SANDIA REPORT

SAND2006-6829
Unlimited Release
Printed November 2006

GBL-2D Version 1.0: A 2D Geometry Boolean Library

Corey L. McBride, Victor Yarberry, Rodney C. Schmidt, and Ray J. Meyers

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94-AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd.
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2006-6829
Unlimited Release
Printed November 2006

GBL-2D Version 1.0: A 2D Geometry Boolean Library

Corey L. McBride and Ray J. Meyers
Elemental Technologies
17 North Merchant Street
American Fork, UT 84003

Victor Yarberry and Rodney C. Schmidt
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185

Abstract

This report describes version 1.0 of GBL-2D, a geometric Boolean library for 2D objects. The library is written in C++ and consists of a set of classes and routines. The classes primarily represent geometric data and relationships. Classes are provided for 2D points, lines, arcs, edgeuses, loops, surfaces and mask sets. The routines contain algorithms for geometric Boolean operations and utility functions. Routines are provided that incorporate the Boolean operations: Union(OR), XOR, Intersection and Difference. A variety of additional analytical geometry routines and routines for importing and exporting the data in various file formats are also provided.

The GBL-2D library was originally developed as a geometric modeling engine for use with a separate software tool, called SummitView [1], that manipulates the 2D mask sets created by designers of Micro-Electro-Mechanical Systems (MEMS). However, many other practical applications for this type of software can be envisioned because the need to perform 2D Boolean operations can arise in many contexts.

Contents

1 Introduction and Background **31**

2 Overview of GBL-2D 1.0 **33**

2.1 Supported Compilers 33

2.2 2D Geometry Classes 33

2.2.1 Geometry Data Structures 34

2.2.1.1 DDPoint 34

2.2.1.2 DDEdge 34

2.2.1.3 DDLine 34

2.2.1.4 DDArc 34

2.2.2 Topology Classes 34

2.2.2.1 DDEdgeUse 34

2.2.2.2 DDLoop 35

2.2.2.3 DDSurface 35

2.2.3 Virtual Topology Classes 36

2.2.3.1 DDVirtualLoop 36

2.2.3.2 DDVirtualSurface 36

2.3 2D Geometric Routines 36

2.3.1 Union Operation Routines 36

2.3.2 XOR Operation Routines 38

2.3.3 Imprinting Routines 38

2.3.4 Offset Routines 40

3 2D Data Classes **43**

3.1 DDVector Class Reference 43

3.1.1 Detailed Description 44

3.1.2 Constructor & Destructor Documentation 44

3.1.2.1 DDVector::DDVector () 44

3.1.2.2 DDVector::DDVector (double *x*, double *y*) 44

3.1.2.3 DDVector::DDVector (double *xy*[2]) 45

3.1.2.4 DDVector::DDVector (const **DDVector** & *copy_from*) 45

3.1.3 Member Function Documentation 45

3.1.3.1 void DDVector::get_xy (double & *x*, double & *y*) 45

3.1.3.2 void DDVector::get_xy (double *xy*[2]) 45

3.1.3.3 double DDVector::length () 45

3.1.3.4 double DDVector::length_squared () 45

3.1.3.5 double DDVector::normalize () 45

3.1.3.6 **DDVector**& DDVector::operator ∗= (const double *scalar*) 46

3.1.3.7 void DDVector::operator delete (void ∗ *p*, void ∗) 46

3.1.3.8 void∗ DDVector::operator new (size_t *size*) 46

3.1.3.9 **DDVector**& DDVector::operator+= (const **DDVector** & *vector*) . 46

3.1.3.10 **DDVector** DDVector::operator- () 46

3.1.3.11 **DDVector**& DDVector::operator-= (const **DDVector** & *vector*) . 46

3.1.3.12 **DDVector**& DDVector::operator/= (const double *scalar*) 46

3.1.3.13 **DDVector**& DDVector::operator= (const **DDVector** & *from*) . . . 46

3.1.3.14 void DDVector::perpendicular () 46

3.1.3.15 void DDVector::set (double *xy*[2]) 47

3.1.3.16 void DDVector::set (double *x*, double *y*) 47

3.1.3.17 double DDVector::x () const 47

3.1.3.18 void DDVector::x (double *x*) 47

3.1.3.19 double DDVector::y () const 47

3.1.3.20 void DDVector::y (double *y*) 47

3.1.4 Friends And Related Function Documentation 47

3.1.4.1 **DDVector** operator ∗ (const double *scalar*, const **DDVector** & *vector*) `[friend]` 47

3.1.4.2 **DDVector** operator ∗ (const **DDVector** & *vector*, const double *scalar*) `[friend]` 47

3.1.4.3 double operator ∗ (const **DDVector** & *vector1*, const **DDVector** & *vector2*) `[friend]` 48

3.1.4.4 bool operator!= (const **DDVector** & *vector1*, const **DDVector** & *vector2*) `[friend]` 48

3.1.4.5 double operator% (const **DDVector** & *vector1*, const **DDVector** & *vector2*) `[friend]` 48

3.1.4.6 **DDVector** operator+ (const **DDVector** & *vector1*, const **DDVector** & *vector2*) `[friend]` 48

3.1.4.7 **DDVector** operator- (const **DDVector** & *vector1*, const **DDVector** & *vector2*) `[friend]` 48

3.1.4.8 **DDVector** operator/ (const **DDVector** & *vector*, const double *scalar*) `[friend]` 48

3.1.4.9 bool operator== (const **DDVector** & *vector1*, const **DDVector** & *vector2*) `[friend]` 48

3.2 DDPoint Class Reference 49

3.2.1 Detailed Description 50

3.2.2 Constructor & Destructor Documentation 50

3.2.2.1 DDPoint::DDPoint (unsigned long *id* = `0xffffffff`) 50

3.2.2.2 DDPoint::∼**DDPoint** () 50

3.2.3 Member Function Documentation 50

3.2.3.1 bool DDPoint::are_edges_registered () 50

3.2.3.2 void DDPoint::delete_all_instances () `[static]` 51

3.2.3.3 DD_RESULT DDPoint::delete_point (**DDPoint** ∗& *rp_point*) [static] 51

3.2.3.4 unsigned long DDPoint::get_ID () 51

3.2.3.5 void DDPoint::get_ID (unsigned long & *r_id*) 51

3.2.3.6 unsigned long DDPoint::get_ID_generator () [static] 51

3.2.3.7 void DDPoint::get_instance_map (**DDHashMap**< unsigned long, **DDPoint** ∗ > & *r_point_map*) [static] 51

3.2.3.8 unsigned long DDPoint::get_instance_map_size () [static] . . 51

3.2.3.9 DD_RESULT DDPoint::get_point (unsigned long & *r_id*, **DDPoint** ∗& *rp_point*) [static] 52

3.2.3.10 void DDPoint::get_registered_edges (std::list< **DDEdge** ∗ > & *r_edge_list*) 52

3.2.3.11 **DDPoint**& DDPoint::operator+= (**DDVector** & *r_vector*) 52

3.2.3.12 **DDPoint**& DDPoint::operator= (const **DDPoint** & *r_from_point*) 52

3.2.3.13 DD_RESULT DDPoint::register_edge (**DDEdge** ∗ *p_edge*) 52

3.2.3.14 DD_RESULT DDPoint::unregister_edge (**DDEdge** ∗ *p_edge*) . . . 53

3.2.4 Friends And Related Function Documentation 53

3.2.4.1 bool operator!= (**DDPoint** & *r_p1*, **DDPoint** & *r_p2*) [friend] 53

3.2.4.2 **DDVector** operator- (const **DDPoint** & *r_point_1*, const **DDPoint** & *r_point_2*) [friend] 53

3.2.4.3 bool operator== (**DDPoint** & *r_p1*, **DDPoint** & *r_p2*) [friend] 53

3.2.5 Member Data Documentation 53

3.2.5.1 double **DDPoint::mX** 53

3.2.5.2 double **DDPoint::mY** 54

3.3 DDEdge Class Reference 55

3.3.1 Detailed Description 56

3.3.2 Constructor & Destructor Documentation 56

3.3.2.1 DDEdge::DDEdge (**DDPoint** ∗ *p_start_point*, **DDPoint** ∗ *p_end_point*, unsigned long *id* = `0xffffffff`) 56

3.3.2.2 virtual DDEdge::∼**DDEdge** () `[virtual]` 57

3.3.3 Member Function Documentation 57

3.3.3.1 virtual void DDEdge::calculate_bounding_box (**DDBoundingBox** & *r_bounding_box*) `[pure virtual]` 57

3.3.3.2 virtual double DDEdge::calculate_length () `[pure virtual]` . 57

3.3.3.3 virtual void DDEdge::calculate_midpoint (**DDPoint** & *r_midpoint*) `[pure virtual]` 57

3.3.3.4 virtual void DDEdge::calculate_point (double *t*, **DDPoint** & *r_point*) `[pure virtual]` 58

3.3.3.5 virtual void DDEdge::create_connected_copy_of_edge (**DDPoint** ∗ *p_start_point*, **DDPoint** ∗ *p_end_point*, **DDEdge** ∗& *rp_copy*) `[pure virtual]` 58

3.3.3.6 virtual void DDEdge::create_integrated_copy_of_edge (**DDEdge** ∗& *rp_copy*) `[pure virtual]` 58

3.3.3.7 virtual void DDEdge::create_isolated_copy_of_edge (**DDEdge** ∗& *rp_copy*) `[pure virtual]` 59

3.3.3.8 void DDEdge::delete_all_instances () `[static]` 59

3.3.3.9 DD_RESULT DDEdge::delete_edge (**DDEdge** ∗& *rp_edge*) `[static]` 59

3.3.3.10 DD_RESULT DDEdge::get_edge (unsigned long & *r_id*, **DDEdge** ∗& *rp_edge*) `[static]` 59

3.3.3.11 virtual void DDEdge::get_edge_type (DDEdgeType & *r_edge_type*) `[pure virtual]` 60

3.3.3.12 void DDEdge::get_edge_uses (**DDEdgeUse** ∗& *rp_forward_edge_use*, **DDEdgeUse** ∗& *rp_backward_edge_use*) 60

3.3.3.13 void DDEdge::get_end_point_ids (unsigned long & *r_start*, unsigned long & *r_end*) 60

3.3.3.14 void DDEdge::get_end_points (**DDPoint** ∗& *rp_start_point*, **DDPoint** ∗& *rp_end_point*) 61

3.3.3.15 unsigned long DDEdge::get_ID () 61

3.3.3.16 void DDEdge::get_ID (unsigned long & *r_id*) 61

3.3.3.17 unsigned long DDEdge::get_ID_generator () `[static]` 61

3.3.3.18 void DDEdge::get_instance_map (**DDHashMap**< unsigned long, **DDEdge** ∗ > & *r_edge_map*) `[static]` 61

3.3.3.19 unsigned long DDEdge::get_instance_map_size () 61

3.3.3.20 virtual DD_RESULT DDEdge::is_equal (**DDEdge** & *r_edge*, double *tolerance*) `[pure virtual]` 62

3.3.3.21 virtual DD_RESULT DDEdge::offset_edge (double *distance*) `[pure virtual]` 62

3.3.3.22 DD_RESULT DDEdge::register_edge_use (**DDEdgeUse** ∗ *p_edge_use*) 62

3.3.3.23 virtual DD_RESULT DDEdge::replace_point (**DDPoint** ∗ *p_old_point*, **DDPoint** ∗ *p_new_point*) `[virtual]` 62

3.3.3.24 virtual DD_RESULT DDEdge::replace_point_and_update (**DDPoint** ∗ *p_old_point*, **DDPoint** ∗ *p_new_point*) `[virtual]` 63

3.3.3.25 DD_RESULT DDEdge::unregister_edge_use (**DDEdgeUse** ∗ *p_edge_use*) 64

3.4 DDLine Class Reference 65

3.4.1 Detailed Description 65

3.4.2 Constructor & Destructor Documentation 65

3.4.2.1 DDLine::DDLine (**DDPoint** ∗ *p_start_point*, **DDPoint** ∗ *p_end_point*, unsigned long *id* = 0xffffffff) 65

3.4.2.2 DDLine::∼**DDLine** () `[inline]` 66

3.4.3 Member Function Documentation 66

3.4.3.1 void DDLine::calculate_bounding_box (**DDBoundingBox** & *r_bounding_box*) `[virtual]` 66

3.4.3.2 double DDLine::calculate_length () `[virtual]` 66

3.4.3.3 void DDLine::calculate_midpoint (**DDPoint** & *r_midpoint*) `[virtual]` 67

3.4.3.4 void DDLine::calculate_point (double *t*, **DDPoint** & *r_point*) [virtual] 67

3.4.3.5 virtual void DDLine::create_connected_copy_of_edge (**DDPoint** ∗ *p_start_point*, **DDPoint** ∗ *p_end_point*, **DDEdge** ∗& *rp_copy*) [virtual] 67

3.4.3.6 void DDLine::create_integrated_copy_of_edge (**DDEdge** ∗& *rp_copy*) [virtual] 68

3.4.3.7 void DDLine::create_isolated_copy_of_edge (**DDEdge** ∗& *rp_copy*) [virtual] 68

3.4.3.8 virtual void DDLine::get_edge_type (DDEdgeType & *r_edge_type*) [virtual] 68

3.4.3.9 DD_RESULT DDLine::is_equal (**DDEdge** & *r_edge*, double *tolerance*) [virtual] 68

3.4.3.10 DD_RESULT DDLine::offset_edge (double *distance*) [virtual] 69

3.5 DDArc Class Reference 70

3.5.1 Detailed Description 71

3.5.2 Constructor & Destructor Documentation 71

3.5.2.1 DDArc::DDArc (**DDPoint** ∗ *p_start_point*, **DDPoint** ∗ *p_end_point*, **DDPoint** ∗ *p_center_point*, unsigned long *id* = 0xffffffff) 71

3.5.2.2 DDArc::∼**DDArc** () 72

3.5.3 Member Function Documentation 72

3.5.3.1 void DDArc::calculate_angle_relative_to_start_angle (**DDPoint** & *r_point*, double & *r_angle*) 72

3.5.3.2 double DDArc::calculate_area () 72

3.5.3.3 void DDArc::calculate_bounding_box (**DDBoundingBox** & *r_bounding_box*) [virtual] 73

3.5.3.4 double DDArc::calculate_chordal_distance () 73

3.5.3.5 double DDArc::calculate_length () [virtual] 73

3.5.3.6 void DDArc::calculate_midpoint (**DDPoint** & *r_midpoint*) [virtual] 73

3.5.3.7 void DDArc::calculate_parameteric_parameter (**DDPoint** & *r_point*, double & *r_t*) 73

3.5.3.8 void DDArc::calculate_point (double *t*, **DDPoint** & *r_point*) `[virtual]` 74

3.5.3.9 double DDArc::calculate_radius () 74

3.5.3.10 void DDArc::calculate_radius (double & *r_radius*) 74

3.5.3.11 void DDArc::calculate_start_stop_angle (double & *r_start_angle*, double & *r_stop_angle*) 74

3.5.3.12 double DDArc::calculate_sweep_angle () 75

3.5.3.13 virtual void DDArc::create_connected_copy_of_edge (**DDPoint** ∗ *p_start_point*, **DDPoint** ∗ *p_end_point*, **DDEdge** ∗& *rp_copy*) `[virtual]` 75

3.5.3.14 void DDArc::create_integrated_copy_of_edge (**DDEdge** ∗& *rp_copy*) `[virtual]` 75

3.5.3.15 void DDArc::create_isolated_copy_of_edge (**DDEdge** ∗& *rp_copy*) `[virtual]` 75

3.5.3.16 void DDArc::get_center_point (**DDPoint** ∗& *rp_center_point*) . . . 76

3.5.3.17 virtual void DDArc::get_edge_type (DDEdgeType & *r_edge_type*) `[virtual]` 76

3.5.3.18 DD_RESULT DDArc::is_equal (**DDEdge** & *r_edge*, double *tolerance*) `[virtual]` 76

3.5.3.19 DD_RESULT DDArc::offset_edge (double *distance*) `[virtual]` 77

3.5.3.20 void DDArc::position_center_point_based_upon_sweep_angle (double *sweep_angle*) 77

3.5.3.21 DD_RESULT DDArc::replace_point (**DDPoint** ∗ *p_old_point*, **DDPoint** ∗ *p_new_point*) `[virtual]` 78

3.5.3.22 DD_RESULT DDArc::replace_point_and_update (**DDPoint** ∗ *p_old_point*, **DDPoint** ∗ *p_new_point*) `[virtual]` 78

3.5.4 Member Data Documentation 79

3.5.4.1 **DDPoint**∗ **DDArc::mpCenterPoint** `[protected]` 79

3.6 DDEdgeUse Class Reference 80

3.6.1 Detailed Description 81

3.6.2 Constructor & Destructor Documentation 81

3.6.2.1 DDEdgeUse::DDEdgeUse (**DDEdge** ∗ *p_edge*, DDEdgeUseSense *use_sense*, unsigned long *id* = `0xffffffff`) 81

3.6.2.2 DDEdgeUse::∼**DDEdgeUse** () 82

3.6.3 Member Function Documentation 82

3.6.3.1 void DDEdgeUse::calculate_bounding_box (**DDBoundingBox** & *r_bounding_box*) 82

3.6.3.2 double DDEdgeUse::calculate_length () 82

3.6.3.3 void DDEdgeUse::calculate_midpoint (**DDPoint** & *r_midpoint*) . 82

3.6.3.4 void DDEdgeUse::calculate_point (double *distance*, **DDPoint** & *r_point*) 82

3.6.3.5 void DDEdgeUse::create_connected_copy_of_edge_use (**DDPoint** ∗ *p_start_point*, **DDPoint** ∗ *p_end_point*, **DDEdgeUse** ∗& *rp_copy*) 83

3.6.3.6 void DDEdgeUse::create_integrated_copy_of_edge_use (**DDEdgeUse** ∗& *rp_copy*) 83

3.6.3.7 void DDEdgeUse::create_isolated_copy_of_edge_use (**DDEdgeUse** ∗& *rp_copy*) 83

3.6.3.8 void DDEdgeUse::delete_all_instances () `[static]` 84

3.6.3.9 DD_RESULT DDEdgeUse::delete_edgeuse (**DDEdgeUse** ∗& *rp_edgeuse*) `[static]` 84

3.6.3.10 DD_RESULT DDEdgeUse::get_edge (**DDEdge** ∗& *rp_edge*, DDEdgeUseSense & *r_use_sense*) 84

3.6.3.11 void DDEdgeUse::get_edge_type (DDEdgeType & *r_type*) 84

3.6.3.12 DD_RESULT DDEdgeUse::get_edge_use (unsigned long & *r_id*, **DDEdgeUse** ∗& *rp_edge_use*) `[static]` 84

3.6.3.13 DDEdgeUseSense DDEdgeUse::get_edge_use_sense () 85

3.6.3.14 void DDEdgeUse::get_end_point_ids (unsigned long & *r_start*, unsigned long & *r_end*) 85

3.6.3.15 void DDEdgeUse::get_end_points (**DDPoint** ∗& *rp_start_point*, **DDPoint** ∗& *rp_end_point*, DDEdgeUseSense & *r_use_sense*) . . 85

3.6.3.16 unsigned long DDEdgeUse::get_ID () 85

3.6.3.17 void DDEdgeUse::get_ID (unsigned long & *r_id*) 86

3.6.3.18 unsigned long DDEdgeUse::get_ID_generator () `[static]` 86

3.6.3.19 DD_RESULT DDEdgeUse::get_instance_map (**DDHashMap**< unsigned long, **DDEdgeUse** ∗ > & *r_edge_use_map*) `[static]` 86

3.6.3.20 unsigned long DDEdgeUse::get_instance_map_size () `[static]` 86

3.6.3.21 DD_RESULT DDEdgeUse::get_loop (**DDLoop** ∗& *rp_loop*) . . . 86

3.6.3.22 DDLoop::iterator DDEdgeUse::get_loop_iterator () 87

3.6.3.23 DD_RESULT DDEdgeUse::offset_edgeuse (double *distance*) . . . 87

3.6.3.24 DD_RESULT DDEdgeUse::register_loop (**DDLoop** ∗ *p_loop*, DDLoop::iterator *iterator*) 87

3.6.3.25 DD_RESULT DDEdgeUse::replace_edge (**DDEdge** ∗ *p_old_-edge*, **DDEdge** ∗ *p_new_edge*, DDEdgeUseSense *new_use_sense*) 87

3.6.3.26 DD_RESULT DDEdgeUse::unregister_loop (**DDLoop** ∗ *p_loop*) 88

3.7 DDLoop Class Reference 89

3.7.1 Detailed Description 90

3.7.2 Constructor & Destructor Documentation 90

3.7.2.1 DDLoop::DDLoop (unsigned long *id* = `0xffffffff`) 90

3.7.2.2 virtual DDLoop::∼**DDLoop** () `[virtual]` 91

3.7.3 Member Function Documentation 91

3.7.3.1 virtual void DDLoop::calculate_bounding_box (**DDBounding-Box** & *r_bounding_box*) `[virtual]` 91

3.7.3.2 void DDLoop::calculate_centroid_of_end_points (**DDPoint** & *r_centroid*) 91

3.7.3.3 virtual void DDLoop::clear () `[virtual]` 91

3.7.3.4 virtual void DDLoop::create_integrated_copy_of_loop (**DDLoop** ∗& *rp_copy*) `[virtual]` 92

3.7.3.5 virtual void DDLoop::create_isolated_copy_of_loop (**DDLoop** ∗& *rp_copy*) [virtual] 92

3.7.3.6 void DDLoop::delete_all_instances () [static] 92

3.7.3.7 DD_RESULT DDLoop::delete_loop (**DDLoop** ∗& *rp_loop*) [static] 92

3.7.3.8 virtual DDEdgeUseList::iterator DDLoop::erase (std::list< **DDEdgeUse** ∗ >::iterator *pos*) [virtual] 92

3.7.3.9 virtual unsigned long DDLoop::get_ID () [virtual] 93

3.7.3.10 virtual void DDLoop::get_ID (unsigned long & *r_id*) [virtual] 93

3.7.3.11 unsigned long DDLoop::get_ID_generator () [static] 93

3.7.3.12 DD_RESULT DDLoop::get_instance_map (**DDHashMap**< unsigned long, **DDLoop** ∗ > & *r_loop_map*) [static] 93

3.7.3.13 unsigned long DDLoop::get_instance_map_size () [static] . . 93

3.7.3.14 virtual double DDLoop::get_length () [virtual] 93

3.7.3.15 DD_RESULT DDLoop::get_loop (unsigned long & *r_id*, **DDLoop** ∗& *rp_loop*) [static] 94

3.7.3.16 virtual void DDLoop::get_loop_type (DDLoopType & *r_loop_type*) [virtual] 94

3.7.3.17 virtual void DDLoop::get_neighboring_loops (std::list< **DDLoop** ∗ > & *r_neighboring_loops*) [virtual] 94

3.7.3.18 virtual void DDLoop::get_point (double *t*, **DDPoint** & *r_point*) [virtual] 94

3.7.3.19 virtual void DDLoop::get_points (std::list< **DDPoint** ∗ > & *r_point_list*) [virtual] 95

3.7.3.20 virtual void DDLoop::get_surface (**DDSurface** ∗& *rp_surface*) [virtual] 95

3.7.3.21 virtual DDEdgeUseList::iterator DDLoop::insert (DDEdgeUseList::iterator *pos*, **DDEdgeUse** ∗& *rp_edge_use*) [virtual] . . . 95

3.7.3.22 virtual void DDLoop::pop_back () [virtual] 95

3.7.3.23 virtual void DDLoop::pop_front () [virtual] 95

3.7.3.24 virtual void DDLoop::push_back (**DDEdgeUse** ∗& *rp_edge_use*) [virtual] 96

3.7.3.25 virtual void DDLoop::push_front (**DDEdgeUse** ∗& *rp_edge_use*) [virtual] 96

3.7.3.26 virtual DD_RESULT DDLoop::register_surface (**DDSurface** ∗ *p_surface*, DDLoopType *loop_type*) [virtual] 96

3.7.3.27 virtual void DDLoop::remove (**DDEdgeUse** ∗ *p_edge_use*) [virtual] 96

3.7.3.28 virtual DD_RESULT DDLoop::unregister_surface (**DDSurface** ∗ *p_surface*) [virtual] 96

3.8 DDSurface Class Reference 98

3.8.1 Detailed Description 99

3.8.2 Constructor & Destructor Documentation 99

3.8.2.1 DDSurface::DDSurface (**DDLoop** ∗ *p_loop*, unsigned long *id* = 0xffffffff) 99

3.8.2.2 DDSurface::∼**DDSurface** () 100

3.8.3 Member Function Documentation 100

3.8.3.1 DD_RESULT DDSurface::add_interior_loop (**DDLoop** ∗ *p_loop*) 100

3.8.3.2 void DDSurface::calculate_bounding_box (**DDBoundingBox** & *r_bounding_box*) 100

3.8.3.3 void DDSurface::create_integrated_copy_of_surface (**DDSurface** ∗& *rp_copy*) 100

3.8.3.4 void DDSurface::create_isolated_copy_of_surface (**DDSurface** ∗& *rp_copy*) 100

3.8.3.5 void DDSurface::delete_all_instances () [static] 101

3.8.3.6 void DDSurface::delete_surface (**DDSurface** ∗& *rp_surface*) [static] 101

3.8.3.7 void DDSurface::get_all_edgeuses (**DDEdgeUseList** & *r_edgeuse_list*) 101

3.8.3.8 DD_RESULT DDSurface::get_edgeuses (**DDPoint** & *r_point*, std::list< **DDEdgeUse** ∗ > & *r_start_edgeuse_list*, std::list< **DDEdgeUse** ∗ > & *r_end_edgeuse_list*) 101

3.8.3.9 DD_RESULT DDSurface::get_edgeuses (**DDPoint** & *r_point*, **DDEdgeUse** ∗& *rp_start_edgeuse*, **DDEdgeUse** ∗& *rp_end_edgeuse*) 102

3.8.3.10 void DDSurface::get_exterior_loop (**DDLoop** ∗& *rp_loop*) 102

3.8.3.11 unsigned long DDSurface::get_ID () 102

3.8.3.12 void DDSurface::get_ID (unsigned long & *r_id*) 103

3.8.3.13 unsigned long DDSurface::get_ID_generator () `[static]` 103

3.8.3.14 void DDSurface::get_instance_map (**DDHashMap**< unsigned long, **DDSurface** ∗ > & *r_surface_map*) `[static]` 103

3.8.3.15 unsigned long DDSurface::get_instance_map_size () `[static]` . 103

3.8.3.16 void DDSurface::get_interior_loop_list (std::list< **DDLoop** ∗ > & *r_interior_loop_list*) 103

3.8.3.17 void DDSurface::get_list_of_loops (std::list< **DDLoop** ∗ > & *r_list_of_loops*) 104

3.8.3.18 void DDSurface::get_list_of_loops (std::list< **DDEdgeUseList** ∗ > & *r_list_of_edgeuse_lists*) 104

3.8.3.19 void DDSurface::get_list_of_surface_instances (std::list< **DDSurface** ∗ > & *r_surface_list*) `[static]` 104

3.8.3.20 void DDSurface::get_neighboring_surfaces (std::list< **DDSurface** ∗ > & *r_neighbors*) 104

3.8.3.21 void DDSurface::get_points (std::list< **DDPoint** ∗ > & *r_point_list*) 105

3.8.3.22 DD_RESULT DDSurface::get_surface (unsigned long & *r_id*, **DDSurface** ∗& *rp_surface*) `[static]` 105

3.8.3.23 void DDSurface::remove_all_interior_loops () 105

3.8.3.24 void DDSurface::remove_exterior_loop () 105

3.8.3.25 DD_RESULT DDSurface::remove_interior_loop (**DDLoop** ∗ *p_loop*) 105

3.8.3.26 DD_RESULT DDSurface::replace_exterior_loop (**DDLoop** ∗ *p_old_loop*, **DDLoop** ∗ *p_new_loop*) 106

3.9 DDVirtualLoop Class Reference 107

3.9.1 Detailed Description 107

3.9.2 Constructor & Destructor Documentation 108

3.9.2.1 DDVirtualLoop::DDVirtualLoop () 108

3.9.2.2 virtual DDVirtualLoop::∼**DDVirtualLoop** () `[virtual]` 108

3.9.3 Member Function Documentation 108

3.9.3.1 void DDVirtualLoop::calculate_bounding_box (**DDBoundingBox** & *r_bounding_box*) 108

3.9.3.2 virtual DD_RESULT DDVirtualLoop::create_integrated_copy_of_loop (**DDLoop** ∗& *rp_copy*) `[virtual]` 108

3.9.3.3 DD_RESULT DDVirtualLoop::delete_loop (**DDVirtualLoop** ∗& *rp_loop*) `[static]` 109

3.9.3.4 virtual DD_RESULT DDVirtualLoop::get_loop_type (DDLoopType & *r_loop_type*) `[virtual]` 109

3.9.3.5 virtual DD_RESULT DDVirtualLoop::get_points (std::list< **DDPoint** ∗ > & *r_point_list*) `[virtual]` 109

3.9.3.6 virtual DD_RESULT DDVirtualLoop::get_surface (**DDVirtualSurface** ∗& *rp_surface*) `[virtual]` 109

3.9.3.7 virtual DD_RESULT DDVirtualLoop::register_surface (**DDVirtualSurface** ∗ *p_surface*, DDLoopType *loop_type*) `[virtual]` 110

3.9.3.8 virtual DD_RESULT DDVirtualLoop::unregister_surface (**DDVirtualSurface** ∗ *p_surface*) `[virtual]` 110

3.10 DDVirtualSurface Class Reference 111

3.10.1 Detailed Description 111

3.10.2 Constructor & Destructor Documentation 112

3.10.2.1 DDVirtualSurface::DDVirtualSurface (**DDVirtualLoop** ∗ *p_loop*) 112

3.10.2.2 DDVirtualSurface::∼**DDVirtualSurface** () 112

3.10.3 Member Function Documentation 112

3.10.3.1 DD_RESULT DDVirtualSurface::add_interior_loop (**DDVirtualLoop** ∗ *p_loop*) 112

3.10.3.2 void DDVirtualSurface::calculate_bounding_box (**DDBoundingBox** & *r_bounding_box*) 112

3.10.3.3 DD_RESULT DDVirtualSurface::delete_surface (**DDVirtual-Surface** ∗& *rp_surface*) `[static]` 113

3.10.3.4 void DDVirtualSurface::get_exterior_loop (**DDVirtualLoop** ∗& *rp_loop*) 113

3.10.3.5 std::list<**DDVirtualLoop**∗>∗ DDVirtualSurface::get_interior_-loop_list () 113

3.10.3.6 void DDVirtualSurface::get_interior_loop_list (std::list< **DDVirtualLoop** ∗ > & *r_interior_loop_list*) 113

3.10.3.7 DD_RESULT DDVirtualSurface::get_neighboring_surfaces (std::list< **DDSurface** ∗ > & *r_neighbors*) 113

3.10.3.8 unsigned long DDVirtualSurface::get_number_of_interior_loops () 114

3.10.3.9 DD_RESULT DDVirtualSurface::remove_interior_loop (**DDVirtualLoop** ∗ *p_loop*) 114

3.10.3.10 DD_RESULT DDVirtualSurface::replace_exterior_loop (**DDVirtualLoop** ∗ *p_old_loop*, **DDVirtualLoop** ∗ *p_new_loop*) . 114

