LOOK
A Lazy Object-Oriented Kernel for Geometric Computation*

Stefan Funke *Kurt Mehlhorn*
Max-Planck-Institut für Informatik
Im Stadtwald
66123 Saarbrücken
Germany
{funke,mehlhorn}@mpi-sb.mpg.de

Abstract

In this paper we describe and discuss a new kernel design for geometric computation in the plane. It combines different kinds of floating-point filter techniques and a lazy evaluation scheme with the exact number types provided by LEDA allowing for efficient and exact computation with rational and algebraic geometric objects.

It is the first kernel design which uses floating-point filter techniques on the level of geometric constructions.

The experiments we present – partly using the CGAL framework – show a great improvement in speed and – maybe even more important for practical applications – memory consumption when dealing with more complex geometric computations.

1 Introduction

Geometric algorithms are usually designed for the so-called Real RAM, a random access machine that can handle real numbers at unit cost. The *exact computation paradigm* ([16]) advocates to give the implementer of a geometric algorithm the illusion of a real RAM by providing exact number types and exact geometric predicates. The geometric kernels of the popular libraries CGAL ([4]) and LEDA ([14]) are based on the exact computation paradigm.

The evaluation of a geometric predicate amounts to the computation of the sign of an arithmetic expression. The naive way to compute the sign of an expression is to compute the value of the expression (using exact arithmetic) and to read off the sign from the value. A much more efficient technique is the use of *floating-point filters* [10, 13]. A floating-point filter computes an approximate value of an expression and a bound for the maximal deviation from the true value. If the error bound is smaller than the absolute value of the approximation, approximation and exact value have the same sign and hence the sign of the approximation may be returned. In this way the true sign can be obtained quickly. The geometry kernels of CGAL and LEDA use floating-point filters. The advocates of floating-point filters claim that filters at the predicate level realize the exact computation paradigm at little cost; the running time is claimed to be no more than twice the running time of a pure floating-point implementation.

This is true as long as the algorithms do not construct new geometric objects. The class of such algorithms is limited: convex hulls and Delaunay diagrams belong to the class, but already line segment intersection does not. What is the problem with the construction of new objects? In the kernels mentioned, points (and all other geometric objects) are represented by their coordinates (either cartesian or homogeneous) and hence the construction of an object requires the *exact* computation of its coordinates.

The exact computation of object coordinates has two undesirable consequences. On one hand the running time increases dramatically when operating with constructed objects due to the increased numerical complexity, but maybe even more importantly, the space consumption grows rapidly. So in practice, users either write their own routines to allow for filtering on construction level (which can be supported by tools like EXPCOMP [6]) or use rounding schemes after every construction to keep the numerical complexity and space requirements low, see [9] for example. Note though, that these rounding schemes actually require a proof that they do not affect the final result considerably. These proofs are non-trivial and usually cannot be generalized. So each application has to be considered separately.

In this paper we present a new kernel called LOOK (Lazy Object-Oriented Kernel for geometric computation) which supports filtering also for geometric constructions. The main idea is that geometric objects are not represented by their coordinates, but by the geometric operation that produced them. Hence exact computation of the coordinates is still possible, but does not have to be performed on construction, and only if needed at all.

*This research was partially supported by ESPRIT LTR project 28155 (GALIA) and a scholarship 'Graduiertenkolleg' by the Deutsche Forschungsgesellschaft.

In section 2 we will briefly survey the features of LOOK which are fairly standard and match existing kernels for geometric computation. The third section is the core of this paper, as it discusses in detail the concepts of the implementation that allow for filtering on geometric object level. Section 4 shows how to use and extend the framework provided by LOOK. Finally, the last section gives extensive experimental results, which show the benefits of using LOOK.

2 Features of LOOK

The features of LOOK in terms of geometric objects and operations are fairly standard. Like the LEDA RatKernel [14], the standard geometric kernel in LEDA, which builds upon exact homogeneous integer representation, or the CGAL kernels, LOOK provides class representations for all basic geometric objects like points, lines, segments, circles, etc. In LOOK they are called `OPOINT, OLINE, ...`

Similar to the CGAL kernels when parametrized with `leda_real` as representation type ([5]), LOOK also provides support for computation with algebraic geometric objects as they occur for example when intersecting circles.

The geometric constructions supported by LOOK currently include line (segment) intersection, computing the center of a triangle, intersection of circles, etc. Of course, all common geometric predicates like orientation test, incircle test etc. are also available.

Programmers can easily replace the default LEDA RatKernel in LEDA's geometric algorithms by LOOK. Due to the genericity of CGAL, it is also no problem to use LOOK as a kernel implementation for the algorithms in CGAL.

3 Implementation Concepts

What distinguishes our kernel from other existing kernels is the idea of 'bookkeeping' on object level. This approach was already suggested in [10] as 'lazy constructor evaluation', but so far if a programmer wanted to use it, he had to hand-code everything — from the object representations to the predicates. We have developed a tool called EXPCOMP [6] which supports implementations with lazy constructor evaluation but it does not provide a general framework for geometric computation; the programmer still has to implement all predicates and constructions from scratch.

