ViennaMesh
A Highly Flexible Meshing Framework

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

Institute for Microelectronics
Technische Universität Wien
Vienna - Austria - Europe
http://www.iue.tuwien.ac.at

Table of Contents

What Is ViennaMesh?

Open source C++ meshing framework

- LGPL

Based on a highly flexible data structure

- Focus on abstract topology and orthogonality

Abstract concepts enable uniform interfaces

- Generic programming

Problems Addressed by ViennaMesh

Simultaneous use of different meshing algorithms/libraries

- Challenging due to incompatible data structures and interfaces

Extensibility

- Most libraries are hard to extend

Reusability

- Code written against a library can hardly be reused with another library

Flexibility

- Data structure of libraries is often static and inflexible

Framework Details

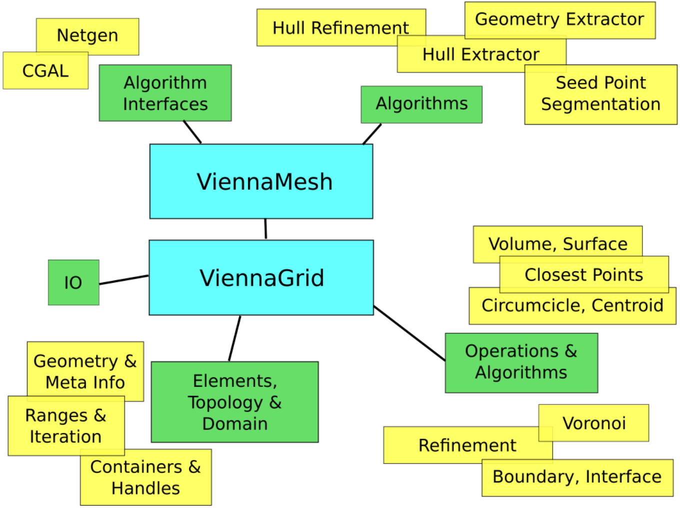
C++ library

- Data structure and core functionality header-only
- C++11 support
- Cmake support

Interfaces to proven libraries

- Meshing: Netgen, CGAL, VERDICT
- Statistics: Boost.Accumulators
- More to come: Tetgen, Triangle, Mesquite, Metis, ...

Framework Overview



Framework Overview Explained

ViennaGrid

- Low level meshing data structure
- Central base of ViennaMesh
- Focus on topology, geometry as layer above topology

ViennaMesh Core

- Core Meshing functionality
- Abstract domain concept
- Abstract algorithm concept

ViennaMesh algorithms

- Interfaces to external libraries

ViennaGrid - Types

Types are queried by result_of

- Similar to C++ result_of
- Tags represent element types

```
typedef  result_of::element<DomainType, ElementTag>
    ::type ElementType;
typedef  result_of::handle<DomainType, ElementTag>
    ::type ElementHandle;

typedef  result_of::vertex<DomainType>
    ::type VertexType;
typedef  result_of::vertex_handle<DomainType>
    ::type VertexHandle;
```

ViennaGrid - Elements

Elements represent topological entities

- Explicit storage of boundary elements
- Using handles to store boundary elements
- Co-Boundary and neighbour elements calculated when needed

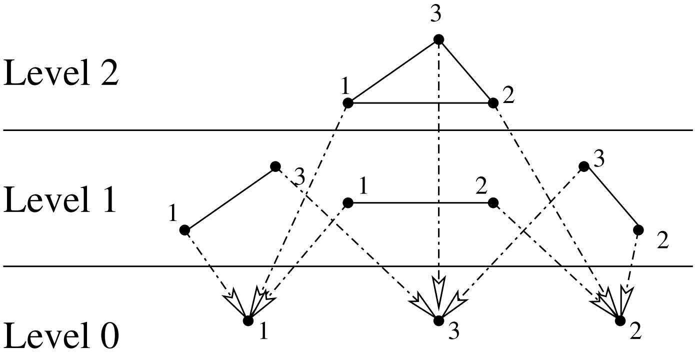
Topological structures of arbitrary topological dimension

