Contents

Chapter 1

Introduction to the use of LEDA in computational geometry

1.1 General introduction

If you peruse the many outlets for algorithmic research — journals, symposia proceedings, textbooks — it is difficult not to be impressed by the wealth of useful algorithms there. The mood can quickly change, though, if you set out to implement one of these algorithms, especially if it belongs to the domain of computational geometry (CG). One reason is the prominent divide between the abstract CG algorithms found in academic papers and the programs that must be implemented to realise them. More precisely, the "paper algorithms" often rely on assumptions that are untenable in practice (such as the existence of infinite precision arithmetic) and on data structures that are either exotic, requiring extensive ground-up development, or overly complex for practical use. Recent years have seen a number of initiatives to address this imbalance, including papers[1], at least one journal[2], and a handful of large, publicly available software libraries that have already found widespread use [4][5]. LEDA is one such library and the remainder of this chapter gives an short introduction to it, including a brief look at how it can help develop computational geometry applications.

1.2 What is LEDA ?

1.2.1 History and motivation

LEDA development began in the fall of 1988 at the Max Planck Institut für Informatik in Saarbrücken, Germany and was originated by Kurt Mehlhorn and Stefan Näher. Their original motivation was a realisation that "*algorithms research must include implementation if the field wants to have maximum impact.*" [3, p.xii - emphasis orig.] and that there was a need for a repository of robust components for use in algorithm implementation. The latter was apparent from the number of researchers and students who "reinvented the wheel" by implementing the same generic data structures and auxiliary functions as their colleagues, thus wasting time and effort that could have been better expended on advancing the state of the art. As a result, LEDA features most of the basic building blocks of computer science, including data structures such as heaps, queues and graphs and algorithms for sorting and searching. During its years of development, LEDA has gone further, though, and now includes far more sophisticated structures, many of which are templatized, i.e. based on C++ templates which makes them more generic and potentially useful than they would otherwise be.

At the time of writing, LEDA version 4.0 is freely available from [5] for non-commercial purposes and has been ported to the most popular platforms (UNIX/Linux, MS-Windows, Solaris, etc.).

1.2.2 Related products

LEDA is not the only library to come from the efforts to bridge the gap between theory and practice. The CGAL[3] library is a publicly available product developed and maintained by a consortium of academic institutions, six

[1] compgeom.cs.uiuc.edu/~jeffe/compgeom/taskforce.html
[2] www.jea.acm.org
[3] abbr. for Computational Geometry Algorithms Library.

European and one Israeli. It is similar to LEDA in its goals and intentions, but is more narrowly focused on computational geometry and therefore less comprehensive. It does, however, have strong support for exact arithmetic and other means of dealing with the inherent problems of computational geometry algorithms which may make it better suited for some applications. CGAL source code and documentation is available from [4].

While LEDA and CGAL are probably the two most comprehensive non-commercial C++ libraries available, there are also a number of smaller software packages, including some for use with the Java language. An annotated list can be found at [1].

1.3 Why use LEDA ?

There are some obvious advantages to using LEDA. For one, you benefit from a large selection of free, ready-to-use data structures and algorithms that are both reliable and efficient. That last point is documented by the statistics in the table below which are taken from the recently published book on LEDA[3]. They show the execution times for various simple operations on a representative data structure that both LEDA and the ANSI C++ Standard Template Library (STL) implement, namely a list (template) class, and as you can see, LEDA fares quite well in terms of efficiency. There are, of course, far more complex components in the LEDA toolbox than the relatively simple list class, but wherever possible, the team behind it have chosen the algorithm or structure with the best known or optimal time complexity, giving good overall efficiency compared to similar language extensions like the STL.

There are, however, also a few drawbacks. Firstly, the size of the executable may increase from using LEDA, both because of the heavy reliance on C++ templates which are known to cause "code bloat" if multiple instantiations are required and because of the general overhead incurred by using a large scale general library. Secondly, there may be an efficiency concern although the statistics seem to suggest otherwise. This can become manifest in certain graphics applications which may require the absolute optimum and therefore cannot make do with what LEDA has to offer. All in all, though, the scales seem to turn in favour of using LEDA for one's algorithm implementations whenever possible.