The main idea of our implementation is the following: when we construct a geometric object, we do not compute its coordinates exactly, but only their floating-point approximations and also store references to the objects which were involved in the construction. If later a more precise or exact coordinate representation of the object is required, we can compute this using this 'history' of the object. So with every geometric object we have a so called *object dependency graph* associated, which is a directed acyclic graph recording the 'construction' of the object. Every node of this directed acyclic graph corresponds to a geometric construction. Figure 1 shows the object dependency graph for a point object

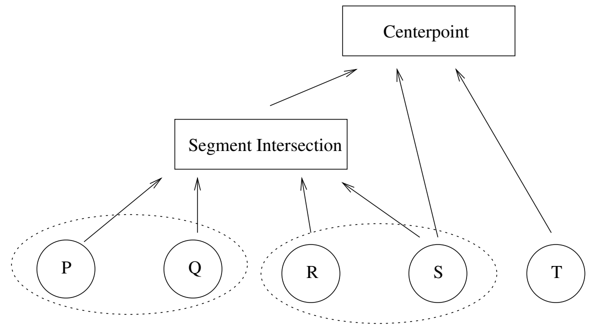

Figure 1: Object dependency graph for a segment intersection followed by a center computation

resulting from a line segment intersection of two segments $\overline{PQ}$ and $\overline{RS}$ followed by a centerpoint computation of the intersection point and S and T.

The same approach, but on the level of operators in arithmetic expressions is taken in the number type LEDA `real`. First, one only computes a floating-point approximation for the expression, but also builds an acyclic directed graph recording its 'construction' with the nodes corresponding to arithmetic operators. If the current floating-point approximation of an expression value does not suffice e.g. for a sign determination, one can recompute a better approximation using this 'history' information. But as it was already mentioned in [10] and can be seen in [5], this 'bookkeeping' on arithmetic operator level has its cost. An overhead factor of around 10–50 can be expected when compared to pure floating-point computation.

When performing 'bookkeeping' on object level, the overhead occurs only once for every geometric construction instead of every arithmetic operation, so we can expect a considerable gain in running time. But what can we expect when comparing with the present kernels for rational geometric objects like the LEDA RatKernel or the parametrized kernels of CGAL? Clearly there is a potential for improving the performance, too, as they *always* perform geometric constructions exactly using a datatype for exact/arbitrary precision arithmetic; only the predicate evaluations are tuned using floating-point filter mechanisms. Especially when we deal with deeply nested geometric constructions, where the exact constructions get very expensive, our approach should pay off.

When designing a geometric kernel, it is very important to wrap these techniques as transparent as possible to the programmer. In the following we will touch upon the main issues in our kernel implementation and how it is wrapped up. Note that the core object of our kernel is the point (we call it `OPOINT`), since all other geometric objects currently present in our kernel are represented using a tuple of points.

3.1 Floating-point filters

The use of floating-point filters to speed up the exact evaluation of predicates has become standard when implementing

geometric algorithms. If a predicate is expressed as the sign of an arithmetic expression, the idea is first to evaluate this expression using floating-point arithmetic but also compute an error bound to determine whether the outcome reliable.

According to the way this error bound is computed, we can classify floating-point filters into *static* (error bound is computed completely before runtime), *semi-static* (the error bound is partly computed before runtime) and *dynamic* (the error bound is computed completely at runtime). Obviously, static error bounds incur the least overhead at runtime but tend to be rather weak, whereas the dynamic bounds are typically pretty tight but incur a larger overhead. Examples for these techniques can be found in [3, 6, 7, 10, 13]

As we are dealing with possibly nested computations, using static filters seems to be a bad idea because after few levels of computation their error bounds get very imprecise. So we decided to use dynamic filtering techniques for all constructions and predicates, but also use semi-static filters as the very first step in predicate evaluations. In the actual implementation, we use the interval filters from [3] which make use of the IEEE rounding modes to keep the overhead compared to floating-point computation relatively low (around a factor of 4–6) and – for some predicates – the preprocessing tool EXPCOMP [6] which automatically generates semi-static filter code (which usually has an overhead of only about 1–2). For this purpose, we had to slightly modify the version of EXPCOMP presented in [6] to make it compatible with the interval datatype from [3].

3.2 Cartesian and Homogeneous Representations

When designing a geometric kernel, one can choose between a representation of the points in the plane using *cartesian* or *homogeneous* coordinates. With cartesian coordinates, a point is represented by two (integer, rational or algebraic) values p_x, p_y. With homogeneous coordinates, the point is represented by three values p_X, p_Y, p_W where $p_x = p_X/p_W$ and $p_y = p_Y/p_W$.

One advantage of the homogeneous representation is the fact that if we restrict ourselves to rational objects, we can perform all computations using integer arithmetic without divisions. This is the approach taken in the LEDA RatKernel. There is one problem, though: if the computation is nested, the homogeneous coordinates tend to get very large. Consider the following example:

Given three points p, q, r represented using homogeneous integer coordinates of bitlength 16. Compute the center c of these points. The expressions for computing the homogeneous coordinates of c have degree 6, so the bitlength of c_X, c_Y,c_W will be around 96.

This becomes especially a problem if we use this homogeneous representation for the floating-point approximation of c as well. Since an incircle test expressed in homogeneous coordinates has a degree of 12, we would get a value of bitlength more than 1100! Remembering that a variable of type `double` according to the IEEE standard [12] can have a value of at most 2^{1024}, we see that in these cases, a floating-point filter will *always* fail. Note that this failure is not due to geometric 'difficulty' but only due to the numerical representation.

For this reason we decided to compute the floating-point approximation of a point using cartesian coordinates first, and only if necessary, compute its homogeneous floating-point coordinates (as in some cases they allow for detection of degenerate cases with the floating-point filter, which is not as easy with the cartesian representation). With the cartesian representation, only predicates difficult in a *geometric* sense (i.e. almost or exactly collinear, cocircular, etc.) cannot be decided by the filter.

For the exact representations we decided to store homogeneous `leda_integer` coordinates and cartesian `leda_real` coordinates. Remember though, that – as for the homogeneous floating-point approximations – memory for these representations is only allocated if their value is actually requested and computed. We will go into more detail how this works in section 3.4.

