ViennaGrid 1.0.0

User Manual

Institute for Microelectronics
Gußhausstraße 27-29 / E360
A-1040 Vienna, Austria

Institute for Analysis and Scientific Computing
Wiedner Hauptstraße 8-10 / E101
A-1040 Vienna, Austria/Europe

Main authors:

Karl Rupp (Project Head)
Josef Weinbub

Contributors:

Peter Lagger

Institute for Microelectronics
Vienna University of Technology
Gußhausstraße 27-29 / E360
A-1040 Vienna, Austria/Europe

  Phone   +43-1-58801-36001
  FAX   +43-1-58801-36099
  Web   `http://www.iue.tuwien.ac.at`

Institute for Analysis and Scientific Computing
Vienna University of Technology
Wiedner Hauptstraße 8-10 / E101
A-1040 Vienna, Austria/Europe

  Phone   +43-1-58801-10101
  Web   `http://www.asc.tuwien.ac.at`

Contents

Introduction

The tessellation of surfaces and solids into complexes of small elements such as triangles, quadrilaterals, tetrahedra or hexahedra is one of the major ingredients for many computational algorithms. Applications range from rendering, most notably in computer games, to computational science, in particular for the numerical solution of partial differential equations on complex domains for the study of physical phenomena. These various areas lead to a broad range of different requirements on a mesh library, which certainly cannot be fulfilled by a single, predetermined datastructure. Unlike other mesh libraries, `ViennaGrid` provides the abililty to easily adjust the internal representation of meshes, while providing a uniform interface for the storage and access of data on mesh elements as well as STL-compatible iteration over such elements.

As example, consider the basic building block of triangular meshes, a triangle: The three vertices fully define the shape of the triangle, the edges can be derived from vertices if a common reference orientation of the triangles is provided. Depending on the underlying algorithm, edges of the triangle may or may not be of interest:

- Consider a class `triangle`, holding the three vertices only. A triangular mesh is then some array or list of `triangle`s and an algorithm `algo1` working only on vertices on a per-cell basis can be executed efficiently. An example for such an algorithm is the assembly of a linear, nodal finite element method.

- An `algo2` may need to have global edge information available, i.e. only one instance of an interfacing edge of two triangles should exist in an explicit manner. Thus, storing the edges globally in the domain will allow the use of `algo2`, but will at the same time introduce unnecessary edge information for `algo1`. Finite volume schemes can be seen as an example for this second type of algorithms.

- A third algorithm `algo3` may need global edge information, as well as information about the local orientation of edges with respect to each triangular cell. In such a case it may be preferred to additionally store mappings from global orientations to local orientations of the edges on each triangle if fast execution is desired. Such an additional storage of orientations will render the datastructure well suited for `algo3`, but less suited for `algo1` and `algo2`. An example for such a third type of algorithm are to some extent high-order finite element methods.

The situation for tetrahedral meshes is even more complicated, because additional orientation issues of shared facets come into play.

The aim of `ViennaGrid` is to be highly customizable such that all three algorithms outlined above can be supported with an optimal data layout. In particular, `ViennaGrid` allows for a convenient specification of the storage of elements, in particular which boundary elements are stored inside the domain as well as which topological information is stored
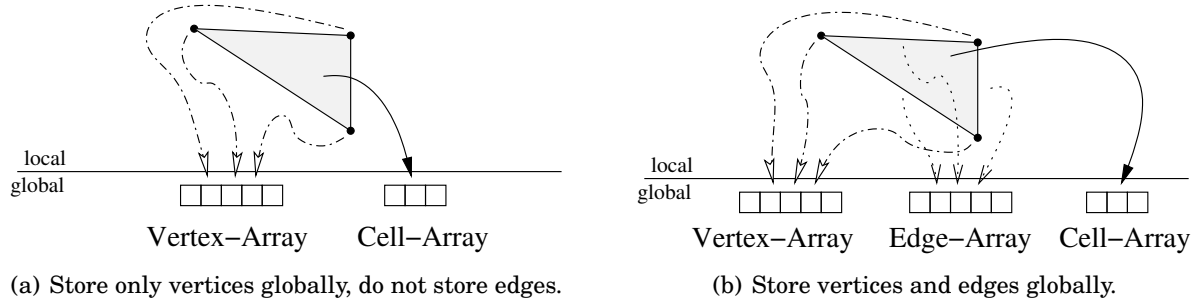
(a) Store only vertices globally, do not store edges.

(b) Store vertices and edges globally.

Figure 1: Two storage schemes for a triangle and the underlying triangular mesh datastructure.

on each element. After a brief introduction into the nomenclature used in this manual in Chapter 2, the configuration of the data storage layout is explained in Chapter 3, and the basic steps required to fill a mesh with cells is explained in Chapter 4.

In addition to a high flexibility with respect to the underlying data structures, `ViennaGrid` provides STL-compatible iterators and access to sub-elements of a mesh, cf. Chapter 5. This allows to write generic code that is a-priori independent of the underlying spatial dimension. In particular, a single implementation for algorithms operating in multiple dimensions and using different mesh types (triangular, hexahedral, etc.) can be obtained.

One of the strengths of `ViennaGrid` is the generic facility provided for storing arbitrary quantities on a mesh, cf. Chapter 6. This is achieved by the use of the special purpose library `ViennaData` [1], providing maximum flexibility for the user.

A typical requirement for a meshing library is mesh refinement. This is in particular of interest for computational science, where singularities near corners need to be resolved sufficiently well. `ViennaGrid` provides both uniform and adaptive refinement algorithms, cf. Chapter 7, where also other geometric algorithms such as Voronoi information is covered.

IO facilities are discussed in Chapter 8, then library internals are discussed in Chapter 9 and and design decisions in Chapter 10.

There are of course other free software libraries having functional overlap with `ViennaGrid`. We give a brief discussion of the pros and cons of selected libraries in the following. This should allow potential users of our library to get a better feeling of what to expect and what not to expect from `ViennaGrid`. We have carefully checked the documentation of each project, but clearly cannot guarantee that all information is fully accurate.

- **CGAL** [2]: The focus of the Computational Geometry Algorithms Library (`CGAL`) is on geometrical algorithms such as the computation of convex hulls of point sets. It offers a mesh generation facility and provides iterators over cell vertices. However, the storage of quantities and the convenient traversal of mesh elements is not provided.

- **DUNE** [3]: `DUNE` follows a similar approach for the generic representation of meshes. It provides support for conforming and non-conforming grids, as well as support for parallel and distributed meshes. However, unlike `ViennaGrid`, we could not find any mechanism providing a convenient means to store data on mesh elements (users are essentially required to handle their data themselves), and for the customization about the internal storage of mesh elements.

- **GrAL** [4]: The Grid Algorithms library (`GrAL`) provides mesh data structured and

algorithms operating on them. A number of principles used in `ViennaGrid` such as *n-cells* already show up in `GrAL` as *k-Elements*. The library does not provide any facility to store data on mesh elements. Mesh refinement is also not provided.

- **libmesh** [5]: The `libmesh` library is not only a mesh library, but also a framework for numerical simulations. Since `ViennaGrid` is designed to be as general as possible without prematurely restricting to a particular application, we only compare the parts in `libmesh` related to mesh handling. `libmesh` supports one-, two- and three-dimensional meshes and also allows to generate meshes for simple domains. Iterations over elements of a mesh are carried out in a runtime manner, thus causing potential overhead. One of the strengths of `libmesh` is the support for mesh refinement and parallel computations. Support for user-defined data on mesh elements is also provided.

- **OpenMesh** [6]: `OpenMesh` provides a generic datastructure for representing and manipulating polygonal meshes. The main goals are flexibility, efficiency and easy-to-use. Similar to `ViennaGrid`, generic programming paradigms are used. `OpenMesh` allows to store custom data of arbitrary type on mesh elements, but it seems to rely on potentially slow string comparisons at run-time to retrieve the data. Moreover, `OpenMesh` is specifically designed for surface (i.e. non-volumetric) meshes, and thus only the concepts of vertices, edges and faces are used.

- **trimesh2** [7]: `trimesh2` is a C++ library that is particularly designed for triangular meshes in 3D only. It explicitly targets efficiency, possibly at the expense of some generality. We could not find further information for a comparison with `ViennaGrid` from the documentation provided.

- **VCGlib** [8]: `VCGlib` processes triangular and tetrahedral meshes. Similar to `OpenMesh`, `VCGlib` uses the concepts of vertices, edges and faces only, so the processing of volume meshes is hampered. Again similar to `OpenMesh`, the provided facility to store data on mesh elements relies on potentially slow string comparisons.

Chapter 1

Installation

This chapter shows how `ViennaGrid` can be integrated into a project and how the examples are built. The necessary steps are outlined for several different platforms, but we could not check every possible combination of hardware, operating system, and compiler. If you experience any trouble, please write to the mailing list at

viennagrid-support@lists.sourceforge.net

1.1 Dependencies

- A recent C++ compiler (e.g. `GCC` version 4.2.x or above and Visual C++ 2005 or above are known to work)

- `ViennaData` [1], version 1.0.1 or above. To make `ViennaGrid` self-contained, a copy of the `ViennaData` sources is available in the `viennadata/` folder.

- `CMake` [9] as build system (optional, but recommended for building the examples)

1.2 Generic Installation of ViennaGrid

Since `ViennaGrid` is a header-only library, it is sufficient to copy the `viennagrid/` source folder either into your project folder or to your global system include path. If you do not have `ViennaData` installed, proceed in the same way for the respective source folder `viennadata/`.

On Unix-like operating systems, the global system include path is usually `/usr/include/` or `/usr/local/include/`. On Windows, the situation strongly depends on your development environment. We advise to consult the documentation of the compiler on how to set the include path correctly. With Visual Studio 9.0 this is usually something like `C:\Program Files\Microsoft Visual Studio 9.0\VC\include` and can be set in `Tools -> Options -> Projects and Solutions -> VC++-Directories`.

File	Purpose
`tutorial/algorithms.cpp`	Demonstrates the algorithms provided, cf. Chapter 7
`tutorial/domain_setup.cpp`	Fill a domain with cells, cf. Chapter 4
`tutorial/finite_volumes.cpp`	Generic implementation of the finite volume method (assembly)
`tutorial/io.cpp`	Explains input-output operations, cf. Chapter 8
`tutorial/iterators.cpp`	Shows how the domain and segments can be traversed, cf. Chapter 5
`tutorial/multi_segment.cpp`	Explains multi-segment capabilities, cf. Chapter 4
`tutorial/slim_domain.cpp`	Customizes ViennaGrid such that no edges are stored, cf. Chapter 3
`tutorial/store_access_data.cpp`	Shows how `ViennaData` is used for storing quantities, cf. Chapter 6

Table 1.1: Overview of the sample applications in the `examples/` folder

1.3 Building the Examples and Tutorials

For building the examples, we suppose that `CMake` is properly set up on your system. The various examples and their purpose are listed in Tab. 1.1.

1.3.1 Linux

To build the examples, open a terminal and change to:

```
$> cd /your-ViennaGrid-path/build/
```

Execute

```
$> cmake ..
```

to obtain a Makefile and type

```
$> make
```

to build the examples. If desired, one can build each example separately instead:

```
$> make algorithms     #builds the algorithms tutorial
```

Speed up the building process by using multiple concurrent jobs, e.g. `make -j4`.

1.3.2 Mac OS X

The tools mentioned in Section 1.1 are available on Macintosh platforms too. For the `GCC` compiler the Xcode [10] package has to be installed. To install `CMake`, external portation tools such as Fink [11], DarwinPorts [12], or MacPorts [13] have to be used.

The build process of `ViennaGrid` is similar to Linux.

1.3.3 Windows

In the following the procedure is outlined for `Visual Studio`: Assuming that `CMake` is already installed, `Visual Studio` solution and project files can be created using `CMake`:

- Open the `CMake` GUI.

- Set the `ViennaGrid` base directory as source directory.

- Set the `build/` directory as build directory.

- Click on 'Configure' and select the appropriate generator (e.g. `Visual Studio 9 2008`)

- Click on 'Generate' (you may need to click on 'Configure' one more time before you can click on 'Generate')

- The project files can now be found in the `ViennaGrid` build directory, where they can be opened and compiled with Visual Studio (provided that the include and library paths are set correctly, see Sec. 1.2).

Note that the examples should be executed from the `build/Debug` and `build/Release` folder respectively in order to access the correct input files.

Chapter 2

The Main Entities

In the following, the main entities of ViennaGrid are explained. The nomenclature essentially follows the convention from topology and can partly be found in other mesh libraries. Note that the purpose of this manual is not to give exact definitions from the field of geometry or topology, but rather to establish the link between abstract concepts and their representation in code within `ViennaGrid`.

2.1 Points (Geometrical Objects)

The underlying space in `ViennaGrid` is the m-dimensional Euclidian space $\mathbb{E}^m$, which we identify with the real coordinate space $\mathbb{R}^m$ in the following. A *point* refers to an element x in $\mathbb{R}^m$ and does not carry any topological information. On the other hand, a point equivalently refers to the vector from the origin pointing to x.

