Journal of Computational and Applied Mathematics 270 (2014) 166–177

Contents lists available at ScienceDirect

Journal of Computational and Applied Mathematics

journal homepage: www.elsevier.com/locate/cam

The meshing framework ViennaMesh for finite element applications

Florian Rudolf*, Josef Weinbub, Karl Rupp, Siegfried Selberherr

Institute for Microelectronics, Technische Universität Wien, Gußhausstraße 27-29/E360, A-1040 Wien, Austria

A R T I C L E   I N F O

Article history:
Received 30 September 2013
Received in revised form 20 January 2014

Keywords:
ViennaMesh
Meshing
Finite element method
Uniform programming interface

A B S T R A C T

The applicability of the meshing framework ViennaMesh for finite element simulations is investigated. Meshing tools are highly diverse, meaning that each software package offers specific properties, such as the conforming Delaunay property. The feasibility of these properties tends to be domain specific, thus restricting the general application of a meshing tool. For research purposes, it is desirable to have a rich toolset consisting of the various meshing packages in order to be able to quickly apply the various packages to the problem at hand. Different meshing tools have to be utilized to support a broader range of mesh properties. Further contributing to this problem is the lack of a common programming interface, impeding convenient switching of meshing backends. ViennaMesh tackles this challenge by providing a uniform meshing interface and reusable mesh-related tools, like CGAL, Gmsh, Netgen, and Tetgen. We depict the feasibility of our approach by discussing two applications relevant to finite element simulations, being a local mesh optimization and an adaptive mesh refinement application.

1. Introduction

The finite element method (FEM) has proven to be efficient for solving boundary value problems and thus has become very popular in various fields of scientific computing. The FEM requires the simulation domain, commonly referred to as Ω, to be discretized by a mesh, being a discrete topological and geometrical data structure [1].

Several commercial and free open source mesh generation tools are available, utilizing different implementations of meshing algorithms [2–5]. Each meshing approach generates meshes with different properties, for instance, three-dimensional (3D) unstructured meshes which satisfy the conforming Delaunay property. However, certain mesh properties are desirable when solving a boundary value problem using the FEM. Elements of *good* shape directly affect the numerical robustness of the meshing algorithm as well as the solution quality of a numerical simulation utilizing the mesh [6]. However, what is considered a *good* shape of a mesh element depends on the characteristics of the boundary value problem. For instance, when solving the drift-diffusion model for a field-effect transistor, the FEM benefits from anisotropic elements near the channel region, while elements in the inactive regions of the device should be isotropic [7].

This highly diverse set of properties puts pressure on mesh generation tools, as a single meshing implementation struggles to serve as the sole meshing backend for simulation applications. Applications – investigating several different simulation problems – need to either stick to one meshing software or combine multiple meshing packages. While the first

* Corresponding author. Tel.: +43 15880136054.
E-mail addresses: rudolf@iue.tuwien.ac.at (F. Rudolf), weinbub@iue.tuwien.ac.at (J. Weinbub), rupp@iue.tuwien.ac.at (K. Rupp), selberherr@iue.tuwien.ac.at (S. Selberherr).

http://dx.doi.org/10.1016/j.cam.2014.02.005
0377-0427/© 2014 Elsevier B.V. All rights reserved.

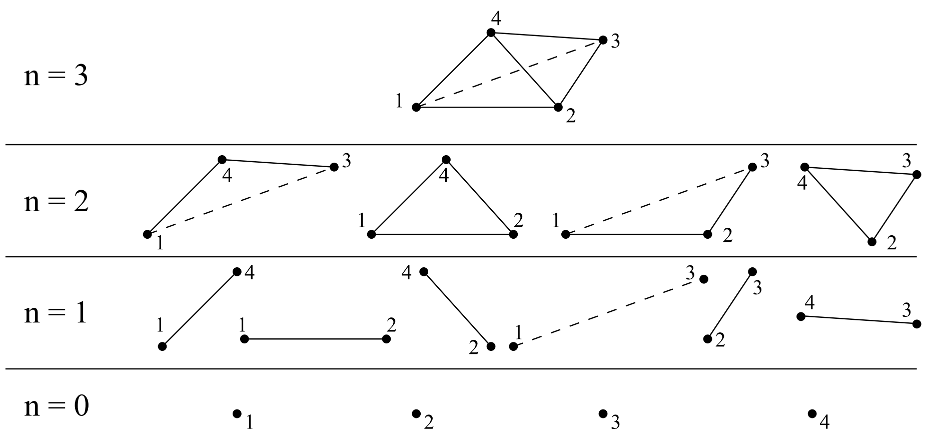


Fig. 1. A 3-simplex, i.e., a tetrahedron, is split up into its boundary elements. *Dimension* $n = 3$: The 3-simplex has a topological dimension of $n = 3$. *Dimension* $n = 2$: The boundary elements with the topological dimension $n = 2$ are 2-simplices, i.e., triangles. *Dimension* $n = 1$: The boundary elements with the topological dimension $n = 1$ are edges. *Dimension* $n = 0$: The boundary elements with the topological dimension $n = 1$ are vertices. In total, there are four boundary 2-simplices, six boundary edges, and four boundary vertices.

might lead to poor numerical robustness or high discretization errors [8], the latter introduces considerable development overhead, due to the lack of a common programming interface.

ViennaMesh [9] addresses these problems by defining a uniform interface on top of a flexible topological mesh data structure, enabling to add mesh-related tools in a modular manner. Based on this flexible interface, external meshing tools, such as the mesh generation packages Netgen [4] or Tetgen [5], can be used in a unified manner. Additionally, a set of auxiliary mesh-related algorithms – such as quality metrics – is provided, reducing development efforts required to implement advanced mesh-based applications.

In this work we depict the feasibility of ViennaMesh for finite element simulations, by discussing two relevant applications in detail. The first application covers local mesh optimization, similar to an approach to drastically improve the quality of a tetrahedral mesh [10]. The element with the worst quality is identified, a local cavity surrounding this element is created, and then re-meshed. The second application shows an adaptive mesh refinement application, solving a drift-diffusion model using a mixed finite element–finite volume approach [11].

This work is organized as follows. Section 2 provides a concise overview of meshes and mesh generation. Section 3 presents relevant related work. Section 4 introduces our meshing framework ViennaMesh. Section 5 discusses two mesh applications, relevant for applications based on the FEM.