Furthermore, we have ensured that arbitrary precision computations are performed using `leda_integer` arithmetic as long as possible. So only objects which have square roots involved in their construction may have `leda_real` representations.

3.3 Reference Counting and Handles

Before we define an actual object representation for our points, we have to think about some requirements. It is clear that the points we are dealing with may be defined by input data, but they also might be the result of some geometric construction. Nevertheless, when we use a point later on, we shouldn't have to care about that. Current geometry kernels solve this problem by simply computing the coordinates of the point using exact arithmetic; then there is no difference to a point defined by input data with the same coordinates. But as we do not want to perform the constructions exactly, we have to think about something different.

A first approach would be to write an abstract base class `base_point` which defines the properties of a point. Then for every possible construction, we derive a specialized class from `base_point` . For example a class `doubleC_point` representing points initialized by cartesian double coordinates fills in the necessary functionality based on the double precision input data it is initialized with, another example would be a class `Intersection_Point` , also derived from `base_point` , which fills in the functionality using the data computed by the intersection of two lines.

With this scheme, we could write all predicates in terms of the abstract base class `base_point` and still plug in different 'derivates' into the predicates. Nevertheless there are some caveats when using this 'direct' scheme especially with memory management which make it inconvenient to use.

One possible solution for this problem is the use of *ref-*

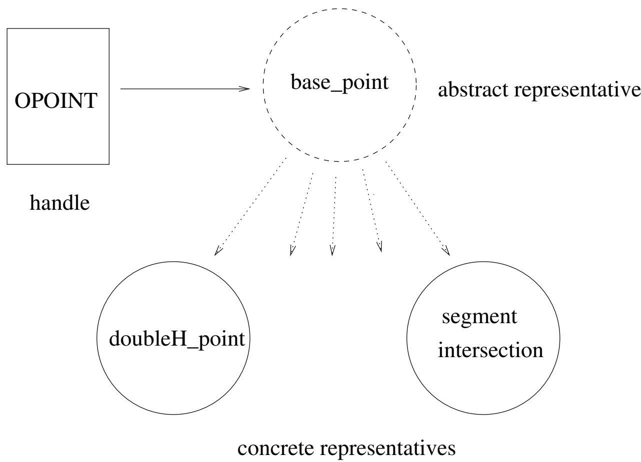

Figure 2: Object classes in LOOK

erence counting. For every distinct point object there exists exactly one *representative* and the programmer can access this representative only via a so called *handle*. If an assignment takes place (which is between handles, then), only the pointer to the representative is copied and a reference counter associated with it is incremented. The representative lives as long as it is referenced by at least one handle. The advantage of this scheme, apart from faster assignments of large objects, is the simplification in memory handling. And, of course, if the same object is copied frequently, the memory consumption is also much lower.

In our concrete implementation, we have a handle class `OPOINT` , which basically just contains a pointer of type `base_point` . `base_point` itself is the abstract base class for all point types. Figure 2 gives an overview of the class scheme in LOOK.

So if one wants to add a construction to LOOK (for example computing the midpoint of two points), one has to derive from the abstract base class `base_point` and implement the necessary virtual member functions. As we will see in section 4, these member functions basically describe how this midpoint is computed using different arithmetic datatypes.

3.4 Lazy Evaluation

3.4.1 Constructions

As mentioned, we use a lazy evaluation scheme for all geometric objects in our kernel. This implies that on construction of a geometric object, we only compute its floating-point approximation and store references to its defining objects. Only if later on, the exact coordinates as `leda_real` or `leda_integer` are requested, they are computed using the information about the defining objects. Of course, this usually triggers an exact evaluation of these defining objects as well.

To wrap the lazy-evaluation functionality in a transparent manner, we allow the programmer to access the coordinates of a geometric object only using member functions. The member function for the exact integer X-Coordinate looks as follows:

```
leda_integer base_point::X()
{
  if (!(Status&COMPUTED_HEXACT))
    SharpenHEXACT();
  return  Ext->_XCoord;
}
```

It is first checked, whether we already have computed the exact homogeneous coordinates, and if not, we compute them by calling the `SharpenHEXACT()` member function. `SharpenHEXACT()` again is a wrapper function:

```
void base_point::SharpenHEXACT()
{
  if (Ext==NULL)  // memory allocation necessary ?
    Ext=new ExtendedBlock;

  ComputeHExact();

  // sharpen homog. FP-APX
  Ext->_XCoordAPX=INTERVAL(Ext->_XCoord);
  Ext->_YCoordAPX=INTERVAL(Ext->_YCoord);
  Ext->_WCoordAPX=INTERVAL(Ext->_WCoord);

  // sharpen cart. FP-APX
  leda_rational xR(Ext->_XCoord,Ext->_WCoord),
                yR(Ext->_YCoord,Ext->_WCoord);
  __xCoordAPX=INTERVAL(xR);
  __yCoordAPX=INTERVAL(yR);

  Status|=(COMPUTED_HAPX|COMPUTED_HEXACT
          |COMPUTED_CAPX);
  Birthday++;
}
```

We first check if we have to allocate the memory for the extended datastructure, since by default we only store (and allocate memory for) the cartesian double approximation of an object. Then we call the actual function computing the coordinates and use this exact value to sharpen the floating-point approximations of the current object. Finally we increment the 'birthday' of the current object. The next section will explain the purpose of that in more detail.

