A Tutorial for Designing Flexible Geometric Algorithms

Vikas Kapoor[*] Dietmar Kühl[†] Alexander Wolff[‡]

Abstract

The implementation of an algorithm is faced with the issues efficiency, flexibility, and ease-of-use. In this paper, we suggest a design concept that greatly increases the flexibility of an implementation without sacrificing ease-of-use and efficiency.

We demonstrate the advantages of our concept through a C++ implementation of a simple rectangle-intersection algorithm, which follows the well-known sweep-line paradigm. We lead the reader, who should be familiar with C++ and its template mechanism, from a naive interface in a step-by-step guide to an interface offering full flexibility. The gain in flexibility can reduce the overall implementation effort by facilitating code reuse. Reusability in turn helps to achieve correctness simply because more users mean more testing.

Though most of the ingredients of our concept have already been suggested elsewhere, to our knowledge this is the first time that they are applied vigorously in a geometric setting. We include a thorough experimental analysis on random and real world data.

1 Introduction

When implementing an algorithm, especially in the initial design phase, the key issues are efficiency, flexibility, and ease-of-use. While focusing on flexibility, we demonstrate that there is no reason for sacrificing ease-of-use. We do an experimental evaluation to show that a gain in flexibility does not necessarily cause a loss in efficiency.

In this paper we present a design concept for geometric algorithms that is based on the *generic programming* paradigm. Generic programming is about making implementations more flexible by making them more general. Abstracting from concrete input or output data representation is an example of generic programming. In contrast to conventional programs, the parameters of generic programs are often quite rich in structure. Such parameters might be other programs, types or type constructors, or even other programming paradigms.

In this paper we will focus on the generic programming approach that has been introduced by the C++ *Standard Template Library*, or STL for short [MS96]. The STL was so successful that it soon became part of the C++ standard. The STL is a library of generic components, i.e. of algorithms, data containers, and *iterators* mediating between the former two. Iterators help to decouple algorithms from the type of data container they operate on. While iterators have been known before, the real novelty of the STL was the introduction of a requirements-based taxonomy of iterators, which gives a guideline for full decoupling, and an implementation of this taxonomy using the C++ template mechanism.

A completely different view of genericity is that of inheriting from an abstract base class. This so-called *polymorphism* also allows a programmer to abstract from a concrete representation of input and output data. Polymorphism is ubiquitous in object-oriented design, but in this paper we want to give an example that shows that STL-style template programming can be an interesting alternative. More information about the pros and cons of the two approaches can be found in [SLL99, FGK$^+$00].

[*] datango AG, Berlin, Germany, www.datango.de, `vikas@datango.de`

[†] Phaidros Software AG, Ilmenau, Germany, www.phaidros.de, `dietmar_kuehl@yahoo.com`

[‡] Institut für Mathematik und Informatik, Universität Greifswald, Germany, `awolff@uni-greifswald.de`

After the introduction of the STL further generic programming "tools" such as *data accessors* have been suggested in the C++ literature to help programmers make the interfaces to their implementations even more flexible [Küh96, Wei97]. Data accessors are a means to further decouple an implementation from the representation of input and output data [KW97]. So far, these extensions have been applied predominantly to graph problems [NW96]. Exceptions such as [Wei98, Ket99] deal with the representation of geometric objects, not with the design of geometric *algorithms*, our main interest here.

In order to show the relevance of STL-style generic programming including later extensions as data accessors for geometric algorithms, we investigate a simple rectangle-intersection algorithm that follows the well-known sweep-line paradigm. Using this example we lead the reader step by step from an inflexible, naive interface to a truly flexible interface that supports code reuse. We expect the reader to be familiar with C++ (that we used for the code fragments in our tutorial) and its template mechanism that is essential for STL-style generic programming. The aforementioned steps of our tutorial reflect our own change of perspective during the implementation of algorithms for a map-labeling project. Note that reusability helps to lower implementation costs in the long run and to achieve correctness—for the simple reason that more users mean more testing.

In order to support our concept with experimental data, we implemented a sweep-line algorithm for the rectangle-intersection problem in C++ in two ways: (a) based on the naive initial interface of our tutorial and (b) following the design concepts that we present here. We compare the running times of both implementations, as well as the size of their source code and executables. The data we used for the runtime comparison stems from random generators as well as from real world instances. The source code of our implementations, the documentation of the interfaces, the test data and its description are accessible via the WWW[1].

While the ingredients of our concept are already known, to our knowledge this is the first time that they are applied so rigorously to a geometric problem, made accessible in the form of a tutorial, and accompanied by a thorough experimental analysis.

C++ libraries that offer geometric data types and algorithms, are LEDA, the Library of Efficient Data Types and Algorithms [MN89, MN99], and CGAL, the Computational Geometry Algorithms Library [Ove96]. A report about the design of CGAL includes a section about the pros and cons of generic versus object-oriented programming [FGK$^+$00]. Other than the STL, LEDA was designed having mostly ease-of-use rather than genericity in mind. The resulting short-comings in terms of interfacing with user-defined data structures are investigated in [Küh96]. This has led to the implementation of a LEDA extension package for graph iterators [NW96]. Similar to the LEDA extension package the Boost Graph Library, previously GGCL, the Generic Graph Components Library [SLL99, SLL00], implements a graph library using and extending STL-style generic programming techniques. The authors compare the runtime of some graph algorithms implemented in LEDA (without the extension package) and GGCL [SLL99]. They report that depth and breadth first search as well as Dijkstra's shortest-path algorithm are about 10 times faster in GGCL than in LEDA.

This paper is structured as follows. In Section 2, we describe our example algorithm for the rectangle-intersection problem. In Section 3, we present a naive interface for this algorithm, investigate its disadvantages and modify it step-by-step to a generic and thus flexible interface. Finally, in Section 4, we compare the implementations of the naive and the most flexible interface in terms of runtime.

2 Algorithm

We demonstrate our design concepts through the example of a sweep-line algorithm for detecting intersections among axis-parallel rectangles in the plane [EM81]. For typical instances, see Figures 1 to 4. Our sweep line is a vertical line sweeping the plane from left to right. As usual, the sweep line is supported by two data structures, the *event-point schedule* and the *sweep-line status*.

Event points are the x-values where the sweep line must stop because either its status changes or inter-

[1]see `http://www.math-inf.uni-greifswald.de/map-labeling/design/`

sections have to be reported. In our case all event points are known before the sweep begins: they are the x-values of the left and right edges of the rectangles. Thus a sorted list suffices to implement our event-point schedule. An event point must be stored in such a way that it is clear whether it refers to the left or to the right edge of a rectangle.

