiple devices. The host part consists of a CPU and the host memory, whereas the devices are graphics cards with their GPUs and the device memory. Intercommunication between host and device is provided by the PCI-express (PCIe) interface. Just like regular code for single thread processing on the host is compiled with a C compiler, the files with CUDA extensions have to be compiled with the NVIDIA CUDA C compiler (nvcc). A typical CUDA program provides data for processing on the GPU by copying it from host to device memory. The GPU then executes the instructions provided by the host and stores the result data

in device memory. The presented implementation performs the data copying only once at the beginning of the simulation and relies entirely on the fast-access device memory for following voxelization intervals. The ePiP algorithm is seamlessly embedded into the efficient lattice Boltzmann solver ELBE, a CUDA-accelerated CFD solver based on an efficient Lattice Boltzmann method, that operates on device memory only and hence allows for a very efficient coupling to the grid generator.

5.1 Cost function approach

Before the actual performance measurements on recent GPGPU hardware, some theoretical remarks on the expected performance of ePiP are given. Consider the per triangle computational cost function

$$\mathcal{C}(n_B) = k + \lambda n_B \tag{5.1}$$

being a linear function of the number of grid points n_B found in that triangle's bounding box. The constant per triangle cost k and per grid point cost λ are parameters of the cost function. The dimensions of the bounding box depend on the triangle's axes projected and normalized dimensions l_x, l_y, and l_z measured in grid units:

$$n_B(l_x,l_y,l_z) = (l_x+\Delta B)\,(l_y+\Delta B)\,(l_z+\Delta B)\,. \tag{5.2}$$

The bounding box overlap ΔB is the number of additional points in each direction, introduced to guarantee that a closed inner shell of points can be identified. Therefore, for constant factors k and λ, the cost function can be expressed as a function of the triangle dimensions:

$$\mathcal{C}(l_x,l_y,l_z) = k + \lambda\,(l_x+\Delta B)\,(l_y+\Delta B)\,(l_z+\Delta B)\,. \tag{5.3}$$

Mesh refinement causes a change in cost according to

$$\mathcal{C}(l_x,l_y,l_z) \to 2\,\mathcal{C}(f_x l_x, f_y l_y, f_z l_z)\,, \tag{5.4}$$

where f_x, f_y, and f_z range from 0.5 to 1 depending on the triangle's shape and orientation. I.e. a refinement step leads to performance gains if the cost of two triangles with smaller dimensions is less than the cost of the single triangle with larger dimensions. Assuming that the number of triangles exceeds the number of cores on the GPU[‡], a performance maximum is observed if

$$\mathcal{C}(l_x,l_y,l_z) = 2\,\mathcal{C}(f_x l_x, f_y l_y, f_z l_z)\,. \tag{5.5}$$

For an isosceles triangle with diminishing area (shape) and $l_x = l_y = l_z$ (orientation), i.e. a three dimensional diagonal, the dimension factors f_x, f_y, and f_z approach the minimum of 0.5. In this case the performance maximum condition ((5.5) with (5.3)) reads

$$\frac{3}{4}\,{l_{x,y,z}}^3 + 6\,{l_{x,y,z}}^2 - 64 - \frac{k}{\lambda} = 0\,, \tag{5.6}$$

[‡]If the number of triangles is significantly less than the number of cores, i.e. after a refinement step there still is at least one core per triangle, a mesh refinement step will always have a positive effect on performance.

where the bounding box overlap has been set to ΔB=4. The optimal projected normalized length is $l_{x,y,z}^{\text{optimal}} \approx 3.3$ for $\frac{k}{\lambda} = 30$. This value for $\frac{k}{\lambda}$ has been estimated by comparing the number of algorithmic operations involved per triangle and per grid point. Thus, the optimal normalized triangle edge size is

$$L_{\text{diagonal}}^{\text{optimal}} = \sqrt{3} \cdot l_{x,y,z}^{\text{optimal}} \approx 6\,. \tag{5.7}$$

