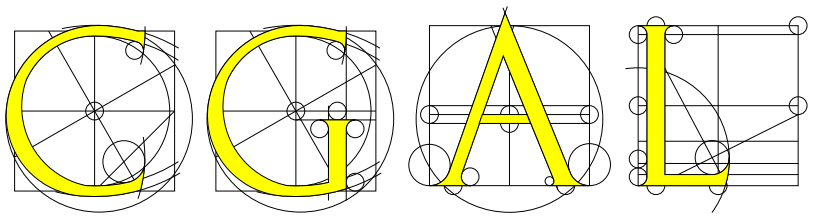

Head_mesher

Separate Build

October 26, 2006

Contents

Chapter 1

Head Mesher

Marc Scherfenberg

1.1 Introduction

This package extends CGAL to be capable of meshing objects which consists of more than one surface. Meshing multi-surface objects can be desired for several reasons. In contrary to a single manifold surface (which nevertheless could consist of several connected components), several surfaces can contain each other. As the mesher respects all surfaces, the resulting mesh not only models the outer surface and the enclosed volume of an object, additionally also its inner structure is maintained. Furthermore it is possible to set different meshing criteria and parameter for different parts of an object's volume, depending on which surfaces enclose the concerning part. An example for a mesh consisting of parts with different bounds on the tetrahedra's size is shown by figure 1.1.

The illustrated mesh is generated by the demo program *Head_mesher* which shows how to use the classes which are contained in this package. It has been developed for an actual medical application and thus has given this package its name.

Several steps have to be done before the mesher can be applied:

- Defining the multi surface
- Constructing the multi surface traits
- Setting the criteria for the mesh

The following sections discuss these issues more detailed.

1.2 Defining the multi surface

As a multi surface consists of several single surfaces, at first these single surfaces have to be instantiated. Any model which statisfies the concept Surface_3 . . . . . page **??** is fine for that. Given three gray-level-images, one possibility would be to do it like this:

