# Lab 3: part 1

## Course goals

- To understand fundamental data structures concepts: binary heap.
- To implement and use data structures and algorithms in application programs.
    - To use a priority queue to implement an event-driven simulation.

## Getting started

This exercise is about improving the given implementation of a simulation system for particles collision.

Please ensure you read the entire lab description thoroughly, before starting any exercises, including the preparation steps given below.

Completing the following tasks before the Lab3 HA session will help you make the most of your lab time.

- Create a folder for this lab part named e.g. `Lab3-part1`.
- Download the zipped folder with the files for this exercise from the course website and unzip it into the folder created in the previous step.
- Execute the remaining instructions in the file `README.md` (can be opened with Notepad). The instructions given in `README.md` are like the ones given for the labs in TNM094.
- Run the program (with the `main` function located in `lab3.cpp`). When asked for a particle file, enter `billiards10.txt` to start the particle collision simulation. Then, repeat the process using `p20000.txt` and then `sam4.txt`.
- Review the concepts introduced in lecture 7 and read sections 6.1 to 6.4 of the course book.
- Do exercise 1 and exercise 2.

Note that you cannot modify the given code, except where explicitly indicated by the lab instructions.

If you have any specific question about the exercises, then send us an e-mail. Be short and concrete, otherwise you won't get a quick answer. You can write your e-mail in Swedish. Add the course code to the e-mail's subject, i.e. "`TND004:` …".

## Event-driven simulation: particles collision system

This exercise is about simulating the motion of $N > 0$ colliding particles according to the laws of elastic collision. Such simulations are widely used in molecular dynamics to understand and predict properties of physical systems at the particle level. This includes the motion of molecules in a gas, the dynamics of chemical reactions, and atomic diffusion. The same techniques apply to other domains that involve physical modeling of particle systems including computer graphics, computer games, and robotics.

The *hard sphere model* is used (or billiard ball model) which is an idealized model of the motion of atoms or molecules in a container. We focus on the two-dimensional version called the *hard disc model* which makes the following assumptions.

- $N$ particles in motion, confined in a unit box.
- Particle $i$ has position ($rx_i, ry_i$)[1], velocity ($vx_i, vy_i$), mass $m_i$, and radius $\sigma_i$.
- Particles interact via elastic collisions with each other and with the reflecting boundary (walls).
- No other forces are exerted. Thus, particles travel in straight lines at constant speed between collisions.

There are two natural approaches for simulating the system of particles.

- *Time-driven simulation*. Discretize time into time intervals of size *dt*. Update the position of each particle after every *dt* units of time and check for overlaps. If there is an overlap, roll back the clock to the time of the collision, update the velocities of the colliding particles, and continue the simulation. The major drawback of this approach is time efficiency. To ensure a reasonably accurate simulation, we must choose *dt* to be very small, and this slows down the simulation. On the other hand, we may miss collisions if *dt* is too large.
- *Event-driven simulation*. Because it's possible to predict when collisions will occur (see section "Particles"), we focus only on those times at which interesting events[2] occur, when using event-driven simulation approach. Thus, our main challenge is to determine the ordered sequence of particle collisions. We address this challenge by maintaining a *priority queue* for future events, ordered by time.

In this exercise, we follow the event-driven simulation approach. Classes `Particle`, `Event`, `PriorityQueue`, and `CollisionSystem` implement the needed functionality. These classes are briefly described below.

The `main` function in `lab3.cpp` executes the simulation. The simulation duration is set to 10.000 seconds.

## Data files

Several input files with particles are given in the folder `collisionsystem\data`. As illustrated below, the first line contains the number of particles $N > 0$. The remaining $N$ lines contain six real values (a particle's center's position, velocity, mass, and radius) followed by three integers (particle's color in RGB). Particles do neither overlap with other particles nor with walls (the simulation box boundaries).

```
N
rx ry vx vy mass radius r g b
rx ry vx vy mass radius r g b
…
```

[1] Coordinates of the particle's center point. A particle can be idealized as a circle.
[2] e.g. collisions

## Exercise 1: priority queue

The system utilizes a priority queue, which is implemented as a sorted vector. This implementation is provided as a template class in the file `priorityqueue-vector.h`.

This exercise involves re-implementing the `PriorityQueue` template class using a binary heap, as covered in lecture 7 of the course. To achieve this, follow these steps:

- Start by commenting "`#define USE_PRIORITY_QUEUE_VECTOR`" near the top of the file `collisionsystem.h`[3].
- Implement each member function in the file `priorityqueue.h` (search for "`ADD CODE HERE`"). Additional public member functions are not allowed, but private member functions may be added. Ensure that the implementation is based on a binary heap.
- Test your implementation. Make sure the line "`#define TEST_PRIORITY_QUEUE`" (near the top of the file `priorityqueue.h`) remains uncommented, execute the code, and check that no assertion fails.
- Check whether it's possible to run the simulation program. To this end, comment the line "`#define TEST_PRIORITY_QUEUE`" and then run the program with the given data files. Be sure to use the files `p2000.txt`, `brownian.txt`, and `sam4.txt`, as they contain large number of particles.