3.11 DDMaskDefinitions Class Reference 115

3.11.1 Detailed Description 115

3.11.2 Member Function Documentation 115

3.11.2.1 void DDMaskDefinitions::add_mask_definition (std::string & *r_-layer*, std::list< **DDSurface** ∗ > ∗ *p_surface_list*) 115

3.11.2.2 **DDBoundingBox** DDMaskDefinitions::get_bounding_box () . . . 116

3.11.2.3 void DDMaskDefinitions::get_layer_names (std::list< std::string > & *r_layer_names*) 116

3.11.2.4 DD_RESULT DDMaskDefinitions::get_surfaces_by_layer (std::string & *r_layer*, std::list< **DDSurface** ∗ > & *r_surface_list*) 116

3.11.2.5 void DDMaskDefinitions::print_information () 116

3.11.2.6 void DDMaskDefinitions::set_bounding_box (**DDBoundingBox** & *r_bounding_box*) 117

3.11.3 Member Data Documentation 117

3.11.3.1 std::map<std::string,std::list<**DDSurface**∗>∗> **DDMaskDefinitions::mMaskMap** [protected] 117

3.12 DDBoundingBox Class Reference 118

3.12.1 Detailed Description 118

3.12.2 Member Enumeration Documentation 119

3.12.2.1 anonymous enum 119

3.12.3 Constructor & Destructor Documentation 119

3.12.3.1 DDBoundingBox::DDBoundingBox () [inline] 119

3.12.3.2 DDBoundingBox::DDBoundingBox (double *xmax*, double *xmin*, double *ymax*, double *ymin*) [inline] 119

3.12.4 Member Function Documentation 119

3.12.4.1 void DDBoundingBox::calculate_centroid (**DDPoint** & *r_point*) . 119

3.12.4.2 int DDBoundingBox::calculate_inside_outside_code (**DDPoint** & *r_point*) 119

3.12.4.3 **DDBoundingBox**& DDBoundingBox::operator ∗= (const double & *r_scalar*) [inline] 120

3.12.4.4 **DDBoundingBox**& DDBoundingBox::operator+= (const double & *r_scalar*) [inline] 120

3.12.5 Friends And Related Function Documentation 120

3.12.5.1 bool operator% (**DDBoundingBox** & *box1*, **DDBoundingBox** & *box2*) [friend] 120

3.12.5.2 **DDBoundingBox** operator+ (const **DDBoundingBox** & *r_box1*, const **DDBoundingBox** & *r_box2*) [friend] 120

3.12.6 Member Data Documentation 120

3.12.6.1 double **DDBoundingBox::mXmax** 120

3.12.6.2 double **DDBoundingBox::mXmin** 120

3.12.6.3 double **DDBoundingBox::mYmax** 121

3.12.6.4 double **DDBoundingBox::mYmin** 121

3.13 DDEdgeUseList Class Reference 122

3.13.1 Detailed Description 122

3.14 DDHashMap Class Reference 123

3.14.1 Detailed Description 123

3.15 DDQuadTree< X, E > Class Template Reference 124

3.15.1 Detailed Description 124

3.15.2 Constructor & Destructor Documentation 124

3.15.2.1 template<class X, class E> **DDQuadTree**< X, E >::**DDQuadTree** (std::list< X ∗ > & *r_points*, double *tolerance*, int *min_nodes_per_box* = `-1`, double *min_box_dimension* = `-1.0`) 124

3.15.2.2 template<class X, class E> **DDQuadTree**< X, E >::∼**DDQuadTree** () 125

3.15.3 Member Function Documentation 125

3.15.3.1 template<class X, class E> void **DDQuadTree**< X, E >::points_near (**DDVector** & *r_position*, std::list< X ∗ > & *r_result_list*) 125

3.15.3.2 template<class X, class E = DDDefaultPointQuery<X>> void **DDQuadTree**< X, E >::tree_size (std::list< int > & *r_count_at_each_level*, std::list< int > & *r_leaves_at_each_level*) 125

3.16 DDAGUIntersection Class Reference 126

3.16.1 Detailed Description 126

3.16.2 Constructor & Destructor Documentation 127

3.16.2.1 DDAGUIntersection::DDAGUIntersection () `[inline]` 127

3.16.2.2 DDAGUIntersection::∼**DDAGUIntersection** () `[inline]` 127

3.16.3 Member Function Documentation 127

3.16.3.1 void DDAGUIntersection::reset () `[inline]` 127

3.16.4 Member Data Documentation 127

3.16.4.1 double **DDAGUIntersection::mAOrdering** 127

3.16.4.2 DD_RESULT **DDAGUIntersection::mAPointType** 127

3.16.4.3 double **DDAGUIntersection::mAX** 127

3.16.4.4 double **DDAGUIntersection::mAY** 128

3.16.4.5 double **DDAGUIntersection::mBOrdering** 128

3.16.4.6 DD_RESULT **DDAGUIntersection::mBPointType** 128

3.16.4.7 double **DDAGUIntersection::mBX** 128

3.16.4.8 double **DDAGUIntersection::mBY** 128

3.16.4.9 DD_RESULT **DDAGUIntersection::mIntersectionType** 128

3.16.4.10 **DDEdge**∗ **DDAGUIntersection::mpAEdge** 128

3.16.4.11 **DDEdgeUse**∗ **DDAGUIntersection::mpAEdgeUse** 128

3.16.4.12 **DDPoint**∗ **DDAGUIntersection::mpAPoint** 128

3.16.4.13 **DDEdge**∗ **DDAGUIntersection::mpBEdge** 129

3.16.4.14 **DDEdgeUse**∗ **DDAGUIntersection::mpBEdgeUse** 129

3.16.4.15 **DDPoint**∗ **DDAGUIntersection::mpBPoint** 129

3.16.4.16 bool **DDAGUIntersection::mVisited** 129

4 2D Geometric Routine **131**

4.1 DDUnionOperator Namespace Reference 131

4.1.1 Detailed Description 131

4.1.2 Function Documentation 132

4.1.2.1 DD_RESULT union_cw_loop_with_cw_loop (**DDLoop** ∗& *rp_loop_A*, **DDLoop** ∗& *rp_loop_B*, double *tolerance*) 132

4.1.2.2 DD_RESULT union_list_of_cw_loops (std::list< **DDLoop** ∗ > & *r_loop_list*, double *tolerance*) 132

4.1.2.3 DD_RESULT union_list_of_surfaces (std::list< **DDSurface** ∗ > & *r_surface_list*, double *tolerance*) 132

4.1.2.4 DD_RESULT union_surface_A_with_surface_B (**DDSurface** ∗& *rp_surface_A*, **DDSurface** ∗& *rp_surface_B*, double *tolerance*) . . 133

4.2 DDVirtualMergeOperator Namespace Reference 134

4.2.1 Detailed Description 134

4.2.2 Function Documentation 134

4.2.2.1 void create_virtual_surface_from_surface (**DDSurface** & *r_surface*, **DDVirtualSurface** ∗& *rp_virtual_surface*) 134

4.2.2.2 DD_RESULT merge_list_of_surfaces (std::list< **DDSurface** ∗ > & *r_list_of_surfaces*, **DDVirtualSurface** ∗& *rp_virtual_surface*, **DDVirtualLoop** ∗& *rp_border_loop*) 135

4.2.2.3 void merge_virtual_surface (**DDVirtualSurface** & *r_virtual_surface*, **DDSurface** ∗& *rp_surface*) 135

4.3 DDXOROperator Namespace Reference 136

4.3.1 Detailed Description 136

4.3.2 Function Documentation 136

4.3.2.1 DD_RESULT fast_xor_list_of_surfaces_A_with_list_of_surfaces_B (std::list< **DDSurface** ∗ > & *r_surfaces_A*, std::list< **DDSurface** ∗ > & *r_surfaces_B*, double *tolerance*) . . . 136

4.3.2.2 DD_RESULT xor_surface_A_with_surface_B (**DDSurface** ∗& *rp_surface_A*, **DDSurface** ∗& *rp_surface_B*, double *tolerance*, std::list< **DDSurface** ∗ > & *r_difference_AB_list*, std::list< **DDSurface** ∗ > & *r_difference_BA_list*) 137

4.4 DDOffsetOperator Namespace Reference 138

4.4.1 Detailed Description 138

4.4.2 Function Documentation 138

4.4.2.1 DD_RESULT create_offset_loop (**DDEdgeUseList** & *r_edgeuse_list*, double *tolerance*, double *distance*, std::list< **DDLoop** ∗ > & *r_similar_loops*, std::list< **DDLoop** ∗ > & *r_opposite_loops*) 138

4.4.2.2 DD_RESULT create_offset_surfaces (**DDVirtualSurface** & *r_surface*, double *tolerance*, double *distance*, std::list< **DDSurface** ∗ > & *r_surface_list*, **DDVirtualLoop** ∗ *p_boundary_loop* = 0) 139

4.4.2.3 DD_RESULT create_offset_surfaces (**DDSurface** & *r_surface*, double *tolerance*, double *distance*, std::list< **DDSurface** ∗ > & *r_surface_list*) 139

4.4.2.4 DD_RESULT subdivide_surfaces_at_offset (std::list< **DDSurface** ∗ > & *r_adjacent_surface_list*, double *tolerance*, double *distance*, std::map< **DDSurface** ∗, std::list< **DDSurface** ∗ > ∗ > & *r_offset_map*, std::map< **DDSurface** ∗, std::list< **DDSurface** ∗ > ∗ > & *r_original_map*, std::list< **DDSurface** ∗ > & *r_old_surfaces*) 140

4.5 DDAnalyticalGeometryUtil Namespace Reference 141

4.5.1 Detailed Description 141

4.5.2 Function Documentation 141

4.5.2.1 double calculate_area_enclosed_by_edgeuse_list (**DDEdgeUseList** & *r_list*) 141

4.5.2.2 double distance_point_point (**DDPoint** & *r_point1*, **DDPoint** & *r_point2*) 142

4.5.2.3 DD_RESULT find_intersection_edge_edge (**DDEdge** & *r_edge_1*, **DDEdge** & *r_edge_2*, double *tolerance*, **DDAGUIntersection** & *r_intersection_1*, **DDAGUIntersection** & *r_intersection_2*, **DDAGUIntersection** & *r_intersection_3*, **DDAGUIntersection** & *r_intersection_4*) 142

4.5.2.4 DD_RESULT find_intersection_ray_edge (**DDPoint** & *r_start_point*, **DDVector** & *r_direction*, **DDEdge** & *r_edge*, double *tolerance*, **DDAGUIntersection** & *r_intersection_1*, **DDAGUIntersection** & *r_intersection_2*) 142

4.5.2.5 DD_RESULT is_circle_point_on_arc (**DDPoint** & *r_point*, **DDArc** & *r_arc*, double *tolerance*, double & *r_parametric_parameter*) 143

4.5.2.6 DD_RESULT is_loop_inside_surface (**DDLoop** & *r_loop*, **DDSurface** & *r_surface*, double *tolerance*) 143

4.5.2.7 DD_RESULT is_point_enclosed_by_loop (**DDPoint** & *r_point*, **DDLoop** & *r_loop*, double *tolerance*) 144

4.5.2.8 DD_RESULT is_point_in_surface (**DDPoint** & *r_point*, **DDSurface** & *r_surface*, double *tolerance*) 144

4.5.2.9 bool is_within_tolerance (**DDPoint** & *r_point1*, **DDPoint** & *r_point2*, double *tolerance*) 144

4.5.2.10 DD_RESULT number_of_line_approximations_of_arc (**DDArc** & *r_arc*, double *tolerance*, int & *r_number_of_lines*) 145

4.6 DDModifyGeometryTool Namespace Reference 146

4.6.1 Detailed Description 147

4.6.2 Function Documentation 147

4.6.2.1 DD_RESULT copy_list_of_surfaces (std::list< **DDSurface** ∗ > & *r_surface_list*, std::list< **DDSurface** ∗ > & *r_new_surface_list*) 147

4.6.2.2 void copy_surface (**DDSurface** & *r_surface*, **DDSurface** ∗& *rp_surface*) 147

4.6.2.3 void create_line_approximation_for_arc (**DDArc** & *r_arc*, DDEdgeUseSense *sense*, double *tolerance*, std::list< **DDEdge** ∗ > & *r_edge_list*) 148

4.6.2.4 void create_surface_from_bounding_box (**DDBoundingBox** & *r_bounding_box*, **DDSurface** ∗& *rp_surface*) 148

4.6.2.5 void find_edgeuses_with_no_neighbor (**DDSurface** & *r_surface*, **DDEdgeUseList** & *r_edgeuse_list*) 148

4.6.2.6 DD_RESULT find_inside_edgeuses (**DDSurface** & *r_surface_A*, **DDSurface** & *r_surface_B*, std::map< **DDPoint** ∗, unsigned long > & *r_point_types*, unsigned int *overlapping_index*, unsigned int *opposite_overlapping_index*, double *tolerance*, unsigned int *inside_index*) 148

4.6.2.7 DD_RESULT intersect_loop_with_self (**DDLoop** & *r_loop*, double *tolerance*, std::map< **DDPoint** ∗, unsigned long > & *r_point_types*) 149

4.6.2.8 DD_RESULT intersect_surface_surface (**DDSurface** & *r_surface_A*, **DDSurface** & *r_surface_B*, double *tolerance*, std::map< **DDPoint** ∗, unsigned long > & *r_point_types*) 150

4.6.2.9 DD_RESULT remove_interior_loops (**DDSurface** & *r_surface*, double *tolerance*, **DDVirtualLoop** ∗& *rp_virtual_loop*) 150

4.6.2.10 void remove_small_edges (**DDSurface** & *r_surface*, double *tolerance*, **DDEdgeUseList** & *r_removed_edgeuses*) 151

4.6.2.11 void remove_small_edges (**DDLoop** ∗ *p_loop*, double *tolerance*, **DDEdgeUseList** & *r_removed_edgeuses*) 151

4.6.2.12 DD_RESULT replace_arc_with_line_if_within_tolerance (**DDArc** ∗& *rp_arc_in*, double *tolerance*, **DDLine** ∗& *rp_line_out*) 152

4.6.2.13 DD_RESULT replace_point_A_with_B (**DDPoint** & *r_point_A*, **DDPoint** & *r_point_B*, double *tolerance*) 152

4.6.2.14 DD_RESULT reverse_loop (**DDLoop** & *r_loop*) 152

4.6.2.15 DD_RESULT split_edge_use_at_point (**DDEdgeUse** & *r_edge_use_in*, **DDPoint** & *r_point*, double *tolerance*, **DDEdge** ∗& *rp_old_edge_in*, **DDEdgeUse** ∗& *rp_new_edge_use_out*) 153

4.6.2.16 DD_RESULT split_self_intersecting_loop (**DDLoop** & *r_loop*, std::list< **DDLoop** ∗ > & *r_result_loop*) 153

4.6.2.17 DD_RESULT subdivide_to_remove_interior_loops (**DDSurface** ∗& *rp_surface*, double *tolerance*, std::list< **DDSurface** ∗ > & *r_surface_list*) 154

5 Interface Routines **155**

5.1 DDInterface Namespace Reference 156

5.1.1 Detailed Description 156

5.1.2 Function Documentation 157

5.1.2.1 DD_RESULT calculate_bounding_box_for_list_of_surfaces (std::list< **DDSurface** ∗ > & *r_surface_list*, **DDBoundingBox** & *r_bounding_box*) 157

5.1.2.2 DD_RESULT close_dxf_file (std::ofstream & *r_dxf_file*) 157

5.1.2.3 std::string get_version () 157

5.1.2.4 DD_RESULT open_dxf_file_for_writing (std::string & *r_filename*, std::ofstream & *r_dxf_file*) 157

5.1.2.5 void write_edgeuse_list_to_file_dxf_polyline (**DDEdgeUseList** & *r_edgeuse_list*, std::ofstream & *r_dxf_file*, std::string & *r_layer_name*) 158

5.1.2.6 DD_RESULT write_list_of_surfaces_to_file_dxf_format (std::list< **DDSurface** ∗ > & *r_surface_list*, double *tolerance*, std::ofstream & *r_dxf_file*, std::string & *r_layer_name*) 158

5.1.2.7 DD_RESULT write_list_of_surfaces_to_file_dxf_format (std::list< **DDSurface** ∗ > & *r_surface_list*, double *tolerance*, std::string & *r_file_name*, std::string & *r_layer_name*) 159

5.1.2.8 DD_RESULT write_list_of_surfaces_to_file_dxf_format_through_subdivision (std::list< **DDSurface** ∗ > & *r_surface_list*, double *tolerance*, std::ofstream & *r_dxf_file*, std::string & *r_layer_name*) 159

5.1.2.9 DD_RESULT write_list_of_surfaces_to_file_dxf_format_through_subdivision (std::list< **DDSurface** ∗ > & *r_surface_list*, double *tolerance*, std::string & *r_file_name*, std::string & *r_layer_name*) 159

5.1.2.10 DD_RESULT write_surface_to_file_dxf_format (**DDSurface** & *r_surface*, double *tolerance*, std::ofstream & *r_dxf_file*, std::string & *r_layer_name*) 160

5.1.2.11 DD_RESULT write_surface_to_file_dxf_format (**DDSurface** & *r_surface*, double *tolerance*, std::string & *r_file_name*, std::string & *r_layer_name*) 160

5.2 DDMEMMaskReader Class Reference 161

5.2.1 Detailed Description 161

5.2.2 Constructor & Destructor Documentation 161

5.2.2.1 DDMEMMaskReader::DDMEMMaskReader () 161

5.2.2.2 DDMEMMaskReader::∼**DDMEMMaskReader** () 161

5.2.3 Member Function Documentation 162

5.2.3.1 DD_RESULT DDMEMMaskReader::create_mask_definitions (ifstream & *r_file_in*, double *tolerance*, std::list< std::string > & *r_layers*, **DDMaskDefinitions** & *r_mask_definitions*) 162

5.2.3.2 DD_RESULT DDMEMMaskReader::create_mask_definitions (ifstream & *r_file_in*, double *tolerance*, **DDMaskDefinitions** & *r_mask_definitions*) 162

5.2.3.3 void DDMEMMaskReader::create_surfaces_from_mask_file (ifstream & *r_file_in*, **DDMaskDefinitions** & *r_mask_definitions*) . . 162

5.2.3.4 void DDMEMMaskReader::SubDivideList (**DDMEMPolygonList** ∗ *list*, int *xInterval*, int *yInterval*, double *offset*, std::vector< **DDMEMPolygonList** ∗ > & *new_lists*) 163

References **165**

Appendix

Appendix 1: Example Code that uses GBL-2D 166

List of Figures

2.1 The Union of two isolated surfaces. 37

2.2 The XOR of two isolated surfaces. 38

2.3 Imprint one surface with another. 39

2.4 Imprinting a group of interconnected surfaces (Surfaces 1,2,3,4) by one isolated surface (Surface 5). 39

2.5 New isolated surface ”B” created by offsetting surface ”A”. 40

2.6 Imprinting a group of connected surfaces (Surfaces 1,2,3,4,5) at an offset from a subgroup of connected surfaces (Surfaces 1,2,3). 41

List of Tables

2.1 Boolean OR Operation 37

2.2 Boolean XOR Operation 38

Chapter 1

Introduction and Background

GBL-2D is a software library of C++ classes and routines whose central theme is the performance of Boolean operations on 2D geometry data sets. It was originally developed as a geometric modeling engine for use with a separate software tool, called SummitView [1], that manipulates the 2D mask sets created by designers of Micro-Electro-Mechanical Systems (MEMS). These 2D mask sets are contained in MEM format files (see [2]). To understand why such a library is important in this context, a brief overview of the development of MEMS design tools at Sandia is useful.

The development of advanced multi-level surface micro-machining (SMM) micro-fabrication technologies such SUMMiT V [3] has enabled the ability to create very complex 3D MEMS devices. The 3D structure of a MEMS device results from the interaction of the individual fabrication process steps and the associated set of 2D layout masks created by the designer. However, because the 2D masks do not directly reveal 3D geometry, the structures that result from this interaction can be very difficult to visualize without special tools. In addition, the development of a finite-element mesh suitable for accurately representing the geometric complexities of the design for needed analysis is likewise problematic.

To help address these problems, Sandia developed several computational design tools [4]: the 2D Process Visualizer [5, 6], a Design Rule Checking System [7], and a first generation 3D Geometry Modeler [8]. Of particular note here is the 3D Geometry Modeler, which is a solid geometry modeler based on the ACIS kernel [9]. Its function is to simulate the interaction between the SUMMiT V process and a designer's mask set, at the geometric level, to create a 3D model suitable for visualization, rapid prototyping, and analysis.

The usefulness of the 3D Geometry Modeler as a design tool at Sandia prompted a subsequent effort to improve upon it. The main limitations of this first generation tool had to do with robustness problems, large computational costs, and difficulty producing numerical meshes of the final 3D models.

The limitations with robustness and computational cost are directly linked to the use of the ACIS 3D Geometric Modeling Engine for data structures and geometric routines. ACIS is designed as an extremely general tool that can be used as "the geometry foundation within virtually any end user 3D modeling application" [9]. This broad scope requires the data structures and routines to be sophisticated and flexible enough to handle a wide variety of modeling situations. However, this sophistication and flexibility comes at the price of increased size and complexity. Extensive experience with the 3D Geometry Modeler made it clear that the use of ACIS data structures and

routines to perform numerous large and complicated 3D Boolean operations was in large measure responsible for the low robustness and large computational costs of the 3D Modeler.

The difficulty in producing numerical meshes comes from the complicated 3D nature of the final ACIS models. The 3D Geometry Modeler simulates each step of the SUMMiT V process by manipulating the model geometry to produce results consistent with the actual fabrication steps. Each process step is simulated in order, with the model that results from the current step being used as input for the next step. Thus the final geometry of ACIS model evolves as each process step is simulated. Intricate MEMS models that are produced using this development method are more complicated than those produced by modeling the geometry directly. It is this added complexity that is largely responsible for the difficulty in producing numerical meshes.

The SummitView computer code [1] was developed to overcome the aforementioned problems of the 3D Geometry Modeler. A key aspect of this tool is that all geometric objects manipulated by the code are what can be called "$2\frac{1}{2}$D." That is, they consist of a 2D geometry with an associated thickness. Using $2\frac{1}{2}$ D geometry allows the modeling problem to be completely solved using 2D data structures and 2D geometric routines. Functionally, SummitView performs essentially the same role as the 3D Geometry modeler. However, because it is based on 2D instead of 3D data structures and operations, it has significant speed and robustness advantages. In addition, it can be far simpler to produce numerical meshes of $2\frac{1}{2}$D geometries than of 3D geometries

In order to efficiently perform the various 2D Boolean operations that represent actions occurring during different process steps the GBL-2D geometric Boolean library described in this report was developed in parallel with SummitView. Developing a custom modeling engine as part of this work had many advantages. For example, by freeing the developer from using ACIS routines, it allowed for source-level control of data structures and geometric routines. In addition to allowing greater flexibility, this also allows for data structures and routines to be stream-lined for a specific application. Stream-lined data structures and routines leads directly to reduced computational costs and improved overall robustness. These advantages have been verified by tests comparing SummitView 1.0 (which uses GBL-2D) with the 3D Geometry modeler (which uses ACIS) that demonstrated a consistent speed-up of approximately 2 orders of magnitude.

This report describes version 1.0 of GBL-2D, a geometric Boolean library for 2D objects.

Chapter 2

Overview of GBL-2D 1.0

GBL-2D is a software library, written in C++, that consists of a set of classes and routines. The classes primarily represent 2D geometric data and relationships. The geometric data classes are designed to allow multiple instances of the classes to be created. For example, each point in 2D space would be represented by an instance of the DDPoint class. The routines contain various geometric algorithms and utility functions. Namespaces are used to group these routines according to their function. All classes and namespaces in the library begin with the prefix "DD" which identifies them as belonging to the GBL-2D. For example, the class DDPoint is the GBL-2D representation of a point.

Listings from two simple example codes that illustrate how to use the GBL-2D C++ classes and associated routines are found in Appendix 1.

2.1 Supported Compilers

The GBL-2D software library has been tested with the following 32 bit compilers.

- Microsoft Visual C++ 6.0 with STLport Version 4.5.3 [10]
- Microsoft Visual C++ .NET 2003
- Microsoft Visual c++ 2005 Express Edition
- G++ 3.2 or greater

2.2 2D Geometry Classes

The GBL-2D data classes represent three types of geometric data: 2D geometry, topology and virtual topology.

2.2.1 Geometry Data Structures

2.2.1.1 DDPoint

A DDPoint is a 0-dimensional entity that represents a location in 2-dimensional space. Each DDPoint instance has an X coordinate and a Y coordinate. Consequently the points represented by DDPoint are constrained to the X-Y plane.

2.2.1.2 DDEdge

DDEdge is an abstract base class that represents a 1-dimensional entity in 2-dimensional space. Each DDEdge references two instances of DDPoint. One DDPoint defines the start of the edge and the other defines the end of the edge. Currently there are two types of DDEdge implemented: DDLine and DDArc.

2.2.1.3 DDLine

DDLine is a 1-dimensional entity that defines a line segment in 2-dimensional space. Each instance of DDLine references two instances of DDPoint. One DDPoint defines the start of the line segment and the other DDPoint defines the end of the line segment. Because an instance of DDPoint is used for each end point of a DDLine, the represented line is constrained to the X-Y plane.

2.2.1.4 DDArc

DDArc is a 1-dimensional entity that defines an arc segment in 2-dimensional space. Each instance of DDArc references three instances of DDPoint. One DDPoint defines the start of the arc segment and one DDPoint defines the end point of the arc segment. The third instance of DDPoint is used to define the center point of the parent circle on which the arc lies. Every arc is defined between the start point and the end point in the counter clockwise direction. Because instances of DDPoint are used for each end point and the center point of a DDArc, the represented arc is constrained to the X-Y plane.

2.2.2 Topology Classes

2.2.2.1 DDEdgeUse

DDEdgeUse associates an instance of DDEdge with a sense. The sense defines the direction that the edge is being used. An edge that is used in the forward sense is traversed from the edge's

start point to end point. An edge that is used in the backward sense is traversed from the edge's end point to start point. An instance of DDEdge may be referenced once in the forward sense and once in the backwards sense. Each instance of DDEdgeUse may only belong to one instance of DDLoop.

2.2.2.2 DDLoop

DDLoop is an ordered list of DDEdgeUses. The purpose a DDLoop is to define the boundary for a region of 2D space. To be considered a valid DDLoop the following constraints must be observed.

- The DDEdge within the DDLoop must form a closed circuit. Specifically, the first DDPoint in the list and the last DDPoint in the list must be the same instance.
- DDEdgeUses that are adjacent in the list must reference instances of adjacent DDEdges.
- DDEdges are considered adjacent if one instance of DDPoint is shared between both instances of DDEdge. Specifically the shared DDPoint must correspond to the end point of one DDEdgeUse and the start point of the adjacent DDEdgeUse. Thus the sense that each DDEdges is being used defines which end point must be shared.
- A DDLoop must not be self-intersecting. This means that geometry defined by the DDEdges must not overlap other DDEdges in the same loop.

2.2.2.3 DDSurface

DDSurface defines a surface in 2D space. A DDSurface contains one instance of DDLoop as an exterior loop and zero to many instances of DDLoop as interior loops. The exterior DDLoop defines the outside bounds of the surface. Any interior DDLoops define the holes in the surface. To be considered a valid DDSurface the following constraints must be observed.

- The DDEdges and DDEdgeUses of the exterior DDLoop are ordered in such a way that traversing the loop in the forward direction traverses the geometry in the counter clockwise direction.
- The DDEdges and DDEdgeUses of any interior DDLoops are ordered such that traversing the loop in the forward direction traverses the geometry in the clockwise direction.
- The geometry defined in one instance of DDLoop must not overlap any geometry defined in any other DDLoop in the same DDSurface. The bounds defined in each interior DDLoop must reside completely inside the bounds defined by the exterior DDLoop.
- No interior DDLoop may reside inside the bounds defined by another interior DDLoop.

2.2.3 Virtual Topology Classes

2.2.3.1 DDVirtualLoop

A DDVirtualLoop is similar to a DDLoop in that it is an ordered list of DDEdgeUses. A DDVirtualLoop follows the same constraints as a DDLoop. However, DDVirtualLoops are not registered with the DDEdgeUses they contain. This allows a DDEdgeUse to belong to one DDLoop and multiple DDVirtualLoops. One common use for DDVirtualLoops is to define the boundaries of a group of interconnected DDSurfaces. This allows a virtual merge of the DDSurfaces to be performed without destroying the original DDSurfaces.

2.2.3.2 DDVirtualSurface

A DDVirtualSurface is similar to a DDSurface in that defines a surface in 2D space. A DDVirtualSurface follows the same constraints as a DDSurface. However, a DDVirtualSurface contains DDVitrualLoops instead of DDLoops. As with DDVirtualLoops, one common use of DDVirtualSurfaces is to define the total area covered by a group of interconnected DDSurfaces without destroying the original DDSurfaces.

2.3 2D Geometric Routines

There are two types of geometric routines in the 2D Boolean Library. The first type are those routines that do not respect neighboring topology. These routines are only defined for DDSurfaces that are not topologically connected to any other DDSurfaces. Two DDSurfaces are topologically connected if one or more DDEdges are shared through multiple DDEdgeUses. Thus in order for these routines to work correctly each DDEdge that defines the geometry of a DDSurface can only belong to one DDEdgeUse, DDLoop and DDSurface. The purpose of these routines is to create new isolated DDSurfaces from the original isolated DDSurfaces.

The second type of routines are those that respect neighboring topology. These routines are designed to work with groups of surfaces that are topologically connected. The purpose of these routines is to create new connected DDSurfaces from the original connected DDSurfaces.

The following sections give an overview of the major geometric routines contained within GBL-2D.

2.3.1 Union Operation Routines

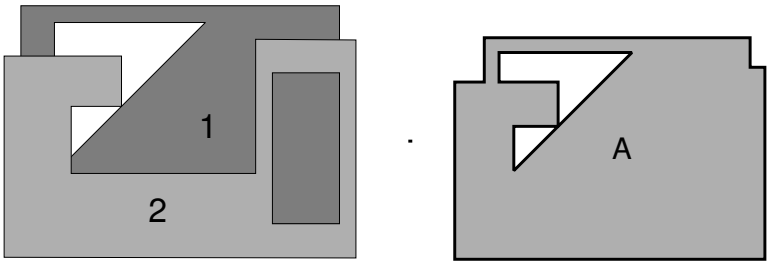
The purpose of the Union Operation is to combine DDSurfaces in a manner analogous to the Boolean OR operation. Table 2.1 shows the truth table of the OR operation. The routines that

define the OR Operation are contained in the DDUnionOperator namespace. One or many isolated DDSurfaces may be combined. The primitives that make up the original DDSurfaces are used to create the new DDSurfaces. Thus the original DDSurfaces are consumed by the routines.

We note that this routine was originaly designed for merging all of the surfaces that reside on one layer of a MEMS mask set.

Table 2.1. Boolean OR Operation

A	B	OR
T	T	T
T	F	T
F	T	T
F	F	F



(a) Two isolated surfaces that overlap.

(b) Result of the Union operation.

Figure 2.1. The Union of two isolated surfaces.

This routine only functions properly if each input DDSurface is topologically isolated. The result is one or many isolated DDSurface. For example, Figure 2.1(a) shows two overlapping topologically isolated DDSurfaces. Figure 2.1(b) shows the one topologically isolated DDSurface that is a result of the Union operation.