- Simplices
- Hypercubes

Dynamic types supported

- Polygons and PLCs

ViennaGrid - Elements



Topological structure of a triangle

ViennaGrid - Topology

Topological complex

- Set of elements
- Intersection of 2 elements $\rightarrow$ empty or another element
- Same elements stored only once
- Non-conforming complexes supported (but with some restrictions)

Example: Create Elements

```
typedef config::tetrahedral_3d_domain DomainType;

DomainType domain;

typedef result_of::element<DomainType, vertex_tag>
    ::type VertexType;
typedef result_of::element<DomainType, tetrahedron_tag>
    ::type CellType;

typedef result_of::handle<DomainType, vertex_tag>
    ::type VertexHandle;
typedef result_of::handle<DomainType, tetrahedron_tag>
    ::type CellHandle;

VertexHandle vertices[4];
for (int i = 0; i < 4; ++i)
    vertices[i]=create_element<VertexType>(domain);

CellHandle cell=create_element<CellType>(domain, vertices);
```

Example: Create Elements

```
typedef config::tetrahedral_3d_domain DomainType;

DomainType domain;

typedef result_of::vertex_handle<DomainType>
    ::type VertexHandle;
typedef result_of::cell_handle<DomainType>
    ::type CellHandle;

VertexHandle vertices[4];
for (int i = 0; i < 4; ++i)
    vertices[i] = create_vertex(domain);

CellHandle cell = create_tetrahedron(domain,
    vertices[0], vertices[1], vertices[2], vertices[3]);
```

Meta Information

Default meta information: geometric point information

- Topology only stores vertices → Geometric information needed

Storage of Geometric information

- Within domain object
- Separate object

Same interface is used

- look_up for domain separate information
- Usage: look_up(domain_or_lunch, element)

Example: Meta Information

```
ElementType element;

deque<double> scalar_values;
map< result_of::id_type<ElementType>::type,
    string > string_values;

look_up(scalar_values, element) = 42.0;
look_up(string_values, element) = "my_element";

VertexType vertex;
std::deque<PointType> geometric_container;

point(domain, vertex) = PointType(0, 0, 1);
point(geometric_container, vertex) = PointType(0, 1, 0);
```

ViennaGrid - Domain, View

Domain object represents collection of elements

- Adds geometric information to topology
- Can be configured to work with any elements

View represents subsets of the domain

- Uses handle to store references
- Can be used to define segments

Example: Geometric Information

```
typedef config::tetrahedral_3d_domain DomainType;
typedef result_of::point_type<DomainType>::type PointType;

DomainType domain;

typedef result_of::vertex_handle<DomainType>
    ::type VertexHandle;
typedef result_of::cell_handle<DomainType>
    ::type CellHandle;

VertexHandle vertices[4];
for (int i = 0; i < 4; ++i)
{
  vertices[i] = create_vertex(domain);
  point( domain, vertices[i] ) = PointType(i, i*i, i*i*i);
}

CellHandle cell = create_tetrahedron(domain,
    vertices[0], vertices[1], vertices[2], vertices[3]);
```

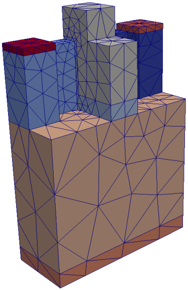
Example: Create View

```
typedef config::tetrahedral_3d_domain DomainType;
typedef config::tetrahedral_3d_view ViewType;

DomainType domain;
ViewType view = create_view( domain );

typedef result_of::vertex_handle<DomainType>
    ::type VertexHandle;
typedef result_of::cell_handle<DomainType>
    ::type CellHandle;

VertexHandle vertices[4];
for (int i = 0; i < 4; ++i)
    vertices[i] = create_vertex(domain);

CellHandle cell = create_tetrahedron(view,
    vertices[0], vertices[1], vertices[2], vertices[3]);
```

ViennaMesh - Segment Support

Support for segments

- Subsets of the mesh
- Preserve interfaces through meshing process



ViennaGrid - Segmentation

Segmentations with Views

- Using Views for segmentation
- One per segment
- Using seed points

Segmentation object

- Stores segment information per element
- Trivial segment information: segment id
- Segment information for hull domain: with orientation

