Author manuscript, published in "Library Centric Software Design (LCSD) (2006)"

A Generic Lazy Evaluation Scheme for Exact Geometric Computations

[Extended Abstract]

Sylvain Pion
INRIA, Sophia-Antipolis, FRANCE
Sylvain.Pion@sophia.inria.fr

Andreas Fabri
GeometryFactory, Grasse, FRANCE
andreas.fabri@geometryfactory.com

inria-00344960, version 1 - 7 Dec 2008

ABSTRACT
We present a generic C++ design to perform efficient and exact geometric computations using lazy evaluations. Exact geometric computations are critical for the robustness of geometric algorithms. Their efficiency is also critical for most applications, hence the need for delaying the exact computations at run time until they are actually needed. Our approach is generic and extensible in the sense that it is possible to make it a library which users can extend to their own geometric objects or primitives. It involves techniques such as generic functor adaptors, dynamic polymorphism, reference counting for the management of directed acyclic graphs and exception handling for detecting cases where exact computations are needed. It also relies on multiple precision arithmetic as well as interval arithmetic. We apply our approach to the whole geometric kernel of CGAL.

Keywords
computational geometry, exact geometric computation, numerical robustness, interval arithmetic, lazy evaluation, generic programming, C++, CGAL

1. INTRODUCTION
Non-robustness issues due to numerical approximations are well known in geometric computations, especially in the computational geometry literature. The development of the CGAL library, a large collection of geometric algorithms implemented in C++, expressed the need for a generic and efficient treatment of these problems.

Typical solutions to solve these problems involve exact arithmetic computations. However, due to efficiency issues, good implementations make use of arithmetic filtering techniques to benefit from the speed of certified floating-point approximations like interval arithmetic, hence calling the costly multi-precision routines rarely.

One efficient approach is to perform lazy exact computations at the level of geometric objects. It is mentioned in [13] and an implementation is described in [7]. Unfortunately, this implementation does not use the generic programming paradigm, although the approach is general. This is exactly the novelty of this paper.

In this paper, we devise a generic design to provide the most generally applicable methods to a large number of geometric primitives. Our design makes it easy to apply to the complete geometry kernel of CGAL, and is extensible to the user's new geometric objects and geometric primitives.

Our design thus implements lazy evaluation of the exact geometric objects. The computation is delayed until a point where the approximation with interval arithmetic is not precise enough to decide safely comparisons, which may hopefully never be needed.

Section 2 describes in more detail the context and motivation in geometric computing, as well as the basics of a generic geometric kernel parameterized by the arithmetic, and what can be done at this level. Then, Section 3 discusses our design in detail, namely how geometric predicates, constructions and objects are adapted. Section 4 illustrates how our scheme can be applied to the users' own geometric objects and primitives. We then provide in Section 5 some benchmarks that confirm the benefit of our design and implementation. Finally, we list a few open questions related to our design in Section 6, and conclude with ideas for future work.

2. EXACT GEOMETRIC COMPUTATIONS AND THE CGAL KERNEL

2.1 Exact Geometric Computations
Many geometric algorithms such as convex hull computations, Delaunay triangulations, mesh generators, are notoriously prone to robustness issues due to the approximate nature of floating-point computations. This is due to the dual nature of geometric algorithms: on one side numerical data is used, such as coordinates of points, and on the other side discrete structures are built, such as the graph representing a mesh.

The bridges between the numerical data and the Boolean decisions which allow to build a discrete structure, are called

the geometric *predicates*. These are functions taking geometric objects such as points as input and returning a Boolean or enumerated value. Internally, these functions typically perform comparisons of numerical values computed from the input. A classical example is the `orientation` predicate of three points in the plane, which returns if the three points are doing a left turn, a right turn, or if they are collinear (see Figure 1). Using Cartesian coordinates for the points, the orientation is the sign (as a three-valued function: -1, 0, 1) of the following 3-dimensional determinant which reduces to a 2-dimensional one:

$$\begin{vmatrix} 1 & 1 & 1 \\ p.x() & q.x() & r.x() \\ p.y() & q.y() & r.y() \end{vmatrix} = \begin{vmatrix} q.x() - p.x() & r.x() - p.x() \\ q.y() - p.y() & r.y() - p.y() \end{vmatrix}$$

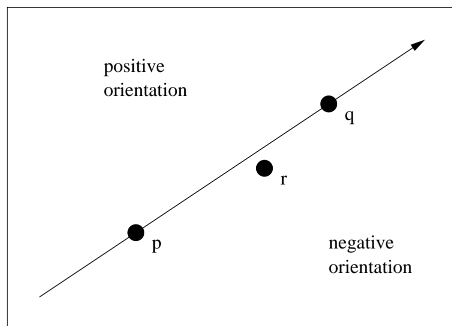

Figure 1: The orientation predicates of 3 points in the plane.