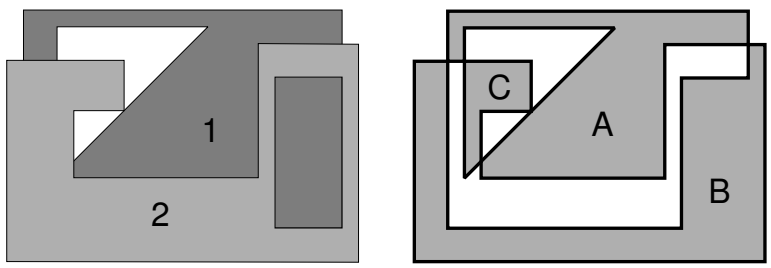
A version of this routine also exists for use with DDVirtualSurface and DDVirtualLoops. The purpose of these routines is to create DDVirtualSurfaces that represent the area covered by a set of interconnected DDSurfaces. The primitves that make up the new DDVirtualSurfaces come from the original DDSurfaces. However, because of the nature of virtual geometry, the original DDSurfaces are not consumed. Thus, the virtual merge routines provide a means to virtually merge interconnected DDSurfaces without destroying the original DDSurfaces or their geometric relationships with other DDSurfaces. These virtual merge routines are contained in the DDVirtualMergeOperator namespace.

2.3.2 XOR Operation Routines

The purpose of the XOR operation is to combine DDSurfaces in a way analogous to the Boolean XOR operation. Table 2.2 shows the truth table of the XOR operation. The routines that define the XOR Operation are contained in the DDXOROperator namespace.

Table 2.2. Boolean XOR Operation

A	B	XOR
T	T	F
T	F	T
F	T	T
F	F	F



(a) Two isolated surfaces that overlap.

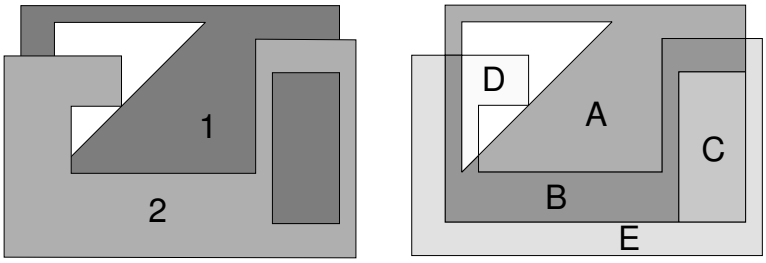
(b) Result of the XOR operation.

Figure 2.2. The XOR of two isolated surfaces.

This routine only functions properly if each input DDSurface is topologically isolated. The result of this routine is one or many isolated DDSurface. For example, Figure 2.2(a) shows two overlapping topologically isolated DDSurfaces. Figure 2.2(b) shows the three topologically isolated DDSurfaces that result from the XOR operation.

2.3.3 Imprinting Routines

The purpose of the Imprinting routines is to subdivide a DDSurface or group of interconnected DDSurfaces by the geometry of another DDSuface. The DDSurface that is used to guide the subdivision must be topologically isolated from any other DDSuface. This is because the primitives that make up the DDSurface are used to form new DDSurfaces. The routines that define the Imprinting Operation are contained in the DDImprintOperator namespace.

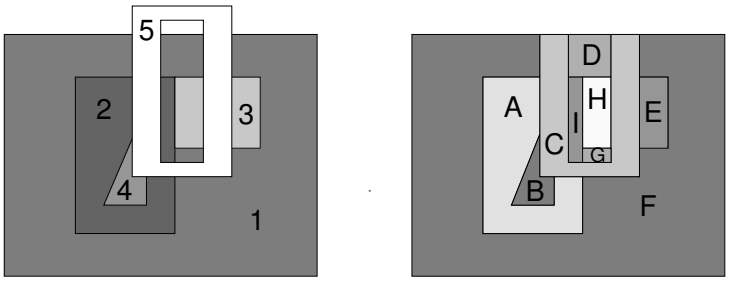


(a) Two isolated surfaces that overlap.

(b) Result of the Imprint operation.

Figure 2.3. Imprint one surface with another.

There are several possible results of the Imprinting operations. For example, Figure 2.3(a) shows two overlapping, topologically isolated DDSurfaces. Figure 2.3(b) shows the result of the imprinting DDSurface 1 with DDSurface 2. DDSurfaces ”A”,”B” and ”C” are interconnected with each other. DDSurfaces ”C”,”D” are isolated.



(a) A group of interconnected surfaces overlapped by one isolated surface.

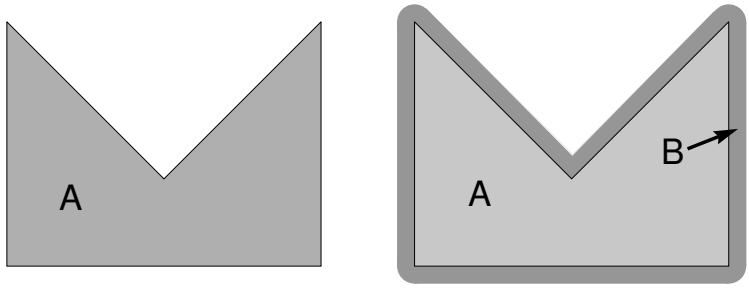
(b) Result of the Imprint operation.

Figure 2.4. Imprinting a group of interconnected surfaces (Surfaces 1,2,3,4) by one isolated surface (Surface 5).

Figure 2.4(a) shows a group of four interconnected DDSurfaces, (Surfaces 1,2,3,4) overlapped by a topologically isolated DDSurfaces (Surface 5). Figure 2.4(b) shows the new set of interconnected DDSurfaces that result from the imprinting. The isolated DDSurface that represents the remainder of surface ”5” is not shown.

2.3.4 Offset Routines

The purpose of the Offset routines is to create new DDSurfaces using a rolling ball offset of the boundaries of the original DDSurfaces. The routines that define the Imprinting Operation are contained in the DDOffsetOperator namespace. Single isolated DDSurfaces or multiple interconnected DDSurfaces can be offset. The offset can both expand or shrink the original areas.



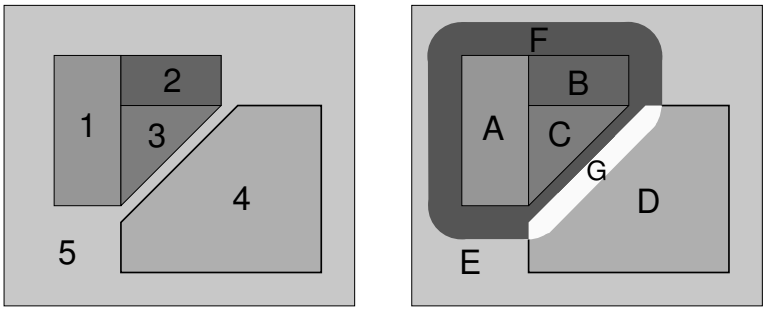
(a) One isolated surface.

(b) Result of the Offset operation.

Figure 2.5. New isolated surface ”B” created by offsetting surface ”A”.

For example, Figure 2.5(a) shows a single isolated DDSurface (surface A). Figure 2.5(b) shows a new DDSurface (surface B) that is the result of offsetting the original DDSurface. DDSurface ”B” covers the same area as surface ”A” plus the area of the offset. The two surfaces are shown overlapping in Figure 2.5(b) to show the effect of the offset.

Routines are also provided to subdivide a set of interconnected DDSurfaces at an offset. These routines use a combination of the the rolling ball offset routines and imprinting routines. For example, Figure 2.6(a) shows a set of interconnected DDSurfaces (Surfaces 1,2,3,4,5). Surfaces ”1”, ”2” and ”3” were offset as a group of adjacent interconnected DDSurfaces. The complete original set of DDSurfaces was then imprinted based upon the new offset DDSurface. Figure 2.6(b) shows the result of subdividing the original set of DDSurfaces based on the offset of a subset. All seven DDSurfaces are interconnected.



(a) A group of connected surfaces.

(b) Result of the Offset operation.

Figure 2.6. Imprinting a group of connected surfaces (Surfaces 1,2,3,4,5) at an offset from a subgroup of connected surfaces (Surfaces 1,2,3).

Chapter 3

2D Data Classes

This chapter describes the interfaces for the geometric data classes. Each section contains a general description of a class and a list of its interface methods. A description of each interface method is then provided.

3.1 DDVector Class Reference

The DDVector represents a vector in 2d space. All of the standard operators are overloaded.

`#include <DDVector.hpp>`

Public Member Functions

- **DDVector** ()
- **DDVector** (double x, double y)
- **DDVector** (double xy[2])
- **DDVector** (const **DDVector** &copy_from)
- void **set** (double x, double y)
- void **set** (double xy[2])
- void **x** (double x)
- void **y** (double y)
- double **x** () const
- double **y** () const
- void **get_xy** (double xy[2])
- void **get_xy** (double &x, double &y)
- double **length** ()
- double **length_squared** ()
- void **perpendicular** ()
- double **normalize** ()
- **DDVector** & **operator+=** (const **DDVector** &vector)

- **DDVector** & **operator-=** (const **DDVector** &vector)
- **DDVector** & **operator** ∗**=** (const double scalar)
- **DDVector** & **operator/=** (const double scalar)
- **DDVector operator-** ()
- **DDVector** & **operator=** (const **DDVector** &from)
- void ∗ **operator new** (size_t size)
- void **operator delete** (void ∗p, void ∗)

Friends

- **DDVector operator+** (const **DDVector** &vector1, const **DDVector** &vector2)
- **DDVector operator-** (const **DDVector** &vector1, const **DDVector** &vector2)
- double **operator** ∗ (const **DDVector** &vector1, const **DDVector** &vector2)
- double **operator%** (const **DDVector** &vector1, const **DDVector** &vector2)
- **DDVector operator** ∗ (const **DDVector** &vector, const double scalar)
- **DDVector operator** ∗ (const double scalar, const **DDVector** &vector)
- **DDVector operator/** (const **DDVector** &vector, const double scalar)
- bool **operator==** (const **DDVector** &vector1, const **DDVector** &vector2)
- bool **operator!=** (const **DDVector** &vector1, const **DDVector** &vector2)

3.1.1 Detailed Description

The DDVector represents a vector in 2d space. All of the standard operators are overloaded.

3.1.2 Constructor & Destructor Documentation

3.1.2.1 DDVector::DDVector ()

Constructor.

Initializes DDVector to (0.0 , 0.0).

3.1.2.2 DDVector::DDVector (double *x*, double *y*)

Constructor.

Creates a DDVector from two components.

3.1.2.3 DDVector::DDVector (double *xy*[2])

Constructor.

Creates a DDVector from a tuple.

3.1.2.4 DDVector::DDVector (const DDVector & *copy_from*)

Copy Constructor.

3.1.3 Member Function Documentation

3.1.3.1 void DDVector::get_xy (double & *x*, double & *y*)

Get the x and y components.

3.1.3.2 void DDVector::get_xy (double *xy*[2])

Get the x and y components in the form of the tuple *xy[2]*.

3.1.3.3 double DDVector::length ()

Calcualtes the length of the DDVector.

3.1.3.4 double DDVector::length_squared ()

Calculates the length of the DDVector squared.

3.1.3.5 double DDVector::normalize ()

Normalizes the DDVector.

Return values:

returns the length of the original DDVector;

3.1.3.6 DDVector& DDVector::operator ∗= (const double *scalar*)

Overload operator: multiplication [this = this ∗ scalar].

3.1.3.7 void DDVector::operator delete (void ∗ *p*, void ∗)

A macro to define in place memory allocation methods.

3.1.3.8 void∗ DDVector::operator new (size_t *size*)

A macro to define memory allocation methods.

3.1.3.9 DDVector& DDVector::operator+= (const DDVector & *vector*)

Overload operator: compound addition. [this = this + vector].

3.1.3.10 DDVector DDVector::operator- ()

Overload operator: unary negation.

3.1.3.11 DDVector& DDVector::operator-= (const DDVector & *vector*)

Overload operator: compound subtraction. [this = this - vector].

3.1.3.12 DDVector& DDVector::operator/= (const double *scalar*)

Overload operator: division [this = this / scalar].

3.1.3.13 DDVector& DDVector::operator= (const DDVector & *from*)

Equals.

3.1.3.14 void DDVector::perpendicular ()

Transforms the DDVector into a perpendicular one. x=-y and y=x.

3.1.3.15 void DDVector::set (double *xy*[2])

Change the DDVector components to *xy[0],ey[1]*.

3.1.3.16 void DDVector::set (double *x*, double *y*)

Change the DDVector components to *x* and *y*.

3.1.3.17 double DDVector::x () const

Returns the x component.

3.1.3.18 void DDVector::x (double *x*)

Set the x component.

3.1.3.19 double DDVector::y () const

Returns the y component.

3.1.3.20 void DDVector::y (double *y*)

Set the y component.

3.1.4 Friends And Related Function Documentation

3.1.4.1 DDVector operator ∗ (const double *scalar*, const DDVector & *vector*) `[friend]`

Scalar ∗ Vector.

3.1.4.2 DDVector operator ∗ (const DDVector & *vector*, const double *scalar*) `[friend]`

Vector ∗ scalar.

3.1.4.3 double operator ∗ (const DDVector & *vector1*, const DDVector & *vector2*) [friend]

The cross product between two DDVector.

3.1.4.4 bool operator!= (const DDVector & *vector1*, const DDVector & *vector2*) [friend]

Inequality operator.

3.1.4.5 double operator% (const DDVector & *vector1*, const DDVector & *vector2*) [friend]

Vector dot product.

3.1.4.6 DDVector operator+ (const DDVector & *vector1*, const DDVector & *vector2*) [friend]

Vector addition.

3.1.4.7 DDVector operator- (const DDVector & *vector1*, const DDVector & *vector2*) [friend]

Vector subtraction.

3.1.4.8 DDVector operator/ (const DDVector & *vector*, const double *scalar*) [friend]

Vector / scalar.

3.1.4.9 bool operator== (const DDVector & *vector1*, const DDVector & *vector2*) [friend]

Equality operator.

3.2 DDPoint Class Reference

The DDPoint represents a point in 2d space.

`#include <DDPoint.hpp>`

Public Member Functions

- **DDPoint** (unsigned long id=0xffffffff)
- ~**DDPoint** ()
- void **get_ID** (unsigned long &r_id)
- unsigned long **get_ID** ()
- void **get_registered_edges** (std::list< **DDEdge** * > &r_edge_list)
- bool **are_edges_registered** ()
- DD_RESULT **register_edge** (**DDEdge** *p_edge)
- DD_RESULT **unregister_edge** (**DDEdge** *p_edge)
- **DDPoint** & **operator=** (const **DDPoint** &r_from_point)
- **DDPoint** & **operator+=** (**DDVector** &r_vector)

Static Public Member Functions

- DD_RESULT **get_point** (unsigned long &r_id, **DDPoint** *&rp_point)
- void **get_instance_map** (**DDHashMap**< unsigned long, **DDPoint** * > &r_point_map)
- unsigned long **get_instance_map_size** ()
- void **delete_all_instances** ()
- DD_RESULT **delete_point** (**DDPoint** *&rp_point)
- unsigned long **get_ID_generator** ()

Public Attributes

- double **mX**
- double **mY**

Friends

- **DDVector operator-** (const **DDPoint** &r_point_1, const **DDPoint** &r_point_2)
- bool **operator==** (**DDPoint** &r_p1, **DDPoint** &r_p2)
- bool **operator!=** (**DDPoint** &r_p1, **DDPoint** &r_p2)

3.2.1 Detailed Description

The DDPoint represents a point in 2d space.

Author:
Corey McBride

Date:
02/04/03

3.2.2 Constructor & Destructor Documentation

3.2.2.1 DDPoint::DDPoint (unsigned long *id* = `0xffffffff`)

A macro to define memory allocation methods. Constructor.

Initializes the position to (0.0 , 0.0).

Parameters:

id If *id* is 0 then the DDPoint will not be added to the static list of all DDPoints and will not be given a unique id. Instead the instance will have an id of 0. If *id* is 0xffffffff then the instance will be given a unique ID based upon the order of creation of the instance. The DDPoint will also be added to the list of all DDPoints. If *id* is between 0 and 0xffffffff then the instance will be assigned the ID *id*. However, the IDs must be assigned in assending order or the constructor will assert.

3.2.2.2 DDPoint::~DDPoint ()

Destructor.

If the DDPoint is part of any **DDEdge**(p. 55) then the method will assert. Removes the DDPoint from the static list of all DDPoints.

3.2.3 Member Function Documentation

3.2.3.1 bool DDPoint::are_edges_registered ()

Determines if the any DDEdges are registered with the DDPoint.

Returns true if the there are any DDEdges registered with this instance. This indicates that the DDPoint is not being used by any other primitives.

3.2.3.2 void DDPoint::delete_all_instances () [static]

This function deletes all instance of the class.

This function should only be used to clean up memory at the end of a program. Higher order objects should be deleted first to make sure that this instance is not being used by another class.

3.2.3.3 DD_RESULT DDPoint::delete_point (DDPoint *& *rp_point*) [static]

Deletes the DDPoint if it is not being used by any **DDEdge**(p. 55).

3.2.3.4 unsigned long DDPoint::get_ID ()

Retrieves the ID of the DDPoints.

3.2.3.5 void DDPoint::get_ID (unsigned long & *r_id*)

Retrieves the ID of the DDPoints.

3.2.3.6 unsigned long DDPoint::get_ID_generator () [static]

This function returns the ID of the last instance created; A new instance that is created and assigned an id manually should be assigned an id that is greater that the returned value.

3.2.3.7 void DDPoint::get_instance_map (DDHashMap< unsigned long, DDPoint * > & *r_point_map*) [static]

Get a map containing pointers to all of the instances of the class indexed by instance id.

Parameters:

r_point_map The map where the pointers are to be stored. The instance id is the key to the map.

3.2.3.8 unsigned long DDPoint::get_instance_map_size () [static]

Returns the number of instances of the class. See the description of the constructor for more information on which instances are saved in the instance map.

3.2.3.9 DD_RESULT DDPoint::get_point (unsigned long & *r_id*, DDPoint *& *rp_point*) [static]

Get the DDPoint with the id *r_id*.

Parameters:

r_id The id of the DDPoint to get.

rp_point The pointer used to return the DDPoint.

Return values:

DD_SUCCESS If the DDPoint was found.

DD_ERROR_PRIMITIVE_NOT_FOUND If the DDPoint was not found.

3.2.3.10 void DDPoint::get_registered_edges (std::list< DDEdge * > & *r_edge_list*)

Retrieves the list of DDEdges that are registered with this instance of DDPoint.

Parameters:

r_edge_list The list used to return the DDEdges that are registered with this instance of DDPoint.

3.2.3.11 DDPoint& DDPoint::operator+= (DDVector & *r_vector*)

Translates the DDPoint by a **DDVector**(p. 43).

[This = This + Vector]

3.2.3.12 DDPoint& DDPoint::operator= (const DDPoint & *r_from_point*)

Sets the coordinates of this instance of DDPoint to the cooridinates of *r_from_point*.

3.2.3.13 DD_RESULT DDPoint::register_edge (DDEdge * *p_edge*)

Registers the **DDEdge**(p. 55) that is using this instance of DDPoint.

Return values:

DD_X_NULL_POINTER If *rp_edge* contains a NULL pointer.

DD_SUCCESS If the **DDEdge**(p. 55) was registered.

3.2.3.14 DD_RESULT DDPoint::unregister_edge (DDEdge * *p_edge*)

Unregisters the **DDEdge**(p. 55) that used this instance of DDPoint.

The DDPoint does not notify the **DDEdge**(p. 55) that it was unregistered. Thus it is the developers responsibility to insure that the link from DDPoint to **DDEdge**(p. 55) is maintained properly.

Return values:

DD_SUCCESS If the **DDEdge**(p. 55) was unregistered.

DD_X_NULL_POINTER If *rp_edge* contains a NULL pointer.

3.2.4 Friends And Related Function Documentation

3.2.4.1 bool operator!= (DDPoint & *r_p1*, DDPoint & *r_p2*) `[friend]`

Checks the coordinates of the DDPoints to see if they are not equal.

3.2.4.2 DDVector operator- (const DDPoint & *r_point_1*, const DDPoint & *r_point_2*) `[friend]`

Creates a vector from *r_point_1* - *r_point_2*.

3.2.4.3 bool operator== (DDPoint & *r_p1*, DDPoint & *r_p2*) `[friend]`

Checks the coordinates of the DDPoints to see if they are equal.

Checks to see if the coordinates are exactly equal. There is no room for tolerance.

3.2.5 Member Data Documentation

3.2.5.1 double DDPoint::mX

The X position of the DDPoint in 2d space.

This data member is public on purpose.

3.2.5.2 double DDPoint::mY

The Y posiition of the DDPoint in 2d space.

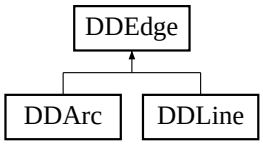
This data member is public on purpose.

3.3 DDEdge Class Reference

The DDEdge represents an edge in 2d space.

`#include <DDEdge.hpp>`

Inheritance diagram for DDEdge::



Public Member Functions

- **DDEdge** (**DDPoint** ∗p_start_point, **DDPoint** ∗p_end_point, unsigned long id=0xffffffff)
- virtual ∼**DDEdge** ()
- void **get_ID** (unsigned long &r_id)
- unsigned long **get_ID** ()
- virtual void **calculate_bounding_box** (**DDBoundingBox** &r_bounding_box)=0
- virtual void **get_edge_type** (DDEdgeType &r_edge_type)=0
- void **get_end_points** (**DDPoint** ∗&rp_start_point, **DDPoint** ∗&rp_end_point)
- virtual DD_RESULT **replace_point** (**DDPoint** ∗p_old_point, **DDPoint** ∗p_new_point)
- virtual DD_RESULT **replace_point_and_update** (**DDPoint** ∗p_old_point, **DDPoint** ∗p_-new_point)
- void **get_edge_uses** (**DDEdgeUse** ∗&rp_forward_edge_use, **DDEdgeUse** ∗&rp_backward_-edge_use)
- DD_RESULT **register_edge_use** (**DDEdgeUse** ∗p_edge_use)
- DD_RESULT **unregister_edge_use** (**DDEdgeUse** ∗p_edge_use)
- virtual void **create_isolated_copy_of_edge** (**DDEdge** ∗&rp_copy)=0
- virtual void **create_connected_copy_of_edge** (**DDPoint** ∗p_start_point, **DDPoint** ∗p_end_-point, **DDEdge** ∗&rp_copy)=0
- virtual void **create_integrated_copy_of_edge** (**DDEdge** ∗&rp_copy)=0
- virtual void **calculate_midpoint** (**DDPoint** &r_midpoint)=0
- virtual void **calculate_point** (double t, **DDPoint** &r_point)=0
- void **get_end_point_ids** (unsigned long &r_start, unsigned long &r_end)
- unsigned long **get_instance_map_size** ()
- virtual DD_RESULT **is_equal** (**DDEdge** &r_edge, double tolerance)=0
- virtual DD_RESULT **offset_edge** (double distance)=0
- virtual double **calculate_length** ()=0

Static Public Member Functions

- DD_RESULT **get_edge** (unsigned long &r_id, **DDEdge** ∗&rp_edge)
- void **get_instance_map** (**DDHashMap**< unsigned long, **DDEdge** ∗ > &r_edge_map)
- void **delete_all_instances** ()
- unsigned long **get_ID_generator** ()
- DD_RESULT **delete_edge** (**DDEdge** ∗&rp_edge)

3.3.1 Detailed Description

The DDEdge represents an edge in 2d space.

Author:
Corey McBride

Date:
02/04/03

This is a pure virtual class. Because of the 2d nature of the domain. Each DDEdge can only be used in two DDEdgeUses. One **DDEdgeUse**(p. 80) in the forward sense and one in the backward sense.

3.3.2 Constructor & Destructor Documentation

3.3.2.1 DDEdge::DDEdge (DDPoint ∗ *p_start_point*, DDPoint ∗ *p_end_point*, unsigned long *id* = `0xffffffff`)

Constructor.

Creates an DDEdge in the forward sense, this means that the DDEdge extends from the **DDPoint**(p. 49) *p_start_point* to the **DDPoint**(p. 49) *p_end_point*. If either **DDPoint**(p. 49) pointer is NULL there will be an assertion failure. This instance is registered with each **DDPoint**(p. 49) by calling the DDPoint's register routine.

Parameters:

p_start_point The instance of **DDPoint**(p. 49) to use as the start point of the DDEdge.

p_end_point The the instance of **DDPoint**(p. 49) to use as the end point of the DDEdge.

id If *id* is 0 then the DDEdge will not be added to the static list of all DDEdges and will not be given a unique id. Instead the instance will have an id of 0. If *id* is 0xffffffff then the instance will be given a unique ID based upon the order of creation of the instance. The DDEdge will also be added to the list of all DDEdges. If *id* is between 0 and 0xffffffff then the instance will be assigned the ID *id*. However, the IDs must be assigned in assending order or the constructor will assert.

3.3.2.2 virtual DDEdge::~DDEdge () `[virtual]`

Destructor.

If the DDEdge is part of any DDEdgeUses then the method will assert. This instance is unregistered with each **DDPoint**(p. 49) by calling the DDPoint's unregister routine. The DDEdge is removed from the static list of all DDEdge instances.

3.3.3 Member Function Documentation

3.3.3.1 virtual void DDEdge::calculate_bounding_box (DDBoundingBox & *r_bounding_box*) `[pure virtual]`

Calculates the bounding box of the DDEdge. The routine is defined be each derived class.

Parameters:

r_bounding_box The **DDBoundingBox**(p. 118) used to store the calculated bounding box.

Implemented in **DDArc** (p. 73), and **DDLine** (p. 66).

3.3.3.2 virtual double DDEdge::calculate_length () `[pure virtual]`

Calculates the length of the edge. This routine is defined by each derived class.

Implemented in **DDArc** (p. 73), and **DDLine** (p. 66).

3.3.3.3 virtual void DDEdge::calculate_midpoint (DDPoint & *r_midpoint*) `[pure virtual]`

Calculates the midpoint of the DDEdge.

Parameters:

r_midpoint The variable used to store the cooridnates of the midpoint.

Implemented in **DDArc** (p. 73), and **DDLine** (p. 67).

3.3.3.4 virtual void DDEdge::calculate_point (double *t*, DDPoint & *r_point*) `[pure virtual]`

Calculates the point that is the parameteric distance *r_t* along the edge.

Parameters:

t The parameteric parameter between 0 and 1 along the DDEdge.

r_point The variable used to store coordinates of the point at the parameteric parameter *t* along the DDEdge.

Implemented in **DDArc** (p. 74), and **DDLine** (p. 67).

3.3.3.5 virtual void DDEdge::create_connected_copy_of_edge (DDPoint ∗ *p_start_point*, DDPoint ∗ *p_end_point*, DDEdge ∗& *rp_copy*) `[pure virtual]`

Creates a connected copy of the DDEdge.

Parameters:

p_start_point A pointer to the **DDPoint**(p. 49) that is to be used as the start point of the new DDEdge. If **DDPoint**(p. 49) is specified then the provided **DDPoint**(p. 49) is used to create the DDEdge. If the pointer is null then a new **DDPoint**(p. 49) is created with the same coordinates as the corresponding end point of this instance.

p_end_point A pointer to the **DDPoint**(p. 49) that is to be used as the end point of the new DDEdge. If **DDPoint**(p. 49) is specified then the provided **DDPoint**(p. 49) is used to create the DDEdge. If the pointer is null then a new **DDPoint**(p. 49) is created with the same coordinates as the corresponding end point of this instance.

rp_copy The variable used to store the new DDEdge.

Implemented in **DDArc** (p. 75), and **DDLine** (p. 67).

3.3.3.6 virtual void DDEdge::create_integrated_copy_of_edge (DDEdge ∗& *rp_copy*) `[pure virtual]`

Creates a new DDEdge which uses the same instances of the **DDPoint**(p. 49) that are used as end points as this instance.

Parameters:

rp_copy The variable used to store the new DDEdge.

Implemented in **DDArc** (p. 75), and **DDLine** (p. 68).

3.3.3.7 virtual void DDEdge::create_isolated_copy_of_edge (DDEdge *& *rp_copy*) [pure virtual]

Creates a new DDEdge by creating new instances of the **DDPoint**(p. 49) to use as end points that are at the same coordinates as the DDPoints of this instance.

Parameters:

rp_copy The variable used to store the new DDEdge.

Implemented in **DDArc** (p. 75), and **DDLine** (p. 68).

3.3.3.8 void DDEdge::delete_all_instances () [static]

This function deletes all instance of the class.

This function should only be used to clean up memory at the end of a program. Higher order objects should be deleted first to make sure that this instance is not being used by another class.

3.3.3.9 DD_RESULT DDEdge::delete_edge (DDEdge *& *rp_edge*) [static]

Deletes the DDEdge if it is not being used by any DDEdgeUses. This method also deletes the primitive that make up the DDEdge if they are not used by any other primitives. Calls the delete function of each primitive that makes up the DDEdge.

3.3.3.10 DD_RESULT DDEdge::get_edge (unsigned long & *r_id*, DDEdge *& *rp_edge*) [static]

Get the DDEdge with the id *r_id*.

Parameters:

r_id The id of the DDEdge to get.

rp_edge The variable where the DDEdge is returned.

Return values:

DD_SUCCESS If the DDEdge was found.

DD_ERROR_PRIMITIVE_NOT_FOUND If the DDEdge was not found.

3.3.3.11 virtual void DDEdge::get_edge_type (DDEdgeType & *r_edge_type*) `[pure virtual]`

A pure virtual function that returns the DDEdge type of the instance.

Each different edge type that is derived from this base class must overload this method.

Parameters:

r_edge_type Used to return the DDEdge type.

Implemented in **DDArc** (p. 76), and **DDLine** (p. 68).

3.3.3.12 void DDEdge::get_edge_uses (DDEdgeUse *& *rp_forward_edge_use*, DDEdgeUse *& *rp_backward_edge_use*)

Retrieves the DDEdgeUses that use the DDEdge.

NULL is a valid pointer that can be returned in the storage variables. It means that the DDEdge is not being used in that sense.

Parameters:

rp_forward_edge_use Variable used to return the **DDEdgeUse**(p. 80) that uses the DDEdge in the forward sense.

rp_backward_edge_use Variable used to return the **DDEdgeUse**(p. 80) that uses the DDEdge in the backward sense.

3.3.3.13 void DDEdge::get_end_point_ids (unsigned long & *r_start*, unsigned long & *r_end*)

Gets the ids of the DDPoints used as end points.

Parameters:

r_start The variable where the id of the **DDPoint**(p. 49) used as the DDEdge's start point is stored.

r_end The variable where the id of the **DDPoint**(p. 49) used as the DDEdge's end point is stored.

3.3.3.14 void DDEdge::get_end_points (DDPoint *& *rp_start_point*, DDPoint *& *rp_end_point*)

Retrieves pointers to the DDPoints that are used as end points for the DDEdge.

Parameters:

rp_start_point The variable used to return the instance of **DDPoint**(p. 49) used as the start point of the DDEdge.

rp_end_point The variable used to return the instance of **DDPoint**(p. 49) used as the end point of the DDEdge.

3.3.3.15 unsigned long DDEdge::get_ID ()

Retrieves the ID of the loop.

3.3.3.16 void DDEdge::get_ID (unsigned long & *r_id*)

Retrieves the ID of the loop.

3.3.3.17 unsigned long DDEdge::get_ID_generator () `[static]`

This function returns the ID of the last instance created; A new instance that is created and assigned an id manually should be assigned an id that is greater that the returned value.

3.3.3.18 void DDEdge::get_instance_map (DDHashMap< unsigned long, DDEdge * > & *r_edge_map*) `[static]`

Get a map containing pointers to all of the instances of the class indexed by instance id.