2. Meshing background

Due to the finite resources of a computer, the transition from real world objects to the computer domain requires a discretization. Therefore, any object is, in a first step, characterized by a so-called geometry, defining the object's shape and size. The geometry of an object can be represented by different approaches, for instance, by a boundary representation or a constructive solid geometry (CSG).

In a second step, a discretization process consists of partitioning the geometry into a finite number of elements according to certain properties, such as no two elements may overlap. The thus generated partitioned simulation object is referred to as a mesh. A mesh is a collection of elements [8], for instance simplicial elements. A k-simplex is a geometrical object, which is defined as the non-degenerated convex hull of points $p_i \in \mathbb{R}^n$, $i = 1, \ldots, k+1$, for instance, a 2-simplex and a 3-simplex relate to a triangle and a tetrahedron, respectively. Fig. 1 shows an example of a 3-simplex and its boundary elements. Beside cells, which are elements with the maximum topological dimension within a mesh, a mesh typically also includes boundary elements, like faces, edges, and vertices.

There are three main types of mesh generation algorithms: advancing front [12], quad- and octree [13], and incremental Delaunay [14]. The advancing front algorithms create mesh elements starting from the boundary and advance inwards. This approach tends to create elements with *good* quality at the domain boundary, while elements in the interior are likely to be of inferior quality. Advancing front algorithms have been particularly successful for applications requiring anisotropic mesh elements, such as for many fluid dynamics problems [8]. Quad- and octree algorithms use a background mesh which spans the entire domain and adapt it to satisfy the domain boundary. This approach produces elements with good quality in the interior, whereas elements on the boundary are usually of inferior quality. Incremental Delaunay algorithms start by creating an initial mesh and then incrementally insert vertices inside the mesh or on the boundary to create elements according to the Delaunay property.

Table 1
An overview of the discussed meshing software. Topological dimension refers to the topological dimension of generated mesh elements. ● indicates that the software has the corresponding property. For the API evaluation, + refers to a good API (is easy to use), ∼ refers to a moderate API (minor difficulties in usage), and − refers to a poor API (hard to use or with incomplete documentation).

	License	Geometrical dimension	Topological dimension	Structured	Unstructured	Multiple segments	Incremental Delaunay	Advancing front	API
CGAL	GPL	2, 3	2, 3		●	●	●		+
Gmsh	GPL	2, 3	2, 3	●	●	●	●	●	−
Netgen	LGPL	2, 3	2, 3		●	●[a]		●	−
Tetgen	X11	3	3		●	●	●		∼

[a] Although support for multiple segments is not inherently provided, it can be implemented manually.

Like quad- and octree algorithms, meshes created by incremental Delaunay algorithms are likely to have elements of good shape in the interior, while elements on the boundary are of inferior quality. Unlike advancing front and quad- and octree algorithms, some incremental Delaunay algorithms have mathematical guarantees of mesh element quality.

Aside from generating a mesh, an already available mesh has often to be adapted according to specific criteria. This process is referred to as mesh adaptation. A mesh may be adapted in different ways, for instance, via refinement or coarsening methods, where the first locally increases the resolution of a mesh by introducing additional mesh elements, the latter performs the inverse operation. A typical use case of mesh adaptation is to increase the quality of a mesh, driven by mesh classification mechanisms. In essence, mesh classification uses quality metrics, like the radius-edge ratio used by tetrahedral elements [6], to evaluate the quality of a mesh element and, by extension, the whole mesh. Such a step is usually required to improve a subsequent numerical simulation, further reducing the numerical errors and thus increasing the solution quality.

Certain areas of application require the mesh to be partitioned, according to, for instance, material regions. For the remainder of this work, we refer to the partitions as segments. In the presence of segments, the interfaces between the individual segments are treated as constraints and thus have to be preserved during the meshing process.

3. Related work

This section discusses a selection of popular mesh generation tools with a focus on free open source packages. The tool discussion is followed by an analysis of the investigated research field in order to serve as motivation for this work. We focus on a selection of free open source tools due to their natural affinity with research work enabled by accessible source code. Table 1 gives an overview of the introduced meshing software and their individual properties.

The *Computational Geometry Algorithms Library (CGAL)* [2] is a C++ general geometric algorithms library distributed under several licenses, depending on the particular CGAL package. The fundamental parts of CGAL are distributed under the GNU lesser general public license (LGPL), while the higher level algorithms, such as 3D mesh generation, are released under the GNU general public license (GPL). Among the rich set of provided geometric algorithms, CGAL offers incremental Delaunay meshing algorithms for two- and three-dimensional geometries. A peculiarity of CGAL is the generic software design which significantly increases the flexibility [15]. For instance, CGAL enables to utilize arbitrary C++ function objects for defining user-specific mesh element criteria. CGAL also supports arbitrary input geometry types using its abstract oracle mechanism.

Gmsh [3] is a meshing framework, providing a couple of meshing algorithms such as structured and unstructured mesh generation of two- and three-dimensional meshes. Gmsh is distributed under the GNU GPL. For 3D unstructured mesh generation, the tool uses Tetgen combined with an additional mesh optimization step. Gmsh also has an interface to Netgen, allowing the use of Netgen's advancing front algorithm. Unlike other meshing applications, Gmsh also includes a scripting language for controlling the meshing process. Although Gmsh provides an application programming interface (API), it is not exhaustively documented, thus impeding the utilization by external applications.

Netgen [16] is a tetrahedral advancing front mesh generation tool. The software is available under the GNU LGPL and coded in C++. Netgen also provides an application programming interface API called *NGLib*, which is capable to handle triangular hull meshes as well as CSGs as input geometries. Although the Netgen application itself supports multiple segments, this feature is not supported by the API. However, due to its nature as an advancing front algorithm, Netgen preserves the hull elements of the input mesh. This peculiarity enables to implement multi-segment meshing support by treating each segment as a separate mesh. As usually with all advancing front algorithms, the quality of the generated volume mesh strongly depends on the quality of the input hull mesh. Therefore, Netgen provides a hull mesh adaptation algorithm in order to improve the input hull mesh prior to the volume meshing step. This hull mesh adaptation algorithm is also available via the API, but does not offer multi-segment support. The restricted API impedes Netgen's full potential for external applications, nevertheless it is popular and frequently used.

