Object-Oriented Programming in C++ (SS 2022)
Exercise 1: April 7, 2022

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

March 9, 2022

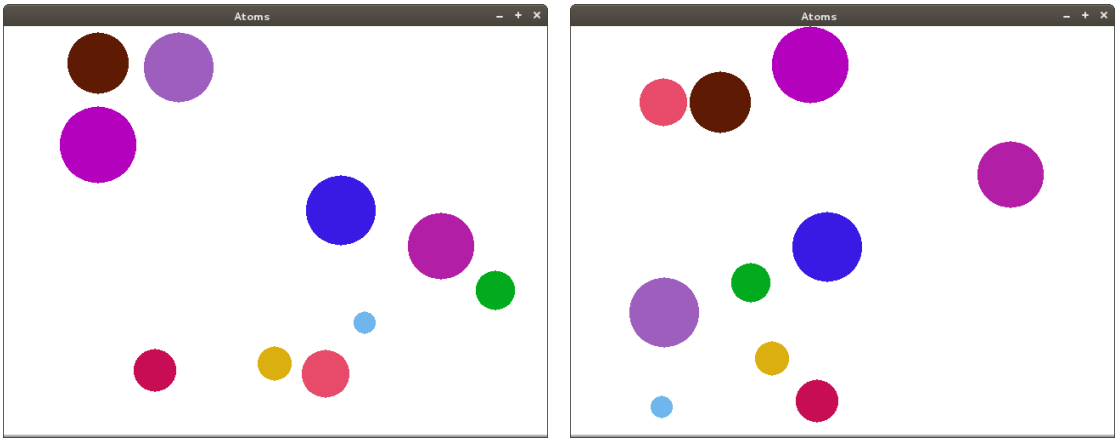
The exercise is to be submitted by the denoted deadline via the submission interface of the Moodle course as a single file in zip (`.zip`) or tarred gzip (`.tgz`) format which contains the following files:

- 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: Atoms

Write a program `Atoms` that animates a collection of bouncing disks ("atoms"):



Each atom moves with a certain velocity in a certain direction. If it collides with the boundary of the window, it bounces back from the boundary; likewise, if two atoms collide, they bounce back from each other (see also the animation on the course site).

Introduce a (structure/class) type `Atom` whose values represent atoms by its color c (an integer), radius r, center position (x, y), and velocity vector (v_x, v_y) (all double precision floating point numbers). The main program shall then execute the following piece of code:

```
#include <iostream>
#include <cstdlib>
#include <cmath>

#include "Drawing.h"

#if defined(_WIN32) || defined (_WIN64)
#include <windows.h>
#else
#include <time.h>
static void Sleep(int ms)
{
  struct timespec ts;
  ts.tv_sec = ms/1000;
  ts.tv_nsec = (ms%1000)*1000000;
  nanosleep(&ts, NULL);
}
#endif
```

```
using namespace std;
using namespace compsys;

...

int main(int argc, const char* argv[])
{
  beginDrawing(W, H, "Atoms", 0xFFFFFF, false);
  int n = number(argc, argv);
  Atom *atoms = new Atom[n];
  init(n, atoms, argc, argv);
  draw(n, atoms);
  cout << "Press <ENTER> to continue..." << endl;
  string s; getline(cin, s);
  for (int i = 0; i < F; i++)
  {
    update(n, atoms);
    draw(n, atoms);
    Sleep(S);
  }
  delete[] atoms;
  cout << "Close window to exit..." << endl;
  endDrawing();
}
```

The program first creates a window of size $W \times H$ with white background on which the results of output operations are not immediately shown (above screenshots are for $W = 640$ and $H = 480$, use these values also in your program). It then determines the number n of atoms and creates a corresponding array which is appropriately initialized. Then it draws the n atoms on the screen and waits until the user presses the "Enter" key. The program then updates in F iterations the atoms (i.e., it computes their new positions and velocities), draws the atoms and waits for S milliseconds before the next iteration is started. When the user closes the window, the allocated array is deallocated and the program is terminated.

Your task is to implement the functions `number`, `init`, `draw`, and `update` as follows (whenever numerical constants are described, introduce correspondingly named constants in your program and give them values of your choice):

- If the program is called without command line arguments (i.e., `argc==1`), `number` returns a pre-defined value N and `init` creates N atoms in random colors with random sizes in some predefined interval $[R_0, R_1]$ with random velocities in some predefined interval $[V_0, V_1]$ at random positions; however, the atoms must be fully contained in the window and must not intersect with each other (try to place a new atom 3 times; if then no suitable position can be found, the program may be aborted).

For this purpose, write on the basis of the library function `rand()` that returns a non-negative random integer an auxiliary function that creates a random non-negative integer within a given interval. Before generating the random numbers, initialize the random number generator by executing the command `srand(time(0))`. Each program run will then generate a new set of atoms.