Parameters:

r_edge_map The map where the pointers are to be stored.

3.3.3.19 unsigned long DDEdge::get_instance_map_size ()

Returns the number of instances of the class. See the description of the constructor for more information on which instances are saved in the instance map.

3.3.3.20 virtual DD_RESULT DDEdge::is_equal (DDEdge & *r_edge*, double *tolerance*) `[pure virtual]`

The purpose of this routine is to determine if r_edge is equal,within tolerance, to the instance that owns the method.

The conditions for equality are determined by each derived type.

Implemented in **DDArc** (p. 76), and **DDLine** (p. 68).

3.3.3.21 virtual DD_RESULT DDEdge::offset_edge (double *distance*) `[pure virtual]`

Offsets the DDEdge by *distance*.

This routine is defined by each derived class.

Parameters:

distance The distance to offset the DDEdge.

Implemented in **DDArc** (p. 77), and **DDLine** (p. 69).

3.3.3.22 DD_RESULT DDEdge::register_edge_use (DDEdgeUse * *p_edge_use*)

Registers the **DDEdgeUse**(p. 80) that is using the DDEdge. The DDEdge does not notify the **DDEdgeUse**(p. 80) that it was registered. Because of the 2d nature of the domain, each DDEdge may only be used twice. Once for the forward sense and once for the backward sense.

Parameters:

p_edge_use The **DDEdgeUse**(p. 80) to register.

Return values:

DD_SUCCESS If the **DDEdgeUse**(p. 80) was registered.

DD_FAILURE If the DDEdge is already being used by another **DDEdgeUse**(p. 80) in the same sense as p_edge_use.

3.3.3.23 virtual DD_RESULT DDEdge::replace_point (DDPoint * *p_old_point*, DDPoint * *p_new_point*) `[virtual]`

Replaces one **DDPoint**(p. 49) used by the DDEdge for another **DDPoint**(p. 49).

This instance is unregistered with the original **DDPoint**(p. 49). This instance is registered with the new **DDPoint**(p. 49).

Parameters:

p_old_point The original **DDPoint**(p. 49) to be replaced.

p_new_point The **DDPoint**(p. 49) to replace the original **DDPoint**(p. 49).

Return values:

DD_X_NULL_POINTER If either **DDPoint**(p. 49) pointer supplied is NULL.

DD_ERROR_PRIMITIVE_NOT_FOUND If the original **DDPoint**(p. 49) is not being used by the **DDArc**(p. 70).

DD_SUCCESS If the routine was successful.

Reimplemented in **DDArc** (p. 78).

3.3.3.24 virtual DD_RESULT DDEdge::replace_point_and_update (DDPoint * *p_old_point*, DDPoint * *p_new_point*) `[virtual]`

Replaces one **DDPoint**(p. 49) used by the DDEdge for another **DDPoint**(p. 49) and then updates.

This instance is unregistered with the original **DDPoint**(p. 49). This instance is registered with the new **DDPoint**(p. 49). This function is also intended to allow each DDEdge to perform any updates that are needed as a result of replacing a **DDPoint**(p. 49).

Parameters:

p_old_point The **DDPoint**(p. 49) to be replaced.

p_new_point The **DDPoint**(p. 49) to replace the original **DDPoint**(p. 49).

Return values:

DD_X_NULL_POINTER If either **DDPoint**(p. 49) pointer is NULL.

DD_ERROR_PRIMITIVE_NOT_FOUND IF the original **DDPoint**(p. 49) is not being used by the DDEdge.

DD_SUCCESS If the routine was successful.

Reimplemented in **DDArc** (p. 78).

3.3.3.25 DD_RESULT DDEdge::unregister_edge_use (DDEdgeUse * *p_edge_use*)

Unregister a **DDEdgeUse**(p. 80) that no longer uses the DDEdge. The DDEdge does not notify the **DDEdgeUse**(p. 80) that it was unregistered.

Parameters:

p_edge_use The **DDEdgeUse**(p. 80) to unregister.

Return values:

DD_SUCCESS If the **DDEdgeUse**(p. 80) was unregistered.

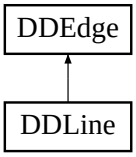
DD_ERROR_PRIMITIVE_NOT_FOUND If the **DDEdgeUse**(p. 80) is not registered with this instance of DDEdge.

3.4 DDLine Class Reference

The DDLine represents a line in 2d space.

`#include <DDLine.hpp>`

Inheritance diagram for DDLine::



Public Member Functions

- **DDLine** (**DDPoint** ∗p_start_point, **DDPoint** ∗p_end_point, unsigned long id=0xffffffff)
- ∼**DDLine** ()
- virtual void **get_edge_type** (DDEdgeType &r_edge_type)
- void **create_isolated_copy_of_edge** (**DDEdge** ∗&rp_copy)
- virtual void **create_connected_copy_of_edge** (**DDPoint** ∗p_start_point, **DDPoint** ∗p_end_point, **DDEdge** ∗&rp_copy)
- void **create_integrated_copy_of_edge** (**DDEdge** ∗&rp_copy)
- void **calculate_midpoint** (**DDPoint** &r_midpoint)
- void **calculate_point** (double t, **DDPoint** &r_point)
- DD_RESULT **offset_edge** (double distance)
- DD_RESULT **is_equal** (**DDEdge** &r_edge, double tolerance)
- double **calculate_length** ()
- void **calculate_bounding_box** (**DDBoundingBox** &r_bounding_box)

3.4.1 Detailed Description

The DDLine represents a line in 2d space.

3.4.2 Constructor & Destructor Documentation

3.4.2.1 DDLine::DDLine (DDPoint ∗ *p_start_point*, DDPoint ∗ *p_end_point*, unsigned long *id* = 0xffffffff)

Constructor.

Creates an line in the forward sense, this means that the line extends from *p_start_point* to *p_end_point*. If either **DDPoint**(p. 49) pointer is NULL there will be an assertion failure. This instance is registered with each **DDPoint**(p. 49) by calling the DDPoint's register routine.

Parameters:

p_start_point The **DDPoint**(p. 49) that is used as the start point of the DDLine.

p_end_point The **DDPoint**(p. 49) that is used as the end point of the DDLine.

id If *id* is 0 then the DDLine will not be added to the static list of all DDEdges and will not be given a unique id. Instead the instance will have an id of 0. If *id* is 0xffffffff then the instance will be given a unique ID based upon the order of creation of the instance. The DDLine will also be added to the list of all DDEdges. If *id* is between 0 and 0xffffffff then the instance will be assigned the ID *id*. However, the IDs must be assigned in assending order or the constructor will assert.

3.4.2.2 DDLine::~DDLine () `[inline]`

Destructor.

If the **DDEdge**(p. 55) is part of any **DDEdgeUse**(p. 80) then the function will assert. This instance is unregistered with each **DDPoint**(p. 49) by calling the DDPoint's unregister routine. The **DDEdge**(p. 55) is removed from the static list of all edge instances.

3.4.3 Member Function Documentation

3.4.3.1 void DDLine::calculate_bounding_box (DDBoundingBox & *r_bounding_box*) `[virtual]`

Calculates the bounding box of the line.

Parameters:

r_bounding_box The variable used to return the calculated bounding box.

Implements **DDEdge** (p. 57).

3.4.3.2 double DDLine::calculate_length () `[virtual]`

Calculates the length of the line.

Implements **DDEdge** (p. 62).

3.4.3.3 void DDLine::calculate_midpoint (DDPoint & *r_midpoint*) [virtual]

Calculates the midpoint of the DDLine.

Parameters:

r_midpoint The variable used to return the coordinates of the midpoint for the DDLine.

Implements **DDEdge** (p. 57).

3.4.3.4 void DDLine::calculate_point (double *t*, DDPoint & *r_point*) [virtual]

Calculates the point that is the parameteric distance *r_t* along the line.

Parameters:

t The parameteric parameter between 0 and 1 along the line.

r_point The variable used to return the coordinates of the point at the parameteric parameter *t* along the line.

Implements **DDEdge** (p. 58).

3.4.3.5 virtual void DDLine::create_connected_copy_of_edge (DDPoint * *p_start_point*, DDPoint * *p_end_point*, DDEdge *& *rp_copy*) [virtual]

Creates a connected copy of the DDLine.

If either **DDPoint**(p. 49) is specified then the provided **DDPoint**(p. 49) is used to create the new DDLine. However, if a specified **DDPoint**(p. 49) is null then a new **DDPoint**(p. 49) is created with the same coordinates as the corresponding **DDPoint**(p. 49) of this instance.

Parameters:

p_start_point A pointer to the **DDPoint**(p. 49) that is to be the start point of the new DDLine. If this pointer is NULL then a new **DDPoint**(p. 49) is created.

p_end_point A pointer to the **DDPoint**(p. 49) that is to be the end point of the new DDLine. If this pointer is NULL then a new **DDPoint**(p. 49) is created.

rp_copy Used to return the new DDLine.

Implements **DDEdge** (p. 58).

3.4.3.6 void DDLine::create_integrated_copy_of_edge (DDEdge *& *rp_copy*) [virtual]

This routine creates a new DDLine which uses the same instances of the DDPoints as this instance.

Parameters:

rp_copy The variable used to return the new DDLine.

Implements **DDEdge** (p. 58).

3.4.3.7 void DDLine::create_isolated_copy_of_edge (DDEdge *& *rp_copy*) [virtual]

Creates a new DDLine by creating new instances of the **DDPoint**(p. 49) that are at the same coordinates as the DDPoints of this instance.

Parameters:

rp_copy The variable used to store the new DDLine.

Implements **DDEdge** (p. 59).

3.4.3.8 virtual void DDLine::get_edge_type (DDEdgeType & *r_edge_type*) [virtual]

A virtual function of the base class that must be implemented for each edge type.

Parameters:

r_edge_type The variable used to return the edge type DDLineType.

Implements **DDEdge** (p. 60).

3.4.3.9 DD_RESULT DDLine::is_equal (DDEdge & *r_edge*, double *tolerance*) [virtual]

Determines if *r_edge* is equal to this instance.

The purpose of this routine is to determine if r_edge is equal to this instance. Equality is checked for by checking to see if the DDPoints used by both DDEdges are the same instance. If the DDPoints used are the same instance then the maximum distance between each **DDEdge**(p. 55) is compared. Thus a DDLine and **DDArc**(p. 70) may be considered equal if they share the same instances of **DDPoint**(p. 49) and the maximum distance between each edge is less than or equal to tolerance.

Parameters:

r_edge The **DDEdge**(p. 55) to compare to this instance.

tolerance The tolerance used to check for equality.

Return values:

DD_FAILURE If the two DDEdges are not equal.

DD_PRI_EQUAL If the two DDEdges share the same instance of **DDPoint**(p. 49) for the start point and the same instance of **DDPoint**(p. 49) for the end point and the maximum distance is less than or equal to tolerance.

DD_PRI_EQUAL_OPPOSITE If the two DDEdges share the opposite instances of **DDPoint**(p. 49) for the start point and end point and the maximum distance is less than or equal to tolerance.

Implements **DDEdge** (p. 61).

3.4.3.10 DD_RESULT DDLine::offset_edge (double *distance*) `[virtual]`

Offsets the line by *distance*.

The DDPoints that are used as the start and end points of the line are offset the distance *distance* along a vector perpendicular to the line.

Parameters:

distance The distance to offset each end point.

Return values:

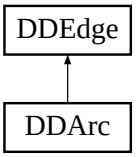
DD_SUCCESS If the line was was offset.

3.5 DDArc Class Reference

The DDArc represents an arc in 2d space.

`#include <DDArc.hpp>`

Inheritance diagram for DDArc::



Public Member Functions

- **DDArc** (**DDPoint** ∗p_start_point, **DDPoint** ∗p_end_point, **DDPoint** ∗p_center_point, unsigned long id=0xffffffff)
- ∼**DDArc** ()
- virtual void **get_edge_type** (DDEdgeType &r_edge_type)
- void **get_center_point** (**DDPoint** ∗&rp_center_point)
- void **calculate_radius** (double &r_radius)
- double **calculate_radius** ()
- double **calculate_chordal_distance** ()
- void **calculate_start_stop_angle** (double &r_start_angle, double &r_stop_angle)
- void **calculate_angle_relative_to_start_angle** (**DDPoint** &r_point, double &r_angle)
- double **calculate_sweep_angle** ()
- void **calculate_parameteric_parameter** (**DDPoint** &r_point, double &r_t)
- void **calculate_point** (double t, **DDPoint** &r_point)
- DD_RESULT **replace_point** (**DDPoint** ∗p_old_point, **DDPoint** ∗p_new_point)
- DD_RESULT **replace_point_and_update** (**DDPoint** ∗p_old_point, **DDPoint** ∗p_new_point)
- void **create_isolated_copy_of_edge** (**DDEdge** ∗&rp_copy)
- virtual void **create_connected_copy_of_edge** (**DDPoint** ∗p_start_point, **DDPoint** ∗p_end_point, **DDEdge** ∗&rp_copy)
- void **create_integrated_copy_of_edge** (**DDEdge** ∗&rp_copy)
- void **calculate_midpoint** (**DDPoint** &r_midpoint)
- DD_RESULT **is_equal** (**DDEdge** &r_edge, double tolerance)
- void **position_center_point_based_upon_sweep_angle** (double sweep_angle)
- double **calculate_area** ()
- DD_RESULT **offset_edge** (double distance)
- double **calculate_length** ()
- void **calculate_bounding_box** (**DDBoundingBox** &r_bounding_box)

Protected Attributes

- **DDPoint ∗ mpCenterPoint**

3.5.1 Detailed Description

The DDArc represents an arc in 2d space.

Author:
Corey McBride

Date:
02/07/03

- The following rules need to be observed when using DDArcs.

1. A DDArc defines arcs from the start point to the end point in the counter clockwise direction.
2. A DDArc cannot have the same instance of **DDPoint**(p. 49) for the start point and end point.
3. Each instance of DDArc must have a unique instance of **DDPoint**(p. 49) for the center point. Or in other words, no two DDArcs can share the same instance of **DDPoint**(p. 49) for the center point.
4. Each **DDLoop**(p. 89) needs to be a closed loop. This means that the first and last instance of **DDPoint**(p. 49) in the loop must be the same instance.

Because of items 2 and 4 it is not possible to have one DDArc define a complete circle. It is necessary to have at least two DDArcs to define a circle.

3.5.2 Constructor & Destructor Documentation

3.5.2.1 DDArc::DDArc (DDPoint ∗ *p_start_point*, DDPoint ∗ *p_end_point*, DDPoint ∗ *p_center_point*, unsigned long *id* = `0xffffffff`)

Constructor.

Creates a DDArc in the forward sense, this means that the DDArc extends from the **DDPoint**(p. 49) *p_start_point* to the **DDPoint**(p. 49) *p_end_point* it the counter clockwise direction. If either end point pointer is NULL there will be an assertion failure. This instance is registered with each **DDPoint**(p. 49) by calling the DDPoint's register routine.

Parameters:

p_start_point The **DDPoint**(p. 49) that used as the start point for the DDArc.

p_end_point The **DDPoint**(p. 49) that is used as the end point for the DDArc.

p_center_point The **DDPoint**(p. 49) that is used as the center point of the DDArc.

id If *id* is 0 then the DDArc will not be added to the static list of all DDEdges and will not be given a unique id. Instead the instance will have an id of 0. If *id* is 0xffffffff then the instance will be given a unique ID based upon the order of creation of the instance. The DDArc will also be added to the list of all DDEdges. If *id* is between 0 and 0xffffffff then the instance will be assigned the ID *id*. However, the IDs must be assigned in assending order or the constructor will assert.

3.5.2.2 DDArc::∼DDArc ()

Destructor.

If the **DDEdge**(p. 55) is used by any **DDEdgeUse**(p. 80) then the function will assert. The edge lists of the used DDPoints are updated to remove this instance. The **DDEdge**(p. 55) is removed from the static list of all **DDEdge**(p. 55) instances.

3.5.3 Member Function Documentation

3.5.3.1 void DDArc::calculate_angle_relative_to_start_angle (DDPoint & *r_point*, double & *r_angle*)

Calculates the angle that *r_point* makes in relation to the start **DDPoint**(p. 49).

This routine does not check to make sure that *r_point* is on the DDArc's parent circle. If the **DDPoint**(p. 49) is not on the parent circle then the calculations will be incorrect.

Parameters:

r_point The **DDPoint**(p. 49) on the parent circle to calculate the angle for.

r_angle The variable used to return the calculated angle for the **DDPoint**(p. 49) /e r_point.

3.5.3.2 double DDArc::calculate_area ()

Calculates the area that is bounded by the DDArc and a line that extends between the end points of the DDArc.

3.5.3.3 void DDArc::calculate_bounding_box (DDBoundingBox & *r_bounding_box*) `[virtual]`

Calculates the bounding box of the arc.

Parameters:

r_bounding_box The variable used to return the calculated bounding box.

Implements **DDEdge** (p. 57).

3.5.3.4 double DDArc::calculate_chordal_distance ()

Calculates the cordal distance of the DDArc.

Returns:

The cordal distance of the DDArc.

3.5.3.5 double DDArc::calculate_length () `[virtual]`

Calculates the length of the arc.

Implements **DDEdge** (p. 57).

3.5.3.6 void DDArc::calculate_midpoint (DDPoint & *r_midpoint*) `[virtual]`

Calculates the midpoint of the DDArc.

Parameters:

r_midpoint The variable used to return the coordinates of the midpoint for the DDArc.

Implements **DDEdge** (p. 57).

3.5.3.7 void DDArc::calculate_parameteric_parameter (DDPoint & *r_point*, double & *r_t*)

Calculates the parameteric parameter along the DDArc for *r_point*. The routine does not check to make sure that the **DDPoint**(p. 49) is on the DDArc's parent circle. If the **DDPoint**(p. 49) is not on the parent circle then the calculations will be incorrect. The parametric parameter is based on values of 0.0 for the start point and 1.0 for the end point.

Parameters:

r_point The **DDPoint**(p. 49) with coordinates that place it on the DDArc's parent circle.

r_t The variable used to return the parameteric parameter calculated for *r_point* on the DDArc.

3.5.3.8 void DDArc::calculate_point (double *t*, DDPoint & *r_point*) `[virtual]`

Calculates the coordinates for the point that is parameteric distance *t* along the DDArc.

Parameters:

t The parameteric parameter between 0.0 and 1.0 along the DDArc.

r_point The variable used to return the coordinates of the point at the parameteric parameter *t* along the DDArc.

Implements **DDEdge** (p. 58).

3.5.3.9 double DDArc::calculate_radius ()

Calculates the radius of the DDArc.

Returns:

The radius of the DDArc.

3.5.3.10 void DDArc::calculate_radius (double & *r_radius*)

Calculates the radius of the DDArc.

Parameters:

r_radius The variable used to return the radius of the DDArc.

3.5.3.11 void DDArc::calculate_start_stop_angle (double & *r_start_angle*, double & *r_stop_angle*)

Calculates the start and stop angles of the DDArc relative to the positive X axis.

Parameters:

r_start_angle The variable used to return the calculated angle of the start **DDPoint**(p. 49) relative to the positive X axis.

r_stop_angle The variable used to return the calculated angle of the end **DDPoint**(p. 49) relative to the positive X axis.

3.5.3.12 double DDArc::calculate_sweep_angle ()

Calculates the sweep angle of the DDArc.

Returns:
The calculated sweep angle of the DDArc.

3.5.3.13 virtual void DDArc::create_connected_copy_of_edge (DDPoint * *p_start_point*, DDPoint * *p_end_point*, DDEdge *& *rp_copy*) [virtual]

This routine creates a new connected copy of the DDArc.

If either **DDPoint**(p. 49) is specified then the provided **DDPoint**(p. 49) is used to create the new DDArc. However, if a specified **DDPoint**(p. 49) is null then a new **DDPoint**(p. 49) is created with the same coordinates as the corresponding **DDPoint**(p. 49) of this instance. A new center **DDPoint**(p. 49) is always created with the same coordinates as this instance's center **DDPoint**(p. 49).

Parameters:

p_start_point A pointer to the **DDPoint**(p. 49) that is to be the start point of the new DDArc. If this pointer is NULL then a new **DDPoint**(p. 49) is created.

p_end_point A pointer to the **DDPoint**(p. 49) that is to be the end point of the new DDArc. If this pointer is NULL then a new **DDPoint**(p. 49) is created.

rp_copy Used to return the new DDArc.

Implements **DDEdge** (p. 58).

3.5.3.14 void DDArc::create_integrated_copy_of_edge (DDEdge *& *rp_copy*) [virtual]

This routine creates a new DDArc which uses the same instances of the DDPoints as this instance.

Parameters:

rp_copy The variable used to return the new DDArc.

Implements **DDEdge** (p. 58).

3.5.3.15 void DDArc::create_isolated_copy_of_edge (DDEdge *& *rp_copy*) [virtual]

Creates a new DDArc by creating new instances of **DDPoint**(p. 49) that are at the same coordinates as the DDPoints of this instance.

Parameters:

rp_copy The variable used to store the new DDArc.

Implements **DDEdge** (p. 59).

3.5.3.16 void DDArc::get_center_point (DDPoint *& *rp_center_point*)

Retrieves the **DDPoint**(p. 49) that is used as the center point of the DDArc.

Parameters:

rp_center_point The pointer used to return the **DDPoint**(p. 49) used as the center point.

3.5.3.17 virtual void DDArc::get_edge_type (DDEdgeType & *r_edge_type*) `[virtual]`

A virtual function of the base class **DDEdge**(p. 55) that must be implemented for each edge type.

Parameters:

r_edge_type The variable used to return the edge type DDArcType.

Implements **DDEdge** (p. 60).

3.5.3.18 DD_RESULT DDArc::is_equal (DDEdge & *r_edge*, double *tolerance*) `[virtual]`

Determines if *r_edge* is equal to this instance.

The purpose of this routine is to determine if r_edge is equal to this instance. Equality is checked for by checking to see if the DDPoints used by both DDEdges are the same instance. If the DDPoints used are the same instance then the maximum distance between each **DDEdge**(p. 55) is compared. Thus a **DDLine**(p. 65) and DDArc may be considered equal if they share the same instances of **DDPoint**(p. 49) and the maximum distance between each edge is less than or equal to tolerance.

Parameters:

r_edge The **DDEdge**(p. 55) to compare to this instance.

tolerance The tolerance used to check for equality.

Return values:

DD_FAILURE If the two DDEdges are not equal.

DD_PRI_EQUAL If the two DDEdges share the same instance of **DDPoint**(p. 49) for the start point and the same instance of **DDPoint**(p. 49) for the end point and the maximum distance is less than or equal to tolerance.

DD_PRI_EQUAL_OPPOSITE If the two DDEdges share the opposite instances of **DDPoint**(p. 49) for the start point and end point and the maximum distance is less than or equal to tolerance.

Implements **DDEdge** (p. 62).

3.5.3.19 DD_RESULT DDArc::offset_edge (double *distance*) `[virtual]`

Offsets the DDArc by *distance*.

The start and end DDPoints of the DDArc are offset the distance *distance* along a vector from the center **DDPoint**(p. 49) to the each end **DDPoint**(p. 49). The center **DDPoint**(p. 49) is not moved.

Parameters:

distance The distance to offset each end **DDPoint**(p. 49).

Return values:

DD_SUCCESS If the DDArc was was offset.

DD_FAILURE If the DDArc was inverted by the offset. This happens when *distance* is >= the radius.

Implements **DDEdge** (p. 62).

3.5.3.20 void DDArc::position_center_point_based_upon_sweep_angle (double *sweep_angle*)

Adjusts the coordinates of the center point so that the arc will have *sweep_angle* radians between the end points.

If the **DDPoint**(p. 49) instance that is used for the center point is shared by multiple edges or arcs then problems may arise when the coordinates of the instance are changed. To avoid this it is recommended that each DDArc have a unique **DDPoint**(p. 49) instance.

Parameters:

sweep_angle The angel to enforce between each end point.

3.5.3.21 DD_RESULT DDArc::replace_point (DDPoint * *p_old_point*, DDPoint * *p_new_point*) `[virtual]`

Replaces one **DDPoint**(p. 49) used by the DDArc for another **DDPoint**(p. 49).

This instance is unregistered with the original **DDPoint**(p. 49). This instance is registered with the new **DDPoint**(p. 49).

Parameters:

p_old_point The **DDPoint**(p. 49) to be replaced.

p_new_point The **DDPoint**(p. 49) to replace the original **DDPoint**(p. 49).

Return values:

DD_X_NULL_POINTER If either **DDPoint**(p. 49) pointer supplied is NULL.

DD_ERROR_PRIMITIVE_NOT_FOUND IF the original **DDPoint**(p. 49) is not being used by the DDArc.

DD_SUCCESS If the routine was successful.

Reimplemented from **DDEdge** (p. 62).

3.5.3.22 DD_RESULT DDArc::replace_point_and_update (DDPoint * *p_old_point*, DDPoint * *p_new_point*) `[virtual]`

Replaces one **DDPoint**(p. 49) used by the DDArc for another **DDPoint**(p. 49) and then updates.

This instance is unregistered with the original **DDPoint**(p. 49). This instance is registered with the new **DDPoint**(p. 49). This function is also intended to allow each **DDEdge**(p. 55) to perform any updates that are needed as a result of replacing a **DDPoint**(p. 49).

Parameters:

p_old_point The **DDPoint**(p. 49) to be replaced.

p_new_point The **DDPoint**(p. 49) to replace the original **DDPoint**(p. 49).

Return values:

DD_X_NULL_POINTER If either **DDPoint**(p. 49) pointer is NULL.

DD_ERROR_PRIMITIVE_NOT_FOUND IF the original **DDPoint**(p. 49) is not being used by the **DDEdge**(p. 55).

DD_SUCCESS If the routine was successful.

Reimplemented from **DDEdge** (p. 63).

3.5.4 Member Data Documentation

3.5.4.1 DDPoint* DDArc::mpCenterPoint [protected]

A macro to define memory allocation methods.

3.6 DDEdgeUse Class Reference

The DDEdgeUse represents an edge and the sense that it is used.

`#include <DDEdgeUse.hpp>`

Public Member Functions

- **DDEdgeUse** (**DDEdge** ∗p_edge, DDEdgeUseSense use_sense, unsigned long id=0xffffffff)
- ∼**DDEdgeUse** ()
- void **get_ID** (unsigned long &r_id)
- unsigned long **get_ID** ()
- void **get_end_points** (**DDPoint** ∗&rp_start_point, **DDPoint** ∗&rp_end_point, DDEdgeUseSense &r_use_sense)
- DDEdgeUseSense **get_edge_use_sense** ()
- void **create_isolated_copy_of_edge_use** (**DDEdgeUse** ∗&rp_copy)
- void **create_connected_copy_of_edge_use** (**DDPoint** ∗p_start_point, **DDPoint** ∗p_end_point, **DDEdgeUse** ∗&rp_copy)
- void **create_integrated_copy_of_edge_use** (**DDEdgeUse** ∗&rp_copy)
- void **calculate_midpoint** (**DDPoint** &r_midpoint)
- void **calculate_bounding_box** (**DDBoundingBox** &r_bounding_box)
- void **calculate_point** (double distance, **DDPoint** &r_point)
- void **get_edge_type** (DDEdgeType &r_type)
- DD_RESULT **replace_edge** (**DDEdge** ∗p_old_edge, **DDEdge** ∗p_new_edge, DDEdgeUseSense new_use_sense)
- DD_RESULT **get_edge** (**DDEdge** ∗&rp_edge, DDEdgeUseSense &r_use_sense)
- void **get_end_point_ids** (unsigned long &r_start, unsigned long &r_end)
- DD_RESULT **register_loop** (**DDLoop** ∗p_loop, DDLoop::iterator iterator)
- DD_RESULT **unregister_loop** (**DDLoop** ∗p_loop)
- DD_RESULT **get_loop** (**DDLoop** ∗&rp_loop)
- DDLoop::iterator **get_loop_iterator** ()
- DD_RESULT **offset_edgeuse** (double distance)
- double **calculate_length** ()

Static Public Member Functions

- DD_RESULT **get_edge_use** (unsigned long &r_id, **DDEdgeUse** ∗&rp_edge_use)
- DD_RESULT **get_instance_map** (**DDHashMap**< unsigned long, **DDEdgeUse** ∗ > &r_edge_use_map)

- unsigned long **get_instance_map_size** ()
- void **delete_all_instances** ()
- unsigned long **get_ID_generator** ()
- DD_RESULT **delete_edgeuse** (**DDEdgeUse** ∗&rp_edgeuse)

3.6.1 Detailed Description

The DDEdgeUse represents an edge and the sense that it is used.

Author:
Corey McBride

Date:
02/11/03

3.6.2 Constructor & Destructor Documentation

3.6.2.1 DDEdgeUse::DDEdgeUse (DDEdge ∗ *p_edge*, DDEdgeUseSense *use_sense*, unsigned long *id* = `0xffffffff`)

A macro to define memory allocation methods. Constructor.

Parameters:

p_edge The **DDEdge**(p. 55) to be used by this instance.

use_sense The sense that *p_edge* is used. If *p_edge* is NULL then there will be an assertion failure. If *p_edge* is already being used by another DDEdgeUse with the same sense then the function will assert. This is determined by the DDEdgeUse list maintained by *p_edge*.

id If *id* is 0xffffffff then the instance will be given a unique ID based upon the order of creation of the instance. The DDEdgeUse will also be added to the static list of all DDEdgeUse. If *id* is 0 then the DDEdgeUse will not be added to the static list of all DDEdgeUse and will not be given a unique id. Instead the instance will have an id of 0. If *id* is between 0 and 0xffffffff then the instance will be assigned the ID *id*. However, the IDs must be assigned in assending order or the constructor will assert. This instance is registered with the **DDEdge**(p. 55) by calling the DDEdge's register routine. Thus the link from **DDEdge**(p. 55) to DDEdgeUse is maintained automatically. The **DDLoop**(p. 89) pointer for the **DDLoop**(p. 89) using this instance is set to NULL.

3.6.2.2 DDEdgeUse::~DDEdgeUse ()

Destructor.

If the DDEdgeUse is being used by any **DDLoop**(p. 89) then this function will assert. This instance is unregistered with the **DDEdge**(p. 55) by calling the DDEdge's unregister routine. Thus the link from **DDEdge**(p. 55) to DDEdgeUse is maintaine automatically. The DDEdgeUse is removed from the static list of all DDEdgeUse instances.

3.6.3 Member Function Documentation

3.6.3.1 void DDEdgeUse::calculate_bounding_box (DDBoundingBox & *r_bounding_box*)

Calcualtes the bounding box of the **DDEdge**(p. 55) used by this DDEdgeUse.

Parameters:

r_bounding_box The **DDBoundingBox**(p. 118) used to store the calculated bounding box of the **DDEdge**(p. 55).

3.6.3.2 double DDEdgeUse::calculate_length ()

Calculates the length of the **DDEdge**(p. 55). The calculate_length method is called for the **DDEdge**(p. 55) used by the DDEdgeUse.

3.6.3.3 void DDEdgeUse::calculate_midpoint (DDPoint & *r_midpoint*)

Calculates the midpoint of the **DDEdge**(p. 55).

Parameters:

r_midpoint The coordinates of the midpoint are returned in this variable.

3.6.3.4 void DDEdgeUse::calculate_point (double *distance*, DDPoint & *r_point*)

Calculates the point that is the distance *distance* along the **DDEdge**(p. 55). The sense of the DDEdgeUse is considered when calculating the distance.