LIST<int>	LEDA	STL
build list	0.020 sec.	0.040 sec.
pop and push	0.030 sec.	0.030 sec.
reversing	0.020 sec.	0.030 sec.
copy constr	0.050 sec.	0.050 sec.
assignment	0.020 sec.	0.040 sec.
clearing	0.000 sec.	0.020 sec.
sorting	0.130 sec.	0.400 sec.
sorting again	0.140 sec.	0.330 sec.
merging	0.030 sec.	0.080 sec.
unique	0.080 sec.	0.080 sec.
unique again	0.000 sec.	0.010 sec.
iteration	0.000 sec.	0.000 sec.
total	*0.520 sec.*	*1.110 sec.*

LIST<class>	LEDA	STL
build list	0.090 sec.	0.030 sec.
pop and push	0.100 sec.	0.030 sec.
reversing	0.070 sec.	0.030 sec.
copy constr	0.140 sec.	0.060 sec.
assignment	0.120 sec.	0.030 sec.
clearing	0.080 sec.	0.020 sec.
sorting	0.770 sec.	0.510 sec.
sorting again	0.900 sec.	0.380 sec.
merging	0.200 sec.	0.090 sec.
unique	0.250 sec.	0.100 sec.
unique again	0.010 sec.	0.000 sec.
iteration	0.010 sec.	0.000 sec.
total	*2.740 sec.*	*1.280 sec.*

Table 1.1: LEDA vs. STL on a list of `ints` (left) and objects (right)

1.4 How to use LEDA in computational geometry

1.4.1 Preliminaries

LEDA is a big library and to make some sense of all its features, we need to focus on the components most relevant to CG and more or less disregard the rest. Before we can do so, however, a brief look at some LEDA basics is in order.

Items and iterators

Two fundamentals in LEDA are *items* and *iterators*. The latter concept may be familiar to you since it is occasionally used in computer science textbooks and because it is a key ingredient in the STL library. An iterator can be seen as a sophisticated pointer or, more suggestively, as a kind of "slide rule" and is used to iterate through, access and modify sequences. The former concept, *items*, is the LEDA term for a particular type of access mechanism that roughly corresponds to C++ pointers. Put crudely, an *item* is a form of syntactic sugar that allows you to declare and use an object in the way you would expect when in reality you are working with a reference to that object. Consider as an example the below piece of code:

```
point p(20.0,30.0), q(10.0,40.0); // declare object of LEDA class point
p.distance(q); // returns the Euclidean distance between p and q
```

Here, variables `p` and `q` appear to be perfectly normal objects and can indeed be used as such. In effect, however, they are more like pointers since they do not themselves store the coordinates given as arguments to the constructors. Instead, they contain references to underlying container objects (of class `point_container` in this case) that hold the coordinate data and provide functions such as `distance`. There are good reasons for adding this intermediary layer between user and library: efficiency (copying pointers is faster than copying large objects), customized memory management (LEDA could do automatic garbage collection without inconveniencing the user) and extensibility (the *item* structures can be continously refined in future releases).

The `point` is an *independent item*, meaning that it behaves much like a conventional C++ object. The other type of item is, not surprisingly, called *dependent* because it can only exist within a larger framework, i.e. it needs a parent. One example of dependent item can be found in use of the `sortseq<K,I>` structure which associates a set of keys of type K with a set of items of type I. The below piece of code illustrates dependent items and also demonstrates use of *iteration macros* which are another form of syntactic sugar that can help sweeten the life of a programmer:

```
sortseq<string, string> s;
s.insert("Goddag", "Buenas dias");
s.insert("Hund", "Perro");
s.insert("Taske", "Bolso");
sortseq<string, string>::item   s_item;
forall_items(s_item, s)
{ std::cout << s.key(s_item) << " " << s.inf(s_item) << std::endl; }
```

The above code first declares a `sortseq` data structure and then inserts 3 key/value pairs into it. The next step is to iterate through the newly constructed sequence and print its contents to standard output. To that end, the `forall_items` iteration macro is used. It is nothing more than a compact form of a normal for-loop and has the semantics you would expect: it assigns the elements in `s` to `s_item` one at a time, beginning with the first and ending with the last and after each such assignment, it executes the code in the block immediately below the macro. There are several such macros available in LEDA and although you are not required to use them (they can always be replaced by equivalent for-loops), they do seem to improve readability. But what about the promised illustration of dependent items ? Well, just before the iteration macro, the item type member from the `sortseq` template is used to declare the variable `s_item` which is then used in the iteration macro for repeated assignment of the items from `s`. That variable is a dependent item because it can only be used if the container it is associated with, i.e. `s`, is "alive" (hasn't been deallocated) and is in that respect more like a regular pointer than independent items such as `point` and the like. Note also that accessing the "contents" of `s_item` is not done by member access but rather by passing the item to its parent structure using member functions `key()` and `inf()` (LEDA uses the term "information" for the value component of a key/value pair).