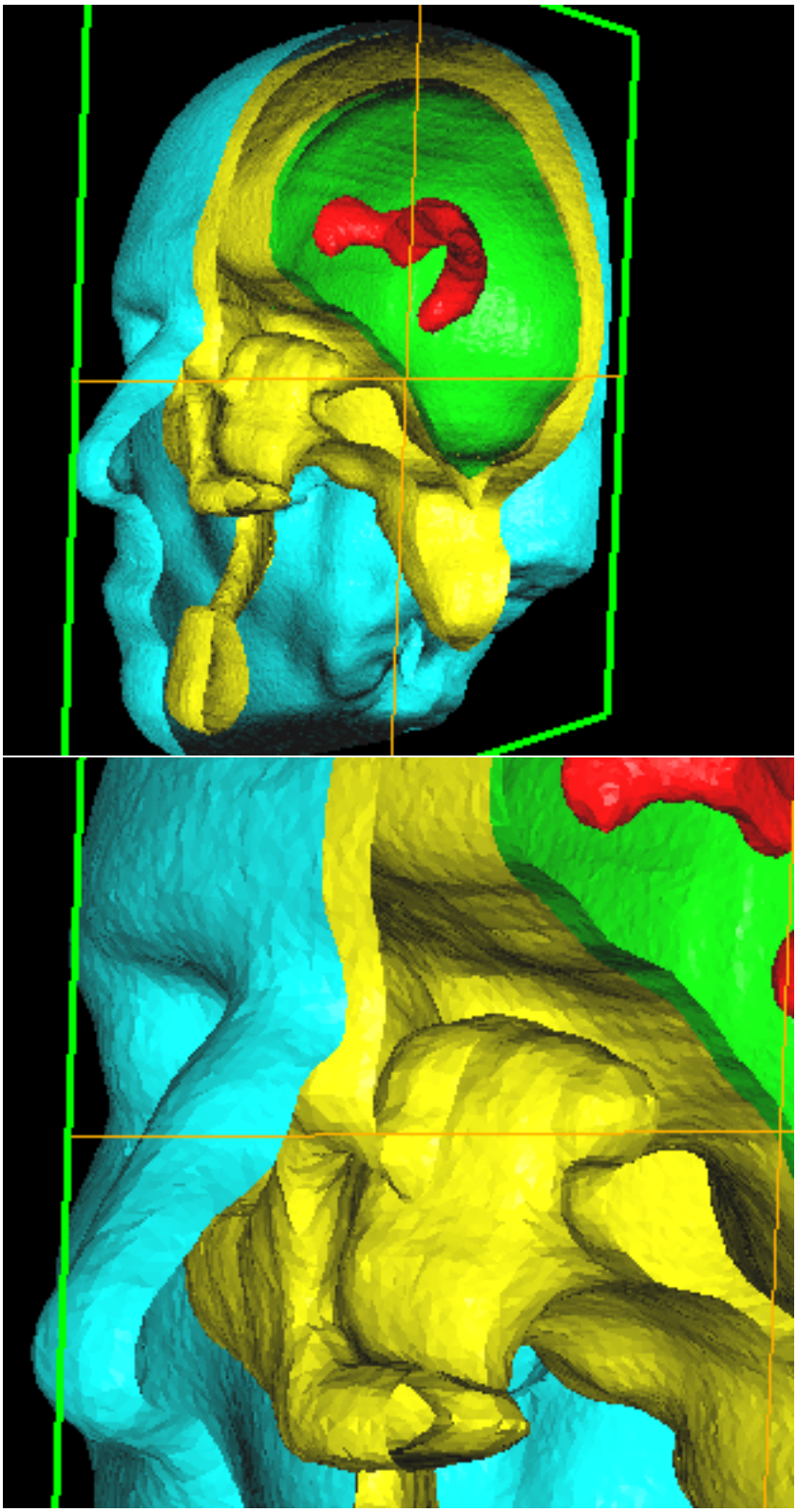

Figure 1.1: There are four surfaces defined: skin, skull, brain and ventricles.