Tetgen [17] is an open source incremental Delaunay mesh generating software written in C++ and distributed under a slightly modified X11 license for non-commercial use. Tetgen generates 3D unstructured tetrahedral meshes, including support for multiple segments. A so-called piecewise linear complex (PLC) is used as input geometry. A PLC is a set of polygons with holes as well as additional vertices and line segments, introducing constraints to the mesh generation. PLCs

Table 2
An overview of meshing functionalities available in ViennaMesh. `PLC` refers to piecewise linear complex, `TRI` refers to triangle, `TET` refers to tetrahedron, and `SAI` is used when the output is of the same type as the input.

Software	Geometric dimension	Input cell type	Output cell type	Algorithm information
CGAL	2, 3	PLC	TRI	Incremental Delaunay
CGAL	3	TRI	TET	Incremental Delaunay
NGLib	3	TRI	TET	Advancing front
Tetgen	3	PLC, TRI	TET	Incremental Delaunay
ViennaGrid	2, 3	TRI, TET	SAI	Refinement of tagged elements
ViennaGrid	3	TET	TET	Refinement of tagged elements
ViennaMesh	3	TRI	TRI	Segmentation based on points
ViennaMesh	3	TET	TRI	Hull extraction
ViennaMesh	3	TRI	PLC	PLC geometry extraction
Vgmodeler	3	TRI	TRI	Hull adaption

Table 3
An overview of features supported by external meshing tools. Relevant features for FEM are selected. $\bigoplus$ indicates that the corresponding feature is supported by the meshing tool and by ViennaMesh, $+$ indicates that this feature is supported by the meshing tool but not by ViennaMesh, and $-$ indicates that this feature is not supported by the meshing tool.

	Mesh generation	Mesh refinement	Mesh coarsening	Multiple segment	Delaunay	Local element sizing function	Input geometry
CGAL 2D conforming meshes	$\bigoplus$	$+$	$-$	$-$	$\bigoplus$	$+$	Boundary representation
CGAL 3D mesh generation	$\bigoplus$	$+$	$-$	$+$	$\bigoplus$	$\bigoplus$	Arbitrary using CGAL oracles[a]
NGLib	$\bigoplus$	$+$	$-$	$\bigoplus$	$\bigoplus$	$+$	CSG, Boundary representation[b]
Tetgen	$\bigoplus$	$+$	$+$	$\bigoplus$	$\bigoplus$	$+$	Boundary representation

[a] ViennaMesh currently supports polyhedral oracles only.
[b] ViennaMesh currently supports boundary representation geometries only.

are often used as boundary representation geometries. The source code and the programming interface are well documented and straightforward to use. Only using a C-string for passing all parameters to the Tetgen library has usability issues with respect to setting up the string or possible collisions when merging multiple setting strings.

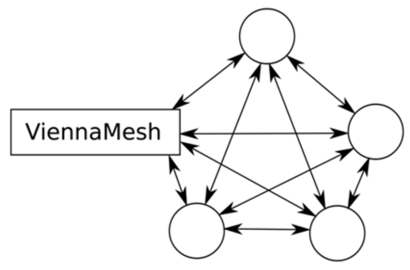
Although a plethora of meshing tools is available, investigations on combining modern concepts for software library design with meshing algorithms are rare. The generic scientific simulation environment framework (GSSE) provides a generic and flexible data structure as well as interfaces to external libraries like CGAL, Netgen, and Tetgen [18,19]. However, the software itself is not available to the public. A design for a flexible meshing framework has been introduced, but only 3D simplex meshes are considered [20]. Another example is the MeshKit [21] framework, offering a flexible graph-based meshing approach. The library is written in C++ and uses the interoperable tools for advanced petascale simulations (ITAPS) [22] mesh and geometry interface for handling data [23]. Although the library offers a significant level of expandability, the API requires significant development effort for basic tasks. For instance, a basic meshing task requires the setup of a graph-based execution mechanism, resulting in development overhead. Additionally, the ITAPS interface only supports a limited number of mesh elements. Although the most popular mesh elements are available, using arbitrary or uncommon element types, like simplices with topological dimension greater than 3, is not possible. Due to the design of the MeshKit graph engine, each data object, which is output of one algorithm and input of another, has to be stored in the ITAPS format. Therefore, the use of external algorithms requires two conversions each time the algorithm is executed. This results in unnecessary conversion operations if one algorithm is utilized multiple times using the same mesh. Additionally, algorithm data used internally by the meshing algorithm, like temporarily calculated information, is only available in the native data structure of the algorithm, thus might be lost when conversion to a central data structure is required.

Overall, the availability of a highly diverse set of meshing tools and the lack of a common interface introduces the need for a flexible meshing framework in order to support arbitrary meshing tasks and a unified interface to various meshing backends.

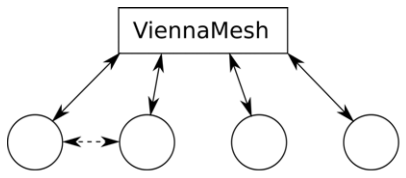
4. ViennaMesh

ViennaMesh is a meshing framework with a uniform and easy-to-use interface for meshing algorithms. ViennaMesh uses a flexible data structure as a common basis for different meshing algorithms.

Table 2 gives an overview of the meshing functionalities which are supported by ViennaMesh and Table 3 lists relevant features supported by external meshing tools. CGAL, NGLib, and Tetgen are used for tetrahedral mesh generation based on triangular hulls as input. An algorithm for creating a two- and three-dimensional triangulation based on a PLC geometry is available using the CGAL library. The NGLib algorithm requires an oriented hull while the Tetgen algorithm is able to process PLC input geometries. Both algorithms support multiple segments. Additionally, ViennaMesh includes a multi-segment hull adaption algorithm, named Vgmodeler, which is based on the respective Netgen implementation. Refinement based on



(a) The naive conversion policy requires two conversion functions for each pair of data structures. $O(n^2)$ conversion functions are required.



(b) The conversion policy of ViennaMesh requires only two conversion functions for each external data structure, i.e., the conversion from and to a specific ViennaGrid mesh. $O(2n)$ conversion functions are needed. Optionally, direct conversion between two native data structures can be provided to increase performance.