Note that the code for triggering the lazy evaluation mechanism is completely encapsulated in the `::X()` and `::SharpenHEXACT()` member functions. If a new class is derived from `base_point` , the programmer only has to provide the member functions which actually compute the coordinates (here: `ComputeHExact()`).

To sum it up, every point object derived from `base_point` in our kernel provides the following member functions for accessing the coordinates in different formats and types:

name	Type	Format
`::x() ::y()`	`leda_real`	cartesian
`::xAPX() ::yAPX()`	`INTERVAL`	cartesian
`::X() ::Y() ::W()`	`leda_integer`	homog.
`::XAPX() ::YAPX() ::WAPX()`	`INTERVAL`	homog.

Keep in mind that because of the lazy evaluation mechanism, the corresponding values are only computed when requested by a call to one of these member functions [1].

3.4.2 Predicates

Of course, the predicates must also know about the lazy-evaluation mechanism in the objects they are working on. We will briefly sketch how we implemented the very common orientation test. Note that this code fragment includes special statements which are preprocessed by EXPCOMP ([6]) – a tool for automatically generating efficient floating-point filter code.

```
int orientation(const OPOINT &a, const OPOINT &b,
                const OPOINT &c)
{
 int sgn_res=NO_IDEA;
 BEGIN_FILTER // 1st stage generated by EXPCOMP
 {
 DECLARE_ATTRIBUTES real_apx_type FOR a.x() a.y()
                              b.x() b.y() c.x() c.y();
   exact AX=a.x(); exact AY=a.y();
   exact BX=b.x(); exact BY=b.y();
   exact CX=c.x(); exact CY=c.y();
   exact D=(AX-BX)*(AY-CY)-(AY-BY)*(AX-CX);
   sgn_res=sign(D);
 }
 END_FILTER

 if (sgn_res==NO_IDEA)   // 2nd stage
 {
   INTERVAL AX=a.xAPX(); INTERVAL AY=a.yAPX();
   INTERVAL BX=b.xAPX(); INTERVAL BY=b.yAPX();
   INTERVAL CX=c.xAPX(); INTERVAL CY=c.yAPX();
   fpu::round_up();
   INTERVAL D=(AX-BX)*(AY-CY)-(AY-BY)*(AX-CX);
   sgn_res=msign(D);
   fpu::round_nearest();
 }
 if (sgn_res==NO_IDEA)  // 3rd stage
 {
  if ((a.RatType() && b.RatType() && c.RatType())
  {
     /*homogeneous test using           */
     /*interval-arithmetic              */
     /*accessing a.XAPX() .... c.WAPX() */
    if (sgn_res==NO_IDEA)
      /*homogeneous test using exact    */
      /*exact integer arithmetic        */
  }
  else                  // 4th stage
   sgn_res=sign((a.x()-b.x())*(a.y()-c.y())
             -(a.y()-b.y())*(a.x()-c.x()));
 }
 return sgn_res;
}
```

The evaluation strategy is as follows:

1. cartesian evaluation using a fast filter implementation generated by EXPCOMP (this accesses the `::xAPX()` and `::yAPX()` member functions)

[1] In the actual implementation we always compute the cartesian fp approximation on instantiation and hence can save a redirection for this case.

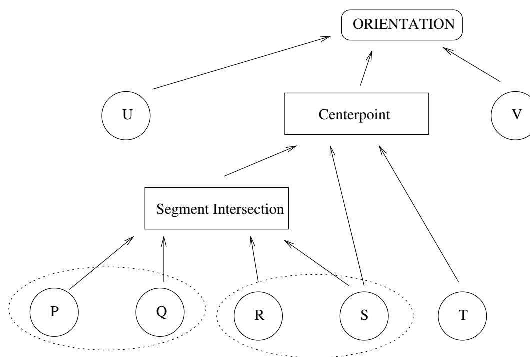

Figure 3: Object dependency graph for an orientation test

2. cartesian evaluation using interval arithmetic (tighter error bounds, but slower; accesses the `::xAPX()` and `::yAPX()` member functions)

3. If the involved objects are all of rational type:

   (a) homogeneous evaluation using interval arithmetic (accesses the `::XAPX()`, `::YAPX()` and `::WAPX()` member functions)

   (b) homogeneous evaluation using exact LEDA `integer` arithmetic (accesses the `::XAPX()`, `::YAPX()` and `::WAPX()` member functions)

4. If they are of algebraic type:

   (a) cartesian evaluation using exact LEDA `real` arithmetic (accesses the `::x()` and `::y()` member functions)

So only if a predicate cannot be decided by the earlier stages, additional time and space is spent on a more accurate computation of the involved objects and the predicate itself.

3.5 Progressive Exact Evaluation

Using the lazy evaluation scheme, we first try to evaluate the predicate using floating-point arithmetic and if the outcome cannot be proved to be correct, we trigger an exact computation for all objects involved in that predicate (which in turn triggers exact computations of their 'defining' objects) and then evaluate the predicate using exact arithmetic. But can we do better?

Let us first extend the definition of the object dependency graph and also allow geometric predicates as nodes (actually we only allow them as the root of an odg). Consider the object dependency graph in figure 3. Let us define the depth of an object in the object dependency graph as the maximum length of a path from the root to that object.