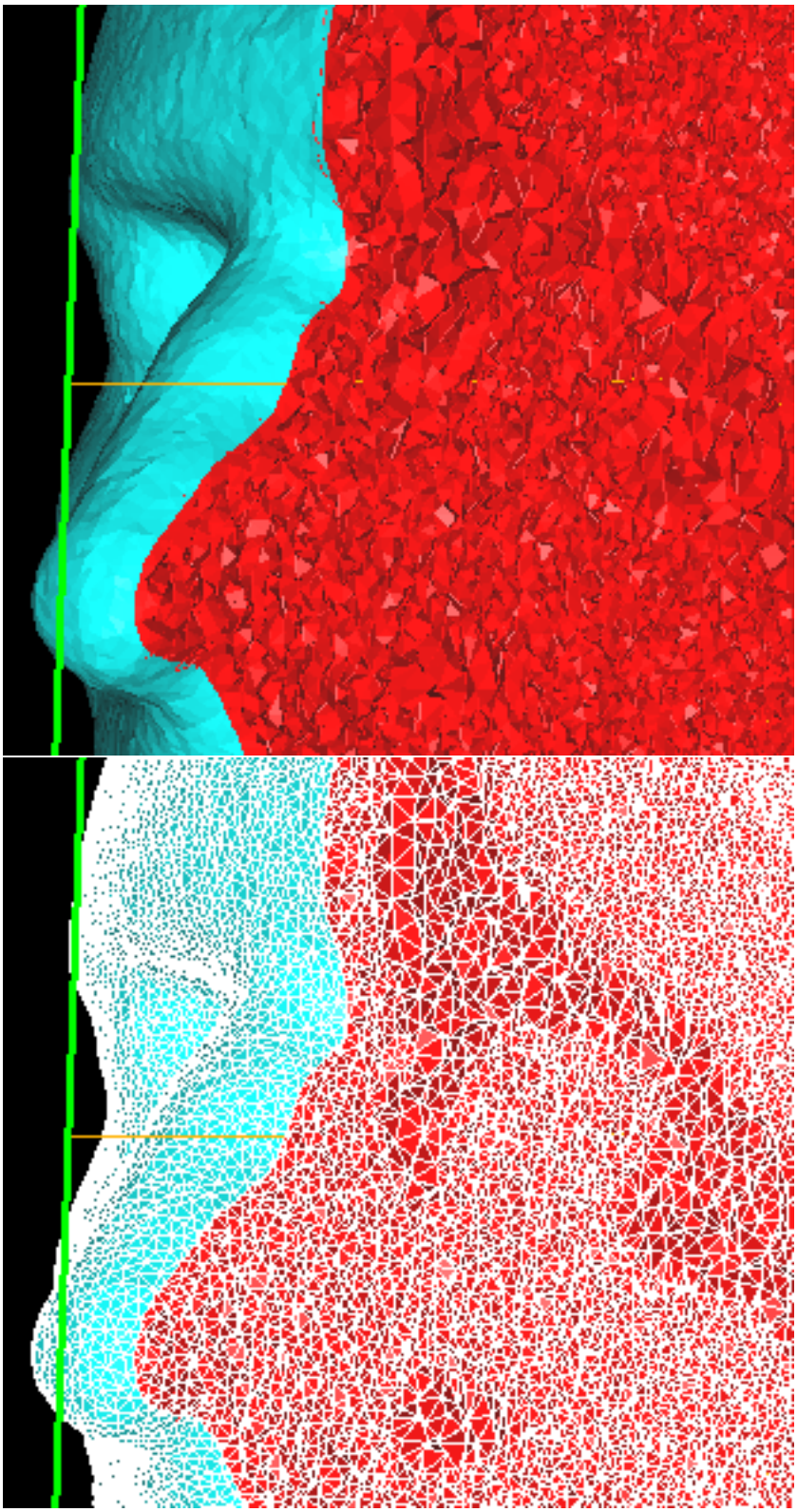

Figure 1.2: The tetraheda touching the cutting plane. Cells are allowed to be bigger within the skull.

```
Gray_level_image image_1("ventricules_0.23.inr.gz", 0.23);
Gray_level_image image_2("brain_1.0.inr.gz", 1.0);
Gray_level_image image_2("skull_2.9.inr.gz", 2.9);

const Sphere_3 bounding_sphere(Point_3(122., 102., 117.), 80000);

Implicit_surface_3 single_surface_1(image_1, bounding_sphere, 1e-3));
Implicit_surface_3 single_surface_2(image_2, bounding_sphere, 1e-3));
Implicit_surface_3 single_surface_3(image_3, bounding_sphere, 1e-3));
```

A multi surface stores its single surfaces in a binary tree. It is necessary to define this tree for the *type* of the multi surface as well as for the multi surface object itself (see *Multi_surface_3*). The trees could be build like this:

```
// definition of the multi surface type
typedef Multi_surface_3<Position_vector,
                        Implicit_surface_3,
                        Implicit_surface_3> Sub_multi_surface_1;

typedef Multi_surface_3<Position_vector,
                        Sub_multi_surface_1,
                        Implicit_surface_3> Multi_surface;

// definition of the multi surface object
Sub_multi_surface_1 sub_multi_surface(single_surface_1, single_surface_2);
Multi_surface       multi_surface(sub_multi_surface, single_surface_3);
```

1.3 Constructing the multi surface traits

The surface traits for a multi surface consists of the set of traits which correspond to the single surfaces. They must be chained to a tree in an analogous way as the surfaces and its types. At first the traits for the single surfaces have to be defined:

```
// definition of the traits type of a single surface
typedef CGAL::Surface_mesher::Implicit_surface_oracle_3<
  Geom_traits,
  Implicit_surface_3,
  Point_with_surface_index_creator,
  Set_indices> Single_trait;

Single_trait single_traits_1(Set_indices(1));
Single_trait single_traits_2(Set_indices(2));
Single_trait single_traits_3(Set_indices(3));
```

Afterwards an instance of the *Multi_surface_mesh_traits_3* can be created. Again it is necessary to chain the types as well as the objects:

```
// definition of the traits type of the multi surface
typedef Multi_surface_mesh_traits_3<Position_vector,
```

```
                                    Single_trait,
                                    Single_trait> Sub_traits_1;
typedef Multi_surface_mesh_traits_3<Position_vector,
                                    Sub_traits_1,
                                    Single_trait> Surface_traits;

// definition of the multi surface traits object
Sub_traits_1   sub_traits_1(single_traits_1, single_traits_2);
Surface_traits surface_traits(sub_traits_1, single_traits_3);
```

1.4 Setting the criteria for the mesh

Except for the exudation of slivers, all parameters which influence the properties of the mesh's cells are defined in criteria-classes. It is distinguished between criteria which can be used for the cells in the volume between two surfaces and criteria usable for facets of the surfaces (which obviously indirectly also influence the cells containing the surface facets). A criterion contains a predicate *Is_bad()*, which is applied to every cell of the mesh or facet of the surface, respectively. If it turns out, that a cell or facet does not fullfill a criterion, the cell is normally refined until it does. Thus the set of the defined criteria determines how long a mesh is generated by this refinement and how it will look like in the end. After this refinement process optionally slivers of the mesh can be detected and exuded, which may lead to a slight violation of the defined criteria.

The concept which defines a volume criterion is MeshCriteria_3 . . . . . page **??**, the one defining a criterion for surface facets is called SurfaceMeshCriteria_3 . . . . . page **??**. Within this package two criteria, which are capable of considering the additional information of multi surface meshes, are contained. They are named *Edge_length_cell_criteria* and *Edge_length_surface_criteria* and are applicable for cells in the volume and surface facets, respectively. Both have in common that they upper bound the edges' lengths according to a given sizing field. A sizing field is a datastructur which maps the so called position vector of a point to an user defined bound. A position vector, in turn, contains the information about which surface encloses a point and wich does not and thus is a vector of bits. The class which implements a sizing field is *Sizing_bounds*, a concept describing the position vector is PositionVector page **??**. For more detailled information about how they work, see their documentation.