Given a configuration class `Config` for `ViennaGrid` (cf. Chap. 3), a point is defined and manipulated as follows:

```cpp
using namespace viennagrid;

// obtain the point type from a meta-function
typedef result_of::point<Config>::type    PointType;
// For a three-dimensional Cartesian space (double precision),
// the type of the point is returned as point_t<double, 3>

// Instantiate two points:
PointType p1(0, 1, 2);
PointType p2(2, 1, 0);

// Add/Subtract points:
PointType p3 = p1 + p2;
std::cout << p1 - 2.0 * p3 << std::endl;
std::cout << "x-coordinate of p1: " << p1[0] << std::endl;
```

The operators `+`, `-`, `*`, `/`, `+=`, `-=`, `*=` and `/=` can be used in the usual mnemonic manner. `operator[]` grants access to the individual coordinate entries.

Aside from the standard Cartesian coordinates, `ViennaGrid` can also handle polar, spherical and cylindrical coordinate systems. This is typically defined globally within the configuration class `Config` for the whole domain, and the meta-function in the previous snippet
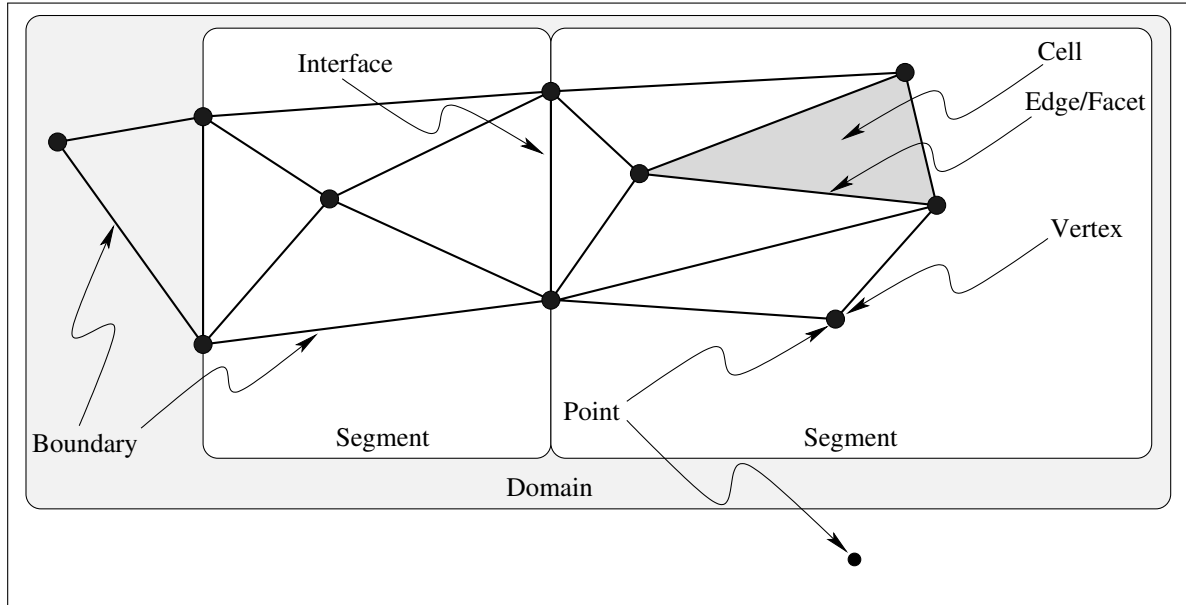
Figure 2.1: Overview of the main entities in `ViennaGrid` for a triangular mesh. A point refers to any location in the geometric space and does not carry topological information.

creates the correct point type. However, if no global configuration class is available, the point types can be obtained as

```
typedef point<double, 1>                CartesianPoint1d;
typedef point<double, 2>                CartesianPoint2d;
typedef point<double, 2, polar_cs>      PolarPoint2d;
typedef point<double, 3>                CartesianPoint3d;
typedef point<double, 3, spherical_cs>  SphericalPoint3d;
typedef point<double, 3, cylindrical_cs> CylindricalPoint3d;
```

Conversions between the coordinate systems are carried out implicitly:

```
CylindricalPoint3d p1(1, 1, 5);
CartesianPoint3d p2 = p1; //implicit conversion
```

An explicit conversion to the Cartesian coordinate system is provided by the free function `to_cartesian()`.

> For details on the coordinate systems, refer to the reference documentation in `doc/doxygen/`.

Since all coordinate systems refer to a common underlying Euclidian space, the operator overloads remain valid even if operands are given in different coordinate systems. In such a case, the coordinate system of the resulting point is given by the coordinate system of the left hand side operand:

```
CylindricalPoint3d p1(1, 1, 5);
CartesianPoint3d p2 = p1; //implicit conversion

// the result of p1 + p2 is in cylindrical coordinates
CylindricalPoint3d p3 = p1 + p2;
```

```
// the result of p2 + p1 is in Cartesian coordinates,
// but implicitly converted to cylindrical coordinates:
CylindricalPoint3d p4 = p2 + p1;
```

For additional algorithms acting on points, e.g. `norm()` for computing the norm/length of a vector, please refer to Chapter 7.

> VillnaGrid is not restricted to one, two or three geometric dimensions! Cartesian
> coordinate systems for arbitrary dimensions are available.

2.2 n-Cells (Topological Objects)

While the point type defines the underlying geometry, elements define the topological connections among distinguished points. Each of these distinguished points is called a *vertex* and describes the corners or intersection of geometric shapes. Vertices are often also referred to as the *nodes* of a mesh.

An *edge* is a line segment joining two vertices. Note that this is a topological characterization – the underlying geometric space can have arbitrary dimension.

A *cell* is an element of maximum topological dimension N within the set of elements considered. The topological dimension of cells can be smaller than the underlying geometric space, which is for example the case in triangular surface meshes in three dimensions. Note that the nomenclature used among scientists is not entirely consistent: Some refer to topologically three-dimensional objects independent from the topological dimension of the full mesh as cells, which is not the case here.

The surface of a cell consists of *facets*, which are objects of topological dimension $N-1$. Some authors prefer the name *face*, which is by other authors used to refer to objects of topological dimension two. Again, the definition of a facet refers in `VillnaGrid` to topological dimension $N-1$.

A brief overview of the corresponding meanings of vertices, edges, facets and cells is given in Tab. 2.1. Note that edges have higher topological dimension than facets in the one-dimensional case, while they coincide in two dimensions. Refer also to Fig. 2.1.

For a parametric description of these objects it is more appropriate to switch n-**cells**, which are objects of topological dimension n. The terms *vertex*, *edge*, *facet* and *cell* thus refer to 0-cell, 1-cell, $N-1$-cell and N-cell respectively. The maximum topological dimension N is in `VillnaGrid` considered to be a parameter available at compile time, which allows to pick the correct n-cell type for the algorithms provided. Note that one can still use a run

	1-d	2-d	3-d	n-d
Vertex	Point	Point	Point	Point
Edge	Line	Line	Line	Line
Facet	Point	Line	Triangle, etc.	$n-1$-Simplex, etc.
Cell	Line	Triangle, etc.	Tetrahedron, etc.	n-Simplex

Table 2.1: Examples for the vertices, edges, facets and cells for various topological dimensions.

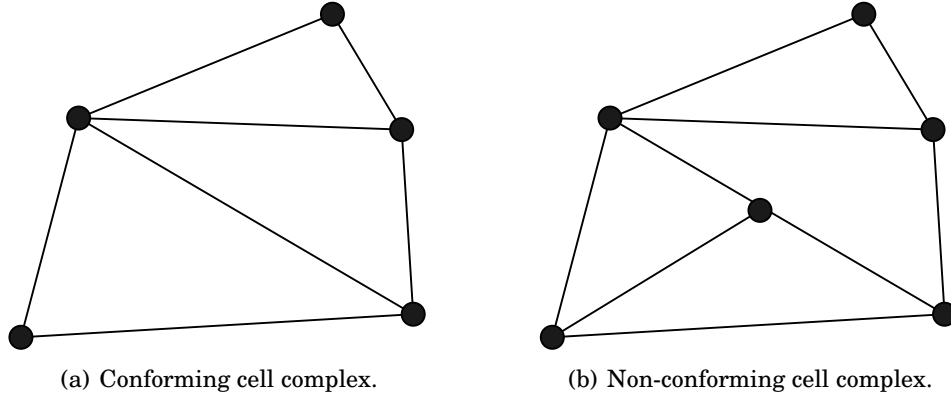(a) Conforming cell complex.   (b) Non-conforming cell complex.

Figure 2.2: Illustration of conforming and non-conforming cell complexes. The vertex in the center of (b) intersects an edge in the interior, violating the conformity criterion.

time dispatch at the cost of increased compilation times if the dimension of the domain is only available at run time. More details can be found in Chapters 5 and 7.

2.3   Domain

A *domain* Ω is the top level object in `ViennaGrid` and is a container for its topological elements. There are no topological restrictions on the elements inside a domain.

A typical use-case for a domain is to store a cell complex. We characterize a cell complex as a collection of n-cells, $n = 0, \ldots, N$, such that the intersection of a n_1-cell from the cell complex with a n_2-cell from the cell complex is a n_3-cell from the cell complex. A domain stores the various n-cells of the complex, but can be configured to store n-cells for selected values of n only, cf. Chapter 3.

It should be noted that `ViennaGrid` allows the use of conforming and non-conforming complexes. Here, a *conforming* complex is characterized by the property that the intersection n_3-cell from above is a boundary n_3-cell from both the n_1-cell and the n_2-cell. If this is not the case, the complex is denoted *non-conforming*, cf. Fig. 2.2.

The instantiation of a `ViennaGrid` domain object requires a configuration class `Config`, as will be discussed in Chapter 3. Given such a class, the domain type is retrieved and the domain object constructed as

```
using namespace viennagrid;

// Type retrieval, method 1: use meta-function (recommended)
typedef result_of::domain<Config>::type     DomainType;

// Type retrieval, method 2: direct (discouraged, may change in future
   versions)
typedef domain_t<Config>     DomainType

DomainType domain; //create the domain object
```

2.4 Segment

A *segment* Ω_i refers to a subset of the N-cells in a domain Ω. In addition, a segment may or may not have explicit knowledge of the boundary n-cells of its N-cells. Unlike a domain, a segment is not a container for its elements. Instead, only references (pointers) to the elements in the domain are stored. In common C++ language, a *segment* represents a so-called *view* on the domain.

The typical use-case for a segment is the decomposition of the domain into pieces of a common property. For example, a solid consisting of different materials can be set up in `ViennaGrid` such that each regions of the same material are represented in a common segment.

> In `ViennaGrid 1.0.0`, a N-cell is limited to be member of at most one segment. Segments need to be initialized during domain setup, cf. Chapter 4.

Chapter 3

Domain and Element Configuration

A domain and all its topological elements as well as the underlying geometric space are specified in a common configuration class. The setup of such a configuration class is explained in detail in Section 3.1.

Even though mathematically all n-cells of a domain are well defined and available on paper, it may not make sense to store explicit representations of these elements in memory. As discussed in the Introduction, a particular algorithm may not require the storage of edges and/or facets of a domain, thus keeping these elements in memory can be a considerable waste of resources. Section 3.2 explains how the underlying storage scheme of `ViennaGrid` can be adjusted to the library user's requirements and to be as memory-efficient as possible.

3.1   The Configuration Class

A valid configuration class for `ViennaGrid` is any class that provides the following three public member types:

`numeric_type`	The underlying scalar type of the geometric space.
`coordinate_system_tag`	A tag specifying the underlying coordinate system.
`cell_tag`	A tag that identifies the cell type inside the domain.

More details on the types are given in the following:

- `numeric_type`: This refers to the type of the coordinates of a point in the geometric space. In most cases, one may want to use **double**. However, it may also be the case that the user prefers to use integer coordinates, or high-precision floating point libraries such as ARPREC [14].

- `coordinate_system_tag`: Any of the following predefined classes from namespace `viennagrid` can be defined as `coordinate_system_tag` to select the coordinate system of the underlying geometric space:

`cartesian_cs<dim>`	Cartesian coordinate system of dimension `dim`
`polar_cs`	Polar coordinate system in two dimensions
`spherical_cs`	Spherical coordinates in three dimensions
`cylindrical_cs`	Cylindrical coordinates in three dimensions

- `cell_tag`: A tag that specifies the type of the elements with maximum topological dimension in the domain. The following two families of topological elements are provided with `ViennaGrid` in namespace `viennagrid`:

`simplex<n>`	Refers to an n-simplex
`hypercube<n>`	Refers to an n-hypercube

It should be noted that `simplex<1>` and `hypercube<1>` both refer to a line segment. Typical examples from these families for common values of n are as follows:

	`ViennaGrid` tag type	Convenience **typedef**
Vertex	`point_tag`	-
Line	`simplex_tag<1>`	`line_tag`
Line	`hypercube_tag<1>`	-
Triangle	`simplex_tag<2>`	`triangle_tag`
Quadrilateral	`hypercube_tag<2>`	`quadrilateral_tag`
Tetrahedron	`simplex_tag<3>`	`tetrahedron_tag`
Hexahedron	`hypercube_tag<3>`	`hexahedron_tag`

The reference orientations of these cells can be found in Appendix A.

