Computer Systems (SS 2012)
Exercise 1: April 2, 2012

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Wolfgang.Schreiner@risc.jku.at

March 8, 2012

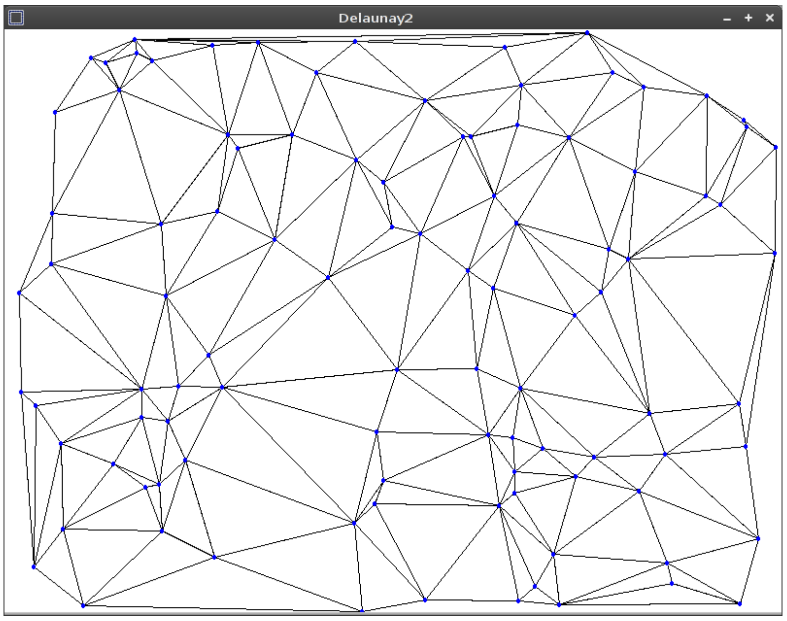
The exercise is to be submitted by the denoted deadline via the submission interface of the Moodle course as a single file in zip (`.zip`) or tarred gzip (`.tgz`) format which contains the following files:

- A PDF file `Exercise`*`Number-MatNr`*`.pdf` (where *Number* is the number of the exercise and *MatNr* is your "Matrikelnummer") which consists of the following parts:
  1. A decent cover page with the title of the course, the number of the exercise, and the author of the solution (identified by name, Matrikelnummer and email address).
  2. For every source file, a listing in a *fixed width font*, e.g. `Courier`, (such that indentations are appropriately preserved) and an appropriate *font size* such that source code lines do not break.
  3. A description of all tests performed (copies of program inputs and program outputs) explicitly highlighting, if some test produces an unexpected result.
  4. Any additional explanation you would like to give. In particular, if your solution has unwanted problems or bugs, please document these explicitly (you will get more credit for such solutions).
- Each source file of your solution (no object files or executables).

Please obey the coding style recommendations posted on the course site.

Exercise 1: Delaunay Triangulation

Write a program that computes and visualizes a Delaunay triangulation as shown below:



Algorithm A Delaunay triangulation $DT(S)$ of a set of points S in the plane is a subdivision of the convex hull of S into a set of non-overlapping triangles such that the set of the triangles' corner points equals S and the circumcircle of every triangle $\triangle_{a,b,c}$ with corner points a, b, c does not contain any other point from S[1].

We may compute $DT(S)$ by first computing a triangulation of $S \cup \{u, v, w\}$ where u, v, w are three points such that $S \subseteq \triangle_{u,v,w}$ and no point of S is on an edge of $\triangle_{u,v,w}$ ("u, v, w enclose S"). For every point $p \in S$, we choose the unique triangle that contains s and splits it into three triangles. By this split, the Delaunay property may be violated; we thus have to check for violations and restore the property.

This is performed by maintaining a set E that contains all possibly violating edges a, b together with the triangle t_0 with third point c from which this edge originated. We look for the opposite triangle t_1 of this edge which has some third point d. If d is in the circumcircle of t_0 or c_0 is in the circumcircle of t_1, the property is violated. It is restored by removing the edge a, b and adding the edge c, d to the triangulation. By this "flip", the edges a, d and d, b may violate the Delaunay property and have to be checked as well.

