PROFI

Project number:	FP6-511572
Project acronym:	PROFI
Title:	Perceptually-relevant Retrieval Of Figurative Images

Deliverable No: D4.4: Implementation of curve matching

Short description:
We implemented the randomized algorithms for finding similarities between two shapes A and B, that were described in deliverable 4.3. Shapes are modelled by sets of line segments or polygonal curves. The allowed classes of transformations are translations, rigid motions, homotheties and similarities. The default transformation class is similarities.
The major idea is to take random samples of points from both shapes and give a vote for that transformation (translation, rigid motion, or similarity) matching one sample with the other. If that experiment is repeated frequently we obtain by the votes, a certain distribution of points in the space of transformations. Clusters of this point set indicate which transformations give the best match between the two figures.
This document describes the class design and Application Programming Interface.

Due month:	M18
Delivery month:	M18
Lead partner:	Freie Universität Berlin
Partners contributed:	Freie Universität Berlin
Classification:	RE

Information Society Technologies

Project funded by the European Community under the “Information Society Technologies” Programme

Contents

1 Introduction

The library described in this API provides functionality for the comparison of shapes that represent figurative images. Shapes are modelled by sets of planar polygonal curves.

Two shapes S_1 and S_2 are considered to resemble each other, if there exists a transformation $t \in T : \mathbb{R}^2 \to \mathbb{R}^2$ from a distinct class of allowable transformations T such that many parts of the transformed shape $t(S_2)$ are in the proximity of parts of the other shape S_1 and vice versa. Supported classes of allowable transformations are translations, homotheties, rigid motions and similarities. The default is the class of similarities.

We use a distance measurer between shapes, i.e., a mapping that assigns to any pair (S_1, S_2) of shapes a nonnegative real number $d(S_1, S_2)$ which is the value of dissimilarity of these two shapes.

Matching two shapes S_1 and S_2 in this sense means to determine the transformation $t^* \in T$ that minimizes $d(S_1, t^*(S_2))$ and to compute the value $d^* = d(S_1, t^*(S_2))$.

The library uses the probabilistic approach described in deliverable D4.3 section 2.2 to determine the optimal transformations.

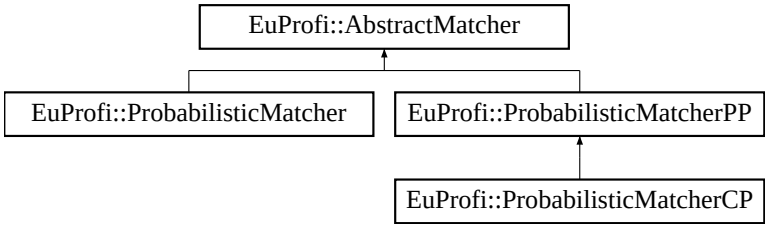
The distance measure used is the one described in deliverable D4.3 section 2.3.

2 Matching

2.1 EuProfi::AbstractMatcher Class Reference

`#include <probabilisticMatcher.h>`

Inheritance diagram for EuProfi::AbstractMatcher::



2.1.1 Detailed Description

Base class for the matching algorithms.

This class defines methods for matching two shapes and computing the distance between them, as well as for setting a fixed first shape and comparing any other shape to the fixed one.

The shapes to be compared are assumed to be modeled by sets of polygonal curves, and can be given as class PolylineSet, PolylineSet2, or by a string (`char*`), which is interpreted as the file name of the file containing the shape description. The classes PolylineSet and PolylineSet2 are defined within the PROFI project framework and contain vertex coordinates of the polygonal curves.

The general procedure is the following: For two shapes S_1 and S_2 some candidate transformations t^i, $i = 1, 2, \ldots$ are computed that map S_2 to S_1. For each candidate transformation t the transformed shape $t^i(S_2)$ and S_1 are compared using our distance measure.

Typical usage:

- Calling match(s2,s1) to determine a transformation that lets s2 match best s1 and to compute the corresponding distance value.
- Calling distance(s2,s1) to determine the distance value corresponding to the best matching of the shapes.
- Calling setFirst(s1) to determine the first shape for comparison.
- Calling matchToFixedFirst(s2) to find the best transformation and the corresponding distance value for the preset shape s1 and a given shape s2. Requires prior call setFixedFirst(s1).
- Calling distanceToFixedFirst(s2) in order to determine just the distance value for the preset shape s1 and the given shape s2 corresponding to the best matching of the shapes. Requires prior call setFixedFirst(s1).

This class performs data type transform from the types PolylineSet and PolylineSet2, defined in the project framework, to the internal representation Shape, captures reading the shape data from a file, and provides the functionality of the implemented algorithms according to the predefined interfaces. The core implementation of the algorithms is done in classes Matcher, Algorithm01 and AverageWeightDistance.

See also:

ProbabilisticMatcher, PartialProbabilisticMatcher, Matcher

Public Types

- typedef PolylineSet **argument_type**
- typedef PolylineSet2 **argument_type_2**
- typedef char * **argument_type_s**
- typedef SimilarityInfo **full_result_type**

Public Member Functions

- full_result_type match (argument_type_s shape2, argument_type_s shape1)
- full_result_type match (argument_type shape2, argument_type shape1)
- full_result_type match (argument_type_2 shape2, argument_type_2 shape1)
- double distance (argument_type_s shape2, argument_type_s shape1)
- double distance (argument_type shape2, argument_type shape1)

- double distance (argument_type_2 shape2, argument_type_2 shape1)
- void setFirst (argument_type_s shape1)
- void setFirst (argument_type shape1)
- void setFirst (argument_type_2 shape1)
- full_result_type matchToFixedFirst (argument_type_s shape2)
- full_result_type matchToFixedFirst (argument_type shape2)
- full_result_type matchToFixedFirst (argument_type_2 shape2)
- double distanceToFixedFirst (argument_type_s shape2)
- double distanceToFixedFirst (argument_type shape2)
- double distanceToFixedFirst (argument_type_2 shape2)
- void clearFirst ()

2.1.2 Member Function Documentation

2.1.2.1 MatchingResult EuProfi::AbstractMatcher::match (argument_type_s *shape2*, argument_type_s *shape1*)

Finds and returns the best transformation mapping shape2 to shape1, along with the distance value corresponding to that transformation.

Shape data are read from files with names given by shape1 and shape2.

2.1.2.2 MatchingResult EuProfi::AbstractMatcher::match (argument_type *shape2*, argument_type *shape1*)

Finds and returns the best transformation mapping shape2 to shape1, along with the distance value corresponding to that transformation.

Shapes are given as PolylineSet.

2.1.2.3 MatchingResult EuProfi::AbstractMatcher::match (argument_type_2 *shape2*, argument_type_2 *shape1*)

Finds and returns the best transformation mapping shape2 to shape1, along with the distance value corresponding to that transformation.

Shapes are given as PolylineSet2.

2.1.2.4 double EuProfi::AbstractMatcher::distance (argument_type_s *shape2*, argument_type_s *shape1*)

Finds the best transformation mapping shape2 to shape1, and computes and returns the distance value corresponding to that transformation.

Shape data are read from files with names given by shape1 and shape2.

2.1.2.5 double EuProfi::AbstractMatcher::distance (argument_type *shape2*, argument_type *shape1*)

Finds the best transformation mapping shape2 to shape1, computes and returns the distance value corresponding to that transformation.

Shapes are given as PolylineSet.

2.1.2.6 double EuProfi::AbstractMatcher::distance (argument_type_2 *shape2*, argument_type_2 *shape1*)

Finds the best transformation mapping shape2 to shape1, computes and returns the distance value corresponding to that transformation.

Shapes are given as PolylineSet2.

2.1.2.7 void EuProfi::AbstractMatcher::setFirst (argument_type_s *shape1*)

Presets a shape to be compared by consecutive calls of matchToFixedFirst(shape2).

Shape is read from a file with name given by shape1.

2.1.2.8 void EuProfi::AbstractMatcher::setFirst (argument_type *shape1*)

Presets a shape to be compared by consecutive calls of matchToFixedFirst(shape2).

Shape is given as PolylineSet.

2.1.2.9 void EuProfi::AbstractMatcher::setFirst (argument_type_2 *shape1*)

Presets a shape to be compared by consecutive calls of matchToFixedFirst(shape2).

Shape is given as PolylineSet2.

2.1.2.10 MatchingResult EuProfi::AbstractMatcher::matchToFixedFirst (argument_type_s *shape2*)

Finds and returns the best transformation mapping shape2 to the predefined shape, along with the distance value corresponding to the transformation.

The shape is read from a file whose name is given by shape2. The first shape should be set previously via call setFirst(shape1).

2.1.2.11 MatchingResult EuProfi::AbstractMatcher::matchToFixedFirst (argument_type *shape2*)

Finds and returns the best transformation mapping shape2 to the predefined shape, along with the distance value corresponding to the transformation.

The shape is given as PolylineSet. The first shape should be set previously via call setFirst(shape1).

2.1.2.12 MatchingResult EuProfi::AbstractMatcher::matchToFixedFirst (argument_type_2 *shape2*)

Finds and returns the best transformation mapping shape2 to the predefined shape, along with the distance value corresponding to the transformation.

The shape is given as PolylineSet2. The first shape should be set previously via call setFirst(shape1).

2.1.2.13 double EuProfi::AbstractMatcher::distanceToFixedFirst (argument_type_s *shape2*)

Finds the best transformation mapping shape2 to the predefined shape, computes and returns the distance value corresponding to the transformation.

The shape is read from a file with name given by shape2. The first shape should be set previously via call setFirst(shape1).

2.1.2.14 double EuProfi::AbstractMatcher::distanceToFixedFirst (argument_type *shape2*)

Finds the best transformation mapping shape2 to the predefined shape, computes and returns the distance value corresponding to the transformation.

The shape is given as PolylineSet. The first shape should be set previously via call setFirst(shape1).

2.1.2.15 double EuProfi::AbstractMatcher::distanceToFixedFirst (argument_type_2 *shape2*)

Finds the best transformation mapping shape2 to the predefined shape, computes and returns the distance value corresponding to the transformation.

The shape is given as PolylineSet2. The first shape should be set previously via call setFirst(shape1).

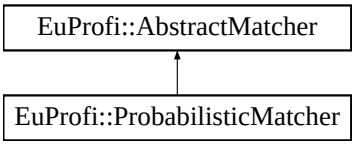
2.1.2.16 void EuProfi::AbstractMatcher::clearFirst ()

Resets the predefined shape.