The sweep-line status stores intervals corresponding to the intersections of the sweep line with the given rectangles. The endpoints of the intervals are the y-values of the lower and upper edges of the input rectangles. Initially the sweep-line status is empty. When a left edge of a rectangle is encountered during the sweep, the interval corresponding to the edge is inserted into the sweep-line status. A rectangle is reported if its interval is currently in the sweep-line status and intersects the new interval. When a right edge of a rectangle is encountered, the corresponding interval is deleted from the sweep-line status.

This reduces the rectangle intersection problem to the problem of maintaining a set of intervals such that intervals can be efficiently inserted and deleted, and interval-intersection queries can be answered quickly. To achieve this, we implement our sweep-line status with the interval-tree data structure [EM81]. We have implemented a semi-dynamic version that must be initialized with the endpoints of all intervals it is going to contain during the sweep. The preprocessing, namely sorting the points and building up an empty balanced tree, takes $O(n \log n)$ time, where n is the number of intervals. Inserting intervals can then be done in $O(\log n)$ time, deleting in constant time, while a query takes $O(k + \log n)$ steps, where k is the number of intervals reported. Since every interval is stored only once in the interval tree, the storage consumption is linear.

3 Step by Step Towards Good Design

In this section we start with a naive interface for the algorithm described in the previous section. Then we consider the limitations of this approach and show how these can be overcome. We will improve the straight-forward solution in several steps, each of which is independent of the others. Each refinement can be used to implement its predecessors without explicit data conversion.

We will take the viewpoint of a programmer who implements algorithms for a large number of programmers whose wishes and requirements cannot be anticipated in detail. We will refer to these clients as our *users* (as opposed to the end-users of some software package). Code fragments that show how a client could use our interfaces will be marked with the term “client code”. We hope to convince our readers that our design concept is worth being applied already in small projects where an interface is written for two or three users. Our concept is especially valuable in the long run when reusability starts to pay.

3.1 The Naive Approach

How would a naive interface for an implementation of the rectangle-intersection algorithm described above look like? Our first attempt is a procedure whose input consists of an array of rectangles and their number. The output could either consist of a list of pairs of intersecting rectangles, or, more user-friendly, of a so-called intersection graph. In this graph, each rectangle is represented by a node, and two nodes are connected by an edge if and only if the corresponding rectangles intersect. This is the interface sketched in Program 1.

```
class Rectangle;
class Graph;
Graph* rectangle_intersection( Rectangle* rectangle_array, int n);
```

Program 1: A naive functional interface.

This interface would force the programmer of the rectangle-intersection algorithm to implement the data structures `Rectangle` and `Graph`. On the other hand, it would force the user in most cases to convert his rectangles into those required by the interface, and, after calling `rectangle_intersection`, extract the desired information from the intersection graph. In order to do so, the user would have to study the interfaces of the input and output data structures. Another disadvantage of the naive interface is that it would not even allow the user to hand over rectangles in a *container* different from an array, like a list or a set.

3.2 Decoupling Algorithm and Data Organization

The most obvious disadvantage of our previous interface is the tight link between the algorithm and its input and output data structures. In order to decouple algorithm and data organization, we must solve two problems.

1. *Container independence:* in our case we would prefer not to force the user to hand over an *array* of rectangles.
2. *Representation independence* means not to require a fixed input or output format, but to accept any representation that fulfills certain requirements.

The first problem can be solved with the aid of *iterators*. Iterators are light-weight objects that point to other objects. As the name suggests, iterators are used to iterate over a range of objects: if an iterator points to an element in a range, it can be incremented so that it points to the next element or to an end-of-range marker. Iterators represent a versatile link between containers and algorithms. If an algorithm's interface takes iterators as arguments, then the algorithm can be applied to any container that provides access to its elements via iterators. STL iterators are a generalization of C pointers; one of the merits of the STL was to provide a complete hierarchy of iterators each of which has different requirements and abilities [MS96].

Consequently, our next interface will expect iterators to handle the input. Handling the output via iterators is not so simple since the intersection graph is not a linear structure. This problem is attacked in the following section. For the time being, we ask the user to provide his definition of a graph as a template parameter to our interface.

```
// client code
class User_Rectangle;               // rectangle type
class User_graph<User_Rectangle*>;  // graph type
```

Of course, this definition must fulfill some requirements. The implementation expects the following member functions for inserting nodes and edges.

```
// our requirements
typedef Graph::node_type node;
node Graph::insert_node( User_Rectangle*);
void Graph::insert_edge( node&, node&);
```

Note that the user is not forced to write his own graph data structure; he can use that of any library and write a simple wrapper that implements our requirements with the aid of the library graph. This approach also solves the second problem, i.e. representation independence, for the output. In the next section we will see that there is a more elegant solution which does not force the user to supply a graph data structure at all if the results are not to be stored in a graph.

In order to solve the second problem for the input, we use so-called *data accessors*. While iterators realize access to objects, data accessors are used to access the data associated with these objects [KW97]. Data accessors have two parts, a data accessing function, which is responsible for the actual access, and a light-weight object. This object is also referred to as the data accessor. It encapsulates the data type to be accessed and is used to select the correct data accessing function. Our next interface will require the user to provide such data accessors for his representation of the input data. Assume that the user declares the following `User_Rectangle` type.

```
// client code
typedef double CoordType;
struct  User_Rectangle { CoordType llx, lly, urx, ury; };
typedef User_Rectangle* User_iterator;
```

where `llx, lly, urx,` and `ury` represent the coordinates of the lower left and upper right corner of the rectangle, respectively. The data accessor for the x-coordinate of the lower left corner can then be defined as follows.

```
// client code
// - data accessor
struct LLXDA { typedef CoordType value_type };
// - data accessing function
CoordType get( LLXDA const& da, User_iterator const& it)
{ return (*it).llx; }
```

Program 2: A data accessor and a data accessing function for the x-coordinate of the lower left corner of an input rectangle.

Parameter overloading allows to implement four data accessing functions `get()` whose interfaces only differ in the type of their first parameter, the data accessor. All our algorithm needs to know about the user's rectangles are the coordinates of their corners. This is exactly what the data accessing functions supply. Note the interplay between data accessing function and its arguments in Program 3: the first parameter, the data accessor, selects the right data accessing function, while the second parameter, the iterator, selects the rectangle.

```
template < class Iterator, class Graph,
           class LLXDA, class LLYDA, class URXDA, class URYDA >
void rectangle_intersection( Iterator begin, Iterator end,
                             Graph& graph,
                             LLXDA llxda, LLYDA llyda,
                             URXDA urxda, URYDA uryda)
{
  for ( Iterator rect_it = begin; rect_it != end; ++rect_it) {
    typename LLXDA::value_type ll_x = get( llxda, rect_it);
    typename LLYDA::value_type ll_y = get( llyda, rect_it);
    typename URXDA::value_type ur_x = get( urxda, rect_it);
    typename URYDA::value_type ur_y = get( uryda, rect_it);
    // process the input coordinates
  }
  // construct the intersection graph: insert nodes
  Iterator rect_it1, rect_it2;
  typename Graph::node_type node1 = graph.insert_node( rect_it1),
                            node2 = graph.insert_node( rect_it2);
  // if the rectangles corresponding to rect_it1 and rect_it2 intersect,
  // draw an edge between node1 and node2 in the intersection graph
  if (...) graph.insert_edge( node1, node2);
}
```

Program 3: A functional, data-organization independent interface.