All geometric and topological types inside the domain are then derived from the configuration class. To this end, `ViennaGrid` provides a number of metafunctions that reside in namespace `viennagrid::result_of`. The naming follows the conventions in the Boost libraries [15]. For a configuration class `Config`, the respective types are obtained as follows:

```
using namespace viennagrid;

typedef result_of::domain<Config>::type      DomainType;
typedef result_of::segment<Config>::type      SegmentType;
typedef result_of::ncell<Config, 0>::type      VertexType;
typedef result_of::ncell<Config, 1>::type      EdgeType;
```

In particular, the type of any n-cell in the domain is obtained as

```
typedef result_of::ncell<Config, n>::type      ElementType;
```

where n has to be replaced with the respective value. This allows to formulate algorithms such as a domain boundary detection in a very general manner and can be used for recursively iterating through the n-cells of different dimension. In order to obtain the types of cells and facets, one should use the topological dimension from the cell tag:

```
typedef Config::cell_tag                               CellTag;
typedef result_of::ncell<Config, CellTag::dim-1>::type  FacetType;
typedef result_of::ncell<Config, CellTag::dim>::type    CellType;
```

	Amount	Bytes/Object	Total Memory
Vertices	4913	24	115 KB
Edges	31024	16	485 KB
Facets	50688	48	2376 KB
Cells	24576	112	2688 KB
Total			**5664 KB**

Table 3.1: Minimum memory consumption for a tetrahedral mesh of the unit cube where all n-cells are stored explicitly and boundary information on each n-cell is stored. Vertices use `double`-precision coordinates and elements are linked with pointers sized eight bytes each.

	Amount	Bytes/Object	Total Memory
Vertices	4913	24	115 KB
Edges	0	-	0 KB
Facets	0	-	0 KB
Cells	24576	32	768 KB
Total			**883 KB**

Table 3.2: Minimum memory consumption for a tetrahedral mesh of the unit cube where only 0-cells and 3-cells are stored explicitly. 3-cells have knowledge of their 0-cells only. Vertices use `double`-precision coordinates and elements are linked with pointers sized eight bytes each.

> Please note that in template classes and template functions one needs to add an extra `typename` after each `typedef` keyword in the code snippets above.

3.2 Customizing Storage of n-Cells

One of the outstanding features of `ViennaGrid` over other libraries related to grid handling is the possibility to customize the internal storage of elements. By default, all n-cells of a domain are stored explicitly inside the domain. In addition, each n-cell stores links to its boundary k-cells, with $k < n$. While this overhead is not a concern for topologically one-dimensional meshes and moderate for topologically two-dimensional meshes, it can be a considerable issue for dimensions three and above.

As an example, linear finite elements need information about the cells and the vertices inside the domain, while edges and facets may not be needed depending on the boundary conditions imposed.

As shown in Tab. 3.1 and Tab. 3.2, storing facets and edges explicitly increases the total memory consumption by a factor of more than six. In cases where additional maps for the mapping from local to global n-cell orientations are stored together with the topological boundary information, the difference in memory consumption is even higher.

`ViennaGrid` can be customized to switch seamlessly between the different storage models,

as will be shown in the following subsections.

> Note that in `ViennaGrid` a cell always stores pointers to its vertices. Similarly, the domain always stores the cells and vertices.

3.2.1 Boundary n-Cells

Disabling the storage of boundary n-cells can be achieved in two ways:

- **Method 1**: Use the provided macros. The storage of elements can be customized either for a particular configuration class, or globally for all domain configuration classes. To disable the storage of edges (i.e. 1-cells) inside a tetrahedron for a domain with configuration class `MyConfig`, the line

```
VIENNAGRID_DISABLE_BOUNDARY_NCELL(MyConfig,
                                  viennagrid::tetrahedron_tag, 1)
```

right after the `ViennaGrid`-includes in the source file containing `main()` is sufficient. Make sure that the macro is placed in the global namespace. The first argument is the configuration class, the second class is the tag for the respective n-cell for which the boundary element should be disabled, and the third parameter is the topological dimension for which the handling should be disabled. If no edges should be stored for a tetrahedron irrespective of the configuration class, the macro

```
VIENNAGRID_GLOBAL_DISABLE_BOUNDARY_NCELL(viennagrid::tetrahedron_tag,
                                         1)
```

should be used. If one wishes to selectively enable the handling of boundary cells within a globally disabled environment, one can use e.g.

```
VIENNAGRID_ENABLE_BOUNDARY_NCELL(MyConfig,
                                 viennagrid::tetrahedron_tag, 1)
```

to enable the storage of edges inside a tetrahedron for a configuration class `MyConfig`.

- **Method 2**: The macros above are shortcuts for class specializations. The first two macros expand to the following code:

```cpp
namespace viennagrid { namespace result_of {
    template <>
    struct bndcell_handling<MyConfig,
                            viennagrid::tetrahedron_tag, 1>{
      typedef no_handling_tag    type;
    };

    template <typename ConfigType>
    struct bndcell_handling<ConfigType,
                            viennagrid::tetrahedron_tag, 1>{
      typedef no_handling_tag    type;
    };
} }
```
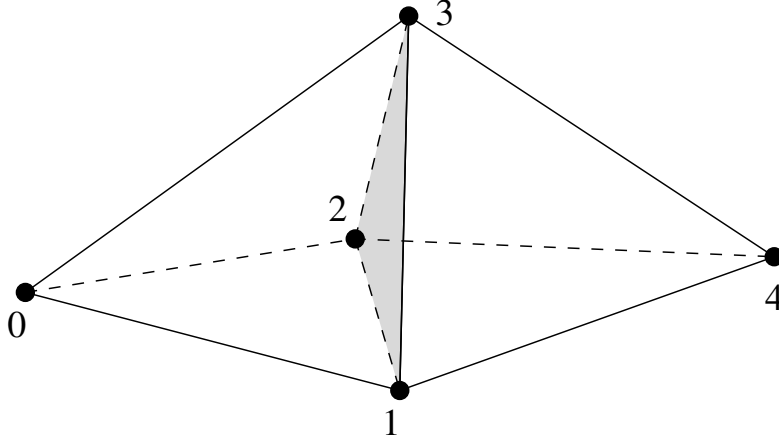
Figure 3.1: Boundary n-cells have different orientations with respect to different cells.

The third macro explained for method 1 results in the same code as the first macro, with the exception that `no_handling_tag` is replaced with `full_handling_tag`.

Finally, we wish to point at a potential pitfall when disabling n-cells that are neither 0-cells, $N-1$-cells, nor N-cells: Consider the case of disabling edges for a tetrahedron. Using the line

```
VIENNAGRID_DISABLE_BOUNDARY_NCELL(MyConfig,
                                  viennagrid::tetrahedron_tag, 1)
```

only disables the edges for the tetrahedron, but still edges are pushed and stored inside the domain, because the triangular facets of the tetrahedron store pointers to their edges. Thus, to completely disable the storage of any edges, one also has to disable the storage of edges for triangles. As a check, one should verify that the number of edges stored inside a domain `my_domain` and obtained by

```
viennagrid::ncells<1>(my_domain).size()
```

is zero.

> Have a look at `examples/tutorial/slim_domain.cpp` and selectively enable or disable the storage of elements to see the impact on total memory consumption. Note that the bytes per object may differ from Tab. 3.1 and Tab. 3.2 depending on the use of local-to-global-orienters and the eventual use of integral IDs.

3.2.2 Local Orientations

In cases where n-cells store boundary k-cells, $k < n$, it may be required to know the orientation of the globally (i.e. in the domain) stored k-cell with respect to the local reference orientation of the n-cell. The common facet of the two tetrahedra in Fig. 3.1 may be stored globally using the sequence of vertex indices $[1, 2, 3]$. Inside the tetrahedron $[0, 1, 2, 3]$ the facet again possesses the orientation $[1, 2, 3]$, thus matching the global orientation. However, within the tetrahedron $[4, 2, 1, 3]$ the facet has the local orientation $[2, 1, 3]$ (refer to the Appendix for an overview of reference orientations).

For applications such as high-order finite elements, such a local orientation with respect to the cell is required.

By default, `ViennaGrid` stores the local orientation of each boundary k-cell inside the hosting n-cell, where one byte per vertex of each k-cell is consumed. In order to disable the storage of these local orientations for scenarios where the local orientation is not needed, there are again two methods provided:

– **Method 1**: Using the macros provided. To disable the storage of local orientations of edges (i.e. 1-cells) inside a tetrahedron for a domain with configuration class `MyConfig`, the line

```
VIENNAGRID_DISABLE_BOUNDARY_NCELL_ORIENTATION(MyConfig,
                           viennagrid::tetrahedron_tag, 1)
```

is sufficient. The second argument denotes the hosting n-cell tag, while the third argument denotes the topological dimension k of the boundary k-cells.

Similar to the previous section, one can disable the storage of local orientations globally for all configuration classes by

```
VIENNAGRID_GLOBAL_DISABLE_BOUNDARY_NCELL_ORIENTATION(
                           viennagrid::tetrahedron_tag, 1)
```

To selectively enable the storage of local orientations in a globally disabled setting, the macro `VIENNAGRID_ENABLE_BOUNDARY_NCELL_ORIENTATION` can be used in the same way as `VIENNAGRID_DISABLE_BOUNDARY_NCELL_ORIENTATION`. Again, it is recommended to define the macros in the source file containing `main()` after the inclusion of the `ViennaGrid` header files.

– **Method 2**: Instead of using the macros, one may manually adjust the metafunction `bndcell_orientation`. The first macro of method 1 expands to

```
namespace viennagrid { namespace result_of {
  template <>
  struct bndcell_orientation<MyConfig,
                             viennagrid::tetrahedron_tag, 1>
  {
    typedef no_handling_tag    type;
  };
}  }
```

Using the type `full_handling_tag` instead of `no_handling_tag` enables the storage of local orientations.

If the storage of boundary k-cells is disabled (cf. Sec. 3.2.1), there are automatically no local orientations for boundary k-cells stored. The local orientations do not need to be disabled explicitly in this case.

3.2.3 n-Cell IDs

It is often convenient to have enumerated n-cells within a domain such that every n-cell is aware of its index inside the random access container it resides in. This allows

for some algorithms to use fast random-accesses with $\mathcal{O}(1)$ costs rather than searching trees with $\mathcal{O}(\log N)$ costs or linked lists with $\mathcal{O}(N)$ costs, where N here denotes the number of n-cells inside the domain. Conversely, if $\mathcal{O}(\log N)$ access times are acceptable, then the memory for storing the container index (which we refer to in the following as *n-cell ID*) is wasted. To accomodate for these two scenarios, `ViennaGrid` allows to selectively enable or disable the storage of n-cell IDs. By default, n-cell IDs are stored, since they have negligible impact on the total memory consumption as soon as other boundary k-cells are stored for an n-cell.

As for the storage of boundary k-cells and local orientations, there are two possibilities for customizing the storage of IDs:

- **Method 1**: Using the provided macros. To disable the storage of IDs for tetrahedrons inside a domain with configuration `MyConfig`, the line

```
VIENNAGRID_DISABLE_NCELL_ID(viennagrid::config::tetrahedral_3d,
                            viennagrid::tetrahedron_tag)
```

  is sufficient. The second argument denotes the element tag for which the storage of IDs should be disabled.
  To globally disable the storage of IDs for tetrahedrons, use

```
VIENNAGRID_GLOBAL_DISABLE_NCELL_ID(viennagrid::tetrahedron_tag)
```

  To selectively enable the storage of IDs for certain configuration classes in a globally disabled setting, one can use

```
VIENNAGRID_ENABLE_NCELL_ID(viennagrid::config::tetrahedral_3d,
                           viennagrid::tetrahedron_tag)
```

- **Method 2**: The macros from the first method expand to

```cpp
template <>
struct element_id_handler<MyConfig, viennagrid::tetrahedron_tag>
{  typedef pointer_id    type;  };
```

  for the configuration class `MyConfig`, and to

```cpp
template <typename ConfigType>
struct element_id_handler<ConfigType, viennagrid::tetrahedron_tag>
{  typedef pointer_id    type;  };
```

  for globally disabling the storage of IDs. To selectively enable IDs, `integral_id` should be used instead of `pointer_id`.

Have a look at `examples/tutorial/slim_domain.cpp` and selectively enable or disable the storage of elements, local orientations and IDs to see the impact on total memory consumption. Note that the bytes per object may differ from Tab. 3.1 and Tab. 3.2 depending on the use of local orientations and integral IDs.

Disabling n-cell IDs eliminates the possibility to store and access data for the respective n-cells with $\mathcal{O}(1)$ costs, see Chapter 6.

Chapter 4

Domain and Segment Setup

This chapter explains how a `VicnnaGrid` domain can be filled with cells. Since this is a necessary step in order to do anything useful with `ViennaGrid`, it is explained in detail in the following. Existing file readers and writers are explained in Chapter 8.

> A tutorial code can be found in `examples/tutorial/domain_setup.cpp`. 💡

In the following, the simple triangular mesh shown in Fig. 4.1 will be set up. Thus, the domain type using the provided configuration class for two-dimensional triangular classes will be used:

```
typedef viennagrid::config::triangular_2d                 ConfigType;
typedef viennagrid::result_of::domain<ConfigType>::type   DomainType;

DomainType domain;    //The domain to be set up in the following
```

If these lines are used inside a template class or template function, an additional **typename** needs to be put after **typedef** in the second line. The created domain object will be filled with vertices and cells in the following.
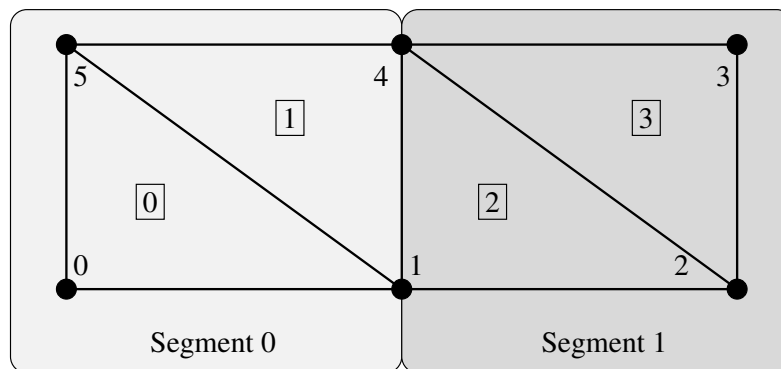

Figure 4.1: An exemplary mesh for demonstrating the use of `ViennaGrid`. Cell-indices are boxed.