- The program may also be called with one command line argument (i.e., `argc==2`). This argument `argv[1]` is the name of a text file that contains a sequence of lines

  ```
  n
  c1 r1 x1 y1 vx1 vy1
  c2 r2 x2 y2 vx2 vy2
  ...
  cn rn xn yn vxn vyn
  ```

  whose first line contains the number n of atoms. The file then contains n additional lines each of which describes the values of one atom. The function `number` shall read the first line of the file and return n, the function `init` shall read the remaining n lines and create the corresponding atoms. If the input file does not exist or is ill-formed, the program may be aborted with an error message.
- However the program is called (with or without argument), the number n of atoms and the descriptions of the initial values of the atoms are printed to the standard output in the form shown above.
- The function `draw` first clears the screen by drawing a white rectangle of size $W \times H$. It then draws each atom as a filled circle at its current position with its specific size and color. After drawing all atoms, it calls the function `flush()` which updates the content of the window, i.e., makes all atoms visible.
- A first version of `update` only checks for collisions of every atom with every wall (i.e., the atoms may run through each other): if the distance of the center of the atom from a wall is less than or equal the radius, the atom is re-positioned at a distance which is equal to the radius; the corresponding component of the velocity vector is then negated (v_x for a collision with a vertical wall, v_y for a collision with a horizontal wall).

Test your program once without argument and once with the file `input.txt` given on the course site. Give in each case as a deliverable the text output of your program (showing the initial values of the atoms), the screenshot of the initial situation and the screenshot of the final situation after $F = 200$ updates (if you set $S = 40$, then about 1000/40=25 updates per second are performed, thus the animation runs less than 10 seconds).

After that, extend `update` to also consider the collisions between atoms. In more detail, after updating the position of all atoms and handling their collisions with the window boundaries, perform for each pair of atoms i and j with $i < j$ the following tasks:

1. Determine the distance between the atoms i and j; if the distance is bigger than the sum of the radii of the atoms, they do not collide and we are done. Otherwise, we continue as described below.

2. If the distance is smaller than the sum of the radii (i.e., the atoms overlap), we move one atom along the vector between the centers of both atoms so far away that the distance of both atoms becomes exactly the sum (i.e., both atoms just touch).

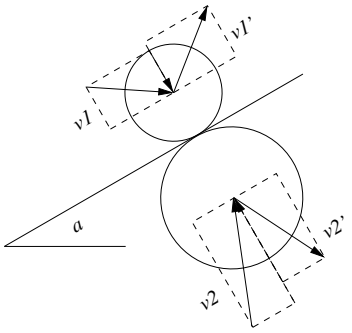
3. Assume both atoms would move towards each other's centers with velocities v_1 and v_2 (the velocities thus have different signs). Then the theory of elastic impacts[1] tells us their new velocities v_1' and v_2' as

$$v_1' = 2V - v_1, v_2' = 2V - v_2$$

where

$$V = \frac{m_1 \cdot v_1 + m_2 \cdot v_2}{m_1 + m_2}$$

denotes the velocity of the common center of gravity of both atoms and m_1 and m_2 denote their masses (we consider in our program as the masses the squares of the atom's radii, i.e., the masses are proportional to their areas as circular discs).



4. To apply this knowledge to the general case (where atoms may collide at arbitrary angles), we proceed as follows (see the figure above):

   a) We compute the vector of the tangent of the intersection point of both atoms (the normal of the vector between their centers).

   b) We transform this vector from Cartesian form (horizontal and vertical coordinate) into polar form (length and angle); this gives us its angle a towards the horizontal.

   c) We transform both velocity vectors into polar form and subtract a from their angles (we thus rotate the coordinate system such that the tangent is along the horizontal axis).

   d) We transform the rotated velocity vectors back into Cartesian form; the vertical components of both vectors denote the two velocities v_1 and v_2 by which the circles move towards each others center; from these the new vertical velocities v_1' and v_2' are computed as indicated above.

[1] http://de.wikipedia.org: Stoß (Physik); https://en.wikipedia.org/wiki/Elastic_collision

e) The resulting vectors (with unchanged horizontal components and transformed vertical components) are converted into polar form; we add a to their angles (and thus rotate the coordinate system back into its original position).

f) The resulting velocity vectors are converted into Cartesian form; these forms are stored as the new velocities in the atoms.

Please note that above strategy is physically not completely correct since the repositioning of an atom (due to its moving inside another atom) might position it into a wall or a third atom; however, we will ignore this problem in our exercise (in rare situations you might observe this behavior in your animation).

In your program avoid code duplication (and recomputation of values) but make extensive use of auxiliary functions (and variables). In particular, the functions

```
void toPolar(double x, double y, double &r, double &a)
{
  a = atan2(y, x);
  r = sqrt(x*x+y*y);
}

void toCartesian(double r, double a, double &x, double &y)
{
  x = r*cos(a);
  y = r*sin(a);
}
```

can be used to perform the conversions of a vector from Cartesian form (x, y) into polar form (length r and angle a) and back.

Test your updated programs in the same way as the original one and give the screenshots of the new final situations.

Deliver in your report the complete source code of the final version of the program with the new `update` function (it is not necessary to also provide the original one).