1.4.2 Using LEDA

It is time to look at how LEDA can help realise computational geometry algorithms and since such algorithms are often inherently visual, what better place to start than the `window` class which is LEDA's basic means of graphical input and output. Using `window` objects is not difficult and can be useful both during development (by aiding understanding and debugging) and afterwards when the program is put to use. An example will illustrate this:

Creating windows

```
#include <LEDA/window.h>

void main()
{
window w(600,400,"Example"); // creates 600x400 pixel window with caption "Example"
w.init(0,300,0);                // configure the coordinate system
w.display(0,0);         // open window at 0,0
w.draw_segment(10,10,10,100);   // 10,10 and 10,100 are endpoint coordinates
w.draw_circle(50,50,10); // draws circle with radius 10 centered at 100,100
w.draw_filled_rectangle( point(100,50), point(150,180), green); // draws green rectangle
w.message("Left click to exit");
```

```
w.read_mouse();
}
```

The above code first creates a 600x400 window with the label "Example" appearing in its title bar. It then goes on to call the `init()` member function which sets up the coordinate system. This may seem odd; after all, the window was passed its maximum x and y coordinates when constructed so why this need for additional coordinate information ? The reason is that LEDA makes a distinction between the *pixel coordinate system* and the coordinate system you specify, the *user coordinate system*. In the former system, x-coordinates increase from left to right, y-coordinates from top to bottom and the upper left corner has coordinates (0,0), but in the latter, y-coordinates increase from bottom to top and it is the lower left corner which has coordinates (0,0). So the pixel coordinate system is the conventional means of addressing a bitmap, whereas the user coordinate system is more like the Cartesian layout we know from mathematics, and it is this latter system which is configured by the call to `init`. The three arguments in the call are the minimum and maximum x-coordinates and the minimum y-coordinate. The maximum y-coordinate is not required as it is computed from the physical dimensions of the window and the x-coordinate range of the user coordinate system.

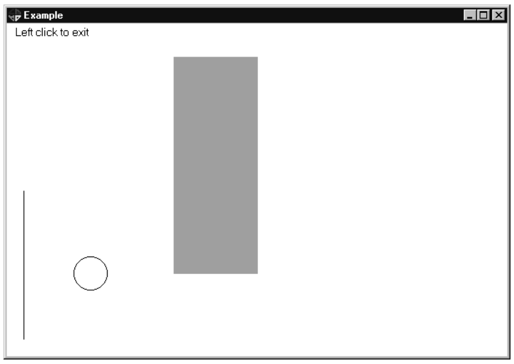


Figure 1.1: Output from the `window` code example

After calling `display()` to open the window, the program goes on to call some of the drawing primitives in the `window` class. Specifically, it draws a line segment, a circle and filled rectangle by calling appropriate functions. The interface and functionality should be self-explanatory and is well documented in the manual, but note the use of the `point` class in the call to `draw_filled_rectangle`. This hints at the techniques you will need to adopt if you want to draw more advanced geometric structures in a convenient way — more on that later.

Adding a panel

The last two function calls in the previous example, to `message()` and `read_mouse()`, respectively, show some of the non-graphical `window` functionality that you have at your disposal. In addition to displaying messages and reading mouse input, you can customize most features of the user interface and, just as importantly, you can add a full-fledged menu system with buttons and other controls that connect to functions you define. An example:

```
#include <LEDA/window.h>
#include <iostream>

static int old_value = 0;

void f(int new_value)
{
std::cout << "Old value: " << old_value << std::endl << "New value: " << new_value << std::endl;
}
void main()
{
        window wp(800,600,"Window with panel");
        wp.init(0,400,0);
        wp.int_item("A slider control", old_value, 0, 500, f);
        wp.button("Click me", 0);
        wp.enable_panel();
        wp.display();
        wp.read_mouse();
}
```

Here, a window has been equipped with two controls, a push button and a so called slider control which is used for graphical adjustment of an integer variable. There are several more controls that can be added to a panel but these two are indicative of the rest and demonstrate the most important concepts. For example, the slider control is added by a call to the `int_item` member function which takes a function pointer as its last argument. Such *event handling* can be done for all controls, not just sliders and it works by triggering a call to the specified function (`f` in the above case) each time the value of the associated control changes. The integer argument passed in that function call is the new value of the control; the old value is available from the variable that is associated with the control when it is set up (`old_value` in the above example).