For other triangle shapes and orientations the dimension factors f_x, f_y, and f_z are more than 0.5 and, therefore, the optimal triangle edge size is also more than it is for the most beneficial case of a three dimensional diagonal:

$$L^{\text{optimal}} \geq 6. \tag{5.8}$$

5.2 Performance measurements

Following these theoretical remarks, performance measurements are carried out. To give performance predictions for the grid mapping process of arbitrarily shaped, complex triangular surface meshes, the following predictor function T is used:

$$T(n_\Delta, n_{bb}, n_{active}) = n_\Delta T_\Delta + n_{bb} T_{bb} + n_{active} T_{active} + \frac{n_{active}}{n_\Delta} T_0\,. \tag{5.9}$$

The total computational costs for the grid mapping of one specific triangular surface mesh is assumed to be a function of the total number of triangles n_Δ, the number of grid nodes in the triangles' bounding boxes, n_{bb}, and the number of active nodes, n_{active}, that have been finally marked as solid body grid nodes. For the calibration, several parameter studies for a simple spherical geometry (Fig. 10) were conducted, on consecutively refined grids. The performance of the grid generator was analyzed in terms of MNAPS (million node activations per second),

$$P_{ePiP} = \frac{n_{active}}{T}\,. \tag{5.10}$$

The grid mapping was repeated 100 times in a row, to get a reliable estimate for the computational time T. All the computations were done on an NVIDIA GTX Titan GPGPU board, that was connected to a multi-core host machine via an external PCI-Express expansion chassis. Single precision arithmetics and CUDA 5.5 [22] were used.

In Fig. 11, the resulting performance is plotted, against the number of active nodes per triangle. As expected, and predicted by the cost function model in the previous section, the performance maximum is achieved for $L^{\text{optimal}} \geq 6$, corresponding to a number of active nodes per triangle of approximately 36. Secondly, an increasing performance with increasing total number of triangles can be observed. Apart from parallelization issues, this reflects the fact that, with increasing number of triangles, the overhead of inactive nodes in the triangle bounding boxes decreases (which is accounted for by the T_0-term in

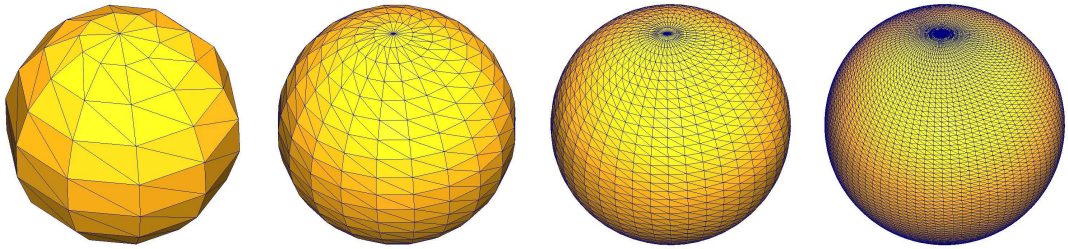

Figure 10: Illustration of the spherical test geometry that was used for the calibration of the ePip performance prediction model. Exemplarily, four selected refinement levels with 160, 720, 3040, and 12480 nodes are displayed.