The algorithm depends on two tests

[1] `http://de.wikipedia.org/wiki/Delaunay-Triangulation`
`http://en.wikipedia.org/wiki/Delaunay_triangulation`

```
function DELAUNAY($S$)                                        ▷ $S$ is a set of points in the plane
    choose counter-clockwise points $u, v, w$ that enclose $S$
    $T \leftarrow \{\triangle_{u,v,w}\}$                      ▷ $T$ is always a triangulation of $\triangle_{u,v,w}$
    for $p \in S$ do
        choose $\triangle_{a,b,c} \in T$ such that $p \in \triangle_{a,b,c}$
        $T \leftarrow T \backslash \{\triangle_{a,b,c}\} \cup \{\triangle_{a,b,p}, \triangle_{b,c,p}, \triangle_{c,a,p}\}$   ▷ $p$ splits $\triangle_{a,b,c}$ in three triangles
        $E \leftarrow \{\langle a, b, \triangle_{a,b,p}\rangle, \langle b, c, \triangle_{b,c,p}\rangle, \langle c, a, \triangle_{c,a,p}\rangle\}$   ▷ edges $E$ may violate property
        while $E \neq \emptyset$ do
            choose $\langle a, b, t_0\rangle \in E$           ▷ $t_0$ has two corners $a, b$
            $E \leftarrow E \backslash \{\langle a, b, t_0\rangle\}$
            search for $t_1 \in T$ such that $t_1 \neq t_0$ and $t_1$ has two corners $a, b$
            if such $t_1$ exists then                         ▷ $t_0$ and $t_1$ share edge $a, b$
                let $c$ be the third corner of $t_1$ (different from $a, b$)
                if $c \in \bigcirc_{t_0}$ then               ▷ polygon $a, d, b, c$ violates property
                    $t_2 \leftarrow \triangle_{a,d,c}$        ▷ flip edge $a, b$ to edge $c, d$
                    $t_3 \leftarrow \triangle_{d,b,c}$
                    $S \leftarrow S \backslash \{t_0, t_1\} \cup \{t_2, t_3\}$   ▷ restore Delaunay property
                    $E \leftarrow E \cup \{\langle a, d, t_2\rangle, \langle d, b, t_3\rangle\}$   ▷ we have new possible violations
                end if
            end if
        end while
    end for
    return $\{t \in T \mid t$ has not an edge on $\triangle_{u,v,w}\}$
end function
```

1. $p \in \triangle_{a,b,c}$ (point p is in the triangle with corners a, b, c): this can be determined by checking whether p is on the same side (left or right) of each of the three edges of the triangle. For this purpose, compute the coefficients e, f, g of the line equation $ex + fy + g = 0$ of the edge and then determine the value $ep_x + fp_y + g$. Point p is in the triangle, if this value has the same sign for all three edges (why?).

2. $p \in \bigcirc_t$ (point p is in the circumcircle of the triangle $t = \triangle_{a,b,c}$): Provided that a, b, c are the corners of t in *counter-clockwise* order (we maintain this convention), this can be determined by checking whether the following determinant is positive:

$$\begin{vmatrix} a_x - p_x & a_y - p_y & (a_x^2 - p_x^2) + (a_y^2 - p_y^2) \\ b_x - p_x & b_y - p_y & (b_x^2 - p_x^2) + (b_y^2 - p_y^2) \\ c_x - p_x & c_y - p_y & (c_x^2 - p_x^2) + (c_y^2 - p_y^2) \end{vmatrix} > 0$$

Program It shall be possible to invoke the program with the name of a file, e.g.

```
DelaunayMain input.txt
```

where `input.txt` contains a sequence of text lines of form

```
width height
number
x_0 y_0
...
```

where

- *width* and *height* describe the size of the window in which all points of S can be visualized (i.e., all points have coordinates from 0, inclusive, to *width* respectively *height*, exclusive);
- *number* is the number of points;
- x_i and y_i are the integer coordinates of the points.