Many predicates are built on top of signs of polynomial expressions over the coordinates of the input points. Evaluating such a function with floating-point arithmetic is going to introduce roundoff errors, which can have for consequence that the sign of the approximate value differs from the sign of the exact value. The impact of wrong signs on the geometric algorithms which call the predicates can be disastrous, as for example it can break some invariants like planarity of a graph, or make the algorithm loop. Didactic examples of consequences can be found in [12] as well as in the computational geometry literature.

Operations building new geometric objects, like the point at the intersection of two lines, the circumcenter of three non-collinear points, or the midpoint of two points, are called geometric *constructions*. We will use the term geometric *primitives* when referring to either predicates or constructions.

In order to tackle these non-robustness issues, many solutions have been proposed. We focus here on the *exact geometric computation paradigm* [16], as it is a general solution. This paradigm states that, in order to ensure the correct execution of the algorithms, it is enough that all decisions based on predicates are taken correctly. Concretely, this means that all comparisons of numerical values need to be performed exactly.

A natural way to perform the exact evaluation of predicates is to evaluate the numerical expressions using exact arithmetic. For example, since most computations are signs of polynomials, it is enough to use multi-precision rational arithmetic which is provided by libraries such as GMP [8]. Note that exact arithmetic is also available for all algebraic computations using libraries such as CORE [10] or LEDA [5], which is useful when doing geometry over curved objects. This solution works well, but it tends to be very slow.

2.2 The Geometry Kernel of CGAL

CGAL [1] is a large collection of computational geometry algorithms. These algorithms are parameterized by the geometry they apply to. The geometry takes the form of a *kernel* [9, 4] regrouping the types of the geometric objects such as points, segments, lines, ... as well as the basic primitives operating on them, in the form of functors. The CGAL kernel provides over 100 predicates and 150 constructions, hence uniformity and genericity is crucial when treating them, from a maintenance point of view.

CGAL provides several models of kernels. The basic families are the template classes `Cartesian` and `Homogeneous` which are parameterized by the type representing the coordinates of the points. They respectively use Cartesian and homogeneous representations of the coordinates, and their implementation looks as follows:

```
template < class NT >
struct Cartesian {
  // Geometric objects
  typedef ...           Point_2;
  typedef ...           Point_3;
  typedef ...           Segment_2;
  ...
  // Functors for predicates
  typedef ...           Compare_x_2;
  typedef ...           Orientation_2;
  ...
  // Functors for constructions
  typedef ...           Construct_midpoint_2;
  typedef ...           Construct_circumcenter_2;
  ...
};
```

These simple template models already allow to use `double` arithmetic or multi-precision rational arithmetic for example. CGAL therefore provides a hierarchy of concepts for the *number types*, which describe the requirements for types to be pluggable into these kernels, such as addition, multiplication, comparisons... The functors are implemented in the following way (here the return type of the predicate is a three-valued enumerated type, moreover some `typename` keywords are removed for clarity):

```
template < class Kernel >
class Orientation_2 {
  typedef Kernel::Point_2     Point;
  typedef Kernel::FT          FT;
public:
  typedef CGAL::Orientation   result_type;

  result_type
  operator()(Point p, Point q, Point r) const
```

inria-00344960, version 1 - 7 Dec 2008

```
  {
    FT det = (q.x() - p.x()) * (r.y() - p.y())
           - (r.x() - p.x()) * (q.y() - p.y());
    if (det > 0) return POSITIVE;
    if (det < 0) return NEGATIVE;
    return ZERO;
  }
};

template < class Kernel >
class Construct_midpoint_2 {
  typedef Kernel::Point_2     Point;
public:
  typedef Point               result_type;

  result_type
  operator()(Point p, Point q) const
  {
    return Point( (p.x() + q.x()) / 2,
                  (p.y() + q.y()) / 2 );
  }
};
```

As much as conversions between number types are useful, CGAL also provides tools to convert geometric objects between different kernels. We shortly present these here as they will be referred to in the sequel. A kernel converter is a functor whose function operator is overloaded for each object of the source kernel and which returns the corresponding object of the target kernel. Such conversions may depend on the details of representation of the geometric objects, such as homogeneous versus Cartesian representation. CGAL provides such converters parameterized by converters between number types, for example the converter between kernels of the `Cartesian` family:

```
template < class K1, class K2, class NT_conv =
                Default_conv<K1::FT, K2::FT> >
struct Cartesian_converter {
  NT_conv cv;

  K2::Point_2
  operator()(K1::Point_2 p) const
  {
    return K2::Point_2( cv(p.x()), cv(p.y()) );
  }
  ...
};
```

Related to this, CGAL also provides a way to find out the type of a geometric object (say, a 3D segment) in a given kernel, given its type in another kernel and this second kernel. This is in practice the return type of the function operator of the kernel converter described above.

```
template < class O1, class K1, class K2 >
struct Type_mapper {
  typedef ...  type;
};
```

The current implementation works by specializing on all known kernel object types like `K1::Point_2`, `K1::Segment_3`. A more extensible approach could be sought, although this is not the main point of this paper.

