Object-Oriented Programming in C++ (SS 2024)
Exercise 1: April 11, 2024

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

February 5, 2024

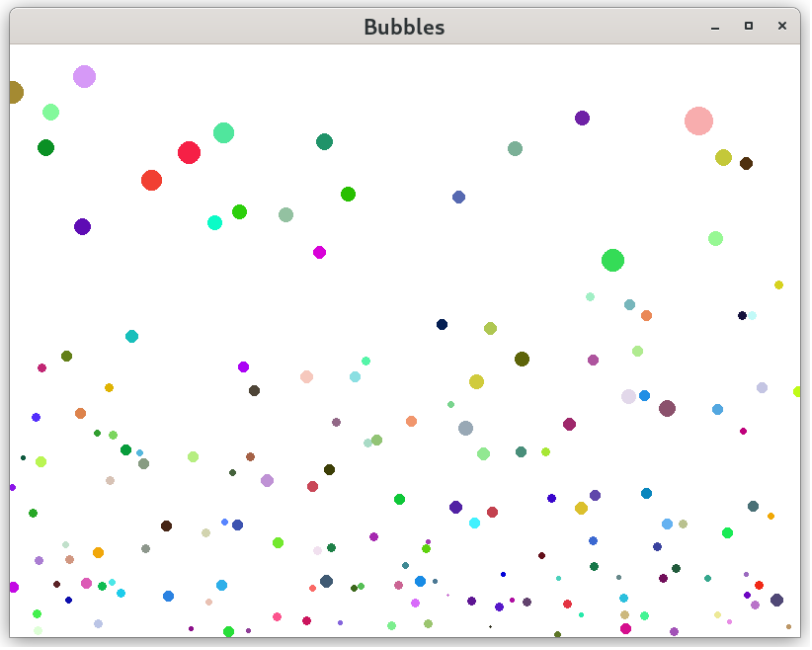
The exercise is to be submitted by the denoted deadline via the submission interface of the Moodle course as a single file in zip (`.zip`) or tarred gzip (`.tgz`) format which contains the following files:

- 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: Bubbles in a Pool

Write a program `Bubbles` that animates (in two dimensions) the motion of colored air bubbles in a pool filled with water:



Model The pool is modeled as a rectangle of size $W \times H$ (we model physical lengths in units of mm where every pixel on the screen represents one square millimeter). Bubbles are spheres that emerge at the bottom of the pool at random times and random positions with a random color and a random volume in the interval $(VMIN, VMAX)$.

The animation runs in F iterations that are separated by pauses of S ms; each iteration simulates the passing of $T = M \cdot S/1000$ s physical time where the factor M may be used to "slow down" ($M < 1$) or "speed up" ($M > 1$) the animation. In each iteration new bubbles may be created at the bottom, all bubbles move up, and eventually disappear at the top. There may exist at most N bubbles: thus, if there are currently B bubbles, $N - B$ new bubbles may be created; every new bubble is created with probability $A\%$ at a random position in the interval $(0, W)$. If a bubble touches the upper end of the pool, it bursts, i.e., it disappears from the simulation.

Our sample animation uses $W = 800$, $H = 600$, $VMIN = 10$, $VMAX = 500$, $F = 500$, $S = 20$, $M = 0.1$, $N = 200$, $A = 20$.

The movement of a bubble is guided by the following considerations:

- Each bubble emerges at depth $D := H$ with some random volume V_D. However, the product $M := D \cdot V_D$ is proportional to the bubble's mass which remains constant during its lifetime. Thus the volume V_d of this bubble at any depth d can be calculated as $V_d := M/d$; from $V_d = 4/3 \cdot {r_d}^3 \cdot \pi$ the radius r_d of the bubble at depth d can be determined.