So far, if the floating-point evaluation of the orientation test fails, exact arithmetic computations of all objects involved (P, Q, R, S, T, U, V, intersection point, centerpoint) is triggered. But it may help to compute the coordinates of the objects on depth 2 (here: the intersection point) exactly,

improve their floating-point approximations using the exact values and then recompute the floating-point approximations of the 'higher' objects (here: the centerpoint). Since we get better error bounds now, this may suffice to decide the orientation predicate. Hence, if the floating-point evaluation of a predicate fails, we proceed as follows:

1. determine the deepest node in the object dependency graph which has not been evaluated exactly; assume its depth is d; if $d = 1$, evaluate the predicate exactly
2. evaluate all objects with depth d exactly and improve their floating-point approximations
3. reevaluate the floating-point approximations of all objects with distance less than d to the root
4. if no decision could be made, start from 1. again

Note that actually 'exact evaluation' is possible only for rational objects, nevertheless we can do something similar for algebraic objects by improving their approximation up to double precision and then using this value for the later floating-point computations.

To allow for an efficient implementation, we store a *birthdate* for every object. Objects built of input data have birthdate 0. When a new geometric object is constructed (and its floating-point approximation is computed), it gets as birthdate the sum of the birthdates of its defining objects. When a object is evaluated exactly and its floating-point approximation is sharpened according to this exact value, its birthday is incremented by one. Then, in the process of progressive exact evaluation, if an object realizes that the sum of the birthdates of its defining objects is greater than its own birthdate, it knows that it can improve its floating-point approximation by just recomputing using the improved floating-point approximations of its defining objects.

We skip this additional code in the predicate evaluation, but it only adds a few lines to the code given for the orientation test.

3.6 Conservative Memory Management

So far our strategy is to perform exact computation only on demand, but once computed keep the result such that if later the same result is required, no recomputation is necessary. For some applications, though, where memory consumption is an issue, it may be more appropriate not to keep the exact results, i.e. after a predicate evaluation which possibly triggered a sequence of arbitrary arithmetic computations, we discard the arbitrary precision results but of course keep the refined floating-point approximations.

We have incorporated a simple toggle in the LOOK kernel to switch between discarding and keeping results computed using arbitrary precision arithmetic. If a predicate evaluation required arbitrary precision arithmetic and the 'discard' toggle is set, it recursively frees all extended blocks allocated during the evaluation of the predicate.

As it turns out, there are cases where discarding the computed results results in a better runtime than keeping them. This is probably due to the cache getting less effective, if memory consumption is very high.

4 How to Use it as a Programmer

4.1 'Normal' Use

The nice thing about LOOK is the fact that it encapsulates and hides all of the above mechanisms from the programmer. If he does not want to introduce new geometric constructions or predicates, he can use the geometric object representations provided by LOOK in exactly the same manner as if he was using a geometric kernel of LEDA or CGAL.

4.2 Extending LOOK

Nevertheless, adding functionality to LOOK is not hard either. For instance adding additional predicates is very easy, given the examples already present in LOOK. They show how the different floating-point filter stages are combined to get best performance. For very efficient code, we also recommend using EXPCOMP for the first filter stage as it was done in case of the orientation predicate.

But even if she wants to extend LOOK beyond the geometric constructions already provided, the effort is not much more than writing these constructions for another kernel, for example the LEDA or CGAL kernels. Basically she only has to derive a new point class from the abstract class `base_point` and implement the member functions which actually compute the coordinates of the new point. With little more effort, she can also support the progressive exact evaluation code. In the following we will consider the simple example of line intersections.

4.2.1 Example: Intersection of Lines

For sake of simplicity, we neglect the case where the lines are parallel or identical. Of course, in the actual implementation these cases are treated as well.

We first derive a new class from the abstract representation class `base_point_rep` . An instance of this new class `line_intersection` upon instantiation stores the defining lines, determines its type (either rational or algebraic), triggers the computation of its approximated cartesian coordinates and stores its 'birthday' which is needed for the progressive exact evaluation mechanism.

```
class line_intersection : public base_point_rep
{
  OLINE L1, L2;

  public:
  line_intersection(const OLINE &l1,
                    const OLINE &l2)
  {  L1=l1;L2=l2; SharpenCAPX();
     if (L1.RatType() &&
         L2.RatType())
       Status|=RAT_TYPE;
     Birthday=L1.Birthday()+L2.Birthday();
```

```
  }

  private:
    virtual int ComputeHExact();
    virtual int ComputeHApprox();
    virtual int ComputeCExact();
    virtual int ComputeCApprox();

    virtual void Reincarnate();    //optional
    virtual int DeepestInexact();  //optional
    virtual void SharpenAtDepth(int d); //optional
};
```

The programmer only has to implement the virtual `ComputeXXX()` member functions as they are called from the lazy-evaluation mechanism of the parent `base_point_rep` class. Optionally, if the new class should also make use of the progressive exact evaluation scheme, implementations for the `Reincarnate()`, `DeepestInexact()` and `SharpenAtDepth()` member functions can be given, though these implementations are almost generic and only depend on the number and type of 'defining' objects.

As we neglect the degenerate cases of identical or parallel lines, the implementation of our `IntersectLines()` [2] function is trivial. We just initialize a new `OPOINT`-handle with the representative of the line intersection:

```
int IntersectLines(const OLINE &l1,
                   const OLINE &l2,
                   OPOINT &s)
{
  s=OPOINT(new line_intersection(l1,l2));
  return 1;
}
```