3.3 Tightening Control

Suppose the user of our implementation is only interested in a tiny fraction of the output, like the number of intersections or all rectangles intersecting a given rectangle. Or suppose he would like to abort the execution of the algorithm when the sweep line reaches a certain x-value or if some other condition becomes true. With the interface suggested in the previous subsections, he would not be able to take advantage of such a situation. It is clear that a functional implementation cannot provide such a degree of interaction between the user and the algorithm. Thus we will switch to a class interface, which allows our implementation to have a state and to offer the user more information about the algorithm's progress.

The key to more control is the *loop kernel* [Küh96, Wei97]. The loop kernel is a method which encapsulates the body of the central loop of the algorithm. It can be advanced in single steps and informs the user about the current state of the algorithm. The loop kernel makes the whole algorithm look like an iterator that can be incremented until the execution is finished.

For our example algorithm, we would define the following states: `none` at the beginning, `done` at the end, `rectangle_begin` when the sweep line hits the left edge of a rectangle, and `rectangle_end` when a right edge is reached. We implement the loop kernel by a member function `step()` that advances the sweep line to the next event point and returns the current state, see Program 4.

The knowledge about whether the sweep line is at the left or at the right edge of a rectangle, however, may not be sufficient for applications with animation or user interaction or simply for debugging client code. This is where the concept of *full logical inspectability* comes into play. An algorithm is fully logically inspectable if the user can access all relevant intermediate results during the execution. In this context relevant refers to results that define the state of the algorithm in a more mathematical sense than in the previous paragraph, namely to those intermediate results that would allow us to anticipate the algorithm's reaction to the next event.

In our example this would mean access to the content of the whole sweep-line status, not just to those rectangles that intersect the current rectangle, i.e. the one that corresponds to the current event point. Hence we offer two pairs of iterators, one referring to the whole sweep-line status, and one marking just the range of current intersections. Their type is discussed in Section 3.5. These iterators would make it possible, for instance, to display in blue all rectangles that currently intersect the sweep line and in red all those among them that intersect the current rectangle.

```
template < class Iterator,
           class LLXDA, class LLYDA, class URXDA, class URYDA >
class Rectangle_intersection
{
public:
  // constructor
  Rectangle_intersection( Iterator begin, Iterator end);
  // return the result in user supplied graph
  template <class Graph> void run( Graph& graph);
  // loop kernel
  enum  state   { none, rectangle_begin, rectangle_end, done };
  bool  valid() { return state != done; }
  state step();
  // full logical inspectability
  Iterator current(); // rectangle represented by curr. event point
  typedef  /* ... */  Solution_iterator;
  Solution_iterator begin();
  Solution_iterator end();
  // queries: report all rectangles intersecting the curr. rectangle
  Solution_iterator current_begin();
  Solution_iterator current_end();
};
```

Program 4: A class interface.

Note that the user can still get the output in a graph representation as before, but the existence of a graph data structure is no longer a prerequisite to using the algorithm. (This can also be achieved without templated member functions like `run()`, which are standard conform, but not yet supported by all C++ compilers.) Additionally the user has the possibility to construct his own intersection graph (e.g. if his data structure does not conform to our requirements), simply by accessing the current intersections via the member functions `current_begin` and `current_end`.

3.4 Decoupling Critical Decisions

Another important question is the following. Which definition of "intersection" do we implement? Does touching already imply an intersection? What about inclusion? Of course, the answers to these questions will differ from application to application. Instead of trying to cover all possible interpretations, we leave the definition of an intersection to the user. To do so, we must isolate the decision making parts of our implementation such that no information local to the algorithm is needed. Then the user can provide *function objects* with which the algorithm is parameterized.

Our rectangle intersection algorithm has two basic data structures, the event-point schedule and the sweep-line status. The order of event points decides, whether the projections of the corresponding rectangles intersect on the x-axis. Similarly, a query of the sweep-line status returns rectangles whose projections intersect that of the current rectangle on the y-axis. To differentiate between these two categories of intersections, we require the user to provide two function objects, one that enables us to sort the event-point schedule accordingly and the other for determining the behavior of the interval tree that implements the sweep-line status.

In the following we give examples of function objects that view rectangles as topologically open and thus do not report rectangles touching each other. To sort the event points accordingly, we just have to make sure that an event point e corresponding to the left edge of a rectangle will be inserted into the schedule after all event points that belong to right edges with the same x-coordinate. Then all of the latter rectangles are already removed from the sweep-line status and will not be reported when we reach e.

Since an event point is internal to our algorithm, it cannot be accessed directly by the user. Thus we have to isolate the information needed to sort the event points. If two event points have the same x-value, we need to know at least whether the first corresponds to a left or to a right edge of the respective rectangle. This information can easily be obtained via the x-coordinate of an event point plus the iterator pointing to the corresponding rectangle. Their types are of course known to the user. Thus we require a function object that realizes a comparison between two pairs of the corresponding types, see the example in Program 5. There `p` gets priority over `q` if the x-coordinate of `p` is strictly less than that of `q` or, in the case of equality, if `p` corresponds to the right edge of the respective rectangle. This *external* function object is then used by our implementation to *internally* sort the event points. The data structure `pair<T1,T2>` that we use in Program 5 is defined in the STL; it is a container that stores two objects, the first of type `T1`, the second of type `T2`.

```
// client code
class Compare_x
{
public:
  typedef pair<CoordType, User_iterator> Pair;
  bool operator()( const Pair& p, const Pair& q) {
    return (p.first < q.first) ||
           ( (p.first == q.first) && (p.first == p.second->urx) );
  }
};
```

Program 5: Example of a compare function object for sorting the event-point schedule.

The second function object that is used for querying the interval tree is quite simple, see the example in Program 6. Note the difference of this technique to the approach of implementing the most general definition of intersection and then filtering out all undesired information. Our function objects have the

```
// client code
class Compare_y {
public:
  bool operator()( CoordType const y1, CoordType const y2)
  { return y1 < y2; }
};
```

Program 6: Example of a compare function object for querying the sweep-line status.

potential to reduce the complexity not just of the *output*, but also of the *computation*.