2.2 EuProfi::ProbabilisticMatcher Class Reference

`#include <probabilisticMatcher.h>`

Inheritance diagram for EuProfi::ProbabilisticMatcher::



2.2.1 Detailed Description

This class implements the probabilistic algorithm for complete matching as described in deliverable D4.3 section 2.2.

The distance function used to evaluate the quality of match is described in deliverable D4.3 section 2.3.

For usage and method signature see documentation of class AbstractMatcher.

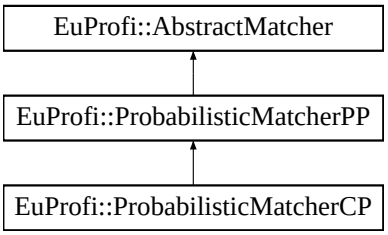
See also:

AbstractMatcher

2.3 EuProfi::ProbabilisticMatcherPP Class Reference

`#include <probabilisticMatcher.h>`

Inheritance diagram for EuProfi::ProbabilisticMatcherPP::



2.3.1 Detailed Description

This class implements the probabilistic algorithm for partial matching as described in deliverable D5.3 section 2.2.

The distance function used to evaluate the quality of match is described in deliverable D5.3 section 2.3. The current implementation of the distance function follows the approach of considering only the parts of the shapes that have a correspondence in the other shape for valuating the quality of match and then weighting the result with the relative size of the matched parts.

The full matching result contains the transformation which yields the best match, the distance value corresponding to that transformation, the one-sided resemblances of the matched parts of the two shapes, the relative size of the matched parts and the matched parts of the shapes.

See also:

AbstractMatcher

Public Types

- typedef SimilarityInfoP **full_result_type**
- typedef std::vector< full_result_type > **results_type**

Public Member Functions

- full_result_type match (argument_type_s shape2, argument_type_s shape1)
- full_result_type match (argument_type shape2, argument_type shape1)
- full_result_type match (argument_type_2 shape2, argument_type_2 shape1)
- full_result_type matchToFixedFirst (argument_type_s shape2)
- full_result_type matchToFixedFirst (argument_type shape2)
- full_result_type matchToFixedFirst (argument_type_2 shape2)
- results_type getResults ()

2.3.2 Member Function Documentation

2.3.2.1 PartialMatchingResult EuProfi::ProbabilisticMatcherPP::match (argument_type_s *shape2*, argument_type_s *shape1*)

Finds and returns the best transformation mapping shape2 to shape1, along with the distance value corresponding to that transformation.

Shape data are read from files with names given by shape1 and shape2.

2.3.2.2 PartialMatchingResult EuProfi::ProbabilisticMatcherPP::match (argument_type *shape2*, argument_type *shape1*)

Finds and returns the best transformation mapping shape2 to shape1, along with the distance value corresponding to that transformation.

Shapes are given as PolylineSet.

2.3.2.3 PartialMatchingResult EuProfi::ProbabilisticMatcherPP::match (argument_type_2 *shape2*, argument_type_2 *shape1*)

Finds and returns the best transformation mapping shape2 to shape1, along with the distance value corresponding to that transformation.

Shapes are given as PolylineSet2.

2.3.2.4 PartialMatchingResult EuProfi::ProbabilisticMatcherPP::matchToFixedFirst (argument_type_s *shape2*)

Finds and returns the best transformation mapping shape2 to predefined shape, along with the distance value corresponding to the transformation.

Shape is read from a file with name given by shape2. The first shape should be set previously via call setFirst(shape1).

2.3.2.5 PartialMatchingResult EuProfi::ProbabilisticMatcherPP::matchToFixedFirst (argument_type *shape2*)

Finds and returns the best transformation mapping shape2 to predefined shape, along with the distance value corresponding to the transformation.

Shape is given as PolylineSet. The first shape should be set previously via call setFirst(shape1).

2.3.2.6 PartialMatchingResult EuProfi::ProbabilisticMatcherPP::matchToFixedFirst (argument_type_2 *shape2*)

Finds and returns the best transformation mapping shape2 to predefined shape, along with the distance value corresponding to the transformation.

Shape is given as PolylineSet2. The first shape should be set previously via call setFirst(shape1).

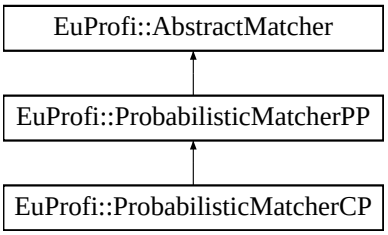
2.3.2.7 PartialMatchingResults EuProfi::ProbabilisticMatcherPP::getResults ()

Returns all candidate matches found by the last algorithm run, that is, one of the following should be called first: match(), distance(), matchToFixedFirst(), or distanceToFixedFirst().

2.4 EuProfi::ProbabilisticMatcherCP Class Reference

`#include <probabilisticMatcher.h>`

Inheritance diagram for EuProfi::ProbabilisticMatcherCP::



2.4.1 Detailed Description

This class implements the probabilistic algorithm for complete-partial probabilistic matching as described in deliverable D5.3 section 2.2.

The distance function indicates how good the complete first shape is matched to some parts of the second shape.

The full matching result contains the transformation yielding the best complete-partial match, the distance value corresponding to that transformation and the one-sided resemblance values for both shapes.

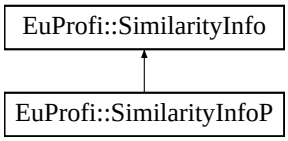
See also:

AbstractMatcher

2.5 EuProfi::SimilarityInfo Struct Reference

`#include <probabilisticMatcher.h>`

Inheritance diagram for EuProfi::SimilarityInfo::



2.5.1 Detailed Description

Structure SimilarityInfo contains the matching result for complete-complete or complete-partial matching, which is a best transformation along with the distance value, corresponding to the transformation.

The transformation can in general be an affine transformation, though typically a similarity transformation is used.

Public Member Functions

- double dist () const

Public Attributes

- double distance
- AffineTfm atm

2.5.2 Member Function Documentation

2.5.2.1 double EuProfi::SimilarityInfo::dist () const [inline]

Returns the distance between the matched shapes.

2.5.3 Member Data Documentation

2.5.3.1 double EuProfi::SimilarityInfo::distance

Distance between the matched shapes.

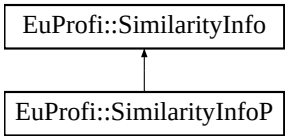
2.5.3.2 AffineTfm EuProfi::SimilarityInfo::atm

Transformation yielding the best match.

2.6 EuProfi::SimilarityInfoP Struct Reference

`#include <probabilisticMatcher.h>`

Inheritance diagram for EuProfi::SimilarityInfoP::



2.6.1 Detailed Description

Structure SimilarityInfoP contains the result of the partial matching.

It contains the best found transformation and the distance corresponding to that transformation.

In case of partial-partial matching it also contains the information about the relative size of the matched parts of each shape and the one-sided similarity values of the matched parts corresponding to the transformation. Additionally the matched parts of the shapes are returned as sets of polygonal curves. The distance value of the match depends on the distances between the matched parts and on the relative size of the matched parts.

In case of complete-partial matching the ratio-values and the matched parts are unset. The one-sided similarity values are computed for both shapes. The distance value indicates how good the complete first shape is matched to some parts of the second shape.

The transformation can in general be an affine transformation, though typically a similarity transformation is used.

Public Attributes

- double ratioFirst
- double ratioSecond
- double simFirst
- double simSecond

- PolylineSet partsFirst
- PolylineSet partsSecond

2.6.2 Member Data Documentation

2.6.2.1 double EuProfi::SimilarityInfoP::ratioFirst

Relative size of the matched parts of the first shape.

This value is not set in the result of complete-partial matching.

2.6.2.2 double EuProfi::SimilarityInfoP::ratioSecond

Relative size of the matched parts of the second shape.

This value is not set in the result of complete-partial matching.

2.6.2.3 double EuProfi::SimilarityInfoP::simFirst

One-sided similarity value between the matched parts of the first shape and the matched parts of the second shape.

2.6.2.4 double EuProfi::SimilarityInfoP::simSecond

One-sided similarity value between the matched parts of the second shape and the matched parts of the first shape.

2.6.2.5 PolylineSet EuProfi::SimilarityInfoP::partsFirst

Matched parts of the first shape.

This value is not set in the result of complete-partial matching.

2.6.2.6 PolylineSet EuProfi::SimilarityInfoP::partsSecond

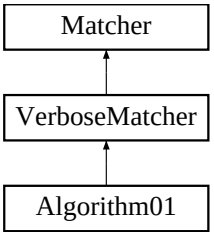
Matched parts of the second shape.

This value is not set in the result of complete-partial matching.

2.7 Matcher Class Reference

`#include <matcher.h>`

Inheritance diagram for Matcher::



2.7.1 Detailed Description

Base class for algorithms that match 2 sets of (polygonal) curves.

The procedure is as follows: For 2 sets S_2 and S_1 of polygonal curves, transformations t^i are computed that map S_2 to S_1 and the mapped sets $t^i(S_2)$ and S_1 are compared using a location dependant distance measure. The distance is defined as the minimum value of $t_2^*(S_2)$ and S_1 . Typical usage: Calling of setFixed(s1) to determine the first shape S_1 once. Callings of getDistance(si) to get the distance of $t_i^*(S_i)$ and S_1 for arbitrarily many sets $S_2...S_n$.

Public Member Functions

- Matcher (const Shape ∗s2, const Shape ∗s1, DistanceMeasure ∗dm=0)
- Matcher (DistanceMeasure ∗dm=0)
- virtual void setShapes (const Shape ∗s2, const Shape ∗s1)
- void setFixed (const Shape ∗s1)
- WeightedTransformation ∗ result (const Shape ∗s, double ∗distance=0, double ∗s1=0, double ∗s2=0)
- bool **hasDistanceMeasure** ()
- virtual void run ()
- virtual WeightedTransformation ∗ nextResult (double ∗distance=0, bool useArray=true)
- virtual bool hasNextResult (bool useArray=true)
- virtual void resetResultIterator ()
- WeightedTransformation ∗ result (double ∗distance=0, double ∗s1=0, double ∗s2=0)
- bool hasVerboseOption ()
- virtual vector< Transformation > ∗ getTransformationSpace ()
- double getDistance (WeightedTransformation ∗t, double ∗s1=0, double ∗s2=0)
- Shape **getMatchedParts** (WeightedTransformation ∗t, unsigned int shape_index)
- double **getMatchedRatio** (WeightedTransformation ∗t, unsigned int shape_index)

Protected Member Functions

- virtual void **setShape** (const Shape ∗s, unsigned int index)

2.7.2 Constructor & Destructor Documentation

2.7.2.1 Matcher::Matcher (const Shape ∗ *s2*, const Shape ∗ *s1*, DistanceMeasure ∗ *dm* = 0) [inline]

A Matcher for the two shapes S_2 and S_1 using DistanceMeasure dm.