Parameters:

distance The distance along the **DDEdge**(p. 55) to calculate the coordinates.

r_point The **DDPoint**(p. 49) used to store the coordinates of the point calculated at *distance* along the **DDEdge**(p. 55).

3.6.3.5 void DDEdgeUse::create_connected_copy_of_edge_use (DDPoint * *p_start_point*, DDPoint * *p_end_point*, DDEdgeUse *& *rp_copy*)

Creates a copy of the DDEdgeUse.

Parameters:

p_start_point A pointer to the **DDPoint**(p. 49) that is to be used as the start point of the new DDEdgeUse. If **DDPoint**(p. 49) is specified then the provided **DDPoint**(p. 49) is used to create the DDEdgeUse. If the pointer is null then a new **DDPoint**(p. 49) is created with the same coordinates as the corresponding end point of this instance.

p_end_point A pointer to the **DDPoint**(p. 49) that is to be used as the end point of the new DDEdgeUse. If **DDPoint**(p. 49) is specified then the provided **DDPoint**(p. 49) is used to create the DDEdgeUse. If the pointer is null then a new **DDPoint**(p. 49) is created with the same coordinates as the corresponding end point of this instance.

rp_copy The variable used to store the new DDEdgeUse.

3.6.3.6 void DDEdgeUse::create_integrated_copy_of_edge_use (DDEdgeUse *& *rp_copy*)

Creates a copy of the DDEdgeUse.

Creates a copy of the DDEdgeUse that uses the same **DDPoint**(p. 49) instances of the end points as this instance. But has a new instance of the **DDEdge**(p. 55) that is a copy of the one used by this instance.The new DDEdgeUse does not belong to the same **DDLoop**(p. 89) as this instance.

Parameters:

rp_copy The variable used to store the new DDEdgeUse.

3.6.3.7 void DDEdgeUse::create_isolated_copy_of_edge_use (DDEdgeUse *& *rp_copy*)

Creates a copy of the DDEdgeUse.

The new DDEdgeUse has new instances of the DDPoints and **DDEdge**(p. 55) that are copies of the ones used by this instance. The new DDEdgeUse does not belong to the same **DDLoop**(p. 89) as this instance.

Parameters:

rp_copy The variable used to store the new DDEdgeUse.

3.6.3.8 void DDEdgeUse::delete_all_instances () [static]

This function deletes all instance of the class.

This function should only be used to clean up memory at the end of a program. Higher order objects should be deleted first to make sure that this instance is not being used by another class.

3.6.3.9 DD_RESULT DDEdgeUse::delete_edgeuse (DDEdgeUse *& *rp_edgeuse*) [static]

This method deletes a DDEdgeUse if it is not being used by a **DDLoop**(p. 89).

This method also deletes the primitive that make up the DDEdgeUse if they are not used by any other primitives. Calls the delete function of each primitive that makes up the DDEdgeUse.

3.6.3.10 DD_RESULT DDEdgeUse::get_edge (DDEdge *& *rp_edge*, DDEdgeUseSense & *r_use_sense*)

Get the **DDEdge**(p. 55) used by this instance of DDEdgeUse and the and sense of its use.

Parameters:

rp_edge The variable where the **DDEdge**(p. 55) will be stored.

r_use_sense The variable where the sense will be stored.

3.6.3.11 void DDEdgeUse::get_edge_type (DDEdgeType & *r_type*)

Gets the **DDEdge**(p. 55) type.

3.6.3.12 DD_RESULT DDEdgeUse::get_edge_use (unsigned long & *r_id*, DDEdgeUse *& *rp_edge_use*) [static]

Get the DDEdgeUse with the id *r_id*.

Parameters:

r_id The id of the DDEdgeUse to get.

rp_edge_use The variable where the DDEdgeUse is returned.

Return values:

DD_SUCCESS If the DDEdgeUse was found.

DD_ERROR_PRIMITIVE_NOT_FOUND If a DDEdgeUse was not found with that id.

3.6.3.13 DDEdgeUseSense DDEdgeUse::get_edge_use_sense ()

Returns the sense that the **DDEdge**(p. 55) is used.

Return values:

DDForwardSense The **DDEdge**(p. 55) is used in the forward sense.

DDBackwardSense The **DDEdge**(p. 55) is used in the backward sense.

3.6.3.14 void DDEdgeUse::get_end_point_ids (unsigned long & *r_start*, unsigned long & *r_end*)

Gets the ids of the DDPoints that are used as end points by this instance. The use sense is taken into account when retieving the ids.

Parameters:

r_start The variable where the id of the **DDPoint**(p. 49) used as the DDEdgeUse's start point is stored.

r_end The variable where the id of the **DDPoint**(p. 49) used as the DDEdgeUse's end point is stored.

3.6.3.15 void DDEdgeUse::get_end_points (DDPoint *& *rp_start_point*, DDPoint *& *rp_end_point*, DDEdgeUseSense & *r_use_sense*)

Retrieves the DDPoints that are used by the **DDEdge**(p. 55) based upon the use sense.

Parameters:

rp_start_point The variable to store the start point of the **DDEdge**(p. 55).

rp_end_point The variable to store the end point of the **DDEdge**(p. 55).

r_use_sense The sense that the **DDEdge**(p. 55) is used. If the use sense is in the forward sense then the DDPoints are returned in the same order as the **DDEdge**(p. 55) class. If the use sense is backward then the DDPoints are returned in the opposite order as the **DDEdge**(p. 55) class.

3.6.3.16 unsigned long DDEdgeUse::get_ID ()

Returns the ID of the DDEdgeUse.

3.6.3.17 void DDEdgeUse::get_ID (unsigned long & *r_id*)

Retrieves the ID of the DDEdgeUse.

Parameters:

r_id The variable used to return the id.

3.6.3.18 unsigned long DDEdgeUse::get_ID_generator () [static]

This function returns the ID of the last instance created; A new instance that is created and assigned an id manually should be assigned an id that is greater that the returned value.

3.6.3.19 DD_RESULT DDEdgeUse::get_instance_map (DDHashMap< unsigned long, DDEdgeUse * > & *r_edge_use_map*) [static]

Get a map containing pointers to all of the instances of the class indexed by instance id. See the description of the constructor for more information on which instances are saved in the instance map.

Parameters:

r_edge_use_map The map where the pointers are returned.

3.6.3.20 unsigned long DDEdgeUse::get_instance_map_size () [static]

Returns the number of instances of the class. See the description of the constructor for more information on which instances are saved in the instance map.

3.6.3.21 DD_RESULT DDEdgeUse::get_loop (DDLoop *& *rp_loop*)

Get the **DDLoop**(p. 89) that uses this instance of DDEdgeUse.

NULL is a valid value to return in *rp_loop*. It means that this instance of DDEdgeUse is not used by any **DDLoop**(p. 89).

Parameters:

rp_loop The variable used to store the returned **DDLoop**(p. 89).

3.6.3.22 DDLoop::iterator DDEdgeUse::get_loop_iterator ()

Returns the DDLoop::interator that this instance belongs to.

This is only valid if the edgeuse belongs to a loop.

3.6.3.23 DD_RESULT DDEdgeUse::offset_edgeuse (double *distance*)

Offsets the **DDEdge**(p. 55) used by this DDEdgeUse by *distance*. If the use sense is backward then the **DDEdge**(p. 55) is offset by -distance.

The offset_edge function is called for the **DDEdge**(p. 55). The return values are defined by each class derived from **DDEdge**(p. 55).

Parameters:

distance The distance to offset the **DDEdge**(p. 55)

3.6.3.24 DD_RESULT DDEdgeUse::register_loop (DDLoop * *p_loop*, DDLoop::iterator *iterator*)

Register the **DDLoop**(p. 89) that is using this instance of DDEdgeUse.

The DDEdgeUse does not notify the **DDLoop**(p. 89) that it was registered. Because of the 2d nature of the domain, each DDEdgeUse may only be used by one **DDLoop**(p. 89). If the DDEdgeUse is already being used by another **DDLoop**(p. 89) then the new **DDLoop**(p. 89) will not be added.

Parameters:

p_loop The **DDLoop**(p. 89) to register.

iterator The **DDLoop**(p. 89) iterator for this DDEdgeUse instance.

Return values:

DD_SUCCESS The **DDLoop**(p. 89) was registered.

DD_FAILURE The **DDLoop**(p. 89) was not registered.

3.6.3.25 DD_RESULT DDEdgeUse::replace_edge (DDEdge * *p_old_edge*, DDEdge * *p_new_edge*, DDEdgeUseSense *new_use_sense*)

Replaces one **DDEdge**(p. 55) for another.

This instance of DDEdgeUse is unregistered with the original **DDEdge**(p. 55) and registered with the new **DDEdge**(p. 55). Thus the link from **DDEdge**(p. 55) to DDEdgeUse is maintained automatically.

Parameters:

p_old_edge The original **DDEdge**(p. 55).

p_new_edge The **DDEdge**(p. 55) to replace the original **DDEdge**(p. 55).

new_use_sense The new DDEdgeUseSense for the new DDEdgeUse.

Return values:

DD_X_NULL_POINTER If either **DDEdge**(p. 55) pointer is NULL.

DD_ERROR_PRIMITIVE_NOT_FOUND IF the original **DDEdge**(p. 55) is not being used by this instance of DDEdgeUse.

DD_SUCCESS If the replace was successful.

DD_FAILURE If the new **DDEdge**(p. 55) is already being used by another instance of DDEdgeUse

3.6.3.26 DD_RESULT DDEdgeUse::unregister_loop (DDLoop * *p_loop*)

Unregisters a **DDLoop**(p. 89) that no longer uses this instance of DDEdgeUse.

The DDEdgeUse does not notify the **DDLoop**(p. 89) that it was unregistered.

Parameters:

p_loop The **DDLoop**(p. 89) to unregister.

Return values:

DD_SUCCESS If the **DDLoop**(p. 89) was unregistered.

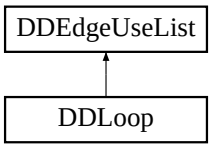
DD_ERROR_PRIMITIVE_NOT_FOUND If the **DDLoop**(p. 89) was not registered with this instance of DDEdgeUse.

3.7 DDLoop Class Reference

The DDLoop represents a closed loop of edges.

`#include <DDLoop.hpp>`

Inheritance diagram for DDLoop::



Public Member Functions

- **DDLoop** (unsigned long id=0xffffffff)
- virtual ∼**DDLoop** ()
- virtual void **get_ID** (unsigned long &r_id)
- virtual unsigned long **get_ID** ()
- virtual void **get_point** (double t, **DDPoint** &r_point)
- virtual double **get_length** ()
- void **calculate_centroid_of_end_points** (**DDPoint** &r_centroid)
- virtual void **create_isolated_copy_of_loop** (**DDLoop** ∗&rp_copy)
- virtual void **create_integrated_copy_of_loop** (**DDLoop** ∗&rp_copy)
- virtual void **calculate_bounding_box** (**DDBoundingBox** &r_bounding_box)
- virtual void **get_neighboring_loops** (std::list< **DDLoop** ∗ > &r_neighboring_loops)
- virtual DD_RESULT **register_surface** (**DDSurface** ∗p_surface, DDLoopType loop_type)
- virtual void **get_loop_type** (DDLoopType &r_loop_type)
- virtual DD_RESULT **unregister_surface** (**DDSurface** ∗p_surface)
- virtual void **get_surface** (**DDSurface** ∗&rp_surface)
- virtual void **get_points** (std::list< **DDPoint** ∗ > &r_point_list)
- virtual void **push_front** (**DDEdgeUse** ∗&rp_edge_use)
- virtual void **push_back** (**DDEdgeUse** ∗&rp_edge_use)
- virtual void **pop_front** ()
- virtual void **pop_back** ()
- virtual DDEdgeUseList::iterator **insert** (DDEdgeUseList::iterator pos, **DDEdgeUse** ∗&rp_-edge_use)
- virtual DDEdgeUseList::iterator **erase** (std::list< **DDEdgeUse** ∗ >::iterator pos)
- virtual void **remove** (**DDEdgeUse** ∗p_edge_use)
- virtual void **clear** ()

Static Public Member Functions

- DD_RESULT **get_loop** (unsigned long &r_id, **DDLoop** ∗&rp_loop)
- DD_RESULT **get_instance_map** (**DDHashMap**< unsigned long, **DDLoop** ∗ > &r_loop_-map)
- void **delete_all_instances** ()
- unsigned long **get_instance_map_size** ()
- DD_RESULT **delete_loop** (**DDLoop** ∗&rp_loop)
- unsigned long **get_ID_generator** ()

3.7.1 Detailed Description

The DDLoop represents a closed loop of edges.

Author:
Corey McBride

Date:
02/07/03

The order of the edges in the loop is important because it shows which edges are next to each other. Edges that are next to each other should share a common point. This class is dereived from **DDEdgeUseList**(p. 122). A sub set of std::list functions have been overloaded to automatically maintain the link from **DDEdgeUse**(p. 80) to DDLoop. These function deal with adding and removing **DDEdgeUse**(p. 80) from the DDLoop. If these overloaded functions are used the link will be maintained automatically. Otherwise it is the developers responsiblity to maintain the link. All of the other functions and iterators for std:list work as designed. This was done so that it would not be necessary to create a complete set of interface functions to the list class.

3.7.2 Constructor & Destructor Documentation

3.7.2.1 DDLoop::DDLoop (unsigned long *id* = `0xffffffff`)

A macro to define memory allocation methods. Constructor.

Parameters:

id If *id* is 0xffffffff then the instance will be given a unique ID based upon the order of creation of the instance. The new DDLoop will also be added to the list of all DDLoops. If *id* is 0 then the DDLoop will not be added to the static list of all DDLoop and will not be given

a unique id. Instead the instance will have an id of 0. If *id* is between 0 and 0xffffffff then the instance will be assigned the ID *id*. However, the IDs must be assigned in assending order or the constructor will assert.

3.7.2.2 virtual DDLoop::~DDLoop () [virtual]

Destructor.

If the DDLoop is used by any **DDSurface**(p. 98) then the function will assert. This instance of DDLoop is unregistered with the DDEdgeUses that are used by the DDLoop. Thus the link between **DDEdgeUse**(p. 80) and DDLoop is maintained automatically. This instance is removed from the static list of all DDLoop instances.

3.7.3 Member Function Documentation

3.7.3.1 virtual void DDLoop::calculate_bounding_box (DDBoundingBox & *r_bounding_box*) [virtual]

Calculates the bounding box of the DDLoop.

Parameters:

r_bounding_box The **DDBoundingBox**(p. 118) used to return the bounding box calculated for the DDLoop.

3.7.3.2 void DDLoop::calculate_centroid_of_end_points (DDPoint & *r_centroid*)

Calculates the centroid of the end points of the edges that make up the loop.

Parameters:

r_centroid The variable used to return the coordinates of the centoid of the DDPoints that make up the DDLoop.

3.7.3.3 virtual void DDLoop::clear () [virtual]

Remove all of the **DDEdgeUse**(p. 80) from the DDLoop.

Unregisters the DDLoop with each **DDEdgeUse**(p. 80).

3.7.3.4 virtual void DDLoop::create_integrated_copy_of_loop (DDLoop *& *rp_copy*) [virtual]

Creates a copy of the loop.

The copy uses the same instances of the **DDPoint**(p. 49) as this instance. But has a new instance of **DDEdge**(p. 55) and **DDEdgeUse**(p. 80) that are a copy of the one used by this instance. The new DDLoop does not belong to the same **DDSurface**(p. 98) as this instance.

3.7.3.5 virtual void DDLoop::create_isolated_copy_of_loop (DDLoop *& *rp_copy*) [virtual]

Creates a copy of the DDLoop that has new instances of the DDPoints. DDEdges, and DDEdge-Uses that are copies of the ones used by this instance.

The new DDLoop does not belong to the same **DDSurface**(p. 98) as this instance.

3.7.3.6 void DDLoop::delete_all_instances () [static]

This function deletes all instance of the class.

This function should only be used to clean up memory at the end of a program. Higher order objects should be deleted first to make sure that this instance is not being used by another class.

3.7.3.7 DD_RESULT DDLoop::delete_loop (DDLoop *& *rp_loop*) [static]

Deletes the instance of DDLoop if it doesn't belong to a **DDSurface**(p. 98).

This method also deletes the primitive that make up the DDLoop if they are not used by any other primitives. Calls the delete function of each primitive that makes up the DDLoop.

3.7.3.8 virtual DDEdgeUseList::iterator DDLoop::erase (std::list< DDEdgeUse * >::iterator *pos*) [virtual]

Remove the **DDEdgeUse**(p. 80) represented by *pos* from the DDLoop.

The return value is an iterator to the **DDEdgeUse**(p. 80) immediately following the one removed. Unregisters the DDLoop with the **DDEdgeUse**(p. 80).

3.7.3.9 virtual unsigned long DDLoop::get_ID () [virtual]

Returns the ID of the DDLoop.

3.7.3.10 virtual void DDLoop::get_ID (unsigned long & *r_id*) [virtual]

Retrieves the ID of the DDLoop.

Parameters:

r_id The variable used to return the id.

3.7.3.11 unsigned long DDLoop::get_ID_generator () [static]

This function returns the ID of the last instance created; A new instance that is created and assigned an id manually should be assigned an id that is greater that the returned value.

3.7.3.12 DD_RESULT DDLoop::get_instance_map (DDHashMap< unsigned long, DDLoop * > & *r_loop_map*) [static]

Get a map containing pointers to all of the instances of the class indexed by instance id. See the description of the constructor for more information on which instances are saved in the instance map.

Parameters:

r_loop_map The map where the pointers are to be returned.

3.7.3.13 unsigned long DDLoop::get_instance_map_size () [static]

Returns the number of instances of the class. See the description of the constructor for more information on which instances are saved in the instance map.

3.7.3.14 virtual double DDLoop::get_length () [virtual]

Calculates the length of the DDLoop.

3.7.3.15 DD_RESULT DDLoop::get_loop (unsigned long & *r_id*, DDLoop *& *rp_loop*) [static]

Get the DDLoop with the id *r_id*.

Parameters:

r_id The id of the DDLoop to get.

rp_loop The variable where the DDLoop is returned.

Return values:

DD_SUCCESS If the DDLoop was found.

DD_ERROR_PRIMITIVE_NOT_FOUND If a DDLoop was not found with the id *r_id*.

3.7.3.16 virtual void DDLoop::get_loop_type (DDLoopType & *r_loop_type*) [virtual]

Gets the DDLoop type.

Return values:

DDExteriorLoop If the DDLoop is an exterior loop of a **DDSurface**(p. 98).

DDInteriorLoop If the DDLoop is an interior loop of a **DDSurface**(p. 98).

3.7.3.17 virtual void DDLoop::get_neighboring_loops (std::list< DDLoop * > & *r_neighboring_loops*) [virtual]

Finds all of the DDLoops that are neighbors to this loop.

The neighboring DDLoops share common DDEdges through instances of **DDEdgeUse**(p. 80).

3.7.3.18 virtual void DDLoop::get_point (double *t*, DDPoint & *r_point*) [virtual]

Calculates the coordinates of the point that is the distance *r_t* along the DDLoop.

Parameters:

t The distance along the loop.

r_point The coordinates of the point at the distance *t* along the DDLoop.

3.7.3.19 virtual void DDLoop::get_points (std::list< DDPoint * > & *r_point_list*) [virtual]

Get the DDPoints in the order the are used around the DDLoop. The method checks to make sure that each instance of **DDPoint**(p. 49) is entered into the list. Thus if the DDLoop is not closed all the instances of **DDPoint**(p. 49) will be added in the order that they appear in the DDLoop. If the DDLoop is closed then the first and last instance of **DDPoint**(p. 49) returned will be the same.

Parameters:

r_point_list The variable used to store the list of pointers of DDPoints in the order they are used around the DDLoop. The list is cleared before it is filled.

3.7.3.20 virtual void DDLoop::get_surface (DDSurface *& *rp_surface*) [virtual]

Get the **DDSurface**(p. 98) that uses this instance of DDLoop.

Parameters:

rp_surface The variable used to return the **DDSurface**(p. 98) that is using this instance of DDLoop. NULL is a valid value to return in *rp_surface*. It means that the loop is not used in any surfaces.

3.7.3.21 virtual DDEdgeUseList::iterator DDLoop::insert (DDEdgeUseList::iterator *pos*, DDEdgeUse *& *rp_edge_use*) [virtual]

Insert the **DDEdgeUse**(p. 80) before the position indicated by *pos*.

If the **DDEdgeUse**(p. 80) is already being used in another DDLoop, then the **DDEdgeUse**(p. 80) is not inserted into the DDLoop and an iterator to end() is returned. Otherwise if the **DDEdgeUse**(p. 80) was inserted then an iterator to the **DDEdgeUse**(p. 80) is returned. Registers the DDLoop with the **DDEdgeUse**(p. 80).

3.7.3.22 virtual void DDLoop::pop_back () [virtual]

Remove the **DDEdgeUse**(p. 80) from the back of the DDLoop.

Unregisters the DDLoop with the **DDEdgeUse**(p. 80).

3.7.3.23 virtual void DDLoop::pop_front () [virtual]

Remove the **DDEdgeUse**(p. 80) from the front of the DDLoop.

Unregisters the DDLoop with the **DDEdgeUse**(p. 80).

3.7.3.24 virtual void DDLoop::push_back (DDEdgeUse *& *rp_edge_use*) [virtual]

Push an **DDEdgeUse**(p. 80) to the back of the DDLoop.

Doesn't insert the DDEdgeUseif the **DDEdgeUse**(p. 80) is already being used in another DDLoop. Registers the DDLoop with the **DDEdgeUse**(p. 80).

3.7.3.25 virtual void DDLoop::push_front (DDEdgeUse *& *rp_edge_use*) [virtual]

Push a **DDEdgeUse**(p. 80) to the front of the DDLoop.

Doesn't insert the **DDEdgeUse**(p. 80) if the **DDEdgeUse**(p. 80) is already being used in another DDLoop. Registers the DDLoop with the **DDEdgeUse**(p. 80).

3.7.3.26 virtual DD_RESULT DDLoop::register_surface (DDSurface * *p_surface*, DDLoopType *loop_type*) [virtual]

Registers the **DDSurface**(p. 98) that uses this instance.

The method does not notify the **DDSurface**(p. 98) that it has been registered. Because of the 2d nature of the domain, each DDLoop use may only be used by one **DDSurface**(p. 98).

Return values:

DD_SUCCESS If the **DDSurface**(p. 98) was registered.

DD_FAILURE If the DDLoop is already being used by another **DDSurface**(p. 98).

3.7.3.27 virtual void DDLoop::remove (DDEdgeUse * *p_edge_use*) [virtual]

Remove the **DDEdgeUse**(p. 80) *p_edge_use* from the DDLoop.

Unregisters the DDLoop with the **DDEdgeUse**(p. 80).

3.7.3.28 virtual DD_RESULT DDLoop::unregister_surface (DDSurface * *p_surface*) [virtual]

Unregisters the **DDSurface**(p. 98) that no longer uses this instance of DDLoop.

Does not notify the **DDSurface**(p. 98) that it was unregistered.

Parameters:

p_surface The **DDSurface**(p. 98) to unregister.

Return values:

DD_SUCCESS If the surface was unregistered.

DD_ERROR_PRIMITIVE_NOT_FOUND If *p_surface* is not using this instance of DDLoop.

3.8 DDSurface Class Reference

The DDSurface represents a non intersecting surface in 2d space.

`#include <DDSurface.hpp>`

Public Member Functions

- **DDSurface** (**DDLoop** ∗p_loop, unsigned long id=0xffffffff)
- ∼**DDSurface** ()
- void **get_ID** (unsigned long &r_id)
- unsigned long **get_ID** ()
- DD_RESULT **replace_exterior_loop** (**DDLoop** ∗p_old_loop, **DDLoop** ∗p_new_loop)
- void **get_exterior_loop** (**DDLoop** ∗&rp_loop)
- void **get_interior_loop_list** (std::list< **DDLoop** ∗ > &r_interior_loop_list)
- DD_RESULT **add_interior_loop** (**DDLoop** ∗p_loop)
- DD_RESULT **remove_interior_loop** (**DDLoop** ∗p_loop)
- void **remove_all_interior_loops** ()
- void **remove_exterior_loop** ()
- DD_RESULT **get_edgeuses** (**DDPoint** &r_point, **DDEdgeUse** ∗&rp_start_edgeuse, **DDEdgeUse** ∗&rp_end_edgeuse)
- DD_RESULT **get_edgeuses** (**DDPoint** &r_point, std::list< **DDEdgeUse** ∗ > &r_start_-edgeuse_list, std::list< **DDEdgeUse** ∗ > &r_end_edgeuse_list)
- void **get_neighboring_surfaces** (std::list< **DDSurface** ∗ > &r_neighbors)
- void **get_all_edgeuses** (**DDEdgeUseList** &r_edgeuse_list)
- void **get_list_of_loops** (std::list< **DDEdgeUseList** ∗ > &r_list_of_edgeuse_lists)
- void **get_list_of_loops** (std::list< **DDLoop** ∗ > &r_list_of_loops)
- void **create_isolated_copy_of_surface** (**DDSurface** ∗&rp_copy)
- void **create_integrated_copy_of_surface** (**DDSurface** ∗&rp_copy)
- void **calculate_bounding_box** (**DDBoundingBox** &r_bounding_box)
- void **get_points** (std::list< **DDPoint** ∗ > &r_point_list)

Static Public Member Functions

- DD_RESULT **get_surface** (unsigned long &r_id, **DDSurface** ∗&rp_surface)
- void **get_instance_map** (**DDHashMap**< unsigned long, **DDSurface** ∗ > &r_surface_map)
- void **delete_all_instances** ()
- unsigned long **get_ID_generator** ()
- void **delete_surface** (**DDSurface** ∗&rp_surface)

- unsigned long **get_instance_map_size** ()
- void **get_list_of_surface_instances** (std::list< **DDSurface** ∗ > &r_surface_list)

3.8.1 Detailed Description

The DDSurface represents a non intersecting surface in 2d space.

Author:
Corey McBride

Date:
02/07/03

The surface contains one exterior loop and zero or more interior loops. Exterior loops are forward loops or counter clockwise loops while interior loops are backward loops or clockwise loops.

3.8.2 Constructor & Destructor Documentation

3.8.2.1 DDSurface::DDSurface (DDLoop ∗ *p_loop*, unsigned long *id* = `0xffffffff`)

A macro to define memory allocation methods. Constructor.

Creates a DDSurface with *p_loop* as the exterior **DDLoop**(p. 89).

If the **DDLoop**(p. 89) is already being used in another DDSurface then the method will assert. This instance of DDSurface is registered with the **DDLoop**(p. 89). Thus the link from **DDLoop**(p. 89) to DDSurface is maintained automatically.

Parameters:

p_loop The instance of **DDLoop**(p. 89) used as the exterior loop.

id If *id* is 0xffffffff then the instance will be given a unique ID based upon the order of creation of the instance. The DDSurface will also be added to the list of all DDSurface. If *id* is 0 then the DDSurface will not be added to the static list of all DDSurfaces and will not be given a unique id. Instead the instance will have an id of 0. If *id* is between 0 and 0xffffffff then the instance will be assigned the ID *id*. However, the IDs must be assigned in assending order or the constructor will assert.

3.8.2.2 DDSurface::~DDSurface ()

Destructor.

The DDSurface is removed from the static list of all DDSurface instances. The DDSurface is unregistered with the interior and exterior loops.

3.8.3 Member Function Documentation

3.8.3.1 DD_RESULT DDSurface::add_interior_loop (DDLoop * *p_loop*)

Adds the **DDLoop**(p. 89) *p_loop* as an interior loop in the DDSurface. The DDSurface is registered with the **DDLoop**(p. 89).

Return values:

DD_FAILURE If the **DDLoop**(p. 89) already is being used by another DDSurface.

DD_X_NULL_POINTER If *rp_loop* contains a NULL pointer.

3.8.3.2 void DDSurface::calculate_bounding_box (DDBoundingBox & *r_bounding_box*)

Calculates the bounding box of the DDSurface.

Parameters:

r_bounding_box The **DDBoundingBox**(p. 118) used to return the bounding box calculated for the DDSurface.

3.8.3.3 void DDSurface::create_integrated_copy_of_surface (DDSurface *& *rp_copy*)

Creates a copy of the DDSurface.

The new DDSurface uses the origianl instance of **DDPoint**(p. 49) but has new instances of **DDEdge**(p. 55), **DDEdgeUse**(p. 80) and **DDLoop**(p. 89) that are copies of the ones used by this instance.

3.8.3.4 void DDSurface::create_isolated_copy_of_surface (DDSurface *& *rp_copy*)

Creates a copy of the DDSurface.

The new DDSurface contains new instances of the **DDPoint**(p. 49), **DDEdge**(p. 55), **DDEdge-Use**(p. 80) and **DDLoop**(p. 89) that are copies of the ones used by this instance.

3.8.3.5 void DDSurface::delete_all_instances () [static]

This method deletes all instance of the class.

This function should only be used to clean up memory at the end of a program. Higher order objects should be deleted first to make sure that this instance is not being used by another class.

3.8.3.6 void DDSurface::delete_surface (DDSurface *& *rp_surface*) [static]

Deletes the DDSurface.

Deletes the DDSurface and any of the primitives that make up the DDSurface if the primitives are not being used by other primitives.

3.8.3.7 void DDSurface::get_all_edgeuses (DDEdgeUseList & *r_edgeuse_list*)

Retrieves a list of all of the DDEdgeUses used by the DDSurface.

The DDEdgeUses that are used by each **DDLoop**(p. 89) in this DDSurface are returned in one list. The order that the DDEdgeUses appears in the list is not guarranteed.

Parameters:

r_edgeuse_list The list where all of the DDEdgeUses used by this DDSurface are returned.

3.8.3.8 DD_RESULT DDSurface::get_edgeuses (DDPoint & *r_point*, std::list< DDEdgeUse * > & *r_start_edgeuse_list*, std::list< DDEdgeUse * > & *r_end_edgeuse_list*)

This function finds the DDEdgeUses in the DDSurface that use the provided **DDPoint**(p. 49).

Parameters:

r_point The **DDPoint**(p. 49) from the DDSurface.

r_start_edgeuse_list The list where the **DDEdgeUse**(p. 80) that uses the **DDPoint**(p. 49) as the start point is returned.

r_end_edgeuse_list The list where the DDEdgeUses that uses the **DDPoint**(p. 49) as the end point is returned

Return values:

DD_SUCCESS If the DDEdgeUses were found.

DD_FAILURE If the DDEdgeUses were not found.

3.8.3.9 DD_RESULT DDSurface::get_edgeuses (DDPoint & *r_point*, DDEdgeUse *& *rp_start_edgeuse*, DDEdgeUse *& *rp_end_edgeuse*)

This function finds the DDEdgeUses in the DDSurface that use the provided **DDPoint**(p. 49).

Because the DDSurfaces are not self intersecting each **DDPoint**(p. 49) should only be used twice by each DDSurface. This method does not check to make sure that a DDSurface is not self intersecting.

Bug

There are instances where an interior **DDLoop**(p. 89) may be tangent to another **DDLoop**(p. 89) in the same DDSurface. After the DDSurface is intersected with another DDSuface this tangent point may be merged with another **DDPoint**(p. 49). This results in a situation where one **DDPoint**(p. 49) is used multiple times in a DDSurface. The other version of this function takes this into account.

Parameters:

r_point The **DDPoint**(p. 49) from the DDSurface.

rp_start_edgeuse The variable where the **DDEdgeUse**(p. 80) that uses the **DDPoint**(p. 49) as the start point is returned.

rp_end_edgeuse The variable where the DDEdgeUses that uses the **DDPoint**(p. 49) as the end point is returned

Return values:

DD_SUCCESS If the DDEdgeUses were found.

