CGAL Reference Manual

Part 1: Kernel Library

Release 1.0, April 1998

Preface

CGAL is a *Computational Geometry Algorithms Library* written in C++, which is developed by the ESPRIT project CGAL. This project is carried out by a consortium consisting of Utrecht University (The Netherlands), ETH Zürich (Switzerland), Freie Universiät Berlin (Germany), INRIA Sophia-Antipolis (France), Max-Planck Institut für Informatik, Saarbrücken (Germany), RISC Linz (Austria) and Tel-Aviv University (Israel). You find more information on the project on the CGAL home page at URL `http://www.cs.uu.nl/CGAL/`.

Should you have any questions, comments, remarks or criticism concerning CGAL, please send a message to `cgal@cs.uu.nl`.

Editors

Hervé Brönnimann, Andreas Fabri (INRIA Sophia-Antipolis).
Stefan Schirra (Max-Planck Institut für Informatik).
Remco Veltkamp (Utrecht University).

Authors

Andreas Fabri (INRIA Sophia-Antipolis).
Geert-Jan Giezeman (Utrecht University).
Lutz Kettner (ETH Zürich).
Stefan Schirra (Max-Planck Institut für Informatik).
Sven Schönherr (Freie Universiät Berlin).

Design and Implementation History

At a meeting at Utrecht University in January 1995, Olivier Devillers, Andreas Fabri, Wolfgang Freiseisen, Geert-Jan Giezeman, Mark Overmars, Stefan Schirra, Otfried Schwarzkopf (now Otfried Cheong), and Sven Schönherr discussed the foundations of the CGAL kernel. Many design and software engineering issues were addressed, e.g. naming conventions, coupling of classes (flat versus deep class hierarchy), memory allocation, programming conventions, modifiability of atomic objects, points and vectors, storing additional information, orthogonality of operations on the kernel objects, viewing non-constant-size objects like polygons as dynamic data structures (and hence not as part of the (innermost) kernel).

The people attending the meeting delegated the compilation of a draft specification to Stefan Schirra. The resulting draft specification was intentionally modeled on CGAL's precursors C++GAL and PLAGEO as well as on the geometric part of LEDA. The specification already featured coexistence of Cartesian and homogeneous representation of point/vector data and parameterization by number type(s). During the discussion of the draft a kernel design group was formed. The members of this group were Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. The work of the kernel design group led to significant changes and improvements of the original design, e.g. the strong separation between points and vectors. Probably the most important enhancement was the design of a common superstructure for the previously uncoupled Cartesian and homogeneous representations. One can say, that the kernel was designed by this group.

A first version of the kernel was internally made available at the beginning of the CGAL-project (ESPRIT LTR IV project number 21957). Since then many more people contributed to the evolution of the kernel through discussions on the CGAL mailing lists. The implementation based on Cartesian representation was (mostly) provided by Andreas Fabri, the homogeneous representation (mostly) by Stefan Schirra. Intersection and distance computations were implemented by Geert-Jan Giezeman. The kernel is now maintained by Hervé Brönnimann, Geert-Jan Giezeman, and Stefan Schirra.

Acknowledgement

This work was supported by the Graduiertenkolleg 'Algorithmische Diskrete Mathematik', under grant DFG We 1265/2-1, and by the ESPRIT IV Long Term Research Project No. 21957 (CGAL).

Contents

Chapter 1

Introduction

This part of the reference manual covers the kernel. The kernel contains objects of constant size, such as point, vector, direction, line, ray, segment, triangle, iso-oriented rectangle and tetrahedron. With each type comes a set of functions which can be applied to an object of this type. You will typically find access functions (e.g. to the coordinates of a point), tests of the position of a point relative to the object, a function returning the bounding box, the length, or the area of an object, and so on. The CGAL kernel further contains basic operations such as affine transformations, detection and computation of intersections, and distance computations.

The correctness proof of nearly all geometric algorithms presented in theory papers assumes exact computation with real numbers. This leads to a fundamental problem with the implementation of geometric algorithms. Naively, often the exact real arithmetic is replaced by inexact floating-point arithmetic in the implementation. This often leads to acceptable results for many input data. However, even for the implementation of the simplest geometric algorithms this simplification occasionally does not work. Rounding errors introduced by an inaccurate arithmetic may lead to inconsistent decisions, causing unexpected failures for some correct input data. There are many approaches to this problem, one of them is to compute exactly (compute so accurate that all decisions made by the algorithm are exact) which is possible in many cases but more expensive than standard floating-point arithmetic. C. M. Hoffmann [Hof89a, Hof89b] illustrates some of the problems arising in the implementation of geometric algorithms and discusses some approaches to solve them. A more recent overview is given in [Sch97]. The exact computation paradigm is discussed by Yap and Dubé [YD95] and Yap [Yap97].

In CGAL you can choose the underlying number types and arithmetic. You can use different types of arithmetic simultaneously and the choice can be easily changed, e.g. for testing. So you can choose between implementations with fast but occasionally inexact arithmetic and implementations guaranteeing exact computation and exact results. Of course you have to pay for the exactness in terms of execution time and storage space.

CGAL and LEDA

LEDA [1] is a *Library of Efficient Data types and Algorithms* under development at the *Max-Planck Institut für Informatik*, Saarbrücken, Germany. CGAL is partially based on LEDA. It makes a strong combination with the combinatorial part of it, and can be used independently as well.

[1] `http://www.mpi-sb.mpg.de/LEDA/leda.html`

Organization of this manual

This document is organized as follows. Chapter 2 explains the basic concepts. It is a more technical introduction and we recommend to read it before the rest of the manual. Chapter 3 introduces a number of notions used throughout the kernel. Chapters 4 through 13 cover the 2D kernel, chapters 14 through 21 cover the 3D kernel.

Chapter 2

Preliminaries

Our object of study is the d-dimensional affine Euclidean space. Here we are concerned with cases $d = 2$ and $d = 3$. Objects in that space are sets of points. A common way to represent the points is the use of Cartesian coordinates, which assumes a reference frame (an origin and d orthogonal axes). In that framework, a point is represented by a d-tuple $(c_0, c_1, \ldots, c_{d-1})$, and so are vectors in the underlying linear space. Each point is represented uniquely by such Cartesian coordinates. Another way to represent points is by homogeneous coordinates. In that framework, a point is represented by a $(d+1)$-tuple $(h_0, h_1, \ldots, h_d)$. Via the formulae $c_i = h_i/h_d$, the corresponding point with Cartesian coordinates $(c_0, c_1, \ldots, c_{d-1})$ can be computed. Note that homogeneous coordinates are not unique. For $\lambda \neq 0$, the tuples $(h_0, h_1, \ldots, h_d)$ and $(\lambda \cdot h_0, \lambda \cdot h_1, \ldots, \lambda \cdot h_d)$ represent the same point. For a point with Cartesian coordinates $(c_0, c_1, \ldots, c_{d-1})$ a possible homogeneous representation is $(c_0, c_1, \ldots, c_{d-1}, 1)$. Homogeneous coordinates in fact allow to represent objects in a more general space, the projective space $\P_d$. In CGAL, we do not compute in projective geometry. Rather, we use homogeneous coordinates to avoid division operations, since the additional coordinate can serve as a common denominator.

2.1 Representation Class (R)

Almost all the kernel objects (and the corresponding functions) are templates with a parameter that allows the user to choose the representation of the kernel objects. A type that is used as an argument for this parameter must fulfill certain requirements on syntax and semantics. The list of requirements defines an abstract concept which is called a *representation class* in CGAL and denoted by R throughout this manual. The complete list of requirements is beyond the scope of this introduction. It is important only if you want to define your own representation class. It is sufficient to know that a representation class provides the actual implementations of the kernel objects.

CGAL offers two families of concrete models for the concept representation class, one based on the Cartesian representation of points and one based on the homogeneous representation of points. The interface of the kernel objects is designed such that it works well with both Cartesian and homogeneous representation, for example, points in 2D have a constructor with three arguments as well. The common interfaces parameterized with a representation class allow one to develop code independent of the chosen representation. We said "families" of models, because both families are parameterized too. A user can choose the number type used to represent the coordinates.

For reasons that will become evident later, a representation class provides two typenames for number types, namely *R::FT* and *R::RT*.[1] The type *R::FT* must fulfill the requirements on what is called

[1] The double colon :: is the C++ scope operator.

a *field type* in CGAL. This roughly means that *R::FT* is a type for which operations +, −, ∗ and / are defined with semantics (approximately) corresponding to those of a field in a mathematical sense. Note that, strictly speaking, the built-in type *int* does not fullfil the requirements on a field type, since *int*s correspond to elements of a ring rather than a field, especially operation / is not the inverse of ∗. The requirements on the type *R::RT* are weaker. This type must fulfill the requirements on what is called a *ring type* in CGAL. This roughly means that *R::RT* is a type for which operations +, −, ∗ are defined with semantics (approximately) corresponding to those of a ring in a mathematical sense. A very limited division operation / must be available as well. It must work for exact (i.e., no remainder) integer divisions only. Furthermore, both number types should fulfill CGAL's requirements on a number type. Note that a ring type is always a field type but not the other way round.

We describe below the two representations provided for CGAL kernel objects, namely *CGAL_Cartesian* and *CGAL_Homogeneous*.

2.1.1 Cartesian Representation (*CGAL_Cartesian*<*FT*>)

With *CGAL_Cartesian*<*FT*> you can choose Cartesian representation of coordinates. When you choose Cartesian representation you have to declare at the same time the type of the coordinates. A number type used with the *CGAL_Cartesian* representation class should be a *field type* as described above. As mentioned above, the built-in type *int* is not a field type. However, for some computations with Cartesian representation, no division operation is needed, i.e., a *ring type* is sufficient in this case.)

The declaration for a point with FT = `double` and with coordinates $(1/3, 5/3)$ looks as follows:

```
CGAL_Point_2< CGAL_Cartesian<double> > p(1.0/3.0, 5.0/3.0);
```

The keyword `double` makes that the program allocates memory for storing the x and y coordinate in double precision format.

With *CGAL_Cartesian*<*NT*>, both *CGAL_Cartesian*<*NT*>*::FT* and *CGAL_Cartesian*<*NT*>*::RT* are mapped to number type NT.

2.1.2 Homogeneous Representation (*CGAL_Homogeneous*<*RT*>)

As we said before, homogeneous coordinates permit to avoid division operations in numerical computations, since the additional coordinate can serve as a common denominator. Avoiding divisions can be useful for exact geometric computation. With *CGAL_Homogeneous*<*RT*> you can choose homogeneous representation of coordinates with the kernel objects. As for Cartesian representation you have to declare at the same time the type used to store the homogeneous coordinates. Since the homogeneous representation allows one to avoid the divisions, the number type associated with a homogeneous representation class must be a model for the weaker concept ring type only. However, some operations provided by this kernel involve division operations, for example computing squared distances or returning a Cartesian coordinate. To keep the requirements on the number type parameter of *CGAL_Homogeneous* low, the number type *CGAL_Quotient*<*RT*> is used instead. This number type turns a ring type into a field type. It maintains numbers as quotients, i.e. a numerator and a denominator. Thereby, divisions are circumvented.

A variable declaration for a point at Cartesian coordinates $(1/3, 5/3)$ represented with homogeneous coordinates with ring type *double* then looks as follows:

```
CGAL_Point_2< CGAL_Homogeneous<double> > p(1.0, 5.0, 3.0);
```

With *CGAL_Homogeneous*<*NT*>, *CGAL_Homogeneous*<*NT*>*::FT* is equal to *CGAL_Quotient*<*NT*> while *CGAL_Homogeneous*<*NT*>*::RT* is equal to NT.

2.1.3 Choosing a Representation Class

If you start with integral Cartesian coordinates, many geometric computations will involve integral numerical values only. Especially, this is true for geometric computations that evaluate only predicates, which are tantamount to determinant computations. Examples are triangulation of point sets and convex hull computation. In this case, the Cartesian representation is probably the first choice, even with a ring type. You might use limited precision integer types like *int* or *long*, use *double* to present your integers (they have more bits in their mantissa than an *int* and overflow nicely), or an arbitrary precision integer type like the wrapper *CGAL_Gmpz* for the GMP integers or *leda_integer*. Note, that unless you use an arbitrary precision integer type, incorrect results might arise due to overflow.

If new points are to be constructed, for example the intersection point of two lines, computation of Cartesian coordinates usually involves divisions, so you need to use a field type with Cartesian representation or have to switch to homogeneous representation. *double* is a possible, but imprecise field type. You can also put any ring type into *CGAL_Quotient* to get a field type and put it into *CGAL_Cartesian*, but you better put the ring type into *Homogeneous*. *leda_rational* and *leda_real* are valid field types, too.

If it is crucial for you that the computation is reliable, the right choice are probably number types that guarantee exact computation. The number type *leda_real* guarantees that all decisions and hence all branchings in a computation are correct. They also allow you to compute approximations to whatever precision you need. Furthermore computation with *leda_real* is faster than computation with arbitrary precision arithmetic. So if you would like to avoid surprises caused by imprecise computation, this is a good choice. In fact, it is a good choice with both representations, since divisions slow down the computation of the reals and hence it might pay-off to avoid them.

Still other people will prefer the built-in type `double`, because they need speed and can live with approximate results, or even algorithms that, from time to time, crash or compute incorrect results due to accumulated rounding errors.

Chapter 3

Kernel Utilities

In this chapter we introduce some basic enumeration types and constants. Futhermore we define the class *CGAL_Object*, which is a generic object that can contain an object of any type.

3.1 Order types

In geometric algorithms we often want to know the orientation type of a sequence of $d+1$ points in d dimensional space. CGAL provides the following enumeration type:

enum CGAL_Sign { *CGAL_NEGATIVE = -1, CGAL_ZERO, CGAL_POSITIVE*}*;*

and a

typedef CGAL_Sign CGAL_Orientation;.

For the two-dimensional space, different names are often used in the literature. Here one wants to know whether three points perform a *leftturn*, or a *rightturn*, or if they are *collinear*. The latter includes the case that two or even all three points have the same coordinates. Therefore, CGAL also provides

const CGAL_Orientation *CGAL_LEFTTURN = CGAL_POSITIVE;*

const CGAL_Orientation *CGAL_RIGHTTURN = CGAL_NEGATIVE;*

const CGAL_Orientation *CGAL_COUNTERCLOCKWISE = CGAL_POSITIVE;*

const CGAL_Orientation *CGAL_CLOCKWISE = CGAL_NEGATIVE;*

const CGAL_Orientation *CGAL_COPLANAR = CGAL_ZERO;*

const CGAL_Orientation *CGAL_COLLINEAR = CGAL_ZERO;*

const CGAL_Orientation *CGAL_DEGENERATE = CGAL_ZERO;*

3.2 Relative Position

Geometric objects in CGAL have member functions that test the position of a point relative to the object. Full dimensional objects and their boundaries are represented by the same type, e.g. halfspaces and hyperplanes are not distinguished, neither are spheres and circles. Such objects split the ambient space into two full-dimensional parts, a bounded part and an unbounded part (e.g. circles), or two unbounded parts (e.g. hyperplanes). By default these objects are oriented, i.e., one of the resulting parts is called the positive side, the other one is called the negative side. Both of these may be unbounded.