The whole process operated by the *Edge_length_cell_criteria* is to calculate the barycenter of a given cell, to determine the position vector of that barycenter with the help of the class *Multi_surface_mesh_traits_3*, then to determine the size bound which belongs to that position vector with the help of the class *Size_bounds* and finally to check whether one of the tetrahedra's edges exceeds the size bound.

The steps taken by *Edge_length_surface_criteria* are only slightly different. For determining the propper size bound, it calculates the two size bounds belonging to the position vectors of the barycenter of the two cells which are incident to the given facet and applies the smaller one to the facet's edges. Thus it applies the bound of the incident volume which is more restrictive.

Two things have to be done by the user for applying these criteria in the refinement process of a mesh: One is to create an external ASCII-file containing rules which define the map from position vectors to bounds. A description of the format of this file is contained in the documentation of *Size_bounds*. The other thing is to instantiate both criteria-classes. This could be done like follows:

```
typedef boost::dynamic_bitset<> Position_vector;

typedef Size_bounds<Position_vector, double> Size_bounds;

typedef Edge_length_surface_criteria<Tr,
```

```
                                    Multi_surface,
                                    Surface_traits,
                                    Size_bounds> Surface_criteria;

typedef Edges_length_cell_criteria<Tr,
                                   Multi_surface,
                                   Surface_traits,
                                   Size_bounds> Cell_criteria;

Size_bounds size_bounds;

// parameter for another volume cell condition
// which is contained in Edges_length_cell_criteria
const double tets_radius_edge_bound = 2.5;

Surface_criteria surface_criteria(tr,
                                  multi_surface,
                                  surface_traits,
                                  size_bounds);

Cell_criteria tets_criteria(tr,
                            multi_surface,
                            surface_traits,
                            size_bounds,
                            tets_radius_edge_bound);;

size_bounds.read_bounds("mesh_size_bounds");
```

1.5 Generating the multi surface mesh

Finally the mesh can be generated by having the *Implicit_surface_mesher_3* refining a small initial mesh it:

```
typedef CGAL::Implicit_surfaces_mesher_3<C2t3,
                                         Multi_surface,
                                         Surface_criteria,
                                         Cell_criteria,
                                         Surface_traits> Mesher;

// the triangulation, for the definition of the type see Head_mesher.cpp
Tr tr;

// the surface-part of the triangulation
typedef CGAL::Complex_2_in_triangulation_3<Tr> C2t3;
C2t3 c2t3;

// construction of initial points on the surfaces
surface_traits.construct_initial_points_object()(multi_surface,
                                                 CGAL::inserter(tr));
// refinement
mesher.refine_mesh();
```

```
// sliver exudation
CGAL::Mesh_3::Slivers_exuder<C2t3> exuder(tr);
exuder.pump_vertices(0.2);
```

An overall example, which compiles and was used to produce the images which are shown in the introduction, is the program *Head_mesher.cpp*.

Head Mesher Reference Manual

Marc Scherfenberg

1.6 Concepts

1.7 Classes

1.8 Alphabetical List of Reference Pages

Class

CGAL::Head_mesher::Cell_with_volume_index<GeomTraits, Cb=CGAL::Triangulation_ds_cell_base_3<>>

Definition

The class *Head_mesher::Cell_with_volume_index<GeomTraits, Cb=CGAL::Triangulation_ds_cell_base_3<>>* extends the given cell base type *Cb* by storing an additional integer as volume index. This includes the overriding of the stream operators in order to be able to make the additional information persistent.

#include <CGAL/Head_mesher/Cell_with_volume_index.h>

Creation

Head_mesher::Cell_with_volume_index<GeomTraits, Cb=CGAL::Triangulation_ds_cell_base_3<>> cell;

Default constructor, the index is set to *-1*.

Operations

int	*cell.volume_index()*	Returns the volume index.
void	*cell.set_volume_index(const int i)*	Sets the volume index to *i*.

See Also

Mesh_3_IO.

CGAL::Head_mesher::Edge_length_cell_criteria<Tr, Multi_surface_3, Multi_surface_mesh_traits_3, Size_bounds>

Class

Definition

The class *Edge_length_cell_criteria* defines a criterion for cells which guarantees that

1. the edges of the tetrahedron are not longer than the bound given by *Size_bounds* according to the cell's position vector (calculated by *Multi_surface_mesh_traits_3*) for the barycenter of that cell

2. the aspect ratio between the shortest edge of the terahedra and the radius of its circumsphere is not higher than a given bound

Note: The bounds of *Size_bounds* are assumed to be defined as a constant value, not as a function.