Example: Segmentation

```
typedef config::triangular_3d_domain DomainType;
typedef config::triangular_3d_segmentation SegmentationType;

DomainType domain;
SegmentationType segmentation =
    viennagrid::create_segmentation(domain);

TriangleType triangle;

segmentation.segment_info(triangle).
    positive_orientation_segment_id = segment_id_0;
segmentation.segment_info(triangle).
    negative_orientation_segment_id = segment_id_1;
```

ViennaGrid - Containers, Handles and Ranges

Generic low level storage system

- Container and handle types are configurable
- Default container: std::deque
- Default handle: pointer to element

Range defines a set or subset of elements

- Needed for iteration

ViennaGrid - Iteration

Iteration is type-independent

- Easy access of associated elements
- Same source code for different types

Element Iteration

- Iterating over elements of a domain or view

ViennaGrid - Iteration

Boundary Element Iteration

- Iterating over boundary elements of an element
- Same as element iteration

Co-Boundary Element Iteration

- Iterating over co-boundary elements of an element
- Scope domain or view is needed

Example: Iteration Domain/View

```
DomainType domain;

typedef result_of::element_range<DomainType, vertex_tag>
    ::type VertexRangeType;
typedef result_of::iterator<VertexRangeType>
    ::type VertexRangeIterator;

VertexRangeType vertices = elements( domain );
for (VertexRangeIterator it = vertices.begin();
     it != vertices.end(); ++it)
{
    cout << point(domain, *it) << endl;
}
```

Example: Boundary Element Iteration

```
ElementType element;

typedef result_of::element_range<ElementType, vertex_tag>
    ::type VertexOnElementRangeType;
typedef result_of::iterator<VertexOnElementRangeType>
    ::type VertexOnElementRangeIterator;

VertexOnElementRangeType vertices = elements( element );
for (VertexOnElementRangeIterator it = vertices.begin();
    it != vertices.end(); ++it)
{
    cout << point(domain, *it) << endl;
}
```

Example: Co-Boundary Element Iteration

```
DomainType domain;
VertexType vertex;

typedef result_of::coboundary_range<DomainType, triangle_tag>
    ::type TrianglesOfVertexRangeType;
typedef result_of::iterator<TrianglesOfVertexRangeType>
    ::type TrianglesOfVertexRangeIterator;

TrianglesOfVertexRangeType triangles =
    coboundary_elements( domain, vertex );
for (TrianglesOfVertexRangeIterator it = triangles.begin();
    it != triangles.end(); ++it)
{
    // do something with triangle *it
}
```

IO

Supported file reader

- VTK
- Netgen
- Tetgen poly (PLC)

Supported file writer

- VTK
- OpenDX

VTK reader and writer supports meta data

- support for vertices and cells

Overview ViennaGrid Operations and Algorithms

Element based operations and algorithms

- Use only local element information
- e.g. volume

Domain based algorithms

- Requires domain/view context
- e.g. refine

Element based operations and algorithms

More element operations and algorithms available

- Volume
- Surface
- Inner product
- Norm
- Cross product
- Centroid
- Circumcircle
- Closest points

Example: Volume data transfer

Transfer data from triangle to vertex

- value weighted with triangle volume

```
for (auto v : viennagrid::vertices( domain ) )
{
  numeric_type weighted_value = 0, total_volume = 0;

  for ( auto t : viennagrid::triangles(domain, v) )
  {
    numeric_type current_volume = volume( domain, t );
    total_volume += current_volume;
    weighted_value += current_volume * value(t);
  }

  value(v) = weighted_value / total_volume;
}
```

Example: Volume data transfer

Type independent implementation

- to_tag and from_tag specify the types

```
for (auto v :  viennagrid::elements<to_tag>( domain ) )
{
  numeric_type weighted_value = 0, total_volume = 0;

  for ( auto t :  viennagrid::coboundary_elements<from_tag>(domain, v) )
  {
    numeric_type current_volume = volume( domain , t );
    total_volume += current_volume;
    weighted_value += current_volume * value(t);
  }

  value(v) = weighted_value / total_volume;
}
```

Domain based algorithms

Boundary

- Determines if the current element is a boundary element
- Requires domain/view context