For these objects there is a function *oriented_side()* that determines whether a test point is on the positive side, the negative side, or on the oriented boundary. These function returns an enumeration type

enum CGAL_Oriented_side { *CGAL_ON_NEGATIVE_SIDE = -1,*
CGAL_ON_ORIENTED_BOUNDARY,
CGAL_POSITIVE_SIDE}

Accordingly, there are the three member functions *has_on_negative_side(CGAL_Point_d<R>)*, *has_on_positive_side(CGAL_Point_d<R>)*, and *has_on_boundary(CGAL_Point_d<R>)*, returning a boolean value, where d is 2 or 3, according to the dimension of the ambient space.

Those objects that split the space in a bounded and an unbounded part, have a member function *bounded_side()* with the return type

enum = CGAL_Bounded_side { *CGAL_ON_BOUNDED_SIDE = -1,*
CGAL_ON_BOUNDARY,
CGAL_ON_UNBOUNDED_SIDE}

Accordingly, there are the member functions *has_on_bounded_side(CGAL_Point_d<R>)* and *has_on_unbounded_side(CGAL_Point_d<R>)*, returning a boolean value.

If an object is lower dimensional, e.g. a triangle in three-dimensional space or a segment in two-dimensional space, there is only a test whether a point belongs to the object or not. This member function, which takes a point as an argument and returns a boolean value, is called *has_on()*.

3.3 Comparison Results

enum CGAL_Comparison_result { *CGAL_SMALLER = -1, CGAL_EQUAL, CGAL_LARGER*}*;*

3.4 Generic Object (*CGAL_Object*)

Definition

Some functions can return different types of objects. A typical C++ solution to this problem is to derive all possible return types from a common base class, to return a pointer to this class and to perform a dynamic cast on this pointer. The class *CGAL_Object* provides an abstraction. An object *obj* of the class *CGAL_Object* can represent an arbitrary class. The only operations it provides is to make copies and assignments, so that you can put them in lists or arrays. Note that *CGAL_Object* s NOT a common base class for the elementary classes. Therefore, there is no automatic conversion from these classes to *CGAL_Object*. Rather this is done with the method *CGAL_makeobject()*. This encapsulation mechanism requires the use of *CGAL_assign* to use the functionality of the encapsulated class.

Creation

CGAL_Object obj; introduces an uninitialized variable.

CGAL_Object obj(o); Copy constructor.

CGAL_Object	*CGAL_make_object(T t)*	Creates an object that contains t.

Operations

CGAL_Object &	$obj = o$	Assignment.
bool	*CGAL_assign(CGAL_Class &c, o)*	assigns o to c if o was constructed from an object of type *CGAL_Class*. Returns true, if the assignment was possible.

Example

In the following example, the object class is used as return value for the intersection computation, as there are possibly different return values.

```
{
    CGAL_Point_2< CGAL_Cartesian<double> > point;
    CGAL_Segment_2< CGAL_Cartesian<double> > segment,  segment_1, segment_2;

    cin >> segment_1 >> segment_2;

   CGAL_Object obj = CGAL_intersection(segment_1, segment_2);

    if (CGAL_assign(point, obj)) {
        /* do something with point */
    } else if ((CGAL_assign(segment, obj)) {
        /* do something with segment*/
    }
    /*  there was no intersection */
}
```

The intersection routine itself looks roughly as follows:

```
template < class R >
CGAL_Object  CGAL_intersection(CGAL_Segment_2<R> s1, CGAL_Segment_2<R> s2)
{
    if (/* intersection in a point */ ) {
       CGAL_Point_2<R> p = ...  ;
       return CGAL_make_object(p);
    } else if (/* intersection in a segment */ ) {
       CGAL_Segment_2<R> s = ...  ;
       return CGAL_make_object(s);
    }
    return CGAL_Object();
}
```

Chapter 4

The 2D Kernel: an Overview

This chapters presents an overview of the two-dimensional kernel of CGAL. The kernel consists of elementary 2D objects (such as points, lines, etc.), predicates on these objects (such as coordinate comparison, orientation, in-circle, etc.), and methods to compute distances and intersections between these objects. We also provide affine transformations on these objects. These classes are all templated by the representation class R, which is currently either *CGAL_Cartesian<FT>* or *CGAL_Homogeneous<RT>* for a so-called field tupe FT, and ring type RT, see Chapter 2.

4.1 Elementary 2D objects

CGAL provides points, vectors, and directions. A *point* is a point in the two-dimensional Euclidean plane $\mathbb{E}_2$, a *vector* is the difference of two points p_2, p_1 and denotes the direction and the distance from p_1 to p_2 in the vector space $\mathbb{R}^2$, and a *direction* represents the family of vectors that are positive multiples of each other. Their interface is described in Chapter 5.

CGAL_Point_2<R>
CGAL_Vector_2<R>
CGAL_Direction_2<R>

Lines in CGAL are directed, that is, they induce a partition of the plane into a positive side and a negative side. Any two points on a line induce an orientation of this line. A ray is semi-infinite interval on a line, and this line is oriented from the finite endpoint of this interval towards any other point in this interval. A segment is a bounded interval on a directed line, and the endpoints are ordered so that they induce the same direction as that of the line. Their interface is described in Chapter 6.

CGAL_Line_2<R>
CGAL_Ray_2<R>
CGAL_Segment_2<R>

Next we introduce triangles and iso-oriented rectangles. More complex polygons can be obtained from the basic library (*CGAL_Polygon_2*), so they are not part of the kernel. In the same category, we introduce circles in the plane. As with any Jordan curves, triangles, iso-oriented rectangles and circles separate the plane into two regions, one bounded and one unbounded. Their interface is described in the Chapters 7, 8, and 9.

CGAL_Triangle_2<*R*>
CGAL_Iso_rectangle_2<*R*>
CGAL_Circle_2<*R*>

4.2 Predicates and functions

For testing where a point p lies with respect to the line defined by two points q and r, one may be tempted to construct the line *CGAL_Line_2*<*R*>*(q,r)* and use the function call *oriented_side(p)*. Pays off if many tests with respect to the line are made. Nevertheless, unless the number type is exact, the constructed line is only approximated, and round-off errors may lead *oriented_side(p)* to return an orientation which is different from the orientation of p, q, and r. This is the well-known problem of *robustness*.

In CGAL, we provide predicates in which such geometric decisions are made directly with a reference to the input points p, q, r, without an intermediary object like a line. This enables to use exact predicates that are cheaper than exact number types, a concept that has been the focus of much research recently in computational geometry. For the above test, the recommended way to get the result is to use *CGAL_orientation(p,q,r)*.

Consequently, we propose the most common predicates in Chapter 10. They use the elementary classes of the 2D kernel.

Affine transformations allow to generate new object instances under arbitrary affine transformations. These transformations include translations, rotations and scaling. Most of the classes above have a member function *transform(CGAL_Aff_transformation t)* which applies the transformation to the object instance. The interface of the transformation class is described in Chapter 11.

CGAL_Aff_transformation_2<*R*>

CGAL also provides a set of functions that detect or compute the intersection between any two objects of the 2D kernel, and calculate their squared distance. These functions and their input types are described in Chapters 12 and 13.

Chapter 5

2D Point, Vector and Direction

We strictly distinguish between points, vectors and directions. A *point* is a point in the two-dimensional Euclidean plane $\mathbb{E}_2$, a *vector* is the difference of two points p_2, p_1 and denotes the direction and the distance from p_1 to p_2 in the vector space $\mathbb{R}^2$, and a *direction* is a vector where we forget about its length. They are different mathematical concepts. For example, they behave different under affine transformations and an addition of two points is meaningless in affine geometry. By putting them in different classes we not only get cleaner code, but also type checking by the compiler which avoids ambiguous expressions. Hence, it pays twice to make this distinction.

5.1 2D Point (*CGAL_Point_2*<*R*>)

Definition

An object of the class *CGAL_Point_2* is a point in the two-dimensional Euclidean plane $\mathbb{E}_2$.

Remember that *R::RT* and *R::FT* denote a ring type and a field type. For the representation class *CGAL_Cartesian*<*T*> the two types are the same. For the representation class *CGAL_Homogeneous*<*T*> the ring type is *R::RT* == *T*, and the field type is *R::FT* == *CGAL_Quotient*<*T*>.

#include <*CGAL/Point_2.h*>

Creation

CGAL_Point_2<*R*> *p(R::RT hx, R::RT hy, R::RT hw = R::RT(1));*

introduces a point p initialized to $(hx/hw, hy/hw)$. If the third argument is not explicitly given, it defaults to *R::RT(1)*.

Operations

bool	$p == q$	Test for equality. Two points are equal, iff their x and y coordinates are equal.

bool	$p \mathrel{!=} q$	Test for inequality.

There are two sets of coordinate access functions, namely to the homogeneous and to the Cartesian coordinates. They can be used independently from the chosen representation type R.

R::RT	*p.hx()*	returns the homogeneous x coordinate.
R::RT	*p.hy()*	returns the homogeneous y coordinate.
R::RT	*p.hw()*	returns the homogenizing coordinate.

Here come the Cartesian access functions. Note that you do not loose information with the homogeneous representation, because then the field type is a quotient.

R::FT	*p.x()*	returns the Cartesian x coordinate, that is hx/hw.
R::FT	*p.y()*	returns the Cartesian y coordinate, that is hy/hw.

The following operations are for convenience and for making this point class compatible with code for higher dimensional points. Again they come in a Cartesian and homogeneous flavor.

R::RT	*p.homogeneous(int i)*	
		returns the i'th homogeneous coordinate of p, starting with 0. *Precondition*: $0 \leq i \leq 2$.
R::FT	*p.cartesian(int i)*	
		returns the i'th Cartesian coordinate of p, starting with 0. *Precondition*: $0 \leq i \leq 1$.
R::FT	*p[int i]*	returns *cartesian(i)*. *Precondition*: $0 \leq i \leq 1$.
int	*p.dimension()*	
		returns the dimension (the constant 2).
CGAL_Bbox_2	*p.bbox()*	returns a bounding box containing p. Note that bounding boxes are not parameterized with whatsoever.
CGAL_Point_2<R>	*p.transform(CGAL_Aff_transformation_2<R> t)*	
		returns the point obtained by applying t on p.

The following operations can be applied on points:

CGAL_Vector_2<R>	$p - q$	returns the difference vector between q and p.

CGAL_Point_2<*R*> $\quad p + $ *CGAL_Vector_2*<*R*> v

returns a point obtained by translating p by the vector v.

CGAL_Point_2<*R*> $\quad p - $ *CGAL_Vector_2*<*R*> v

returns a point obtained by translating p by the vector $-v$.

See Also

2D Geometric Predicates, Chapter 10.

Example

The following declaration creates two points with Cartesian double coordinates.

```
CGAL_Point_2< CGAL_Cartesian<double> > p, q(1.0, 2.0);
```

The variable `p` is uninitialized and should first be used on the left hand side of an assignment.

```
p = q;

cout << p.x() << "  " << p.y() << endl;
```

5.2 Point Conversion

For convenience, CGAL provides functions for conversion of points between homogeneous and Cartesian representation.

#include <*CGAL/cartesian_homogeneous_conversion.h*>

Conversion from Cartesian representation to homogeneous representation with the same number type is straightforward. The homogenizing coordinate is set to 1, all other homogeneous coordinates are copied from the corresponding Cartesian coordinate.

CGAL_Point_2< *CGAL_Homogeneous*<*RT*> >

CGAL_cartesian_to_homogeneous(CGAL_Point_2< *CGAL_Cartesian*<*RT*> > *cp)*

converts 2d point *cp* with Cartesian representation into a 2d point with homogeneous representation with the same number type.

Conversion from homogeneous representation to Cartesian representation with the same number type involves division by the homogenizing coordinate.

CGAL_Point_2< CGAL_Cartesian<FT> >

CGAL_homogeneous_to_cartesian(CGAL_Point_2< CGAL_Homogeneous<FT> > hp)

converts 2d point *hp* with homogeneous representation into a 2d point with Cartesian representation with the same number type.

Since conversion involves division, concerning exactness, the correspondence is rather between homogeneous representation with number type RT and Cartesian representation with number type *CGAL_Quotient<RT>* than between representation with the same number type.

CGAL_Point_2< CGAL_Cartesian<CGAL_Quotient<RT> > >

CGAL_homogeneous_to_quotient_cartesian(CGAL_Point_2<CGAL_Homogeneous<RT> > hp)

converts 2d point *hp* with homogeneous representation with number type RT into a 2d point with Cartesian representation with number type *CGAL_Quotient<RT>*.

CGAL_Point_2< CGAL_Homogeneous<RT> >

CGAL_quotient_cartesian_to_homogeneous(CGAL_Point_2< CGAL_Cartesian< CGAL_Quotient<RT> > > cp)

converts 2d point *cp* with Cartesian representation with number type *CGAL_Quotient<RT>* into a 2d point with homogeneous representation with number type RT.

Of the last two functions, the conversion from homogeneous representation to Cartesian representation with quotients is always exact. The Conversion from Cartesian representation with quotients to homogeneous representation, however, might be inexact with some number types due to overflow or rounding in multiplications.

5.3 2D Vector (*CGAL_Vector_2<R>*)

Definition

An object of the class *CGAL_Vector_2* is a vector in the two-dimensional vector space $\mathbb{R}^2$. Geometrically spoken, a vector is the difference of two points p_2, p_1 and denotes the direction and the distance from p_1 to p_2.

CGAL defines a symbolic constant *CGAL_NULL_VECTOR*. We will explicitly state where you can pass this constant as an argument instead of a vector initialized with zeros.

#include <CGAL/Vector_2.h>

Creation

CGAL_Vector_2<R> v(R::RT hx, R::RT hy, R::RT hw = R::RT(1));

introduces a vector v initialized to $(hx/hw, hy/hw)$. If the third argument is not explicitly given, it defaults to *R::RT(1)*.

Operations

bool	$v == w$	Test for equality: two vectors are equal, iff their x and y coordinates are equal. You can compare a vector with the *CGAL_NULL_VECTOR*.
bool	$v\ != w$	Test for inequality. You can compare a vector with the *CGAL_NULL_VECTOR*.

There are two sets of coordinate access functions, namely to the homogeneous and to the Cartesian coordinates. They can be used independently from the chosen representation type R.

R::RT	*v.hx()*	returns the homogeneous x coordinate.
R::RT	*v.hy()*	returns the homogeneous y coordinate.
R::RT	*v.hw()*	returns the homogenizing coordinate.

Here come the Cartesian access functions. Note that you do not loose information with the homogeneous representation, because then the field type is a quotient.

R::FT	*v.x()*	returns the x-coordinate of v, that is hx/hw.
R::FT	*v.y()*	returns the y-coordinate of v, that is hy/hw.

The following operations are for convenience and for making the class *CGAL_Vector_2* compatible with code for higher dimensional vectors. Again they come in a Cartesian and homogeneous flavor.