DD_FAILURE If the DDEdgeUses were not found.

3.8.3.10 void DDSurface::get_exterior_loop (DDLoop *& *rp_loop*)

Get the exterior **DDLoop**(p. 89) that the DDSurface uses.

Parameters:

rp_loop The pointer where the exterior **DDLoop**(p. 89) is returned. NULL is a valid value to return in *rp_loop*. It means that the DDSurface does not have a exterior **DDLoop**(p. 89).

3.8.3.11 unsigned long DDSurface::get_ID ()

Retrieves the ID of the DDSurface.

3.8.3.12 void DDSurface::get_ID (unsigned long & *r_id*)

Retrieves the ID of the DDSurface.

Parameters:

r_id The variable where the id is returned.

3.8.3.13 unsigned long DDSurface::get_ID_generator () `[static]`

This function returns the ID of the last instance created; A new instance that is created and assigned an id manually should be assigned an id that is greater that the returned value.

3.8.3.14 void DDSurface::get_instance_map (DDHashMap< unsigned long, DDSurface * > & *r_surface_map*) `[static]`

Get a map containing pointers to all of the instances of the class indexed by instance id. See the description of the constructor for more information on which instances are saved in the instance map.

Parameters:

r_surface_map The variable where the static surface instance map is copied.

3.8.3.15 unsigned long DDSurface::get_instance_map_size () `[static]`

Returns the number of instances of the class. See the description of the constructor for more information on which instances are saved in the instance map.

3.8.3.16 void DDSurface::get_interior_loop_list (std::list< DDLoop * > & *r_interior_loop_list*)

Retrieves a list of DDLoops that are used as interior loops in the DDSurface.

Parameters:

r_interior_loop_list The list used to return the DDLoops that are used as interior loops in this instance of DDSurface.

3.8.3.17 void DDSurface::get_list_of_loops (std::list< DDLoop * > & *r_list_of_loops*)

Retrieves a list of all of the DDLoops that are used by the DDSurface.

The order that the DDLoops appear in the list is not guaranteed.

Parameters:

r_list_of_loops The list where all of the DDLoops used by this DDSurface are returned.

3.8.3.18 void DDSurface::get_list_of_loops (std::list< DDEdgeUseList * > & *r_list_of_edgeuse_lists*)

Retrieves a list of all of the DDLoops that are used by the DDSurface.

The order that the DDLoops appear in the list is not guaranteed.

Parameters:

r_list_of_edgeuse_lists The list where all of the DDLoops used by this DDSurface are returned.

3.8.3.19 void DDSurface::get_list_of_surface_instances (std::list< DDSurface * > & *r_surface_list*) `[static]`

Gets a list of all of the DDSurface instances.

Parameters:

r_surface_list The list where all of the DDSurface instances are returned.

3.8.3.20 void DDSurface::get_neighboring_surfaces (std::list< DDSurface * > & *r_neighbors*)

Returns a list of all of the DDSurfaces that are neighbors of this DDSurface.

A neighboring DDSurface has a least one instance of **DDEdge**(p. 55) in common with this instance.

Parameters:

r_neighbors The list where the neighboring DDSurfaces are returned.

3.8.3.21 void DDSurface::get_points (std::list< DDPoint * > & *r_point_list*)

Gets a list of all of the DDPoints used in the DDSurface The order that the DDPoints appear in the list is not guaranteed.

Parameters:

r_point_list The list where all of the DDPoints used by this DDSurface are returned.

3.8.3.22 DD_RESULT DDSurface::get_surface (unsigned long & *r_id*, DDSurface *& *rp_surface*) `[static]`

Get the DDSurface with the id *r_id*.

Parameters:

r_id The id of the DDSurface to get.

rp_surface The variable used to return the DDSurface.

Return values:

DD_SUCCESS If the DDSurface was found.

DD_ERROR_PRIMITIVE_NOT_FOUND If the DDSurface was not found.

3.8.3.23 void DDSurface::remove_all_interior_loops ()

Removes all DDLoops from the DDSurface.

Unregisters the DDSurface with the each **DDLoop**(p. 89).

3.8.3.24 void DDSurface::remove_exterior_loop ()

Removes the exterior **DDLoop**(p. 89) from the DDSurface.

Unregisters the DDSurface with the exterior **DDLoop**(p. 89).

3.8.3.25 DD_RESULT DDSurface::remove_interior_loop (DDLoop * *p_loop*)

Removes the **DDLoop**(p. 89) *rp_loop* as an interior loop in the DDSurface.

Unregisters the DDSurface with the **DDLoop**(p. 89).

Return values:

DD_SUCCESS If the **DDLoop**(p. 89) was removed.

DD_X_NULL_POINTER If *rp_loop* contains a NULL pointer.

3.8.3.26 DD_RESULT DDSurface::replace_exterior_loop (DDLoop * *p_old_loop*, DDLoop * *p_new_loop*)

Replaces the exterior **DDLoop**(p. 89) with another.

This DDSurface is unregistered with the original **DDLoop**(p. 89) and registered with the new **DDLoop**(p. 89).

Parameters:

p_old_loop The **DDLoop**(p. 89) to be replaced.

p_new_loop The **DDLoop**(p. 89) to replace the original **DDLoop**(p. 89).

Return values:

DD_X_NULL_POINTER If either **DDLoop**(p. 89) pointer is NULL.

DD_ERROR_PRIMITIVE_NOT_FOUND IF the original **DDLoop**(p. 89) is not being used by the DDSurface.

DD_SUCCESS If the replace was successful.

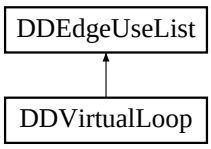
DD_FAILURE If the new **DDLoop**(p. 89) already is being used by another DDSurface.

3.9 DDVirtualLoop Class Reference

The DDVirtualLoop represents a closed loop of DDEdges. The order of the DDEdges in the DDVirtualLoop is important because it shows which DDEdges are next to each other. DDEdges that are next to each other should share a common point. This class is dereived from **DDEdgeUseList**(p. 122). This loop is virtual because it is not registered with the DDEdgeUses that is contains. This allows an DDEdgeUses to belong to one **DDLoop**(p. 89) and multiple DDVirtualLoop.

`#include <DDVirtualLoop.hpp>`

Inheritance diagram for DDVirtualLoop::



Public Member Functions

- **DDVirtualLoop** ()
- virtual ∼**DDVirtualLoop** ()
- void **calculate_bounding_box** (**DDBoundingBox** &r_bounding_box)
- virtual DD_RESULT **register_surface** (**DDVirtualSurface** ∗p_surface, DDLoopType loop_type)
- virtual DD_RESULT **get_loop_type** (DDLoopType &r_loop_type)
- virtual DD_RESULT **unregister_surface** (**DDVirtualSurface** ∗p_surface)
- virtual DD_RESULT **get_surface** (**DDVirtualSurface** ∗&rp_surface)
- virtual DD_RESULT **get_points** (std::list< **DDPoint** ∗ > &r_point_list)
- virtual DD_RESULT **create_integrated_copy_of_loop** (**DDLoop** ∗&rp_copy)

Static Public Member Functions

- DD_RESULT **delete_loop** (**DDVirtualLoop** ∗&rp_loop)

3.9.1 Detailed Description

The DDVirtualLoop represents a closed loop of DDEdges. The order of the DDEdges in the DDVirtualLoop is important because it shows which DDEdges are next to each other. DDEdges

that are next to each other should share a common point. This class is dereived from **DDEdgeUseList**(p. 122). This loop is virtual because it is not registered with the DDEdgeUses that is contains. This allows an DDEdgeUses to belong to one **DDLoop**(p. 89) and multiple DDVirtualLoop.

3.9.2 Constructor & Destructor Documentation

3.9.2.1 DDVirtualLoop::DDVirtualLoop ()

Constructor.

3.9.2.2 virtual DDVirtualLoop::∼DDVirtualLoop () [virtual]

/brief Destructor

If the DDVirtualLoop is registered with any **DDVirtualSurface**(p. 111) then the function will assert.

3.9.3 Member Function Documentation

3.9.3.1 void DDVirtualLoop::calculate_bounding_box (DDBoundingBox & *r_bounding_box*)

Calculates the bounding box of the DDVirtualLoop.

Parameters:

r_bounding_box The **DDBoundingBox**(p. 118) used to return the bounding box calculated for the DDVirtualLoop.

3.9.3.2 virtual DD_RESULT DDVirtualLoop::create_integrated_copy_of_loop (DDLoop *& *rp_copy*) [virtual]

Creates a **DDLoop**(p. 89) copy of the DDVirtualLoop.

The copy uses the same instances of the end DDPoints as this instance. But has a new instance of **DDEdge**(p. 55) and **DDEdgeUse**(p. 80) that are a copy of the one used by this instance.

3.9.3.3 DD_RESULT DDVirtualLoop::delete_loop (DDVirtualLoop *& *rp_loop*) [static]

Deletes the DDVirtualLoop if it doesn't belong to a **DDVirtualSurface**(p. 111).

Also deletes the primitive that make up the DDVirtualLoop if they are not used by any other primitives. Calls the delete function of each primitive that makes up the loop.

3.9.3.4 virtual DD_RESULT DDVirtualLoop::get_loop_type (DDLoopType & *r_loop_type*) [virtual]

Gets the **DDVirtualSurface**(p. 111) type.

Return values:

DDExteriorLoop If the **DDVirtualSurface**(p. 111) is an exterior loop of a **DDVirtualSurface**(p. 111).

DDInteriorLoop If the **DDVirtualSurface**(p. 111) is an interior loop of a **DDVirtualSurface**(p. 111).

3.9.3.5 virtual DD_RESULT DDVirtualLoop::get_points (std::list< DDPoint * > & *r_point_list*) [virtual]

Get the DDPoints in order around the DDVirtualLoop.

Clears the list before filling it with the DDPoints of the DDVirtualLoop. The method checks to make sure that each **DDPoint**(p. 49) is entered into the list. Thus if the DDVirtualLoop is not closed all the DDPoints will be added in the order that they appear in the DDVirtualLoop. If the DDVirtualLoop is closed then the first and last DDPoints will be the same.

3.9.3.6 virtual DD_RESULT DDVirtualLoop::get_surface (DDVirtualSurface *& *rp_surface*) [virtual]

Get the **DDVirtualSurface**(p. 111) that uses the DDVirtualLoop.

NULL is a valid value to return in *rp_surface*. It means that the DDVirtualLoop is not used in any **DDVirtualSurface**(p. 111).

Reimplemented from **DDEdgeUseList** (p. 122).

3.9.3.7 virtual DD_RESULT DDVirtualLoop::register_surface (DDVirtualSurface * *p_surface*, DDLoopType *loop_type*) `[virtual]`

Registers the **DDVirtualSurface**(p. 111) that uses this DDVirtualLoop.

Does not notify the **DDVirtualSurface**(p. 111) that is is being used. Because of the 2d nature of the domain, each DDVirtualLoop use may only be used once.

Return values:

DD_SUCCESS If the **DDVirtualSurface**(p. 111) was added.

DD_FAILURE If the DDVirtualLoop is already being used by a **DDVirtualSurface**(p. 111).

3.9.3.8 virtual DD_RESULT DDVirtualLoop::unregister_surface (DDVirtualSurface * *p_surface*) `[virtual]`

Unregisters the **DDVirtualSurface**(p. 111) with the DDVirtualLoop.

Does not notify the **DDVirtualSurface**(p. 111) that it was unregistered.

Return values:

DD_SUCCESS If the **DDVirtualSurface**(p. 111) was removed.

DD_ERROR_PRIMITIVE_NOT_FOUND If p_surface is not currently being used by the DDVirtualLoop.

3.10 DDVirtualSurface Class Reference

The DDVirtualSurface represents a non intersecting surface in 2d space. The surface contains one exterior loop and zero or more interior loops. Exterior loops are forward loops while interior loops are backward loops. The surface is composed of virtual loops. Thus a virtual surface may span multiple real surfaces.

`#include <DDVirtualSurface.hpp>`

Public Member Functions

- **DDVirtualSurface** (**DDVirtualLoop** ∗p_loop)
- ∼**DDVirtualSurface** ()
- DD_RESULT **replace_exterior_loop** (**DDVirtualLoop** ∗p_old_loop, **DDVirtualLoop** ∗p_-new_loop)
- void **get_exterior_loop** (**DDVirtualLoop** ∗&rp_loop)
- void **get_interior_loop_list** (std::list< **DDVirtualLoop** ∗ > &r_interior_loop_list)
- std::list< **DDVirtualLoop** ∗ > ∗ **get_interior_loop_list** ()
- DD_RESULT **add_interior_loop** (**DDVirtualLoop** ∗p_loop)
- DD_RESULT **remove_interior_loop** (**DDVirtualLoop** ∗p_loop)
- DD_RESULT **get_neighboring_surfaces** (std::list< **DDSurface** ∗ > &r_neighbors)
- void **calculate_bounding_box** (**DDBoundingBox** &r_bounding_box)
- unsigned long **get_number_of_interior_loops** ()

Static Public Member Functions

- DD_RESULT **delete_surface** (**DDVirtualSurface** ∗&rp_surface)

3.10.1 Detailed Description

The DDVirtualSurface represents a non intersecting surface in 2d space. The surface contains one exterior loop and zero or more interior loops. Exterior loops are forward loops while interior loops are backward loops. The surface is composed of virtual loops. Thus a virtual surface may span multiple real surfaces.

Author:
Corey McBride

Date:
06/07/03

3.10.2 Constructor & Destructor Documentation

3.10.2.1 DDVirtualSurface::DDVirtualSurface (DDVirtualLoop ∗ *p_loop*)

A macro to define memory allocation methods.

/brief Constructor

The exterior **DDVirtualLoop**(p. 107) is assigned to the **DDVirtualLoop**(p. 107) contained in *p_loop*. The method will assert if the **DDVirtualLoop**(p. 107) pointer is NULL.

3.10.2.2 DDVirtualSurface::∼DDVirtualSurface ()

Destructor.

3.10.3 Member Function Documentation

3.10.3.1 DD_RESULT DDVirtualSurface::add_interior_loop (DDVirtualLoop ∗ *p_loop*)

Adds a **DDVirtualLoop**(p. 107) as an interior loop of the DDVirtualSurface.

Parameters:

p_loop The **DDVirtualLoop**(p. 107) to add as an interior loop.

Return values:

DD_SUCCESS If the **DDVirtualLoop**(p. 107) was added.

DD_FAILURE If the **DDVirtualLoop**(p. 107) already is being used by another DDVirtualSurface.

DD_X_NULL_POINTER If *rp_loop* contains a NULL pointer.

3.10.3.2 void DDVirtualSurface::calculate_bounding_box (DDBoundingBox & *r_bounding_box*)

Calculates the bounding box of the DDVirtualSurface.

Parameters:

r_bounding_box The **DDBoundingBox**(p. 118) used to return the bounding box calculated for the DDVirtualSurface.

3.10.3.3 DD_RESULT DDVirtualSurface::delete_surface (DDVirtualSurface *& *rp_surface*) [static]

Delete the DDVirtualSurface.

Deletes the **DDSurface**(p. 98) and any of the primitives that make up the DDVirtualSurface if the primitives are not being used by other primitives.

3.10.3.4 void DDVirtualSurface::get_exterior_loop (DDVirtualLoop *& *rp_loop*)

Get the exterior **DDVirtualLoop**(p. 107) that the DDVirtualSurface uses.

Parameters:

rp_loop The variable used to return the exterior **DDVirtualLoop**(p. 107).

3.10.3.5 std::list<DDVirtualLoop*>* DDVirtualSurface::get_interior_loop_list ()

Returns the interior **DDVirtualLoop**(p. 107) that are part of this DDVirtualSurface.

3.10.3.6 void DDVirtualSurface::get_interior_loop_list (std::list< DDVirtualLoop * > & *r_interior_loop_list*)

Retrieves the interior **DDVirtualLoop**(p. 107) that are part of this DDVirtualSurface.

Parameters:

r_interior_loop_list The list in which the interior DDVirtualLoops are returned.

3.10.3.7 DD_RESULT DDVirtualSurface::get_neighboring_surfaces (std::list< DDSurface * > & *r_neighbors*)

Returns a list of all of the DDSurfaces that are neighbors of this DDVirtualSurface.

Because the surface is virtual it may span multiple surfaces and thus this routine will return all of the neighbors of all of the real surfaces that this virtual surface spans. A neighboring DDSurfaces have a least one instance of **DDEdge**(p. 55) in common.

Parameters:

r_neighbors The list where the neighboring DDSurfaces are returned.

3.10.3.8 unsigned long DDVirtualSurface::get_number_of_interior_loops ()

Returns the number of interior **DDVirtualLoop**(p. 107) that make up the DDVirtualSurface.

3.10.3.9 DD_RESULT DDVirtualSurface::remove_interior_loop (DDVirtualLoop * *p_loop*)

Removes the **DDVirtualLoop**(p. 107) as an interior loop of the DDVirtualSurface.

Return values:

DD_SUCCESS If the **DDVirtualLoop**(p. 107) was removed.

DD_X_NULL_POINTER If *rp_loop* contains a NULL pointer.

3.10.3.10 DD_RESULT DDVirtualSurface::replace_exterior_loop (DDVirtualLoop * *p_old_loop*, DDVirtualLoop * *p_new_loop*)

Replaces exterior **DDVirtualLoop**(p. 107) for another **DDVirtualLoop**(p. 107).

Parameters:

p_old_loop The **DDVirtualLoop**(p. 107) to be replaced.

p_new_loop The **DDVirtualLoop**(p. 107) to replace the original **DDVirtualLoop**(p. 107).

Return values:

DD_X_NULL_POINTER If either **DDVirtualLoop**(p. 107) pointer is NULL.

DD_ERROR_PRIMITIVE_NOT_FOUND If the original **DDVirtualLoop**(p. 107) is not being used by the DDVirtualSurface.

DD_SUCCESS If the replace was successful.

DD_FAILURE If the **DDVirtualLoop**(p. 107) already is being used by another DDVirtualSurface.

3.11 DDMaskDefinitions Class Reference

This class is used to store all of the surfaces that make up a mask set. The masks are grouped according to layer name.

`#include <DDMaskDefinitions.hpp>`

Public Member Functions

- void **add_mask_definition** (std::string &r_layer, std::list< **DDSurface** ∗ > ∗p_surface_list)
- DD_RESULT **get_surfaces_by_layer** (std::string &r_layer, std::list< **DDSurface** ∗ > &r_surface_list)
- void **get_layer_names** (std::list< std::string > &r_layer_names)
- **DDBoundingBox get_bounding_box** ()
- void **set_bounding_box** (**DDBoundingBox** &r_bounding_box)
- void **print_information** ()

Protected Attributes

- std::map< std::string, std::list< **DDSurface** ∗ > ∗ > **mMaskMap**

3.11.1 Detailed Description

This class is used to store all of the surfaces that make up a mask set. The masks are grouped according to layer name.

Author:
Corey McBride

Date:
05/15/03

3.11.2 Member Function Documentation

3.11.2.1 void DDMaskDefinitions::add_mask_definition (std::string & *r_layer*, std::list< DDSurface ∗ > ∗ *p_surface_list*)

Stores a list of **DDSurface**(p. 98) pointers indexed by a layer name.

Parameters:

r_layer The layer name used to index the list of DDSurfaces.

p_surface_list The list of **DDSurface**(p. 98) pointers to store.

3.11.2.2 DDBoundingBox DDMaskDefinitions::get_bounding_box ()

Returns the **DDBoundingBox**(p. 118) that was set with set_bounding_box.

This method does not calculate the bounding box.

3.11.2.3 void DDMaskDefinitions::get_layer_names (std::list< std::string > & *r_layer_names*)

Gets a list of all of the layer namer currently stored.

Parameters:

r_layer_names The list used to return the layer names.

3.11.2.4 DD_RESULT DDMaskDefinitions::get_surfaces_by_layer (std::string & *r_layer*, std::list< DDSurface * > & *r_surface_list*)

Gets a list of **DDSurface**(p. 98) pointers stored by the layer name.

Parameters:

r_layer The layer name used to index the list of DDSurfaces.

r_surface_list The list used to return the **DDSurface**(p. 98) pointers.

Return values:

DD_SUCCESS If a list was found under the provided layer name.

DD_FAILURE If a list was not found under the provided layer name.

3.11.2.5 void DDMaskDefinitions::print_information ()

Prints the layer names used to index the **DDSurface**(p. 98) lists.

3.11.2.6 void DDMaskDefinitions::set_bounding_box (DDBoundingBox & *r_bounding_box*)

Sets the **DDBoundingBox**(p. 118) for this set of mask definitions.

This method does not calculate the bounding box.

Parameters:

r_bounding_box The **DDBoundingBox**(p. 118) for this mask set.

3.11.3 Member Data Documentation

3.11.3.1 std::map<std::string,std::list<DDSurface*>*> DDMaskDefinitions::mMaskMap [protected]

A macro to define the memory allocation functions.

3.12 DDBoundingBox Class Reference

The min and max coordinates of a bounding box..

`#include <DDLoop.hpp>`

Public Types

- enum

Public Member Functions

- **DDBoundingBox** ()
- **DDBoundingBox** (double xmax, double xmin, double ymax, double ymin)
- **DDBoundingBox** & **operator** ∗= (const double &r_scalar)
- **DDBoundingBox** & **operator+=** (const double &r_scalar)
- int **calculate_inside_outside_code** (**DDPoint** &r_point)
- void **calculate_centroid** (**DDPoint** &r_point)

Public Attributes

- double **mXmin**
- double **mXmax**
- double **mYmin**
- double **mYmax**

Friends

- **DDBoundingBox operator+** (const **DDBoundingBox** &r_box1, const **DDBoundingBox** &r_box2)
- bool **operator%** (**DDBoundingBox** &box1, **DDBoundingBox** &box2)

3.12.1 Detailed Description

The min and max coordinates of a bounding box..

3.12.2 Member Enumeration Documentation

3.12.2.1 anonymous enum

An enum for the inside outside codes.

Each code containts bits that are set if a value is outside the corresponding bounding box values.

0x01010000 | 0x00010000 | 0x00010100 | | ———————.....——————— . . 0x01000000 . 0x00000000 . 0x00000100 . . ———————..... .....——————— | | 0x01000001 | 0x00000001 | 0x00000101

3.12.3 Constructor & Destructor Documentation

3.12.3.1 DDBoundingBox::DDBoundingBox () `[inline]`

Constructor.

Initializes the values to 0.0.

3.12.3.2 DDBoundingBox::DDBoundingBox (double *xmax*, double *xmin*, double *ymax*, double *ymin*) `[inline]`

Constructor.

Initializes the bounding box to the values provided.

3.12.4 Member Function Documentation

3.12.4.1 void DDBoundingBox::calculate_centroid (DDPoint & *r_point*)

Calculates the centroid of the bounding box.

3.12.4.2 int DDBoundingBox::calculate_inside_outside_code (DDPoint & *r_point*)

Calculates the inside-outside code for a point.

3.12.4.3 DDBoundingBox& DDBoundingBox::operator ∗= (const double & *r_scalar*) [inline]

Scales the bounding box by the value provided.

3.12.4.4 DDBoundingBox& DDBoundingBox::operator+= (const double & *r_scalar*) [inline]

extends the bounding box by the value provided.

3.12.5 Friends And Related Function Documentation

3.12.5.1 bool operator% (DDBoundingBox & *box1*, DDBoundingBox & *box2*) [friend]

Intersection operator.

Return values:

True if the two bounding boxes intersect.

False if the two bounding boxes do not intersect.

3.12.5.2 DDBoundingBox operator+ (const DDBoundingBox & *r_box1*, const DDBoundingBox & *r_box2*) [friend]

Returns a bounding box that encloses the bounding boxes provided.

3.12.6 Member Data Documentation

3.12.6.1 double DDBoundingBox::mXmax

The maximum X value of the bounding box.

3.12.6.2 double DDBoundingBox::mXmin

The minimin X value of the bounding box.

3.12.6.3 double DDBoundingBox::mYmax

The maximum X value of the bounding box.

3.12.6.4 double DDBoundingBox::mYmin

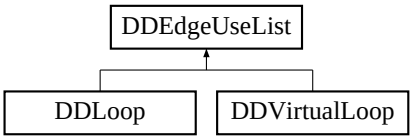
The minimin Y value of the bounding box.

3.13 DDEdgeUseList Class Reference

This class is a wrapper class for std::list<DDEdgeUse∗>.

`#include <DDEdgeUseList.hpp>`

Inheritance diagram for DDEdgeUseList::



3.13.1 Detailed Description

This class is a wrapper class for std::list<DDEdgeUse∗>.

The purpose of this class is to wrap the std::list<DDEdgeUse∗> class. This was done to provide a cleaner interface and to provide common memory allocation methods. Common memory allocation methods are necessary for the library to work properly. This class behaves exactly as a std::list<DDEdgeUse∗>.

Reimplemented in **DDLoop** (p. 90), and **DDVirtualLoop** (p. 107).

3.14 DDHashMap Class Reference

These classes are wrappers for the hash_map and hash_set classes.

```
#include <DDHashMap.hpp>
```

3.14.1 Detailed Description

These classes are wrappers for the hash_map and hash_set classes.

Author:
Corey McBride

Date:
12/10/03

hash_map and hash_set are included in stlport and in the GNU c.xx compilers. These wrapper classes overcome some problems with stlport.

3.15 DDQuadTree< X, E > Class Template Reference

This template class is used to perform a spatial sort of items in 2d space.

`#include <DDQuadTree.hpp>`

Public Member Functions

- **DDQuadTree** (std::list< X ∗ > &r_points, double tolerance, int min_nodes_per_box=-1, double min_box_dimension=-1.0)
- ∼**DDQuadTree** ()
- void **points_near** (**DDVector** &r_position, std::list< X ∗ > &r_result_list)
- void **tree_size** (std::list< int > &r_count_at_each_level, std::list< int > &r_leaves_at_each_-level)

3.15.1 Detailed Description

template<class X, class E = DDDefaultPointQuery<X>> class DDQuadTree< X, E >

This template class is used to perform a spatial sort of items in 2d space.

Author:
Corey McBride

Date:
05/19/04

3.15.2 Constructor & Destructor Documentation

3.15.2.1 template<class X, class E> DDQuadTree< X, E >::DDQuadTree (std::list< X ∗ > & *r_points*, double *tolerance*, int *min_nodes_per_box* = `-1`, double *min_box_dimension* = `-1.0`)

Constructor.

Parameters:

r_points The list of items to sort.

tolerance The tolerance used for comparison of items.

min_nodes_per_box The number of items or less to reside in each box.

min_box_dimension The minimum dimension or less of each box.

3.15.2.2 template<class X, class E> DDQuadTree< X, E >::~DDQuadTree ()

Destructor.

3.15.3 Member Function Documentation

3.15.3.1 template<class X, class E> void DDQuadTree< X, E >::points_near (DDVector & *r_position*, std::list< X * > & *r_result_list*)

Gets the list of all items for which the distance between the passed position and the item is less than or equal to the tolerance value pass to the constructor.

3.15.3.2 template<class X, class E = DDDefaultPointQuery<X>> void DDQuadTree< X, E >::tree_size (std::list< int > & *r_count_at_each_level*, std::list< int > & *r_leaves_at_each_level*)

Get the size of the tree at each level.

3.16 DDAGUIntersection Class Reference

The class stores the information for an intersection between two DDEdges.

`#include <DDAnalyticalGeometryUtil.hpp>`

Public Member Functions

- **DDAGUIntersection** ()
- ∼**DDAGUIntersection** ()
- void **reset** ()

Public Attributes

- **DDEdge** ∗ **mpAEdge**
- **DDEdgeUse** ∗ **mpAEdgeUse**
- **DDPoint** ∗ **mpAPoint**
- double **mAX**
- double **mAY**
- double **mAOrdering**
- DD_RESULT **mAPointType**
- **DDEdge** ∗ **mpBEdge**
- **DDEdgeUse** ∗ **mpBEdgeUse**
- **DDPoint** ∗ **mpBPoint**
- double **mBX**
- double **mBY**
- double **mBOrdering**
- DD_RESULT **mBPointType**
- DD_RESULT **mIntersectionType**
- bool **mVisited**

3.16.1 Detailed Description

The class stores the information for an intersection between two DDEdges.

3.16.2 Constructor & Destructor Documentation

3.16.2.1 DDAGUIntersection::DDAGUIntersection () [inline]

Constructor.

Sets the pointers to Null, the values to 0.0 and types to DD_FAILURE.

3.16.2.2 DDAGUIntersection::~DDAGUIntersection () [inline]

Destructor

3.16.3 Member Function Documentation

3.16.3.1 void DDAGUIntersection::reset () [inline]

This routine sets the parameters to the same state that they were in when they were first created.

For more information see the comments for the Constructor.

3.16.4 Member Data Documentation

3.16.4.1 double DDAGUIntersection::mAOrdering

The parametric parameter of the intersection along *mpAEdge*.

3.16.4.2 DD_RESULT DDAGUIntersection::mAPointType

The type of intersection point for *mpAEdge*.

3.16.4.3 double DDAGUIntersection::mAX

The X coordinate of the intersection.

3.16.4.4 double DDAGUIntersection::mAY

The Y coordinate of the intersection.

3.16.4.5 double DDAGUIntersection::mBOrdering

The parametric parameter of the intersection along *mpBEdge*.

3.16.4.6 DD_RESULT DDAGUIntersection::mBPointType

A pointer to one **DDEdge**(p. 55) involved in the interesection.

3.16.4.7 double DDAGUIntersection::mBX

The X coordinate of the intersection.

3.16.4.8 double DDAGUIntersection::mBY

The Y coordinate of the intersection.

3.16.4.9 DD_RESULT DDAGUIntersection::mIntersectionType

The intersection type.

3.16.4.10 DDEdge* DDAGUIntersection::mpAEdge

A pointer to one **DDEdge**(p. 55) involved in the interesection.

3.16.4.11 DDEdgeUse* DDAGUIntersection::mpAEdgeUse

A pointer to the **DDEdgeUse**(p. 80) that uses *mpAEdge*.

3.16.4.12 DDPoint* DDAGUIntersection::mpAPoint

An existing **DDPoint**(p. 49) from *mpAEdge* that lies at the intersection.

3.16.4.13 DDEdge* DDAGUIntersection::mpBEdge

A pointer to one **DDEdge**(p. 55) involved in the interesection.

3.16.4.14 DDEdgeUse* DDAGUIntersection::mpBEdgeUse

A pointer to the **DDEdgeUse**(p. 80) that uses *mpBEdge*.

3.16.4.15 DDPoint* DDAGUIntersection::mpBPoint

An existing **DDPoint**(p. 49) from *mpBEdge* that lies at the intersection.

3.16.4.16 bool DDAGUIntersection::mVisited

A flag to indicate that the intersection has been visited.

Chapter 4

2D Geometric Routine

This chapter describes the major geometric routines. The routines are grouped in a namespace according to functionality. Each section describes a namespace and gives a list of its routines. A description of each routine is then provided.

4.1 DDUnionOperator Namespace Reference

This namespace contains the routines to Union (OR) one surface with another.

Functions

- DD_RESULT **union_list_of_surfaces** (std::list< **DDSurface** ∗ > &r_surface_list, double tolerance)
- DD_RESULT **union_surface_A_with_surface_B** (**DDSurface** ∗&rp_surface_A, **DDSurface** ∗&rp_surface_B, double tolerance)
- DD_RESULT **union_cw_loop_with_cw_loop** (**DDLoop** ∗&rp_loop_A, **DDLoop** ∗&rp_loop_B, double tolerance)
- DD_RESULT **union_list_of_cw_loops** (std::list< **DDLoop** ∗ > &r_loop_list, double tolerance)

4.1.1 Detailed Description

This namespace contains the routines to Union (OR) one surface with another.