2.3 A Generic Lazy Exact Number Type

In order to speed up the exact evaluation of predicates, people have observed that, given that the floating-point evaluation gives the right answer in most cases, it should be enough to add a way to detect the cases where it can change the sign, and rely on the costly multi-precision arithmetic only in those cases. These techniques are usually referred to as arithmetic filtering.

There are many variants of arithmetic filters, but we are going to focus on one which applies nicely in a generic context, and is based on interval arithmetic [3], a well known tool to control roundoff errors in floating-point computations. The idea is that we implement a new number type which forwards its operations to an interval arithmetic type, and also remembers the way it was constructed by storing the history of operations in a directed acyclic graph (DAG) [2]. Figure 2 illustrates the history DAG of the expression $\sqrt{x}+\sqrt{y}-\sqrt{x+y+2\sqrt{xy}}$.

When a comparison is performed on this number type and the intervals overlap, then the DAG is used to recompute the values with an exact multi-precision type, hence giving the exact result. CGAL provides such a lazy number type called `Lazy_exact_nt<NT>` parameterized by the exact type used to perform the exact computations when needed (such as a rational number type). Somehow, this can be seen as a wrapper on top of its template parameter, which delays the computations until they are needed, as hopefully they won't be needed at all.

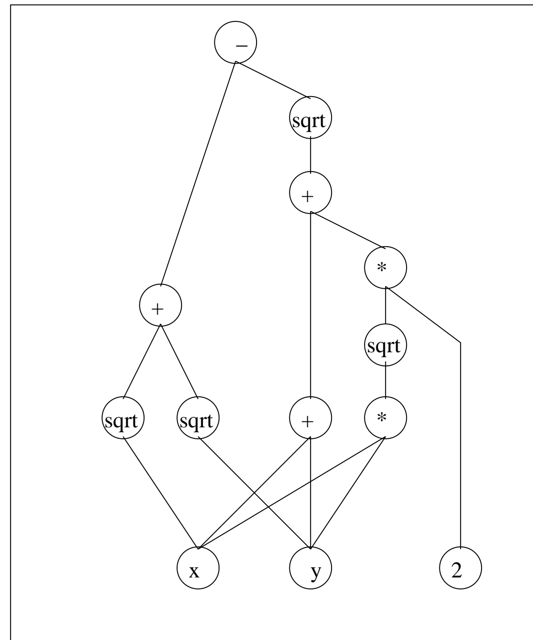

Figure 2: Example Dag: $\sqrt{x}+\sqrt{y}-\sqrt{x+y+2\sqrt{xy}}$.

This solution works very well. It can however be further

inria-00344960, version 1 - 7 Dec 2008

improved in terms of efficiency. Indeed we note that there are several overheads which can be optimized. First, a node of the DAG is created for each arithmetic operation, so it would be nice to be able to regroup them in order to diminish the number of memory allocations as well as the memory footprint. Second, rounding mode changes for interval arithmetic computations are made for each arithmetic operation, so again, it would be nice to be able to regroup them to optimize away these mode changes.

These remark have lead to a new scheme mentioned in [13], and the description of an implementation has also been proposed in [7]. The idea is to introduce a DAG at the geometric level, by considering geometric primitives for the nodes. The next section describes such an optimized setup. Our design differs from the one in [7] in that we followed the generic programming paradigm and extensive use of templates to make it as easily extensible as possible.

3. DESIGN OF THE LAZY EXACT COMPUTATION FRAMEWORK

The previously described design of lazy computation is based only on genericity over the number type. In this section, we make use of the genericity at the higher level of geometric primitives, in order to provide a more efficient solution. We first describe how to filter the predicates. Then we extend the previous idea of `Lazy_exact_nt` to geometric objects and constructions.