R::RT	*v.homogeneous(int i)*	returns the i'th homogeneous coordinate of v, starting with 0. *Precondition*: $0 \leq i \leq 2$.
R::FT	*v.cartesian(int i)*	returns the i'th Cartesian coordinate of v, starting at 0. *Precondition*: $0 \leq i \leq 1$.
R::FT	*v[int i]*	returns *coordinate(i)*. *Precondition*: $0 \leq i \leq 1$.
int	*v.dimension()*	returns the dimension (the constant 2).
CGAL_Vector_2<R>	*v.transform(CGAL_Aff_transformation_2<R> t)*	returns the vector obtained by applying t on v.
CGAL_Vector_2<R>	*v.perpendicular(CGAL_Orientation o)*	returns the vector perpendicular to v in clockwise or counterclockwise orientation.

The following operations can be applied on vectors:

CGAL_Vector_2<R>	$v + w$	Addition.
CGAL_Vector_2<R>	$v - w$	Subtraction.
CGAL_Vector_2<R>	$-v$	returns the opposite vector.
R::FT	$v * w$	returns the scalar product (= inner product) of the two vectors.
CGAL_Vector_2<R>	v * *R::RT s*	Multiplication with a scalar from the right. Although it would be more natural, CGAL does not offer a multiplication with a scalar from the left. (This is due to problems of some compilers.)
CGAL_Vector_2<R>	v * *CGAL_Quotient<RT> s*	Multiplication with a scalar from the right.
CGAL_Vector_2<R>	v / *R::RT s*	Division by a scalar.
CGAL_Direction_2<R>	*v.direction()*	returns the direction which passes through v.

5.4 2D Direction ($CGAL\_Direction\_2$<R>)

Definition

An object of the class *CGAL_Direction_2* is a vector in the two-dimensional vector space $\mathbb{R}^2$ where we forget about its length. They can be viewed as unit vectors, although there is no normalization internally, since this is error prone. Directions are used whenever the length of a vector does not matter. For example, you can ask for the direction orthogonal to an oriented plane, or the direction of an oriented line. Further, they can be used to indicate angles. The slope of a direction is *dy()/dx()*.

#include <CGAL/Direction_2.h>

Creation

CGAL_Direction_2<R> d(CGAL_Vector_2<R> v);

introduces the direction d of vector v.

CGAL_Direction_2<R> d(R::RT x, R::RT y);

introduces a direction d passing through the point at (x, y).

Operations

R::RT	*d.delta(int i)*	returns the i'th value of the slope of d. *Precondition*: : $0 \le i \le 1$.
R::RT	*d.dx()*	returns the dx value of the slope of d.
R::RT	*d.dy()*	returns the dy value of the slope of d.

There is a total order on directions. We compare the angles between the positive x-axis and the directions in counterclockwise order.

bool	$d == e$	Test for equality.
bool	$d\ != e$	Test for inequality.
bool	$d < e$	
bool	$d > e$	
bool	$d <= e$	
bool	$d >= e$	

bool	*d.counterclockwise_in_between(d1, d2)*	
CGAL_Direction_2<*R*>	$-d$	The direction opposite to d.
CGAL_Vector_2<*R*>	*d.vector()*	returns a vector that has the same direction as d.
CGAL_Direction_2<*R*>	*d.transform(CGAL_Aff_transformation_2*<*R*> *t)*	returns the direction obtained by applying t on d.

5.5 Conversion between Points and Vectors

We stated earlier that it does not make sense to add two points, but it does make sense to subtract them and the result should be a vector. CGAL defines a symbolic constant *CGAL_ORIGIN* which denotes the point at the origin. Subtracting it from a point p results in the locus vector of p.

```
CGAL_Point_2< CGAL_Cartesian<double> >  p(1.0, 1.0), q;

CGAL_Vector2< CGAL_Cartesian<double> >  v;

v = p - CGAL_ORIGIN;

q = CGAL_ORIGIN + v;
```

In order to obtain the point corresponding to a vector v you simply have to add v to *CGAL_ORIGIN*. If you want to determine the point q in the middle between two points p_1 and p_2, you can write

```
q = p_1 + (p_2 - p_1) / 2.0;
```

Note that these constructions do not involve any performance overhead for the conversion with the currently available representation classes, if compiler optimisation is used, see also below.

5.6 Implementation

Points, vectors and directions use a handle/representative mechanism. A handle is an intelligent pointer, a representative is an object with a reference counter. The three classes have the same internal representation, namely a tuple of coordinates (plus a homogenizing coordinate in the case of homogeneous coordinates), which makes assignment, copy constructors and type conversion cheap.

An assignment makes the handle of the left hand side point to the representative the handle on the right hand sidepoints to. The copy constructor creates a new handle which points to the same representative. This especially pays if your coordinates are of non-constant size.

What about conversion? We explained that you convert by subtracting the origin from a point to obtain its locus vector, or by adding a vector to the origin to obtain the corresponding point. Although the origin behaves like a point, *CGAL_ORIGIN* is not an object of the class *CGAL_Point_2*<*R*> but of the class *CGAL_Origin*. All constructors and operators taking a point as argument are overloaded with the origin class in order to avoid memory allocation (for a point with coordinates zero), and arithmetic operations (where numbers would be added to zero). To give an example: when you “add” a vector v to the origin, only a new handle is created which points to the representation of v.

Chapter 6

2D Line, Ray and Segment

6.1 2D Line (*CGAL_Line_2<R>*)

Definition

An object l of the data type *CGAL_Line_2* is a directed straight line in the two-dimensional Euclidean plane $\mathbb{E}_2$. It is defined by the set of points with Cartesian coordinates (x, y) that satisfy the equation

$$l:\ a\,x + b\,y + c = 0.$$

The line splits $\mathbb{E}_2$ in a *positive* and a *negative* side.[1] A point p with Cartesian coordinates (px, py) is on the positive side of l, iff $a\,px + b\,py + c > 0$, it is on the negative side of l, iff $a\,px + b\,py + c < 0$.

#include <CGAL/Line_2.h>

Creation

CGAL_Line_2<R> *l(R::RT a, R::RT b, R::RT c);*

introduces a line l with the line equation $ax + by + c = 0$.

CGAL_Line_2<R> *l(CGAL_Point_2<R> p, CGAL_Point_2<R> q);*

introduces a line l passing through the points p and q. Line l is directed from p to q.

CGAL_Line_2<R> *l(CGAL_Point_2<R> p, CGAL_Direction_2<R> d);*

introduces a line l passing through point p with direction d.

[1] See Chapter 3 for the definition of *CGAL_Oriented_side*.

Operations

bool	$l == h$	Test for equality: two lines are equal, iff they have a non empty intersection and the same direction.
bool	$l\ != h$	Test for inequality.
R::RT	*l.a()*	returns the first coefficient of $\mathcal{L}$.
R::RT	*l.b()*	returns the second coefficient of $\mathcal{L}$.
R::RT	*l.c()*	returns the third coefficient of $\mathcal{L}$.
CGAL_Point_2<R>	*l.point(int i)*	returns an arbitrary point on l. It holds *point(i) == point(j)*, iff *i==j*. Furthermore, l is directed from *point(i)* to *point(j)*, for all $i < j$.
CGAL_Direction_2<R>	*l.direction()*	returns the direction of l.
R::FT	*l.x_at_y(R::FT y)*	returns the x-coordinate of l at a given y-coordinate. *Precondition*: l is not horizontal.
R::FT	*l.y_at_x(R::FT x)*	returns the y-coordinate of l at a given x-coordinate. *Precondition*: l is not vertical.
CGAL_Line_2<R>	*l.perpendicular(CGAL_Point_2<R> p)*	returns the line perpendicular to l passing through p where the direction is the direction of l rotated counterclockwise by 90 degrees.
CGAL_Line_2<R>	*l.opposite()*	returns the line with opposite direction.
CGAL_Point_2<R>	*l.projection(CGAL_Point_2<R> p)*	returns the orthogonal projection of p onto l.
bool	*l.is_degenerate()*	line l is degenerate, if the coefficients a and b of the line equation are zero.
bool	*l.is_horizontal()*	
bool	*l.is_vertical()*	
CGAL_Oriented_side	*l.oriented_side(CGAL_Point_2<R> p)*	returns *CGAL_ON_ORIENTED_BOUNDARY*, *CGAL_ON_NEGATIVE_SIDE*, or the constant *CGAL_ON_POSITIVE_SIDE*, depending on where point p is relative to the oriented line l.

For convenience we provide the following boolean functions:

bool	*l.has_on(CGAL_Point_2<R> p)*	
bool	*l.has_on_boundary(CGAL_Point_2<R> p)*	
		returns *has_on()*.
bool	*l.has_on_positive_side(CGAL_Point_2<R> p)*	
bool	*l.has_on_negative_side(CGAL_Point_2<R> p)*	
CGAL_Line_2<R>	*l.transform(CGAL_Aff_transformation_2<R> t)*	
		returns the line obtained by applying t on a point on l and the direction of l.

Implementation

Lines are implemented as a line equation. Construction from points or a point and a direction, might lead to loss of precision if the number type is not exact.

Example

Let us first define two Cartesian two-dimensional points in the Euclidean plane $\mathbb{E}_2$. Their dimension and the fact that they are Cartesian is expressed by the suffix *_2* and the representation type *CGAL_Cartesian*.

```
CGAL_Point_2< CGAL_Cartesian<double> >  p(1.0,1.0), q(4.0,7.0);
```

To define a line l we write:

```
CGAL_Line_2< CGAL_Cartesian<double> > l(p,q);
```

6.2 2D Ray (*CGAL_Ray_2<R>*)

Definition

An object r of the data type *CGAL_Ray_2* is a directed straight ray in the two-dimensional Euclidean plane $\mathbb{E}_2$. It starts in a point called the *source* of r and goes to infinity.

#include <CGAL/Ray_2.h>

Creation

CGAL_Ray_2<R> r(CGAL_Point_2<R> p, Point_2 q);

introduces a ray r with source p and passing through point q.

CGAL_Ray_2<R> r(CGAL_Point_2<R> p, CGAL_Direction_2<R> d);

introduces a ray r starting at source p with direction d.

Operations

bool	$r == h$	Test for equality: two rays are equal, iff they have the same source and the same direction.
bool	r != h	Test for inequality.
CGAL_Point_2<R>	*r.source()*	returns the source of r.
CGAL_Point_2<R>	*r.point(int i)*	returns a point on r. *point(0)* is the source, *point(i)*, with $i > 0$, is different from the source. *Precondition*: $i \geq 0$.
CGAL_Direction_2<R>	*r.direction()*	returns the direction of r.
CGAL_Line_2<R>	*r.supporting_line()*	returns the line supporting r which has the same direction.
CGAL_Ray_2<R>	*r.opposite()*	returns the ray with the same source and the opposite direction.
bool	*r.is_degenerate()*	ray r is degenerate, if the source and the second defining point fall together (that is if the direction is degenerate).
bool	*r.is_horizontal()*	
bool	*r.is_vertical()*	

bool *r.has_on(CGAL_Point_2<R> p)*

A point is on r, iff it is equal to the source of r, or if it is in the interior of r.

bool *r.collinear_has_on(CGAL_Point_2<R> p)*

checks if point p is on r. This function is faster than function *has_on()* if the precondition checking is disabled.
Precondition: p is on the supporting line of r.

CGAL_Ray_2<R> *r.transform(CGAL_Aff_transformation_2<R> t)*

returns the ray obtained by applying t on the source and on the direction of r.

Implementation

A ray is stored as a point and a direction.

6.3 2D Segment (*CGAL_Segment_2<R>*)

Definition

An object s of the data type *CGAL_Segment_2* is a directed straight line segment in the two-dimensional Euclidean plane $\mathbb{E}_2$, i.e. a straight line segment $[p, q]$ connecting two points $p, q \in \mathbb{R}^2$. The segment is topologically closed, i.e. the end points belong to it. Point p is called the *source* and q is called the *target* of s. The length of s is the Euclidean distance between p and q. Note that there is only a function to compute the square of the length, because otherwise we had to perform a square root operation which is not defined for all number types, is expensive, and may be inexact.

#include <CGAL/Segment_2.h>

Creation

CGAL_Segment_2<R> s(CGAL_Point_2<R> p, CGAL_Point_2<R> q);

introduces a segment s with source p and target q. The segment is directed from the source towards the target.

Operations

bool	$s == q$	Test for equality: Two segments are equal, iff their sources and targets are equal.
bool	s != q	Test for inequality.
CGAL_Point_2<R>	*s.source()*	returns the source of s.
CGAL_Point_2<R>	*s.target()*	returns the target of s.
CGAL_Point_2<R>	*s.min()*	returns the point of s with lexicographically smallest coordinate.
CGAL_Point_2<R>	*s.max()*	returns the point of s with lexicographically largest coordinate.
CGAL_Point_2<R>	*s.vertex(int i)*	returns source or target of s: *vertex(0)* returns the source of s, *vertex(1)* returns the target of s. The parameter i is taken modulo 2, which gives easy access to the other vertex.
CGAL_Point_2<R>	*s.point(int i)*	returns *vertex(i)*.
CGAL_Point_2<R>	*s[int i]*	returns *vertex(i)*.
R::FT	*s.squared_length()*	returns the squared length of s.

CGAL_Direction_2<R>	*s.direction()*	returns the direction from source to target of s.
CGAL_Segment_2<R>	*s.opposite()*	returns a segment with source and target point interchanged.
CGAL_Line_2<R>	*s.supporting_line()*	returns the line l passing through s. Line l has the same orientation as segment s.
bool	*s.is_degenerate()*	segment s is degenerate, if source and target are equal.
bool	*s.is_horizontal()*	
bool	*s.is_vertical()*	
bool	*s.has_on(CGAL_Point_2<R> p)*	A point is on s, iff it is equal to the source or target of s, or if it is in the interior of s.
bool	*s.collinear_has_on(CGAL_Point_2<R> p)*	checks if point p is on segment s. This function is faster than function *has_on()*. *Precondition*: p is on the supporting line of s.
CGAL_Bbox_2	*s.bbox()*	returns a bounding box containing s.
CGAL_Segment_2<R>	*s.transform(CGAL_Aff_transformation_2<R> t)*	returns the segment obtained by applying t on the source and the target of s.

Implementation

A segment is internally represented by two points.

Chapter 7

2D Simplex

7.1 2D Triangle (*CGAL_Triangle_2<R>*)

Definition

An object t of the class *CGAL_Triangle_2* is a triangle in the two-dimensional Euclidean plane $\mathbb{E}_2$. Triangle t is oriented, i.e., its boundary has clockwise or counterclockwise orientation. We call the side to the left of the boundary the positive side and the side to the right of the boundary the negative side.

As any Jordan curve the boundary of a triangle splits the plane in two open regions, a bounded one and an unbounded one.

#include <CGAL/Triangle_2.h>

Creation

CGAL_Triangle_2<R> t(CGAL_Point_2<R> p, CGAL_Point_2<R> q, CGAL_Point_2<R> r);

introduces a triangle t with vertices p, q and r.