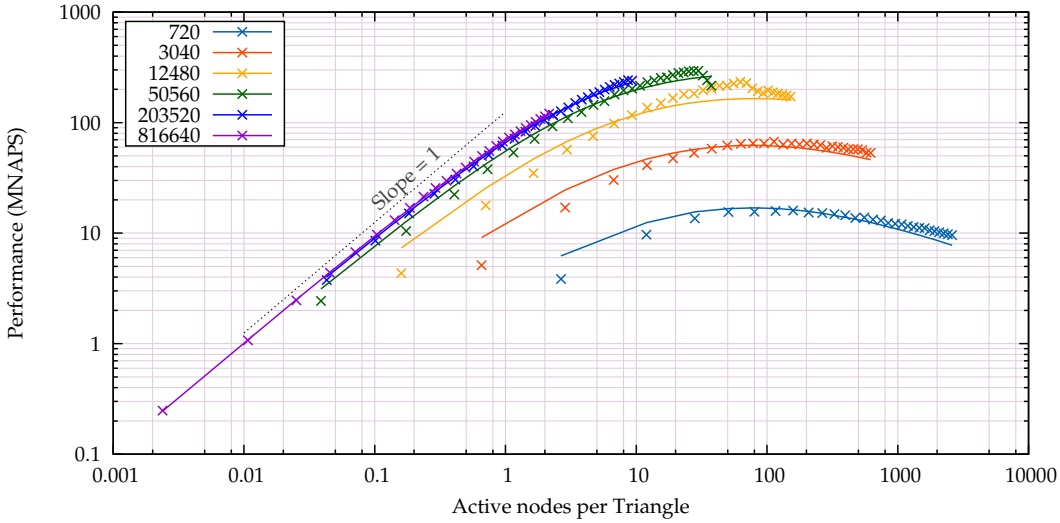

Figure 11: 3D ePiP sphere test case — predicted (-) and measured (x) performance plotted against the number of active nodes per triangle for six different configurations from 720 to 816,640 triangles.

(5.9)). The test case with the highest number of triangles ($n_\Delta = 816{,}640$) has been used to calibrate the unknown factors T_i in the performance prediction model as follows:

$$T_\Delta = 5.85\text{E-}9\text{s}, \quad T_{active} = 1.38\text{E-}9\text{s}, \quad T_{bb} = 5.64\text{E-}11\text{s}, \quad T_0 = 1.6\text{E-}6\text{s}. \tag{5.11}$$

The resulting model then has been used to predict the runtimes of the remaining test cases (with 720 to 203,520 triangles), and a very good agreement was found, see Fig. 11 (symbols vs. straight lines). For all the test cases, the performance of the grid generation tool was between 10 and 100 million node activations per second.

Fig. 12 shows the inverse relation, i.e. the computational time for the grid mapping over the number of triangles per activated node. Even in the worst case of low numbers of triangles per active node, the total computational costs of the grid generation is below 0.1 seconds.

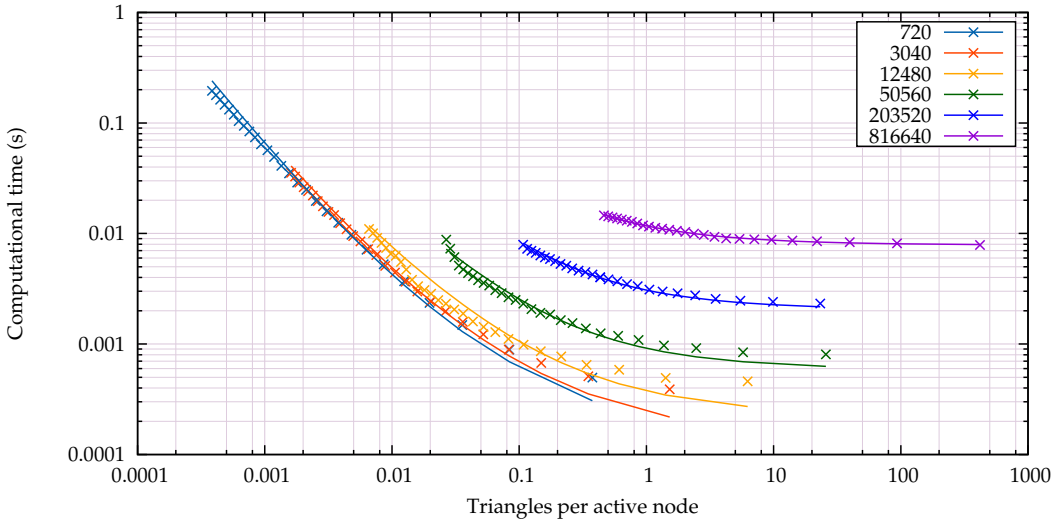

Figure 12: 3D ePiP sphere test case — predicted (-) and measured (x) computational time for the grid update, plotted against the number of triangles per active nodes for six different configurations from 720 to 816,640 triangles.