- The upward velocity v of a bubble can be approximated by the equation $v := r^2 \cdot G \cdot \nu / C$. Here r is the current radius of the bubble, $G = 9806.65$ mm/s^2 is the gravity of Earth, $\nu = 14$ mm^2/s is the kinematic viscosity of water, and C is a constant factor with $C = 3$ for $r < 0.5$ mm and $C = 9$ else. Therefore the upward velocity of a bubble is proportional to its cross section.

If two bubbles collide, they "merge" in the following way: the smaller bubble disappears from the simulation while the larger bubble "inherits" the "mass value" M of the smaller bubble by adding it to its own value; this subsequently increases the upward velocity of the bubble.

Implementation To implement the model, introduce a (structure/class) type `Bubble` whose values represent bubbles by their liveness status *alive* (a boolean), color c (an integer value), "mass value" M, horizontal position x, and depth d (the pair (x, d) denotes the center of the bubble). Except for *alive* and c, all values are represented as double precision floating point numbers. The program shall then have the structure depicted in Figure 1 (for all the parameters described in the model introduce in your program correspondingly named constants):

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. It then determines the number n of bubbles (by default $n = N$, but see below) and creates an array with n bubbles. The program then updates in F iterations the states of the bubbles, draws them, and waits S milliseconds for the start of the next iteration. At the end, the states of the bubbles are printed, the array is deallocated, and the program waits for the user to close the window before it terminates

Your task is to implement the functions `init`, `update`, `draw`, and `print` as follows:

- If the program is called without command line arguments (i.e., `argc==1`), `init` sets n to N and creates an array of n bubbles that are all set to "dead".
- 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
  alive1 c1 M1 x1 d1
  ...
  aliven cn Mn xn dn
  ```

  whose first line contains the number n of bubbles. The file then contains n additional lines each of which describes the values of one bubble. The function `init` reads the first line to determine n; it then creates an array with n bubbles that are initialized as described by the file.
- The function `print` prints the final state of the simulation in the form shown above. Its output can be captured in a file with which the program can then be started with the name of this file to continue the simulation.
- The function `draw` first clears the screen by drawing a white rectangle of size $W \times H$. It then draws each alive bubble as a filled circle at its current position with its current size and specified color. After drawing the bubbles, it calls the function `flush()` which updates the content of the window, i.e., makes all alive bubbles visible.
- The function `update` iterates over all bubbles; a dead bubble is raised from the dead with probability A% and initialized with depth H and the other values explained in the model

```
#include <iostream>
#include <cmath>
#include <random>

#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, char* argv[])
{
  beginDrawing(W, H, "Bubbles", 0xFFFFFF, false);
  int n; Bubble* bubbles;
  init(n, bubbles, argc, argv);
  for (int i = 0; i < F; i++)
  {
    update(n, bubbles);
    draw(n, bubbles);
    Sleep(S);
  }
  print(n, bubbles);
  delete[] bubbles;
  cout << "Close window to exit..." << endl;
  endDrawing();
}
```

Figure 1: The Program Structure

description. If a bubble is alive (or has just been raised), its upward velocity v is calculated as explained in the model description; then (since each iteration represents physical time duration T) its depth is decreased by $v \cdot T$. If its depth becomes less than the radius of the bubble, the bubble dies. The first version of `update` need not check for collision of bubbles with each other (i.e., bubbles may "pass" through each other).

In your program avoid code duplication (and recomputation of values) but make extensive use of auxiliary functions (and variables). In particular, 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(0)` to make the simulations reproducible. Alternatively, you may call `srand(time(0))`; then each program run will then generate a new simulation.

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 final values of the bubbles), the screenshot of the initial situation and the screenshot of the final situation.

After that, extend `update` to also consider the collisions between bubbles. In more detail, after updating the position of all bubbles, check for each pair of alive bubbles i and j with $i < j$ whether the distance of their centers is less than the sum of their radii; if yes, the smaller bubble dies and the larger one inherits its mass value.

Test your updated programs in the same way as the original one and give the screenshot 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).