Operations

bool	*t == t2*	Test for equality: two triangles are equal, iff there exists a cyclic permutation of the vertices of $t2$, such that they are equal to the vertices of t.
bool	*t != t2*	Test for inequality.
CGAL_Point_2<R>	*t.vertex(int i)*	returns the i'th vertex modulo 3 of t.
CGAL_Point_2<R>	*t[int i]*	returns *vertex(i)*.

bool	*t.is_degenerate()*	triangle t is degenerate, if the vertices are collinear.
CGAL_Orientation	*t.orientation()*	returns the orientation of t.
CGAL_Oriented_side	*t.oriented_side(CGAL_Point_2<R> p)*	
		returns *CGAL_ON_ORIENTED_BOUNDARY*, *CGAL_POSITIVE_SIDE*, or the constant *CGAL_ON_NEGATIVE_SIDE*, depending on where point p is.
CGAL_Bounded_side	*t.bounded_side(CGAL_Point_2<R> p)*	

For convenience we provide the following boolean functions:

bool	*t.has_on_positive_side(CGAL_Point_2<R> p)*	
bool	*t.has_on_negative_side(CGAL_Point_2<R> p)*	
bool	*t.has_on_boundary(CGAL_Point_2<R> p)*	
bool	*t.has_on_bounded_side(CGAL_Point_2<R> p)*	
bool	*t.has_on_unbounded_side(CGAL_Point_2<R> p)*	
CGAL_Triangle_2<R>	*t.opposite()*	returns a triangle where the boundary is oriented the other way round (this flips the positive and the negative side, but not the bounded and unbounded side).
CGAL_Bbox_2	*t.bbox()*	returns a bounding box containing t.
CGAL_Triangle_2<R>	*t.transform(CGAL_Aff_transformation_2<R> at)*	
		returns the triangle obtained by applying *at* on the three vertices of t.

Implementation

A triangle is internally represented as a triple of points.

Chapter 8

2D Iso-oriented Objects

In many applications the objects are iso-oriented, which means that their sides are parallel to the axes of the coordinate system. These objects not only have a compact description, but there are many algorithms which are faster for iso-oriented objects.

Bounding boxes are used to approximate objects and the only operations you can perform on them is to get their bounds and to check if two bounding boxes overlap. They are axis-parallel and the difference to the class *CGAL_Iso_rectangle_2*<*R*> is that they are not templated. A typical situation where to use them is to check if two complicated polygons intersect. You can first check if their respective bounding boxes intersect. If the answer is negative, you are done, otherwise you really have to check if the polygons intersect.

8.1 2D Iso Rectangle (*CGAL_Iso_rectangle_2*<*R*>)

Definition

An object s of the data type *CGAL_Iso_rectangle* is a rectangle in the Euclidean plane $\mathbb{E}_2$ with sides parallel to the x and y axis of the coordinate system.

Although they are represented in a canonical form by only two vertices, namely the lower left and the upper right vertex, we provide functions for "accessing" the other vertices as well. The vertices are returned in counterclockwise order.

Iso-oriented rectangles and bounding boxes are quite similar. The difference however is that bounding boxes have always double coordinates, whereas the coordinate type of an iso-oriented rectangle is chosen by the user.

#include <CGAL/Iso_rectangle_2.h>

Creation

CGAL_Iso_rectangle_2<*R*> *r(CGAL_Point_2*<*R*> *p, CGAL_Point_2*<*R*> *q);*

introduces an iso-oriented rectangle r with diagonal opposite vertices p and q. Note that the object is brought in the canonical form.

Operations

bool	*r == r2*	Test for equality: two iso-oriented rectangles are equal, iff their lower left and their upper right vertices are equal.
bool	*r != r2*	Test for inequality.
CGAL_Point_2<R>	*r.vertex(int i)*	returns the i'th vertex modulo 4 of r in counter-clockwise order, starting with the lower left vertex.
CGAL_Point_2<R>	*r[int i]*	returns *vertex(i)*.
CGAL_Point_2<R>	*r.min()*	returns the lower left vertex of r (= *vertex(0)*).
CGAL_Point_2<R>	*r.max()*	returns the upper right vertex of r (= *vertex(2)*).
bool	*r.is_degenerate()*	the iso-oriented rectangle r is degenerate, if all vertices are collinear.
CGAL_Bounded_side	*r.bounded_side(CGAL_Point_2<R> p)*	returns either *CGAL_ON_BOUNDED_SIDE*, *CGAL_ON_BOUNDARY* or the constant *CGAL_ON_UNBOUNDED_SIDE*, depending on where point p is.
bool	*r.has_on_boundary(CGAL_Point_2<R> p)*	
bool	*r.has_on_bounded_side(CGAL_Point_2<R> p)*	
bool	*r.has_on_unbounded_side(CGAL_Point_2<R> p)*	
CGAL_Bbox	*r.bbox()*	returns a bounding box containing r.
CGAL_Iso_rectangle_2<R>	*r.transform(CGAL_Aff_transformation_2<R> t)*	returns the iso-oriented rectangle obtained by applying t on the lower left and the upper right corner of r. *Precondition*: The angle at a rotation must be a multiple of $\pi/2$, otherwise the resulting rectangle does not have the same side length. Note that rotating about an arbitrary angle can even result in a degenerate iso-oriented rectangle.

8.2 2D Bbox (*CGAL_Bbox_2*)

Definition

An object b of the class *CGAL_Bbox_2* is a bounding box in the two-dimensional Euclidean plane $\mathbb{E}_2$. This class is not templated.

#include <*CGAL/Bbox_2.h*>

Creation

CGAL_Bbox_2 b(double x_min, double y_min, double x_max, double y_max);

introduces a bounding box b with lower left corner at *(xmin, ymin)* and with upper right corner at *(xmax, ymax)*.

Operations

bool	$b == c$	Test for equality.
bool	b != q	Test for inequality.
double	*b.xmin()*	
double	*b.ymin()*	
double	*b.xmax()*	
double	*b.ymax()*	
CGAL_Bbox_2	$b + c$	returns a bounding box of b and c.
bool	*CGAL_do_overlap(bb1, bb2)*	

Chapter 9

2D Curved Objects

9.1 2D Circle (*CGAL_Circle_2<R>*)

Definition

An object of type *CGAL_Circle_2<R>* is a circle in the two-dimensional Euclidean plane $\mathbb{E}_2$. The circle is oriented, i.e. its boundary has clockwise or counterclockwise orientation. The boundary splits $\mathbb{E}_2$ into a positive and a negative side, where the positive side is to the left of the boundary. The boundary further splits $\mathbb{E}_2$ into a bounded and an unbounded side. Note that the circle can be degenerated, i.e. the squared radius may be zero.

#include <CGAL/Circle_2.h>

Creation

CGAL_Circle_2<R> circle(CGAL_Point_2<R> center,
R::FT squared_radius,
CGAL_Orientation orientation = CGAL_COUNTERCLOCKWISE)

introduces a variable *circle* of type *CGAL_Circle_2<R>*. It is initialized to the circle with center *center*, squared radius *squared_radius* and orientation *orientation*.
Precondition: *squared_radius* ≥ 0, and also *orientation* $\neq$ *CGAL_COLLINEAR*.

CGAL_Circle_2<R> circle(CGAL_Point_2<R> p, CGAL_Point_2<R> q, CGAL_Point_2<R> r);

introduces a variable *circle* of type *CGAL_Circle_2<R>*. It is initialized to the unique circle which passes through the points p, q and r. The orientation of the circle is the orientation of the point triple p, q, r.
Precondition: p, q, and r are not collinear.

CGAL_Circle_2<R> circle(CGAL_Point_2<R> p,

CGAL_Point_2<R> q,
CGAL_Orientation orientation = CGAL_COUNTERCLOCKWISE)

introduces a variable *circle* of type *CGAL_Circle_2<R>*. It is initialized to the circle with diameter $\overline{pq}$ and orientation *orientation*.
Precondition: *orientation* $\neq$ *CGAL_COLLINEAR*.

CGAL_Circle_2<R> circle(CGAL_Point_2<R> center,
CGAL_Orientation orientation = CGAL_COUNTERCLOCKWISE)

introduces a variable *circle* of type *CGAL_Circle_2<R>*. It is initialized to the circle with center *center*, squared radius zero and orientation *orientation*.
Precondition: *orientation* $\neq$ *CGAL_COLLINEAR*.
Postcondition: *circle.is_degenerate()* = *true*.

Access Functions

CGAL_Point_2<R>	*circle.center()*	returns the center of *circle*.
R::FT	*circle.squared_radius()*	returns the squared radius of *circle*.
CGAL_orientation	*circle.orientation()*	returns the orientation of *circle*.

Equality Tests

bool	*circle == circle2*	returns *true*, iff *circle* and *circle2* are equal, i.e. if they have the same center, same squared radius and same orientation.
bool	*circle != circle2*	returns *true*, iff *circle* and *circle2* are not equal.

Predicates

CGAL_Oriented_side *circle.oriented_side(CGAL_Point_2<R> p)*

This returns the constant *CGAL_ON_ORIENTED_BOUNDARY*, *CGAL_ON_POSITIVE_SIDE*, or *CGAL_ON_NEGATIVE_SIDE*, iff *p* lies on the boundary, properly on the positive side, or properly on the negative side of *circle*, resp.

CGAL_Bounded_side *circle.bounded_side(CGAL_Point_2<R> p)*

returns *CGAL_ON_BOUNDED_SIDE*, *CGAL_ON_BOUNDARY*, or *CGAL_ON_UNBOUNDED_SIDE* iff *p* lies properly inside, on the boundary, or properly outside of *circle*, resp.

bool *circle.has_on_positive_side(Point p)*

returns *true*, iff *p* lies properly on the positive side of *circle*.

bool *circle.has_on_negative_side(Point p)*

returns *true*, iff *p* lies properly on the negative side of *circle*.

bool *circle.has_on_boundary(Point p)*

returns *true*, iff *p* lies on the boundary of *circle*.

bool *circle.has_on_bounded_side(Point p)*

returns *true*, iff *p* lies properly inside *circle*.

bool *circle.has_on_unbounded_side(Point p)*

returns *true*, iff *p* lies properly outside of *circle*.

bool *circle.is_degenerate()*

returns *true*, iff *circle* is degenerate, i.e. if *circle* has squared radius zero.

Miscellaneous

CGAL_Circle_2<R> *circle.opposite()*

returns the circle with the same center and squared radius as *circle* but with opposite orientation.

CGAL_Circle_2<R> *circle.orthogonal_transform(CGAL_Aff_transformation_2<R> at)*

returns the circle obtained by applying *at* on *circle*.
Precondition: *at* is an orthogonal transformation.

CGAL_Bbox_2 *circle.bbox()*

returns a bounding box containing *circle*.

Implementation

A circle is internally represented as a point, a squared radius and an orientation.

Chapter 10

2D Geometric Predicates

10.1 Order Type Predicates

CGAL_Orientation   *CGAL_orientation(CGAL_Point_2<R> p,*
CGAL_Point_2<R> q,
CGAL_Point_2<R> r)

returns *CGAL_LEFTTURN*, if r lies to the left of the oriented line l defined by p and q, returns *CGAL_RIGHTTURN* if r lies to the right of l, and returns *CGAL_COLLINEAR* if r lies on l.

The following functions return *true* or *false* depending on whether the orientation of three points is equal to one of the constants mentioned in 3.1.

bool   *CGAL_leftturn(CGAL_Point_2<R> p,*
CGAL_Point_2<R> q,
CGAL_Point_2<R> r)

bool   *CGAL_rightturn(CGAL_Point_2<R> p,*
CGAL_Point_2<R> q,
CGAL_Point_2<R> r)

bool   *CGAL_collinear(CGAL_Point_2<R> p,*
CGAL_Point_2<R> q,
CGAL_Point_2<R> r)

10.2 The Incircle Test

Instead of constructing a circle and performing the test if a given point lies inside or outside you might use the following predicate:

CGAL_Bounded_side *CGAL_side_of_bounded_circle(CGAL_Point_2<R> p, CGAL_Point_2<R> q, CGAL_Point_2<R> r, CGAL_Point_2<R> test)*

returns the relative position of point *test* to the circle defined by p, q and r. The order of the points p, q and r does not matter.
Precondition: p, q and r are not collinear.

CGAL_Oriented_side *CGAL_side_of_oriented_circle(CGAL_Point_2<R> p, CGAL_Point_2<R> q, CGAL_Point_2<R> r, CGAL_Point_2<R> test)*

returns the relative position of point *test* to the oriented circle defined by p, q and r. The order of the points p, q and r is important, since it determines the orientation of the implicitly constructed circle.
Precondition: p, q and r are not collinear.

10.3 Comparison of Coordinates of Points

In order to check if two points have the same x or y coordinate we provide the following functions. They allow to write code that does not depend on the representation type.

#include <CGAL/predicates_on_points_2.h>

bool *CGAL_x_equal(CGAL_Point_2<R> p, CGAL_Point_2<R> q)*

returns *true*, iff p and q have the same x-coordinate.

bool *CGAL_y_equal(CGAL_Point_2<R> p, CGAL_Point_2<R> q)*

returns *true*, iff p and q have the same y-coordinate.

The above functions are decision versions of the following comparison functions returning a *CGAL_Comparison_result*.

CGAL_Comparison_result *CGAL_compare_x(CGAL_Point_2<R> p, CGAL_Point_2<R> q)*

CGAL_Comparison_result *CGAL_compare_y(CGAL_Point_2<R> p, CGAL_Point_2<R> q)*

CGAL offers the same functions for points given implicitly as intersection of two lines. We provide these functions because we can provide better code for the test, than simply computing the intersection and calling the respective function for points.

Precondition: Lines that define an intersection point may not be parallel.

#include <CGAL/predicates_on_lines_2.h>

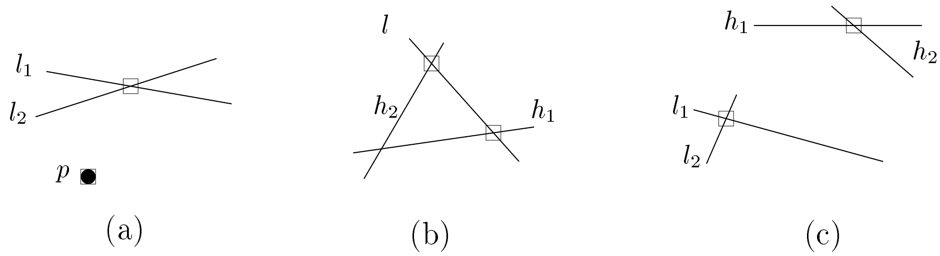


Figure 10.1: Comparison of the x or y-coordinates of the (implicitly given) points in the boxes.

```
CGAL_Comparison_result    CGAL_compare_x( CGAL_Point_2<R> p,
                                          CGAL_Line_2<R> l1,
                                          CGAL_Line_2<R> l2)
```

compares the x-coordinates of p and the intersection of lines $l1$ and $l2$ (Figure 10.1 (a)).

```
CGAL_Comparison_result    CGAL_compare_x( CGAL_Line_2<R> l,
                                          CGAL_Line_2<R> h1,
                                          CGAL_Line_2<R> h2)
```

compares the x-coordinates of the intersection of line l with line $h1$ and with line $h2$ (Figure 10.1 (b)).

```
CGAL_Comparison_result    CGAL_compare_x( CGAL_Line_2<R> l1,
                                          CGAL_Line_2<R> l2,
                                          CGAL_Line_2<R> h1,
                                          CGAL_Line_2<R> h2)
```

compares the x-coordinates of the intersection of lines $l1$ and $l2$ and the intersection of lines $h1$ and $h2$ (Figure 10.1 (c)).

```
CGAL_Comparison_result    CGAL_compare_y( CGAL_Point_2<R> p,
                                          CGAL_Line_2<R> l1,
                                          CGAL_Line_2<R> l2)
```

compares the y-coordinates of p and the intersection of lines $l1$ and $l2$ (Figure 10.1 (a)).

```
CGAL_Comparison_result    CGAL_compare_y( CGAL_Line_2<R> l,
                                          CGAL_Line_2<R> h1,
                                          CGAL_Line_2<R> h2)
```

compares the y-coordinates of the intersection of line l with line $h1$ and with line $h2$ (Figure 10.1 (b)).