4.1.2 Function Documentation

4.1.2.1 DD_RESULT union_cw_loop_with_cw_loop (DDLoop *& *rp_loop_A*, DDLoop *& *rp_loop_B*, double *tolerance*)

Union two clockwise DDLoops together.

Parameters:

rp_loop_A The first **DDLoop**(p. 89) to be unioned. The result of the operation is stored here also.

rp_loop_B The second **DDLoop**(p. 89) to be unioned. The result of the operation may be stored here.

tolerance The tolerance of the operation.

Return values:

DD_FAILURE There two **DDLoop**(p. 89) do not overlap. The **DDLoop**(p. 89) were not altered.

DD_SUCCESS The two **DDLoop**(p. 89) were unioned together. *rp_loop_A* and *rp_loop_B* were consumed in the process.

4.1.2.2 DD_RESULT union_list_of_cw_loops (std::list< DDLoop * > & *r_loop_list*, double *tolerance*)

Unions the clockwise DDLoops in the list together.

Also known as the OR operation.

Parameters:

r_loop_list The list of DDLoops to be unioned together. The results are also returned in this list.

tolerance The tolerance of the operation.

4.1.2.3 DD_RESULT union_list_of_surfaces (std::list< DDSurface * > & *r_surface_list*, double *tolerance*)

Unions the DDSurfaces in the list together.

Also known as the OR operation.

Parameters:

r_surface_list The list of DDSurfaces to be unioned together. The results are also returned in this list.

tolerance The tolerance of the operation.

4.1.2.4 DD_RESULT union_surface_A_with_surface_B (DDSurface *& *rp_surface_A*, DDSurface *& *rp_surface_B*, double *tolerance*)

Union two DDSurfaces together.

Parameters:

rp_surface_A The first **DDSurface**(p. 98) to be unioned. The result of the operation is stored here also.

rp_surface_B The second **DDSurface**(p. 98) to be unioned. The result of the operation may be stored here.

tolerance The tolerance of the operation.

Return values:

DD_FAILURE There two **DDSurface**(p. 98) do not overlap. The **DDSurface**(p. 98) were not altered.

DD_SUCCESS The two **DDSurface**(p. 98) were unioned together. *rp_surface_A* and *rp_surface_B* were consumed in the process.

4.2 DDVirtualMergeOperator Namespace Reference

This namepsace contains the routines to virtually merge multiple surfaces together. This means that virtual loops are created that represent what the geometry would look like if the surfaces were merged.

Functions

- DD_RESULT **merge_list_of_surfaces** (std::list< **DDSurface** ∗ > &r_list_of_surfaces, **DDVirtualSurface** ∗&rp_virtual_surface, **DDVirtualLoop** ∗&rp_border_loop)
- void **merge_virtual_surface** (**DDVirtualSurface** &r_virtual_surface, **DDSurface** ∗&rp_-surface)
- void **create_virtual_surface_from_surface** (**DDSurface** &r_surface, **DDVirtualSurface** ∗&rp_virtual_surface)

4.2.1 Detailed Description

This namepsace contains the routines to virtually merge multiple surfaces together. This means that virtual loops are created that represent what the geometry would look like if the surfaces were merged.

Author:
Corey McBride

Date:
06/29/03

4.2.2 Function Documentation

4.2.2.1 void create_virtual_surface_from_surface (DDSurface & *r_surface*, DDVirtualSurface ∗& *rp_virtual_surface*)

Creates a **DDVirtualSurface**(p. 111) from a **DDSurface**(p. 98).

Parameters:

r_surface The **DDSurface**(p. 98) to create the **DDVirtualSurface**(p. 111) from.

rp_virtual_surface The variable where the new **DDVirtualSurface**(p. 111) is returned.

4.2.2.2 DD_RESULT merge_list_of_surfaces (std::list< DDSurface * > & *r_list_of_surfaces*, DDVirtualSurface *& *rp_virtual_surface*, DDVirtualLoop *& *rp_border_loop*)

Virtualy merge a list of interconneted DDSurfaces together.

For DDSurfaces to be interconnected they must share common DDEdges instances. The resulting **DDVirtualSurface**(p. 111) is what the DDSurfaces would look like if the were actually merged.

Parameters:

r_list_of_surfaces The list of DDSurfaces to virtually merge.

rp_virtual_surface The new **DDVirtualSurface**(p. 111) that was created by the merge is returned here.

rp_border_loop The first **DDVirtualLoop**(p. 107) created that has a **DDEdge**(p. 55) that belongs to only one **DDLoop**(p. 89).

Return values:

If the method was successful.

DD_X_CRITICAL_ERROR If a critical error was encountered.

4.2.2.3 void merge_virtual_surface (DDVirtualSurface & *r_virtual_surface*, DDSurface *& *rp_surface*)

Create a **DDSurface**(p. 98) from a **DDVirtualSurface**(p. 111).

The new **DDSurface**(p. 98) has new DDLoops but uses the original DDEdgeUses. If a **DDEdgeUse**(p. 80) belonged to another **DDLoop**(p. 89), it is removed from the original **DDLoop**(p. 89) by this method.

Parameters:

r_virtual_surface The **DDVirtualSurface**(p. 111) to create the new **DDSurface**(p. 98) from.

rp_surface The new **DDSurface**(p. 98) created from the **DDVirtualSurface**(p. 111).

Return values:

If the method was successful.

DD_X_CRITICAL_ERROR If a critical error was encountered.

4.3 DDXOROperator Namespace Reference

This namespace contains the routines to XOR one surface with another.

Functions

- DD_RESULT **xor_surface_A_with_surface_B** (**DDSurface** ∗&rp_surface_A, **DDSurface** ∗&rp_surface_B, double tolerance, std::list< **DDSurface** ∗ > &r_difference_AB_list, std::list< **DDSurface** ∗ > &r_difference_BA_list)
- DD_RESULT **fast_xor_list_of_surfaces_A_with_list_of_surfaces_B** (std::list< **DDSurface** ∗ > &r_surfaces_A, std::list< **DDSurface** ∗ > &r_surfaces_B, double tolerance)

4.3.1 Detailed Description

This namespace contains the routines to XOR one surface with another.

4.3.2 Function Documentation

4.3.2.1 DD_RESULT fast_xor_list_of_surfaces_A_with_list_of_surfaces_B (std::list< DDSurface ∗ > & *r_surfaces_A*, std::list< DDSurface ∗ > & *r_surfaces_B*, double *tolerance*)

A special routine for calculating the XOR of two list of DDSurfaces.

There is a special condition that must be met for this routine to produce the correct result. This condition is that the DDSurfaces in one list do not overlap with any other DDSurfaces in the same list. This condition allows the routine to be faster than n^2. The results are returned in the first list.

Parameters:

r_surfaces_A The first list of DDSurfaces to be XORed. The DDSurfaces in this list cannot overlap each other. The results are returned in this list.

r_surfaces_B The second DDSurfaces to be XORed. The DDSurfaces in this list cannot overlap each other.

tolerance The tolerance of the operation.

Return values:

DD_SUCCESS The operation was a success.

DD_X_CRITICAL_ERROR The method encountered a critical error.

4.3.2.2 DD_RESULT xor_surface_A_with_surface_B (DDSurface *& *rp_surface_A*, DDSurface *& *rp_surface_B*, double *tolerance*, std::list< DDSurface * > & *r_difference_AB_list*, std::list< DDSurface * > & *r_difference_BA_list*)

XORs two DDSurfaces.

The results are stored in two seperate lists. The lists as a whole represent the complete result. The two lists can be combined by simply adding the contents of one list to the end of the other.

Parameters:

rp_surface_A The first **DDSurface**(p. 98) to be XORed.

rp_surface_B The second **DDSurface**(p. 98) to be XORed.

tolerance The tolerance of the operation.

r_difference_AB_list The variable where the DDSurfaces that are in A and not in B are returned.

r_difference_BA_list The veriable where the DDSurfaces that are in B and not in A are returned.

Return values:

DD_FAILURE The two DDSurfaces do not overlap. *rp_surface_A* and *rp_surface_B* were not altered and their pointers are still valid.

DD_SUCCESS The operation was a success. *rp_surface_A* and *rp_surface_B* were consumed in the process. However, while the two pointers will be NULL it doesn't mean that the actual instances were destroyed. They may have been used to create the new surfaces.

4.4 DDOffsetOperator Namespace Reference

This namespace contains the routines to create offset DDSurfaces.

Functions

- DD_RESULT **create_offset_surfaces** (**DDSurface** &r_surface, double tolerance, double distance, std::list< **DDSurface** ∗ > &r_surface_list)
- DD_RESULT **create_offset_surfaces** (**DDVirtualSurface** &r_surface, double tolerance, double distance, std::list< **DDSurface** ∗ > &r_surface_list, **DDVirtualLoop** ∗p_boundary_loop=0)
- DD_RESULT **create_offset_loop** (**DDEdgeUseList** &r_edgeuse_list, double tolerance, double distance, std::list< **DDLoop** ∗ > &r_similar_loops, std::list< **DDLoop** ∗ > &r_opposite_loops)
- DD_RESULT **subdivide_surfaces_at_offset** (std::list< **DDSurface** ∗ > &r_adjacent_surface_list, double tolerance, double distance, std::map< **DDSurface** ∗, std::list< **DDSurface** ∗ > ∗ > &r_offset_map, std::map< **DDSurface** ∗, std::list< **DDSurface** ∗ > ∗ > &r_original_map, std::list< **DDSurface** ∗ > &r_old_surfaces)

4.4.1 Detailed Description

This namespace contains the routines to create offset DDSurfaces.

4.4.2 Function Documentation

4.4.2.1 DD_RESULT create_offset_loop (DDEdgeUseList & *r_edgeuse_list*, double *tolerance*, double *distance*, std::list< DDLoop ∗ > & *r_similar_loops*, std::list< DDLoop ∗ > & *r_opposite_loops*)

Create an offset **DDLoop**(p. 89).

Parameters:

r_edgeuse_list The **DDLoop**(p. 89) to offset.

tolerance The tolerance of the operation.

distance The distance of the offset. A positive value will created an expanded **DDLoop**(p. 89). A negative value will create a shrunk **DDLoop**(p. 89).

r_similar_loops The list where the offset DDLoops with the same direction as *r_edgeuse_list* are returned.

r_opposite_loops The list where the offset DDLoops with the opposite direction as *r_edgeuse_-list* are returned.

Return values:

DD_SUR_LOOP_CLOSED Offsetting the **DDSurface**(p. 98) completely closed it in.

DD_SUCCESS The operation was a success.

DD_X_CRITICAL_ERROR The operation experienced a critical error.

4.4.2.2 DD_RESULT create_offset_surfaces (DDVirtualSurface & *r_surface*, double *tolerance*, double *distance*, std::list< DDSurface * > & *r_surface_list*, DDVirtualLoop * *p_boundary_loop* = 0)

Creates a **DDSurface**(p. 98) that is the offset of a **DDVirtualSurface**(p. 111).

Parameters:

r_surface The **DDVirtualSurface**(p. 111) to be offset.

tolerance The tolerance of the operation.

distance The distance of the offset. A positive value will expand the **DDVirtual-Surface**(p. 111). A negative value will shrink the **DDVirtualSurface**(p. 111).

r_surface_list Pointers to the DDSurfaces that result from the offset.

p_boundary_loop If this address is the same as the exterior loop of the virtual surface then the exterior loop is not offset.

Return values:

DD_SUR_LOOP_CLOSED The loop completely closed in.

DD_SUCCESS The operation was a success.

DD_X_CRITICAL_ERROR The operation experienced a critical error.

4.4.2.3 DD_RESULT create_offset_surfaces (DDSurface & *r_surface*, double *tolerance*, double *distance*, std::list< DDSurface * > & *r_surface_list*)

Create an offset **DDSurface**(p. 98).

Parameters:

r_surface The **DDSurface**(p. 98) to offset.

tolerance The tolerance of the operation.

distance The distance of the offset. A positive value will expand the surface. A negative value will shrink the surface.

r_surface_list Pointers to the DDSurfaces that result from the offset.

Return values:

DD_SUR_LOOP_CLOSED The surface completely closed in.

DD_SUCCESS The operation was a success.

DD_X_CRITICAL_ERROR The operation experienced a critical error.

4.4.2.4 DD_RESULT subdivide_surfaces_at_offset (std::list< DDSurface * > & *r_adjacent_surface_list*, double *tolerance*, double *distance*, std::map< DDSurface *, std::list< DDSurface * > * > & *r_offset_map*, std::map< DDSurface *, std::list< DDSurface * > * > & *r_original_map*, std::list< DDSurface * > & *r_old_surfaces*)

Subdivide a set of interconnected DDSurfaces at an offset.

The DDSurfaces are interconnected such that they share common DDEdges and DDPoints. This set of DDSurfaces are treated as one big surface. Thus the offset will be of the DDEdges that form the external boundary of the set of DDSurfaces.

Parameters:

r_adjacent_surface_list The set of interconnected DDSurfaces.

tolerance The tolerance of the operation.

distance The distance of the offset. A positive value will expand the surface. A negative value will shrink the surface.

r_offset_map Pointers to the resulting DDSurfaces that come from the offset portion of the subdivide. The key to the map is the adddress of the origianl parent **DDSurface**(p. 98). The data of the map is the resulting DDSurfaces that come from the parent.

r_original_map Pointers to the resulting DDSurfaces that come from the original portion of the subdivide. The key to the map is the adddress of the origianl parent **DDSurface**(p. 98). The data of the map is the resulting DDSurfaces that come from the parent.

r_old_surfaces The addresses to the original DDSurfaces that were consumed by the routine. These DDSurfaces need to be deleted.

Return values:

DD_SUR_LOOP_CLOSED The loop completely closed in. All of the original DDSurfaces are offset DDSurfaces. The original DDSurfaces are stored in the map as the parent and child.

DD_SUCCESS The operation was a success.

DD_X_CRITICAL_ERROR The operation experienced a critical error.

4.5 DDAnalyticalGeometryUtil Namespace Reference

The namespace perfroms analytical geometry calculations on 2d points ,vectors, lines and arcs.

Functions

- double **distance_point_point** (**DDPoint** &r_point1, **DDPoint** &r_point2)
- bool **is_within_tolerance** (**DDPoint** &r_point1, **DDPoint** &r_point2, double tolerance)
- double **calculate_area_enclosed_by_edgeuse_list** (**DDEdgeUseList** &r_list)
- DD_RESULT **is_point_in_surface** (**DDPoint** &r_point, **DDSurface** &r_surface, double tolerance)
- DD_RESULT **is_loop_inside_surface** (**DDLoop** &r_loop, **DDSurface** &r_surface, double tolerance)
- DD_RESULT **is_point_enclosed_by_loop** (**DDPoint** &r_point, **DDLoop** &r_loop, double tolerance)
- DD_RESULT **find_intersection_edge_edge** (**DDEdge** &r_edge_1, **DDEdge** &r_edge_2, double tolerance, **DDAGUIntersection** &r_intersection_1, **DDAGUIntersection** &r_intersection_2, **DDAGUIntersection** &r_intersection_3, **DDAGUIntersection** &r_intersection_4)
- DD_RESULT **find_intersection_ray_edge** (**DDPoint** &r_start_point, **DDVector** &r_direction, **DDEdge** &r_edge, double tolerance, **DDAGUIntersection** &r_intersection_1, **DDAGUIntersection** &r_intersection_2)
- DD_RESULT **is_circle_point_on_arc** (**DDPoint** &r_point, **DDArc** &r_arc, double tolerance, double &r_parametric_parameter)
- DD_RESULT **number_of_line_approximations_of_arc** (**DDArc** &r_arc, double tolerance, int &r_number_of_lines)

4.5.1 Detailed Description

The namespace perfroms analytical geometry calculations on 2d points ,vectors, lines and arcs.

4.5.2 Function Documentation

4.5.2.1 double calculate_area_enclosed_by_edgeuse_list (DDEdgeUseList & *r_list*)

Calculates the area inclosed by the list of DDEdgeUses.

The routine assumes that the list of edgeuses form a closed loop.

Returns:

Returns the area inclosed by the list of DDEdgeUses. The sign of the value indicates the direction of the ordering of the DDEdgeUses. A positive sign indicates counter clockwise direction. A negative sign indicates a clockwise direction.

4.5.2.2 double distance_point_point (DDPoint & *r_point1*, DDPoint & *r_point2*)

Calculates the distance bewteen two DDPoints.

Return values:

The distance between the *r_point1* and *r_point2*.

4.5.2.3 DD_RESULT find_intersection_edge_edge (DDEdge & *r_edge_1*, DDEdge & *r_edge_2*, double *tolerance*, DDAGUIntersection & *r_intersection_1*, DDAGUIntersection & *r_intersection_2*, DDAGUIntersection & *r_intersection_3*, DDAGUIntersection & *r_intersection_4*)

The routine finds the intersections between two DDEdges.

Parameters:

r_edge_1 The first **DDEdge**(p. 55) in the intersection.

r_edge_2 The second **DDEdge**(p. 55) in th intersection.

tolerance The tolerance used in the routine.

r_intersection_1 Used to return the first intersection found between the two DDEdges.

r_intersection_2 Used to return the second intersection found between the two DDEdges.

r_intersection_3 Used to return the second intersection found between the two DDEdges.

r_intersection_4 Used to return the second intersection found between the two DDEdges.

Return values:

DD_SUCCESS The two DDEdges intersect.

DD_FAILURE The two DDEdges do not intersect.

4.5.2.4 DD_RESULT find_intersection_ray_edge (DDPoint & *r_start_point*, DDVector & *r_direction*, DDEdge & *r_edge*, double *tolerance*, DDAGUIntersection & *r_intersection_1*, DDAGUIntersection & *r_intersection_2*)

The routine finds the intersections between a **DDEdge**(p. 55) and a ray.\.

Parameters:

r_start_point The **DDPoint**(p. 49) that defines the start of the ray.

r_direction The **DDVector**(p. 43) that defines the direction of the ray.

r_edge The **DDEdge**(p. 55) to intersect with the ray.

tolerance The tolerance used in the routine.

r_intersection_1 Used to return the first intersection found between the **DDEdge**(p. 55) and the ray.

r_intersection_2 Used to return the second intersectino found between the **DDEdge**(p. 55) and the ray.

Return values:

DD_SUCCESS The **DDEdge**(p. 55) and ray intersect.

DD_FAILURE The **DDEdge**(p. 55) and ray do not intersect.

4.5.2.5 DD_RESULT is_circle_point_on_arc (DDPoint & *r_point*, DDArc & *r_arc*, double *tolerance*, double & *r_parametric_parameter*)

This routine calculates if a **DDPoint**(p. 49) located on a parent circle lies between the start and end points of an **DDArc**(p. 70) with the same parent circle.

This routine will return incorrect results if the point is not located on the parent circle. This function does not check to see if the point lies on the parent circle.

Parameters:

r_point The **DDPoint**(p. 49) that lies on the parent circle of the **DDArc**(p. 70).

r_arc The **DDArc**(p. 70) that is checked to see if the DDPoints lies on it.

tolerance The tolerance for the routine.

r_parametric_parameter Used to return the parametric parameter that is calculated for location of the **DDPoint**(p. 49) on the **DDArc**(p. 70).

Return values:

DD_ANA_ON_START The point is within *tolerance* of the start point.

DD_ANA_ON_END The point is within *tolerance* of the end point.

DD_ANA_ON_SEGMENT The point is on the arc segment.

4.5.2.6 DD_RESULT is_loop_inside_surface (DDLoop & *r_loop*, DDSurface & *r_surface*, double *tolerance*)

Determines if a **DDLoop**(p. 89) lies within the geometry defined by a **DDSurface**(p. 98).

Return values:

DD_SUCCESS The **DDLoop**(p. 89) is in the surface.

DD_FAILURE The **DDLoop**(p. 89) is not in the surface.

DD_ANA_POINT_ON_EXTERIOR_EDGE *r_loop* is on the exterior edge of the surface.

DD_ANA_POINT_ON_INTERIOR_EDGE *r_loop* is on an interior edge of the surface.

4.5.2.7 DD_RESULT is_point_enclosed_by_loop (DDPoint & *r_point*, DDLoop & *r_loop*, double *tolerance*)

Determines if a point is within a loop.

This routine casts a ray from *r_point* in the positive X direction. It then checks the number of loop boundaries crossed. If the number of boundary crossing is odd then the point is in the loop.

Return values:

DD_SUCCESS The point is in the loop.

DD_FAILURE The point is not in the loop.

DD_ANA_POINT_ON_EXTERIOR_EDGE The point is on the edge of the loop.

4.5.2.8 DD_RESULT is_point_in_surface (DDPoint & *r_point*, DDSurface & *r_surface*, double *tolerance*)

Determines if a **DDPoint**(p. 49) lies within the geometry defined by a **DDSurface**(p. 98).

This is determined by casting a ray from *r_point* in the positive X direction. It then checks the number of surface boundaries crossed. If the number of boundary crossing is odd then the point is in the surface.

Return values:

DD_SUCCESS The point is in the surface.

DD_FAILURE The point is not in the surface.

DD_ANA_POINT_ON_EXTERIOR_EDGE *r_point* is on the exterior edge of the surface.

DD_ANA_POINT_ON_INTERIOR_EDGE *r_point* is on an interior edge of the surface.

4.5.2.9 bool is_within_tolerance (DDPoint & *r_point1*, DDPoint & *r_point2*, double *tolerance*)

Determines if the distance between two DDPoints is less than or equal to tolerance.

Parameters:

r_point1 The first **DDPoint**(p. 49) to check the distance from.

r_point2 The second **DDPoint**(p. 49) to check the distance from.

tolerance The distance to check to see if the two **DDPoint**(p. 49) are within.

Return values:

true If the two DDPoints are within tolerance.

false If the two DDPoints are not within tolerance.

4.5.2.10 DD_RESULT number_of_line_approximations_of_arc (DDArc & *r_arc*, double *tolerance*, int & *r_number_of_lines*)

This routine calculates the number of line segments to use to approximate **DDArc**(p. 70) with in tolerance.

Parameters:

r_arc The **DDArc**(p. 70) to approximate.

tolerance The maximum distance between each line segment and the arc.

r_number_of_lines Used to return the calculated number of line segments needed to approximate the arc with in tolerance.

4.6 DDModifyGeometryTool Namespace Reference

Provides functionality to modify the 2d geometry of points, lines, arcs, edgeuses, loops and surfaces.

Functions

- void **remove_small_edges** (**DDLoop** ∗p_loop, double tolerance, **DDEdgeUseList** &r_removed_edgeuses)
- void **remove_small_edges** (**DDSurface** &r_surface, double tolerance, **DDEdgeUseList** &r_removed_edgeuses)
- DD_RESULT **intersect_surface_surface** (**DDSurface** &r_surface_A, **DDSurface** &r_surface_B, double tolerance, std::map< **DDPoint** ∗, unsigned long > &r_point_types)
- DD_RESULT **intersect_loop_with_self** (**DDLoop** &r_loop, double tolerance, std::map< **DDPoint** ∗, unsigned long > &r_point_types)
- DD_RESULT **replace_arc_with_line_if_within_tolerance** (**DDArc** ∗&rp_arc_in, double tolerance, **DDLine** ∗&rp_line_out)
- void **create_surface_from_bounding_box** (**DDBoundingBox** &r_bounding_box, **DDSurface** ∗&rp_surface)
- DD_RESULT **find_inside_edgeuses** (**DDSurface** &r_surface_A, **DDSurface** &r_surface_B, std::map< **DDPoint** ∗, unsigned long > &r_point_types, unsigned int overlapping_index, unsigned int opposite_overlapping_index, double tolerance, unsigned int inside_index)
- DD_RESULT **remove_interior_loops** (**DDSurface** &r_surface, double tolerance, **DDVirtualLoop** ∗&rp_virtual_loop)
- DD_RESULT **subdivide_to_remove_interior_loops** (**DDSurface** ∗&rp_surface, double tolerance, std::list< **DDSurface** ∗ > &r_surface_list)
- DD_RESULT **copy_list_of_surfaces** (std::list< **DDSurface** ∗ > &r_surface_list, std::list< **DDSurface** ∗ > &r_new_surface_list)
- void **copy_surface** (**DDSurface** &r_surface, **DDSurface** ∗&rp_surface)
- DD_RESULT **reverse_loop** (**DDLoop** &r_loop)
- DD_RESULT **split_self_intersecting_loop** (**DDLoop** &r_loop, std::list< **DDLoop** ∗ > &r_result_loop)
- void **create_line_approximation_for_arc** (**DDArc** &r_arc, DDEdgeUseSense sense, double tolerance, std::list< **DDEdge** ∗ > &r_edge_list)
- DD_RESULT **replace_point_A_with_B** (**DDPoint** &r_point_A, **DDPoint** &r_point_B, double tolerance)
- DD_RESULT **split_edge_use_at_point** (**DDEdgeUse** &r_edge_use_in, **DDPoint** &r_point, double tolerance, **DDEdge** ∗&rp_old_edge_in, **DDEdgeUse** ∗&rp_new_edge_use_out)
- void **find_edgeuses_with_no_neighbor** (**DDSurface** &r_surface, **DDEdgeUseList** &r_edgeuse_list)

4.6.1 Detailed Description

Provides functionality to modify the 2d geometry of points, lines, arcs, edgeuses, loops and surfaces.

Author:
Corey McBride

Date:
03/17/03

4.6.2 Function Documentation

4.6.2.1 DD_RESULT copy_list_of_surfaces (std::list< DDSurface * > & *r_surface_list*, std::list< DDSurface * > & *r_new_surface_list*)

Creates new instances of **DDSurface**(p. 98) that are copies of the original DDSurfaces.

The new DDSurfaces contain new instances of the primitives that make up the DDSurfaces.

Parameters:

r_surface_list The list of pointer to the original DDSurfaces to copy.

r_new_surface_list The list of pointers to the new DDSurfaces.

Return values:

DD_SUCCESS If the routine was successful.

DD_FAILURE If the routine was not successful.

4.6.2.2 void copy_surface (DDSurface & *r_surface*, DDSurface *& *rp_surface*)

Creates new a instance of **DDSurface**(p. 98) that is a copy of the original **DDSurface**(p. 98).

The new **DDSurface**(p. 98) contain new instances of the primitives that make up the **DDSurface**(p. 98).

Parameters:

r_surface The original **DDSurface**(p. 98) to copy.

rp_surface The pointer to the new **DDSurface**(p. 98).

4.6.2.3 void create_line_approximation_for_arc (DDArc & *r_arc*, DDEdgeUseSense *sense*, double *tolerance*, std::list< DDEdge * > & *r_edge_list*)

Creats a sequence of DDLines that approximate a **DDArc**(p. 70) within tolerance.

The DDLines created share instances of **DDPoint**(p. 49) as end points.

Parameters:

r_arc The **DDArc**(p. 70) to approximate.

sense The sense in which the **DDLine**(p. 65) sequence should be created.

tolerance The tolerance of the routine.

r_edge_list The list of pointers to the new DDLines.

4.6.2.4 void create_surface_from_bounding_box (DDBoundingBox & *r_bounding_box*, DDSurface *& *rp_surface*)

Creates a **DDSurface**(p. 98) from the dimensions contains in the bounding box.

Parameters:

r_bounding_box The **DDBoundingBox**(p. 118) that contains the dimensions used to create the **DDSurface**(p. 98)

rp_surface The pointer variable where the new **DDSurface**(p. 98) is returned.

4.6.2.5 void find_edgeuses_with_no_neighbor (DDSurface & *r_surface*, DDEdgeUseList & *r_edgeuse_list*)

Finds edgeuses from the surface that do not have a neighboring edgeuse.

Parameters:

r_surface The **DDSurface**(p. 98) to find the edgeuses from.

r_edgeuse_list The **DDEdgeUseList**(p. 122) where the found edgeuses are returned.

4.6.2.6 DD_RESULT find_inside_edgeuses (DDSurface & *r_surface_A*, DDSurface & *r_surface_B*, std::map< DDPoint *, unsigned long > & *r_point_types*, unsigned int *overlapping_index*, unsigned int *opposite_overlapping_index*, double *tolerance*, unsigned int *inside_index*)

Finds all DDEdgeUses of *r_surface_A* that are inside r_surface_B.

The two DDSurfaces must previously have been intersected so that they share common DDPoints. See the method *intersect_surface_surface*.

Parameters:

r_surface_A The **DDSurface**(p. 98) from which the DDEdgeUses are examined to determine if they are inside *r_surface_B*.

r_surface_B The **DDSurface**(p. 98) which the DDEdgeUses from *r_surface_A* are compared against.

r_point_types The object where the intersections between the two DDSurfaces are stored.

overlapping_index The bit flag index of the DDEdgeUses that marka that it is an overlapping DDEdgeUses.

opposite_overlapping_index The bit flag index of the DDEdgeUses that marks that it is an opposite overlapping DDEdgeUses.

tolerance The tolerance used during the method.

inside_index The bit flag index used to mark that a **DDEdgeUse**(p. 80) from *r_surface_A* is inside the area outlined by *r_surface_B*.

Return values:

DD_SUCCESS If the method complete successfully.

DD_FAILURE If the method wasn't able to determine the status of all DDEdgeUses. The most likely cause of this is that one or more **DDEdgeUse**(p. 80) from *r_surface_A* overlaps DDEdgeUses from *r_surface_B* and the DDEdgeUses were not marked as overlapping.

DD_X_CRITICAL_ERROR The method experienced a critical error and wasn't able to continue.

4.6.2.7 DD_RESULT intersect_loop_with_self (DDLoop & *r_loop*, double *tolerance*, std::map< DDPoint *, unsigned long > & *r_point_types*)

Intersects a **DDLoop**(p. 89) with itself. This routine compares each **DDEdge**(p. 55) in a **DDLoop**(p. 89) with every other **DDEdge**(p. 55) in the same **DDLoop**(p. 89). DDPoints that are already shared as end points between neighboring DDEdges are ignored. The intersections are then implemented by merging overlapping DDPoints and spliting DDEdges and DDEdgeUses at intersection points. The end result is that the DDEdges in the **DDLoop**(p. 89) will share common DDPoints at each intersection.

Parameters:

r_loop The **DDLoop**(p. 89) to self intersect.

tolerance The tolerance to maintain during the operation. DDPoints that are within tolerance are merged and DDEdges that are within tolerance are intersected.

r_point_types A object that contains all of the intersection points and their types. Possible types include DD_SUR_TANGENT_POINT and DD_SUR_CROSSOVER_POINT.

Return values:

DD_SUCCESS If the routine was successful.

DD_FAILURE If the **DDLoop**(p. 89) does not self intersect.

DD_X_CRITICAL_ERROR If a critical error was encountered.

4.6.2.8 DD_RESULT intersect_surface_surface (DDSurface & *r_surface_A*, DDSurface & *r_surface_B*, double *tolerance*, std::map< DDPoint *, unsigned long > & *r_point_types*)

Intersects two DDSurfaces.

This method finds all intersections between the DDEdges that make up the two DDSurfaces. The intersections are then implemented by merging overlapping DDPoints and spliting DDEdges and DDEdgeUses at intersection points. The end result is that the two DDSurfaces will share common DDPoints at each intersection. DDEdges that have a length less than tolerance are also removed.