It shall be also possible to call the program without a file name. In that case, for *width* = 800 and *height* = 600, 50 random points are to be generated. For this purpose, use the C++ random number generator `rand()` initialized by a call of `srand(`n`)` where n are the digits of your birth date (e.g. $n = 24121989$) (you have to use `#include <cstdlib>`). Test the program by

1. calling it with the file `input.txt` given on the web site. The outcome must be as indicated in above picture;
2. calling it without argument. The outcome must be the visualization of the triangulation of a random set as explained above.

Deliver the source code and the screenshots of the two executions indicated above.

In the following, we explain how the program shall be structured into classes. Write for each class C a separate header file C`.h` and (unless for the two classes whose implementation is given inline below) a separate implementation file C`.cpp`. Use a separate file `DelaunayMain.cpp` for your main program.

Points Implement a class `Point` with minimal interface

```
class Point
{
public:
  double x;
  double y;
  static void draw(Point *p);
  static void draw(Point* p0, Point* p1);
  static void clear(Point* p0, Point* p1);
};
```

Objects of this class represent points. The static function `draw(`p`)` draws a point p as a small filled circle; the function `draw(`p_0`,`p_1`)` draws a line from p_0 to p_1 together with the two endpoints. The function `clear(`p_0`,`p_1`)` erases the line (by drawing a line in the background color) and redraws the endpoints.

Triangles Implement a class `Triangle` with minimal interface

```
class Triangle
{
  static int side(Point *p0, Point *p1, Point *p);
public:
  Point* p0;
  Point* p1;
  Point* p2;
  static bool inside(Triangle *t, Point *p);
};
```

Objects of this class represent triangles; the static function `inside(`t`,`p`)` returns `true` if and only if point p is inside triangle t. For this purpose, it uses an auxiliary function `side(`p_0`,`p_1`,`p`)` which returns $+1, 0, -1$ (the sign of the test value explained above).

Memory Management Actually, we represent points and triangles as *pointers* to heap-allocated objects of type `Point` and `Triangle`. In order to keep track of the objects and deallocate them appropriately, implement a class `Heap` with minimal interface

```
class Heap
{
  static Point** point; static int sp; static int np;
  static Point** triangle; static int st; static int nt;
  static void resizePoints();
  static void resizeTriangles();
public:
  static void init();
  static Point* newPoint(double x, double y);
  static Triangle* newTriangle(Point* p0, Point* p1, Point* p2);
  static void clear();
};
```

The class maintains two arrays `point` and `triangle` of sizes `sp` and `st` which hold in the first `np` respectively `nt` positions the pointers to the points and triangles allocated so far. A call of `init()` allocates the initial arrays from the heap (use some reasonable values for `sp` and `st`) and sets `np` and `nt` to 0. The remainder of the program allocates a new point p by a call of `newPoint(`p_x`,`p_y`)` and a new triangle $\triangle_{a,b,c}$ by a call of `newTriangle(`a`,`b`,`c`)`. The functions allocate corresponding objects from the heap and store the returned pointers in their internal arrays. If some array becomes full, a new arry of twice the old size is allocated, the pointers are copied to the new array, and the old array is deleted. At the end of the program, a call of `clear()` deallocates all points, triangles, and the internal arrays (thus returning to the initial state).

Triangle Sets The set T is to be implemented as a pointer to an object of class

```
#include <list>
class TriangleSet
```

```
{
  list<Triangle*> triangles;
  list<Triangle*>::iterator next;

public:
  static void add(TriangleSet* set, Triangle* t)
  { set->triangles.push_back(t); }
  static void remove(TriangleSet* set, Triangle* t)
  { set->triangles.remove(t); }
  static void gotoStart(TriangleSet* set)
  { set->next = set->triangles.begin(); }
  static bool hasNext(TriangleSet* set)
  { return set->next != set->triangles.end(); }
  static Triangle* getNext(TriangleSet* set)
  { Triangle* t = *(set->next); set->next++; return t; }
};
```

This class (whose implementation you need not understand for the purpose of this exercise) allows you to initialize S with `new TriangleSet()` (the empty set). A call `add(`S`,`t`)` adds triangle t to S, `remove(`S`,`t`)` removes t from S. You can iterate over all elements of S by first calling `gotoStart(`S`)`. After that you can repeatedly call `hasNext(`S`)` to determine, if S has another triangle. If yes, a call of `getNext(`S`)` will return that triangle.