3.1 Filtered Predicates

Performing a filtered predicate means first evaluating the predicate with interval arithmetic. If it fails, the predicate is evaluated again, this time with an exact number type. As all predicates of a CGAL kernel are functors we can use the following adaptor:

```
template <class EP, class AP, class C2E, class C2A>
class Filtered_predicate
{
  typedef AP    Approximate_predicate;
  typedef EP    Exact_predicate;
  typedef C2E   To_exact_converter;
  typedef C2A   To_approximate_converter;

  EP  ep;
  AP  ap;
  C2E c2e;
  C2A c2a;

public:

  typedef EP::result_type  result_type;

  template <class A1, class A2>
  result_type
  operator()(const A1 &a1, const A2 &a2) const
  {
    try {
      Protect_FPU_rounding P(FE_TOINFTY);
      return ap(c2a(a1), c2a(a2));
    } catch (Interval_nt_advanced::unsafe_comparison) {
      Protect_FPU_rounding P(FE_TONEAREST);
      return ep(c2e(a1), c2e(a2));
    }
  }
};
```

Function operators with any arity should be provided. This is currently done by hand up till a fixed arity, and will be replaced when variadic templates become available in C++.

Note that `Protect_FPU_rounding` changes the current rounding mode of the FPU to the one specified as argument to the constructor, and saves the old one in the object. Its destructor restores the saved mode, which happens at the return of the function or when an exception is thrown.

The class `Filtered_kernel` is hence obtained from a kernel `K` by adapting all predicates of `K`. This is currently done with the preprocessor. The geometric objects as well as the constructions remain unchanged.

```
template < class K >
struct Filtered_kernel {

  // The various kernels
  typedef Cartesian<double>            CK;
  typedef Cartesian<Interval_nt>       AK;
  typedef Cartesian<Gmpq>              EK;

  // Kernel converters
  typedef Cartesian_converter<CK, AK>  C2A;
  typedef Cartesian_converter<CK, EK>  C2E;

  // Geometric objects
  typedef CK::Point_2                  Point_2;
  ...
  // Functors for predicates
  typedef Filtered_predicate<AK::Compare_x_2,
                             EK::Compare_x_2,
                             C2E, C2A> Compare_x_2;
  ...
};
```

3.2 Lazy Exact Objects

Performing lazy exact constructions means performing constructions with interval approximations, and storing the sequence of construction steps. When later a predicate applied to these approximations cannot return a result that is guaranteed to be correct, the sequence of construction steps is performed again, this time with an exact arithmetic. Now the predicate can be evaluated correctly.

The sequence of construction steps is stored in a DAG. Each node of the DAG stores (i) an approximation, (ii) the exact version of the function that was used to compute the approximation, (iii) and the lazy objects that were arguments to the function. So the out-degree of a node is the arity of the function.

The example illustrates that lazy objects can be of the same type, without being the result of the same sequence of constructions. a, m, and b are all point-ish. Therefore we have a template handle class, with a pointer to a node of the DAG. In our example, only the latter are of different types.

We will now explain some of the classes in Figure 4 in more detail.

`Lazy_exact` is the handle class. It also does reference counting with a design similar to the one described in [11]. It has `Lazy_exact_nt` as subclass, which provides arithmetic operations. Note that this framework handles arithmetic and geometric objects in a unified way. For example a distance between geometric objects yields a lazy exact number, and

inria-00344960, version 1 - 7 Dec 2008

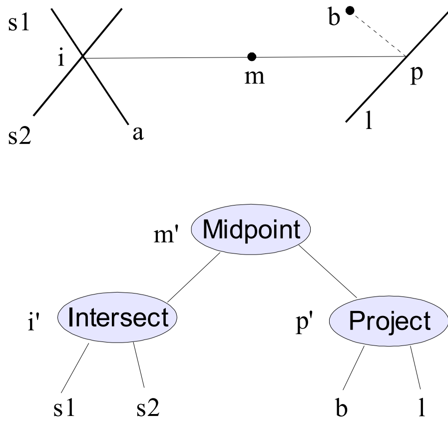

Figure 3: The Dag represents the midpoint of an intersection point and the vertical projection of a point on a line. Testing whether a, m, and b are collinear has a good chance to trigger an exact construction.

a lazy exact number can become the coordinate of a point.

The class `Construction` is an abstract base class. It stores the approximation, and holds a pointer to the exact value. Initially, this pointer is set to `NULL`, and it is the virtual member function `update_exact` which later may compute the exact value and then cache it.