2.7.2.2 Matcher::Matcher (DistanceMeasure ∗ *dm* = 0) [inline]

A Matcher using DistanceMeasure dm.

2.7.3 Member Function Documentation

2.7.3.1 virtual void Matcher::setShapes (const Shape ∗ *s2*, const Shape ∗ *s1*) [inline, virtual]

Determining the two shapes S_2 and S_1 to be matched.

2.7.3.2 void Matcher::setFixed (const Shape * *s1*) [inline]

Sets the fixed shape S_1 .

2.7.3.3 WeightedTransformation* Matcher::result (const Shape * *s*, double * *distance* = 0, double * *s1* = 0, double * *s2* = 0) [inline]

Returns the best mapping t^* computed by the matching algorithm for the given shape $t_2^*(S_2)$ and the fixed shape S_1.

2.7.3.4 virtual void Matcher::run () [inline, virtual]

Runs the matching algorithm that finds one or many candidate transformations.

Reimplemented in Algorithm01.

2.7.3.5 virtual WeightedTransformation* Matcher::nextResult (double * *distance* = 0, bool *useArray* = true) [inline, virtual]

Returns the next candidate transformation.

Function to iterate over the best mappings computed by the matching algorithm. Result hast to be deleted manually.

Reimplemented in Algorithm01.

2.7.3.6 virtual bool Matcher::hasNextResult (bool *useArray* = true) [inline, virtual]

Whether there is a next candidate transformation.

Function to iterate over the best mappings computed by the matching algorithm.

Reimplemented in Algorithm01.

2.7.3.7 virtual void Matcher::resetResultIterator () [inline, virtual]

Goes back to the first candidate transformation.

Functions to iterate over the best mappings computed by the matching algorithm.

Reimplemented in Algorithm01.

2.7.3.8 WeightedTransformation * Matcher::result (double * *distance* = 0, double * *s1* = 0, double * *s2* = 0)

Returns the best mapping t^* computed by the matching algorithm and the distance between the two shapes.

Pointer has to be deleted manually.

2.7.3.9 bool Matcher::hasVerboseOption () [inline]

Returns true if this is a VerboseMatcher.

2.7.3.10 virtual vector<Transformation>* Matcher::getTransformationSpace () [inline, virtual]

Returns a vector containing all transformations that have been collected.

Pointer has to be deleted manually.

Reimplemented in Algorithm01.

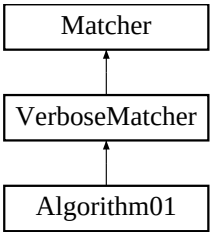
2.7.3.11 double Matcher::getDistance (WeightedTransformation * *t*, double * *s1* = 0, double * *s2* = 0)

The distance of $t(S_2)$ and S_1 for a given transformation t using a preset DistanceMeasure.

2.8 VerboseMatcher Class Reference

`#include <verbose_matcher.h>`

Inheritance diagram for VerboseMatcher::



2.8.1 Detailed Description

Base class for Matcher classes that provide a verbose option.

Public Member Functions

- VerboseMatcher (const Shape *s1, const Shape *s2, DistanceMeasure *dm)
- VerboseMatcher (DistanceMeasure *dm=0)
- virtual bool **step** ()=0
- virtual Transformation * **peek** ()=0
- virtual int **remaining_steps** ()=0
- virtual bool **finished** ()=0

2.8.2 Constructor & Destructor Documentation

2.8.2.1 VerboseMatcher::VerboseMatcher (const Shape * *s1*, const Shape * *s2*, DistanceMeasure * *dm*) [inline]

A VerboseMatcher for the two shapes s1 and s2 using DistanceMeasure dm.

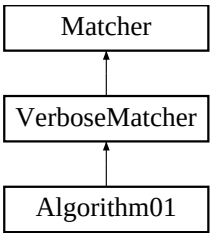
2.8.2.2 VerboseMatcher::VerboseMatcher (DistanceMeasure * *dm* = 0) [inline]

A VerboseMatcher using DistanceMeasure dm.

2.9 Algorithm01 Class Reference

`#include <algorithm01.h>`

Inheritance diagram for Algorithm01::



2.9.1 Detailed Description

Algorithm for matching 2 sets of (polygonal) curves.

As described in deliverable D4.3 section 2.2, for 2 sets S_1 and S_2 of polygonal curves, transformations t_i are computed that map S_2 to S_1 and the mapped sets $t^i(S_2)$ and S_1 are compared yielding a value of similarity in the range $[0, 1]$. The distance between the shapes is derived from the similarity value. The class of allowable transformations is specified by the type of the MapFinder passed to the constructor.

Public Member Functions

- Algorithm01 (Shape ∗s2, Shape ∗s1, MapFinder ∗f, DistanceMeasure ∗dm=0, int seeds=0)
- Algorithm01 (MapFinder ∗f, DistanceMeasure ∗dm=0, int seeds=0)
- void setSeeds (unsigned int seeds)
- void run ()
- WeightedTransformation ∗ nextResult (double ∗distance=0, bool useArray=true)
- bool hasNextResult (bool useArray=true)
- void resetResultIterator ()
- int remaining_steps ()
- vector< Transformation > ∗ getTransformationSpace ()

2.9.2 Constructor & Destructor Documentation

2.9.2.1 Algorithm01::Algorithm01 (Shape ∗ *s2*, Shape ∗ *s1*, MapFinder ∗ *f*, DistanceMeasure ∗ *dm* = 0, int *seeds* = 0) [inline]

Object computing similarity of 2 given sets of line segments.

Parameters:

f The object computing the best transformation. The class of allowable transformations is determined by f.

2.9.2.2 Algorithm01::Algorithm01 (MapFinder * *f*, DistanceMeasure * *dm* = 0, int *seeds* = 0) [inline]

Object computing similarity of 2 sets of line segments.

The sets of line segments have to be specified using Matcher::setShapes() or Matcher::setFixed().

Parameters:

f The object computing the best transformation. The class of allowable transformations is determined by f.

2.9.3 Member Function Documentation

2.9.3.1 void Algorithm01::setSeeds (unsigned int *seeds*) [inline]

Sets the number of random experiments.

2.9.3.2 void Algorithm01::run () [inline, virtual]

Runs the matching algorithm that finds one or many candidate transformations.

Reimplemented from Matcher.

2.9.3.3 WeightedTransformation * Algorithm01::nextResult (double * *distance* = 0, bool *useArray* = true) [virtual]

Returns the next candidate transformation.

Function to iterate over the best mappings computed by the matching algorithm. Result hast to be deleted manually.

Reimplemented from Matcher.

2.9.3.4 bool Algorithm01::hasNextResult (bool *useArray* = true) [inline, virtual]

Whether there is a next candidate transformation.

Function to iterate over the best mappings computed by the matching algorithm.

Reimplemented from Matcher.

2.9.3.5 void Algorithm01::resetResultIterator () [inline, virtual]

Goes back to the first candidate transformation.

Functions to iterate over the best mappings computed by the matching algorithm.

Reimplemented from Matcher.

2.9.3.6 int Algorithm01::remaining_steps () [inline, virtual]

Returns the number of remaining random experiments.

See also:

Matcher

Implements VerboseMatcher.

2.9.3.7 vector<Transformation>* Algorithm01::getTransformationSpace () [inline, virtual]

Returns a vector containing all transformations that have been collected.

Pointer has to be deleted manually.

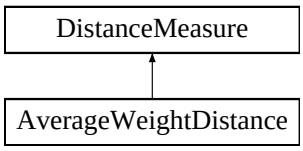
Reimplemented from Matcher.

3 Shape Similarity Computation

3.1 DistanceMeasure Class Reference

`#include <distance_measure.h>`

Inheritance diagram for DistanceMeasure::



3.1.1 Detailed Description

Base class for distance measures that compare 2 sets of (polygonal) curves.

The procedure is as follows: Calling of setFixed(s1) to determine the first shape S_1 once. Callings of getDistance(si) to get the distance of S_1 to arbitrarily many sets $S_2 ... S_n$.

Public Types

- **COMPLETE**
- **PARTIAL**
- **COMPLETE_PARTIAL**
- enum mode { **COMPLETE**, **PARTIAL**, **COMPLETE_PARTIAL** }

Public Member Functions

- virtual void setFixed (Shape *s_1)=0
- virtual double getDistance (Shape *s_2, Shape *s_1, WeightedTransformation *t, double *s1=0, double *s2=0)=0
- virtual mode getMode () const =0

3.1.2 Member Enumeration Documentation

3.1.2.1 enum DistanceMeasure::mode

Whether complete or partial distance is to be computed.

3.1.3 Member Function Documentation

3.1.3.1 virtual void DistanceMeasure::setFixed (Shape * *s_1*) [pure virtual]

Sets the fixed shape S_1 .

Implemented in AverageWeightDistance.

3.1.3.2 virtual double DistanceMeasure::getDistance (Shape * *s_2*, Shape * *s_1*, WeightedTransformation * *t*, double * *s1* = 0, double * *s2* = 0) [pure virtual]

The distance of $t_2^*(S_2)$ and S_1 .

Implemented in AverageWeightDistance.

3.1.3.3 virtual mode DistanceMeasure::getMode () const [pure virtual]

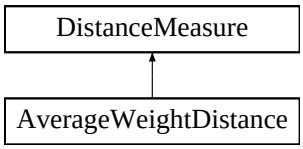
Whether complete or partial or complete partial distance is computed.

Implemented in AverageWeightDistance.

3.2 AverageWeightDistance Class Reference

`#include <average_weight_distance.h>`

Inheritance diagram for AverageWeightDistance::



3.2.1 Detailed Description

Distance measure for sets of line segments.

This class is actually used to compute a value of similarity for 2 sets of line segments.