Fig. 2. The differences of the naive and the ViennaMesh conversion policy. A conversion function is visualized by an arrow, whereas a data structure is denoted by a circle. The dashed arrow line indicates an optional conversion function. The ViennaMesh policy on the right requires only $O(2n)$ conversion functions while the naive approach on the left requires $O(n^2)$.

element tagging is supported. Furthermore, extraction algorithms are provided, enabling to extract the hull mesh from a volume mesh and the PLC geometry from a hull mesh. ViennaMesh currently does not support any algorithm provided by Gmsh.

ViennaMesh also provides mechanisms for evaluating the quality of a mesh based on the VERDICT library [24]. In essence, quality metrics are utilized to evaluate the individual mesh elements, which in turn is used to evaluate the whole mesh. This is of particular importance to subsequent numerical simulations, in particular finite element simulations, as the element with the worst quality significantly contributes to the quality of the solution [6].

4.1. Mesh data structure

ViennaMesh is based on a flexible mesh data structure provided by the ViennaGrid library [25], which acts as a common basis for the supported mesh operations. The data structure is inspired by the previous work [19,26] and is designed to be as flexible as possible, especially regarding the mesh element topology. For instance, mesh elements of arbitrary topology are supported as well as topological operations, such as neighborhood relationships. ViennaGrid uses a boundary approach for defining the local topology of mesh elements. Note that for a specific topological dimension, there might be more than one boundary element type. For instance, a triangular prism has two triangles and three quadrilaterals as boundary elements of topological dimension $n = 2$. ViennaGrid tackles this inhomogeneous setting by utilizing the generic programming paradigm, allowing the implementation of arbitrary topologically structured mesh elements.

ViennaGrid supports unstructured meshes, polygons, PLCs, and hybrid meshes, being meshes with more than one cell type. On top of the meshes, multiple segmentations for the same mesh are supported, enabling, for instance, partitions based on a geometrical property and simultaneously on a physical property. Convenient access and traversal mechanisms are provided, enabling high performance operations due to compile-time optimizations.

4.2. ViennaMesh design

ViennaMesh uses a central ViennaGrid mesh data structure because of its high degree of flexibility. Therefore, instead of a naive conversion policy offering a complexity of $O(n^2)$ (Fig. 2(a)), a more efficient approach with a complexity of $O(2n)$ is supported (Fig. 2(b)).

ViennaMesh provides a generic interface using C++ templates, supporting the addition of user-defined mesh algorithms using their native input and output data structures. Whenever an algorithm is executed using the ViennaMesh interface, the input mesh is required in a native data structure supported by the algorithm. Often, the input mesh is not represented in such a native data structure and a data conversion is required. ViennaMesh provides a template mechanism which automatically takes care of this conversion process. If the algorithm is able to use the input mesh natively, no input conversion is required. Otherwise the best matching native mesh type of the algorithm is determined using a type dispatch. A conversion function is selected at compile time using template overloading and the mesh is automatically converted to the desired data structure. If no direct conversion function from the input mesh type to the selected native mesh type of the algorithm is available, a default implementation selects two conversion functions using a temporary ViennaGrid mesh as the central data structure. If the provided input mesh is already a native mesh type of the algorithm, the mechanism ensures that no unnecessary conversions are performed. For example, if an algorithm uses the output of another algorithm as an input and these algorithms both support the same data structure, no conversion to any other data structure is required. In contrast, the MeshKit framework always converts the output of an algorithm to an ITAPS mesh. A similar mechanism is available for

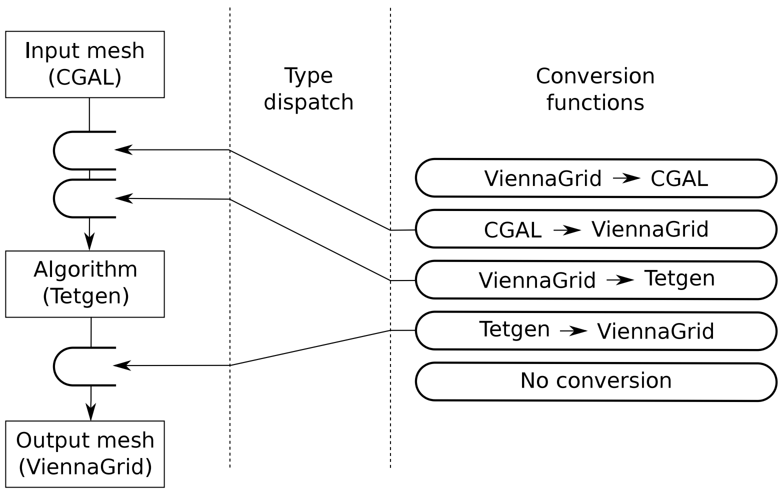


Fig. 3. The conversion mechanism of ViennaMesh illustrated as an example. An input mesh is converted to the native representation of a meshing algorithm if needed. If required, the native output mesh of the algorithm is also converted to the desired output mesh data structure. In this example, the input mesh, represented in a CGAL data structure, has to be converted to Tetgen's native data structure. Due to the lack of a direct conversion function, a temporary ViennaGrid mesh is created to which the input CGAL mesh is converted. This ViennaGrid mesh is then further converted to a Tetgen mesh. This decision and selection process is performed automatically by the conversion mechanism of ViennaMesh. Afterwards, the output of the Tetgen algorithm is converted to ViennaGrid format. Only one conversion is required because a direct conversion function for the mesh from the Tetgen format to the ViennaGrid format is available.

Table 4
An example of ViennaMesh execution time benchmarks of CGAL's and Tetgen's incremental Delaunay tetrahedron mesher. The values for the mesh generation process, the input conversion, and the output conversion are given in seconds (s). The relative input and output conversions are calculated based on the mesh generation process time. The conversion time of the input mesh to the native data structure is less than $4 \cdot 10^{-5}$% of the execution time of the mesh generation process. The conversion time of the native data structure to a ViennaGrid mesh is about 1%–4% of the execution time of the mesh generation process. Both conversion times are negligible compared to the mesh generation process.

	Mesh generation process (s)	Input conversion (s)	Relative input conversion	Output conversion (s)	Relative output conversion
CGAL	$1.4 \cdot 10^{2}$	$5.5 \cdot 10^{-5}$	**3.9% · $\mathbf{10^{-5}}$**	4.7	**3.3%**
Tetgen	$6.9 \cdot 10^{1}$	$2.8 \cdot 10^{-5}$	**4.1% · $\mathbf{10^{-5}}$**	1.2	**1.7%**

output meshes: the output mesh in its native representation of an algorithm is converted to the desired output mesh format if a conversion is needed. Fig. 3 gives an overview of the conversion mechanism illustrated as an example.