The subclass `Construction_2` is used for binary functions. Similar classes exist for the other arities. These classes store the arguments and the exact version of the function. The arguments may be of arbitrary types. In the case of lazy exact geometric objects or lazy exact numbers the arguments are handles as described before.

```
template <class AC, class EC, class LK, class A1, class A2>
class Construction_2
  : public Construction<AC::result_type, EC::result_type, E2A>
  , private EC
{
  typedef AC               Approximate_construction;
  typedef EC               Exact_construction;
  typedef LK::C2E          To_exact_converter;
  typedef LK::C2A          To_approximate_converter;
  typedef LK::E2A          Exact_to_approximate_converter;
  typedef AC::result_type  AT;
  typedef EC::result_type  ET;

  A1 m_a1;
  A2 m_a2;

  const EC& ec() const { return *this; }

public:

  void
  update_exact()
  {
```

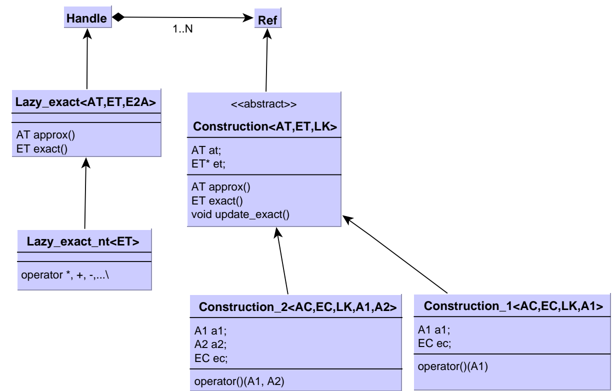

Figure 4: The class hierarchy for the nodes of the Dag.

```
      this->et = new ET(ec()(C2E()(m_a1), C2E()(m_a2)));
      this->at = E2A()(*(this->et));
      // Prune lazy dag
      m_a1 = A1();
      m_a2 = A2();
    }

    Construction_2(const AC& ac, const EC& ec,
           const A1& a1, const A2& a2)
      : Construction<AT,ET,E2A>(ac(C2A()(a1), C2A()(a2)),
                                m_a1(a1), m_a2(a2)
    {}
  };
```

The constructor stores the two arguments. It then takes their approximations and calls the approximate version of the functor.

In case the exact version of the construction is needed, this gets computed in the `update_exact` method. It fetches the exact versions of the arguments, which in turn may trigger their exact computation if they are not already computed and cached. From the exact lazy object one computes again the approximate object, as the object computed with the approximate version of the functor has a good chance to have accumulated more numerical error.

Finally, the Dag is pruned. As the nodes of the Dag are reference counted, some of them may get deallocated by the pruning. Most often `A1` and `A2` will be lazy exact objects. For performance reasons their default constructors generates a handle to a shared static node of the Dag.

Also, we use private derivation of the exact construction `EC`, instead of storing it as data member, in order to benefit from the empty base class optimization.

The other derived classes store the leaves of the Dag. There is a general purpose leaf class, and more specialized ones, for example for creating a lazy exact number from an `int`. They are there for performance reasons.

3.3 The Functor Adaptor

So far we have only explained how lazy constructions are stored, but not how new nodes of the Dag are generated.

inria-00344960, version 1 - 7 Dec 2008

The following functor adaptor is applied to all the constructions we want to make lazy. It has function operators for other arities.

```
template <class LK, class AC, class EC>
class Lazy_construct
{
  typedef LK                    Lazy_kernel;
  typedef AC                    Approximate_construction;
  typedef EC                    Exact_construction;
  typedef LK::AK                AK;
  typedef LK::EK                EK;
  typedef EK::FT                EFT;
  typedef LK::E2A               E2A;
  typedef LK::C2E               C2E;
  typedef AC::result_type       AT;
  typedef EC::result_type       ET;
  typedef Lazy_exact<AT, ET, E2A> Handle;

  AC ac;
  EC ec;

public:

  typedef Type_mapper<AT,AK,LK>::type result_type;

  template <class A1, class A2>
  result_type
  operator()(const A1& a1, const A2& a2) const
  {
    try {
      Protect_FPU_rounding P(FE_TOINFTY);
      return Handle(new Construction_2<AC, EC, LK, A1, A2>
                                     (ac, ec, a1, a2));
    } catch (Interval_nt_advanced::unsafe_comparison) {
      Protect_FPU_rounding P(FE_TONEAREST);
      return Handle(new Construction_0<AT,ET,LK>
                                (ec(C2E()(a1), C2E()(a2))));
    }
  }
};
```

The functor first tries to construct a new node of the Dag. If inside the approximate version of the construction an exception is thrown, we perform the exact version of the construction, and only create a leaf node for the Dag.

3.4 Special-Case Handling

The generic functor adaptor works out of the box for all functors that return lazy exact geometric objects or a lazy exact number.

Functors returning objects which are not made lazy are an easy to handle exception. An example in Cgal is the functor that computes a bounding box with `double` coordinates around geometric objects, whose width is not required to be tight. As the intervals corresponding to the coordinates of the approximate geometric object are already 1-dimensional bounding boxes, we never have to resort to the exact geometric object. The functor adaptor is trivial.