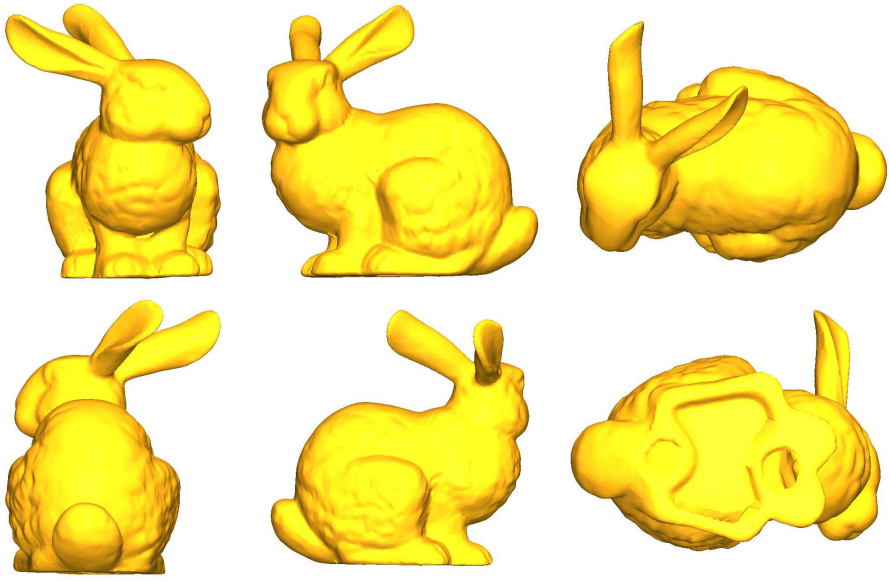

Figure 13: Illustration of the Stanford Bunny test geometry [23, 24].

5.3 Performance prediction for the Stanford bunny

The performance prediction model can be used to estimate the computational time for the grid generation process in a real-world problem with a more complex configuration. As

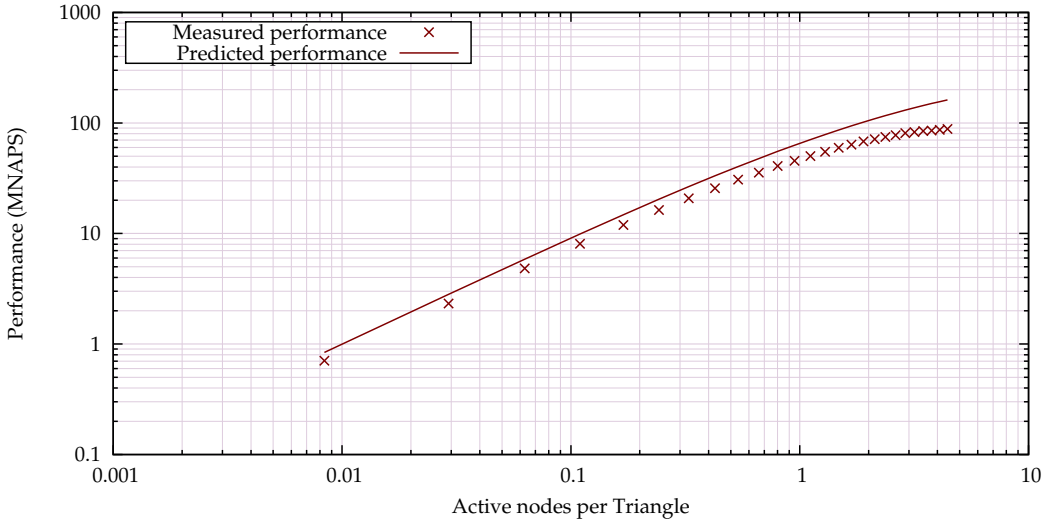

Figure 14: Comparison of predicted (-) and measured (x) performance for the Stanford Bunny test case.