3.5 The Complete Interface

Program 7 shows the interface resulting from our successive improvements. At first sight it might look more complicated than the naive interface. To guarantee a smooth learning curve for users not familiar with generic programming and the algorithm we implemented, a good library would provide a concrete representation of rectangles and graphs, say `Our_rectangle` and `Our_graph`, as well as defaults for the other template parameters. This would make it possible to use our algorithm as in Program 8.

Note that the constructor has been parameterized by objects of each of the class's template parameters. This allows the user to instantiate our algorithm with objects that may be constructed other than by their default constructor. The data accessors `URXDA` and `URYDA` for the coordinates of the upper right corner of the input rectangles might for example be parameterized with a given height and width of the rectangles, which could change from instantiation to instantiation of the intersection algorithm.

Of course, we have also implemented the interval tree for the sweep-line status with the concepts presented here. This is a typical point where the flexibility of our algorithms bears fruit, since we do not have to convert any rectangles into intervals to construct the interval tree. This is due to the fact that the endpoints of the intervals we want to store in the tree correspond to the y-coordinates of the corners of our input rectangles. Thus we just parameterize the nested interval tree class with a subset of the template parameters of the class `Rectangle_intersection`. The required parameters are the types of the rectangle iterator and the compare function object for y-coordinates as well as the data accessors `LLYDA` and `URYDA`, see the private definition of the type `Interval_tree` in Program 7. So in a way, the class `Interval_tree` considers our input rectangles to be nothing but intervals, namely the rectangles' projection on the y-axis.

The iterator `Solution_iterator` needed to traverse the sweep-line status is provided by the class `Interval_tree`. Dereferencing a `Solution_iterator` supplies the user with an iterator of the type with which he has parameterized the class `Rectangle_intersection`.

Program 9 shows how the data structures required by our algorithm could be declared. Our example demonstrates one of the advantages of using data accessors. If the input consists of squares of common size, the user has to store only the coordinates of their lower left corners. When our algorithm needs a coordinate of the opposite corner, the corresponding data accessing function computes it on the fly. This reduces the storage consumption of the input by 50% at the cost of a negligible increase in the running time.

For the selection of a class member C++ offers a so-called pointer-to-member. Pointer-to-members can even be used as template arguments which makes it possible to implement a generic class for data accessors that differ only in the members being accessed. This technique is used in Program 9 for the data accessors `CoordLowDA` and `CoordHighDA`. Note that not all compilers support this pointer-to-member mechanism for template parameters. The obvious workaround is to declare explicitly all four data accessors and accessing functions as in Program 2.

With the data accessors of Program 9 the intersection algorithm for squares can be declared as in Program 10. Program 11 shows how the types declared in Programs 9 and 10 are plugged into our interface.

```
class Our_rectangle; class Our_graph;
class Our_lxda; class Our_lyda; class Our_hxda; class Our_hyda;
class Our_compare_x; class Our_compare_y;

template < class Iterator = Our_rectangle*,
           class LLXDA = Our_lxda, class LLYDA = Our_lyda,
           class URXDA = Our_hxda, class URYDA = Our_hyda,
           class CompareX = Our_compare_x,
           class CompareY = Our_compare_y >
class Rectangle_intersection
{
  // type of the sweep-line status
  typedef Interval_tree<Iterator, LLYDA, URYDA, CompareY>
    Sweep_line_status;
public:
  // type of rectangle coordinates
  typedef typename LLXDA::value_type value_type;
  // type of iterator used to return intersections
  typedef typename Sweep_line_status::iterator Solution_iterator;
  // constructor
  Rectangle_intersection( Iterator begin, Iterator end,
                          LLXDA lxda = LLXDA(), LLYDA lyda = LLYDA(),
                          URXDA hxda = URXDA(), URYDA hyda = URYDA(),
                          CompareX comp_x = CompareX(),
                          CompareY comp_y = CompareY() );
  // loop kernel
  enum  state { none, rectangle_begin, rectangle_end, done };
  bool  valid() const { return state != done; };
  state step();
  // full logical inspectability
  Iterator current(); // rectangle represented by current event point
  value_type current_sweep_line_position() const;
  Solution_iterator begin();
  Solution_iterator end();
  // queries: report all rectangles intersecting the curr. rectangle
  Solution_iterator current_begin();
  Solution_iterator current_end();
  // miscellaneous member functions
  int number_of_intersections() const;
  // return intersection graph of input rectangles
  template <class Graph> void run( Graph& graph);
};
```

Program 7: A flexible interface of the class `Rectangle_intersection`.

```
// client code
Our_rectangle rects[10];
Our_graph our_graph;
// get or compute the rectangle data,
// then declare and run the rectangle intersection algorithm
Rectangle_intersection rectangle_intersection( rects, rects+10);
rectangle_intersection.run( our_graph);
```

Program 8: Ease-of-use: applying `Rectangle_intersection` to library-supplied data structures.

```
// client code
// - representation of a square
typedef int CoordType;
const  CoordType length = 50;
struct Square { CoordType x, y; };

// - data accessors for the above representation
template<CoordType Square::*member>
struct CoordLowDA  { typedef CoordType value_type; };
template<CoordType Square::*member>
struct CoordHighDA { typedef CoordType value_type; };

// - data accessing functions
template <CoordType Square::*member>
inline CoordType get( CoordLowDA<member>  const&, User_iterator const& it)
{ return (*it).*member; };

template <CoordType Square::*member>
inline CoordType get( CoordHighDA<member> const&, User_iterator const& it)
{ return (*it).*member + length; };

// - compare function objects as defined above
class Compare_x;  class Compare_y;
```

Program 9: Flexibility: user-supplied types for using the class `Rectangle_intersection`.

```
// client code
// - algorithm declaration
typedef Rectangle_intersection< User_iterator,
                                CoordLowDA<&(Square::x)>,
                                CoordLowDA<&(Square::y)>,
                                CoordHighDA<&(Square::x)>,
                                CoordHighDA<&(Square::y)>,
                                Compare_x, Compare_y >
        Square_intersection_algo;

// - iterator for access to the solution
typedef typename Square_intersection_algo::Solution_iterator Solution_it;
```

Program 10: Declaration of the class `Rectangle_intersection` for squares of common size.

```
// client code
main()
{
  Square squares[10];  // input of squares...
  Square_intersection_algo my_algo( squares, squares+10);
  while (my_algo.valid())
    if (my_algo.step() == Square_intersection_algo::rectangle_begin) {
      Square* curr = my_algo.current();
      // assuming output operator for Square
      cout << *curr << " intersects: " << endl;
      Solution_it end = my_algo.current_end();
      for (Solution_it it = my_algo.current_begin(); it != end; ++it)
        cout << *it << endl;
    }
}
```

Program 11: A toy application for user-supplied data structures.

4 Experiments

We compare two different implementations of the rectangle intersection problem in terms of running time. The implementations are characterized as follows.