#include <Edge_length_cell_criteria.h>

Is Model for the Concepts

MeshCriteria_3 . . . . . page **??**

Types

Head_mesher::Edge_length_cell_criteria<Tr, Multi_surface_3, Multi_surface_mesh_traits_3, Size_bounds>::Quality

Quality type, see MeshCriteria_3 . . . . . page **??**

Head_mesher::Edge_length_cell_criteria<Tr, Multi_surface_3, Multi_surface_mesh_traits_3, Size_bounds>::Cell_handle

Cell handle type, see MeshCriteria_3 . . . . . page **??**

Creation

Head_mesher::Edge_length_cell_criteria<Tr, Multi_surface_3, Multi_surface_mesh_traits_3, Size_bounds> crit(Tr &tr,

Multi_surface_3 &surface,

Multi_surface_mesh_traits_3 &traits,

*Size_*

bounds &size_bounds,

const

double radius_edge_bound = 2.)

The last parameter is the bound for the aspect ratio between the shortest edge of the terahedra and the radius of its circumsphere.

Operations

bool *crit.is_bad(Cell_handle ch, Quality& qual)*

Function object which can be obtained by *is_bad_object()*. Returns *true*, if at least one of the conditions which are contained in this criteria class is not satisfied and *false* otherwise.

See Also

Edge_length_surface_criteria,
Multi_surface_3,
Multi_surface_mesh_traits_3,
PositionVector . . . . . page **??**,
Size_bounds,
SurfaceMeshCriteria_3 . . . . . page **??**.

CGAL::Head_mesher::Edge_length_surface_criteria<Tr, Multi_surface_3, Multi_surface_mesh_traits_3, Size_bounds>

Class

Definition

The class *Edge_length_surface_criteria* defines a criterion for facets of surfaces which guarantees that

1. all vertices of the facet must belong to the same surface, which is, their points must have the same surface index.
2. the edges of the facet are not longer than the minimum of the bounds correlated to the two volumes which are incident to the surface which the facet belongs to. Both bounds are determined by using the class *Size_bounds* for the position vector of the barycenter of the incident cells. The needed position vectors are calculated with the help of the class *Multi_surface_mesh_traits_3*.

Note: The bounds of *Size_bounds* are assumed to be defined as a constant value, not as a function.

#include <Edge_length_surface_criteria.h>

Is Model for the Concepts

SurfaceMeshCriteria_3 . . . . . page **??**

Types

Head_mesher::Edge_length_surface_criteria<Tr, Multi_surface_3, Multi_surface_mesh_traits_3, Size_bounds> :: Quality

Quality type, see SurfaceMeshCriteria_3 . . . . . page **??**

Head_mesher::Edge_length_surface_criteria<Tr, Multi_surface_3, Multi_surface_mesh_traits_3, Size_bounds> :: Facet

Facet type, see SurfaceMeshCriteria_3 . . . . . page **??**

Creation

Head_mesher::Edge_length_surface_criteria<Tr, Multi_surface_3, Multi_surface_mesh_traits_3, Size_bounds> crit(Tr &tr,

Multi_surface_3 &surface,

Multi_surface_mesh_traits_3 &traits,

Size_bounds &size_bounds)

Operations

bool *crit.is_bad(Facet f, Quality& qual)*

Function object which can be obtained by *is_bad_object()*. Returns *true*, if at least one of the conditions which are contained in this criteria class is not satisfied and *false* otherwise.

See Also

Edge_length_cell_criteria,
Multi_surface_3,
Multi_surface_mesh_traits_3,
PositionVector . . . . . page **??**,
Size_bounds,
SurfaceMeshCriteria_3 . . . . . page **??**.

CGAL::Head_mesher::Multi_surface_3<PositionVector, Surface_a, Surface_b>

Class

Definition

Head_mesher::Multi_surface_3<PositionVector, Surface_a, Surface_b> is a model for the concept Surface_3 page **??** and can be used for describing a surface which itself consists of several indepedent surfaces.

In particular, the class *Multi_surface_3* is the root of a binary tree which leafs are the single surfaces and which inner nodes are of type *Multi_surface_3*, as well. Chaining two subtrees together is done by invoking the constructor of this class.

The order of the single surfaces is based on a depth-first search from left to right within the tree, which yields into a surface order from left to right according to the position of a single surface among the leafs.

When including *Multi_surface_3.h* the *Surface_mesh_traits_generator_3* is partially specialized for this surface and can be used for obtaining the type of the corresponsing surface traits.

#include <Multi_surface_3.h>

Parameters

The first template parameter must be a model of the homonymous concept PositionVector . . . . . page **??**. When not using the *Surface_mesh_traits_generator_3* for getting the type of the corresponding surface traits, it must be considered that the first template parameter of *Multi_surface_mesh_traits_3* must represent the same model as this parameter.

The second and third template parameters are the types of the subtrees which are to be chained by the constructor. These two template parameters lead to a tree of surface types which must have the same structure as the tree of the surface instances.