Some functors of Cgal kernels return a polymorphic object. For example, the intersection of two segments may be empty, or a point, or a segment. In order not to have a base class for all geometric classes, Cgal offers a class `Object`[1] which is capable of storing typed objects. The problem we have to solve is that the lazy exact functor must not return a lazy exact `Object`, but instead must return an `Object` holding a lazy geometric object. This is solved by looping over all Cgal kernel types, to try to cast, and if it works to construct the lazy geometric object and put it in an `Object` again.

[1]The `Object` class is comparable to `boost::any`.

Less trivial cases are functors which pass results of a computation back to reference parameters, or which write into output iterators. They need a special functor as well as special `Construction` classes. It is not hard to write them, but the problem is that they must be dispatched by hand, as we have no means of introspection. One solution would be to introduce functor categories.

3.5 The Lazy Exact Kernel

We are ready to put all pieces together, by defining a new kernel which has an approximate and an exact kernel as template parameters.

```
template < class AK, class EK >
struct Lazy_kernel {

  // Kernel converters
  typedef Lazy_kernel<AK, EK>           LK;
  typedef Approx_converter<LK, AK>      C2A;
  typedef Exact_converter<LK, EK>       C2E;
  typedef Cartesian_converter<EK, AK>   E2A;

  // Geometric objects
  typedef Lazy_exact<AK::Point_2,   EK::Point_2>   Point_2;
  typedef Lazy_exact<AK::Segment_2, EK::Segment_2> Segment_2;

  // Functors for predicates
  typedef Filtered_predicate<EK::Compare_x_2, AK::Compare_x_2,
                             C2E, C2A> Compare_x_2;
  ...

  // Functors for constructions
  typedef Lazy_construct<LK, AK::Construct_midpoint_2,
                         EK::Construct_midpoint_2>
          Construct_midpoint_2;
  ...

  typedef Lazy_Construct_returning_object<LK, AK::Intersection_2,
                                          EK::Intersection_2>
          Intersection_2;
};
```

In the current implementation we use the preprocessor to generate the typedefs from a list of types, and we use the Boost Mpl library for dispatching the special cases. `Approx_converter` simply fetches the stored approximate object. Similarly `Exact_converter` fetches the exact approximate object, possibly triggering its computation.

4. EXTENSIBILITY

We have to distinguish between different levels of extensibility.

When Cgal kernels get extended by geometric objects and constructions this needs changes in the lazy construction framework if the new constructions have “new” interfaces, e.g., two output iterators, followed by two reference parameters to return a result. This would need a new node type for the Dag, a new functor, and hard wired dispatching in the lazy kernel. Otherwise there is nothing to do.

When the Cgal user wants to extend the lazy kernel with his own geometric objects and constructions, he first has to add them to the kernel that then gets into the lazy computation

inria-00344960, version 1 - 7 Dec 2008

machinery, as described in [9]. Then, what we stated in the previous paragraph applies.

The Curved_kernel and the Lazy_curved_kernel of CGAL which provide primitives on circles and circular arcs [14, 6], are examples for both.

5. BENCHMARKS

We now run a simple benchmark that illustrates the benefit of our techniques. We compare the running time and memory consumption of various kernel choices with the following algorithm:

- generate 2000 pairs of 2D points with random coordinates (using drand48()).
- construct 2000 segments out of these points.
- intersect all pairs of segments among these, and store the resulting intersection points.
- shuffle the resulting points
- iterate over consecutive triplets of these points, and compute the orientation predicate of these.

Figure 5 provides the resulting data for a choice of four different kernels:

- SC<Gmpq> stands for the simple Cartesian representation kernel parameterized with Gmpq, which is a C++ wrapper around the multi-precision rational number type provided by GMP,
- SC<Lazy_exact_nt<Gmpq>> uses the lazy exact evaluation mechanism at the arithmetic level,
- Lazy_kernel<SC<Gmpq>> is our approach for performing lazy exact evaluations at the geometric object level,
- Lazy_kernel<SC<Gmpq>> (2) is similar to the previous one, but it does not include the additional optimization which consists in eliminating rounding mode changes, which is allowed by the consecutive interval computations,
- finally, SC<double> is the simple Cartesian representation kernel parameterized with double. It is given for reference as it is not robust in all cases. It shows what the optimal performance could be.

Benchmarks have been performed using the GNU g++ compiler versions 3.4 and 4.1 with the -O2 optimization option. The machine was a Pentium-M laptop at 1.7 GHz, equipped with 1 GB of RAM and 1 MB of cache, running the Fedora Core 3 Linux distribution. The memory consumption is the same for these two compiler versions, however timings differ significantly. Timings are given in seconds and memory in megabytes.

The results show that our approach wins almost a factor of 10 on memory over the basic lazy evaluation scheme. It is also between 2 and 3 times faster. However, it remains 4 times slower than the approximate floating-point evaluation, but of course it is guaranteed for all cases.