1. The *naive approach* implements the interface of Section 3.1. It requires a fixed type of rectangle. Similarly, the underlying interval-tree data structures requires a fixed type of interval. The results of the sweep are stored in a graph (of fixed type) that is made available to the user at the end of the sweep.

2. The *generic approach* implements the interface of Section 3.5. All data structures are implemented according to the concepts suggested in this paper.

The main difference between the two implementations is that all template parameters of the generic interface are hard-coded in the naive interface (except for coordinate type of the input rectangles). We were interested in comparing the cost of the additional indirection that our generic implementation requires versus the cost of copying data from and into containers of fixed type as required by the naive approach.

4.1 Sample Classes

We ran both implementations on four sample classes. In Figures 1 to 4 we depicted an instance of each sample class. The sample generators and the real world data are all available from our Web page. Similar benchmarks have also been used to compare the quality of map-labeling algorithms experimentally [WWKS01]. In each case we first pick a set of points and then place axis-parallel rectangles of fixed or varying size such that their lower left corners coincide with the corresponding points.

The graphs in Appendix A (Figures 5 to 8) show an important parameter of these sample classes, namely their average number of intersections. As in the example implementation of the compare function objects in Section 3.4 we did not count pairs of *touching* rectangles. For all but one sample class (more below) we used samples of 1000, 2000, ... up to 10,000 rectangles. We will refer to the number of rectangles in a sample as the *sample size*. For each sample class (but one, see below) and each sample size (shown on the x-axis), we generated 30 instances and averaged the number of intersections (see y-axis) over these instances.

RandomRect. We choose n points uniformly distributed in a square of size $25n \times 25n$. Each point corresponds to the lower left corner of a rectangle. To determine the size of each rectangle, we choose the length of both edges independently under normal distribution, take its absolute value and add 1 to avoid non-positive values. Finally we multiply both rectangle dimensions by 10, see Figure 1. This procedure yields sparse instances with a linear number of rectangle intersections, see Figure 5.

VariableDensity. This sample class was suggested in an experimental comparison of map-labeling algorithms by Christensen et al. [CMS95]. There, the points are distributed uniformly on a rectangle of size 792×612. All rectangles are of equal size, namely 30×7, see Figure 2. The asymptotic number of intersections is quadratic here, see Figure 6.

FencyRect. In order to also investigate very dense samples we constructed fence-like instances with $\lfloor n/2 \rfloor$ horizontal and $\lceil (n+1)/2 \rceil$ vertical rectangles of dimensions $(n+1) \times 1$ and $1 \times (n+1)$, respectively, such that each horizontal rectangle intersects each vertical, see Figure 3. This gives rise to very dense instances with $O(n^2)$ pairs of intersecting rectangles, see Figure 7. Due to the huge amount of memory that is required to store the intersection graphs of such examples, we restricted the sample sizes to $200, 400, \ldots, 2000$ rectangles. Since the construction method for FencyRect is deterministic there was no need to do the experiment more than once for each sample size.

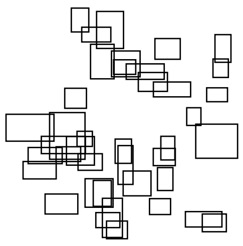


Figure 1: RandomRect (40 rectangles)

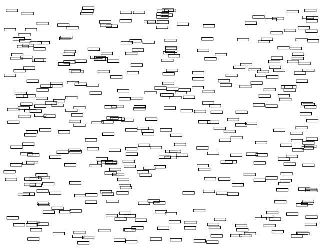


Figure 2: VariableDensity (400 rectangles)

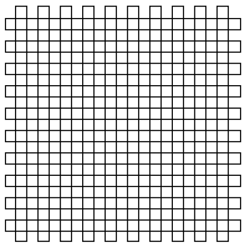


Figure 3: FencyRect (20 rectangles)

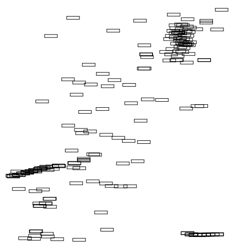


Figure 4: DrillHoles (180 rectangles)

DrillHoles. The municipal authorities of Munich provided us with the coordinates of roughly 19,400 ground-water drill holes within a rectangle given by 442,935,800 $\leq x \leq$ 448,588,183 and 532,525,217 $\leq y \leq$ 536,579,000 (in centimeters) that is centered approximately on the city center. From these sites, we randomly pick a site s and then extract the n sites closest to s according to the maximum norm. Thus we get a rectangular section of the complete instance, see Figure 4. The rectangles we attach to the points are labels of fixed dimensions (30,000 $\times$ 7,000) that contain information about the the drill holes. The number of intersections varies considerably in this sample class. The average number increases slightly superlinearly, see Figure 8.

4.2 Results

The x-axis of the graphs in Appendices B and C gives the sample size as in Appendix A. On the y-axis the graphs in Appendices B and C give running times in CPU seconds averaged over the 30 instances per sample size. The vertical bars mark the minimum and maximum running time on these instances. As stated above there is only one instance per sample size for FencyRect. The average running times of the naive implementation have small circular markers linked by thin lines, those of the generic implementation have square markers linked by bold lines. All running times were measured on an i686 PC with 128 MB RAM under the Linux-2.2.16 operating system. We used the g++-2.95.2 compiler with optimizer option `O3`. We analyze the following two settings.

In Appendix B Figures 9 to 12 show the running times of both the naive and the generic implementation for the interesting special case that the user is merely interested in the number of intersections. This is favorable for the generic implementation since its user is not forced to store the intersection graph. In this setting, the running times were nearly identical for two compilers (SUN CC-4.2 on a Sparc Ultra-1 and mipsPRO CC-7.1 on an SGI IP27) we used in the first version of this article two and a half years ago [KKW00]. The picture has changed with the progress in compiler technology. Now the naive implementation is slower by a factor of 1.2 to 3.1 on all examples of maximum size (i.e. 10,000 or 2,000 rectangles). Roughly speaking, the denser an example class the greater the gap between the running times of the two implementations.

In Appendix C, Figures 13 to 16 show the running times for the case that all intersections are stored in adjacency lists during the execution of the two programs. While the gaps are smaller than in the corresponding graphs of Appendix B, and the naive implementation is even slightly faster for FencyRect samples with up to 1200 rectangles, the general picture remains the same. The naive implementation is slower by a factor of 1.2 to 1.3 on all examples of maximum size. In our previous test bed [KKW00], the generic implementation was nearly always slower in this setting—up to a factor of 1.6.