Aside from mesh input and output parameters, a ViennaMesh algorithm might require additional parameters such as the maximum cell size. In such cases an additional parameter class has to be implemented for the respective parameters.

In the following the flexibility of ViennaMesh is outlined by utilizing Tetgen's tetrahedral mesh generator on an input hull mesh.

```
1  typedef tetgen_tetrahedron_tag algo_tag;
2
3  result_of::settings<algo_tag>::type settings;
4  settings.cell_size = 1;
5
6  viennagrid::triangular_3d_mesh  hull_mesh;
7  /* import input hull into hull_mesh here */
8
9  viennagrid::tetrahedral_3d_mesh volume_mesh;
10
11 run_algo<algo_tag>(hull_mesh, volume_mesh, settings);
```

Based on a specific algorithm indicating the meshing tool – in this case Tetgen – a `settings` object is generated, providing further access to parameters, such as the maximum cell size (Lines 3–4). The input and output mesh objects are created and an input mesh is imported (Lines 6–9). At last, the algorithm is executed (Line 11). Since neither the input mesh object nor the output mesh object is a natively supported data structure of Tetgen, two conversion processes are automatically triggered by `run_algo`. Note the high level of flexibility, as another mesh generation algorithm can be used by changing the corresponding tag. For instance, the previously utilized Tetgen tag can be exchanged with its CGAL counterpart, changing the entire meshing backend. No other changes in the code are required.

Due to the use of a central data structure and conversions, the utilization of ViennaMesh introduces a certain overhead and demands more memory compared to directly using the meshing algorithm. Benchmarks were performed to quantify these penalties. The results of the performance benchmarks and memory benchmarks are given in Tables 4 and 5, respectively. For CGAL, a mesh with $2.2 \cdot 10^{6}$ cells was created and for Tetgen, a mesh with $1.9 \cdot 10^{7}$ cells. The benchmarks were carried out in a Linux 3.8.0 kernel environment using GCC 4.7.3. A computer with an Intel Core i7-3770 CPU and 16 GB

Table 5
An example of ViennaMesh memory benchmarks of CGAL's and Tetgen's incremental Delaunay tetrahedron mesher. All non-relative values are given in bytes (B), the relative values are calculated by dividing the memory of the ViennaGrid mesh by the memory of the native mesh. The ViennaGrid representation of a mesh requires much less memory (26% and 9% for input and output meshes respectively) than its representation in CGAL. In contrast, the ViennaGrid representation of a mesh requires about 2–3 times more memory than its representation in Tetgen.

	ViennaGrid input (B)	Native input (B)	Relative input	ViennaGrid output (B)	Native output (B)	Relative output
CGAL	$2.7 \cdot 10^4$	$1.0 \cdot 10^5$	**26**%	$1.2 \cdot 10^8$	$1.0 \cdot 10^9$	**9**%
Tetgen	$6.6 \cdot 10^4$	$2.8 \cdot 10^4$	**240**%	$1.0 \cdot 10^9$	$3.8 \cdot 10^8$	**270**%

Table 6
The development of the radius-edge ratio mesh quality for 2 iterations. Lower values indicate better quality. The metric value of the worst element drops from 1.45 to 1.18 after 2 iterations.

Original mesh	Iteration 1	Iteration 2
1.45	1.22	1.18

of memory was used. These benchmarks show, that the execution times for the conversion process are negligible compared to those for the mesh generation process. For meshes with a number of cells less than about 10^7 or 10^8 when using CGAL or Tetgen, respectively, the memory penalty also is not problematic. The overhead due to additionally required conversions is compensated by the reduced complexity.

5. Applications

This section discusses two ViennaMesh application examples relevant for finite element simulations. First, a mesh adaptation application is implemented, aiming for improving the worst element in a mesh (Section 5.1). Second, an adaptive mesh refinement application based on a mixed finite element–finite volume simulation is presented (Section 5.2).

5.1. Cavity remeshing

This application aims to improve the quality of the *worst* element in a 3D tetrahedral mesh. The example is inspired by similar research work, using rigorous meshing techniques to improve the quality of a mesh [10]. We identify the worst element in the mesh and initiate remeshing of an appropriate surrounding volume, which – due to a required caving step – is referred to as cavity. Although this algorithm is rather expensive, it is shown that this optimization approach generates mesh elements with extraordinary good quality.

In the following we present the essential implementation parts. Tetrahedrons surrounding the cell with the worst element metric are tagged for elimination.

```
1  typedef radius_edge_ratio_tag MetricTag;
2  auto worst_element = worst_element<MetricTag>(input_mesh);
3  PointType worst_cell_center = centroid(*worst_element);
4
5  for (auto cit : viennagrid::cells(input_mesh))
6    if (is_cell_in_sphere(worst_cell_center, radius, *cit))
7      cavity_marker(*cit) = true;
8
9  extract_hull(input_mesh, cavity_marker, cavity_hull);
```

We start by meshing a simple cube with side length of 10 using NGLib's advancing front meshing algorithm. Although NGLib generates elements with good quality already, we identify the element with the worst quality metric (Line 2) and seek to improve it. A compile-time tag is used to select the metric, in this case the radius-edge ratio [8]. The centroid of the identified element is calculated with the `centroid` function provided by ViennaGrid (Line 3). The centroid is used as the center of a virtual sphere, defined by a `radius` parameter, used to identify the region to be remeshed (Lines 3–7). The identified cells represent a subdomain, from which the hull can be extracted using the `extract_hull` algorithm (Line 9). Fig. 4 shows the original cube and the hull of the cavity as well as the element with the worst quality metric.

The final output mesh is generated by combining the re-meshed cave and the original mesh with the cavity.

```
1  for (auto cit : viennagrid::cells(input_mesh))
2    if (!cavity_marker(*cit))
3      copy_element(*cit, output_mesh);
4  for (auto cit : viennagrid::cells(meshed_cavity))
5    copy_element(*cit, output_mesh);
```

First, the untagged elements of the initial input mesh are transferred to the output mesh (Lines 1–3). Second, the remeshed region is copied to the output mesh, finalizing the meshing result (Lines 4–5).

Table 6 presents the progress of the worst element during the course of two iteration steps. After two iterations, the metric value of the worst element identified in the mesh reduces from 1.45 to 1.18, indicating remarkable increase in the overall mesh quality.