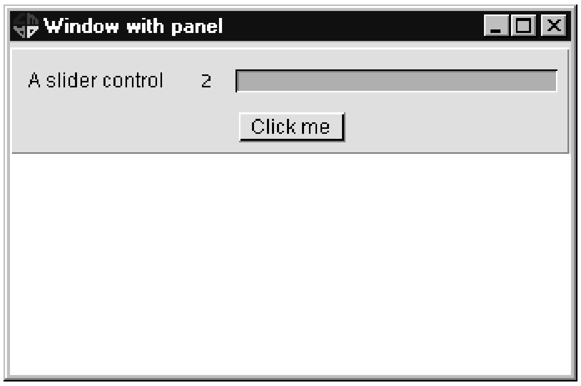


Figure 1.2: `window` object with panel

Beyond the basics

It is quite easy to go from simple drawing operations like the above to considerably more sophisticated uses. A case in point (pardon the pun) is the `point_set` which supports several high-level CG operations such as nearest neighbour queries and range search by maintaining a Delaunay triangulation of the point set it contains. Applying the operations is not difficult nor is drawing their output, as the below example demonstrates:

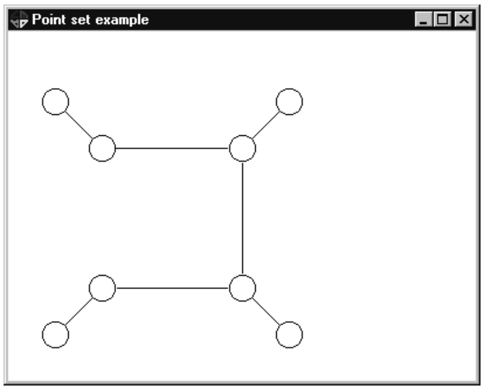
```
#include <LEDA/point_set.h>
#include <LEDA/list.h>
#include <LEDA/point.h>
#include <LEDA/window.h>
#include <iostream>

// global data is usually a no-no, but we'll make an exception for the sake of compactness
window wpt(400,300,"Point set example");

void draw_edge(const point& u, const point& v)
{       wpt.draw_edge(u,v);     }

void draw_node(const point& u)
{       wpt.draw_node(u);       }

void main()
{
        point input_p;
        list<point> list_points;
        do {

std::cout << "Specify a point by entering two floating point values (-1 -1 to end): ";
        std::cin >> input_p;
        } while(input_p != point(-1,-1) && list_points.push_back(input_p));
        point_set p_set(list_points);
        list<edge> msp = p_set.minimum_spanning_tree();
        wpt.display();
        p_set.draw_edges(msp, draw_edge);
        p_set.draw_nodes(draw_node);
        wpt.read_mouse();
}
```

A `point_set` can be created using a `list`[4] of `points`, so the above program begins by building one from user input which is then used to construct a `point_set`. Such a structure has, among other things, a function for

[4]LEDA's "plain-vanilla" version of the well-known doubly-linked list data structure.

computing the minimum spanning tree from its underlying Delaunay triangulation of the points the user has entered.

That function is called and returns a `list` of `edge` objects (which are essentially pairs of endpoints[5]). The program then proceeds to draw the edges and their endpoints using two function calls to the `point_set` object. How can a data structure render graphics, you might ask. It can't. The drawing part is taken care of by functions `draw_edge` and `draw_node` which are called through the pointers passed to `p_set`, once for each node and edge in the point set and MST, respectively. The end result is shown in in the adjacent figure and hopefully illustrates how you can use LEDA for even fairly advanced operations without extensive knowledge of C++.



Graph windows

The `point_set` example hinted at the need for graph drawing capabilities. After all, a minimum spanning tree is a particular type of graph and the field of application for graphs in general is huge, so it would be reasonable to expect `window` to have good facilities for graph rendering. Actually, LEDA does even better by providing a specialization of `window` called `GraphWin` with functions that are tailor-made for graph drawing (in addition, of course, to all the functionality it inherits from `window`). Not surprisingly, this makes `GraphWin` better suited than `window` for some types of display operation as the below example demonstrates:

```
#include <LEDA/graphwin.h>
#include <LEDA/graph.h>
#include <LEDA/graph_gen.h> // graph generators
#include <LEDA/random_source.h> // random number source

void main()
{
        graph g;
        random_graph(g, 10, 20);
        GraphWin gw(g, 800, 400, "A random graph with 10 nodes and 20 edges");
        node n;
        random_source rs;
        forall_nodes(n, g)
        {       gw.set_position(n, point(rs.get() % 800, rs.get() % 400));
        }
        gw.display();
        gw.zoom_graph();
        gw.wait();
}
```