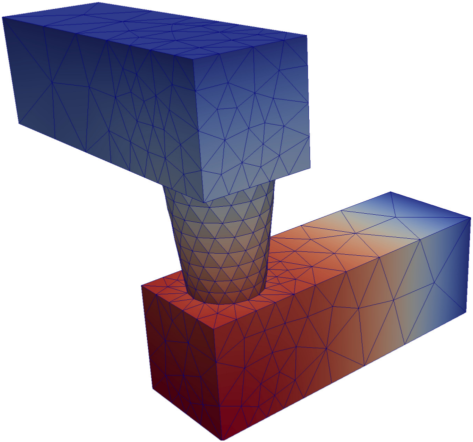
Refine

- Refines previously marked cells

Voronoi

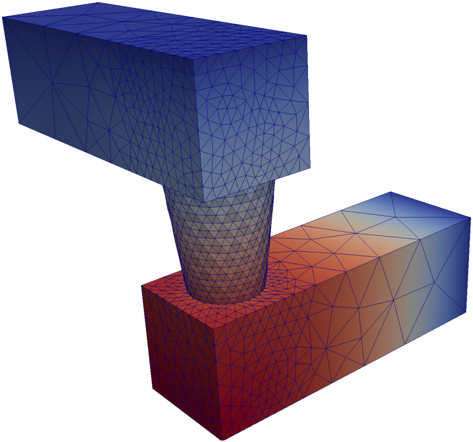
- Calculates the voronoi information of a domain/view

Example: Refinement



Before Refinement

Example: Refinement



After Refinement

Example: Refinement

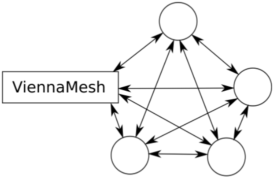
```
for (auto element : cells(domain) )
{
  if (to_refine(element))
    tag_to_refine(element);
}

refine(domain, segments,
       result_domain, result_segments,
       local_refinement_tag() );
```

ViennaMesh Domain Concept

ViennaMesh aims to support external libraries

- Interfaces have to be provided

Each external library comes with its own data structure



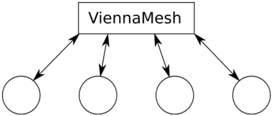
ViennaMesh Domain Concept

ViennaMesh aims to support external libraries

- Interfaces have to be provided

Each external library comes with its own data structure

- Conversions to and from ViennaGrid required
- Other conversions optional



ViennaMesh Algorithm Concept

ViennaMesh provides an uniform interface for complex algorithms

- Query algorithm information and settings object

Execution of algorithm is generic interface

- viennamesh::run_algo<algorithm_tag>(source, destination, settings)
- If required, source and destination is converted implicitly

Returns information of algorithm execution

- Execute time
- Success state
- Errors, warnings and informations

ViennaMesh Algorithms

Internal algorithms in ViennaMesh

- Extract Hull
- Extract PLC Geometry
- Seed point segment marking of hull meshes
- Multi-segment hull refinement
- Mesh doctor: 3D triangular hull

External algorithms with ViennaMesh interface

- Netgen: triangular hull → tetrahedral volume
- CGAL: PLC → triangular hull (2D and 3D)
- CGAL: triangular hull → tetrahedral volume

ViennaMesh Algorithms

External and internal algorithms share common interface

- data structure conversion if needed

```
InputDomainType  input_domain;
OutputDomainType output_domain;
viennamesh::result_of::settings<algorithm_tag>::type
                  settings;

settings.cell_size = 1.0;

viennamesh::run_algo< algorithm_tag >(
    input_domain,
    output_domain,
    settings);
```

ViennaMesh Algorithms

External and internal algorithms share common interface

- Easy exchangeability of algorithms

```
InputDomainType  input_domain;
OutputDomainType output_domain;
viennamesh::result_of::settings< algorithm_tag >::type
                settings;

settings.cell_size = 1.0;

viennamesh::run_algo< algorithm_tag >(
    input_domain,
    output_domain,
    settings);
```

ViennaMesh Examples

Time for some detailed source code! :-)

Conclusion

Flexibility

- Abstract concepts → Write your code only once
- Common interface → Easily change meshing kernel
- High configurability → Use arbitrary topological structures
- High extensibility → Write your own meshing algorithm

Status

- Development release available at sourceforge
- http://viennamesh.sourceforge.net