4.1 Adding Vertices to a Domain

Since vertices carry geometric information by means of an associated point type, we first obtain the respective point type from the meta-function `point<>`:

```
typedef viennagrid::result_of::point<ConfigType>::type    PointType;
```

This already allows us to add the vertices shown in Fig. 4.1 one after another to the domain. One example to add the first vertex is to push the respective point to the domain:

```
PointType p(0,0);
domain.push_back(p);   // add vertex #0
```

The member function `push_back` appends the vertex to the domain and assigns the respective ID, provided that the use of IDs is not disabled, cf. Sec. 3.2.3.

To push the next vertex, one can either reuse the existing vertex:

```
p[0] = 1.0;
p[1] = 0.0;
domain.push_back(p);   // add vertex #1
```

or directly construct the respective points in-place:

```
domain.push_back(PointType(2,0));   // add vertex #2
domain.push_back(PointType(2,1));   // add vertex #3
domain.push_back(PointType(1,1));   // add vertex #4
domain.push_back(PointType(0,1));   // add vertex #5
```

Note that in `ViennaGrid 1.0.0` vertices have to be provided in ascending order to the domain.

> In `ViennaGrid 1.0.0` vertices always have to be added to the domain directly and cannot be added indirectly via a segment.

If the geometric location of a particular vertex needs to be adjusted at some later stage, the free function `ncells` can be used. To e.g. access vertex #4,

```
viennagrid::ncells<0>(domain)[4]
```

returns a reference to the respective element, which can then be manipulated accordingly.

> More details on the `ncells`-function can be found in Chap. 5 as well as in the reference documentation located in `doc/doxygen`.

4.2 Adding Cells to a Domain or Segment

The type of cells in a domain is obtained by the `ncell` meta-function. Since triangles are topologically two-dimensional, one can directly use

```
typedef viennagrid::result_of::ncell<ConfigType, 2>::type   TriangleType;
```

However, it should be emphasized that this type definition does yield the cell type for e.g. tetrahedral domains. Thus, the more generic way of defining the cell type is to use the topological dimension stored in the cell tag of the domain configuration class:

```
typedef ConfigType::cell_tag                               CellTag;
typedef viennagrid::result_of::ncell<ConfigType,
                                     CellTag::dim>::type    CellType;
```

These two lines reliable return the cell type for all types of domains. Note that an additional **typename** is required if these lines are put inside a template class or template function.

For what follows, the vertex type is also required and readily obtained using

```
typedef viennagrid::result_of::ncell<ConfigType, 0>::type    VertexType;
```

An overview of the generic procedure for adding a cell to a domain or segment is the following:

- Set up an array holding the pointers to the vertices *in the domain*. Do not use pointers to vertices defined outside the domain.

- Instantiate a cell object and add the array of vertex pointers to the cell

- Add the cell to the domain. It will be copied to the domain and can be reused for specifying the next cell.

Thus, the array of pointers to vertices is created as usual:

```
VertexType * cell_vertices[3];
```

Instead of hard-coding the number of vertices for a triangle, one can instead use

```
VertexType * cell_vertices[viennagrid::topology::bndcells<CellTag,0>::num];
```

Next, the vertex addresses for the first triangle are stored:

```
cell_vertices[0] = &(viennagrid::ncells<0>(domain)[0]); // vertex #0
cell_vertices[1] = &(viennagrid::ncells<0>(domain)[1]); // vertex #1
cell_vertices[2] = &(viennagrid::ncells<0>(domain)[5]); // vertex #5
```

Make sure that the correct cell orientation is used, cf. Appendix! ⚠️

Then, a cell is instantiated and the vertices are set:

```
CellType cell;
cell.vertices(cell_vertices); //set cell vertices from above
```

Now, the cell is ready to be pushed either to a segment or a domain.

The example mesh consists of two segments, which have to be created first:

```
domain.segments().resize(2);
```

For completeness it should be mentioned that the segment type can be obtained similar to the domain type from a meta-function:

```
typedef viennagrid::result_of::segment<ConfigType>::type   SegmentType;
```

Two shortcut references to the two segments can be obtained as

```
SegmentType & seg0 = domain.segments()[0];
SegmentType & seg1 = domain.segments()[1];
```

The cell set up above is then pushed to `seg0` as

```
seg0.push_back(cell);
```

As for vertices, cells have to be pushed in ascending order in order to get the correct IDs assigned. Note that the cell is always stored inside the domain - a segment keeps a reference to the cell as well as its boundary cells only.

In the same way the other triangles are pushed to the respective segment. For triangle **#**3, the code is

```
cell_vertices[0] = &(viennagrid::ncells<0>(domain)[2]); // vertex #2
cell_vertices[1] = &(viennagrid::ncells<0>(domain)[3]); // vertex #3
cell_vertices[2] = &(viennagrid::ncells<0>(domain)[4]); // vertex #4
cell.vertices(cell_vertices); // set new cell vertices.
seg1.push_back(cell);          // add cell to seg1
```

It should be noted again that one may also `push_back()` the cells directly to the `domain` if no segments are used.

> The restriction of filling segments already at domain setup may be relaxed or removed in future versions of `ViennaGrid`.

22

Chapter 5

Iterators

The (possibly nested) iteration over n-cells of a mesh is one of the main ingredients for a plethora of algorithms. Consequently, `ViennaGrid` is designed such that these iterations can be carried out in a unified and flexible, yet efficient manner.

At the heart of the various possibilities is the concept of a *range*. A range provides iterators for accessing a half-open interval `[first,one_past_last)` of elements and provides information about the number of elements in the range. However, a range does not 'own' the elements which can be accessed through it [15]. Employing the range-concept, any iteration over n-cells in `ViennaGrid` consists of two phases:

- Initialize the range of n-cells over which to iterate.

- Iterate over the range using the iterators returned by the member functions `begin()` and `end()`.

For convenience, a range may also provide access to its elements using **operator**`[]` (i.e. random access) and thus allowing an index-based iteration. The conditions for random access availability will also be given in the following.

A tutorial code can be found in `examples/tutorial/iterators.cpp`.

5.1 n-Cells in a Domain or Segment

As usual, the first step is to obtain the types for the range and the respective iterator. To iterate over all n-cells of a `domain` of type `DomainType`, the types can be obtained from the `ncell_range` and `iterator` meta-functions:

```
using namespace viennagrid;


//non-const:
typedef result_of::ncell_range<DomainType, n>::type    NCellRange;
typedef result_of::iterator<NCellRange>::type          NCellIterator;
```

For segments, the occurances of `DomainType` and `domain` have to be replaced by `SegmentType` and `segment` here and in the following. If **const**-access to the elements is sufficient, the

metafunction `const_ncell_range` should be used instead of `ncell_range`. For instance, the required types for a **const**-iteration over vertices is given by

```
//const:
typedef result_of::const_ncell_range<DomainType, 0>::type
    ConstVertexRange;
typedef result_of::iterator<ConstVertexRange>::type
    ConstVertexIterator;
```

The next step is to set up a range object using the `ncells` function. The general case of n-cells is handled by

```
NCellRange elements = viennagrid::ncells<n>(domain);
```

For the example of const-iteration over vertices, this results in

```
ConstVertexRange vertices = viennagrid::ncells<0>(domain);
```

Since the left hand side operand already contains the topological dimension of elements over which to iterate, the template argument to the `ncells` function can also be omitted:

```
ConstVertexRange vertices = viennagrid::ncells(domain);
```

While the advantage of this shorter variant is clearly shorter code and simpler copy&pasting, the disadvantage is that the topological dimension is specified only once in the respective **typedef**. The longer variant adds a second check for the use of the correct topological dimension.

Once the range is set up, iteration is carried out in the usual C++ STL manner:

```
for (NCellIterator it = elements.begin();
                   it != elements.end();
                 ++it)
{ /* do something */}
```

For reference, the complete code for printing all vertices of a domain without a **using namespace**-directive is:

```
typedef viennagrid::result_of::const_ncell_range<DomainType, 0>::type
    ConstVertexRange;
typedef viennagrid::result_of::iterator<ConstVertexRange>::type
    ConstVertexIterator;

ConstVertexRange vertices = viennagrid::ncells(domain);
for (VertexIterator vit = vertices.begin();
                    vit != vertices.end();
                  ++vit)
{  std::cout << *vit << std::endl; }
```

It should be emphasized that this code snippet is valid for arbitrary geometric dimensions and arbitrary domain configurations (and thus cell types). Inside a template function or template class, the **typename** keyword needs to be added after each **typedef**.

In some cases, e.g. for a parallelization using OpenMP[16], it is preferred to iterate over all cells using an index-based for-loop rather than an iterator-based one. If the range is either a vertex range of a domain, or a cell range of a domain or segment, this can be obtained by

```
NCellRange elements = viennagrid::ncells<n>(domain);
for (std::size_t i=0; i<elements.size(); ++i)
{
  do_something(elements[i]);
}
```

It is also possible to use the range only implicitly:

```
for (std::size_t i=0; i<viennagrid::ncells<n>(domain).size(); ++i)
{
  do_something(viennagrid::ncells<n>(domain)[i]);
}
```

where n has to be replaced with the topological dimension of vertices and cells respectively. However, since the repeated construction of the range object can have non-negligible setup costs, the latter code is not recommended.

> In `ViennaGrid 1.0.0`, `operator[]` is not available for ranges obtained from a domain other than vertex or cell ranges. For segments, `operator[]` is only available for cell ranges. ⚠

5.2 Boundary k-Cells of n-Cells

In addition to an iteration over all n-cells of a domain or segment, it may be required to iterate over boundary k-cells, $k < n$ of each n-cell. Instead of a general description using n-cells and k-cells, the more descriptive case of an iteration over all edges ($k = 1$) of a `cell` of type `CellType` will be considered.

> In `ViennaGrid 1.0.0`, an iteration over all k-cells of an n-cell is not possible if the storage of boundary k-cells is disabled for n-cells, cf. Section 3.2.1. This restriction is expected to be relaxed in future versions of `ViennaGrid`. ⚠

As in the previous section, the range and iterator types are obtained from the `ncell_range` and `iterator` meta-functions:

```
//non-const:
typedef viennagrid::result_of::ncell_range<CellType, 1>::type
    EdgeOnCellRange;
typedef result_of::iterator<EdgeOnCellRange>::type
    EdgeOnCellIterator;
```

The `const`-version is again obtained by using `const_ncell_range` instead of `ncell_range`. Mind that the first argument of `ncell_range` denotes the enclosing entity (the cell) and the second argument denotes the topological dimension (1 for an edge), and thus preserves the structure already used for the type retrieval for iterations on the domain.

Iteration is then carried out in the same manner as for a domain, with `cell` taking the role of the `domain` in the previous chapter:

```
//Note: ... = viennagrid::ncells(cell); will also work in the next line
EdgeOnCellRange edges_on_cell = viennagrid::ncells<1>(cell);
```

```
    for (EdgeOnCellIterator eocit = edges_on_cell.begin();
                            eocit != edges_on_cell.end();
                          ++eocit)
{  std::cout << *eocit << std::endl; }
```

will print all edges of the `cell`.

For all topological dimensions, an index-based iteration is possible provided that the storage of the respective boundary k-cells has not been disabled. The previous code snippet can thus also be written as

```
EdgeOnCellRange edges_on_cell = viennagrid::ncells<1>(cell);
for (std::size_t i=0; i<edges_on_cell.size(); ++i)
{
   do_something(edges_on_cell[i]);
}
```

or

```
for (std::size_t i=0; i<viennagrid::ncells<1>(cell).size(); ++i)
{
   std::cout << viennagrid::ncells<n>(domain)[i] << std::endl;
}
```

The use of the latter is again discouraged for reasons of possible non-negligible repeated setup costs of the ranges involved.

> In `ViennaGrid 1.0.0`, an iteration over all k-cells of an n-cell is not possible if the storage of boundary k-cells is disabled for n-cells, cf. Section 3.2.1. This restriction is expected to be relaxed in future versions of `ViennaGrid`. ⚠

Finally, `ViennaGrid` allows for iterations over the vertices of boundary k-cells of an n-cell in the reference orientation imposed by the n-cell, which is commonly required for ensuring continuity of a quantity along cell interfaces. Note that by default the iteration is carried out along the orientation imposed by the k-cell in the way it is stored globally inside the domain. The correct orientation of vertices with respect to the hosting n-cell is established by the free function `local_vertex()`. For instance, the vertices of a k-cell `cell_k` at the boundary of a n-cell `cell_n` are printed in local orientation using the code lines

```
for (std::size_t i=0; i<viennagrid::ncells<0>(cell_k).size(); ++i)
   std::cout << viennagrid::local_vertex(cell_n, cell_k, i) << std::endl;
```

The use of `local_vertex` can be read as follows: For the n-cell `cell_n`, return the vertex of the boundary k-cell `cell_k` at local position i.