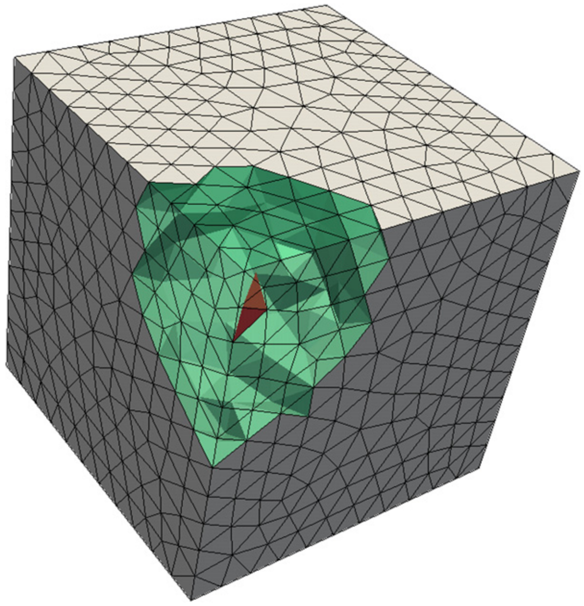


Fig. 4. The hull of the cavity elements (green) and the element with the worst quality metric (red). (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

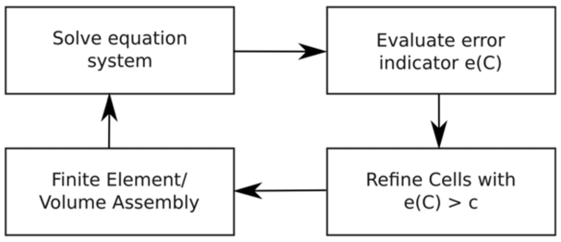
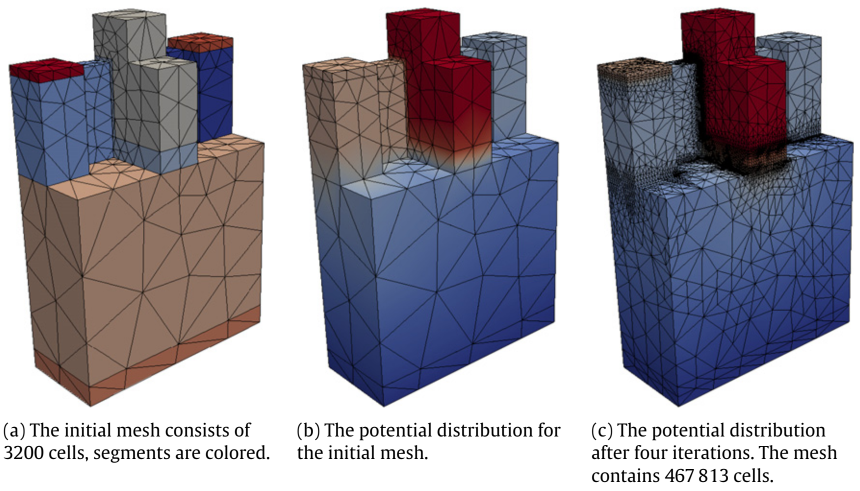

Solve equation system
Evaluate error indicator e(C)
Refine Cells with e(C) > c
Finite Element/ Volume Assembly

Fig. 5. The main iteration loop of the adaptation process. The system of equations is solved followed by the evaluation of the error indicator. Based on this error indicator, all cells with $e(C) > c$ are refined.



(a) The initial mesh consists of 3200 cells, segments are colored. (b) The potential distribution for the initial mesh. (c) The potential distribution after four iterations. The mesh contains 467 813 cells.

Fig. 6. The progression of the adaptive mesh refinement application.

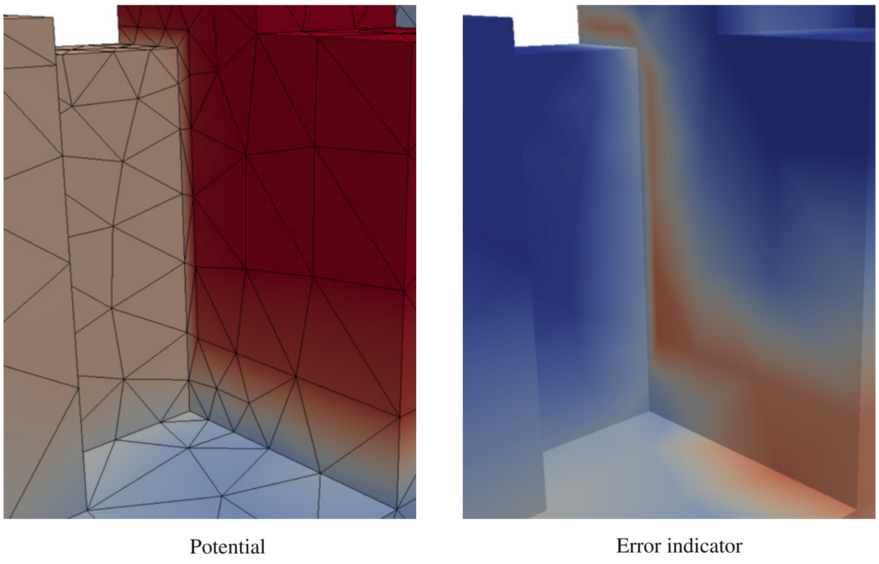


Fig. 7. The potential and the error indicator on the initial mesh. The so-called channel region is shown.