Public Member Functions

- ShapeCorrelation * collectCorrelations (unsigned int from, unsigned int to, EdgeCorrelation *type)
- double computeWeight (unsigned int from, unsigned int to)
- void **init** (Shape *s_right, unsigned int at)
- **AverageWeightDistance** (mode m=COMPLETE)
- void setFixed (Shape *s_right)
- double getDistance (Shape *s_left, Shape *s_right, WeightedTransformation *t, double *s1=0, double *s2=0)
- mode getMode () const

3.2.2 Member Function Documentation

3.2.2.1 AverageWeightDistance::ShapeCorrelation * AverageWeightDistance::collectCorrelations (unsigned int *from*, unsigned int *to*, EdgeCorrelation * *type*)

Collect Correlations between two shapes or of one shape to itself.

3.2.2.2 double AverageWeightDistance::computeWeight (unsigned int *from*, unsigned int *to*)

Compute a value that tells how good one shape is represented by another shape.

3.2.2.3 void AverageWeightDistance::setFixed (Shape * *s_right*) [virtual]

Sets the fixed shape S_1 .

Implements DistanceMeasure.

3.2.2.4 double AverageWeightDistance::getDistance (Shape * *s_left*, Shape * *s_right*, WeightedTransformation * *t*, double * *s1* = 0, double * *s2* = 0) [inline, virtual]

The distance of $t_2^*(S_2)$ and S_1 .

Implements DistanceMeasure.

3.2.2.5 mode AverageWeightDistance::getMode () const [inline, virtual]

Whether complete or partial or complete partial distance is computed.

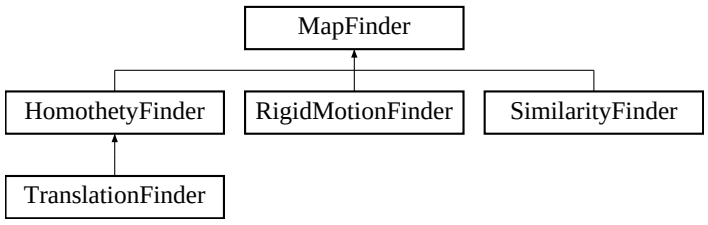
Implements DistanceMeasure.

4 Transformation Computation

4.1 MapFinder Class Reference

`#include <map_finder.h>`

Inheritance diagram for MapFinder::



4.1.1 Detailed Description

Base class for algorithms that compute a transformation mapping 2 sets (sequences) of weighted points.

Instances of deriving classes are used by Algorithm01.

Public Member Functions

- void reset ()
- Preferences * **prefs** ()
- void add (Sample s)
- void add (Sample s, IteratorPosition pos1, IteratorPosition pos2)
- void addFirst (Sample s, IteratorPosition pos1, IteratorPosition pos2)
- void consolidate (bool turnAround=false)
- virtual evaluation evaluateCurrentMapping ()=0
- WeightedTransformation * result ()
- virtual bool useScaling () const
- void setScalingFactor (double scale)
- virtual bool useReflexion () const
- void setReflexion (bool on)
- Shape * toShape ()

Classes

- struct Sample

  Weighted pair of points.

4.1.2 Member Function Documentation

4.1.2.1 void MapFinder::reset () [inline]

Resets to initial state.

4.1.2.2 void MapFinder::add (Sample *s*)

Adds a sample.

4.1.2.3 void MapFinder::add (Sample *s*, IteratorPosition *pos1*, IteratorPosition *pos2*)

Adds a sample and sets the current IteratorPosition.

4.1.2.4 void MapFinder::addFirst (Sample *s*, IteratorPosition *pos1*, IteratorPosition *pos2*)

Adds the first sample and sets the current IteratorPosition.

4.1.2.5 void MapFinder::consolidate (bool *turnAround* = false)

Goes back to the best state so far and puts the first sample at the end of the sequence, if turnAround == true.

4.1.2.6 virtual evaluation MapFinder::evaluateCurrentMapping () [pure virtual]

Computes the transformation according to the matching samples so far.

Whether it is worth to collect more samples in the current direction (according to the error of the last pair in the sequences) is returned.

4.1.2.7 WeightedTransformation * MapFinder::result ()

Returns the best mapping computed so far.

The returned WeightedTransformation has to be deleted manually.

4.1.2.8 virtual bool MapFinder::useScaling () const [inline, virtual]

Returns true iff the using algorithm should use scalings.

Behavoiur depends on the transformation type aimed for. Deriving classes can overwrite this function.

Reimplemented in HomothetyFinder, SimilarityFinder, and TranslationFinder.

4.1.2.9 void MapFinder::setScalingFactor (double *scale*) [inline]

Sets a scale, iff useScaling() returns true.

4.1.2.10 virtual bool MapFinder::useReflexion () const [inline, virtual]

Returns true iff the using algorithm should use reflection.

Behavoiur depends on the transformation type aimed for.

Reimplemented in SimilarityFinder.

4.1.2.11 void MapFinder::setReflexion (bool *on*) [inline]

If samples were mirrored before adding, the MapFinder is to be told to take that reflexion into account.

4.1.2.12 Shape * MapFinder::toShape ()

Returns the samples so far in form of a shape.

4.2 MapFinder::Sample Struct Reference

`#include <map_finder.h>`

4.2.1 Detailed Description

Weighted pair of points.

Public Member Functions

- **Sample** (Point s, Point t, double w=0)

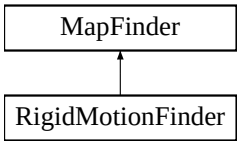
Public Attributes

- Point **p** [2]
- double **weight**

4.3 RigidMotionFinder Class Reference

`#include <rigid_motion_finder.h>`

Inheritance diagram for RigidMotionFinder::



4.3.1 Detailed Description

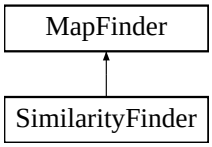
A MapFinder for computing rigid motions (translations + rotations).

The transformation computed is the one that minimizes the sum of the weighted squared errors.

4.4 SimilarityFinder Class Reference

`#include <similarity_finder.h>`

Inheritance diagram for SimilarityFinder::



4.4.1 Detailed Description

A MapFinder for computing similarities (translations + rotations + scalings).

The transformation computed is the one that minimizes the sum of the weighted squared errors.

Public Member Functions

- bool useScaling () const
- bool useReflexion () const

4.4.2 Member Function Documentation

4.4.2.1 bool SimilarityFinder::useScaling () const [inline, virtual]

Returns true.

Reimplemented from MapFinder.

4.4.2.2 bool SimilarityFinder::useReflexion () const [inline, virtual]

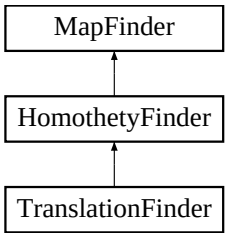
Returns true.

Reimplemented from MapFinder.

4.5 HomothetyFinder Class Reference

`#include <homothety_finder.h>`

Inheritance diagram for HomothetyFinder::



4.5.1 Detailed Description

A MapFinder for computing homotheties (translations + scalings).

The transformation computed is the one that minimizes the sum of the weighted squared errors.

Public Member Functions

- void **initHomothetyFinder** ()
- bool useScaling () const

4.5.2 Member Function Documentation

4.5.2.1 bool HomothetyFinder::useScaling () const [inline, virtual]

Returns true.

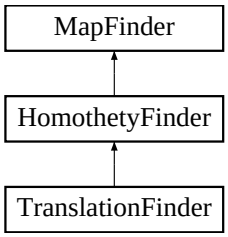
Reimplemented from MapFinder.

Reimplemented in TranslationFinder.

4.6 TranslationFinder Class Reference

`#include <translation_finder.h>`

Inheritance diagram for TranslationFinder::



4.6.1 Detailed Description

A MapFinder for computing translations.

The transformation computed is the one that minimizes the sum of the weighted squared errors.

Public Member Functions

- bool useScaling () const

4.6.2 Member Function Documentation

4.6.2.1 bool TranslationFinder::useScaling () const [inline, virtual]

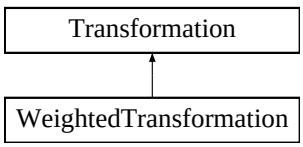
Returns false.

Reimplemented from HomothetyFinder.

4.7 WeightedTransformation Class Reference

`#include <weighted_transformation.h>`

Inheritance diagram for WeightedTransformation::



4.7.1 Detailed Description

Weighted Transformation with additional information.

Public Member Functions

- **WeightedTransformation** (Transformation t, double w=0)
- **WeightedTransformation** (const WeightedTransformation &t)
- double **getWeight** () const
- void **setWeight** (double w)

5 Clustering

5.1 Clustering Class Reference

`#include <clustering.h>`

5.1.1 Detailed Description

Clustering for weighted points in metric space.

As described in deliverable D4.3 section 2.2.5, the space is devided up into regions managed by a tree structure. This class is used by Algorithm01 to cluster the weigthed transformations.

Public Member Functions

- void clear ()
- double performClustering ()
- bool enqueue (WeightedTransformation *m)
- const vector< Cluster * > & result () const
- vector< Transformation > * getAllItems ()

Classes

- class Cluster

  Cluster subsuming points with small pairwise distance.

- struct **Node**

  Node in clustering tree.

5.1.2 Member Function Documentation

5.1.2.1 void Clustering::clear ()

Returns to the initial state.

5.1.2.2 double Clustering::performClustering ()

Performs the clustering.

The scaling factor of the transformation defining the best cluster is returned.

Returns:

Scaling factor of best cluster's centroid transformation.

5.1.2.3 bool Clustering::enqueue (WeightedTransformation * *m*)

Enqueue an item to be integrated in the clustering tree the next time performClustering() is called.

5.1.2.4 const vector<Cluster*>& Clustering::result () const [inline]

Returns the clusters in descending order, according to their weights.

5.1.2.5 vector<Transformation>* Clustering::getAllItems () [inline]

Returns an iterator to traverse all transformations managed by this clustering.

5.2 Clustering::Cluster Class Reference

`#include <clustering.h>`

5.2.1 Detailed Description

Cluster subsuming points with small pairwise distance.

The cluster's centroid is a heavyweigthed point and the cluster contains the points which have small distance to that centroid. Points with small distance to the centroids of more than one cluster are managed by a distinct cluster, but may vote for several clusters.

Public Member Functions

- Cluster (WeightedTransformation *m, Clustering *clustering, Node *node=0)
- const WeightedTransformation * centroid () const
- int size ()
- int wholeSize ()
- WeightedTransformation * item (int index)
- double weight ()
- WeightedTransformation * **result** (bool get_ranges=true)

5.2.2 Constructor & Destructor Documentation

5.2.2.1 Clustering::Cluster::Cluster (WeightedTransformation * *m*, Clustering * *clustering*, Node * *node* = 0)

New cluster with centroid m.