Is Model for the Concepts

Surface_3 . . . . . page **??**

Creation

Head_mesher::Multi_surface_3<PositionVector, Surface_a, Surface_b> multi_surface(Surface_a& surface_a, Surface_b& surface_b)

The parameters are the two subtrees which are to be chained. Each one's type has to be a model of Surface_3 . . page **??**, in particular it can be of this type, which means that a subtree is chained, or of a type describing a single surface.

Operations

Surface_a& *multi_surface.surface_a()*

Returns the left subtree of surfaces.

Surface_b& *multi_surface.surface_b()*

Returns the right subtree of surfaces.

See Also

Multi_surface_mesh_traits_3,
PositionVector . . . . . page **??**. Surface_3 . . . . . page **??**,
Surface_mesh_traits_generator_3,
SurfaceMeshTraits_3 . . . . . page **??**,

CGAL::Head_mesher::Multi_surface_mesh_traits_3<PositionVector, SurfaceMeshTraits_3_a, SurfaceMeshTraits_3_b>

Class

Definition

The class *Multi_surface_mesh_traits_3* is a model of SurfaceMeshTraits_3 . . . . . page **??**. It is a model which works for surfaces of the type *Multi_surface_3* and provides the methods *is_in_volume* and *position_vector* additionally to the members which are required by the concept.

In particular this surface traits model works by building a binary tree which leafs are the traits for a single surface each. Its inner nodes are objects of this class, each chaining two subtrees of traits together.

The generated tree of surface traits must have the same structure as the tree of surfaces which corresponds to an instance of that surface traits class. This means, that each node of the traits tree corresponds to exactly one node in the tree of surfaces.

This in turn implies that a node of the traits tree of the type *Multi_surface_mesh_traits_3* corresponds to a node in the surface tree of the type *Multi_surface_3* and all other nodes (in fact leafs) have a type which is another model of SurfaceMeshTraits_3 . . . . . page **??** or Surface_3 . . . . . page **??** respectively.

The order of the surfaces or surface traits respectively is based on a depth-first search from left to right within the trees, which yields into a surface traits order from left to right according to the position of a traits object among the leafs.

#include <Multi_surface_mesh_traits_3.h>

Parameters

The template parameter *PositionVector* must be a model of the homonymous concept. The same model has to be passed as first template parameter to the class *Multi_surface_3*.

The remaining template parameters determine the type of the two subtrees which this class chains and must be a model of the concept SurfaceMeshTraits_3 page **??**. It must be considered that the tree of the types of surface traits must have the same structure as the tree of the surface traits instances.

Types

typedef typename SurfaceMeshTraits_3_a::Point_3

Point_3;

typedef typename SurfaceMeshTraits_3_a::Segment_3

Segment_3;

typedef typename SurfaceMeshTraits_3_a::Ray_3

Ray_3;

typedef typename SurfaceMeshTraits_3_a::Line_3

Line_3;

typedef typename Multi_surface_3<PositionVector, SurfaceMeshTraits_3_a::Surface_3, SurfaceMeshTraits_3_b::Surface_3>

Multi_surface; *Multi_surface* must represent a tree of surfaces having the same structure as the tree of surface traits represented by an object of this class.

typedef Multi_surface

Surface_3;

Head_mesher::Multi_surface_mesh_traits_3<PositionVector, SurfaceMeshTraits_3_a, SurfaceMeshTraits_3_b> :: Intersect_3

Type of a function object calculating intersections of the multi surface with an object of type *Type1* by providing the operator
CGAL::object operator()(Multi_surface surface, Type1 type1).
Type1 may be a *Segment_3*, a *Ray_3* or a *Line_3*.

Head_mesher::Multi_surface_mesh_traits_3<PositionVector, SurfaceMeshTraits_3_a, SurfaceMeshTraits_3_b> :: Construct_initial_points

Type of a function object providing the following operator to construct initial points on the multi surface
template <class OutputIteratorPoints> operator()(OutputIteratorPoints pts, int n=20).
Outputs a set of points (*n* on each surface).

Is Model for the Concepts

SurfaceMeshTraits_3 . . . . . page **??**

Creation

Head_mesher::Multi_surface_mesh_traits_3<PositionVector, SurfaceMeshTraits_3_a, SurfaceMeshTraits_3_b> traits(SurfaceMeshTraits_3_a& traits_a,

SurfaceMeshT

3_b& traits_b)

The parameters are the two subtrees which are to be chained. Each parameter's type has to be a model of SurfaceMeshTraits_3 . . . . . page **??**, in particular it can be of this type, which means that a subtree is chained, or of a type representing the traits for a single surface.

Operations

Intersect_3 *traits.traits.intersect_3_object()*

Construct_initial_points

traits.traits.construct_initial_points_object()

PositionVector *traits.position_vector(Multi_surface &multi_surface, Point_3 p)*

Returns the position vector of *p* with respect to the surfaces contained in *multi_surface*.