In Table 1 we listed the sizes of executables for identical test programs for both implementations as well as the sizes of the source code of the naive and the generic interface including their implementation. The source code is available via our Web page. It is interesting to note that although the executable of a simple test program for the generic version of the interval tree is larger than that of its naive counterpart (when using no optimizer), it is opposite for the corresponding versions of the rectangle-intersection data structure. The reason for this seems to be that the generic implementation does not need to convert and store the input data for the interval tree explicitly. This difference in the sizes of the executables may become substantial

	size of executable			*size of source code*	
test program for . . .	*with optimizer* `O3`	*without optimizer*	*static linkage*	*interface & implementation*	
naive interval tree	53	107	1556	9.3	350 lines
generic interval tree	52	159	1606	10.1	389 lines
naive rectangle intersection	72	344	1792	6.0	193 lines
generic rectangle intersection	67	259	1707	7.7	286 lines

Table 1: Sizes of executables and source code (in kilobytes and lines of code).

in case of larger hierarchies of algorithms that are built on top of each other.

Acknowledgments

Thanks to Karsten Weihe and Sven Schönherr for fruitful discussions. Thanks to Rudi Krämer, Amt für Informations- und Datenverarbeitung, München, for the drill-hole data. Finally we would like to thank the two anonymous referees for many helpful comments.

Conclusion

We have presented a toolbox of concepts which helps to turn inflexible into generic and thus reusable interfaces. We have studied this transition through a geometric algorithm, namely a sweep-line algorithm for the rectangle-intersection problem. On the road from a naive to a flexible interface for this algorithm, we suggested to decouple algorithms from the organization of their input and output data. Then we presented the loop kernel as an important means of gaining control over the execution of an algorithm. Full logical inspectability introduced additional transparence. Finally, we came up with function objects as a way to parameterize algorithms with information that can be used to influence critical decisions.

In our experiments, we compared a generic to a naive implementation of the rectangle-intersection algorithm. We investigated the running times of the two implementations on four sample classes from random and real-world sources in two different settings. In the first setting, which was favorable for our generic implementation, the naive implementation was slower by a factor of 1.2 to 3.1. In the second setting the naive implementation was faster on small instances from one of the four sample classes, but on large instances it was still slower by a factor of 1.2 to 1.3 in all sample classes.

References

[CMS95] Jon Christensen, Joe Marks, and Stuart Shieber. An empirical study of algorithms for point-feature label placement. *ACM Transactions on Graphics*, 14(3):203–232, 1995.

[EM81] Herbert Edelsbrunner and Hermann A. Maurer. On the intersection of orthogonal objects. *Information Processing Letters*, 13(4–5):177–181, End 1981.

[FGK$^+$00] Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. On the design of CGAL, a computational geometry algorithms library. *Software—Practice and Experience*, 30(11):1167–1202, September 2000.

[Ket99] Lutz Kettner. Using generic programming for designing a data structure for polyhedral surfaces. *Computational Geometry: Theory and Applications*, 13(1):65–90, May 1999.

[KKW00] Vikas Kapoor, Dietmar Kühl, and Alexander Wolff. A generic design concept for geometric algorithms. Technical Report B 00–10, Institut für Informatik, Fachbereich Mathematik und Informatik, Freie Universität Berlin, June 2000.

[Küh96] Dietmar Kühl. Design patterns for the implementation of graph algorithms. Master's thesis, Technische Universität Berlin, 1996.

[KW97] Dietmar Kühl and Karsten Weihe. Data access templates. *C++ Report*, 9(7):18–21, July 1997.

[MN89] Kurt Mehlhorn and Stefan Näher. LEDA: A library of efficient data types and algorithms. In Antoni Kreczmar and Grazyna Mirkowska, editors, *Mathematical Foundations of Computer Science 1989*, volume 379 of *Lecture Notes in Computer Science*, pages 88–106, Porabka-Kozubnik, Poland, 28 August–1 September 1989. Springer-Verlag.

[MN99] Kurt Mehlhorn and Stefan Näher. *LEDA: A Platform of Combinatorial and Geometric Computing*. Cambridge University Press, Cambridge, England, January 1999.

[MS96] David R. Musser and Atul Saini. *STL Tutorial and Reference Guide*. Addison-Wesley, Reading, MA, 1996.

[NW96] Marco Nissen and Karsten Weihe. Combining LEDA with customizable implementations of graph algorithms. Technical Report 17, Fakultät für Mathematik und Informatik, Universität Konstanz, October 1996. ISSN 1430-3558.

[Ove96] Mark H. Overmars. Designing the Computational Geometry Algorithms Library CGAL. In M. C. Lin and D. Manocha, editors, *Applied Computational Geometry (Proc. WACG '96)*, volume 1148 of *Lecture Notes Comput. Sci.*, pages 53–58. Springer-Verlag, 1996.

[SLL99] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. The generic graph component library. In *Proc. 1999 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 399–414, Denver, CO, 1–5 November 1999.

[SLL00] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. The generic graph component library. *Dr. Dobb's Journal*, September 2000.

[Wei97] Karsten Weihe. Reuse of algorithms: Still a challenge to object-oriented programming. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA-97)*, volume 32, 10 of *ACM SIGPLAN Notices*, pages 34–48, New York, October 1997. ACM Press.

[Wei98] Karsten Weihe. Using templates to improve C++ designs. *C++ Report*, 10(2):14–21, 1998.

[WWKS01] Frank Wagner, Alexander Wolff, Vikas Kapoor, and Tycho Strijk. Three rules suffice for good label placement. *Algorithmica Special Issue on GIS*, 30:334–349, 2001.