benchmark geometry, the well-established Stanford Bunny is examined [23]. The Stanford Bunny has been used extensively in various research projects to test geometric algorithms. The digital bunny model was created by a range scan technique [24]. While the original bunny model has 69,451 triangles only, we use an enriched version with 270,000 triangles. The test geometry has a bounding box of 50 times 38.7 times 49.5 length units (LU) cubed and a surface area of 5842 LU squared. The bunny geometry is mapped to a Cartesian, equidistant grid with (at most) 10 lattice nodes per unit length, yielding a computational grid of 95.8 million nodes, and a discrete bunny surface of approximately 584,200 lattice nodes, corresponding to $\frac{n_{active}}{n_\Delta} \approx 2.163$. With the help of Fig. 11, Eq. (5.9) and the calibration factors in Eq. (5.11), a performance maximum of 100 MNAPS is predicted. In Fig. 14, the performance prediction and the actual measured performance are compared for different grid sizes. Indeed, the resulting performance maximum is found to be approximately 80 MNAPS, so that the prediction is rather accurate and the extrapolation of calibration data from convex and ideal spherical geometries to a lot more complex, concave geometries is valid.

5.4 Coupling to CFD solvers

In CFD applications, not only the absolute grid generation time is of interest, but the ratio of computational time for the actual CFD computations and the geometry updates. For the above-mentioned bunny grid, the computational costs for one Lattice-Boltzmann time step can easily be estimated and/or found in literature. In our experience, the maximum performance of three-dimensional GPGPU-accelerated LBM solvers is in the order of 250 MNUPS (million grid node updates per second) for complex multiphysics simu-

lations, e.g. free surface flow simulations [25], and up to 500 MNUPS for singlephase flow problems [26] including complex geometries, boundary conditions and grid updates. Our performance estimate is based on our experiences with the ELBE code, for complex singlephase flows. However, for academic benchmark problems (e.g. lid-driven cavity, simple boundary conditions, domain sizes a power of two) with simple collision operators, several publications report a performance of up to 1.4 GNUPS on a GTX Titan board. This can be reproduced by the ELBE code for very academic settings. The performance decreases as soon as more complex collision operators, such as an MRT-LES, and more complex boundary conditions, such as LIBB or anti-bounce-back pressure boundaries, are used. Moreover, we found that, as soon as the grid sizes are not a power of two, performance decreases remarkably (e.g. comparing a 256 cubed grid with a 300 cubed grid). If we then further consider fluid-structure interaction problems, where the presented grid generator comes into play and a dynamic update of the Eulerian grid is necessary, additional force calculation, local boundary conditions and a refill algorithm for the grid nodes that change from solid to fluid are mandatory in the flow field update. Nevertheless, even for a 1,400 MNUPS fluid solver, the performance of ePiP still is more than sufficient: the theoretical performance of the coupled solver, including the LBM flow field update and the subsequent ePiP grid update, yields

$$P_{total} = \frac{n_{total}}{\frac{n_{total}}{1{,}400\ \text{MNUPS}} + \frac{n_{active}}{80\ \text{MNAPS}}} = \frac{95.8E6}{\frac{95.8E6}{1{,}400\ \text{MNUPS}} + \frac{5.8E5}{80\ \text{MNAPS}}} \approx 1{,}265\ \text{MNUPS}. \tag{5.12}$$

Hence, the overhead of the grid update step decelerates the numerical simulation by only 9.6%. Re-calculating the overhead based on the above-mentioned more realistic ELBE performance values even reduces the overhead to 3.6%, which is deemed to be negligible.

5.5 Applications

After the successful validations, coupled algorithm can be applied to state-of-the-art problems in computational fluid dynamics, including the numerical simulation of fluid-structure interactions and flows in complex geometries. Application examples include thermal cabin flows (Fig. 15), multiphase flows (Fig. 16) and aircraft ditching simulations (Fig. 17). Further application examples with moving geometries and more complex fluid-structure interactions are currently under development and will be published in the near future.