5.3 Coboundary k-Cells of n-Cells

A frequent demand of mesh-based algorithms is to iterate over so-called *coboundary k-cells* of an n-cell T_n, where $k > n$. The coboundary k-cells of an n-cell T_n are given by all k-cells of a set Ω, where one of the boundary n-cells is T_n. For example, the coboundary edges (1-cells) of a vertex (0-cell) T_0 are all edges where one of the two vertices is T_0.

In contrast to boundary k-cells , the number of coboundary k-cells of an n-cell from the family of simplices or hypercubes is not known at compile time. Another difference to the case of boundary k-cells is that the number of coboundary k-cells depends on the set Ω under consideration. Considering the interface edge/facet connecting vertices 1 and 4 in the sample domain from Fig. 4.1, the coboundary 2-cells within the domain are given by the triangles 1 and 2. However, within segment 0, the set of coboundary 2-cells is given by the triangle 1 only, while within segment 1 the set of coboundary 2-cells consists of triangle 2 only. Thus, the use of segments can substantially simplify the formulation of algorithms that act on a subregion of the domain only.

The necessary range types are obtained using the same pattern as in the two previous sections. Assuming that a vertex type `VertexType` is already defined, the range of coboundary edges as well as the iterator are obtained using the `ncell_range` and `iterator` metafunctions in the `viennagrid` namespace:

```
//non-const:
typedef result_of::ncell_range<VertexType, 1>::type  EdgeOnVertexRange;
typedef result_of::iterator<EdgeOnVertexRange>::type EdgeOnVertexIterator;
```

Again, the first argument to `ncell_range` is the reference element for the iteration, and the second argument is the topological dimension of the elements in the range. A range of **const**-edges is obtained using the `const_ncell_range` metafunction instead of the non-**const** metafunction `ncell_range`. Moreover, it shall be noted that an additional **typename** keyword is required inside template functions and template classes.

To set up the range object, the `ncells` function from the `viennagrid` namespace is reused. Unlike in the previous sections, it requires two arguments for setting up a coboundary range: The first argument is the reference element, and the second argument refers to the enclosing container of k-cells and must be either a domain or a segment. The range holding all edges in the `domain` sharing a common vertex `v` is thus set up as

```
EdgeOnVertexRange edges_on_v = viennagrid::ncells<1>(v, domain);
```

where the template parameter 1 is again optional. If the range should hold only the coboundary edges from a segment `seg`, the above code line has to be modified to

```
EdgeOnVertexRange edges_on_v = viennagrid::ncells<1>(v, seg);
```

An iteration over all edges is then possible in the usual STL-type manner. For example, all coboundary edges of v in the range are printed using the code:

```
for (EdgeOnVertexIterator eovit = edges_on_v.begin();
                          eovit != edges_on_v.end();
                        ++eovit)
{
  std::cout << *eovit << std::endl;
}
```

One may also use a shorter form that does not set up the range explicitly:

```
for (EdgeOnVertexIterator eovit = viennagrid::ncells<1>(v,domain).begin();
                          eovit != viennagrid::ncells<1>(v,domain).end();
                        ++eovit)
{
  std::cout << *eovit << std::endl;
}
```

Here, the template parameter 1 for the `ncells` function must not be omitted.

Random access, i.e. **operator**`[]` is available for all topological levels. Thus, the loop above may also be written as

```cpp
for (std::size_t i=0; i<edges_on_v.size(); ++i)
{
  std::cout << edges_on_v[i] << std::endl;
}
```

or

```cpp
for (std::size_t i=0; i<viennagrid::ncells<1>(v, domain).size(); ++i)
{
  std::cout << viennagrid::ncells<1>(v, domain)[i] << std::endl;
}
```

where the latter form is not recommended for reasons of overheads involved in setting up the tempoarary ranges.

Finally, it should be noted that coboundary information is not natively available in the domain datastructure. If and only if for the first time the coboundary k-cells of an n-cell, $k > n$, are requested, an iteration over all k-cells of the domain or segment with nested n-cell boundary iteration is carried out to collect the topological information. This results in extra memory requirements and additional computational costs, hence we suggest to use boundary iterations over coboundary iterations whenever possible.

Prefer the use of boundary iterations ($k < n$) over coboundary iterations ($k > n$) to minimize memory footprint.

Chapter 6

Data Storage and Retrieval

One of the central operations whenever dealing with meshes is the storage and the retrieval of data. A common approach is to model vertices, edges and the like as separate classes and add data members to them. `ViennaGrid` does not follow this approach for three reasons:

1. **Reusability**: As soon as a data member is added to any of these classes, the class is refined towards a particular use case. For example, adding a color data member to a triangle class reduces reusability for e.g. Finite Element methods considerably.

2. **Flexibility**: Whenever a data member needs to be added for a particular functionality, one has to carefully extend the existing class layout. Moreover, it is somewhere between hard to impossible to 'just add a data member for the moment' in a productive environment. Moreover, the class needs to be adjusted if the data type changes.

3. **Efficiency**: A data member that is never used obviously wastes memory. For large numbers of object it might be even advisable to use special containers for data that is relevant for a tiny fraction of all objects only (e.g. domain boundary flags). Apart from reduced memory footprint, the possibly tighter grouping of data allows for better CPU caching.

`ViennaGrid` relies on `ViennaData` [1] for the storage of data associated with topological objects. For full details on the capabilities of `ViennaData`, please refer to the `ViennaData` manual. For the sake of completeness, a brief overview of the basic usage of `ViennaData` for `ViennaGrid` is given in the following.

A tutorial code can be found in `examples/tutorial/store_access_data.cpp`.

6.1   Handling Data

`ViennaData` can be seen in a simplified view as something that behaves like a `std::map <KeyType, DataType>` for each object for which data should be stored or retrieved. For a `key` object of type `KeyType`, `data` of type `DataType` is stored on an object `obj` as

```
viennadata::access<KeyType, DataType>(key)(obj) = data;
```

At some later stage, the data can be conveniently retrieved from the object as

```
DataType my_data = viennadata::access<KeyType, DataType>(key)(obj);
```

One may also erase data stored for the object using

```
viennadata::erase<KeyType, DataType>(key)(obj);
```

6.2 Customizations of the Internal Storage Scheme

It can readily be seen that the free function `access()` function mimics the template arguments of a `std::map<KeyType, DataType>` and takes the key as argument. In fact, the default internal storage scheme of `ViennaData` for objects of type `ObjectType` is

```
std::map<ObjectType const *,
         std::map<KeyType, DataType> >
```

which is generated for each triple of `KeyType`, `DataType` and `ObjectType` by the compiler. Since the tree-based structure is well suited for data that is stored for a small fraction of the total number of objects of the same type, we call it a *sparse* data storage scheme. However, such a nested map is not a good datastructure in many cases and can thus be customized in a rather transparent manner.

For objects that provide integral IDs ranging from 0 to some number N, a storage scheme of the form

```
std::vector< std::map<KeyType, DataType> >
```

for each `ObjectType` is more appropriate. Since memory for all objects of the same type is used then, we refer to this scheme as a *dense* data storage scheme. It can be enabled by first using one of the macros (mind the VIENNAGRID prefix!)

```
VIENNAGRID_ENABLE_NCELL_ID_FOR_DATA(A, B)       //config A, n-cell-tag B
VIENNAGRID_ENABLE_ALL_NCELL_ID_FOR_DATA(A)      //all n-cells with config A
VIENNAGRID_GLOBAL_ENABLE_NCELL_ID_FOR_DATA(B)   //all configs, n-cell-tag B
VIENNAGRID_GLOBAL_ENABLE_ALL_NCELL_ID_FOR_DATA() //all configs, all n-cells
```

with `A` and `B` denoting the configuration class and the n-cell tag respectively. This configures `ViennaData` to use the integral ID of the element. Second, dense storage can then be selectively enabled using one of the macros

```
VIENNADATA_ENABLE_DENSE_DATA_STORAGE_FOR_*
```

where the star refers to `KEY`, `DATA`, `OBJECT` or a mixture of these. For details consult the `ViennaData` manual.

> In order to use a dense storage scheme, the storage of IDs for the particular n-cells must not be disabled, cf. Section 3.2.3! ⚠

Moreover, the `std::map<KeyType, DataType>` type may be too costly due to key comparisons. For example, it is immediately clear that a repeated comparison of the two strings `"rather_long_string_1"` and `"rather_long_string_2"` is not suitable for high-performance environments. In such cases, `ViennaData` can be configured to dispatch based on the key type only, but not based on runtime information. Using the macro

```
VIENNADATA_ENABLE_TYPE_BASED_KEY_DISPATCH(my_key)
```

configures `ViennaData` to use a type-based dispatch for a key class type `my_key`. In this case, the internal data container of `ViennaData` changes to either

```
std::map<ObjectIDType, DataType>
```

with `ObjectIDType` being either a pointer or an integral ID, or using a key type based dispatch to

```
std::vector< DataType >
```

Therefore, `ViennaData` can be configured with a high granularity in order to accomodate for the individual data access patterns and data densities.

> For additional features such as default data types as well as `move()` and `copy()` functions, consult the `ViennaData` manual.

Chapter 7

Algorithms

The iterations and data accessors described in the previous Chapters allow for a plethora of algorithms. Most of them make use of basic functionalities such as the inner product of vectors, or the volume of a n-cell. `ViennaGrid` ships with a number of such basic tools, which are listed in Tab. 7.1 and discussed in the following.

The individual algorithms are located in the `viennagrid/algorithm/` folder. A tutorial covering the algorithms explained in this chapter can be found in `examples/tutorial/algorithms.cpp`.

> Make sure to include the respective header-file when using one of the algorithms explained below!

7.1 Point/Vector-Based

This section details algorithms in `ViennaGrid` requiring geometric information only. The point type in `ViennaGrid` should be seen in this context mostly as a vector type rather than a representation of a geometric location, reflecting the duality of points and vectors in the Euclidian space.

7.1.1 Cross Products

The cross-product of two vectors (i.e. `ViennaGrid` points) $p0$ and $p1$ is defined for the three-dimensional space and computed with `ViennaGrid` as

```
viennagrid::cross_prod(p1, p2)
```

The following code calculates and prints the cross-product of the vectors $(1, 0, 0)^{\mathrm{T}}$ and $(0, 1, 0)^{\mathrm{T}}$:

```
PointType p0(1, 0, 0);
PointType p1(0, 1, 0);
std::cout << viennagrid::cross_prod(p1, p2) << std::endl; //0 0 1
```

If the two vectors are given in different coordinate systems, the result vector will have the same coordinate system as the first argument.

Algorithm	Filename	Interface Function
Cross Product	`cross_prod.hpp`	`cross_prod(a, b)`
Inner Product	`inner_prod.hpp`	`inner_prod(a, b)`
Vector Norms	`norm.hpp`	`norm(a, tag)`
Induced Volume	`spanned_volume.hpp`	`spanned_volume(a, b, ...)`
Centroid computation	`centroid.hpp`	`centroid(ncell)`
Circumcenter comp.	`circumcenter.hpp`	`circumcenter(ncell)`
Surface computation	`surface.hpp`	`surface(ncell)`
Volume computation	`volume.hpp`	`volume(ncell)`
Boundary detection	`boundary.hpp`	`is_boundary(ncell, domseg)`
Interface detection	`interface.hpp`	`is_interface(ncell, seg1, seg2)`
Simplex refinement	`refine.hpp`	`refine(domain, tag)`
Surface computation	`surface.hpp`	`surface(domseg)`
Volume computation	`volume.hpp`	`volume(domseg)`
Voronoi grid	`voronoi.hpp`	`apply_voronoi(domseg, ...)`

Table 7.1: List of algorithms available in `viennagrid/algorithm/` grouped by the objects they are acting on. `a` and `b` denote vectors, `ncell` refers to a n-cell, `domain` to a domain, `seg1` and `seg2` to segments, and `domseg` to either a domain or a segment.

7.1.2 Inner Products

Unlike cross products, inner products (aka. dot products) are well defined for arbitrary dimensions. In `ViennaGrid 1.0.0` an inner product of the form

$$(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=0}^{N-1} x_i y_i \tag{7.1}$$

is available with the function `inner_prod()`. The following code calculates and prints the inner product of the vectors $(1,0,0)^{\mathrm{T}}$ and $(0,1,0)^{\mathrm{T}}$:

```
PointType p0(1, 0, 0);
PointType p1(0, 1, 0);
std::cout << viennagrid::inner_prod(p1, p2) << std::endl; //0
```

If the two vectors are given in different coordinate systems, the result vector will have the same coordinate system as the first argument.