Kernel	time g++ 3.4	time g++ 4.1	mem
SC<Gmpq>	71	70	70
SC<Lazy_exact_nt<Gmpq>>	9.4	7.4	501
Lazy_kernel<SC<Gmpq>> (2)	4.9	3.6	64
Lazy_kernel<SC<Gmpq>>	4.1	2.8	64
SC<double>	0.98	0.72	8.3

Figure 5: Benchmarks comparing different kernels.

The benchmark also illustrates the gain obtained thanks to the elimination of rounding mode changes, which is now allowed by the regrouping of operations on intervals.

Another data point illustrating the improvements is that we measured the number of DAG nodes allocated. For SC<Lazy_exact_nt<Gmpq>>, 29 million nodes were allocated, while for Lazy_kernel<SC<Gmpq>> only 2.5 million nodes were needed. So we have won a factor of more than 10, due to the regrouping allowed by our design.

Note that the algorithm we chose uses random data, hence it does not produce many filter failures, so almost no exact evaluation is performed. Another thing worth noticing is that it uses relatively simple 2D primitives. More complex primitives, especially in higher dimensions, should show more benefits to the method. Finally, real-world geometric applications tend to produce more combinatorial output, hence the relative runtime cost of primitives is smaller, so the slow down factor is lower in those cases. First such experiments on a 3D surface reconstruction algorithm have shown a factor of 6 improvement on memory consumption and a speed up factor of 3.

6. OPEN DESIGN QUESTIONS

Here is a list of open questions related to our framework.

The first question concerns the regrouping of expressions. Our framework asks the user to pass it functors specifying the level at which the regrouping of expressions is made. In CGAL this is not a problem since the primary interface of the kernel towards the geometric algorithms is a list of functors. However it has the drawback of not being automatic. We can think of approaches based on expression templates [15] which would automatically detect sequences of operations and regroup them. Unfortunately, expression templates are limited to single statement expressions and they tend to slow down compilation times considerably. Could there be a way to extend the automatic regrouping to more than single statements? Maybe the auto keyword recently proposed for addition to the C++ language will allow to propagate this through several statements? Or maybe the Axiom feature part of the proposal for concepts in C++ could be used to specify this kind of transformation.

Another question is if similarly delayed computations are

inria-00344960, version 1 - 7 Dec 2008

used in other areas, and if yes, then is it possible to find out a common design, more general than the one we propose.

7. CONCLUSION AND FUTURE WORK

We have presented in this paper a generic framework which implements lazy exact geometric computations, motivated by the needs for robustness and efficiency of geometric algorithms. This framework allows to delay the costly exact evaluation using multi-precision arithmetic when the faster interval arithmetic suffices.

The proposed design is easily extensible to new geometric primitives – predicates and constructions –, as well as new geometric objects. It is based on a template family for representing lazy objects, as well as generic functor adaptors which produce them.

Future work in this area will consist of various added special-case optimizations as well as generalizations. It is for example possible to refine the filtering scheme by growing the precision little by little instead of switching directly to full multi-precision computation in case of insufficiency of precision of the intervals. We also would like to study possibilities of merging the `Filtered_predicate` and `Lazy_construct` functor adaptors. Possible optimizations for specific cases also can be done, using faster schemes than interval arithmetic (so-called static filters). Moreover, the current way of providing a full kernel is by a list of types for the objects and functors, which is provided through the use of the preprocessor, we will therefore try to provide a better design on this particular point.

Finally, we plan to make our implementation part of a future release of CGAL, whose entire geometry kernel already benefits from it.

8. ACKNOWLEDGMENTS

The work reported in this paper has been supported in part by the IST Programme of the EU as a Shared-cost RTD (FET Open) Project under Contract No IST-006413 (ACS - Algorithms for Complex Shapes).

9. REFERENCES

[1] *CGAL User and Reference Manual*, 3.2 edition, 2006.

[2] M. Benouamer, P. Jaillon, D. Michelucci, and J.-M. Moreau. A lazy solution to imprecision in computational geometry. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 73–78, 1993.

[3] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Applied Mathematics*, 109:25–47, 2001.

[4] H. Brönnimann, A. Fabri, G.-J. Giezeman, S. Hert, M. Hoffmann, L. Kettner, S. Schirra, and S. Pion. 2D and 3D kernel. In C. E. Board, editor, *CGAL User and Reference Manual*. 3.2 edition, 2006.

[5] C. Burnikel, K. Mehlhorn, and S. Schirra. The LEDA class real number. Technical Report MPI-I-96-1-001, Max-Planck Institut Inform., Saarbrücken, Germany, Jan. 1996.