5.2.3 Member Function Documentation

5.2.3.1 const WeightedTransformation* Clustering::Cluster::centroid () const [inline]

The centroid of the cluster.

5.2.3.2 int Clustering::Cluster::size () [inline]

Number of items managed by this Cluster.

5.2.3.3 int Clustering::Cluster::wholeSize () [inline]

Number of items voting for this cluster .

5.2.3.4 WeightedTransformation* Clustering::Cluster::item (int *index*) [inline]

Returns the item specified by the given index.

5.2.3.5 double Clustering::Cluster::weight () [inline]

Returns the sum of the weights of all items voting for this cluster.

6 Geometric Objects and Transformations

6.1 Point Class Reference

`#include <point.h>`

6.1.1 Detailed Description

Representation of a 2-dimensional point.

Public Member Functions

- Point (double x=0, double y=0)
- Point operator+ (const Point &other) const
- Point operator- (const Point &other) const
- double operator * (const Point &other) const
- double sqrDist (const Point &other) const
- double sqrDist (const Point &p1, const Point &p2) const
- Point getScaled (const Point &p, double x) const
- Point xInBetween (const Point &p, double x) const
- double getX () const
- double getY () const
- void setX (double x)
- void setY (double y)
- void setCoordinates (double *coord)
- bool operator== (const Point &other)

Friends

- Point operator * (double scale, const Point &p)
- ostream & operator<< (ostream &output, const Point &p)
- ifstream & operator>> (ifstream &input, Point &p)

6.1.2 Constructor & Destructor Documentation

6.1.2.1 Point::Point (double *x* = 0, double *y* = 0)

Constructs a point with coordinates (x,y).

Default Point has x and y set to 0.

Parameters:

x The x-coordinate.
y The y-coordinate.

6.1.3 Member Function Documentation

6.1.3.1 Point Point::operator+ (const Point & *other*) const

Plus operator.

Returns a Point resulting from the vector addition of two points.

Parameters:

other The point to add.

Returns:

```
(*this) + other
```

6.1.3.2 Point Point::operator- (const Point & *other*) const

Minus operator.

Returns a Point resulting from the vector subtraction of two points.

Parameters:

other The point to subtract.

Returns:

```
(*this) - other
```

6.1.3.3 double Point::operator ∗ (const Point & *other*) const

Inner product operator.

Returns a double resulting from the inner product operation.

Parameters:

other The point to multiply.

Returns:

```
<(*this),other>
```

6.1.3.4 double Point::sqrDist (const Point & *other*) const

Squared euclidean distance to an other point.

Parameters:

other The other point.

Returns:

```
<(*this)-other,(*this)-other>
```

6.1.3.5 double Point::sqrDist (const Point & *p1*, const Point & *p2*) const

Squared euclidean distance to a line segment.

Parameters:

p1 First point defining the line segment.

p2 Second point defining the line segment.

Returns:

The squared distance of this point to the nearest point of the line segment defined by `p1` and `p2` .

6.1.3.6 Point Point::getScaled (const Point & *p*, double *x*) const [inline]

Computes a Point on a line through this point and point p.

The resulting Point will be element of the line going through this point Q and the given point p with distance $x \cdot ||Q - p||$ to Q in direction $Q - p$, that is a point $(1 + x)Q - xp$. For $x \in [-1, 0]$ the returned point lies in a segment $s = \overline{pQ}$.

6.1.3.7 Point Point::xInBetween (const Point & *p*, double *x*) const

Computes a Point on a line segment.

The resulting Point will be element of the line segment formed by this point Q and the given point p. The point is computed as $(1 - x)p + xQ$ if $x > 0$, and as $(1 - |x|)Q + |x|p$ otherwise.

Precondition:

```
x > -1 && x < 1 && x != 0
```

6.1.3.8 double Point::getX () const [inline]

The x-coordinate.

Returns:

The x-coordinate.

6.1.3.9 double Point::getY () const [inline]

The y-coordinate.

Returns:

The y-coordinate.

6.1.3.10 void Point::setX (double *x*) [inline]

Sets the x-coordinate.

Parameters:

x The x-coordinate.

6.1.3.11 void Point::setY (double *y*) [inline]

Sets the y-coordinate.

Parameters:

y The y-coordinate.

6.1.3.12 void Point::setCoordinates (double * *coord*) [inline]

Sets the x and y-coordinates to coord[0] and coord[1] respectively.

6.1.3.13 bool Point::operator== (const Point & *other*)

Test for equality of two points.

6.1.4 Friends And Related Function Documentation

6.1.4.1 Point operator * (double *scale*, const Point & *p*) [friend]

Scalar multiplication.

Returns a Point scaled by a given factor.

Parameters:

scale The scalar.

p The point

Returns:

`scale*p`
with `p` treated as a vector.

6.1.4.2 ostream& operator<< (ostream & *output*, const Point & *p*) [friend]

Output streaming operator to stream in files.

Parameters:

output The stream.

p The point.

Returns:

The stream `output`.

6.1.4.3 ifstream& operator>> (ifstream & *input*, Point & *p*) [friend]

Input streaming operator to stream from files.

Parameters:

input The stream.

p The point.

Returns:

The stream `input`.

6.2 Polygon Class Reference

`#include <polygon.h>`

6.2.1 Detailed Description

Representation of a polygonal curve as a sequence of its vertices.

The curve may be closed or open.

Public Member Functions

- Polygon ()
- void addVertex (Point p)
- void setConnected (bool connected)
- void clear ()
- void simplify (float tolerance)
- bool empty () const
- int size () const
- bool connected () const
- const Point & operator[ ] (int index) const

Friends

- class Shape
- ostream & operator<< (ostream &, const Polygon &)
- ifstream & operator>> (ifstream &, Polygon &)

6.2.2 Constructor & Destructor Documentation

6.2.2.1 Polygon::Polygon () `[inline]`

Empty polygon.

6.2.3 Member Function Documentation

6.2.3.1 void Polygon::addVertex (Point *p*) `[inline]`

Add a vertex to the end of the Polygon.

6.2.3.2 void Polygon::setConnected (bool *connected*) `[inline]`

The polygonal curve is treated as a polygon if `connected` is set to true, and as an open curve otherwise.

6.2.3.3 void Polygon::clear () [inline]

Erase all vertices from the Polygon.

6.2.3.4 void Polygon::simplify (float *tolerance*)

Simplifies the Polygon.

Polygon simplification is performed using the Douglas-Peucker algorithm. The resulting polygon approxiamtes the original within the given tolerance bound.

6.2.3.5 bool Polygon::empty () const [inline]

True if polygon has no vertices.

6.2.3.6 int Polygon::size () const [inline]

Returns the number of vertices.

6.2.3.7 bool Polygon::connected () const [inline]

True if the ends of the curve are connected.

6.2.3.8] const Point& Polygon::operator[ ] (int *index*) const [inline]

Random read only access to vertices.

6.2.4 Friends And Related Function Documentation

6.2.4.1 ostream& operator<< (ostream & *output*, const Polygon & *Polygon*) [friend]

Output stream operator.

used to either write Polygon Objects to a file or to print to std::cerr.

print format: $(p_0\ p_1\ p_2\ p_3 \ldots)$: c

where c indicates whether the end vertices are connected (1) or not (0) and p_i is a vertex

6.2.4.2 ifstream& operator>> (ifstream & *input*, Polygon & *Polygon*) [friend]

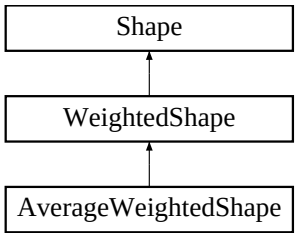
Input stream operator.

used to read a Polygon Object from a File

6.3 Shape Class Reference

`#include <shape.h>`

Inheritance diagram for Shape::



6.3.1 Detailed Description

Representation of a shape as a set of polygonal curves.

The usual way to construct a Shape is to stream from a file.

Public Member Functions

- Shape ()
- Shape (vector< vector< Point > > p)
- Shape (vector< Polygon > p)
- bool empty () const
- int size () const
- int vertices ()
- void update ()
- Point getCenter ()
- const Polygon & operator[ ] (int index) const
- const EdgeLengths & edge_lengths ()
- void clear ()
- const Rectangle & getBBox ()
- void simplify (float tolerance=0)
- Point getRandomPoint ()
- Point getRandomPoint (double t)
- IteratorPosition getIteratorPosition (double x, bool vertex=false)
- IteratorPosition getRandomIteratorPosition (bool vertex=false)
- virtual void apply (const Transformation &t)
- double sqDist (const Transformation t1, const Transformation t2)
- double getRatio (vector< IteratorRange > *range)
- Shape getPartialShape (vector< IteratorRange > *range)

Friends

- ostream & operator<< (ostream &, const Shape &)
- ifstream & operator>> (ifstream &, Shape &)

6.3.2 Constructor & Destructor Documentation

6.3.2.1 Shape::Shape () [inline]

Empty shape.

6.3.2.2 Shape::Shape (vector< vector< Point > > *p*)

Initializes a Shape with `p.size()` polygonal curves, where vertices of the i-th curve are the points of p[i], all polygonal curves are considered to be open.

6.3.2.3 Shape::Shape (vector< Polygon > *p*) [inline]

Initializes a shape with a given set of polygons.

6.3.3 Member Function Documentation

6.3.3.1 bool Shape::empty () const [inline]

True if the set of polygons is empty.

6.3.3.2 int Shape::size () const [inline]

Returns number of polygons.

6.3.3.3 int Shape::vertices () [inline]

Computes the overall number of vertices.

6.3.3.4 void Shape::update () [inline]

Updates the internal structures, such as bounding box and relative edge lengths of the polygons.

6.3.3.5 Point Shape::getCenter () [inline]

Returns the center of the bounding box of the shape.

6.3.3.6] const Polygon& Shape::operator[ ] (int *index*) const [inline]

Random read only access to polygons.

6.3.3.7 const EdgeLengths& Shape::edge_lengths () [inline]

Returns EdgeLengths structure, containing relative lengths of the polygons.

6.3.3.8 void Shape::clear () [inline]

Erases all polygons from the set.

6.3.3.9 const Rectangle& Shape::getBBox () [inline]

Returns the bounding box of the Shape.