Appendix A: Numbers of Pairs of Intersecting Rectangles

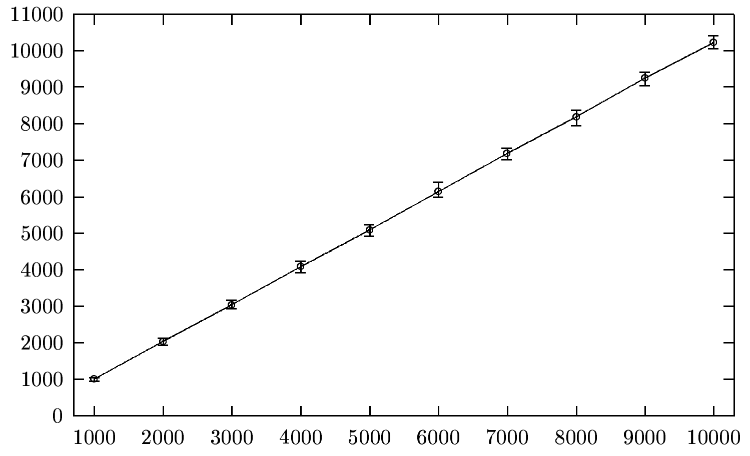


Figure 5: RandomRect

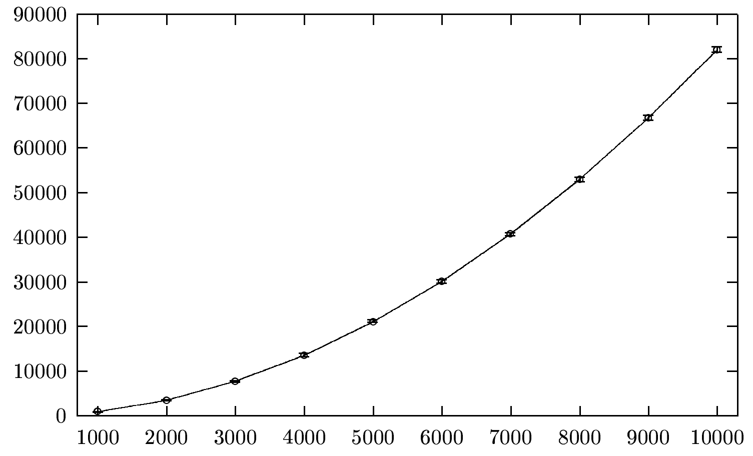


Figure 6: VariableDensity

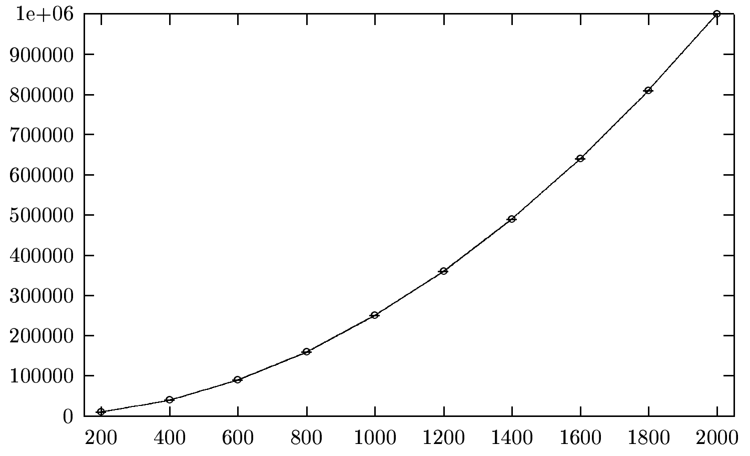


Figure 7: FencyRect

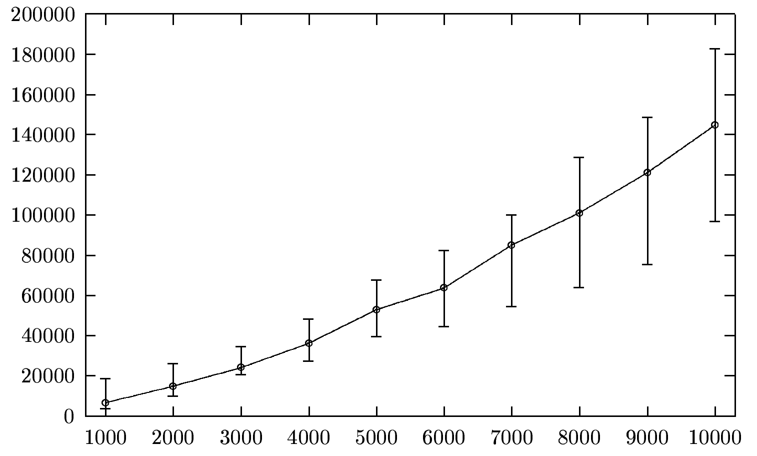
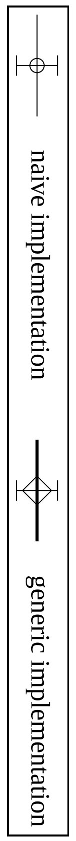