Of course, what is really interesting now, are the implementations of the virtual `ComputeXXX()` member functions.

```
int line_intersection::ComputeCExact()
{
  OPOINT S1=L1.Source(), T1=L1.Target(),
         S2=L2.Source(), T2=L2.Target();

  leda_real dx1=T1.x()-S1.x();
  leda_real dy1=T1.y()-S1.y();
  leda_real dx2=T2.x()-S2.x();
  leda_real dy2=T2.y()-S2.y();
  leda_real w=dy1*dx2-dx1*dy2;

  leda_real c1=T1.x()*S1.y() -
               S1.x()*T1.y();
  leda_real c2=T2.x()*S2.y() -
               S2.x()*T2.y();

  _xCoord()=(c2*dx1-c1*dx2)/w;
  _yCoord()=(c2*dy1-c1*dy2)/w;
  return 1;
}
```

and

```
int line_intersection::ComputeCApprox()
{
  OPOINT S1=L1.Source(), T1=L1.Target(),
         S2=L2.Source(), T2=L2.Target();

  INTERVAL t1x=T1.xAPX(), t1y=T1.yAPX();
  INTERVAL s1x=S1.xAPX(), s1y=S1.yAPX();
  INTERVAL t2x=T2.xAPX(), t2y=T2.yAPX();
  INTERVAL s2x=S2.xAPX(), s2y=S2.yAPX();

  fpu::round_up();
  INTERVAL dx1=t1x-s1x, dy1=t1y-s1y;
  INTERVAL dx2=t2x-s2x, dy2=t2y-s2y;
  INTERVAL w=dy1*dx2-dx1*dy2;

  INTERVAL c1 = t1x*s1y - s1x*t1y;
  INTERVAL c2 = t2x*s2y - s2x*t2y;


  _xCoordAPX()=(c2*dx1-c1*dx2)/w;
  _yCoordAPX()=(c2*dy1-c1*dy2)/w;
  fpu::round_nearest();
  return 1;
}
```

[2] the only thing of this section a 'normal' programmer will ever see

The code for the pair `ComputeHExact()` / `ComputeHApprox()` is analogous. In fact, the code for computing the approximation is in most cases just a copy of the 'exact' code with the arbitrary precision number type replaced by the interval type.

Note that in the code for computing the approximations, we have to switch rounding mode of the floating-point unit before doing any calculations as the interval type `INTERVAL` relies on the correct IEEE rounding mode when determining upper and lower bounds of the intervals (there is also a version, where the switch of the rounding mode is done implicitly, but it is considerably slower as it has to be done before and after every arithmetic operation), see [3] for more details.

By just providing the above code, points generated by intersection of lines can benefit from all advantages of our framework; in particular, an `OPOINT` constructed using `IntersectLines(...)` can be plugged into any predicate within the LOOK kernel. If the predicate turns out to be 'difficult' the exact evaluation of the intersection point is automatically triggered. The code allowing for that is all inherited from `class base_point_rep`, so the programmer does not have worry about that.

To be complete we state the missing code for the optional `Reincarnate()`, `DeepestInexact()` and `SharpenAtDepth()` member functions.

```
virtual int line_intersection::Reincarnate()
{
  if (Status&COMPUTED_HEXACT)
    return 0;
  L1.Reincarnate(); L2.Reincarnate();
  int NewBirthday=L1.Birthday()+L2.Birthday();

  if (NewBirthday>Birthday)
  {
    Birthday=NewBirthday;
    ComputeCApprox();
    return 1;
  }
```

```
  else return 0;
}

virtual int line_intersection::DeepestInexact()
{
  if (Status&COMPUTED_HEXACT) return 0;
  int l1=L1.DeepestInexact();
  int l2=L2.DeepestInexact();

  return MAX(l1,l2)+1;
}

virtual void line_intersection::SharpenAtDepth(int d)
{
  if (d==1) SharpenHEXACT();
  else if (d>1)
  {
    L1.SharpenAtDepth(d-1);
    L2.SharpenAtDepth(d-1);
  }
}
```

Just as a remark, the implementation of the `IntersectSegments()` function is trivial when reducing it to the line intersection:

```
int IntersectSegments(const OSEGMENT &s1,
      const OSEGMENT &s2, OPOINT & p)
{
  int s1_s, s1_t, s2_s, s2_t;
  s1_s=orientation(s1,s2.Source());
  s1_t=orientation(s1,s2.Target());
  s2_s=orientation(s2,s1.Source());
  s2_t=orientation(s2,s1.Target());

  if ((s1_s!=s1_t) && (s2_s!=s2_t))
  {
    p=OPOINT(new line_intersection(s1,s2));
    return 1;
  }
  return 0;
}
```

Note that the orientation tests called from this function are, of course, fully filtered.

5 Experiments

In this section we compare implementations of geometric algorithms using our LOOK kernel with implementations based on the LEDA RatKernel and the CGAL kernel. The test platform was a Sun UltraSparc 333 Mhz with 128 MB RAM running Solaris 2.7. We used `g++` 2.95.2 and LEDA 4.0.

5.1 LOOK compared to LEDA's RatKernel

In our first example, we examine how different geometry kernels behave when we increase the nesting depth of geometric constructions.

We have tested six different implementations of Dwyer's divide-and-conquer algorithm for computing the Delaunay triangulation of a set of points in the plane ([8]). First one using the floating-point kernel of LEDA (FPKernel), then two variants using the rational kernel of LEDA. In both variants, constructions are always performed using exact integer arithmetic; in the first variant (which is the one in the current LEDA version), though, predicates are filtered using a floating-point filter based on homogeneous coordinates, whereas the second variant additionally incorporates a floating-point filter based on cartesian coordinate representation to overcome the problem of overflow as discussed in section 3.2.

Iter.	#Point Objects	# Extended blocks
2	299	0
3	848	245
4	2528	788
5	7562	4865

Table 2: Memory allocation

Finally there are three variants using the LOOK kernel. The first one keeps all results computed using arbitrary arithmetic and uses the progressive evaluation scheme; the second one deletes all results computed using arbitrary precision arithmetic after each predicate evaluation, but also uses the progressive evaluation scheme. Finally, the third variant keeps exact results, but does not use the progressive evaluation scheme.