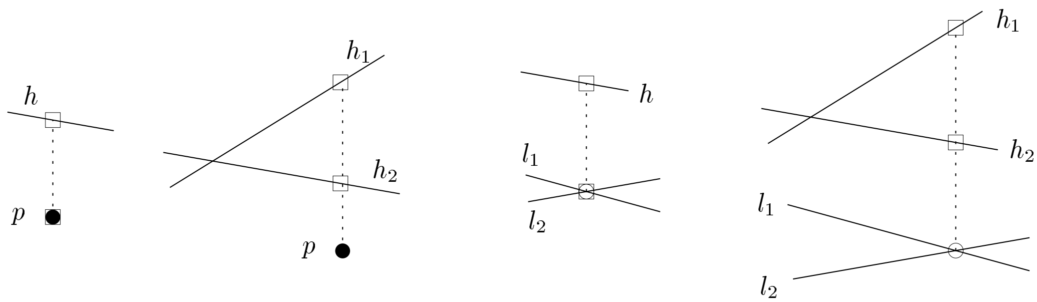
```
CGAL_Comparison_result    CGAL_compare_y( CGAL_Line_2<R> l1,
                                          CGAL_Line_2<R> l2,
                                          CGAL_Line_2<R> h1,
```

CGAL_Line_2<R> h2)

compares the y-coordinates of the intersection of lines $l1$ and $l2$ and the intersection of lines $h1$ and $h2$ (Figure 10.1 (c)).

The following functions compare the y coordinate of an point (that may be given implicitly) with a line.

Precondition: If the point is given as an intersection of two lines these lines may not be parallel. Lines where points are projected on may not be vertical.



(d) (e) (f) (g)

Figure 10.2: Comparison of the y-coordinates of the (implicitly given) points in the boxes, at an x-coordinate. The x-coordinate is either given explicitly (disc) or implicitly (circle).

CGAL_Comparison_result   CGAL_compare_y_at_x(CGAL_Point_2<R> p, CGAL_Line_2<R> h)

compares the y-coordinates of p and the vertical projection of p on h (Figure 10.2 (d)).

CGAL_Comparison_result   CGAL_compare_y_at_x(CGAL_Point_2<R> p,
CGAL_Line_2<R> h1,
CGAL_Line_2<R> h2)

This function compares the y-coordinates of the vertical projection of p on *h1* and on *h2* (Figure 10.2 (e)).

CGAL_Comparison_result   CGAL_compare_y_at_x(CGAL_Line_2<R> l1,
CGAL_Line_2<R> l2,
CGAL_Line_2<R> h)

Let p be the intersection of lines $l1$ and $l2$. This function compares the y-coordinates of p and the vertical projection of p on h (Figure 10.2 (f)).

CGAL_Comparison_result   CGAL_compare_y_at_x(CGAL_Line_2<R> l1,
CGAL_Line_2<R> l2,
CGAL_Line_2<R> h1,
CGAL_Line_2<R> h2)

Let p be the intersection of lines $l1$ and $l2$. This function compares the y-coordinates of the vertical projection of p on *h1* and on *h2* (Figure 10.2 (g)).

For lexicographical comparison CGAL provides

CGAL_Comparison_result *CGAL_compare_lexicographically_xy(CGAL_Point_2<R> p, CGAL_Point_2<R> q)*

Compares the Cartesian coordinates of points p and q lexicographically in xy order: first x-coordinates are compared, if they are equal, y-coordinates are compared.

In addition, CGAL provides the following comparison functions returning *true* or *false* depending on the result of *CGAL_compare_lexicographically_xy(p,q)*.

bool *CGAL_lexicographically_xy_smaller_or_equal(CGAL_Point_2<R> p, CGAL_Point_2<R> q)*

bool *CGAL_lexicographically_xy_smaller(CGAL_Point_2<R> p, CGAL_Point_2<R> q)*

bool *CGAL_lexicographically_xy_larger_or_equal(CGAL_Point_2<R> p, CGAL_Point_2<R> q)*

bool *CGAL_lexicographically_xy_larger(CGAL_Point_2<R> p, CGAL_Point_2<R> q)*

See Also

Distance comparisons, Section 13.2.

Chapter 11

2D Transformations

CGAL provides affine transformations. The 2D primitive objects in CGAL are closed under affine transformations except for iso-oriented objects, bounding boxes, and circles.

The general form of an affine transformation is based on homogeneous representation of points. Thereby all transformations can be realized by matrix multiplication.

Since the general form is based on the homogeneous representation, a transformation matrix multiplication by a scalar does not change the represented transformation. Therefore, any transformation represented by a matrix with rational entries can be represented by a transformation matrix with integer entries as well by multiplying the matrix with the common denominator of the rational entries. Hence it is sufficient to have number type *R::RT* for the entries of an affine transformation.

CGAL offers several specialized affine transformations. Different constructors are provided to create them. They are parameterized with a symbolic name to denote the transformation type, followed by additional parameters. The symbolic name tags solve ambiguities in the function overloading and they make the code more readable, i.e. what type of transformation is created. These name tags are constants of an appropriate type.

const CGAL_Translation	*CGAL_TRANSLATION;*
const CGAL_Rotation	*CGAL_ROTATION;*
const CGAL_Scaling	*CGAL_SCALING;*

11.1 2D Affine Transformation (*CGAL_Aff_transformation_2*<*R*>)

Definition

Since two-dimensional points have three homogeneous coordinates we have a 3×3 matrix (m_{ij}). Following C-style, the indices start at zero.

If the homogeneous representations are normalized such that the homogenizing coordinate is 1, then the upper left 2×2 matrix realizes linear transformations and in the matrix form of a translation, the translation vector $(v_0, v_1, 1)$ appears in the last column of the matrix. In the normalized case, entry hw is always 1. Entries m_{20} and m_{21} are always zero and therefore do not appear in the constructors.

#include <CGAL/Aff_transformation_2.h>

Creation

CGAL_Aff_transformation_2<R> t(const CGAL_Translation, CGAL_Vector_2<R> v);

introduces a translation by a vector v.

CGAL_Aff_transformation_2<R> t(const CGAL_Rotation,
CGAL_Direction_2<R> d,
R::RT num,
R::RT den = RT(1))

approximates the rotation over the angle indicated by direction d, such that the differences between the sines and cosines of the rotation given by d and the approximating rotation are at most num/den each.
Precondition: $num/den > 0$.

CGAL_Aff_transformation_2<R> t(const CGAL_Rotation,
R::RT sine_rho,
R::RT cosine_rho,
R::RT hw = RT(1))

introduces a rotation by the angle rho.
Precondition: $sine\_rho^2 + cosine\_rho^2 == hw^2$

.

CGAL_Aff_transformation_2<R> t(const CGAL_Scaling, R::RT s, R::RT hw = RT(1));

introduces a scaling by a scale factor s/hw.

CGAL_Aff_transformation_2<R> t(R::RT m00,
R::RT m01,
R::RT m02,
R::RT m10,
R::RT m11,
R::RT m12,
R::RT hw = RT(1))

introduces a general affine transformation in the 3×3 matrix form $\begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ 0 & 0 & hw \end{pmatrix}$. The sub-matrix $\frac{1}{hw}\begin{pmatrix} m_{00} & m_{01} \\ m_{10} & m_{11} \end{pmatrix}$ contains the scaling and rotation information, the vector $\frac{1}{hw}\begin{pmatrix} m_{02} \\ m_{12} \end{pmatrix}$ contains the translational part of the transformation.

CGAL_Aff_transformation_2<R> *t(R::RT m00, R::RT m01, R::RT m10, R::RT m11, R::RT hw = RT(1))*

introduces a general linear transformation $\begin{pmatrix} m_{00} & m_{01} & 0 \\ m_{10} & m_{11} & 0 \\ 0 & 0 & hw \end{pmatrix}$, i.e. there is no translational part.

Operations

The main thing to do with transformations is to apply them on geometric objects. Each class *CGAL_Class_2<R>* representing a geometric object has a member function:

CGAL_Class_2<R> transform(CGAL_Aff_transformation_2<R> t).

The transformation classes provide a member function *transform()* for points, vectors, directions, and lines:

CGAL_Point_2<R>	*t.transform(CGAL_Point_2<R> p)*
CGAL_Vector_2<R>	*t.transform(CGAL_Vector_2<R> p)*
CGAL_Direction_2<R>	*t.transform(CGAL_Direction_2<R> p)*
CGAL_Line_2<R>	*t.transform(CGAL_Line_2<R> p)*

CGAL provides function operators for these member functions:

CGAL_Point_2<R>	*t(CGAL_Point_2<R> p)*
CGAL_Vector_2<R>	*t(CGAL_Vector_2<R> p)*
CGAL_Direction_2<R>	*t(CGAL_Direction_2<R> p)*
CGAL_Line_2<R>	*t(CGAL_Line_2<R> p)*

CGAL_Aff_transformation_2<R>	$t * s$	composes two affine transformations.
CGAL_Aff_transformation_2<R>	*t.inverse()*	gives the inverse transformation.
bool	*t.is_even()*	returns *true*, if the transformation is not reflecting, i.e. the determinant of the involved linear transformation is non-negative.
bool	*t.is_odd()*	returns *true*, if the transformation is reflecting.

The matrix entries of a matrix representation of a *CGAL_Aff_transformation_2*<*R*> can be accessed trough the following member functions:

FT	*t.cartesian(int i, int j)*	
FT	*t.m(int i, int j)*	returns entry m_{ij} in a matrix representation in which m_{22} is 1.
RT	*t.homogeneous(int i, int j)*	
RT	*t.hm(int i, int j)*	returns entry m_{ij} in some fixed matrix representation.

For affine transformations no I/O operators are defined.

Implementation

Depending on the constructor we have different internal representations. This approach uses less memory and the transformation can be applied faster.

Affine transformations offer no *transform()* member function for complex objects because they are defined in terms of points vectors and directions. As the internal representation of a complex object is private the transformation code should go there.

Example

```
typedef CGAL_Cartesian<double> RepClass;
typedef CGAL_Aff_transformation_2<RepClass> Transformation;
typedef CGAL_Point_2<RepClass> Point;
typedef CGAL_Vector_2<RepClass> Vector;
typedef CGAL_Direction_2<RepClass> Direction;

Transformation rotate(CGAL_ROTATION, sin(pi), cos(pi));
Transformation rational_rotate(CGAL_ROTATION,Direction(1,1), 1, 100);
Transformation translate(CGAL_TRANSLATION, Vector(-2, 0));
Transformation scale(CGAL_SCALING, 3);

Point q(0, 1);
q = rational_rotate(q);

Point p(1, 1);

p = rotate(p);

p = translate(p);

p = scale(p);
```

The same would have been achieved with

```
Transformation transform = scale * (translate * rotate);
p = transform(Point(1.0, 1.0));
```

Chapter 12

Intersections

An important relation between geometrical objects is the intersection relation. Two objects *obj1* and *obj2* intersect if there is a point p that is part of both *obj1* and *obj2*. The intersection region of those two objects is defined as the set of all points p that are part of both *obj1* and *obj2*.

Note that for objects like triangles and polygons that enclose a bounded region, this region is part of the object. If a segment lies completely inside a triangle, then those two objects intersect and the intersection region is the complete segment.

In the following sections we describe two families of functions. One that checks whether two objects intersect; one that computes the intersection region.

There are two ways to use those functions. The simplest way is to include the header file `CGAL/intersections.h`. Here all intersection routines are declared.

The drawback of this approach is that a lot is included, which results in long compilation times. It is also possible to include only the intersections that are of interest. The naming scheme of the header files is as follows. Intersections of types *CGAL_Type1*<*R*> and *CGAL_Type2*<*R*> are declared in header file `CGAL/Type1_Type2_intersection.h`. So, intersection routines of segments and lines in 2D are declared in `CGAL/Segment_2_Line_2_intersection.h` The order of the type names does not matter. It is also possible to include `CGAL/Line_2_Segment_2_intersection.h`

For intersections of two objects of the same type, the type name should be mentioned twice:
`#include <CGAL/Segment_2_Segment_2_intersection.h>`

Every intersection header file declares both the checking routines and the routines to compute the intersection region.

12.1 Checking

In order to check if two objects of type *Type1* and *Type2* intersect, there are routines of the following form:

bool *CGAL_do_intersect(Type1*<*R*> *obj1, Type2*<*R*> *obj2)*

The types *Type1* and *Type2* can be any of the following:

- *CGAL_Point_2*
- *CGAL_Line_2*
- *CGAL_Ray_2*
- *CGAL_Segment_2*
- *CGAL_Triangle_2*
- *CGAL_Iso_rectangle_2*

12.2 Computing the intersection region

The intersection region of two geometrical objects of type *Type1* and *Type2* can be computed with the following family of routines:

CGAL_Object *CGAL_intersection(Type1<R> obj1, Type2<R> obj2)*

The return value is *CGAL_Object*. This is a special kind of object that can contain an object of any type. A previous section describes the use of this class.

The types *Type1* and *Type2* can be any of the following:

- *CGAL_Point_2*
- *CGAL_Line_2*
- *CGAL_Ray_2*
- *CGAL_Segment_2*
- *CGAL_Triangle_2*
- *CGAL_Iso_rectangle_2*

The family of routines that compute the intersection region is not as complete as the family of intersection checking routines. Not all combinations of types are possible. For instance, for more complicated types like polygons and discs those routines are not available. The result must be representable by a single geometric object. The intersection of a line segment with a polygon can result in several line segments. The intersection of a triangle and a disc can result in an object bounded by curved and straight edges. The implementation of boolean operations in the basic library provides more functionality for computing intersections of this kind.

Example

The following example demonstrates the most common use of intersection routines.

```
#include <CGAL/Segment_2_Line_2_intersection.h>

template <class R>
void foo(CGAL_Segment_2<R> seg, CGAL_Line_2<R> line)
{
```

```
    CGAL_Object result;
    CGAL_Point_2<R> ipoint;
    CGAL_Segment_2<R> iseg;

    result = CGAL_intersection(seg, line);
    if (CGAL_assign(ipoint, result)) {
        // handle the point intersection case.
    } else
        if (CGAL_assign(iseg, result)) {
            // handle the segment intersection case.
        } else {
            // handle the no intersection case.
        }
}
```

12.2.1 Possible Result Values

Here we describe the types that can be contained in the *CGAL_Object* return value for every possible pair of geometric objects, when the result is non-empty.