6 Conclusions and outlook

In this paper, a novel, fast and efficient parallel surface voxelization technique was presented. The necessary preprocessing steps, key concepts of the parallel implementation and performance results were addressed. The computational surplus of the presented

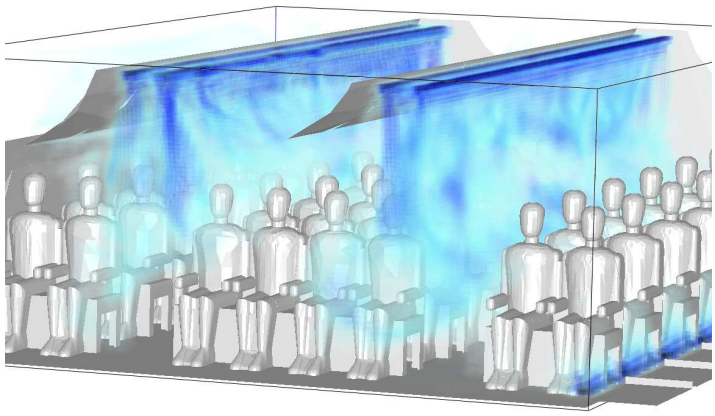

Figure 15: Snapshot of a three-dimensional ELBE singlephase flow simulation: developing flow field (velocity magnitude) in an A380 cabin mock-up [27].

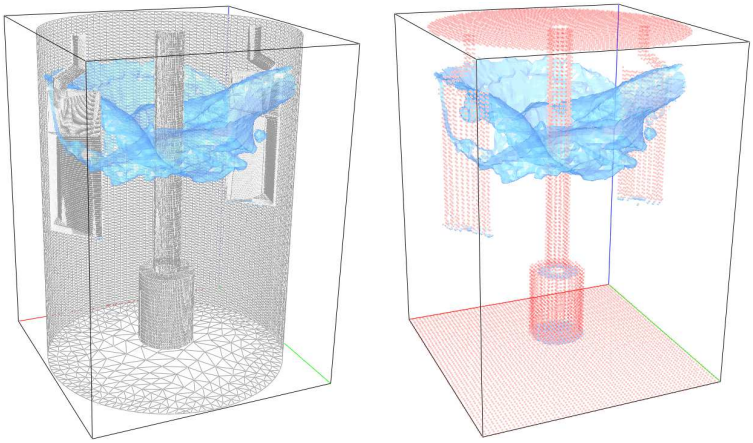

Figure 16: Snapshot of a three-dimensional ELBE multiphase flow simulation: phase interface location in a complex 3D stirring unit (left: surface mesh representation of the unit, right: corresponding surface voxelization).

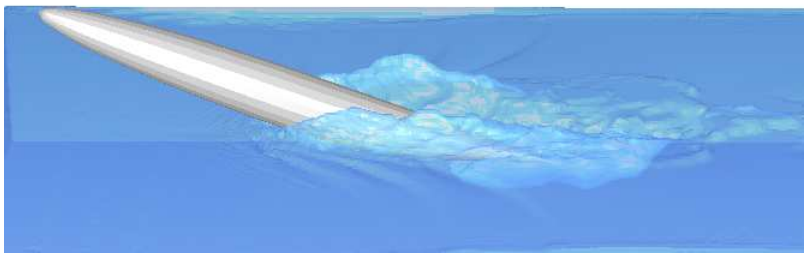

Figure 17: Snapshot of a three-dimensional ELBE freesurface flow simulation: ditching of an aircraft fuselage.

grid generation procedure was shown to be negligible in comparison with the performance of the flow field calculations. The coupled algorithm allows for very efficient fluid-structure interaction simulations, without noticeable performance loss due to the dynamic grid update. More specifically, the (theoretical) performance loss is found to be less than 10%, even for highly-accelerated GPU-LBM solvers.