6.3.3.10 void Shape::simplify (float *tolerance* = 0)

Simplify the polygons.

simplifies the polygons using the Douglas Peucker algorithm.

See also:

Polygon.simplify()

6.3.3.11 Point Shape::getRandomPoint () [inline]

Returns a random point on the edges of the Shape.

Requires time logarithmic in number of vertices.

6.3.3.12 Point Shape::getRandomPoint (double *t*)

Returns a Point on the Shape with a given offset t.

Requires time logarithmic in number of vertices.

Precondition:

```
t >= 0 && t <= 1
```

6.3.3.13 IteratorPosition Shape::getIteratorPosition (double *x*, bool *vertex* = false)

Returns an IteratorPosition for a VertexIterator or an EdgeIterator for this Shape, that corresponds to the given x.

Precondition:

```
x >= 0 && x <= 1
```

6.3.3.14 IteratorPosition Shape::getRandomIteratorPosition (bool *vertex* = false) [inline]

Function provided for convenience, calls getIteratorPosition() with a random double in [0,1].

6.3.3.15 void Shape::apply (const Transformation & *t*) [virtual]

Applies the given Transformation to the Shape.

updating redundant information like bbox and edge_lengths if necessary.

Reimplemented in AverageWeightedShape.

6.3.3.16 double Shape::sqDist (const Transformation *t1*, const Transformation *t2*) [inline]

Returns square euclidean distance between the images of the vertices of the bounding box under the transformations t1 and t2.

6.3.3.17 double Shape::getRatio (vector< IteratorRange > * *range*)

Returns relative length of the parts of this shape specified by the given range.

6.3.3.18 Shape Shape::getPartialShape (vector< IteratorRange > * *range*)

Returns parts of this shape specified by the given range as a new shape object.

6.3.4 Friends And Related Function Documentation

6.3.4.1 ostream& operator<< (ostream & *output*, const Shape & *s*) [friend]

Output stream operator.

Used to either write Shape Objects to a file or to print to std::cerr. Print format:

0: p_0

1: p_1

2: p_2

...

where p_i are the polygons.

6.3.4.2 ifstream& operator>> (ifstream & *input*, Shape & *s*) [friend]

Input stream operator.

Used to read a Shape Object from a File

6.4 Rectangle Class Reference

`#include <rectangle.h>`

6.4.1 Detailed Description

Axis aligned Rectangle.

Public Member Functions

- double **minX** () const
- double **minY** () const
- double **width** () const
- double **height** () const
- void **minX** (double minX)
- void **minY** (double minY)
- void **width** (double width)
- void **height** (double height)
- double getLength () const
- Rectangle unite (const Rectangle &other) const
- double sqDist (const Transformation &t1, const Transformation &t2) const
- vector< Point > getCorners () const

6.4.2 Member Function Documentation

6.4.2.1 double Rectangle::getLength () const [inline]

Returns maximum value of width and height.

6.4.2.2 Rectangle Rectangle::unite (const Rectangle & *other*) const [inline]

Returns a rectangle corresponding to the bounding box of this and a given rectangle.

6.4.2.3 double Rectangle::sqDist (const Transformation & *t1*, const Transformation & *t2*) const [inline]

Returns the maximum squared euclidean distance between the images of corner points of the rectangle under the transformations t1 and t2.

See also:

Shape::sqDist(const Transformation t1, const Transformation t2)

6.4.2.4 vector<Point> Rectangle::getCorners () const [inline]

Returns the corner points of this rectangle.

6.5 Edge Class Reference

`#include <edge.h>`

6.5.1 Detailed Description

Container class for two end points of an edge.

Public Member Functions

- Edge ()
- Edge (const Point &x, const Point &y)
- Point p ()
- Point q ()

6.5.2 Constructor & Destructor Documentation

6.5.2.1 Edge::Edge () [inline]

Constructs an empty edge with uninitialized end points.

6.5.2.2 Edge::Edge (const Point & *x*, const Point & *y*) [inline]

Constructs an edge with given end points.

6.5.3 Member Function Documentation

6.5.3.1 Point Edge::p () [inline]

Returns the first end point.

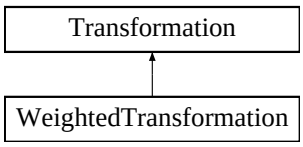
6.5.3.2 Point Edge::q () [inline]

Returns the second end point.

6.6 Transformation Class Reference

`#include <transformation.h>`

Inheritance diagram for Transformation::



6.6.1 Detailed Description

Transformation to be applied on Shapes / Points.

Supported transformations are: translations, rotations, rigid motions, scalings, homotheties, similarities, homomorphisms and general affine transformations. Transformation is represented by a 2x2 Matrix M and a translation vector t. The image of a point p under this transformation is then $M \cdot p + t$.

See also:

Shape

Public Types

- **IDENTITY** = 0
- **TRANSLATION** = 1
- **ROTATION** = 2
- **RIGID_MOTION** = 3
- **SCALING** = 4
- **HOMOTHETY** = 5
- **HOMOMORPHISM** = 6
- **SIMILARITY** = 7
- **AFFINE** = 16
- enum Type {

  IDENTITY = 0, **TRANSLATION** = 1, **ROTATION** = 2, **RIGID_MOTION** = 3,

  SCALING = 4, **HOMOTHETY** = 5, **HOMOMORPHISM** = 6, **SIMILARITY** = 7,

  AFFINE = 16 }

Public Member Functions

- Transformation ()
- Transformation (double s)
- Transformation (RotationMatrix r)
- Transformation (double x, double y)

- Transformation (double s, double x, double y)
- Transformation (double s, RotationMatrix r)
- Transformation (RotationMatrix r, double x, double y)
- Transformation (double s, RotationMatrix r, double x, double y)
- Transformation (const Point &p)
- Transformation (double s, const Point &p)
- Transformation (double s, RotationMatrix r, const Point &p)
- Transformation (vector< double > v, Type type=AFFINE)
- virtual void apply (Point &p) const
- double computeX (const Point &p) const
- double computeY (const Point &p) const
- bool hasTranslation () const
- bool hasRotation () const
- bool hasScaling () const
- bool isAffine () const
- bool empty () const
- double factor () const
- double factor (bool calculate)
- double angle () const
- double angle (bool calculate)
- double getMatrixValue (int i, int j) const
- double getDX () const
- double getDY () const
- double getA () const
- double getB () const
- double sqDist (const Transformation &other, Point p) const
- double sqDist (const Transformation &other, vector< Point > &p) const
- void applyTransformation (const Transformation &t)
- virtual void print () const

Static Public Member Functions

- static Transformation getReflexion ()

Static Public Attributes

- static const Transformation **REFLEXION** = Transformation::getReflexion()

Classes

- class RotationMatrix

  Rotation Matrix, parameter type for Transformation constructors.

6.6.2 Member Enumeration Documentation

6.6.2.1 enum Transformation::Type

Type of Transformation.

6.6.3 Constructor & Destructor Documentation

6.6.3.1 Transformation::Transformation () [inline]

Identity transformation.

6.6.3.2 Transformation::Transformation (double *s*) [inline]

Scaling by factor s.

6.6.3.3 Transformation::Transformation (RotationMatrix *r*) [inline]

Rotation by a given rotation matrix.

6.6.3.4 Transformation::Transformation (double *x*, double *y*) [inline]

Translation by vector $(x, y)^T$.

6.6.3.5 Transformation::Transformation (double *s*, double *x*, double *y*) [inline]

Homothety, that is scaling by factor s and translation by vector $(x, y)^T$.

6.6.3.6 Transformation::Transformation (double *s*, RotationMatrix *r*) [inline]

Homomorphism, that is scaling by factor s and rotation by a given rotation matrix.

6.6.3.7 Transformation::Transformation (RotationMatrix *r*, double *x*, double *y*) [inline]

Rigid motion, that is rotation according to the given matrix and translation by a vector $(x, y)^T$.

6.6.3.8 Transformation::Transformation (double *s*, RotationMatrix *r*, double *x*, double *y*) [inline]

Similarity, that is, scaling by factor s, rotation according to the given matrix, and translation by a vector $(x, y)^T$.

6.6.3.9 Transformation::Transformation (const Point & *p*) [inline]

Translation by position vector of the point p.

6.6.3.10 Transformation::Transformation (double *s*, const Point & *p*) [inline]

Homothety consisting of scaling by factor s and a translation by position vector of the point p.

6.6.3.11 Transformation::Transformation (double *s*, RotationMatrix *r*, const Point & *p*) [inline]

Similarity transformation.

6.6.3.12 Transformation::Transformation (vector< double > *v*, Type *type* = AFFINE) [inline]

Affine transformation given by vector v, the transformation is initialized as $M = \begin{pmatrix} v_1 & v_2 \\ v_3 & v_4 \end{pmatrix}$ and translation vector $t = (v_5, v_6)^T$.

Precondition:

```
v.size() == 6
```

6.6.4 Member Function Documentation

6.6.4.1 virtual void Transformation::apply (Point & *p*) const [inline, virtual]

Apply Transformation to a Point p, the new coordinates of the point p are computed as $M \cdot p + t$.

6.6.4.2 double Transformation::computeX (const Point & *p*) const [inline]

Only compute and return the x coordinate.

6.6.4.3 double Transformation::computeY (const Point & *p*) const [inline]

Only compute and return the y coordinate.

6.6.4.4 bool Transformation::hasTranslation () const [inline]

True if the translation may be part of the transformation.

6.6.4.5 bool Transformation::hasRotation () const [inline]

True if rotation may be part of the transformation.

6.6.4.6 bool Transformation::hasScaling () const [inline]

True if scaling may be part of the transformation.

6.6.4.7 bool Transformation::isAffine () const [inline]

True, if the type of the transformation is set to AFFINE.

6.6.4.8 bool Transformation::empty () const [inline]

True if the type of the transformation is not set.

6.6.4.9 double Transformation::factor () const [inline]

Returns scaling factor.

Precondition:

Transformation may contain scaling.

6.6.4.10 double Transformation::factor (bool *calculate*) [inline]

Returns scaling factor, the factor is calculated from the matrix values if `calculate` is set to `true`.

Precondition:

Transformation may contain scaling.

6.6.4.11 double Transformation::angle () const [inline]

Returns the rotation angle.

Precondition:

Transformation may contain rotation.

6.6.4.12 double Transformation::angle (bool *calculate*) [inline]

Returns the rtation angle, the angle is calculated from the matrix values if `calculate` is set to `true`.