First, the above program uses LEDA's `graph` class to create an object that can used as input to a `GraphWin`. Then, a *graph generator* (one of several found in `graph_gen.h`) is applied to the graph, transforming it into a random graph with 10 nodes and 20 edges. Note, though, that the nodes generated are not embedded in the plane, i.e. they are not given coordinates when they are generated nor do they need to for a graph to be defined. To be displayed in a `GraphWin`, however, they must of course be assigned planar coordinates and this is done next using an iteration macro (see p.3) and a LEDA facility for generating random numbers called `random_source`. The last item of note is the call to the `zoom_graph()` function which brings the now displayed graph into focus by scaling it to the dimensions of the window.

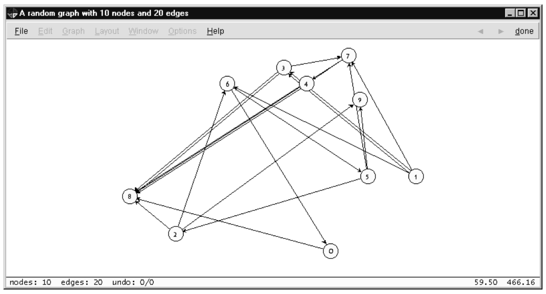


Figure 1.4: A random graph in a `GraphWin`

[5]`edges` can have other properties but we won't be going into that here.

Advanced data structures and algorithms

All the facilities presented in earlier sections are of course applicable in the field of CG, and can help spare you weeks of painstaking development effort. Still, it would be nice to have even higher level functionality at one's disposal since each gain in sophistication extends the range of practical problems that can be addressed. To that end, LEDA provides a wide selection of advanced algorithms and some associated data structures, that can be used for easy and rapid implementation of convex hull computation, polygon triangulation, plane sweeps and more. The below gives an example:

```
#include <LEDA/geo_alg.h> // geometry algorithms
#include <LEDA/random_point.h> // point generators
#include <LEDA/list.h>
#include <LEDA/window.h>
#include <LEDA/polygon.h>
#include <LEDA/circle.h>

void main()
{
        list<point>     lp;
        polygon poly;
        // generate 20 points in [-100,100] X [-100,100] and store in lp
        random_points_in_square(20, 100, lp);
        // compute the convex hull of lp and store result in poly
        CONVEX_HULL(lp, poly);
        window w(600,400,"A convex hull");
        // adjust user coordinate system to polygon
        w.init(-300,300,-300);
        w.display();
        w.draw_filled_polygon(poly, green);
        w.read_mouse();
        lp = poly.vertices();
        list<circle> lc;
        // find all empty circles passing through 3 or more points
        ALL_EMPTY_CIRCLES(lp,lc);
        circle c;
        forall(c, lc)
        {       w.draw_circle(c);       }
        w.read_mouse();
}
```

The above snippet of code first generates a set of points, then computes the convex hull of those points, storing it in a `polygon` object. Next, the polygon is rendered in a window as a solid rectangle and execution halts, waiting for a mouse click. Then, the list of vertices stored in the polygon object is extracted and used in computing the list of all circles that intersect three or more points on the newly computed convex hull. Finally, those circles are drawn in the window using an iteration macro and execution is then halted again to wait for a mouse click to terminate the application.

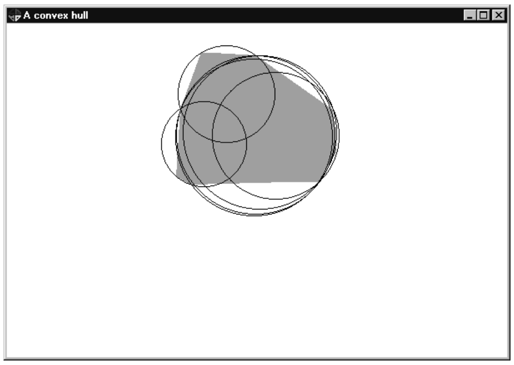


Figure 1.5: Geometry algorithms in action

Further study

The `GraphWin` is a good platform for experimentation with many of the algorithms in the LEDA toolbox since it can help visualize the workings of those algorithms. There are too many to be listed here, but the below list gives a brief summary of the important categories that you could look into if you want to explore LEDA further.