Table 12.1: All available intersection computations

type A	type B	return type
CGAL_Line_2	*CGAL_Line_2*	*CGAL_Point_2* *CGAL_Line_2*
CGAL_Segment_2	*CGAL_Line_2*	*CGAL_Point_2* *CGAL_Segment_2*
CGAL_Segment_2	*CGAL_Segment_2*	*CGAL_Point_2* *CGAL_Segment_2*
CGAL_Ray_2	*CGAL_Line_2*	*CGAL_Point_2* *CGAL_Ray_2*
CGAL_Ray_2	*CGAL_Segment_2*	*CGAL_Point_2* *CGAL_Segment_2*
CGAL_Ray_2	*CGAL_Ray_2*	*CGAL_Point_2* *CGAL_Segment_2* *CGAL_Ray_2*
CGAL_Triangle_2	*CGAL_Line_2*	*CGAL_Point_2* *CGAL_Segment_2*
CGAL_Triangle_2	*CGAL_Segment_2*	*CGAL_Point_2* *CGAL_Segment_2*
CGAL_Triangle_2	*CGAL_Ray_2*	*CGAL_Point_2* *CGAL_Segment_2*
CGAL_Triangle_2	*CGAL_Triangle_2*	*CGAL_Point_2* *CGAL_Segment_2* *CGAL_Triangle_2* *vector*<*CGAL_Point_2*>
CGAL_Iso_rectangle_2	*CGAL_Line_2*	*CGAL_Point_2* *CGAL_Segment_2*
CGAL_Iso_rectangle_2	*CGAL_Segment_2*	*CGAL_Point_2* *CGAL_Segment_2*

continued

CGAL_Iso_rectangle_2	*CGAL_Ray_2*	*CGAL_Point_2* *CGAL_Segment_2*
CGAL_Iso_rectangle_2	*CGAL_Iso_rectangle_2*	*CGAL_Iso_rectangle_2*

Chapter 13

Squared Distances

13.1 Introduction

There is a family of functions called *CGAL_squared_distance* that compute the square of the Euclidean distance between two geometric objects.

The squared distance between two two-dimensional points *p1* and *p2* is defined as $d_x^2 + d_y^2$, where $d_x \equiv$ *p2.x()-p1.x()* and $d_y \equiv$ *p2.y()-p1.y()*.

For arbitrary two-dimensional geometric objects *obj1* and *obj2* the squared distance is defined as the minimal *CGAL_squared_distance(p1, p2)*, where *p1* is a point of *obj1* and *p2* is a point of *obj2*. Note that for objects like triangles and polygons that have an inside (a bounded region), this inside is part of the object. So, the squared distance from a point inside a triangle to that triangle is zero, not the squared distance to the closest edge of the triangle.

The general format of the functions is:

R::FT *CGAL_squared_distance(Type1<R> obj1, Type2<R> obj2)*

where the types *Type1* and *Type2* can be any of the following:

- *CGAL_Point_2*
- *CGAL_Line_2*
- *CGAL_Ray_2*
- *CGAL_Segment_2*
- *CGAL_Triangle_2*

Those routines are defined in the header file *CGAL/squared_distance.h*.

13.1.1 Why the square?

There are routines that compute the square of the Euclidean distance, but no routines that compute the distance itself. Why?

First of all, the two values can be derived from each other quite easily (by taking the square root or taking the square). So, supplying only the one and not the other is only a minor inconvenience for the user.

Second, often either value can be used. This is for example the case when (squared) distances are compared.

Third, the library wants to stimulate the use of the squared distance instead of the distance. The squared distance can be computed in more cases and the computation is cheaper. We do this by not providing the perhaps more natural routine,

The problem of a distance routine is that it needs the *sqrt* operation. This has two drawbacks.

- The *sqrt* operation can be costly. Even if it is not very costly for a specific number type and platform, avoiding it is always cheaper.
- There are number types on which no *sqrt* operation is defined, especially integer types and rationals.

13.2 Distance Comparisons

Instead of computing distances and comparing them the following predicates can be used to that:

#include <CGAL/distance_predicates_2.h>

CGAL_Comparison_result *CGAL_cmp_dist_to_point(CGAL_Point_2<R> p, CGAL_Point_2<R> q, CGAL_Point_2<R> r)*

compares the distances of points q and r to point p. returns *CGAL_SMALLER*, iff q is closer to p as r, *CGAL_LARGER*, iff r is closer to p as q, and *CGAL_EQUAL* otherwise.

bool *CGAL_has_larger_dist_to_point(CGAL_Point_2<R> p, CGAL_Point_2<R> q, CGAL_Point_2<R> r)*

returns *true* iff the distance between q and p is larger as the distance between r and p.

bool *CGAL_has_smaller_dist_to_point(CGAL_Point_2<R> p, CGAL_Point_2<R> q, CGAL_Point_2<R> r)*

returns *true* iff the distance between q and p is smaller as the distance between r and p.

The following compares the signed distances of two points and an oriented line. The sign of the distance of a point to an oriented line is positive (negative) iff the point lies on the positive (negative) side of the line, and zero otherwise.

CGAL_Comparison_result *CGAL_cmp_signed_dist_to_line(CGAL_Line_2<R> l, CGAL_Point_2<R> p, CGAL_Point_2<R> q)*

returns *CGAL_LARGER* iff the signed distance of p and q is larger as the signed distance of q and l, *CGAL_SMALLER*, iff it is smaller, and *CGAL_EQUAL* iff both are equal.

bool *CGAL_has_larger_signed_dist_to_line(CGAL_Line_2<R> l, CGAL_Point_2<R> p, CGAL_Point_2<R> q)*

returns *true* iff the signed distance of p and q is larger as the signed distance of q and l.

bool *CGAL_has_smaller_signed_dist_to_line(CGAL_Line_2<R> l, CGAL_Point_2<R> p, CGAL_Point_2<R> q)*

returns *true* iff the signed distance of p and q is smaller as the signed distance of q and l.

The following functions corresponds to the preceding ones with the exception that the line is implicitly given by the first two points p and q. The points whose signed distance to the line is compared are r and s.

Precondition: Points defining the line are not equal.

CGAL_Comparison_result *CGAL_cmp_signed_dist_to_line(CGAL_Point_2<R> p, CGAL_Point_2<R> q, CGAL_Point_2<R> r, CGAL_Point_2<R> s)*

bool *CGAL_has_smaller_signed_dist_to_line(CGAL_Point_2<R> p, CGAL_Point_2<R> q, CGAL_Point_2<R> r, CGAL_Point_2<R> s)*

bool *CGAL_has_larger_signed_dist_to_line(CGAL_Point_2<R> p, CGAL_Point_2<R> q, CGAL_Point_2<R> r, CGAL_Point_2<R> s)*

Chapter 14

The 3D Kernel: an Overview

This chapter presents an overview of the three-dimensional kernel of CGAL. The kernel consists of elementary 3D objects (such as points, lines, planes, etc.) and of predicates on these objects (such as coordinate comparison, orientation, etc.). We also provide affine transformations on these objects. These classes are all templated by the representation class R, which is currently either *CGAL_Cartesian*<*FT*> or *CGAL_Homogeneous*<*RT*> for a so-called field tupe FT, and ring type RT, see Chapter 2.

14.1 Elementary 3D objects

CGAL provides points, vectors, and directions. A *point* is a point in the three-dimensional Euclidean space $\mathbb{E}_3$, a *vector* is the difference of two points p_3, p_1 and denotes the direction and the distance from p_1 to p_3 in the vector space $\mathbb{R}^3$, and a *direction* represents the family of vectors that are positive multiples of each other. Their interface is described in Chapter 15.

CGAL_Point_3<*R*>
CGAL_Vector_3<*R*>
CGAL_Direction_3<*R*>

Lines in CGAL are directed. Any two points on a line induce an orientation of this line. A ray is semi-infinite interval on a line, and this line is oriented from the finite endpoint of this interval towards any other point in this interval. A segment is a bounded interval on a directed line, and the endpoints are ordered so that they induce a direction which is the same as that of the line. Their interface is described in Chapter 16.

CGAL_Line_3<*R*>
CGAL_Ray_3<*R*>
CGAL_Segment_3<*Re*>

Planes are affine subspaces of dimension two, passing through three points, or a point and a line, ray, or segment. CGAL provides a correspondence between the ambient space $\mathbb{E}^3$ and the embedding of $\mathbb{E}^2$ in that space. Their interface is described in Chapter 17.

CGAL_Plane_3<*R*>

Next we introduce the simplices triangle and tetrahedron. More complex polyhedra can be obtained from the basic library (*CGAL_Polyhedron_3*), so they are not part of the kernel. The simplex interfaces are described in Chapter 18.

CGAL_Triangle_3<R>
CGAL_Tetrahedron_3<R>

14.2 Predicates and functions

For testing where a point p lies with respect to a plane defined by three points q, r and s, one may be tempted to construct the line *CGAL_Plane_3<R>(q,r,s)* and use the method *oriented_side(p)*. This may pay off if many tests with respect to the plane are made. Nevertheless, unless the number type is exact, the constructed plane is only approximated, and round-off errors may lead *oriented_side(p)* to return an orientation which is different from the orientation of p, q, r, and s. This is the well-known problem of *robustness*.

In CGAL, we provide predicates in which such geometric decisions are made directly with a reference to the input points p, q, r, s, without an intermediary object like a plane. This enables to use exact predicates that are cheaper than exact number types, a concept that has been the focus of much research recently in computational geometry. For the above test, the recommended way to get the result is to use *CGAL_orientation(p,q,r,s)*.

Consequently, we propose the most common predicates in Chapter 20. They use the elementary classes of the 3D kernel.

Finally, affine transformations allow to generate new object instances under arbitrary affine transformations. These transformations include translations, rotations and scaling. Most of the classes above have a member function *transform(CGAL_Aff_transformation t)* which applies the transformation to the object instance. The interface of the transformation class is described in Chapter 21.

CGAL_Aff_transformation_3<R>

Chapter 15

3D Point, Vector and Direction

As explained in the chapter about two-dimensional objects why we distinguish between points, vectors and directions. This chapter will only give the functions that can be applied on such objects.

15.1 3D Point (*CGAL_Point_3*<*R*>)

Definition

An object of the class *CGAL_Point_3* is a point in the three-dimensional Euclidean space $\mathbb{E}_3$.

Remember that *R::RT* and *R::FT* denote a ring type and a field type. For the representation class *CGAL_Cartesian*<*T*> the two types are equivalent. For the representation class *CGAL_Homogeneous*<*T*> the ring type is *R::RT* == *T* and the field type is *R::FT* == *CGAL_Quotient*<*T*>.

#include <*CGAL/Point_3.h*>

Creation

CGAL_Point_3<*R*> *p(R::RT hx, R::RT hy, R::RT hz, R::RT hw = R::RT(1));*

introduces a point *p* initialized to $(hx/hw, hy/hw, hz/hw)$. If the third argument is not explicitly given it defaults to *R::RT(1)*.

Operations

bool	$p == q$	Test for equality: Two points are equal, iff their x, y and z coordinates are equal.
bool	$p\ != q$	Test for inequality.

There are two sets of coordinate access functions, namely to the homogeneous and to the Cartesian coordinates. They can be used independently from the chosen representation type *R*.

R::RT	*p.hx()*	returns the homogeneous x coordinate.
R::RT	*p.hy()*	returns the homogeneous y coordinate.
R::RT	*p.hz()*	returns the homogeneous z coordinate.
R::RT	*p.hw()*	returns the homogenizing coordinate.

Here come the Cartesian access functions. Note that you do not loose information with the homogeneous representation, because then the field type is a quotient.

R::FT	*p.x()*	returns the Cartesian x coordinate, that is hx/hw.
R::FT	*p.y()*	returns the Cartesian y coordinate, that is hy/hw.
R::FT	*p.z()*	returns the Cartesian z coordinate, that is hz/hw.

The following operations are for convenience and for making this point class compatible with code for higher dimensional points. Again they come in a Cartesian and homogeneous flavor.

R::RT	*p.homogeneous(int i)*	
		returns the i'th homogeneous coordinate of p, starting with 0. *Precondition*: $0 \leq i \leq 3$.
R::FT	*p.cartesian(int i)*	
		returns the i'th Cartesian coordinate of p, starting with 0. *Precondition*: $0 \leq i \leq 2$.
R::FT	*p*[*int i*]	returns *cartesian(i)*. *Precondition*: $0 \leq i \leq 2$.
int	*p.dimension()*	
		returns the dimension (the constant 3).
CGAL_Bbox_3	*p.bbox()*	returns a bounding box containing p.
CGAL_Point_3<R>	*p.transform(CGAL_Aff_transformation_3<R> t)*	
		returns the point obtained by applying t on p.

The following operations can be applied on points:

CGAL_Vector_3<R>	$p - q$	returns the difference vector between q and p.

CGAL_Point_3<R> p + *CGAL_Vector_3<R>* v

returns a point obtained by translating p by the vector v.

CGAL_Point_3<R> p − *CGAL_Vector_3<R>* v

returns a point obtained by translating p by the vector $-v$.

See Also

3D Geometric Predicates, Chapter 20.

15.2 Point Conversion

For convenience, CGAL provides functions for conversion of points between homogeneous and Cartesian representation.

#include <CGAL/cartesian_homogeneous_conversion.h>

Conversion from Cartesian representation to homogeneous representation with the same number type is straightforward. The homogenizing coordinate is set to 1, all other homogeneous coordinates are copied from the corresponding Cartesian coordinate.

CGAL_Point_3< CGAL_Homogeneous<RT> >

CGAL_cartesian_to_homogeneous(CGAL_Point_3< CGAL_Cartesian<RT> > cp)

converts 3d point *cp* with Cartesian representation into a 3d point with homogeneous representation with the same number type.

Conversion from homogeneous representation to Cartesian representation with the same number type involves division by the homogenizing coordinate.

CGAL_Point_3< CGAL_Cartesian<FT> >

CGAL_homogeneous_to_cartesian(CGAL_Point_3< CGAL_Homogeneous<FT> > hp)

converts 3d point *hp* with homogeneous representation into a 3d point with Cartesian representation with the same number type.

Since conversion involves division, concerning exactness, the correspondence is rather between homogeneous representation with number type RT and Cartesian representation with number type *CGAL_Quotient<RT>* than between representation with the same number type.

CGAL_Point_3< CGAL_Cartesian<CGAL_Quotient<RT> > >

CGAL_homogeneous_to_quotient_cartesian(CGAL_Point_3<CGAL_Homogeneous<RT> > hp)

converts 3d point *hp* with homogeneous representation with number type *RT* into a 3d point with Cartesian representation with number type *CGAL_Quotient<RT>*.

CGAL_Point_3< CGAL_Homogeneous<RT> >

CGAL_quotient_cartesian_to_homogeneous(CGAL_Point_3< CGAL_Cartesian< CGAL_Quotient<RT> > > cp)

converts 3d point *cp* with Cartesian representation with number type *CGAL_Quotient<RT>* into a 3d point with homogeneous representation with number type *RT*.

For the above functions, conversion from homogeneous representation to Cartesian representation with quotients is always exact. Conversion from Cartesian representation with quotients to homogeneous representation, however, might be inexact with some number types due to overflow or rounding in multiplications.