[6] I. Z. Emiris, A. Kakargias, S. Pion, M. Teillaud, and E. P. Tsigaridas. Towards an open curved kernel. In *Proc. 20th Annu. ACM Sympos. Comput. Geom.*, pages 438–446, 2004.

[7] S. Funke and K. Mehlhorn. Look – a lazy object-oriented kernel for geometric computation. *Computational Geometry - Theory and Applications (CGTA)*, 22:99–118, 2002.

[8] T. Granlund. GMP, the GNU multiple precision arithmetic library. `http://www.swox.com/gmp/`.

[9] S. Hert, M. Hoffmann, L. Kettner, S. Pion, and M. Seel. An adaptable and extensible geometry kernel. In *Proc. Workshop on Algorithm Engineering*, volume 2141 of *Lecture Notes Comput. Sci.*, pages 79–90. Springer-Verlag, 2001.

[10] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. *The CORE Library Project*, 1.2 edition, 1999. http://www.cs.nyu.edu/exact/core/.

[11] L. Kettner. Reference counting in library design – optionally and with union-find optimization. In *Workshop on Library Centric Software Design*, october 2005.

[12] L. Kettner, K. Mehlhorn, S. Pion, S. Schirra, and C. Yap. Classroom examples of robustness problems in geometric computations. In *Proc. 12th European Symposium on Algorithms*, volume 3221 of *Lecture Notes Comput. Sci.*, pages 702–713. Springer-Verlag, 2004.

[13] S. Pion. *De la géométrie algorithmique au calcul géométrique*. Thèse de doctorat en sciences, Université de Nice-Sophia Antipolis, France, 1999. TU-0619.

[14] S. Pion and M. Teillaud. 2D circular kernel. In C. E. Board, editor, *CGAL User and Reference Manual*. 3.2 edition, 2006.

[15] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, June 1995. Reprinted in C++ Gems, ed. Stanley Lippman.

[16] C. Yap. Towards exact geometric computation. *Comput. Geom. Theory Appl.*, 7(1):3–23, 1997.

inria-00344960, version 1 - 7 Dec 2008

APPENDIX
A. BENCHMARK CODE

```
#include <CGAL/Simple_cartesian.h>
#include <CGAL/Lazy_kernel.h>
#include <CGAL/Gmpq.h>
#include <CGAL/Lazy_exact_nt.h>
#include <CGAL/intersections.h>
#include <CGAL/Timer.h>
#include <CGAL/Memory_sizer.h>
using namespace CGAL;

// Choosing a kernel:
//typedef Simple_cartesian<Gmpq>                K;
//typedef Simple_cartesian<Lazy_exact_nt<Gmpq> > K;
//typedef Lazy_kernel<Simple_cartesian<Gmpq> >   K;
typedef Simple_cartesian<double>              K;

typedef K::Point_2    Point;
typedef K::Segment_2  Segment;

Point   random_point()   { return Point(drand48(), drand48()); }
Segment random_segment() { return Segment(random_point(), random_point()); }

int main() {
  int loops = 2000, init_mem = Memory_sizer().virtual_size();
  Timer t; t.start();

  std::cout << "Generating initial random segments: " << loops << std::endl;
  std::vector&lt;Segment&gt; segments;
  for (int i = 0; i < loops; ++i)
    segments.push_back(random_segment());

  std::cout << "Counting intersections [brute force algorithm]: " << std::flush;
  std::vector<Point> points;
  for (int i = 0; i < loops-1; ++i)
    for (int j = i+1; j < loops; ++j) {
      Object obj = intersection(segments[i], segments[j]);
      if (const Point* pt = object_cast<Point>(&obj))
        points.push_back(*pt);
    }
  std::cout << points.size() << std::endl;

  // we shuffle the points, as consecutive points have good chance to come
  // from the same segments, hence filter failures in orientation() later...
  std::random_shuffle(points.begin(), points.end());

  std::cout << "Performing orientation tests" << std::endl;
  int negative_ort = 0, positive_ort = 0, collinear_ort = 0;
  for (int i=0; i < points.size()-2; ++i) {
    Orientation o = orientation(points[i], points[i+1], points[i+2]);
    if (o < 0)      ++negative_ort;
    else if (o > 0) ++positive_ort;
    else            ++collinear_ort;
  }
  std::cout << "orientation results : (-) = " << negative_ort
                              << "    (+) = " << positive_ort
                              << "    (0) = " << collinear_ort << std::endl;

  t.stop();
  std::cout << "Total time   = " << t.time() << std::endl;
  std::cout << "Total memory = " << ((Memory_sizer().virtual_size() - init_mem) >>10)
                                 << " KB" << std::endl;
}
```

inria-00344960, version 1 - 7 Dec 2008