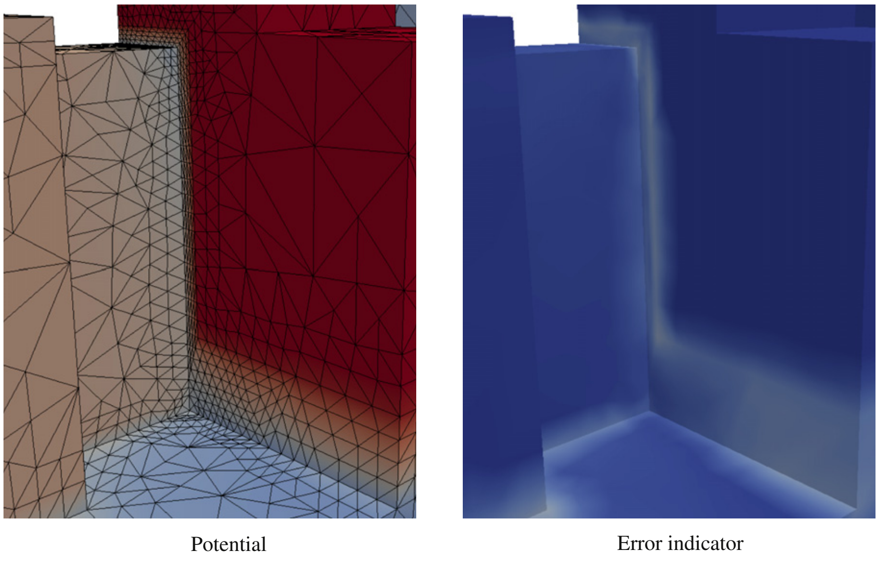


Fig. 8. The potential and the error indicator for the second iteration step.

5.2. *Adaptive mesh refinement*

In this section we present an adaptive mesh refinement application based on a mixed finite element–finite volume method for solving the equations of a drift-diffusion model [1]. Note that for this particular problem, it has been shown that on a discrete level the employed FEM and the finite volume method are the same [11].

At first, an initial mesh is used to calculate the solution to the boundary value problem using finite element assembly. Based on this solution, an error indicator $e(T) := \max_{A \in \text{neighbor}(T)} |\text{value}(T) - \text{value}(A)|$ is used to identify cells for refinement. Elements with $e(T) > c$, for some constant value c, are refined. The new locally adapted mesh is used to calculate a more accurate solution of the boundary value problem. This process is repeated until a certain threshold is achieved. Fig. 5 gives an overview of the adaptive FEM algorithm. In the following, the iteration loop for the iterative process is shown.

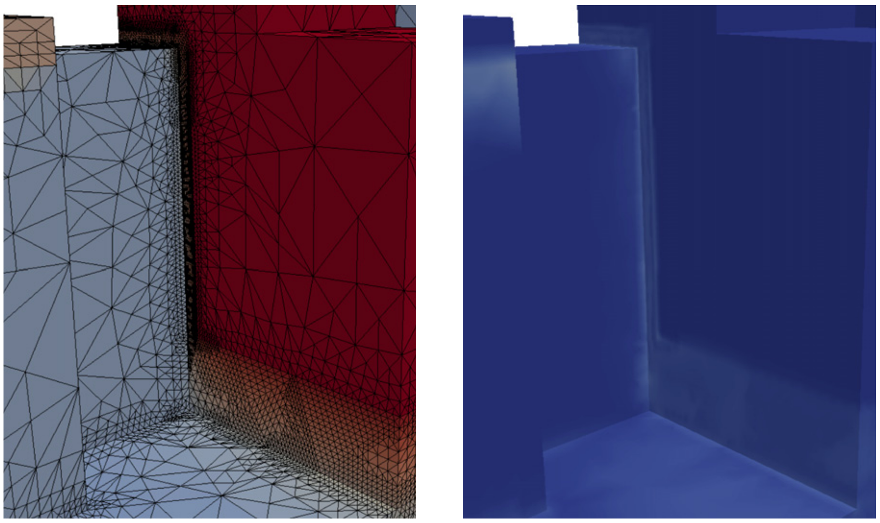


Fig. 9. The potential and the error indicator for the fourth iteration step.

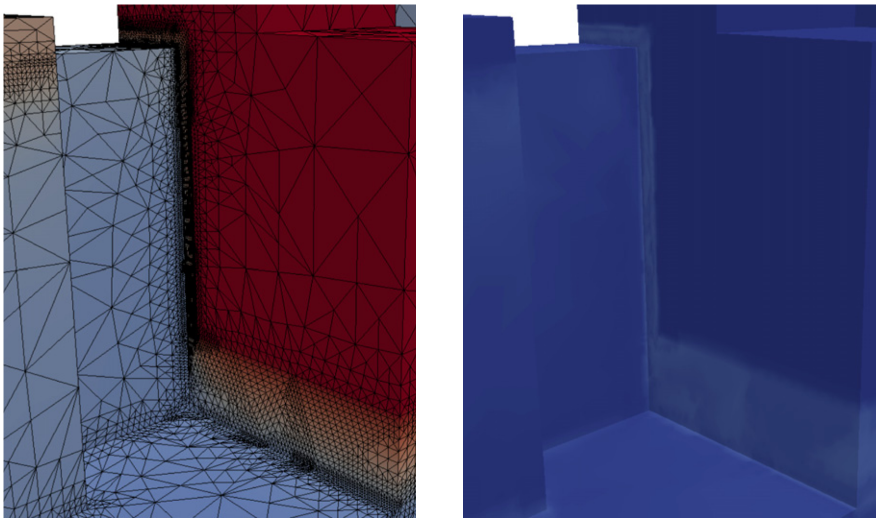


Fig. 10. The potential and the error indicator for the sixth iteration step.

```
 1  while (some_criteria) {
 2    solve_dd(input_mesh, input_segmentation, potential);
 3
 4    for (auto cit = viennagrid::cells(input_mesh))
 5      if (error(*cit, input_mesh, potential) > c )
 6        refinement_marker(*cit) = true;
 7
 8    cell_refine(input_mesh,  input_segmentation,
 9                output_mesh, output_segmentation,
10                refinement_marker);
11
12    input_mesh = output_mesh;
13    input_segmentation = output_segmentation;
14  }
```

The iterative scheme is repeated until specific criteria are met (Lines 1–10), such as indicated by error estimation. The given problem is solved (Line 2), followed by an evaluation of the solution error triggering a tagging mechanism for the subsequent refinement step (Line 3–5). The mesh is refined according to the tagged cells (Lines 6–8) and prepared for the next iteration (Lines 9–10).

We evaluated the feasibility of our application by processing a 3D metal–oxide–semiconductor field-effect transistor (MOSFET) device. A mesh with 3200 tetrahedrons, shown in Fig. 6(a), is used as initial mesh. The solution, being the

Table 7
The number of cells increases in each iteration. At first, the absolute number of bad cells increases and then drops.