See Also

Multi_surface_3,
PositionVector . . . . . page **??**
Surface_3 . . . . . page **??**
Surface_mesh_traits_generator_3,
SurfaceMeshTraits_3 . . . . . page **??**

Concept

Head_mesher::PositionVector

Definition

A Position vector is a type which describes the location of a point with respect to its relative location according to different surfaces within the meshing domain. More precisely it is a bitset in which the *i*th bit corresponds to the result of the method *intersect_3* of a model for the concept *SurfaceMeshTraits_3*. In other words, the *i*th bit of a position vector is 1 if the cell is located within the *i*th surface and 0 otherwise.

Types

The type of the bits is *bool*.

Head_mesher::PositionVector:: size_type	The unsigned integer type for representing the size of the position vector, which is the number of surfaces used for describing a cell's location.

Creation

Head_mesher::PositionVector pv(size_type num_surfaces, unsigned long value = 0);

The *value* parameter is the interpretation of the bitset as an integer, or rather an *unsigned long* from the technical point of view.

Operations

A model of this concept must implement the standard container methods as *empty*, *size* and *push_back*, the assignment operator *operator&=*, the comparision operators *operator==* and *operator!=*, the subscript operator *operator[]* and the following functions

void	*pv.append(bool value)*	extend the position vector for the given bit.
unsigned long	*pv.to_ulong()*	returns the interpretation of the bitset as a number.

Has Models

A model which fullfills the concept Head_mesher::PositionVector is the bitset implemented in the boost-library *boost::dynamic_bitset<>*. In order to use it, *boost/dynamic_bitset.hpp* must be included.

See Also

A class which defines position vectors for given cells is:
Multi_surface_mesh_traits_3<PositionVector, SurfaceMeshTraits_3_a, SurfaceMeshTraits_3_b>.

CGAL::Head_mesher::Size_bounds<PositionVector, Bound>

Class

Definition

The main functionality of the class *Head_mesher::Size_bounds<PositionVector, Bound>* is to map a given position vector to a (size) bound. A position-vector is some data determining the location of a triangulation's cell and must be a model of the concept PositionVector . . . page **??**. The template parameter Bound can be choosen almost arbitrarily, e.g. it can represent a constant or a function.

The first step when using this class is to read the definitions for the mapping from an input-stream, e.g. an external textfile. This can be done by calling the member function *read_bounds*. Afterwards two possibilities for determining bounds are offered. The member function *get_bound* returns a single bound for a given position vector by applying the definition-rules which are read before. If fast access to the bounds is important, the member function *read_all_bounds* should be used instead. It returns a sequential container which contains the bounds for all possible values of the position vector type by precalculating them and storing a bound at the index according to the interpretation of the position vector's bitset as an integer. Note that the space needed for container growths exponentially with the length of the position vector.

#include <Size_bounds.h>

Parameters

The first template argument must be a model of the PositionVector . . . . . page **??** concept.

The second template argument determins the type of the returned bounds. The only restriction is that it has to implement the input operator >>.

Types

The class *Head_mesher::Size_bounds<PositionVector, Bound>* defines the following types:

Head_mesher::Size_bounds<PositionVector, Bound>:: PositionVector

Members of this type represent the location of a cell. Corresponds to the first template parameter.

Head_mesher::Size_bounds<PositionVector, Bound>:: Bound

Type of the bounds. Corresponds to the second template parameter.

Head_mesher::Size_bounds<PositionVector, Bound>:: All_bounds

Fast type that holds all theoretically possible bounds. It is a sequential container implementing the subscript-operator.

typedef pair<string, Bound>

Definition; Rule for mapping a certain pattern for position vectors to a certain bound.

Head_mesher::Size_bounds<PositionVector, Bound>:: Defined_bounds

Container for all defined pattern-bound-rules.

Creation

Head_mesher::Size_bounds<PositionVector, Bound> size_bounds;

Default constructor.

Operations

bool *size_bounds.read_bounds(std::istream& in, std::ostream& out = std::cout)*

Reads rules that define the bounds from the inputstream *in* and outputs user-information on the optional parameter *out*. A rule consists of a pattern for describing when a bound should be applied, and the value of the bound itself. A simple pattern would be a concrete instance of a position vector. In order to avoid the need of defining bounds for all theoretically possible position vectors or to avoid undefined bounds respectively, it is possible to use so called jokers in a pattern. A joker is symbolized by an underscore '_' and means that the bit of a position vector at the position of the joker does not restrict the application of the bound. A rule containing less jokers in comparison to another one is considered as more specialized. If there are more bounds applicable to a postion vector, the rule which is more specialized is given the higher priority. If two applicable rules are equal specialized, the rule which is first mentioned in the inputstream is considered.
Each pattern which is used in the inputstream must have the same length. If the inputsream is well formatted, *read_bounds* reads the definitions and returns *true*, otherwise it stops and returns *false*.
It follows a simple example for a textfile defining the rules for 16 different position vectors:

```
____ 5
01__ 3
0110 2
_10_ 4
```

First it defines a default bound of 5 which is applied if no other pattern matches a position vector, since a pattern only consisting of jokers is least specialized. A position vector 0101 would be mapped to a size bound of 3, because the according rule is defined before the rule mapping to 4.