Graph searching: Include `basic_graph_alg.h` in your program to use various versions of breadth and depth first search algorithms for graph traversal.

Shortest path: Include `shortest_path.h` to use implementations of some classic algorithms for the shortest path problem(Dijkstra, Bellman-Ford) as well as some for the all pairs shortest paths problem.

Flow problems: For network problems, there is the `max_flow.h` header which declares a number of algorithms for computing all manner of flow problems in different kinds of network.

Graph embedding: As explained on p.7, a LEDA graph isn't necessarily embedded in the plane which is in good keeping with the mathematical definition of a graph. An implication of that is that if you want to embed a graph in the plane, you can do so in all sorts of systematic ways, not just the random one used in the `GraphWin` example shown earlier, and LEDA therefore provides functions for different kinds of embedding, all of which can be found in `graph_draw.h`.

The list could go on, but the best way to proceed is probably to follow the old adage "practice makes perfect", and get some hands-on experience, for example by working with the exercise in the next section.

1.4.3 A small exercise

The objective of this exercise is to implement a graphical display of a plane sweep using LEDA facilities and it is assumed that you are familiar with the concept of plane sweeping. If not, a good introduction is given in [2].

Your program must show graphically how a sweep line proceeds across a set of randomly distributed points and must indicate its progress by varying the colour of those points. To be more precise, the program should:

1. Create and display a set of points with colour X.
2. For each point p, beginning with the topmost and proceeding downwards:
   (a) Draw a line through p
   (b) Wait for user interaction
   (c) Remove the line through p
   (d) Change the colour of p to Y

The sweep ends when all points have been intersected by the sweep line, i.e. when they have all had their colour changed.

Your solution should include the above as the absolute minimum and if possible, you should also implement the following:

- Fluid animation - make the sweep line progress smoothly from one point to the next rather than by "stop motion" animation so that the user can better follow the execution of the sweep.
- Postscript save - use `window/GraphWin` member functions to add menu items/push button that allow the user to save Postscript screendumps of your program in action.
- Connect the dots - add functionality to your program for connecting each point to its nearest neighbour and make that functionality available to the user via a push button or menu item.

Your solution should include the source code and one or more screenshots of the running program. Optionally, you should make an executable version of your program publicly available.

1.5 Further resources

1.5.1 The MapDraw application

If you would like to see a somewhat larger example of how to use and customize LEDA, you can download documentation and source code for the *MapDraw* application at the following web address:

`home.worldonline.dk/~lars_yde`

MapDraw is small utility intended for use in conjunction with a map overlay implementation (also available at that same address). It allows the user to draw and save maps by interacting with a modified `GraphWin` window. These maps can then be fed to the overlay program which proceeds to compute an overlay of them in the manner described in [2]. MapDraw is currently at version 1.0, and is therefore fairly simple, providing a good stepping stone for further study of LEDA. You should also consult the many examples that ship with LEDA — they demonstrate almost every feature of the library and can help compensate for the sometimes terse documentation.

1.5.2 Web resources

The WWW has a plethora of information on computational geometry software, so rather than give a full list of links which might as well be obtained from a search engine, the below is brief and restricted to the most referenced CG sites on the net:

www.scs.carleton.ca/~csgs/resources/cg.html A starting point for CG resource exploration

www.ics.uci.edu/~eppstein/geom.html Geometry in action - popular, comprehensive, richly linked

compgeom.cs.uiuc.edu/~jeffe/compgeom/ Jeff Erickson's CG website - well organised, diverse and extensive

www.geom.umn.edu/software/cglist Directory of Computational Geometry Software

www-sop.inria.fr/prisme/personnel/bronnimann/cgt/index.html The CG Tribune - a newsletter on CG

www.dia.uniroma3.it/~gdt/ Graph Drawing Toolkit - for use with LEDA

www.cs.uu.nl/CGAL Homepage for the *Computational Geometry Algorithms Library*

www.mpi-sb.mpg.de/LEDA LEDA homepage

Bibliography

[1] compgeom.csuiuc.edu/ jeffe/compgeom/. Jeff erickson's cg website.

[2] M. de Berg et al. *Computational Geometry.* Springer-Verlag, 1997.

[3] K. Mehlhorn and St. Naher. *The LEDA Platform of Combinatorial and Geometric Computing.* Cambridge University Press, 1999.

[4] www.cs.uu.nl/CGAL. Computational geometry algorithms library.

[5] www.mpi sb.mpg.de/LEDA. Library of efficient data types and algoritms.