15.3 3D Vector (*CGAL_Vector_3*<*R*>)

Definition

An object of the class *CGAL_Vector_3* is a vector in the three-dimensional vector space $\mathbb{R}^3$. Geometrically spoken a vector is the difference of two points p_3, p_1 and denotes the direction and the distance from p_1 to p_3.

CGAL defines a symbolic constant *CGAL_NULL_VECTOR*. We will explicitly state where you can pass this constant as an argument instead of a vector initialized with zeros.

#include <CGAL/Vector_3.h>

Creation

CGAL_Vector_3<*R*> *v(R::RT hx, R::RT hy, R::FT hz, R::RT hw = R::RT(1));*

introduces a vector v initialized to $(hx/hw, hy/hw, hz/hw)$. If the third argument is not explicitly given it defaults to *R::RT(1)*.

Operations

bool	$v == w$	Test for equality: two vectors are equal, iff their x, y and z coordinates are equal. You can compare a vector with the *CGAL_NULL_VECTOR*.
bool	$v \,!= w$	Test for inequality. You can compare a vector with the *CGAL_NULL_VECTOR*.

There are two sets of coordinate access functions, namely to the homogeneous and to the Cartesian coordinates. They can be used independently from the chosen representation type *R*.

R::RT	*v.hx()*	returns the homogeneous x coordinate.
R::RT	*v.hy()*	returns the homogeneous y coordinate.
R::RT	*v.hz()*	returns the homogeneous z coordinate.
R::RT	*v.hw()*	returns the homogenizing coordinate.

Here come the Cartesian access functions. Note that you do not loose information with the homogeneous representation, because then the field type is a quotient.

R::FT	*v.x()*	returns the x-coordinate of v, that is hx/hw.
R::FT	*v.y()*	returns the y-coordinate of v, that is hy/hw.
R::FT	*v.z()*	returns the z coordinate of v, that is hz/hw.

The following operations are for convenience and for making the class *CGAL_Vector_3* compatible with code for higher dimensional vectors. Again they come in a Cartesian and homogeneous flavor.

R::RT	*v.homogeneous(int i)*	returns the i'th homogeneous coordinate of v, starting with 0. *Precondition*: $0 \leq i \leq 3$.
R::FT	*v.cartesian(int i)*	returns the i'th Cartesian coordinate of v, starting at 0. *Precondition*: $0 \leq i \leq 2$.
R::FT	*v*[*int i*]	returns *coordinate(i)*. *Precondition*: $0 \leq i \leq 2$.
int	*v.dimension()*	returns the dimension (the constant 3).
CGAL_Vector_3<*R*>	*v.transform(CGAL_Aff_transformation_3*<*R*> *t)*	returns the vector obtained by applying t on v.

The following operations can be applied on vectors:

CGAL_Vector_3<*R*>	$v + w$	Addition.
CGAL_Vector_3<*R*>	$v - w$	Subtraction.
CGAL_Vector_3<*R*>	$-v$	Negation.
R::FT	$v * w$	returns the scalar product (= inner product) of the two vectors.
CGAL_Vector_3<*R*>	v * *R::RT s*	Multiplication with a scalar from the right. Although it would be more natural, CGAL does not offer a multiplication with a scalar from the left. (This is due to problems of some compilers.)
CGAL_Vector_3<*R*>	v * *CGAL_Quotient*<*RT*> *s*	Multiplication with a scalar from the right.
CGAL_Vector_3<*R*>	v / *R::RT s*	Division by a scalar.
CGAL_Direction_3<*R*>	*v.direction()*	returns the direction of v.

15.4 3D Direction (*CGAL_Direction_3<R>*)

Definition

An object of the class *CGAL_Direction_3* is a vector in the three-dimensional vector space $\mathbb{R}^3$ where we forget about their length. They can be viewed as unit vectors, although there is no normalization internally, since this is error prone. Directions are used whenever the length of a vector does not matter. For example, you can ask for the direction orthogonal to an oriented plane, or the direction of an oriented line.

#include <CGAL/Direction_3.h>

Creation

CGAL_Direction_3<R> d(CGAL_Vector_3<R> v);

introduces a direction d initialised with the direction of vector v.

CGAL_Direction_3<R> d(R::RT x, R::RT y, R::RT z);

introduces a direction d initialised with the direction from the origin to the point with Cartesian coordinates (x, y, z).

Operations

R::RT	*d.delta(int i)*	returns the i'th value of the slope of d. *Precondition*: : $0 \leq i \leq 2$.
R::RT	*d.dx()*	returns the dx value of the slope of d.
R::RT	*d.dy()*	returns the dy value of the slope of d.
R::RT	*d.dz()*	returns the dz value of the slope of d.
bool	$d == e$	Test for equality.
bool	$d\ != e$	Test for inequality.
CGAL_Direction_3<R>	$-d$	The direction opposite to d.
CGAL_Vector_3<R>	*d.vector()*	returns a vector that has the same direction as d.
CGAL_Direction_3<R>	*d.transform(CGAL_Aff_transformation_3<R> t)*	returns the direction obtained by applying t on d.

15.5 Conversion between Points and Vectors

As in the two-dimensional case you subtract a point p from the symbolic constant *CGAL_ORIGIN* to obtain the locus vector of p. In order to obtain the point corresponding to a vector v you simply have to add v to *CGAL_ORIGIN*. See Section 5.5 for an example.

Chapter 16

3D Line, Ray and Segment

16.1 3D Line (*CGAL_Line_3*<*R*>)

Definition

An object l of the data type *CGAL_Line_3* is a directed straight line in the three-dimensional Euclidean space $\mathbb{E}_3$.

#include <*CGAL/Line_3.h*>

Creation

CGAL_Line_3<*R*> *l(CGAL_Point_3*<*R*> *p, CGAL_Point_3*<*R*> *q);*

introduces a line l passing through the points p and q. Line l is directed from p to q.

CGAL_Line_3<*R*> *l(CGAL_Point_3*<*R*> *p, CGAL_Direction_3*<*R*> *d);*

introduces a line l passing through point p with direction d.

Operations

bool	$l == h$	Test for equality: two lines are equal, iff they have a non empty intersection and the same direction.
bool	$l\; != h$	Test for inequality.
CGAL_Plane_3<*R*>	*l.perpendicular_plane(CGAL_Point_3*<*R*> *p)*	returns the plane perpendicular to l passing through p.

CGAL_Line_3<R>	*l.opposite()*	returns the line with opposite direction.
CGAL_Point_3<R>	*l.projection(CGAL_Point_3<R> p)*	returns the orthogonal projection of p on l.
CGAL_Point_3<R>	*l.point(int i)*	returns an arbitrary point on l. It holds *point(i) == point(j)*, iff *i==j*.
CGAL_Direction_3<R>	*l.direction()*	returns the direction of l.
bool	*l.is_degenerate()*	returns true if line l is degenerated to a point.
bool	*l.has_on(CGAL_Point_3<R> p)*	
CGAL_Line_3<R>	*l.transform(CGAL_Aff_transformation_3<R> t)*	returns the line obtained by applying t on a point on l and the direction of l.

16.2 3D Ray (*CGAL_Ray_3<R>*)

Definition

An object r of the data type *CGAL_Ray_3* is a directed straight ray in the three-dimensional Euclidean space $\mathbb{E}_3$. It starts in a point called the *source* of r and it goes to infinity.

#include <CGAL/Ray_3.h>

Creation

CGAL_Ray_3<R> r(CGAL_Point_3<R> p, Point_3 q);

introduces a ray r with source p and passing through point q.

CGAL_Ray_3<R> r(CGAL_Point_3<R> p, CGAL_Direction_3<R> d);

introduces a ray r with source p and with direction d.

Operations

bool	*r == h*	Test for equality: two rays are equal, iff they have the same source and the same direction.
bool	*r != h*	Test for inequality.
CGAL_Point_3<R>	*r.source()*	returns the source of r
CGAL_Point_3<R>	*r.point(int i)*	returns a point on r. *point(0)* is the source. *point(i)*, with $i > 0$, is different from the source. *Precondition*: $i \geq 0$.
CGAL_Direction_3<R>	*r.direction()*	returns the direction of r.
CGAL_Line_3<R>	*r.supporting_line()*	returns the line supporting r which has the same direction.
CGAL_Ray_3<R>	*r.opposite()*	returns the ray with the same source and the opposite direction.
bool	*r.is_degenerate()*	ray r is degenerate, if the source and the second defining point fall together (that is if the direction is degenerate).
bool	*r.has_on(CGAL_Point_3<R> p)*	A point is on r, iff it is equal to the source of r, or if it is in the interior of r.

CGAL_Ray_3<R> *r.transform(CGAL_Aff_transformation_3<R> t)*

returns the ray obtained by applying t on the source and on the direction of r.

Implementation

A ray is stored as a point and a direction.

16.3 3D Segment (*CGAL_Segment_3<R>*)

Definition

An object s of the data type *CGAL_Segment_3* is a directed straight line segment in the three-dimensional Euclidean space $\mathbb{E}_3$, i.e. a straight line segment $[p, q]$ connecting two points $p, q \in \mathbb{R}^3$. The segment is topologically closed, i.e. the end points belong to it. Point p is called the *source* and q is called the *target* of s. The length of s is the Euclidean distance between p and q. Note that there is only a function to compute the square of the length, because otherwise we had to perform a square root operation which is not defined for all number types, is expensive, and may not be exact.

#include <CGAL/Segment_3.h>

Creation

CGAL_Segment_3<R> s(CGAL_Point_3<R> p, CGAL_Point_3<R> q);

introduces a segment s with source p and target q. It is directed from the source towards the target.

Operations

bool	$s == q$	Test for equality: Two segments are equal, iff their sources and targets are equal.
bool	s != q	Test for inequality.
CGAL_Point_3<R>	*s.source()*	returns the source of s.
CGAL_Point_3<R>	*s.target()*	returns the target of s.
CGAL_Point_3<R>	*s.min()*	returns the point of $s$with smallest coordinate (lexicographically).
CGAL_Point_3<R>	*s.max()*	returns the point of $s$with largest coordinate (lexicographically).
CGAL_Point_3<R>	*s.vertex(int i)*	returns source or target of s: *vertex(0)* returns the source, *vertex(1)* returns the target. The parameter i is taken modulo 2, which gives easy access to the other vertex.
CGAL_Point_3<R>	*s.point(int i)*	returns *vertex(i)*.
CGAL_Point_3<R>	*s[int i]*	returns *vertex(i)*.
R::FT	*s.squared_length()*	returns the squared length of s.

CGAL_Direction_3<R>	*s.direction()*	returns the direction from source to target.
CGAL_Segment_3<R>	*s.opposite()*	returns a segment with source and target interchanged.
CGAL_Line_3<R>	*s.supporting_line()*	returns the line l passing through s. Line l has the same orientation as segment s, that is from the source to the target of s.
bool	*s.is_degenerate()*	segment s is degenerate, if source and target fall together.
bool	*s.has_on(CGAL_Point_3<R> p)*	A point is on s, iff it is equal to the source or target of s, or if it is in the interior of s.
CGAL_Bbox_3	*s.bbox()*	returns a bounding box containing s.
CGAL_Segment_3<R>	*s.transform(CGAL_Aff_transformation_3<R> t)*	returns the segment obtained by applying t on the source and the target of s.

Implementation

A segment is internally represented by two points.

Chapter 17

3D Plane

17.1 3D Plane (*CGAL_Plane_3*<*R*>)

Definition

An object h of the data type *CGAL_Plane_3* is an oriented plane in the three-dimensional Euclidean space $\mathbb{E}_3$. It is defined by the set of points with coordinates Cartesian (x, y, z) that satisfy the plane equation

$$h : \; a\,x + b\,y + c\,z + d = 0.$$

The plane splits $\mathbb{E}_3$ in a *positive* and a *negative side*.[1] A point p with Cartesian coordinates (px, py, pz) is on the positive side of h, iff $a\,px + b\,py + c\,pz + d > 0$, it is on the negative side, iff $a\,px + b\,py + c\,pz + d < 0$.

#include <*CGAL/Plane_3.h*>

Creation

CGAL_Plane_3<*R*> *h(R::RT a, R::RT b, R::RT c, R::RT d);*

introduces a plane h defined by the equation $a\,px + b\,py + c\,pz + d = 0$.

CGAL_Plane_3<*R*> *h(CGAL_Point_3*<*R*> *p, CGAL_Point_3*<*R*> *q, CGAL_Point_3*<*R*> *r);*

introduces a plane h passing through the points p, q and r. The plane is oriented such that p, q and r are oriented in a positive sense (that is counterclockwise) when seen from the positive side of h.

[1] See Chapter 3 for the definition of *CGAL_Oriented_side*.

CGAL_Plane_3<R> h(CGAL_Point_3<R> p, CGAL_Direction_3<R> d);

introduces a plane h that passes through point p and that has as an orthogonal direction equal to d.

CGAL_Plane_3<R> h(CGAL_Line_3<R> l, CGAL_Point_3<R> p);

introduces a plane h that is defined through the three points *l.point(0)*, *l.point(1)* and p.

CGAL_Plane_3<R> h(CGAL_Ray_3<R> r, CGAL_Point_3<R> p);

introduces a plane h that is defined through the three points *r.point(0)*, *r.point(1)* and p.

CGAL_Plane_3<R> h(CGAL_Segment_3<R> s, CGAL_Point_3<R> p);

introduces a plane h that is defined through the three points *s.source()*, *s.target()* and p.

Operations

bool	*h == h2*	Test for equality: two planes are equal, iff they have a non empty intersection and the same orientation.
bool	*h != h2*	Test for inequality.
R::RT	*h.a()*	returns the first coefficient of $\mathcal{H}$.
R::RT	*h.b()*	returns the second coefficient of $\mathcal{H}$.
R::RT	*h.c()*	returns the third coefficient of $\mathcal{H}$.
R::RT	*h.d()*	returns the fourth coefficient of $\mathcal{H}$.
CGAL_Line_3<R>	*h.perpendicular_line(CGAL_Point_3<R> p)*	returns the line that is perpendicular to h and that passes through point p. The line is oriented from the negative to the positive side of h.
CGAL_Plane_3<R>	*h.opposite()*	returns the plane with opposite orientation.
CGAL_Point_3<R>	*h.projection(CGAL_Point_3<R> p)*	returns the orthogonal projection of p on h.

CGAL_Point_3<R> h.point()

returns an arbitrary point on h.

CGAL_Vector_3<R> h.orthogonal_vector()

returns a vector that is orthogonal to h and that is directed to the positive side of h.

CGAL_Direction_3<R> h.orthogonal_direction()

returns the direction that is orthogonal to h and that is directed to the positive side of h.

CGAL_Vector_3<R> h.base1()

returns a vector that is orthogonal to *orthogonal_vector()*.

CGAL_Vector_3<R> h.base2()

returns a vector that is orthogonal to both *orthogonal_vector()* and *base1()*, such that *CGAL_orientation(point(), point() + base1(), point()+base2(), point() + orthogonal_vector())* is positive.

The following functions provide conversion between a plane and CGAL's two-dimensional space. The transformation is affine, but not necessarily an isometry. This means, the transformation preserves combinatorics, but not distances.

CGAL_Point_2<R> h.to_2d(CGAL_Point_3<R> p)

returns the image point of the projection of p under an affine transformation, which maps h onto the xy-plane, with the z-coordinate removed.