Figure 8: DrillHoles

Appendix B: Running Times for Computing the Number of Intersections



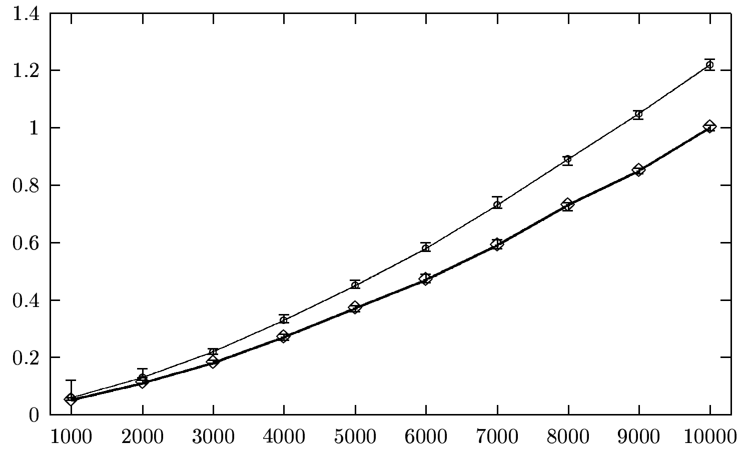


Figure 9: RandomRect

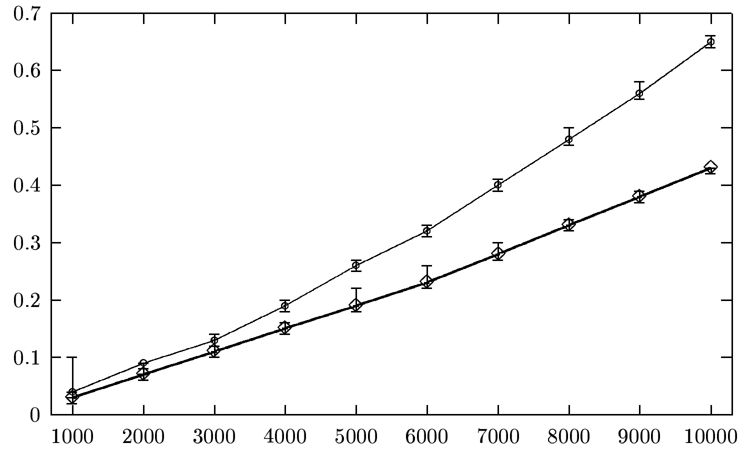


Figure 10: VariableDensity

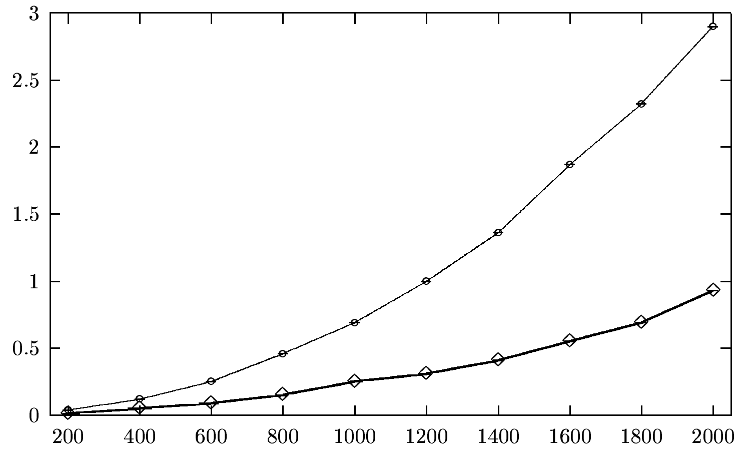


Figure 11: FencyRect

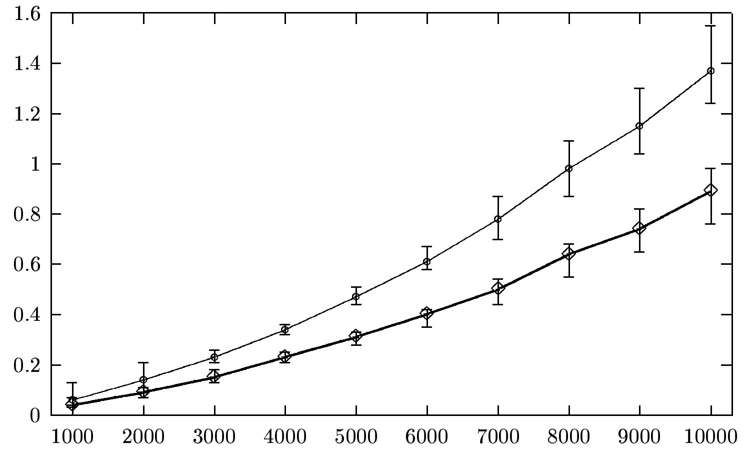


Figure 12: DrillHoles

Appendix C: Running Times for Generating Adjacency Lists

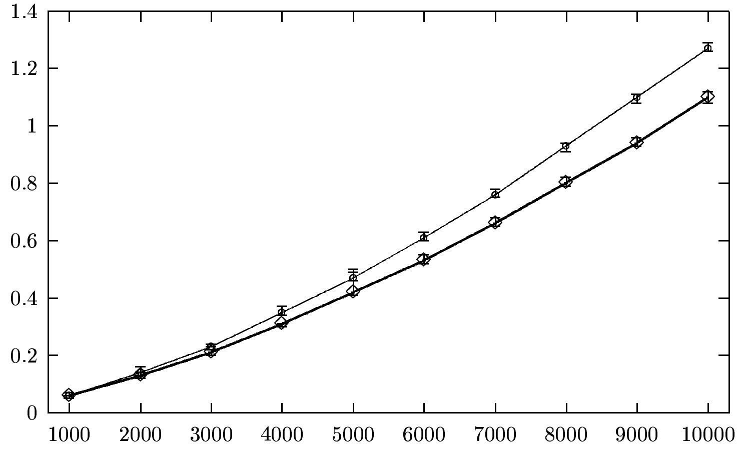


Figure 13: RandomRect

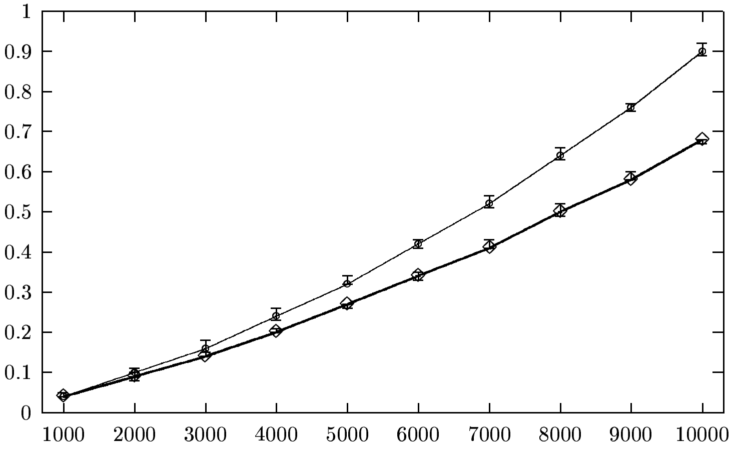
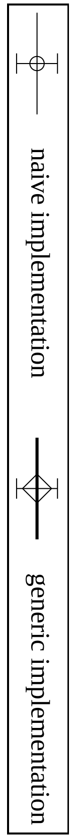


Figure 14: VariableDensity



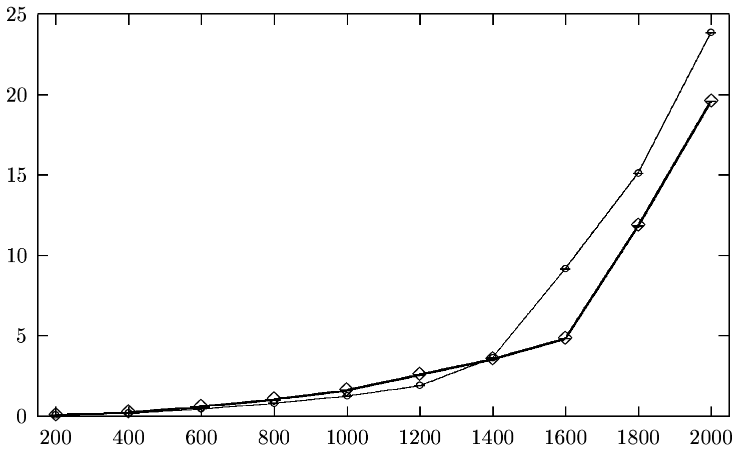


Figure 15: FencyRect

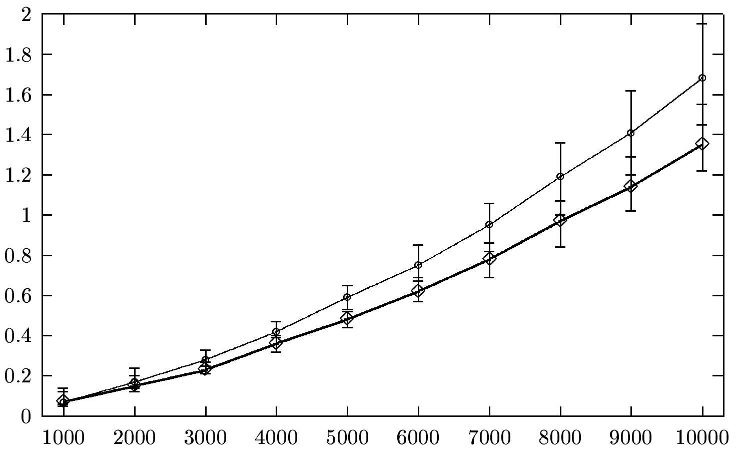


Figure 16: DrillHoles