Bound *size_bounds.get_bound(const PositionVector pv)*

Returns the bound for the given position vector.

All_bounds *size_bounds.get_all_bounds()*

If not done since reading new bounds by *read_bounds*, bounds for all possible position vectors are determined and stored in a sequential container which implements the subscript operator and is returned by this function. A bound according to a position vector can then be determined by interpreting the bitset of the position vector as integer and using it as an index for the container.
This method gives much faster access to bounds than *get_bound*, but needs $O(2^n)$ space with n as the length of a position vector.

Defined_bounds *size_bounds.get_defined_bounds()*

Returns a container consisting of all rules defining the mapping.

unsigned int *size_bounds.number_of_surfaces()*

Returns the number of surfaces which is equal to the length of the used position vectors.

void *size_bounds.print_all_bounds(std::ostream& out = std::cout)*

Prints a table of all possible position vectors and their corresponding bounds to *out*.

See Also

Concept PositionVector . . . . . page **??**.

```
#include <fstream>
#include <boost/dynamic_bitset.hpp>

#include <CGAL/Head_mesher/Size_bounds.h>

using namespace std;

int main()
{
  // boost::dynamic_bitset<> is a model for the concept Position_vector
  typedef boost::dynamic_bitset<> Position_vector;

  // double-values as bound
  typedef CGAL::Head_mesher::Size_bounds<Position_vector, double> Size_bounds;

  ifstream file("test_size_bounds.txt");

  Size_bounds size_bounds;
```

```
  if (!size_bounds.read_bounds(file)) {
    cout << "could not read file or file is in wrong format" << endl;

    return EXIT_FAILURE;
  }

  size_bounds.print_all_bounds();

  // translates the integers of the std::cin into a position vector and prints
  // the corresponding bound
  while (true)
  {
    unsigned int l;

    cin >> l;

    Position_vector pv(3, l);

    cout << pv << ":   " << size_bounds.get_bound(pv) << endl;
  }
}
```

CGAL::Head_mesher::Write_maillage<C2t3>

Functor Class

Definition

The function object class *Head_mesher::Write_maillage<C2t3>* writes a mesh in a format called 'Maillage', which is a format for storing meshes including surfaces. A `.maillage`-file consists of a header and three consecutive parts: points-part, tetrahedra-part and surface-facet-part.

The schema looks like follows:

$$
\begin{aligned}
&npntnf\\
&x_1y_1z_1\\
&\ldots\\
&x_{np}y_{np}z_{np}\\
&it_1jt_1kt_1lt_1vi_1\\
&\ldots\\
&it_{nt}jt_{nt}kt_{nt}lt_{nt}vi_{nt}\\
&if_1jf_1kf_1lf_1si_1\\
&\ldots\\
&if_{nf}jf_{nf}kf_{nf}si_{nf}
\end{aligned}
$$

where

$$
\begin{aligned}
np &- numberoftpoints\\
nt &- numberoftetrahedra\\
nf &- numberoffacetsonsurfaces\\
it_ijt_ikt_ilt_i &- indicesofpointsofithtetrahedra\\
vi_i &- volumeindexofithtetrahedra\\
if_ijf_ikf_i &- indicesoffacetsofithtetrahedra\\
si_i &- surfaceindexofithfacet
\end{aligned}
$$

Furthermore a single file for every surface is created, which containes the same header and points-part, no tetrahedra-part and finally the facets on that surface.

Parameters

The template parameter must be a model of the concept SurfaceMeshComplex_2InTriangulation_3 . . . page **??**.

#include <Maillage_format.h>

Creation

Head_mesher::Write_maillage<C2t3> write_maillage;

Default constructor.

Operations

bool *write_maillage(std::string fileprefix, C2t3 c2t3)*

Writes the mesh *c2t3* in a file named `<fileprefix>.maillage` and additionally generates the `<fileprefix>.surface*`-files.
Returns *false* if the writing fails due to an invalid mesh, otherwise *true* is returned.

See Also

SurfaceMeshComplex_2InTriangulation_3 . . . . . page **??**.

Index

Pages on which definitions are given are presented in **boldface**.