7.1.3 Vector Norms

Currently, p-norms of the form

$$\|x\|_p = \sqrt[p]{\sum_{i=0}^{N-1} x_i^p} \tag{7.2}$$

are implemented in the N-dimensional Euclidian space for $p = 1$, $p = 2$ and $p = \infty$. The three norms for the vector $(1,2,3)^{\mathrm{T}}$ are computed and printed using the lines

```
  PointType p(1, 2, 3);
  std::cout << viennagrid::norm_1(p)   << std::endl; //6
  std::cout << viennagrid::norm_2(p)   << std::endl; //3.74
  std::cout << viennagrid::norm_inf(p) << std::endl; //3
```

which are equivalent to

```
  PointType p(1, 2, 3);
  std::cout << viennagrid::norm(p, viennagrid::one_tag()) << std::endl;
  std::cout << viennagrid::norm(p, viennagrid::two_tag()) << std::endl;
  std::cout << viennagrid::norm(p, viennagrid::inf_tag()) << std::endl;
```

7.1.4 Volume of a Spanned Simplex

It is often handy to compute the n-dimensional volume of a n-simplex embedded in a possibly higher-dimensional geometric space by providing the locations of the vertices only. This is provided by `spanned_volume()`, which is, however, currently limited to $n \in \{1, 2, 3\}$. As an example, the two-dimensional volume a triangle with corners at $(1, 0, 0)$, $(2, 1, 1)$ and $(1, 1, 2)$ is computed and printed by

```
  PointType p0(1, 0, 0);
  PointType p1(2, 1, 1);
  PointType p2(1, 1, 2);
  std::cout << viennagrid::spanned_volume(p0, p1, p2) << std::endl;
```

7.2 n-Cell-Based

In this section, algorithms defined for geometric objects with additional structure are discussed. Additional algorithms are likely to be introduced in future releases.

7.2.1 Centroid

The centroid of a n-cell object `cell_n` in Cartesian coordinates is obtained as

```
  PointType p = viennagrid::centroid(cell_n);
```

and works for arbitrary topological and geometrical dimensions.

7.2.2 Circumcenter

The circumcenter of a n-simplex `cell_n` is obtained in Cartesian coordinates as

```
  PointType p = viennagrid::circumcenter(cell_n);
```

The computation is restricted to values $n \leq 3$. For reasons of uniformity, also n-hypercubes can be passed, for which the circumcenter of an embedded n-simplex is computed. This leads to valid results and makes sense only for certain regular hypercubes. Thus, the user

has to ensure that the n-hypercube actually has a circum-n-sphere. This is e.g. the case for structured tensor-grids.

> There is no warning or error issued if a n-hypercube passed to `circumcenter()` does not have a circumcenter. ⚠

7.2.3 Surface

The surface of a n-cell `cell_n` is defined as the sum of the volumes of its $n-1$ boundary cells. Therefore, in order to make the calling code

```
NumericType surf = viennagrid::surface(cell_n);
```

valid, the storage of $n-1$-cells at the boundary of n-cells must not be disabled, cf. Section 3.2.1. Currently, `surface()` is restricted to n-cells with $n \leq 4$.

> In order to use `surface()` for a n-cell T_n, make sure that boundary $n-1$-cells are not disabled, cf. Section 3.2.1. ⚠

7.2.4 Volume

The n-dimensional volume of a n-cell `cell_n` is returned by

```
NumericType vol = viennagrid::volume(cell_n);
```

and currently restricted to $n \leq 3$. No restrictions with respect to the storage of boundary k-cells of the n-cell apply.

7.3 Domain/Segment-Based

Algorithms acting on a collection of cells are now considered. These collections are given in `ViennaGrid` either as the whole domain, or as segments.

7.3.1 Boundary

Whether or not a n-cell object `ncell` is located on the boundary depends on the collection of elements considered. For example, consider the edge/facet $[1, 4]$ in the triangular sample mesh in Fig. 4.1, which we will refer to as `f`. It is in the interior of the whole `domain`, while it is also at the boundary of the two segments `seg0` and `seg1`. A sample code snippet reflecting this is given by

```
std::cout << viennagrid::is_boundary(f, domain) << std::endl; //false;
std::cout << viennagrid::is_boundary(f, seg0)   << std::endl; //true;
std::cout << viennagrid::is_boundary(f, seg1)   << std::endl; //true;
```

Note that `is_boundary()` induces some additional setup costs at the first time the function is called. However, subsequent calls are accelerated and will usually compensate for the setup costs.

> `is_boundary` requires that facets are stored within cells. Therefore, make sure not to disable the handling of facets, cf. Section 3.2.1. ⚠

7.3.2 Interface

Similar to the detection of boundary facets, n-cells on the interface between two segments are frequently of particular interest. A n-cell `cell_n` can be checked for being on the interface of two segments `seg0` and `seg1` using

```
std::cout << viennagrid::is_interface(cell_n, seg0, seg1) << std::endl;
```

Note that `is_interface()` induces some setup costs the first time it is called for a pair of segments.

> `is_interface()` requires that facets are stored within cells. Therefore, make sure not to disable the handling of facets, cf. Section 3.2.1. ⚠

7.3.3 Refinement

`ViennaGrid 1.0.0` allows a uniform and a local refinement of simplicial domains. The refinement of hypercuboidal domains is scheduled for future releases. It has to be noted that the resulting refined mesh is written to a new domain, thus there are no multigrid/-multilevel capabilities provided yet.

To refine a `domain` uniformly, the line

```
DomainType refined_domain
   = viennagrid::refine(domain, viennagrid::uniform_refinement_tag());
```

or, equivalently,

```
DomainType refined_domain = viennagrid::refine_uniformly(domain);
```

is sufficient. Note that the refinement algorithms in `ViennaGrid` act on domains, not on individual segments. Segment information is preserved upon refinement. It has to be emphasized that no expensive temporary domain is created for the refinement process thanks to the use of a proxy object.

The local refinement of a mesh requires that the respective cells or edges for refinement are tagged. This refinement information is applied to the domain using `ViennaData`, cf. Chapter 6. To tag a cell `c` for refinement, the line

```
viennadata::access<viennagrid::refinement_key, bool>()(c) = true;
```

is sufficient. In a similar way one proceeds for other cells or edges in the domain. The refinement process is then triggered by

```
DomainType refined_domain
   = viennagrid::refine(domain, viennagrid::local_refinement_tag());
```

or, equivalently,

```
DomainType refined_domain = viennagrid::refine_locally(domain);
```

Again, no expensive temporary domain is created for the refinement process thanks to the use of a proxy object.

> `refine()` requires edges to be stored in the domain. Make sure not to disable the handling of edges, cf. Section 3.2.1. ⚠️

7.3.4 Surface

The surface of a domain or segment `domseg` is given by the sum of the volumes of the boundary facets and returned by the convenience overload

```
NumericType surf = viennagrid::surface(domseg);
```

Note that facets interfacing cells outside the segment are also considered as boundary facets of the segment.

> In order to use `surface()` for a domain or a segment, make sure that boundary $n-1$-cells are not disabled, cf. Section 3.2.1. ⚠️

7.3.5 Volume

The volume of a domain or segment `domseg` is returned by

```
NumericType vol = viennagrid::volume(domseg);
```

and currently restricted to maximum topological dimension $n \leq 3$.

7.3.6 Voronoi Information

A Voronoi diagram of a Delaunay tessellation (or triangulation) is a decomposition of the domain into certain boxes containing one vertex each. The boxes have the property that all points inside the box are closer to the vertex within the box than to any other vertex in the domain. By simple geometric arguments one finds that the corners of Voronoi boxes are given by the circumcenters of the triangles.

The function `apply_voronoi()` computes the volumes and interfaces associated with a Voronoi diagram. The following values are stored on the domain (cf. Fig. 7.1)

- The volume $V_{[i,j]}$ of the polyhedron centered around the edge $[i,j]$ with edges given by the connections of the vertices i and j with the circumcenters of the coboundary cells of the edge is stored on the edge $[i,j]$.
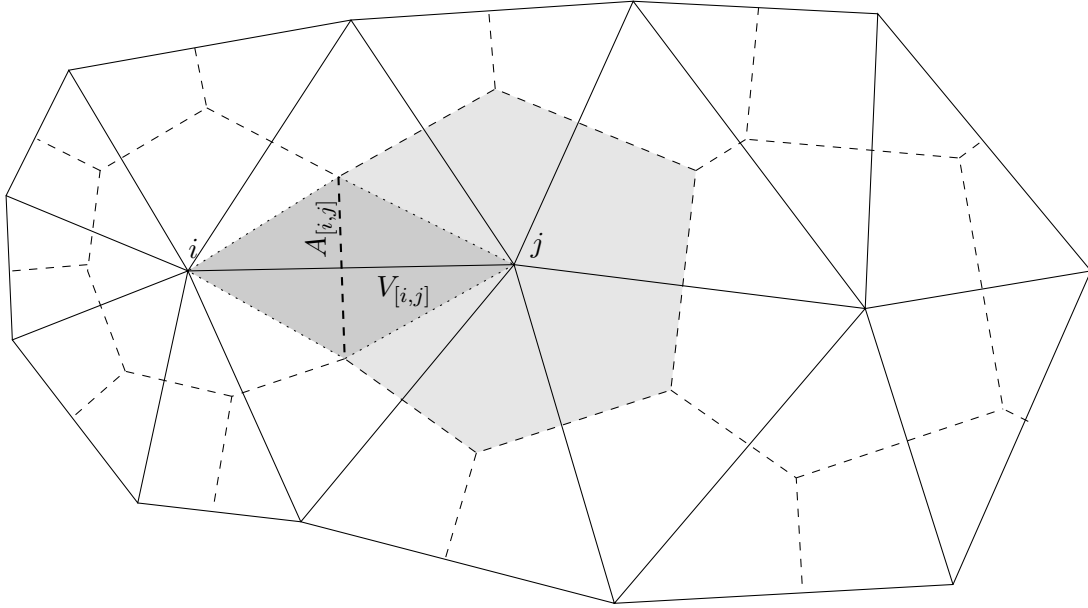
Figure 7.1: Schematic of a Delaunay mesh with its dual Voronoi diagram, where the box containing vertex j is highlighted. The function `voronoi()` computes and stores the Voronoi volume $V_{[i,j]}$ and the interface area $A_{[i,j]}$ associated with each edge $[i,j]$. The total box volume associated with each vertex is also stored on the vertex.

- The interface area $A_{[i,j]}$ of the boxes for the vertices i and j on the edge $[i,j]$.
- The box volume V_i of the box containing vertex i for each vertex i.

Using the lines

```
viennagrid::apply_voronoi(domain);
```

the **double**-typed quantities can be accessed using `ViennaData` (**cf.** Chapter 6) using the type-dispatched keys `voronoi_interface_area_key` for the interface area on edges, and `voronoi_box_volume_key` for accessing $V_{[i,j]}$ on edges and the box volume V_i on vertices. It is also possible to pass custom keys for storing the interface areas and volumes by supplying a second and third parameter to `apply_voronoi()`:

```
std::string interface_area_key = "voronoi_interface";
std::string box_volume_key = "voronoi_volume";
viennagrid::apply_voronoi(domain, interface_area_key, box_volume_key);
```

This uses keys of type string with values `"voronoi_interface"` and `"voronoi_volume"`, which can then be used to access the Voronoi data as

```
viennadata::access<std::string, double>("voronoi_interface")(edge); //A_ij
viennadata::access<std::string, double>("voronoi_volume")(edge);    //V_ij
viennadata::access<std::string, double>("voronoi_volume")(vertex);  //V_i
```

Voronoi information can also be computed for segments only.

> ⚠ Be aware the values on interface edges and vertices will be overwritten when using the same keys for interfacing segments!

Chapter 8

Input/Output

This chapter deals with the typical input and output operations: Reading a mesh from a file, and writing a mesh to a file. In order not to give birth to another mesh file format, `ViennaGrid` does not bring its own file format. Instead, the library mainly relies on the XML-based VTK [17] file format [18].

> A tutorial code can be found in `examples/tutorial/io.cpp`.

> Let us know about your favorite file format(s)! Send an email to our mailinglist: `viennagrid-support@lists.sourceforge.net`. It increases the chances of having a suitable reader and/or writer included in the next `ViennaGrid` release.

8.1 Readers

Due to the high number of vertices and cells in typical meshes, a manual setup of a domain in code is soon inefficient. Therefore, the geometric locations as well as topological connections are typically stored in mesh files.

Currently, `ViennaGrid` supports only two file formats natively. However, readers for other file formats can be easily added by the user when following the explanations for domain setup in Chapter 4. A different approach is to convert other file formats to one of the formats detailed in the following.

8.1.1 Netgen

The `.mesh`-files provided with `Netgen` [19] can be imported directly. These files are obtained from `Netgen` from the `File->Export Mesh...` menu item. Note that only triangular and tetrahedral meshes are supported.

To read a mesh from a `.mesh` file with name `filename` to a `domain`, the lines

```
viennagrid::io::netgen_reader my_netgen_reader;
my_netgen_reader(domain, filename);
```

should be used. Note that the reader might throw an `std::exception` if the file cannot be opened or if there is a parser error.

The fileformat is simplistic: The first number refers to the number of vertices in the domain, then the coordinates of the vertices follow. After that, the number of cells in the domain is specified. Then, each cell is specified by the index of the segment and the indices of its vertices, each using index base 1. For example, the `.mesh`-file for the sample domain in Fig. 4.1 is:

```
6
0 0
1 0
2 0
2 1
1 1
0 1
4
1 1 2 6
1 2 5 6
2 2 3 5
2 3 4 5
```

8.1.2 VTK

The VTK file format is extensively documented [18] and allows to store mesh quantities as well. The simplest way of reading a VTK file `my_mesh.vtu` is similar to the `Netgen` reader:

```
viennagrid::io::vtk_reader<DomainType>  my_vtk_reader;
my_vtk_reader(domain, "my_mesh.vtu");
```

Note that the domain type is required as template argument for the reader class.