Iteration	Total cells	Bad cells	Relative bad cells
1	3 200	2 082	65.06%
2	20 731	8 270	39.89%
3	103 366	40 302	38.99%
4	467 813	63 956	13.67%
5	1 349 050	15 144	1.12%
6	1 727 444	6 194	0.36%

electrostatic potential, on the initial mesh is visualized in Fig. 6(b), while Fig. 6(c) shows the result for the fourth iteration. The progression of the number of cells for six iterations is presented in Table 7. It can be seen that the absolute *bad* cell count, indicating the number of tetrahedron cells with $e(T) > c$, increases at first but then drops significantly. However, the relative bad cell count, defined as the bad cell count divided by the number of total cells, always drops, thus implying convergence. As expected, the setup of the finite element scenario results in a high error indicator for mesh elements near to the channel of the 3D MOSFET transistor. This particular region of interest is visualized in Fig. 7, where the resulting potential and the error indicator are presented. For comparison, Fig. 8 shows the same region of interest after two iterations, Fig. 9 after four iterations, and Fig. 10 after 6 iterations.

6. Conclusion

We presented ViennaMesh, a flexible meshing software framework, providing a uniform interface for mesh-related algorithms. The data structure used by ViennaMesh as well as the design of the framework were discussed. We depicted two applications, being a mesh adaption application, where the mesh quality is improved, as well as an adaptive mesh refinement process on top of a mixed finite element–finite volume simulation.

It has been shown, that it is possible to define and implement uniform programming interfaces for meshing applications without significant overhead in runtime and memory usage.

Acknowledgment

This work has been supported by the European Research Council (ERC) through the grant #247056 MOSILSPIN.

References

[1] S. Selberherr, Analysis and Simulation of Semiconductor Devices, Springer-Verlag, Wien, New York, 1984.
[2] CGAL—Computational Geometry Algorithms Library, September 2013. URL: http://www.cgal.org/.
[3] C. Geuzaine, J.-F. Remacle, Gmsh: a 3-D finite element mesh generator with built-in pre- and post-processing facilities, Internat. J. Numer. Methods Engrg. 79 (2009) 1309–1331. http://dx.doi.org/10.1002/nme.2579.
[4] Netgen Mesh Generator, September 2013. URL: http://sourceforge.net/projects/netgen-mesher/.
[5] Tetgen—A quality tetrahedral mesh generator and a 3D delaunay triangulator, September 2013. URL: http://tetgen.org/.
[6] J.R. Shewchuk, What is a good linear finite element?—interpolation, conditioning, anisotropy, and quality measures, in: Proceedings of the 11th International Meshing Roundtable, 2002, pp. 115–126.
[7] W. Wessner, J. Cervenka, C. Heitzinger, A. Hössinger, S. Selberherr, Anisotropic mesh refinement for the simulation of three-dimensional semiconductor manufacturing processes, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 25 (2006) 2129–2139. http://dx.doi.org/10.1109/TCAD.2005.862750.
[8] S.-W. Cheng, T.K. Dey, J.R. Shewchuk, Delaunay Mesh Generation, CRC Press, 2013.
[9] ViennaMesh, September 2013. URL: http://viennamesh.sourceforge.net/.
[10] B.M. Klingner, J.R. Shewchuk, Agressive tetrahedral mesh improvement, in: Proceedings of the 16th International Meshing Roundtable, Seattle, Washington, 2007, pp. 3–23.
[11] J.J.H. Miller, W.H.A. Schilders, S. Wang, Application of finite element methods to the simulation of semiconductor devices, Rep. Progr. Phys. 62 (3) (1999) 277–353. http://dx.doi.org/10.1088/0034-4885/62/3/001.
[12] C. Frederick, Y. Wong, F. Edge, Two-dimensional automatic mesh generation for structural analysis, Internat. J. Numer. Methods Engrg. 2 (1970) 133–144. http://dx.doi.org/10.1002/nme.1620020112.
[13] P.J. Frey, G. Paul-Louis, Mesh Generation, HERMES Science Publishing, 2000.
[14] S.J. Owen, A survey of unstructured mesh generation technology, in: Proceedings of the 7th International Meshing Roundtable, 1998, pp. 239–267.
[15] C. Jamin, P. Alliez, M. Yvinec, J.-D. Boissonnat, CGALmesh: A Generic Framework for Delaunay Mesh Generation, Research Report 8256, INRIA, 2013.
[16] J. Schöberl, NETGEN—an advancing front 2D/3D-mesh generator based on abstract rules, Comput. Vis. Sci. 1 (1997) 41–52. http://dx.doi.org/10.1007/s007910050004.
[17] H. Si, Three dimensional boundary conforming Delaunay mesh generation, Dissertation, Technische Universität Berlin, 2008.
[18] R. Heinzl, Concepts for scientific computing, Dissertation, Technische Universität Wien, 2007.
[19] R. Heinzl, P. Schwaha, A generic topology library, Sci. Comput. Program. 76 (4) (2011) 324–346. http://dx.doi.org/10.1016/j.scico.2009.09.007.
[20] N. Hitschfeld, C. Lillo, A. Cáceres, M. Bastarrica, M. Rivara, Building a 3D meshing framework using good software engineering practices, in: Advanced Software Engineering: Expanding the Frontiers of Software Technology, Vol. 219, 2006, pp. 162–170. http://dx.doi.org/10.1007/978-0-387-34831-5_13.
[21] MeshKit, September 2013. URL: https://trac.mcs.anl.gov/projects/fathom/wiki/MeshKit/.
[22] ITAPS Interfaces, September 2013. URL: http://www.itaps.org/software/download_interfaces.html.
[23] T.J. Tautges, J. Kraftcheck, J. Porter, MeshKit: an open-source toolkit for mesh generation, in: Book of Abstract of SIAM Conference on Computational Science and Engieering, 2011, pp. 187–188.

[24] P.P. Pébay, D.C. Thompson, J. Shepherd, P. Knupp, C. Lisle, V.A. Magnotta, N.M. Grosland, New applications of the verdict library for standardized mesh verification. pre, post, and end-to-end processing, in: Proceedings of the 16th International Meshing Roundtable, 2007, pp. 535–552. http://dx.doi.org/10.1007/978-3-540-75103-8_30.
[25] ViennaGrid, September 2013. URL: http://viennagrid.sourceforge.net/.
[26] A. Logg, Efficient representation of computational meshes, Int. J. Comput. Sci. Eng. 4 (4) (2009) 283–295. http://dx.doi.org/10.1504/IJCSE.2009.029164.