Precondition:

Transformation may contain rotation.

6.6.4.13 double Transformation::getMatrixValue (int *i*, int *j*) const [inline]

Returns matrix entry in the i-th row and j-th column.

6.6.4.14 double Transformation::getDX () const [inline]

Returns the x-component of the translation vector.

6.6.4.15 double Transformation::getDY () const [inline]

Returns the y-component of the translation vector.

6.6.4.16 double Transformation::getA () const [inline]

Returns the value of $\cos\alpha$ for the transformations containing rotation.

Precondition:

Transformation may contain rotation.

6.6.4.17 double Transformation::getB () const [inline]

Returns the value of $-\sin\alpha$ for the transformations containing rotation.

Precondition:

Transformation may contain rotation.

6.6.4.18 double Transformation::sqDist (const Transformation & *other*, Point *p*) const [inline]

Compute the squared distance between the image of Point p under this Transformation and the image of p under other Transformation.

6.6.4.19 double Transformation::sqDist (const Transformation & *other*, vector< Point > & *p*) const [inline]

Compute the squared distance between the image of every point Point of the vector p under this Transformation and the image of the same point under other Transformation, and return the maximum over all points.

6.6.4.20 void Transformation::applyTransformation (const Transformation & *t*) [inline]

Computes a transformation corresponding to applying the given transformation t first, and then this transformation.

The resulting transformation is assigned to this.

6.6.4.21 virtual void Transformation::print () const [inline, virtual]

Print the parameters of the transformation to stderr.

6.6.4.22 static Transformation Transformation::getReflexion () [inline, static]

Returns a transformation corresponding to a reflexion at y-axis with type SIMILARITY.

6.7 Transformation::RotationMatrix Class Reference

#include <transformation.h>

6.7.1 Detailed Description

Rotation Matrix, parameter type for Transformation constructors.

Holds Rotation matrix for a given angle.

Public Member Functions

- RotationMatrix (double angle)
- RotationMatrix (double a, double b)
- double getAngle ()
- bool hasAngle ()
- double getA () const
- double getB () const
- double getC () const
- double getD () const

6.7.2 Constructor & Destructor Documentation

6.7.2.1 Transformation::RotationMatrix::RotationMatrix (double *angle*) [inline]

Initialises a rotation matrix for the given angle as $M = \begin{pmatrix} \cos(angle) & -\sin(angle) \\ \sin(angle) & \cos(angle) \end{pmatrix}$.

6.7.2.2 Transformation::RotationMatrix::RotationMatrix (double *a*, double *b*) [inline]

Initialises a rotation matrix for the given values: $M = \begin{pmatrix} a & b \\ -b & a \end{pmatrix}$, where $a = \cos\alpha$ and $b = -\sin\alpha$ for some angle α.

6.7.3 Member Function Documentation

6.7.3.1 double Transformation::RotationMatrix::getAngle () [inline]

Returns rotation angle.

6.7.3.2 bool Transformation::RotationMatrix::hasAngle () [inline]

True, if the rotation matrix was initialized with a rotation angle.

6.7.3.3 double Transformation::RotationMatrix::getA () const [inline]

Returns value of $\cos\alpha$.

6.7.3.4 double Transformation::RotationMatrix::getB () const [inline]

Returns value of $-\sin\alpha$.

6.7.3.5 double Transformation::RotationMatrix::getC () const [inline]

Returns value of $\sin\alpha$.

6.7.3.6 double Transformation::RotationMatrix::getD () const [inline]

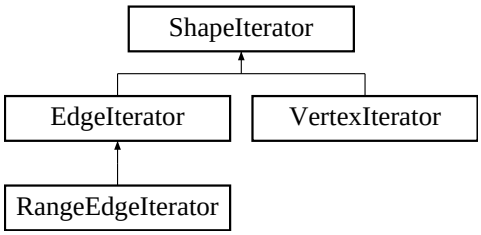
Returns value of $\cos\alpha$.

7 Iterators

7.1 ShapeIterator Class Reference

`#include <shape_iterator.h>`

Inheritance diagram for ShapeIterator::



7.1.1 Detailed Description

Abstract Iterator for Shape.

derived by EdgeIterator and VertexIterator.

ShapeIterator is used by the classes EdgeIterator and VertexIterator serving for their basic functionalties as for example increment and decrement operations.

Behaviour:

- the iterator has a direction. It is changed by calling turnAround() on the iterator, which also swaps the start and current position values.
- the iterator can start at any position of the Shape and will (depending on usage) end at exactly this position setting ShapeIterator::m_finished true.
- if the start Polygon is connected and the iterators mode is POLYGON_WRAP the iterator will wrap the Polygon and be finished at the position started with
- to find out whether the iterator is finished call finished()
- once ShapeIterator::m_finished is set true the increment and decrement operations will take no effect and the iterator stays in the last valid position
- call nextPolygon() to set the iterators position to the beginning of the next Polygon and finished to false if the next Polygon is not the Polygon started with. makes sense in POLYGON_WRAP mode.

Implementation Details:

- a position is stored using two integers. One for the index of the Polygon and one for the index of the Point each in the vector of Polygons or Points in the Shape or Polygon class.
- all increment and decrement operations take constant time.

Public Member Functions

- void reset ()
- bool finished () const
- void turnAround ()
- void inc ()
- void dec ()
- IteratorPosition getPosition () const
- IteratorRange getRange () const
- void print ()

7.1.2 Member Function Documentation

7.1.2.1 void ShapeIterator::reset () [inline]

Resets the iterator to its start-position.

7.1.2.2 bool ShapeIterator::finished () const [inline]

Returns true if iterator position reached the last element to be traversed.

In SHAPE_WRAP mode that means the end of a complete Shape iteration. In POLYGON_WRAP mode the iterator stops when reaching the end of the polygonal curve, if it is not connected, or after reaching the start position for a connected curve.

7.1.2.3 void ShapeIterator::turnAround ()

Swaps values of the current position and the start position and changes direction of the iterator.

7.1.2.4 void ShapeIterator::inc () [inline]

Moves iterator position to the next element in order of traversal.

7.1.2.5 void ShapeIterator::dec () [inline]

Moves iterator position to the previous element in order of traversal.

7.1.2.6 IteratorPosition ShapeIterator::getPosition () const [inline]

Extract the current position of the iterator, in terms of Polygon and vertex position.

The result can be used to initialise a new iterator of the same kind with the start position set to this position.

7.1.2.7 IteratorRange ShapeIterator::getRange () const [inline]

Extract the current range of the iterator in terms of Polygon and vertex position.

The range covers the shape from starting point up to the current position. The result can be used to initialise a new iterator, that will only cover this range.

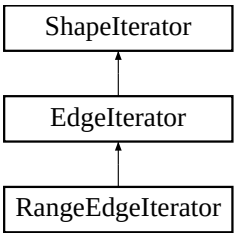
7.1.2.8 void ShapeIterator::print () [inline]

Prints detailed information about the iterator position to stderr.

7.2 EdgeIterator Class Reference

`#include <edge_iterator.h>`

Inheritance diagram for EdgeIterator::



7.2.1 Detailed Description

Iterator to access the Edges of the polylines of a Shape.

See also:

ShapeIterator

Public Member Functions

- EdgeIterator (Shape *s, ShapeIterationMode im=SHAPE_WRAP, int polygon=0, int point=1)
- EdgeIterator (Shape *s, IteratorPosition pos, ShapeIterationMode im=SHAPE_WRAP)
- Edge get ()
- Point p ()
- Point q ()
- double getOffset ()
- double getShapeRelativeLength ()
- double getLength ()
- Point createPoint (double offset)
- Point createRelativePoint (double t)

Friends

- Edge operator * (EdgeIterator &si)
- EdgeIterator & operator++ (EdgeIterator &si)
- EdgeIterator & operator– (EdgeIterator &si)

7.2.2 Constructor & Destructor Documentation

7.2.2.1 EdgeIterator::EdgeIterator (Shape * *s*, ShapeIterationMode *im* = SHAPE_WRAP, int *polygon* = 0, int *point* = 1) [inline]

Initializes an iterator for the segments of a given shape.

Default behavior ist to iterate over all segments of the polygons of the shape and go to the first segment of the first polygon after the last segment of the last polygon has been traversed. Dafault iterator position is the first segment of the the first polygon.

See also:

ShapeIterator, ShapeIterationMode

7.2.2.2 EdgeIterator::EdgeIterator (Shape * *s*, IteratorPosition *pos*, ShapeIterationMode *im* = SHAPE_WRAP) [inline]

Initializes an iterator for the segments of a given shape with starting position set to `pos`.

Default behavior ist to iterate over all segments of the polygons of the shape.

See also:

ShapeIterator, ShapeIterationMode

7.2.3 Member Function Documentation

7.2.3.1 Edge EdgeIterator::get () [inline]

Get a line segment corresponding to the current iterator position.

7.2.3.2 Point EdgeIterator::p () [inline]

Returns the first (in direction of traversal) point of the current segment.

7.2.3.3 Point EdgeIterator::q () [inline]

Returns the second (in direction of traversal) point of the current segment.

7.2.3.4 double EdgeIterator::getOffset () [inline]

Returns total relative length of the line segments traversed up to the current iterator position.

7.2.3.5 double EdgeIterator::getShapeRelativeLength () [inline]

Returns the relative length of the current segment.

7.2.3.6 double EdgeIterator::getLength () [inline]

Returns length of the current segment.

7.2.3.7 Point EdgeIterator::createPoint (double *offset*) [inline]

Returns a point on the current segment.

with distance `offset` to the first vertex `p`

Precondition:

```
offset >= 0 && offset <= getLength()
```

7.2.3.8 Point EdgeIterator::createRelativePoint (double *t*) [inline]

Returns a point on the current line segment $s = \overline{pq}$ defined as $(1-x)p + xq$ if $x > 0$, and $(1-|x|)q + |x|p$ otherwise.

Precondition:

```
t > -1 && t < 1 && ratio != 0
```

See also:

Point.xInBetween()

7.2.4 Friends And Related Function Documentation

7.2.4.1 Edge operator ∗ (EdgeIterator & *si*) [friend]

Get a line segement object corresponding to the current iterator position.

7.2.4.2 EdgeIterator& operator++ (EdgeIterator & *si*) [friend]

Moves iterator position to the next (in traversal direction) line segment of the current polygon or to the first segment of the next polygon.

If the last line segment to be traversed is reached, the subsequent call of finished() would return true.