CGAL_Point_3<R> h.to_3d(CGAL_Point_2<R> p)

returns a point q, such that *to_2d(to_3d(p))* is equal to p.

CGAL_Oriented_side h.oriented_side(CGAL_Point_3<R> p)

returns either *CGAL_ON_ORIENTED_BOUNDARY*, *CGAL_ON_POSITIVE_SIDE*, or the constant *CGAL_ON_NEGATIVE_SIDE*, depending where point p is relative to the oriented plane h.

For convenience we provide the following boolean functions:

bool h.has_on(CGAL_Point_3<R> p)

bool h.has_on_boundary(CGAL_Point_3<R> p)

bool	*h.has_on_positive_side(CGAL_Point_3<R> p)*	
bool	*h.has_on_negative_side(CGAL_Point_3<R> p)*	
bool	*h.has_on(CGAL_Line_3<R> l)*	
bool	*h.has_on_boundary(CGAL_Line_3<R> l)*	
bool	*h.is_degenerate()*	Plane h is degenerate, if the coefficients a, b, and c of the plane equation are zero.
CGAL_Plane_3<R>	*h.transform(CGAL_Aff_transformation_3<R> t)*	returns the plane obtained by applying t on a point of h and the orthogonal direction of h.

Chapter 18

3D Simplices

18.1 3D Triangle (*CGAL_Triangle_3<R>*)

Definition

An object t of the class *CGAL_Triangle_3* is a triangle in the three-dimensional Euclidean space $\mathbb{E}_3$. As the triangle is not a full-dimensional object there is only a test whether a point lies on the triangle or not.

#include <CGAL/Triangle_3.h>

Creation

CGAL_Triangle_3<R> t(CGAL_Point_3<R> p, CGAL_Point_3<R> q, CGAL_Point_3<R> r);

introduces a triangle t with vertices p, q and r.

Operations

bool	*t == t2*	Test for equality: two triangles t and t_2 are equal, iff there exists a cyclic permutation of the vertices of $t2$, such that they are equal to the vertices of t.
bool	*t != t2*	Test for inequality.
CGAL_Point_3<R>	*t.vertex(int i)*	returns the i'th vertex modulo 3 of t.
CGAL_Point_3<R>	*t[int i]*	returns *vertex(int i)*.
bool	*t.is_degenerate()*	Triangle t is degenerate, if the vertices are collinear.

bool	*t.has_on(CGAL_Point_3<R> p)*	
		A point is on t, if it is on a vertex, an edge or the face of t.
CGAL_Bbox_3	*t.bbox()*	returns a bounding box containing t.
CGAL_Triangle_3<R>	*t.transform(CGAL_Aff_transformation_3<R> at)*	
		returns the triangle obtained by applying *at* on the three vertices of t.

18.2 3D Tetrahedron (*CGAL_Tetrahedron_3<R>*)

Definition

An object t of the class *CGAL_Tetrahedron_3* is an oriented tetrahedron in the three-dimensional Euclidean space $\mathbb{E}_3$.

It is defined by four vertices p_0, p_1, p_2 and p_3. The orientation of a tetrahedron is the orientation of its four vertices. That means it is positive when p_3 is on the positive side of the plane defined by p_0, p_1 and p_2.

The tetrahedron itself splits the space E_3 in a *positive* and a *negative* side.[1]

The boundary of a tetrahedron splits the space in two open regions, a bounded one and an unbounded one.

#include <CGAL/Tetrahedron_3.h>

Creation

CGAL_Tetrahedron_3<R> t(CGAL_Point_3<R> p0, CGAL_Point_3<R> p1, CGAL_Point_3<R> p2, CGAL_Point_3<R> p3)

introduces a tetrahedron t with vertices p_0, p_1, p_2 and p_3.

Operations

bool	*t == t2*	Test for equality: two tetrahedra are equal, iff there exists a cyclic permutation of the vertices of $t2$, such that they are equal to the vertices of t.
bool	*t != t2*	Test for inequality.
CGAL_Point_3<R>	*t.vertex(int i)*	returns the i'th vertex modulo 4 of t.
CGAL_Point_3<R>	*t[int i]*	returns *vertex(int i)*.
bool	*t.is_degenerate()*	Tetrahedron t is degenerate, if the vertices are coplanar.
CGAL_Orientation	*t.orientation()*	
CGAL_Oriented_side	*t.oriented_side(CGAL_Point_3<R> p)*	

[1] See Chapter 3 for the definition of *CGAL_Oriented_side*.

CGAL_Bounded_side	*t.bounded_side(CGAL_Point_3<R> p)*	

For convenience we provide the following boolean functions:

bool	*t.has_on_positive_side(CGAL_Point_3<R> p)*	
bool	*t.has_on_negative_side(CGAL_Point_3<R> p)*	
bool	*t.has_on_boundary(CGAL_Point_3<R> p)*	
bool	*t.has_on_bounded_side(CGAL_Point_3<R> p)*	
bool	*t.has_on_unbounded_side(CGAL_Point_3<R> p)*	
CGAL_Bbox_3	*t.bbox()*	returns a bounding box containing t.
CGAL_Tetrahedron_3<R>	*t.transform(CGAL_Aff_transformation_3<R> at)*	returns the tetrahedron obtained by applying at on the three vertices of t.

Chapter 19

3D Iso-oriented Objects

Bounding boxes are used to approximate objects and the only operations you can perform on them is to get their bounds and to check if two bounding boxes overlap. They are always axis-parallel. Just like the 2D bounding box, this class is not templated.

19.1 3D Bbox (*CGAL_Bbox_3*)

Definition

An object b of the class *CGAL_Bbox_3* is a bounding box in the three-dimensional Euclidean space $\mathbb{E}_3$. This class is not templated.

#include <CGAL/Bbox_3.h>

Creation

CGAL_Bbox_3 *b(double x_min, double y_min, double z_min, double x_max, double y_max, double z_max)*

introduces a bounding box b with lexicographically smallest corner point at *(xmin, ymin, zmin)* lexicographically largest corner point at *(xmax, ymax, zmax)*.

Operations

bool	$b == c$	Test for equality: two bounding boxes are equal, if the lower left and the upper right corners are equal.

double	*b.xmin()*	
double	*b.ymin()*	
double	*b.zmin()*	
double	*b.xmax()*	
double	*b.ymax()*	
double	*b.zmax()*	
CGAL_Bbox_3	$b + c$	returns a bounding box of b and c.
bool	*CGAL_do_overlap(bb1, bb2)*	

Chapter 20

3D Geometric Predicates

20.1 Order Type Predicates

#include <CGAL/predicates_on_points_3.h>

CGAL_Orientation *CGAL_orientation(CGAL_Point_3<R> p, CGAL_Point_3<R> q, CGAL_Point_3<R> r, CGAL_Point_3<R> s)*

returns *CGAL_POSITIVE*, if s lies on the positive side of the oriented plane h defined by p, q, and r, returns *CGAL_NEGATIVE* if s lies on the negative side of h, and returns *CGAL_COPLANAR* if s lies on h.

bool *CGAL_coplanar(CGAL_Point_3<R> p, CGAL_Point_3<R> q, CGAL_Point_3<R> r, CGAL_Point_3<R> s)*

returns *true*, if p, q, r, and s are coplanar.

bool *CGAL_collinear(CGAL_Point_3<R> p, CGAL_Point_3<R> q, CGAL_Point_3<R> r)*

returns *true*, if p, q, and r are collinear.

20.2 Comparison of Coordinates of Points

In order to check if two points have the same x, y, or z coordinate we provide the following functions. They allow to write code that does not depend on the representation type.

#include <CGAL/predicates_on_points_3.h>

bool *CGAL_x_equal(CGAL_Point_3<R> p, CGAL_Point_3<R> q)*

returns *true*, iff p and q have the same x-coordinate.

bool *CGAL_y_equal(CGAL_Point_3<R> p, CGAL_Point_3<R> q)*

returns *true*, iff p and q have the same y-coordinate.

bool *CGAL_z_equal(CGAL_Point_3<R> p, CGAL_Point_3<R> q)*

returns *true*, iff p and q have the same z-coordinate.

The above functions are decision versions of the following comparison functions returning a *CGAL_Comparison_result*.

CGAL_Comparison_result *CGAL_compare_x(CGAL_Point_3<R> p, CGAL_Point_3<R> q)*

CGAL_Comparison_result *CGAL_compare_y(CGAL_Point_3<R> p, CGAL_Point_3<R> q)*

CGAL_Comparison_result *CGAL_compare_z(CGAL_Point_3<R> p, CGAL_Point_3<R> q)*

For lexicographical comparison CGAL provides

CGAL_Comparison_result *CGAL_compare_lexicographically_xyz(CGAL_Point_3<R> p, CGAL_Point_3<R> q)*

Compares the Cartesian coordinates of points p and q lexicographically in xyz order: first x-coordinates are compared, if they are equal, y-coordinates are compared, and if both x- and y- coordinate are equal, z-coordinates are compared.

In addition, CGAL provides the following comparison functions returning *true* or *false* depending on the result of *CGAL_compare_lexicographically_xyz(p,q)*.

bool *CGAL_lexicographically_xyz_smaller_or_equal(CGAL_Point_3<R> p, CGAL_Point_3<R> q)*

bool *CGAL_lexicographically_xyz_smaller(CGAL_Point_3<R> p, CGAL_Point_3<R> q)*

Chapter 21

3D Transformations

CGAL provides affine transformations. The primitive objects in CGAL are closed under affine transformations except for iso-oriented objects and bounding boxes.

The general form of an affine transformation is based on homogeneous representation of points. Thereby all transformations can be realized by matrix multiplication.

Since the general form is based on the homogeneous representation, a transformation matrix multiplication by a scalar does not change the represented transformation. Therefore, any transformation represented by a matrix with rational entries can be represented by a transformation matrix with integer entries as well by multiplying the matrix with the common denominator of the rational entries. Hence it is sufficient to have number type *R::RT* for the entries of an affine transformation.

CGAL offers several specialized affine transformations. They are hidden behind the interface class. Different constructors are provided to create them. They are parameterized with a symbolic name to denote the transformation type, followed by additional parameters. The symbolic name tags solve ambiguities in the function overloading and they make the code more readable, i.e. what type of transformation is created. These name tags are constants of an appropriate type.

const CGAL_Translation	*CGAL_TRANSLATION;*
const CGAL_Rotation	*CGAL_ROTATION;*
const CGAL_Scaling	*CGAL_SCALING;*

21.1 3D Affine Transformation (*CGAL_Aff_transformation_3*<*R*>)

Definition

In three-dimensional space we have a 4×4 matrix (m_{ij}). Entries m_{30}, m_{31}, and m_{32} are always zero and therefore do not appear in the constructors.

#include <*CGAL/Aff_transformation_3.h*>

Creation

CGAL_Aff_transformation_3<R> *t(const CGAL_Translation, CGAL_Vector_3<R> v);*

introduces a translation by a vector v.

CGAL_Aff_transformation_3<R> *t(const CGAL_Scaling, R::RT s, R::RT hw = RT(1));*

introduces a scaling by a scale factor s/hw.

CGAL_Aff_transformation_3<R> *t(R::RT m00,*
R::RT m01,
R::RT m02,
R::RT m03,
R::RT m10,
R::RT m11,
R::RT m12,
R::RT m13,
R::RT m20,
R::RT m21,
R::RT m22,
R::RT m23,
R::RT hw = RT(1))

introduces a general affine transformation in the 4×4 matrix form $\begin{pmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ 0 & 0 & 0 & hw \end{pmatrix}$. The sub-matrix $\frac{1}{hw}\begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix}$ contains the scaling and rotation information, the vector $\frac{1}{hw}\begin{pmatrix} m_{03} \\ m_{13} \\ m_{23} \end{pmatrix}$ contains the translational part of the transformation.

CGAL_Aff_transformation_3<R> *t(R::RT m00,*
R::RT m01,
R::RT m02,
R::RT m10,
R::RT m11,
R::RT m12,
R::RT m20,
R::RT m21,
R::RT m22,
R::RT hw = RT(1))

introduces a general linear transformation of the matrix form $\begin{pmatrix} m_{00} & m_{01} & m_{02} & 0 \\ m_{10} & m_{11} & m_{12} & 0 \\ m_{20} & m_{21} & m_{22} & 0 \\ 0 & 0 & 0 & hw \end{pmatrix}$, i.e. an affine transformation without translational part.

Operations

Each class *CGAL_Class_3<R>* representing a geometric object in 3D has a member function:

CGAL_Class_3<R> transform(CGAL_Aff_transformation_3<R> t).

The transformation classes provide a member function *transform()* for points, vectors, directions, and planes:

CGAL_Point_3<R>	*t.transform(CGAL_Point_3<R> p)*	
CGAL_Vector_3<R>	*t.transform(CGAL_Vector_3<R> p)*	
CGAL_Direction_3<R>	*t.transform(CGAL_Direction_3<R> p)*	
CGAL_Plane_3<R>	*t.transform(CGAL_Plane_3<R> p)*	

CGAL provides four function operators for these member functions:

CGAL_Point_3<R>	*t(CGAL_Point_3<R> p)*	
CGAL_Vector_3<R>	*t(CGAL_Vector_3<R> p)*	
CGAL_Direction_3<R>	*t(CGAL_Direction_3<R> p)*	
CGAL_Plane_3<R>	*t(CGAL_Plane_3<R> p)*	
CGAL_Aff_transformation_3<R>	$t * s$	composes two affine transformations.
CGAL_Aff_transformation_3<R>	*t.inverse()*	gives the inverse transformation.
bool	*t.is_even()*	returns *true*, if the transformation is not reflecting, i.e. the determinant of the involved linear transformation is non-negative.
bool	*t.is_odd()*	returns *true*, if the transformation is reflecting.

The matrix entries of a matrix representation of a *CGAL_Aff_transformation_2<R>* can be accessed trough the following member functions:

FT	*t.cartesian(int i, int j)*	
FT	*t.m(int i, int j)*	returns entry m_{ij} in a matrix representation in which m_{22} is 1.
RT	*t.homogeneous(int i, int j)*	

RT	*t.hm(int i, int j)*	returns entry m_{ij} in some fixed matrix representation.

For affine transformations no I/O operators are defined.

Bibliography

[Hof89a] C. Hoffmann. *Geometric and Solid Modeling.* Morgan-Kaufmann, San Mateo, CA, 1989.

[Hof89b] C. M. Hoffmann. The problems of accuracy and robustness in geometric computation. *IEEE Computer*, 22(3):31–41, March 1989.

[Sch97] S. Schirra. Precision and robustness in geometric computations. In Marc van Kreveld, Jürg Nievergelt, Thomas Roos, and Peter Widmayer, editors, *Algorithmic foundations of geographic information systems*, volume 1340 of *Lecture notes in computer science*, pages 255–287. Springer-Verlag, Berlin, 1997.

[Yap97] C. Yap. Towards exact geometric computation. *Comput. Geom. Theory Appl.*, 7:3–23, 1997.

[YD95] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 1 of *Lecture Notes Series on Computing*, pages 452–492. World Scientific, Singapore, 2nd edition, 1995.