Edge Sets The set E is to be implemented as a pointer to an object of class `EdgeStack`

```
#include <list>
class EdgeStack
{
  list<Point*> point0;
  list<Point*> point1;
  list<Triangle*> triangle;

public:
  static void add(EdgeStack* set, Point* p0, Point* p1, Triangle *t)
  { set->point0.push_back(p0); set->point1.push_back(p1); set->triangle.push_back(t); }
  static bool isEmpty(EdgeStack* set)
  { return set->point0.empty(); }
  static Point* getFirstPoint(EdgeStack* set)
  { Point* p = set->point0.back(); set->point0.pop_back(); return p; }
  static Point* getSecondPoint(EdgeStack* set)
  { Point* p = set->point1.back(); set->point1.pop_back(); return p; }
  static Triangle* getTriangle(EdgeStack* set)
  { Triangle* t = set->triangle.back(); set->triangle.pop_back(); return t; }
};
```

This class (whose implementation you need not understand for the purpose of this exercise) allows you to initialize E with `new EdgeStack()` (the empty set). A call `add(`E`,`a`,`b`,`t`)` adds $\langle a, b, t\rangle$ to E. A call `isEmpty(`E`)` determines whether E is empty.

If not, subsequent calls of `getFirstPoint(`E`)`, `getSecondPoint(`E`)`, `getTriangle(`E`)` return a triple $\langle a, b, t\rangle$ of E and remove it from E.

Triangulation The main functionality can be implemented by the following class:

```
class Delaunay
{
  static TriangleSet* T;
  static EdgeSet* E;
  static void insert(Point* point)
  static Triangle* search(Point *point)
  static void restore()
  static Triangle* search(Point *p0, Point *p1, Triangle *t)
  static bool violates(Triangle *t0, Triangle *t1, Point* p0, Point*p1)
  static Point* thirdPoint(Triangle *t, Point* p0, Point*p1)
  static bool inside(Point* p0, Point* p1, Point* p2, Point* p)
public:
  static void triangulate(int width, int height, int number, Point** points)
};
```

This class provides a function `triangulate(`w`,`h`,`n`,`p`)` which displays in a window of size $w \times h$ the Delaunay triangulation of the set of n points stored in array p (with coordinates from 0 inclusive to w respectively h exclusive). A simple enclosing triangle (which one?) has a horizontal edge of length $2w$ with y-coordinate -1 and height $2h+3$.

Rather than to ultimately remove the superfluous edges of the enclosing triangle (as indicated by the last line of the algorithm), it suffices not to draw any edge that leads to one of the corner points. Also it is recommended (this simplifies testing) not to draw only the final triangulation, but to draw every edge as soon as it is constructed and to erase the edge from the screen, as soon as it is removed (you may ignore screen artifacts that may arise). If you call `Sleep(`*`delay`*`)` (see file `Main.cpp` in `Drawing.zip`) whenever you draw/erase an edge, you get a nice animation (similar to the one on the course site).

For the implementation, it may be helpful to use the following auxiliary functions:

- `insert(`p`)` inserts a new point from *points* into the current triangulation.
- `restore()` checks and restores the Delaunay property after the insertion of a point.
- `search(`p`)` delivers that triangle of T that contains p.
- `violates(`t_0`,`t_1`,`a`,`b`)` checks whether the two triangles t_0 and t_1 with common edge a, b violate the Delaunay property.
- `thirdPoint(`t`,`a`,`b`)` returns the third point c of triangle t which has the two other points a and b.
- `inside(`a`,`b`,`c`,`p`)` returns true if p is within the circumcircle $\bigcirc_t$ with $t = \triangle_{a,b,c}$.

Feel free to choose different auxiliary functions. In any case, *make ample use* of auxiliary functions, do *not* implement the algorithm as a single big function (in particular, every function should not contain more than one loop).