As a benchmark we iterated Voronoi diagram computations. Starting with a point set S_0, in iteration i we determined S_{i+1} from S_i by computing the Delaunay triangulation of S_i and adding the circumcenters of all triangles. By this procedure, the number of elements in S_i roughly tripled in each iteration. Note that about $2/3$ of the elements in S_i are created in the $(i-1)$-th iteration, so most predicate evaluations involve data points constructed in the previous stage.

As S_0 we generated a set of N randomly distributed points on a 32bit integer grid such that the final iteration computes the Delaunay triangulation of about 8000 points. To keep the comparison between lazy construction in LOOK and the immediate exact arithmetic construction in the RatKernels fair, we always skipped the construction of the circumcenters in the last iteration. Table 1 shows the results for different numbers of iterations. We give both, the total running time and the time spent for constructing the circumcenters. Of course, the variants using LOOK use very little time for construction as they only store the involved objects and compute a floating-point approximation. But if later, during the computation of the Delaunay triangulation in the next iteration, the coordinates are requested in arbitrary precision arithmetic, the arbitrary precision computation is triggered.

In table 2 we see how many of the extended blocks (which store the coordinates of type `leda_integer`) are allocated after i rounds.

To compare the running time for 'easy' examples, we computed 2 iterations on a set of 10000 randomly generated points with 32bit integer coordinates. This computation is

No. of	N	FPKernel	RatKernel		LOOK		
Iterations			cart.FP	no cart.FP	MaxMem/PEE	MinMem/PEE	MinMem/no PEE
1	8100	(0/0.56)	(0/0.53)	(0/0.55)	(0/1.34)	(0/1.41)	(0/1.26)
2	2700	(0.09/0.52)	(0.42/1.11)	(0.43/1.22)	(0.11/1.89)	(0.12/2.20)	(0.14/1.98)
3	900	(0.11/0.56)	(0.75/1.84)	(0.83/73.1)	(0.15/2.34)	(0.14/3.21)	(0.14/2.99)
4	300	(0.11/0.53)	(3.63/103)	(3.95/836)	(0.15/4.16)	(0.16/6.61)	(0.14/12.0)
5	100	(0.12/0.55)	(33.6/1009)	(37.6/8065)	(0.15/24.23)	(0.16/21.60)	(0.15/78.48)

Table 1: Iterated Voronoi computations; time in secs. (construction/total)

FPKernel	RatKernel(1)	RatKernel(2)	LOOK
3.06	5.47	4.47	6.68

Table 3: Crust computation of 10000 points

RatKernel	LOOK	C<leda_real>	C<double>
0.38	1.00	3.02	0.16

Table 4: ConvexHull of 50000 32bit integer points

basically the *crust* computation as described in [1]. See table 3 for the results.

It turns out, that if the computation does not involve more than one level of construction, LOOK is about 2–3 times slower than the (tuned) LEDA RatKernel. With increasing nesting depth of the constructions, LOOK performs better and better compared to the LEDA RatKernel.

One reason for that is the use of the lazy evaluation scheme. As can be seen in table 2, only some of the constructed points have been evaluated exactly during the algorithm, whereas the RatKernels *always* perform the constructions using exact arithmetic. If we get to higher nesting depths than 5, though, exact arithmetic evaluation of almost all point objects has been triggered. A closer examination of the results showed that this is due to degeneracies which lead to difficult predicate evaluations triggering the exact evaluations of constructions. But nevertheless, the LOOK implementations are still much faster in these cases than the RatKernel implementations due to the improved filtering techniques (interval filters which yield closer error bounds than the semi-static filters used in the RatKernels).

Surprisingly, the LOOK variants using the memory saving scheme of deleting all arbitrary precision results after a predicate evaluation perform very well, especially with increasing nesting depth. Although the deletion of all intermediate arbitrary precision computations requires some recomputations, the reduced memory allocation seems to lead to a more efficient caching, so the overall running time is even better. Note that the progressive exact evaluation scheme helps considerably when using the memory saving scheme.

5.2 LOOK as a CGAL Kernel Traits Class

In the CGAL library [4] all algorithms and data structures are generic in the sense that a programmer can plug-in any geometric kernel that meets certain requirements into the algorithms provided by CGAL. But CGAL also provides templates for geometric kernels where the user only has to plug-in a number type that is used for the coordinate representation. For example, by plugging-in the number type `leda_real` for exact arithmetic with algebraic numbers into the cartesian kernel template, we get a kernel for exact computation with algebraic geometric objects.

We have accommodated LOOK to be compatible with CGAL's algorithms by wrapping it in a so-called kernel-traits class. In the following we will compare the performance of the LOOK kernel with other kernels when plugged into the CGAL algorithms. We have tested the following kernels:

- `LOOK` the LOOK kernel as CGAL plugin
- `RatKernel` the LEDA RatKernel as CGAL plugin
- `C<leda_real>` the CGAL cartesian kernel with `leda_real` representation
- `C<double>` the CGAL cartesian kernel with `double` representation [3]

As the first example we computed the convex hull of 50000 random points with 32bit integer coordinates using the default convex hull algorithm of CGAL. Note that this computation does not involve any geometric constructions, so we expect the RatKernel to perform best of the exact kernels. The results can be seen in table 4.

A more complex example is the computation of the convex hull of points which are not available as input data, but computed as the intersection of circles. Note that this computation involves square-roots and hence we cannot use the RatKernel for this experiment. Table 5 shows the result for the intersections of 500 circles (they have about 26000 intersection points). Also compare with the experimental results in [5].

