Computer Systems (SS 2011)
Exercise 1: April 4, 2011

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

March 7, 2011

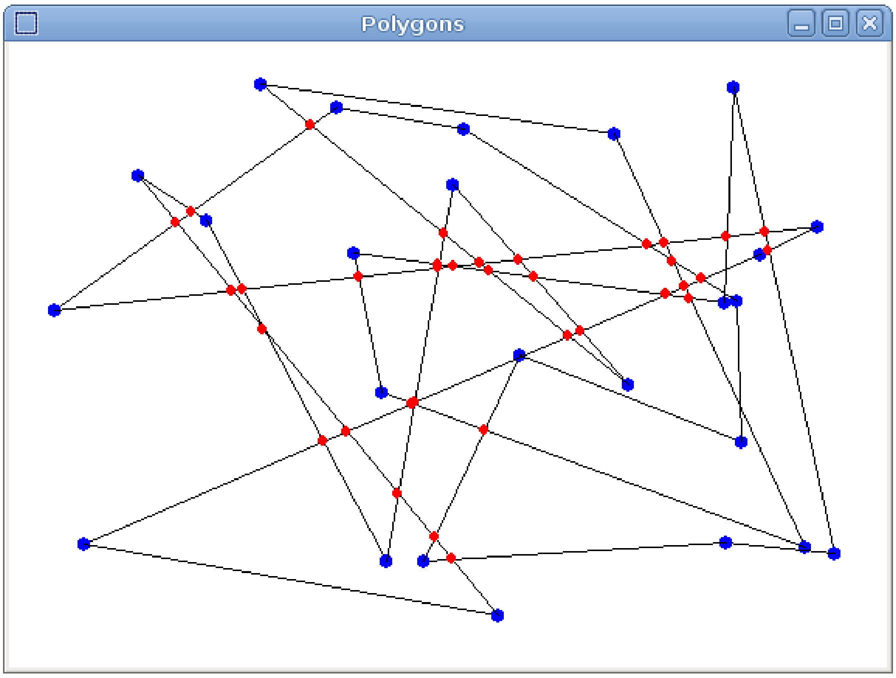
The exercise is to be submitted by the denoted deadline via the submission interface of the Moodle course as a single file in zip (`.zip`) or tarred gzip (`.tgz`) format which contains the following files:

- 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: Polygons

Write a program that reads, processes, and draws closed polygons as depicted by the following picture:



In detail, the program shall consist of the following components with the given public interfaces (you may freely introduce additional private and public functions):

1. A class `Math` for comparing floating point numbers:

```
class Math
{
public:
  static void setAccuracy(double a, double r);
  static bool equals(double c1, double c2);
  static int sign(double c);
};
```

`setAccuracy(`a`,`r`)` stores in static member variables the absolute/relative accuracies a, r for floating point comparisons (choose reasonable defaults, e.g. $a = 1$ equates pixel-identical point coordinates): c_1 and c_2 are "equal", if $|c_1 - c_2| < a$ or if $|(c_1 - c_2)/c_2| < r$ (for the second test, order c_1, c_2 such that $|c_1| < |c_2|$)[1]. Implement this equality in `equals(`c_1`, `c_2`)`. The function `sign(`c`)` returns 0, if c "equals" 0 and otherwise, ± 1, depending on the sign of c. In the rest of the program, only these functions are allowed for comparing/testing floating point numbers.

[1]See `http://www.cygnus-software.com/papers/comparingfloats/Comparing%20floating%20point%20numbers.htm`

2. A class `Point` that implements points in the plane:

```
class Point
{
public:
  Point(double x = 0, double y = 0);
  double getX();
  double getY();
  void draw(unsigned int color=0, int radius=1);
  void draw(Point &p);
};
```

The constructor `Point(`x`,` y`)` constructs a point with coordinates x, y (default 0). The selectors `getX` and `getY` return the coordinates. The function `draw(`c`,`r`)` draws a filled circle whose center is the point with color c (default black) and radius r (default 1).