`ViennaGrid` supports single-segmented `.vtu` files consisting of one `<piece>`. and always reads a single-segmented mesh to its first segment. If a segment does not already exist on the domain, one is created.

For multi-segment meshes, the Paraview [20] data file format `.pvd` can be used, which is a XML wrapper holding information about the individual segments in `.vtu` files only. Vertices that show up in multiple segments are automatically fused. Example meshes can be found in `examples/data/`.

The VTK format allows to store scalar-valued and vector-valued data sets, which are identified by their names, on vertices and cells. These data sets are directly transferred to the `ViennaGrid` domain using `ViennaData` as described in Chapter 6. By default, data is stored using the data name string as key of type `std::string`. Scalar-valued data is stored as **double**, while vector-valued data is stored as `std::vector<`**double**`>`. For example, a scalar quantity `potential` on a vertex `v` can be accessed and further manipulated using

```
viennadata::access<std::string, double>("potential")(v)
```

For efficiency reasons, one may want to have data stored using either a different key, key type or data type. This can be achieved using the following free functions in Tab. 8.1 to set

Function name	Data description
add_scalar_data_on_vertices	scalar-valued, vertex-based
add_vector_data_on_vertices	vector-valued, vertex-based
add_normal_data_on_vertices	vector-valued, vertex-based
add_scalar_data_on_vertices	scalar-valued, cell-based
add_vector_data_on_vertices	vector-valued, cell-based
add_normal_data_on_vertices	vector-valued, cell-based

Table 8.1: Free functions in namespace `viennagrid::io` for customizing reader and writer objects. Note that `normal`-data refers to `vector`-data that is normalized to unit length. The three parameters to each of these functions is the reader object, the `ViennaData` key and the VTK data name.

up the reader object accordingly.

A list of all names identifying data read from the file can be obtained the functions

Function name	Data description
get_scalar_data_on_vertices	scalar-valued, vertex-based
get_vector_data_on_vertices	vector-valued, vertex-based
get_scalar_data_on_vertices	scalar-valued, cell-based
get_vector_data_on_vertices	vector-valued, cell-based

For example, to store scalar-valued vertex-data with name `potential` in a VTK file with name `"my_mesh.vtu"` to the domain using a key `42` of type **long**, and process all other data in the default way, the reader has to be invoked as:

```
viennagrid::io::vtk_reader<DomainType>      my_vtk_reader;
viennagrid::io::add_scalar_data_on_vertices<long,          // key type
                                            double>        // data type
                                           (my_vtk_reader, // reader
                                            42,            // key
                                            "potential");  // VTK name
my_vtk_reader(domain, "my_mesh.vtu");
```

The list of all scalar vertex data sets read is then printed together with the respective segment index as

```
using namespace viennagrid::io;
for (size_t i=0; i<get_scalar_data_on_vertices(reader).size(); ++i)
  std::cout << "Segment " << get_scalar_data_on_vertices(reader)[i].first
            << ": "
            << get_scalar_data_on_vertices(reader)[i].second << std::endl;
```

8.2 Writers

Since `ViennaGrid` does not provide any visualization capabilities, the recommended procedure for visualization is to write to one of the file formats discussed below and use one of

the free visualization suites for that purpose.

> File writer contributions are always welcome! 💡

8.2.1 OpenDX

`OpenDX` [21] is an open source visualization software package based on IBM's Visualization Data Explorer. The writer supports either one vertex-based or one cell-based scalar quantity to be written to an `OpenDX` file.

The simplest way to write a `domain` of type `DomainType` to a file `"my_mesh.out"` is

```
viennagrid::io::opendx_writer<DomainType> my_dx_writer;
my_dx_writer(domain, "my_mesh.out");
```

To write quantities stored on the domain, the free functions from Tab. 8.1 are used. For example, to visualize a scalar vertex-quantity of type **double** stored on the domain using a key `"potential"` of type `std::string` (cf. Chapter 6), the previous snippet is modified to

```
using viennagrid::io;
opendx_writer<DomainType> my_dx_writer;
add_scalar_data_on_vertices<std::string,  // key type
                            double>        // data type
                           (my_dx_writer, // writer
                            "potential",   // key
                            "some_name");  // ignored
my_dx_writer(domain, "my_mesh.out");
```

Note that the data name provided as third argument is ignored for the `OpenDX` writer.

> `ViennaGrid` **can write only one scalar quantity to an** `OpenDX` **file!** ⚠️

8.2.2 VTK

A number of free visualization tools such as Paraview [20] are available for the visualization of VTK files. The simplest way of writing to a VTK file is

```
viennagrid::io::vtk_writer<DomainType> my_vtk_writer;
my_vtk_writer(domain, "outfile");
```

If the domain does not contain segments, the file `"outfile.vtu"` is written. If there are segments in the domain, each segment is written to a separate file, leading to `"outfile_0.vtu"`, `"outfile_1.vtu"`, etc. In addition, a Paraview data file `"outfile_main.pvd"` is written, which links all the segments and should thus be used for visualization.

To write quantities stored on the domain to the VTK file(s), the free functions from Tab. 8.1 are used. For example, to visualize a vector-valued cell-quantity of type `std::vector<` **double**`>` stored on the domain using a key `"stress"` of type `std::string` (cf. Chapter 6), the previous snippet is modified to

```
using viennagrid::io;
vtk_writer<DomainType> my_vtk_writer;
```

```
add_scalar_data_on_vertices<std::string,          // key type
                            std::vector<double> > // data type
                           (my_vtk_writer,        // writer
                            "stress",             // key
                            "vtk_name");          // VTK name
  my_vtk_writer(domain, "outfile");
```

In this way, the quantity is written to all segments and values at the interface coincide.

If discontinuities at the interfaces should be allowed, vertex data may also be written per segment. This can be achieved using the free functions

Function name	Data description
add_scalar_data_on_vertices_per_segment	scalar-valued, vertex-based
add_vector_data_on_vertices_per_segment	vector-valued, vertex-based
add_normal_data_on_vertices_per_segment	vector-valued, vertex-based

in namespace viennagrid::io, which work in the same way as the free functions in Tab. 8.1. The data type needs to provide **operator**[], where the argument is the segment index.

As closing example, a quantity "jump" with type **double** is stored on the vertices using the ViennaData data type std::map<std::size_t, **double**>. For a vertex v at the interface of segments with indices 0 and 1, the values 1.0 for the segment with index 1 and 2.0 for the segment with index 2 are stored:

```
typedef std::map<std::size_t, double>  DataType;
viennadata::access<std::string, DataType>("jump")(v)[0] = 1.0;   // seg0
viennadata::access<std::string, DataType>("jump")(v)[1] = 2.0;   // seg1
```

The quantity is added to the VTK writer as

```
add_scalar_data_on_vertices_per_segment<std::string, DataType>
                                  (my_vtk_writer, "jump", "segment_data");
```

A tutorial code using a VTK writer for discontinuous data at segment boundaries can be found in examples/tutorial/multi-segment.cpp.

Chapter 9

Library Internals

Details about the internals of `ViennaGrid` will be given in the following. They should aid developers to extend the library with additional features and to understand the internal data structures used. Nevertheless, the information provided might be of interest for new users of `ViennaGrid` as well.

9.1 Recursive Inheritance

`ViennaGrid` extensively relies on recursive inheritance to build the individual n-cell types. The n-cells are of type `element_t<ConfigType, ElementTag>`, where `ConfigType` is the configuration class, and `ElementTag` is a tag identifying the topological shape of the element. The class `element_t` itself is almost empty, but inherits the information about its boundary k-cells from a `boundary_ncell_layer`. In addition, the class inherits from a class with the sole purpose of providing an ID mechanism

```
template <typename ConfigType,
          typename ElementTag>
class element_t :
    public boundary_ncell_layer<ConfigType, ElementTag, ElementTag::dim-1>,
    public result_of::element_id_handler<ConfigType, ElementTag>::type
{ ... }
```

The third template parameter of `boundary_ncell_layer` is crucial here, because it refers to the topological dimension of the boundary cells it is responsible for. The `boundary_ncell_layer` at level k then inherits from a `boundary_ncell_layer` for level $k-1$

```
template <typename ConfigType, typename ElementTag, long dim>
class boundary_ncell_layer <ConfigType, ElementTag, dim> :
    public boundary_ncell_layer<ConfigType, ElementTag, dim-1>
```

where the additional technical template arguments customizing the behavior of each boundary layer are omitted. The recursion is terminated at vertex level by a partial template specialization of the class.

An illustration of the resulting class layout is given in Fig. 9.1. Since each layer is configured by respective metafunctions that return the specified user settings, the class size of each layer varies depending on the storage of boundary cells and on the use of local reference orientations.
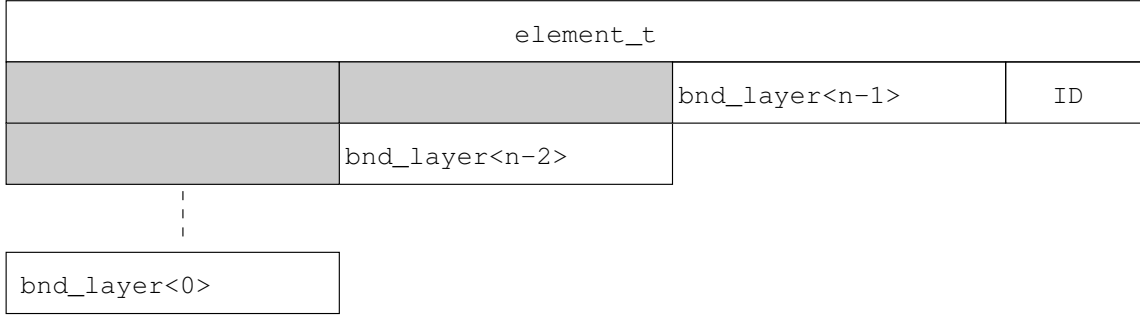
element_t			
		bnd_layer<n-1>	ID
	bnd_layer<n-2>		

bnd_layer<0>

Figure 9.1: Illustration of recursive inheritance for the n-cell class `element_t`. The `boundary_ncell_layer` class is abbreviated as `bnd_layer`. The widths of the boxes refer to the sizes of a single class object.

A problem of recursive inheritance is name hiding. For example, a member function `fun()` defined for one layer will be repeated in every layer and thus become inaccessible from the outside. The resolve this issue, member function overloading of the form

```
void fun(dimension_tag<k>) { }
```

at each level k is used. In this way, every layer is accessible from the outside, which is used for the free functions such as `ncells<>()`.

Domains and segments are set up in essentially the same way with a few internal storage facilities adjusted. Instead of an ID handler, the segment inherits from a class providing a reference to the underlying domain.

9.2 n-Cell Storage in Domain and Segment

For storing the n-cells inside a domain, the natural approach is to use a `std::vector<>` for that purpose. However, a drawback of this datastructure is that the number of elements should be known in advance in order to avoid costly reallocations. This is especially a problem for mesh file formats which do not contain the total number of elements in an explicit way. `ViennaGrid` uses a `std::deque<>` (double ended queue) as container for vertices and cells, because it does not suffer form costly reallocations if the number of elements is a-priori unknown.

For non-vertices and non-cells, unique representations of the respective n-cells are required. For example, the edge connecting vertices 1 and 2 must not lead to an edge $[1, 2]$ and an edge $[2, 1]$ in the domain. While such a distinction is simple for vertices, this is harder to achieve for e.g. quadrilateral facets. In `ViennaGrid`, the global vertex IDs of each n-cell are used as a tuple for the identification of the respective element. Thus, the internal storage inside the domain for each non-vertices and non-cells is given by

```
std::map<TupleType, ElementType>
```

where `TupleType` refers to the tuple of sorted global vertex IDs, and `ElementType` is the type of the n-cell.

For segments, only pointers to the global n-cell objects in the domain are stored. Since uniqueness of n-cells is required in segments as well, an internal storage scheme of type `std::set<ElementType *>` is chosen for non-cells, where `ElementType` denotes the type of

the n-cells. For cells, a `std::deque<ElementType *>` is used for the same reasons as for the domain.

Finally, it should be noted that future versions of `ViennaGrid` may provide additional flexibility in customizing the internal storage scheme for domain and segments. In particular, users may be interested in replacing the `std::map<TupleType, ElementType>` used for non-vertices and non-cells with a `std::vector<>` after the setup phase for reasons of memory consumption, faster (random) access and/or better caching possibilities.

Chapter 10

Design Decisions

In this chapter the various aspects that have lead to `ViennaGrid` in the present form are documented. The discussion focuses on key design decisions mostly affecting usability and convenience for the library user, rather than discussing programming details invisible to the library user. Since the design decisions also reflect the history of `ViennaGrid` and the individual preferences of the authors to a certain degree, a more vital language is chosen in the following.

10.1 Iterators

Consider the iteration over all vertices of a `domain`. Clearly, the choice

```
for (VertexIterator vit = domain.begin(); vit != domain.end(); ++vit) {}
```

is not sufficiently flexible in order to allow for an iteration over edges. Nevertheless, the STL-like setup and retrieval of iterators is appealing.

A run-time dispatched iterator retrieval in the spirit of

```
for (VertexIterator vit  = domain.begin(0);
                    vit != domain.end(0);
                  ++vit) {}
```

will sacrifice efficiency especially for loops with a small number of iterations, which is not an option for high-performance environments. Therefore, iterators should be accessed using a compile time dispatch.