As we have seen in the previous section, we have to pay a little bit for the more involved filtering-techniques, so for very simple examples, where no constructions take place, we lose about a factor of 2–3 compared to the RatKernel, but

[3] Note that this kernel does not guarantee exact computation.

	LOOK	C<leda_real>	C<double>
intersection	0.52	7.51	0.11
total	1.08	10.1	0.17

Table 5: ConvexHull of the intersections of 500 circles; cartesian integer coordinates

are still 3 times faster than the cartesian `leda_real` kernel. If constructions take place, though, as in the example for the convex hull of circle intersections, we gain a factor of about 10 compared to the cartesian `leda_real` kernel. As we have predicted in our discussion in the previous sections, this is due to reducing the bookkeeping overhead from expression level to geometric construction level.

6 Conclusion

We have presented LOOK, a lazy object-oriented kernel design for exact geometric computation. In contrast to previous kernels, LOOK supports various kinds of floating-point filter techniques both on predicate level as well as on construction level. If a problem involves many geometric constructions, LOOK performs about 3–40 times better than the LEDA RatKernel or CGAL kernels for exact computation.

The technique of bookkeeping on object level also allows for many evaluation strategies. If memory consumption is a big issue, one can keep this very low – even close to pure floating-point computation, as the arbitrary precision representation of at most one predicate evaluation is present in memory at any given time. To our very surprise, this approach of discarding all arbitrary precision results after a predicate evaluation performed quite well, especially for more complicated examples. So it seems as if memory allocation is not the main difficulty for deeply nested exact constructions.

Of course, LOOK is not a panacea for exact implementations of geometric algorithms. Although the advanced filtering techniques allow the decision of most predicates without resorting to exact arithmetic computations, the evaluation of really difficult predicates which require exact arithmetic, gets *very* expensive with increasing nesting depth. Here is a point where algorithmic changes may help. There is the idea to design algorithms with only low-degree predicates and thus reducing the numerical complexity, for example [2]. On the other hand one could try to reduce the number of arbitrary precision evaluations even further by allowing some of the predicates to err. Of course, as we want a correct final result, it all depends on *which* predicates we allow to err. In [11] we show that a very simple but powerful idea can reduce the number of arbitrary precision evaluations considerably, in particular in (almost) degenerate cases. Part of our future work will be devoted to combining this approach with LOOK to improve performance for deeply nested computations.

Furthermore we want to increase the number of 'algebraic' constructions in our kernel, e.g. the constructions for Voronoi nodes in the Voronoi diagram of points and line segments. Of course it may be also interesting to apply these ideas to construct a kernel for higher-dimensional geometric computation.

The source code of LOOK and all test programs (as well as the source of EXPCOMP) are available on the author's homepage `http://www.mpi-sb.mpg.de/~funke`.

References

[1] N. Amenta, M. Bern, D. Eppstein: *The Crust and the β-Skeleton: Combinatorial Curve Reconstruction*, (Proceedings of the 14th Symposium on Computational Geometry 1998, ACM Press)

[2] J.-D. Boissonat, J. Snoeyink: *Efficient algorithms for line and curve segment intersection using restricted predicates*, (Prooceedings of the 15th Symposium on CG 1999, ACM Press)

[3] H. Brönnimann, C. Burnikel, S. Pion: *Interval Analysis Yields Efficient Arithmetic Filters for Computational Geometry*, (Proceedings of the 14th Symposium on CG 1998, ACM Press)

[4] The CGAL project `http://www.cs.uu.nl/CGAL/`

[5] C. Burnikel, R. Fleischer, K. Mehlhorn, S. Schirra: *Efficient Exact Geometric Computation Made Easy*, (Proceedings of the 15th Symposium on CG 1999, ACM Press)

[6] C. Burnikel, S. Funke, M. Seel: *Exact Geometric Computation using Cascading*, (to appear in International Journal on Computational Geometry and Applications, preliminary version in Proceedings of the 14th Symposium on CG 1998, ACM Press)

[7] O. Devillers, P. Preparata: *A Probabilistic Analysis of the Power of Arithmetic Filters*, (Computational Geometry: Theory and Applications, Vol. 13, No.2, 1999)

[8] R.A. Dwyer: *A Faster Divide-and-Conquer Algorithm for Constructing Delaunay Triangulations*, (Algorithmica (1987), 2)

[9] S. Fortune: *Vertex-rounding a three-dimensional polyhedral subdivision*, (Proceedings of the 14th Symposium on CG 1998, ACM press)

[10] S. Fortune, C. van Wyk: *Static analysis yields efficient exact integer arithmetic for computational geometry*, (ACM Trans. Graphics, 15(3), pp. 223–248, July 1996).

[11] S. Funke, K. Mehlhorn, S.Näher: *Structural Filtering – A Paradigm for Efficient and Exact Geometric Programs*, (Proceedings of the 11th Canadian Conference on CG 1999)

[12] *IEEE standard 754-1985 for binary floating-point arithmetic*, reprinted in SIGPLAN 22, 2:9-25, 1987

[13] M. Karasick, D. Lieber, L. Nackmann: *Efficient Delaunay Triangulation using rational arithmetic*, (ACM Transactions on Graphics, 10(1):71-91, 1991)

[14] The LEDA homepage `http://www.mpi-sb.mpg.de/LEDA/`

[15] S. Schirra: *A case study on the cost of geometric computing*, (Proc. ALENEX99, 1999)

[16] C. K. Yap, T. Dube: *The exact computation paradigm*, (In D. Du and F. Hwang, editors, Computing in Euclidean Geometry, pages 452-492. World Scientific Press, 1995. 2nd edition.)