Parameters:

r_surface_A One of the DDSurfaces to intersect.

r_surface_B One of the **DDSurface**(p. 98) to intersect.

tolerance The tolerance to maintain during the operation. DDPoints that are within tolerance are merged and DDEdges that are within tolerance are intersected.

r_point_types A object that contains all of the intersection points and their types. Possible types include DD_SUR_TANGENT_POINT and DD_SUR_CROSSOVER_POINT.

Return values:

DD_SUCCESS If the routine was successful.

DD_FAILURE If the two DDSurfaces do not intersect.

DD_X_CRITICAL_ERROR If a critical error was encountered.

4.6.2.9 DD_RESULT remove_interior_loops (DDSurface & *r_surface*, double *tolerance*, DDVirtualLoop *& *rp_virtual_loop*)

Removes any interior DDLoops by creating runners that connect the interior DDLoops to the exterior **DDLoop**(p. 89). A **DDVirtualLoop**(p. 107) is created that represents the exterior **DDLoop**(p. 89). New **DDEdge**(p. 55) and **DDEdgeUse**(p. 80) are created as runners and inserted into the **DDVirtualLoop**(p. 107) that represents the exterior **DDLoop**(p. 89). Thus the orignal **DDSurface**(p. 98) is not changed in any way.

Parameters:

r_surface The **DDSurface**(p. 98) that is to have the interior DDLoops removed.

tolerance The tolerance used to remove the interior DDLoops.

rp_virtual_loop The **DDVirtualLoop**(p. 107) pointer that is used to return the new **DDVirtualLoop**(p. 107).

Return values:

DD_FAILURE If there are no interior loops.

DD_SUCCESS If the method was successful.

4.6.2.10 void remove_small_edges (DDSurface & *r_surface*, double *tolerance*, DDEdgeUseList & *r_removed_edgeuses*)

Removes all of the DDEdges from a **DDSurface**(p. 98) that are less that *tolerance*.

This method checks the length all of the DDEdges in *r_surface*. If the length of a DDEdges is below *tolerance* than the DDEdges is removed from the **DDSurface**(p. 98). This method call remove_small_edges on each **DDLoop**(p. 89) used by the **DDSurface**(p. 98).

Parameters:

r_surface The **DDSurface**(p. 98) to remove the small **DDEdge**(p. 55) from.

tolerance The min length of an **DDEdge**(p. 55) to keep.

r_removed_edgeuses A variable to store all of the removed small DDEdges.

4.6.2.11 void remove_small_edges (DDLoop * *p_loop*, double *tolerance*, DDEdgeUseList & *r_removed_edgeuses*)

Removes all of the DDEdges from a **DDLoop**(p. 89) that are less that *tolerance*.

This function checks the length all of the DDEdges in *p_loop*. If the length of a **DDEdge**(p. 55) is less than *tolerance* than the DDEdges is removed from the **DDLoop**(p. 89).

Parameters:

p_loop The **DDLoop**(p. 89) to remove the small DDEdges from.

tolerance The min length of the **DDEdge**(p. 55) keep.

r_removed_edgeuses A variable to store all of the removed small DDEdges.

4.6.2.12 DD_RESULT replace_arc_with_line_if_within_tolerance (DDArc *& *rp_arc_in*, double *tolerance*, DDLine *& *rp_line_out*)

Replaces a **DDArc**(p. 70) with a **DDLine**(p. 65) if the arc is within tolerance on a line. If the maximum distance between the arc and a line is less than tolerance then the **DDArc**(p. 70) is replaced with a **DDLine**(p. 65).

Parameters:

rp_arc_in The **DDArc**(p. 70) to examine.

tolerance The tolerance to use for the comparison.

rp_line_out The pointer to the new **DDLine**(p. 65) if the **DDArc**(p. 70) was replaced.

Return values:

DD_SUCCESS If the **DDArc**(p. 70) was replaced with a **DDLine**(p. 65).

DD_FAILURE If the **DDArc**(p. 70) was not replaced with a **DDLine**(p. 65).

4.6.2.13 DD_RESULT replace_point_A_with_B (DDPoint & *r_point_A*, DDPoint & *r_point_B*, double *tolerance*)

Replaces **DDPoint**(p. 49) A with **DDPoint**(p. 49) B.

DDPoint(p. 49) A is replaced by **DDPoint**(p. 49) B for all of the DDEdges that are registered with **DDPoint**(p. 49) A. If the **DDEdge**(p. 55) is a **DDArc**(p. 70) then the arc is checked to see if it is within tolerance of a line. If the arc is within tolerance of a line than the **DDArc**(p. 70) is replaced by a **DDLine**(p. 65).

Parameters:

r_point_A The **DDPoint**(p. 49) to be replaced.

r_point_B The **DDPoint**(p. 49) to replace the original.

tolerance The tolerance to compare the arcs to a line.

Return values:

DD_SUCCESS If the routine was successful.

DD_X_CRITICAL_ERROR If the routine experienced an error.

4.6.2.14 DD_RESULT reverse_loop (DDLoop & *r_loop*)

Reverses the direction of a **DDLoop**(p. 89).

This routine reverses the direction of a **DDLoop**(p. 89) by switching the sense that each **DDEdgeUse**(p. 80) uses its **DDEdge**(p. 55). Then the order that the DDEdgeUses appear in the **DDLoop**(p. 89) is reversed. This routine will only work if each **DDEdge**(p. 55) is only used by one **DDEdgeUse**(p. 80).

Parameters:

r_loop The **DDLoop**(p. 89) to reverse.

Return values:

DD_SUCCESS If the **DDLoop**(p. 89) was reversed.

DD_X_CRITICAL_ERROR If the routine experienced an error.

4.6.2.15 DD_RESULT split_edge_use_at_point (DDEdgeUse & *r_edge_use_in*, DDPoint & *r_point*, double *tolerance*, DDEdge ∗& *rp_old_edge_in*, DDEdgeUse ∗& *rp_new_edge_use_out*)

Splits **DDEdgeUse**(p. 80) at the indicated coordinates.

This routine splits the original **DDEdge**(p. 55) and creates a new **DDEdgeUse**(p. 80) for the new **DDEdge**(p. 55). The provided **DDPoint**(p. 49) is used as a new end point for both the original and new **DDEdge**(p. 55). If the original **DDEdgeUse**(p. 80) was part of a **DDLoop**(p. 89) then the new **DDEdgeUse**(p. 80) is inserted into the correct location in the **DDLoop**(p. 89). If the original **DDEdge**(p. 55) was used by another **DDEdgeUse**(p. 80) and cooresponding neighboring **DDLoop**(p. 89) then a new **DDEdgeUse**(p. 80) for the neighboring **DDLoop**(p. 89) is created and inserted into the correct location. If the **DDEdge**(p. 55) is a **DDArc**(p. 70) and is within tolerance of a line, then the **DDArc**(p. 70) is replaced with a **DDLine**(p. 65).

Parameters:

r_point The **DDPoint**(p. 49) used to split the **DDEdgeUse**(p. 80).

rp_old_edge_in The pointer where the old **DDArc**(p. 70) is returned if it was replaced by a line.

rp_edge_use_out The pointer to return the new **DDEdgeUse**(p. 80).

Return values:

DD_SUCCESS If the routine was successful.

DD_X_CRITICAL_ERROR If the routine experienced an error.

4.6.2.16 DD_RESULT split_self_intersecting_loop (DDLoop & *r_loop*, std::list< DDLoop ∗ > & *r_result_loop*)

Splits a self intersection **DDLoop**(p. 89) into multiple DDLoops. This routine is designed to handle self intersecting loops where loops are chained together with tangent points. If a loop is nested

inside another loop with multiple tangent points then the result may not be correct. The routine checks for self intersections by looking for the number of times each **DDPoint**(p. 49) appears in the **DDLoop**(p. 89). If a **DDPoint**(p. 49) appears more than once then there is a self intersection. For this routine to work correctly all self intersections need to have be found and implemented previously. See the intersect_loop_with_self method for a self intersection routine.

Parameters:

r_loop The **DDLoop**(p. 89) to split.

r_result_loop The list of new DDLoops that were created from spliting the original **DDLoop**(p. 89).

Return values:

DD_SUCCESS If the routine was successful;

DD_X_CRITICAL_ERROR If the routine experienced an error.

4.6.2.17 DD_RESULT subdivide_to_remove_interior_loops (DDSurface *& *rp_surface*, double *tolerance*, std::list< DDSurface * > & *r_surface_list*)

Removes any internal DDLoops by subdividing the original **DDSurface**(p. 98) into multiple DDSurfaces without interior DDLoops.

Parameters:

rp_surface The **DDSurface**(p. 98) that is to be subdivided.

tolerance The tolerance used to remove the interior DDLoops.

r_surface_list The list of DDSurfaces that result from subdividing the original **DDSurface**(p. 98).

Return values:

DD_SUCCESS The interior DDLoops were removed.

DD_FAILURE If there were some problems during the subdivision. The list may contain some **DDSurface**(p. 98) with interior DDLoops.

Chapter 5

Interface Routines

This chapter describes the major interface routines. The routines are grouped in a namespace according to functionality. Each section describes a namespace and gives a list of its routines. A description of each routine is then provided.

5.1 DDInterface Namespace Reference

The DDInterface contains a set of routines to interface with the 2d geometry library.

Functions

- DD_RESULT **calculate_bounding_box_for_list_of_surfaces** (std::list< **DDSurface** * > &r_surface_list, **DDBoundingBox** &r_bounding_box)
- DD_RESULT **write_surface_to_file_dxf_format** (**DDSurface** &r_surface, double tolerance, std::string &r_file_name, std::string &r_layer_name)
- DD_RESULT **write_list_of_surfaces_to_file_dxf_format** (std::list< **DDSurface** * > &r_surface_list, double tolerance, std::string &r_file_name, std::string &r_layer_name)
- DD_RESULT **write_list_of_surfaces_to_file_dxf_format_through_subdivision** (std::list< **DDSurface** * > &r_surface_list, double tolerance, std::string &r_file_name, std::string &r_layer_name)
- DD_RESULT **open_dxf_file_for_writing** (std::string &r_filename, std::ofstream &r_dxf_file)
- DD_RESULT **write_surface_to_file_dxf_format** (**DDSurface** &r_surface, double tolerance, std::ofstream &r_dxf_file, std::string &r_layer_name)
- DD_RESULT **write_list_of_surfaces_to_file_dxf_format_through_subdivision** (std::list< **DDSurface** * > &r_surface_list, double tolerance, std::ofstream &r_dxf_file, std::string &r_layer_name)
- DD_RESULT **write_list_of_surfaces_to_file_dxf_format** (std::list< **DDSurface** * > &r_surface_list, double tolerance, std::ofstream &r_dxf_file, std::string &r_layer_name)
- DD_RESULT **close_dxf_file** (std::ofstream &r_dxf_file)
- void **write_edgeuse_list_to_file_dxf_polyline** (**DDEdgeUseList** &r_edgeuse_list, std::ofstream &r_dxf_file, std::string &r_layer_name)
- std::string **get_version** ()

5.1.1 Detailed Description

The DDInterface contains a set of routines to interface with the 2d geometry library.

Author:
Corey McBride

Date:
02/11/03

5.1.2 Function Documentation

5.1.2.1 DD_RESULT calculate_bounding_box_for_list_of_surfaces (std::list< DDSurface * > & *r_surface_list*, DDBoundingBox & *r_bounding_box*)

Calculates the bounding box for a list of surfaces.

The bounding box is calculated by calling the calculate_bounding_box function on each **DDEdgeUse**(p. 80) of each exterior loop for every surface in the list.

5.1.2.2 DD_RESULT close_dxf_file (std::ofstream & *r_dxf_file*)

Closes an open Drawing File (dxf).

This routine writes all of the required footer information and closes the file.

Parameters:

r_dxf_file The dxf file to close.

Return values:

DD_SUCCESS If the file was closed successfully.

DD_FAILURE If the file was not closed successfully.

5.1.2.3 std::string get_version ()

Returns the 2DBoolean Library's current version.

Returns:

A std::string that contains the version information.

5.1.2.4 DD_RESULT open_dxf_file_for_writing (std::string & *r_filename*, std::ofstream & *r_dxf_file*)

Opens Drawing File (dxf) for writing.

This routine opens a file with the name provided and writes the required header information.

Parameters:

r_filename The name of the file to be opened.

r_dxf_file The file object.

Return values:

DD_SUCCESS If the file was opened successfully.

DD_X_FILE_IO_ERROR If the file was not opened successfully.

5.1.2.5 void write_edgeuse_list_to_file_dxf_polyline (DDEdgeUseList & *r_edgeuse_list*, std::ofstream & *r_dxf_file*, std::string & *r_layer_name*)

Writes a list of edgeuses to a file as a Drawing File Format (dxf) wlpolyline.

Parameters:

r_edgeuse_list The list of DDEdgeUses to be written to file.

r_dxf_file The open file were the polyline is to be written.

r_layer_name The name of the layer to place the polyline.

5.1.2.6 DD_RESULT write_list_of_surfaces_to_file_dxf_format (std::list< DDSurface * > & *r_surface_list*, double *tolerance*, std::ofstream & *r_dxf_file*, std::string & *r_layer_name*)

Writes a list of surfaces to an open dxf file.

The surfaces are written as POLYLINEs with all of the interior loops of a surface merged with its exterior loop.

Parameters:

r_surface_list The surfaces to export to the file.

tolerance The tolerance that is used to merge the interior loops with the exterior loop.

r_dxf_file The open dxf file.

r_layer_name The layer to write the surfaces to.

Return values:

DD_SUCCESS If the write was successful.

DD_FAILURE If the write was not successful.

5.1.2.7 DD_RESULT write_list_of_surfaces_to_file_dxf_format (std::list< DDSurface * > & *r_surface_list*, double *tolerance*, std::string & *r_file_name*, std::string & *r_layer_name*)

Writes a list of surfaces to a file in the Drawing File (dxf) format.

Parameters:

r_surface_list The list of surfaces to be written to the file.

tolerance The tolerance that is used to merge the interior loops with the exterior loop.

r_file_name The name of the dxf file to be written.

r_layer_name The name of the layer to place the surfaces.

5.1.2.8 DD_RESULT write_list_of_surfaces_to_file_dxf_format_through_subdivision (std::list< DDSurface * > & *r_surface_list*, double *tolerance*, std::ofstream & *r_dxf_file*, std::string & *r_layer_name*)

Writes multiple surfaces to an open dxf file.

The surface is written as multiple POLYLINEs with all of the interior loops removed through subdivision.

Parameters:

r_surface_list The surfaces to export to the file.

tolerance The tolerance that is used to subdivide the surface.

r_dxf_file The open dxf file.

r_layer_name The layer to write the surface to.

Return values:

DD_SUCCESS If the write was successful.

DD_FAILURE If the write was not successful.

5.1.2.9 DD_RESULT write_list_of_surfaces_to_file_dxf_format_through_subdivision (std::list< DDSurface * > & *r_surface_list*, double *tolerance*, std::string & *r_file_name*, std::string & *r_layer_name*)

Writes a list of surfaces to a file in the Drawing File (dxf) format. The surfaces are written as multiple POLYLINEs with all of the interior loops removed through subdivision.

Parameters:

r_surface_list The list of surfaces to be written to the file.

tolerance The tolerance that is used to subdivide the surfaces.

r_file_name The name of the dxf file to be written.

r_layer_name The layer to write the surfaces to.

Return values:

DD_SUCCESS If the write was successful.

DD_FAILURE If the write was not successful.

5.1.2.10 DD_RESULT write_surface_to_file_dxf_format (DDSurface & *r_surface*, double *tolerance*, std::ofstream & *r_dxf_file*, std::string & *r_layer_name*)

Writes a surface to an open dxf file.

The surface is written as a POLYLINE with all of the interior loops merged with the exterior loop.

Parameters:

r_surface The surface to export to the file.

tolerance The tolerance that is used to merge the interior loops with the exterior loop.

r_dxf_file The open dxf file.

r_layer_name The layer to write the surface to.

Return values:

DD_SUCCESS If the write was successful.

DD_FAILURE If the write was not successful.

5.1.2.11 DD_RESULT write_surface_to_file_dxf_format (DDSurface & *r_surface*, double *tolerance*, std::string & *r_file_name*, std::string & *r_layer_name*)

Writes a surface to a file in the Drawing File (dxf) format.

Parameters:

r_surface The surface to be written to the file.

tolerance The tolerance that is used to merge the interior loops with the exterior loop.

r_file_name The name of the dxf file to be written.

r_layer_name The name of the layer to place the surface.

5.2 DDMEMMaskReader Class Reference

Public interface to mask reading and processing.

`#include <DDMEMMaskReader.hpp>`

Public Member Functions

- **DDMEMMaskReader** ()
- ∼**DDMEMMaskReader** ()
- void **create_surfaces_from_mask_file** (ifstream &r_file_in, **DDMaskDefinitions** &r_mask_-definitions)
- DD_RESULT **create_mask_definitions** (ifstream &r_file_in, double tolerance, **DDMask-Definitions** &r_mask_definitions)
- DD_RESULT **create_mask_definitions** (ifstream &r_file_in, double tolerance, std::list< std::string > &r_layers, **DDMaskDefinitions** &r_mask_definitions)
- void **SubDivideList** (**DDMEMPolygonList** ∗list, int xInterval, int yInterval, double offset, std::vector< **DDMEMPolygonList** ∗ > &new_lists)

5.2.1 Detailed Description

Public interface to mask reading and processing.

Author:
Steven Plimpton
Corey McBride

Date:
02/26/03

5.2.2 Constructor & Destructor Documentation

5.2.2.1 DDMEMMaskReader::DDMEMMaskReader ()

Constructor.

5.2.2.2 DDMEMMaskReader::∼DDMEMMaskReader ()

Destructor.

5.2.3 Member Function Documentation

5.2.3.1 DD_RESULT DDMEMMaskReader::create_mask_definitions (ifstream & *r_file_in*, double *tolerance*, std::list< std::string > & *r_layers*, DDMaskDefinitions & *r_mask_definitions*)

Creates DDSurfaces based upon the polygons in a .mem mask file.

The DDSurfaces are stored in an instance of **DDMaskDefinitions**(p. 115). The surfaces of each layer are unioned(OR) together.

Parameters:

r_file_in An open .mem file.

tolerance The tolerance used during the routine.

r_layers The list of layers in the .mem file.

r_mask_definitions A **DDMaskDefinitions**(p. 115) object used to store the mask set.

5.2.3.2 DD_RESULT DDMEMMaskReader::create_mask_definitions (ifstream & *r_file_in*, double *tolerance*, DDMaskDefinitions & *r_mask_definitions*)

Creates DDSurfaces based upon the polygons in a .mem mask file.

The DDSurfaces are stored in an instance of **DDMaskDefinitions**(p. 115). The surfaces of each layer are unioned(OR) together.

Parameters:

r_file_in An open .mem file.

tolerance The tolerance used during the routine.

r_mask_definitions A **DDMaskDefinitions**(p. 115) object used to store the mask set.

5.2.3.3 void DDMEMMaskReader::create_surfaces_from_mask_file (ifstream & *r_file_in*, DDMaskDefinitions & *r_mask_definitions*)

Creates DDSurfaces based upon the polygons in a .mem mask file.

The DDSurfaces are stored in an instance of **DDMaskDefinitions**(p. 115). The surfaces on each layer are not unioned(OR) together.

Parameters:

r_file_in An open .mem file.

r_mask_definitions A **DDMaskDefinitions**(p. 115) object used to store the mask set.

5.2.3.4 void DDMEMMaskReader::SubDivideList (DDMEMPolygonList $*$ *list*, int *xInterval*, int *yInterval*, double *offset*, std::vector$<$ DDMEMPolygonList $*$ $>$ & *new_lists*)

A macro to define the memory allocation functions.

References

[1] C. L. McBride, V. Yarberry, and R. C. Schmidt. SummitView 1.0: A Code to Automatically Generate 3D Solid Models of Surface Micro-machining based MEMS Designs. Technical report SAND2006-6828, Sandia National Laboratories, P.O. Box 5800, Albuquerque, New Mexico 87185, 2006.

[2] mems.sandia.gov/samples/doc/MEM_File_Format_Documentation.txt.

[3] M. S. Rodgers and J. J. Sniegowski. ”Designing Microelectromechanical Systems-on-a-Chip in a 5-Level Surface Micromachine Technology”. In *2nd International Conference on Engineering Design and Automation*, 1998.

[4] V. R. Yarberry. ”Meeting the MEMS ’design-to-analysis’ challenge: the SUMMiT V design tool environment”. In *Proceedings of the ASME International Mechanical Engineering Congress*, New Orleans, LA, November 17-22 2002.

[5] V. Yarberry and C. Jorgensen. A 2D visualization tool for SUMMiT V design. Technical report SAND2000-3126, Sandia National Laboratories, P.O. Box 5800, Albuquerque, New Mexico 87185, 2000.

[6] V. R. Yarberry. ”A 2D visualization tool for SUMMiT V designs”. In *Proceedings of the Fourth International Conference on Modeling and Simulation of Microsystems*, pages 606–609, Hilton Head Island, CS, March 19-21 2001.

[7] J. Allen. ”MEMS design rule checking: A batch approach for remote operation”. In *Proceedings of the SPIE 5th Annual International Symposium on Smart Electronics and MEMS*, volume 3328, pages 32–39, San Diego, CA, March 1-5 1998.

[8] C. R. Jorgensen and V. Yarberry. ”A 3D geometry modeler for the SUMMiT V MEMS designer”. In *Technical Proceedings of the 2001 International Conference on Modeling and Simulation of Microsystems*, volume 1, pages 594 – 597, 2001.

[9] Spatial Corp. ACIS R12 Online Help. Library documentation, Spatial Corp., A Dassault Systemes company, 10955 Westmoor Drive, Suite 425 Westminster, Colorado 80021, 2003.

[10] www.stlport.org.

Appendix 1: Example Code that uses GBL-2D

Example 1

The example C++ code listed here creates two surfaces, a square and a circle. The square is imprinted with the circle, and the results are written to s DXF file with multiple layers.

```
#include <string>
#include <fstream>
#include "DDSurface.hpp"
#include "DDPoint.hpp"
#include "DDLine.hpp"
#include "DDArc.hpp"
#include "DDEdgeUse.hpp"
#include "DDLoop.hpp"
#include "DDImprintOperator.hpp"
#include "DDModifyGeometryTool.hpp"
#include "DDinterface.hpp"

int main(int argc, char* argv[])
{
  // This example create two surfaces.
  // The first surface is a square. The second surface is a circle.
  // The square is imprinted with the circle.
  // The results are written to a dxf file with multiple layers.

  // Create the square surface.
  // Create Points
  DDPoint* point1=new DDPoint();
  DDPoint* point2=new DDPoint();
  DDPoint* point3=new DDPoint();
  DDPoint* point4=new DDPoint();

  point1->mX=0.0;
  point1->mY=0.0;

  point2->mX=1.0;
  point2->mY=0.0;

  point3->mX=1.0;
  point3->mY=1.0;

  point4->mX=0.0;
  point4->mY=1.0;

  // Create Lines based on points
  DDLine* line1= new DDLine(point1,point2);
  DDLine* line2= new DDLine(point2,point3);
  DDLine* line3= new DDLine(point3,point4);
  DDLine* line4= new DDLine(point4,point1);

  // Create Edgeuses based on lines.
```

```
DDEdgeUse *edgeuse1= new DDEdgeUse(line1,DDForwardSense);
DDEdgeUse *edgeuse2= new DDEdgeUse(line2,DDForwardSense);
DDEdgeUse *edgeuse3= new DDEdgeUse(line3,DDForwardSense);
DDEdgeUse *edgeuse4= new DDEdgeUse(line4,DDForwardSense);

// Create Loop
DDLoop* loop1= new DDLoop();

// Add edgeuses to loop.
// Edges that are next to each other in loop must share common end points
loop1->push_back(edgeuse1);
loop1->push_back(edgeuse2);
loop1->push_back(edgeuse3);
loop1->push_back(edgeuse4);

// Create a surface with an exterior loop.
DDSurface* squareSurface= new DDSurface(loop1);

// Create the circle surface.
// Create points.
DDPoint* point5= new DDPoint();
DDPoint* point6= new DDPoint();
DDPoint* point7= new DDPoint();
DDPoint* point8= new DDPoint();

point5->mX=0.25;
point5->mY=0.25;

point6->mX=0.75;
point6->mY=0.75;

point7->mX=0.5;
point7->mY=0.5;

point8->mX=0.5;
point8->mY=0.5;

// Create arcs.
// Each arc must have a unique instance of the center point.
DDArc *arc1= new DDArc(point5,point6,point7);
DDArc *arc2= new DDArc(point6,point5,point8);

// Create Edgeuses based on the arcs.
DDEdgeUse *edgeuse5= new DDEdgeUse(arc1,DDForwardSense);
DDEdgeUse *edgeuse6= new DDEdgeUse(arc2,DDForwardSense);

// Add edgeuses to loop.
// Edges that are next to each other in loop must share the same instance of the common end point
DDLoop* loop2= new DDLoop();
loop2->push_back(edgeuse5);
loop2->push_back(edgeuse6);

// Create a surface with an exterior loop.
DDSurface* circleSurface= new DDSurface(loop2);
```

```
// Create surface list to store the imprint results.
std::list<DDSurface*> intersection_surfaces;
std::list<DDSurface*> difference_surfaces_from_A;
std::list<DDSurface*> difference_surfaces_from_B;
std::list<DDSurface*> old_surfaces;

// Imprint the square surface with the circle surface.
// The results are stored in the three lists.
// The intersection of the two surfaces is stored in the first list "intersection_surfaces".
// Those areas that are from the first surface but are not covered by the second surface
// are stored in the second list "difference_surfaces_from_A".
// Those areas that are from the second surface but are not covered by the first surface
// are stored in the third list "difference_surfaces_from_B".
// The original surfaces are returned in the last list "old_surfaces" These surfaces are not
// valid and need to be deleted.
DDImprintOperator::subdivide_surface_A_with_imprint_from_surface_B(squareSurface,
                   circleSurface, .001, intersection_surfaces, difference_surfaces_from_A,
                   difference_surfaces_from_B, old_surfaces);

// Now we write the results to a DXF format file.
// The results are stored on different layers.
std::ofstream out_file;
std::string output_filename="output.dxf";
// Open the file.
// This routine writes out the necessary header information.
DDInterface::open_dxf_file_for_writing(output_filename,out_file);

// Write out each set of surfaces as a separate layer.
std::string layer_name="Intersection Surfaces";
DDInterface::write_list_of_surfaces_to_file_dxf_format(intersection_surfaces,
             .001, out_file, layer_name);

layer_name="Difference Surfaces From A";
DDInterface::write_list_of_surfaces_to_file_dxf_format(difference_surfaces_from_A,
             .001, out_file, layer_name);

// For this example there are not surfaces in this list.
// These lines are presented for completeness.
layer_name="Difference Surfaces From B";
DDInterface::write_list_of_surfaces_to_file_dxf_format(difference_surfaces_from_B,
             .001, out_file, layer_name);

// Close the file.
DDInterface::close_dxf_file(out_file);


// Delete the surfaces and all of the primitives that make up the surface.
while(!intersection_surfaces.empty())
{
  DDSurface *temp=intersection_surfaces.front();
  intersection_surfaces.pop_front();
  DDSurface::delete_surface(temp);
}
```

```
  while(!difference_surfaces_from_A.empty())
  {
    DDSurface *temp=difference_surfaces_from_A.front();
    difference_surfaces_from_A.pop_front();
    DDSurface::delete_surface(temp);
  }
  while(!difference_surfaces_from_B.empty())
  {
    DDSurface *temp=difference_surfaces_from_B.front();
    difference_surfaces_from_B.pop_front();
    DDSurface::delete_surface(temp);
  }

  // Delete the original surface.
  // These surfaces are not valid surfaces shouldn't be used for anything.
  // They need to be deleted.
  while(!old_surfaces.empty())
  {
    DDSurface *temp=old_surfaces.front();
    old_surfaces.pop_front();
    delete temp;
  }
}
```

Example 2

The example C++ code listed here XORs two layers from a MEM format file together, and the result is saved as a DXF format file.

```
#include <string>
#include <fstream>
#include "DDInterface.hpp"
#include "DDMEMMaskReader.hpp"
#include "DDMaskDefinitions.hpp"
#include "DDSurface.hpp"
#include "DDXOROperator.hpp"
#include "DDModifyGeometryTool.hpp"

int main(int argc, char* argv[])
{
  // This routine XORs two layers from a MEM format file together.
  // The result is saved as a DXF format file.

  // Open an MEM format file.
  std::string filename;
  filename="Example.mem";

  std::ifstream in_file;
  in_file.open(filename.c_str());
```

```
if (in_file.fail())
{
  std::cout<<"Unable to open file: "<<filename<<"\n";
  exit(1);
}

//Create GBL-2D surfaces for each loop in the MEM file.
// The DDMaskDefinitions class stores the loops from each layer of the MEM file
// as lists of surfaces indexed by layer name.
DDMEMMaskReader mask_reader;
DDMaskDefinitions mask_definitions;
mask_reader.create_surfaces_from_mask_file(in_file,mask_definitions);

// Close the MEM file.
in_file.close();

// Get a list of the layer names that were imported.
std::list<std::string> layers;
mask_definitions.get_layer_names(layers);

// Make sure that there are at least two layers.
int number_of_layers=layers.size();
if(number_of_layers<2)
{
  std::cout<<"There must be at least two layers in mask file.\n";
  exit(1);
}

// Get a list of the surfaces that make up the first layer.
// This returns a list of pointers to the instances stored in the
// DDMaskDefinitions class. Thus if these surfaces are deleted then
// invalid pointers will be found in the DDMaskDefinitions class.

std::string layer_A=layers.front();
std::list<DDSurface*> surfaces;
mask_definitions.get_surfaces_by_layer(layer_A,surfaces);

// Make a copy of the surfaces that can be changed by the XOR routine.
std::list<DDSurface*> surfaces_A;
DDModifyGeometryTool::copy_list_of_surfaces(surfaces,surfaces_A);

// Get a list of the surfaces that make up the second layer.
surfaces.clear();
layers.pop_front();
std::string layer_B=layers.front();
mask_definitions.get_surfaces_by_layer(layer_B,surfaces);

// Make a copy of the surfaces that can be changed by the XOR routine.
std::list<DDSurface*> surfaces_B;
DDModifyGeometryTool::copy_list_of_surfaces(surfaces,surfaces_B);

// XOR the surfaces from the two lists together.
// This routine requires that the surfaces in a list don't over lap any surfaces in
// the same list.
```

```
  // The surfaces in both lists are comsumed by the routine.
  // The new result surfaces are stored in the first list.
  DDXOROperator::fast_xor_list_of_surfaces_A_with_list_of_surfaces_B(surfaces_A,
                   surfaces_B, .001);

  // Write the results to a file in DXF format.
  std::string output_filename="output.dxf";
  std::string layer_name="XOR";
  DDInterface::write_list_of_surfaces_to_file_dxf_format(surfaces_A,
               .001, output_filename, layer_name);
}
```

DISTRIBUTION:

5 Corey L. McBride
Elemental Technologies
17 North Merchant Street
American Fork, UT 84003

1 Ray Myers
Elemental Technologies
17 North Merchant Street
American Fork, UT 84003

5 MS 0316
Rodney C. Schmidt, 1437

1 MS 0316
Richard Schiek, 1437

1 MS 0316
Scott Hutchinson, 1437

1 MS 0321
Jennifer Nelson, 1430

5 MS 1069
Vic Yarberry, 1737

1 MS 1080
Dave Sandison, 1749

1 MS 1243
Steve Kempka, 5535

2 MS 9018
Central Technical Files, 8944

2 MS 0899
Technical Library, 4536