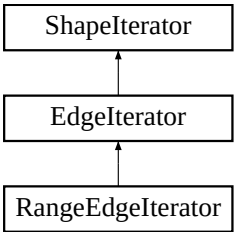
7.2.4.3 EdgeIterator& operator– (EdgeIterator & *si*) [friend]

Moves iterator position to the previous (in traversal direction) line segment of the current polygon or to the last segment of the previous polygon.

7.3 RangeEdgeIterator Class Reference

`#include <range_edge_iterator.h>`

Inheritance diagram for RangeEdgeIterator::



7.3.1 Detailed Description

Iterator to access several Ranges of the Edges of a Shape.

See also:

EdgeIterator

Public Member Functions

- RangeEdgeIterator (Shape ∗s, const vector< IteratorRange > ∗ranges, ShapeIterationMode im=SHAPE_WRAP)

7.3.2 Constructor & Destructor Documentation

7.3.2.1 RangeEdgeIterator::RangeEdgeIterator (Shape * *s*, const vector< IteratorRange > * *ranges*, ShapeIterationMode *im* = SHAPE_WRAP) [inline]

Initializes an iterator for the given shape over the given ranges.

The iterator traverses only the line segments of the polygons of the shapes that lie within the specified ranges.

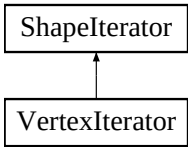
See also:

ShapeIterator, ShapeIterationMode

7.4 VertexIterator Class Reference

`#include <vertex_iterator.h>`

Inheritance diagram for VertexIterator::



7.4.1 Detailed Description

Iterator to access the vertices of the polygons of a shape.

See also:

ShapeIterator

Public Member Functions

- VertexIterator (Shape *s, ShapeIterationMode im=SHAPE_WRAP, int polygon=0, int point=0)
- VertexIterator (Shape *s, IteratorPosition pos, ShapeIterationMode im=SHAPE_WRAP)
- const Point & get ()
- double getShapeRelativeNextLength ()
- double getNextLength ()
- double getShapeRelativePreviousLength ()
- double getPreviousLength ()

Friends

- const Point & operator * (VertexIterator &si)
- VertexIterator & operator++ (VertexIterator &si)
- VertexIterator & operator– (VertexIterator &si)

7.4.2 Constructor & Destructor Documentation

7.4.2.1 VertexIterator::VertexIterator (Shape * *s*, ShapeIterationMode *im* = SHAPE_WRAP, int *polygon* = 0, int *point* = 0) [inline]

Initializes an iterator traversing vertices of the polygons of the given shape.

Default mode is to traverse all polygons. Default starting position is the frst vertex of the first polygon.

See also:

ShapeIterator, ShapeIterationMode

7.4.2.2 VertexIterator::VertexIterator (Shape * *s*, IteratorPosition *pos*, ShapeIterationMode *im* = SHAPE_WRAP) [inline]

Initializes an iterator traversing vertices of the polygons of the given shape with stsrting position set to `pos`.

Default mode is to traverse all polygons.

See also:

ShapeIterator, ShapeIterationMode

7.4.3 Member Function Documentation

7.4.3.1 const Point& VertexIterator::get () [inline]

Returns a reference to the vertex the iterators current position refers to.

7.4.3.2 double VertexIterator::getShapeRelativeNextLength () [inline]

Returns the relative length of the next, in traversal direction, line segment adjacent to the current vertex.

7.4.3.3 double VertexIterator::getNextLength () [inline]

Returns the length of the next, in traversal direction, line segment adjacent to the current vertex.

7.4.3.4 double VertexIterator::getShapeRelativePreviousLength () [inline]

Returns the relative length of the previous, in traversal direction, line segment adjacent to the current vertex.

7.4.3.5 double VertexIterator::getPreviousLength () [inline]

Returns the length of the previous, in traversal direction, line segment adjacent to the current vertex.

7.4.4 Friends And Related Function Documentation

7.4.4.1 const Point& operator * (VertexIterator & *si*) [friend]

Returns a reference to the vertex the iterators current position refers to.

7.4.4.2 VertexIterator& operator++ (VertexIterator & *si*) [friend]

Moves iterator position to the next polygon vertex in traversal direction.

7.4.4.3 VertexIterator& operator– (VertexIterator & *si*) [friend]

Moves iterator position to the previouss polygon vertex in traversal direction.

7.5 IteratorPosition Class Reference

`#include <iterator_position.h>`

7.5.1 Detailed Description

Position defined by Polygon and vertex index.

See also:

ShapeIterator
VertexIterator
EdgeIterator

Public Member Functions

- IteratorPosition (int p=INVALID_POSITION, int v=INVALID_POSITION)
- int polygon () const
- int point () const
- bool operator< (const IteratorPosition &other) const
- bool operator== (const IteratorPosition &other) const
- bool operator!= (const IteratorPosition &other) const
- bool operator<= (const IteratorPosition &other) const
- bool operator> (const IteratorPosition &other) const

Static Public Attributes

- static const int **INFINITE_POSITION** = 0
- static const int **INVALID_POSITION** = -3

Friends

- ostream & **operator**<< (ostream &output, const IteratorPosition &ip)

7.5.2 Constructor & Destructor Documentation

7.5.2.1 IteratorPosition::IteratorPosition (int *p* = INVALID_POSITION, int *v* = INVALID_POSITION) [inline]

Initialises the iterator position with a given polygon and a given vertex.

If no polygon or vertex position is given, the iterator position is invalid.

7.5.3 Member Function Documentation

7.5.3.1 int IteratorPosition::polygon () const [inline]

Returns the index of the polygon corresponding to the iterator position.

7.5.3.2 int IteratorPosition::point () const [inline]

Returns the index of the vertex of the polygon corresponding to the iterator position.

7.5.3.3 bool IteratorPosition::operator< (const IteratorPosition & *other*) const [inline]

Returns true if this iterator position preceeds the `other`.

7.5.3.4 bool IteratorPosition::operator== (const IteratorPosition & *other*) const [inline]

Returns true if the iterator positions are pointing to the same vertices of the same polygons.

7.5.3.5 bool IteratorPosition::operator!= (const IteratorPosition & *other*) const [inline]

returns true if the iterator positions are pointing to the same vertices of the same polygons.

7.5.3.6 bool IteratorPosition::operator<= (const IteratorPosition & *other*) const [inline]

Returns true if this iterator position preceeds the `other` or points to the same vertex/edge.

7.5.3.7 bool IteratorPosition::operator> (const IteratorPosition & *other*) const [inline]

Returns true if the `other` iterator position preceeds this.

7.6 IteratorRange Class Reference

`#include <iterator_range.h>`

7.6.1 Detailed Description

Interval between two IteratorPositions.

See also:

IteratorPosition

Public Member Functions

- IteratorRange ()
- IteratorRange (const IteratorPosition &start, const IteratorPosition &end, double diam=0)
- IteratorPosition start () const
- IteratorPosition end () const
- void start (IteratorPosition pos)
- void end (IteratorPosition pos)
- void mergeInto (list< IteratorRange > &ranges)

- bool isWrapping ()
- void invert ()

Static Public Member Functions

- static vector< IteratorRange > * merge (vector< IteratorRange > &ranges)
- static void merge (vector< IteratorRange > &input, vector< IteratorRange > *output)
- static void print (vector< IteratorRange > *ips)

Friends

- ostream & operator<< (ostream &output, const IteratorRange &ir)

7.6.2 Constructor & Destructor Documentation

7.6.2.1 IteratorRange::IteratorRange () [inline]

Creates an empty range.

7.6.2.2 IteratorRange::IteratorRange (const IteratorPosition & *start*, const IteratorPosition & *end*, double *diam* = 0) [inline]

Creates an iterator range with a given start and end position.

7.6.3 Member Function Documentation

7.6.3.1 IteratorPosition IteratorRange::start () const [inline]

Returns the first iterator position in the range.

7.6.3.2 IteratorPosition IteratorRange::end () const [inline]

Returns the last iterator position in the range.

7.6.3.3 void IteratorRange::start (IteratorPosition *pos*) [inline]

Sets the start position of the range to pos.

7.6.3.4 void IteratorRange::end (IteratorPosition *pos*) [inline]

Sets the end position of the range to pos.

7.6.3.5 vector< IteratorRange > * IteratorRange::merge (vector< IteratorRange > & *ranges*) [static]

Merges overlapping ranges of the input set.

7.6.3.6 void IteratorRange::merge (vector< IteratorRange > & *input*, vector< IteratorRange > * *output*) [static]

Merges overlapping ranges of the input set.

7.6.3.7 void IteratorRange::mergeInto (list< IteratorRange > & *ranges*)

Merges overlapping ranges in the given set, modifies the input list.

7.6.3.8 bool IteratorRange::isWrapping () [inline]

Returns whether the range is wrapping, a range can be wrapping if the respective polygon is connected.

7.6.3.9 void IteratorRange::invert () [inline]

Inverts the range.

7.6.3.10 void IteratorRange::print (vector< IteratorRange > * *ips*) [static]

Writes the first and the last elements of the iterator range to stdout.

7.6.4 Friends And Related Function Documentation

7.6.4.1 ostream& operator<< (ostream & *output*, const IteratorRange & *ir*) [friend]

Writes the first and the last elements of the iterator range to the output stream.

7.7 IteratorRangePair Class Reference

`#include <iterator_range_pair.h>`

7.7.1 Detailed Description

Container class for two IteratorRanges.

See also:

IteratorRange

Public Member Functions

- IteratorRangePair (const IteratorRange &source, const IteratorRange &target)
- const IteratorRange & source () const
- IteratorRange * getSource ()
- const IteratorRange & target () const
- IteratorRange * getTarget ()

7.7.2 Constructor & Destructor Documentation

7.7.2.1 IteratorRangePair::IteratorRangePair (const IteratorRange & *source*, const IteratorRange & *target*) [inline]

Initializes range pair with the given iterator ranges.

7.7.3 Member Function Documentation

7.7.3.1 const IteratorRange& IteratorRangePair::source () const [inline]

Returns the first element of the pair.

7.7.3.2 IteratorRange* IteratorRangePair::getSource () [inline]

Returns a pointer to the first element of the pair.

7.7.3.3 const IteratorRange& IteratorRangePair::target () const [inline]

Returns the second element of the pair.

7.7.3.4 IteratorRange* IteratorRangePair::getTarget () [inline]

Returns a pointer to the second element of the pair.