3. A class `Lines` that implements the intersection of two lines:

```
class Lines
{
public:
  static Point* intersect(Point& p0, Point& p1, Point&p2, Point &p3,
                          bool segment = true);
  static void drawIntersection(Point& p0, Point &p1, Point& q0, Point& q1,
      unsigned int color = 0);
};
```

The function `intersect(`p_0`,` p_1`,` p_2`,` p_3`,` s`)` implements the intersection of two lines running through points $p_0, p1$ and p_2, p_3, respectively. The result is a pointer to the intersection point of the two lines or (if the lines are collinear) the null pointer. If s is true, the two lines are interpreted as line segments bounded by the given points; an intersection point is then only returned, if it is within the bounds of both segments. The logic for this function is:

- If $p_0 = p_1$ and $p_2 = p_3$, then, if $p_0 = p_2$, then return p_0, and else the null pointer.
- If $p_0 = p_1$ then, if p_0 is on the other line/segment, then return p_0, and else the null pointer.
- If $p_2 = p_3$ then, if p_2 is on the other line/segment, then return p_2, and else the null pointer.
- If the lines are collinear, return the null pointer.
- Determine the intersection point.
- If s is true, and the point is not on both segments, return the null pointer.
- Return the point.

In above algorithm, avoid to use of the division operator, since it is only partially defined. For instance, rather than testing whether $a/b = c/d$, you should test whether $a \cdot d = c \cdot b$ (using the comparison operator of class `Math`, of course).

The function `drawIntersection(`p_0`, `p_1`, `p_2`, `p_3`, `s`)` uses `intersect()` to compute and draw the intersection point (the point is to be discarded after drawing).

4. A class `Polygon` that implements closed polygons:

```
class Polygon
{
public:
  Polygon();
  ~Polygon();
  void add(double x, double y);
  void random(int n, int x, int y, int w, int h, int seed = 0);
  bool read(const char* filename);
  void draw(unsigned int color1 = 0, unsigned int color 2 = 0);
  void drawIntersection(Polygon& polygon, unsigned int color = 0);
};
```

The class maintains internally an array that holds the points (objects of type `Point`) $p_0, \ldots, p_n$ of the polygon to which new points may be added. If the array becomes full, a bigger array is allocated and the old array is disposed. The constructor `Polygon()` creates a polygon with no points; the destructor `~Polygon()` disposes the point array. The function `add(`x`,`y`)` adds a point to the polygon (resizing the array, if necessary). The function `random(`n`, `x`, `y`, `w`, `h`, `s`)` adds to the polygon n random points in the coordinate range $x \ldots x+w$ respectively $y \ldots y+h$ using the seed value s for the random number generator (use the standard functions `srand()` and `rand()`). The function `read(`f`)` reads from a text file with name f the contents

```
x1 y1
x2 y2
...
xn yn
```

that represent the coordinates (floating point numbers) of n points and adds these to the array. The return value of `read()` is `true`, if the file could be successfully read, and `false`, if some problem occurred. The function `draw(`c_1`, `c_2`)` draws the closed polygon indicating by bullets of color c_1 the points of the polygon and by somewhat smaller bullets of color c_2 all other points where the segments of the polygon self-intersect. The function `drawIntersection(`p`, `c`)` draws those points in color c where the polygon intersects with another polygon p. Make sure that your code also works with polygons with less than 3 points.

Test these components by a program that performs (at least) the following tasks:

1. It reads file `poly1` and draws the polygon in a window. The outcome must be as indicated in above picture.

2. It reads files `poly1` and `poly2` and draws the polygons and their common intersection points in a window.
3. It reads files `poly1` and `poly3` and draws the polygons and their common intersection points in a window.
4. It generates two random polygons with self-chosen seed values that represent the Unicode values of the initial letters of your given name and your family name and draws the polygons and their intersection points in a window (explicitly give the seed values that you used and place the polygons such that they have some intersection points but do not completely overlap). The number of points n_1 and n_2 of the polygons you may choose on your own.

Avoid code duplication but make extensive use of auxiliary functions. Write for each class C a separate header file C`.h` and an implementation file C`.cpp`. Use a separate file `Polygons.cpp` for your test program. Deliver the source code and screenshots of the four windows indicated above.