Due to these very promising numerical results and the very competitive performance, the present methodology will be further examined and improved in future work. To enhance the numerical accuracy of the boundary representation, second-order boundary conditions will be addressed. Representing an arbitrary surface with a voxelized Cartesian grid introduces discretization errors that scale with the grid spacing and results in numerical simulations with first order accuracy. Second-order accuracy can be obtained by improved boundary conditions based on an additional consideration of the surface distances for the identified body nodes and their respective lattice directions. In the context of LBM, these distances are often referred to as subgrid distances [28]. By design, the extension of the presented methodology to calculate such distances is straightforward.

The left panel of Fig. 18 depicts a circle that is discretized with 360 line elements. In the right panel, the red nodes indicate the nodes inside the axis-aligned bounding box of the red line element, that are tested with the ePiP algorithm. The thin radials mark the unique element domains. The bright green node is identified as a solid body node by the thread that has been assigned to the red line element. The eight discrete lattice directions (thick, light gray) are intersected with the line segment's extended line (thin, red) to compute the node distance (thick, turquoise). A reliable implementation for the computation of the lattice distances in 3D is considerably more computationally expensive and an efficient technique for their determination has to be found in order to realize higher order boundary conditions for smoother body representation on Cartesian grids for the lattice Boltzmann method.

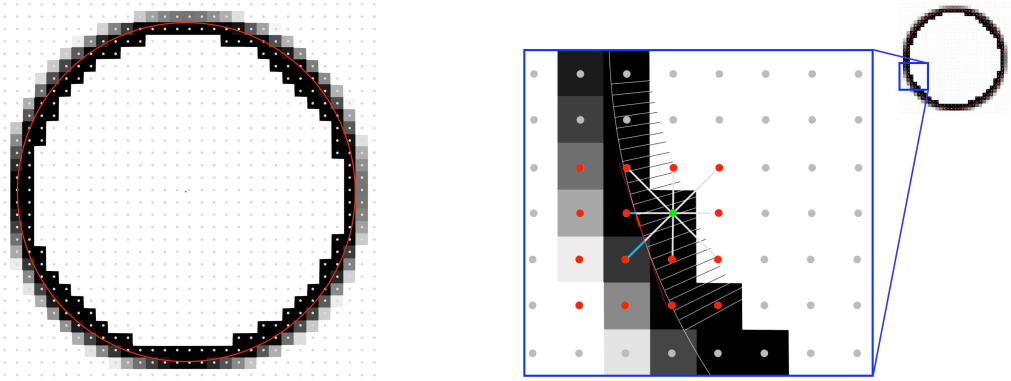

Figure 18: (Left) A circle (red), which is discretized with 360 line elements, is mapped to the Cartesian grid. Cubes colored in shades of gray indicate their proximity to the circle's surface (black = internal or close by; white = distant). (Right) A detailed depiction of a line segment (red line) with its bounding box (red dots), and the computed lattice distances (turquoise).

Acknowledgments

The authors would like to thank Willem Gropengießer for numerous fruitful discussions on the ePiP grid generation algorithm.