Since hard-coding function names is not an option for an parametrized traversal, the next choice is to add a template parameter to the `begin` and `end` member functions:

```
for (VertexIterator vit  = domain.begin<0>();
                    vit != domain.end<0>();
                  ++vit) {}
```

This concept would be sufficiently flexible to allow for an iteration over edges, facets, etc. in a unified way. In fact, this approach is also chosen by DUNE [3] and was also used in an early prototype of `ViennaGrid`. However, there is one pecularity with this approach when it comes to template member functions: According to the C++ standard, the syntax needs to be supplemented with an additional **template** keyword, resulting in

```
for (VertexIterator vit  = domain.template begin<0>();
                    vit != domain.template end<0>();
                  ++vit) {}
```

Apart from the fact that the `template` keyword for member functions is probably unknown to a wide audience, weird compiler messages are issued if the keyword is forgotten. Since a high usability is one of the design goals of `ViennaGrid`, we kept searching for better ways with respect to our measures.

Having high-performance environments in mind, one must not forget about the advantage of index-based for loops such as

```
for (std::size_t i=0; i<3; ++i) { ... }
```

for the iteration over e.g. the vertices of a triangle. The advantage here stems from the fact that the compiler is able to unroll the loop, which is much harder with iterators. Consequently, we looked out for a unified way of both iterator-based traversal as well as an index-based traversal.

In order to stick with a simple iterator-based loop of the form

```
for (VertexIterator vit  = something.begin();
                    vit != something.end();
                  ++vit) {}
```

where `something` is some proxy-object for the vertices in the domain, one finally ends up with the current `viennagrid::ncells<>` approach. Writing the loop in the form

```
for (VertexIterator vit  = ncells<0>(domain).begin();
                    vit != ncells<0>(domain).end();
                  ++vit) {}
```

readily expresses the intention of iterating over 0-cells and does not suffer from the problems related to the `template` keyword. Thanks to the rich overloading rule-set for free functions, an extension to `ncells<0>(segment)` or `ncells<0>(cell)` is immediate. As presented in Chapter 5, the `ncells<>()` approach also allows for an index-based iteration in most cases.

In retrospective, the ranges returned by `ncells<>()` would have come up with coboundary iterators (which were added later), since one then has to pass the enclosing cell complex anyway.

10.2 Default Behavior

A delicate question for a highly configurable library such as `ViennaGrid` is the question of the default behavior. We decided to provide the full functionality by default, even if the price to pay is a possibly slow execution at first instance.

Our decision is based on mostly psychological reasoning as follows: If `ViennaGrid` were tuned for low memory consumption and high speed by default, then a substantial set of functionality would be unavailable by default and causing compilation errors for users that just want to try a particular feature of `ViennaGrid`. It is unlikely that these users will continue to use `ViennaGrid` if they are not able to compile a simple application without

digging through the manual and searching for the correct configuration. On the contrary, if the desired feature works essentially out of the box and is fast enough, users will not even have to care about the further configuration of `ViennaGrid`. However, if additional speed is required, there are plenty of configuration options available, each potentially leading to higher speed or lower memory footprint and thus resulting in a feeling of success.

The bottomline of all this is that we consider it more enjoyable to tune a slower default configuration for maximum speed than to fight with compiler error messages and being forced to work through the manual. We hope that our decision indeed reflects the preferences of our users.

10.3 Segments

Operating on a subset Ω_i of a domain Ω is a common demand. This could be well achieved by what is known as a *view*, i.e. a selection of elements from a domain. However, a view typically possesses a considerably shorter lifetime compared to the domain, thus it is hard to store any data for the view itself. For this reason, segments in `ViennaGrid` are part of the domain, thus having a comparable lifetime in typical situations. In particular, meta information can be stored on a segment during the domain setup stage already.

Another restriction of segments in `ViennaGrid` is that a cell can be member of at most one segment. This restriction is based on an implementation detail and might be removed in future releases.

10.4 The Use of ViennaData

Even though the manual of `ViennaData` as well as Chapter 6 give a number of reasons why data not inherent to the entity modelled by the class should not be stored as a class member, it should be noted that `ViennaData` and `ViennaGrid` were fused in a single library in the very beginning. However, it turned out that what is now `ViennaData` was independent of any mesh-related information, thus the functionality was moved to a stand-alone library. Splitting the complexity involved in storing data for an object on the one hand, and mesh handling on the other hand soon turned out to be beneficial for both libraries, since the reduced complexity in each of them allowed for a clearer view on the core functionalities.

Appendix A

Reference Orientations

The order of the vertices of a n-cell implicitly determines the boundary k-cells, thus it is crucial to provide the vertices in the correct order. While one- and two-dimensional objects provide little space for variations of reference orientations, the situation changes in higher dimensions. The reference orientations of the boundary k-cells provides ample of variations, while only a few choices satisfy requirements such as consistent cell normals.

The reference orientations in `ViennaGrid` are chosen such that the tuples of vertex IDs for all boundary cells are in ascending order. This is accomplished in a generic way for a unit-n-cell from the simplex and hypercube families in the n-dimensional space as follows:

1. Start with a 1-cell at the points $(1, 0, \ldots)$ and $(0, 1, 0, \ldots)$, enumerate the points and set $k = 2$.

2. Prolongate the $k-1$-cell to a k-cell in the k-dimensional subspace induced by the first k unit vectors.

3. Enumerate the new vertices.

4. If $k = n$, stop. Otherwise, go back to 2.

The families of simplices and hypercubes differ in the way the the prolongation is carried out. Details are given in the following subsections.

Boundary k-cells are ordered with respect to the less-than operator acting on the tuple of vertices in ascending order. The first entry has precedence over the second entry, etc.

A.1   Simplices

The prolongation from a $k-1$-simplex to a k-simplex is straightforward, since only one vertex is added. Resulting reference orientations for a triangle and a tetrahedron are given in Fig. A.1.

Boundary k-cell orientations are chosen such that the tuple of vertices is sorted in increasing order. Note that this does not lead to a consistent orientation of n-cell normals. In particular, the boundary k-cells of a triangle and a tetrahedron are thus ordered as follows:
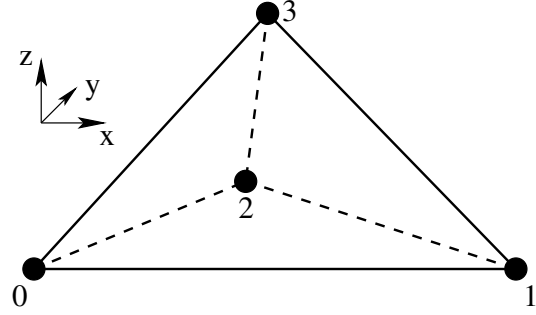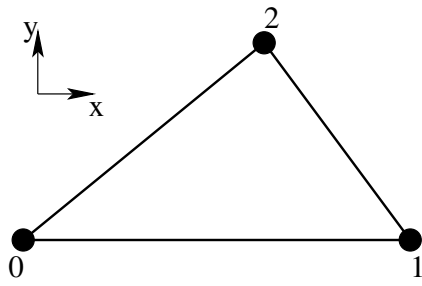
Figure A.1: Reference orientations of a triangle (left) and a tetrahedron (right).
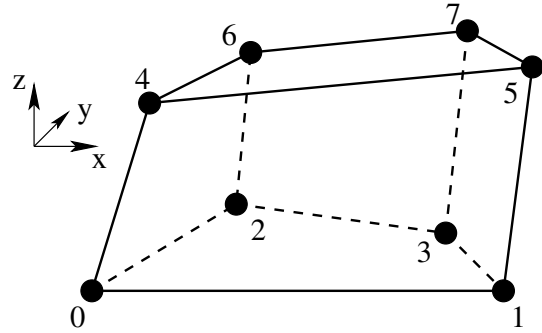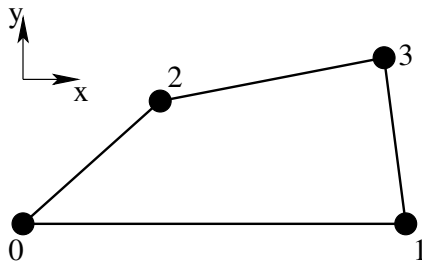

Figure A.2: Reference orientations of a quadrilateral (left) and a hexahedron (right).

	Triangle	Tetrahedron
1-cells	$[0, 1], [0, 2], [1, 2]$	$[0, 1], [0, 2], [0, 3],$ $[1, 2], [1, 3], [2, 3]$
2-cells	$[0, 1, 2]$	$[0, 1, 2], [0, 1, 3],$ $[0, 2, 3], [1, 2, 3]$

A.2 Hypercube

For the prolongation from a unit-$k-1$-hypercube to a unit-k-hypercube, the standard tensor construction is used. This results in a second $k-1$-hypercube shifted along the k-th unit vector, with the new vertices enumerated in the same orientation as the initial $k-1$-hypercube. This procedure can also be seen in Fig. A.2, where the two 1-cells used for the prolongation to the hexahedron are $[0, 1, 2, 3]$ and $[4, 5, 6, 7]$.

For reference, the edges and faces of a quadrilateral and a hexahedron are ordered as follows:

	Quadrilateral	Hexahedron
1-cells	$[0, 1], [0, 2], [1, 3], [2, 3]$	$[0, 1], [0, 2], [0, 4], [1, 3],$ $[1, 5], [2, 3], [2, 6], [3, 7],$ $[4, 5], [4, 6], [5, 7], [6, 7]$
2-cells	$[0, 1, 2, 3]$	$[0, 1, 2, 3], [0, 1, 4, 5], [0, 2, 4, 6],$ $[1, 3, 5, 7], [2, 3, 6, 7], [4, 5, 6, 7]$

Mind that the the reference orientations of a `ViennaGrid` quadrilateral and a `ViennaGrid` hexahedron coincide with those of the VTK types `VTK_PIXEL` and `VTK_VOXEL`, but differ from the orientations of the VTK types `VTK_QUAD` and `VTK_HEXAHEDRON`.

Appendix B

Versioning

Each release of `ViennaGrid` carries a three-fold version number, given by

```
ViennaGrid X.Y.Z .
```

For users migrating from an older release of `ViennaGrid` to a newer one, the following guidelines apply:

- `X` is the *major version number*, starting with `1`. A change in the major version number is not necessarily API-compatible with any versions of `ViennaGrid` carrying a different major version number. In particular, end users of `ViennaGrid` have to expect considerable code changes when changing between different major versions of `ViennaGrid`.

- `Y` denotes the *minor version number*, restarting with zero whenever the major version number changes. The minor version number is incremented whenever significant functionality is added to `ViennaGrid`. The API of an older release of `ViennaGrid` with smaller minor version number (but same major version number) is *essentially* compatible to the new version, hence end users of `ViennaGrid` usually do not have to alter their application code. There may be small adjustments in the public API, which will be extensively documented in the change logs and require at most very little changes in the application code.

- `Z` is the *revision number*. If either the major or the minor version number changes, the revision number is reset to zero. The public APIs of releases of `ViennaGrid`, which only differ in their revision number, are compatible. Typically, the revision number is increased whenever bugfixes are applied, performance and/or memory footprint is improved, or some extra, not overly significant functionality is added.

Always try to use the latest version of `ViennaGrid` before submitting bug reports!

Appendix C

Change Logs

Version 1.0.0

First release

Appendix D

License

Bibliography

[1] "ViennaData." [Online]. Available: http://viennadata.sourceforge.net/

[2] "CGAL - Computational Geometry Algorithms Library." [Online]. Available: http://www.cgal.org/

[3] "DUNE - Distributed and Unified Numerics Environment." [Online]. Available: http://www.dune-project.org/

[4] "GrAL - Grid Algorithms Library." [Online]. Available: http://gral.berlios.de/

[5] "libmesh." [Online]. Available: http://libmesh.sourceforge.net/

[6] "OpenMesh." [Online]. Available: http://www.openmesh.org/

[7] "trimesh2." [Online]. Available: http://gfx.cs.princeton.edu/proj/trimesh2/

[8] "VCGlib." [Online]. Available: http://vcg.sourceforge.net/

[9] "CMake." [Online]. Available: http://www.cmake.org/

[10] "Xcode Developer Tools." [Online]. Available: http://developer.apple.com/technologies/tools/xcode.html

[11] "Fink." [Online]. Available: http://www.finkproject.org/

[12] "DarwinPorts." [Online]. Available: http://darwinports.com/

[13] "MacPorts." [Online]. Available: http://www.macports.org/

[14] "ARPREC high-precision library." [Online]. Available: http://crd.lbl.gov/~dhbailey/mpdist/

[15] "Boost C++ Libraries." [Online]. Available: http://www.boost.org/

[16] "OpenMP." [Online]. Available: http://openmp.org

[17] "VTK - Visualization Toolkit." [Online]. Available: http://www.vtk.org/

[18] "VTK File Formats." [Online]. Available: http://www.vtk.org/VTK/img/file-formats.pdf

[19] "Netgen Mesh Generator." [Online]. Available: http://sourceforge.net/projects/netgen-mesher/

[20] "Paraview - Open Source Scientific Visualization." [Online]. Available: http://www.paraview.org/

[21] "OpenDX." [Online]. Available: http://www.opendx.org/