References

[1] C.F. Janßen et al. The efficient lattice boltzmann environment **elbe**. `http://www.tuhh.de/elbe`.
[2] D. Cohen-Or and A. Kaufman. Fundamentals of surface voxelization. *Graph. Models Image Process.*, 57(6):453–461, November 1995.
[3] J. Huang, R. Yagel, V. Filippov, and Y. Kurzion. An accurate method for voxelizing polygon meshes. In *Proceedings of the 1998 IEEE Symposium on Volume Visualization*, VVS '98, pages 119–126, New York, NY, USA, 1998. ACM.
[4] S. Laine. A topological approach to voxelization. In *Eurographics Symposium on Rendering*, volume 32, 2013.
[5] D. Liao. *Real-time Solid Voxelization Using Multi-core Pipelining*. PhD thesis, Washington, DC, USA, 2009. AAI3344878.
[6] E.-A. Karabassi, G. Papaioannou, and T. Theoharis. A fast depth-buffer-based voxelization algorithm. *J. Graph. Tools*, 4(4):5–10, December 1999.
[7] S. Fang, S. Fang, H. Chen, and H. Chen. Hardware accelerated voxelization. *Computers and Graphics*, 24:200–0, 2000.
[8] E. Eisemann and X. Décoret. Fast scene voxelization and applications. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, pages 71–78, New York, NY, USA, 2006. ACM.
[9] M. Schwarz and H.-P. Seidel. Fast parallel surface and solid voxelization on gpus. *ACM Trans. Graph.*, 29(6):179:1–179:10, December 2010.
[10] R. Rauwendaal and M. Bailey. Hybrid computational voxelization using the graphics pipeline. *Journal of Computer Graphics Techniques (JCGT)*, 2(1):15–37, March 2013.
[11] J. Pantaleoni. Voxelpipe: A programmable pipeline for 3d voxelization. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 99–106, New York, NY, USA, 2011. ACM.
[12] S. Freudiger. *Entwicklung eines parallelen, adaptiven, komponentenbasierten Strömungskerns für hierarchische Gitter auf Basis des Lattice Boltzmann Verfahrens*. PhD thesis, Technische Universität Braunschweig, 2009.
[13] S. Geller. *Ein explizites Modell für die Fluid-Struktur-Interaktion basierend auf LBM und p-FEM*. PhD thesis, TU Carolo-Wilhelmina zu Braunschweig, 2010.
[14] N. Blum. *Algorithmen und Datenstrukturen: Eine anwendungsorientierte Einführung*. Oldenbourg Wissenschaftsverlag, 2004.
[15] E Haines. Point in polygon strategies. In Paul Heckbert, editor, *Graphics Gems IV*, pages 24–46. Academic Press, 1994.
[16] P. C. P. Carvalho and P. R. Cavalcanti. Point in polyhedron testing using spherical polygons. In Alan Paeth, editor, *Graphics Gems V*, pages 42–49. Academic Press, 1995.
[17] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1998.
[18] M. Szucki and J. Suchy. A voxelization based mesh generation algorithm for numerical models used in foundry engineering. *Metallurgy and Foundry Engineering (MaFE)*, 38(1):43–

54, 2012.
[19] G. Gesquiere, S. Thon and R. Raffin. A low cost antialiased space filled voxelization of polygonal objects. In *GraphiCon '04 Proceedings*, pages 71–78. GraphiCon, September 2004.
[20] E. Inclan. Development of pre-processing software for lattice boltzmann fluid dynamics solver. Fiu-arc-2012-800000394-04c-064, U.S. Department of Energy, 2012.
[21] Inc. StereoLithography Interface Specification, 3D Systems, October 1989.
[22] nVIDIA. nVIDIA CUDA. `http://www.nvidia.com/object/cuda_home_new.html`.
[23] G. Turk. The Stanford Bunny. `http://www.gvu.gatech.edu/people/faculty/greg.turk/bunny/bunny.html`.
[24] G. Turk and M. Levoy. Zippered polygon meshes from range images. In *Proc. SIGGRAPH '94 (Orlando, Florida, July 24-29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 311–318, 1994.
[25] C. Janßen and M. Krafczyk. Free surface flow simulations on GPGPUs using LBM. *Computers and Mathematics with Applications*, 61(12):3549–3563, June 2011.
[26] C. Obrecht, F. Kuznik, B. Tourancheau, and J.-J. Roux. A new approach to the lattice boltzmann method for graphics processing units. *Comput. Math. Appl.*, 61(12):3628–3638, June 2011.
[27] M. Kühn, J. Bosbach, and C. Wagner. Experimental parametric study of forced and mixed convection in a passenger aircraft cabin mock-up. *Building and Environment*, 44(5):961–970, May 2009.
[28] M. Bouzidi, M. Firdaouss, and P. Lallemand. Momentum transfer of a lattice-Boltzmann fluid with boundaries. *Phys. Fluids*, 13:3452–3459, 2001.