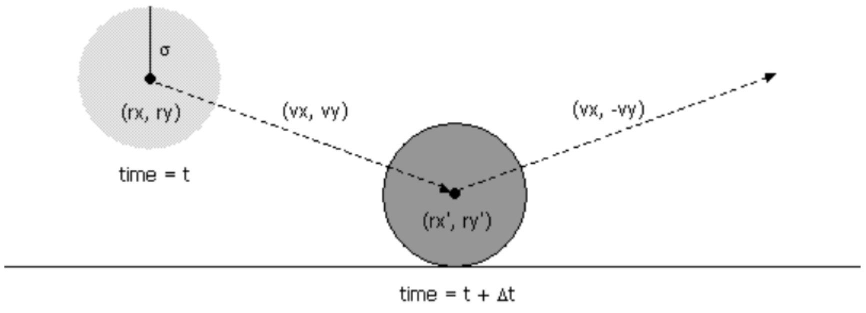
Besides the `PriorityQueue` template class, the system also utilizes other classes, which are detailed in the following section.

## Classes

### Particles

Class `Particle` represents particles and it has member functions to predict collisions of a particle with other particles and walls: `timeToHit`, `timeToHitVerticalWall`, and `timeToHitHorizontalWall`.

We illustrate below how to predict if and when a particle with position (*rx, ry*) at time *t* will collide with a horizontal wall.



Since the coordinates are between 0 and 1, a particle comes into contact with an horizontal wall at time $t + \Delta t$, if the quantity $ry + \Delta t.vy$ equals either σ or (1 - σ). Solving for $\Delta t$ yields:

[3] To rerun the simulation using a priority queue implemented as a vector, you need to uncomment "`#define USE_PRIORITY_QUEUE_VECTOR`" again.

$$\Delta t = \begin{cases} (1 - \sigma - ry) \,/\, vy & \text{if} \quad vy > 0, \\ (\sigma - ry) \,/\, vy & \text{if} \quad vy < 0, \\ \infty & \text{if} \quad vy = 0. \end{cases}$$

An event representing the collision of the particle with wall can then be added to the priority queue with priority given by the predicted time of the collision, i.e. $t + \Delta t$.

It's possible that the formula above predicts that no collision with a horizontal wall will occur ($\Delta t = \infty$) or that a collision occurs too far in the future, and consequently, outside the set simulation duration. In this case, an event is not added to the priority queue. Similar ideas apply to collisions with vertical walls and other particles.

When a collision occurs, then the velocity of the involved particle(s) needs to be updated. This is done by member functions `bounceOf`, `bounceOffVerticalWall`, and `bounceOffHorizontalWall`.

You don't need to focus on fully understanding the details of particle collision physics (i.e. implementation of the `Particle` class member functions).

## Events

Class `Event` represents either a collision or a rendering event. More concretely, there are four types of events: a rendering event; a collision with a vertical wall; a collision with a horizontal wall; or a collision between two particles.

Rendering events cater for the visualization of all particles in the system at a given point in time. Rendering events are automatically generated by the system with a given frequency (e.g. 10 times per time unit).

Each instance of this class stores the event's occurrence time and two pointers to particles (one or both pointers might be null, depending on whether the event represents a collision with a wall or is a rendering event, respectively).

## Collision system

Class `collisionSystem` represents the system of particles and it has a member function to simulate (named `simulate`) the system for a given time period (i.e. the simulation duration).

Collisions modify the direction in which particles move. Since collisions events are not necessarily inserted in the queue by chronological order, it's possible that collision events already in the queue become invalid[4] when another event is inserted.

The main loop of function `simulate` executes the following steps:

1. Pull lowest-time event off event queue. Assume this event takes place at time $t$.
2. If the event corresponds to an invalidated collision, discard it. The event is invalid if one of the particles has participated in a collision since the time the event was inserted onto the priority queue.
3. If the event corresponds to a physical collision between particles $i$ and $j$:
    a. Advance all particles to time $t$ along a straight-line trajectory.

[4] For instance, a collision between two particles at a time $t$ changes their velocity. Thus, some events already in the queue, and that have been predicted for some time $t' > t$, may not occur.

   b. Update the velocities of the two colliding particles *i* and *j* according to the laws of elastic collision.

   c. Determine all future collisions that would occur involving either *i* or *j*, assuming all particles move in straight line trajectories from time *t* onwards. Insert these events onto the priority queue.

4. If the event corresponds to a physical collision between particle *i* and a wall, do the analogous procedure (steps 3.a – 3.c) for particle *i*.

Make sure you understand the function `CollisionSystem::simulate` before you answer the questions in exercise 2.

## Exercise 2

You are now asked to analyze the advantages of using a priority queue implemented with a binary heap for the simulation compared to one implemented as a sorted vector. Consider the function `CollisionSystem::simulate` and and carefully prepare answers to the following questions. Assume there are $n > 0$ particles.

- What is the time complexity of the `for`-loop on line 59, assuming the heap is implemented as a sorted vector? How does this complexity change if the heap is implemented as a binary heap? Use Big-O notation and justify your answer.
- Consider the `while`-loop in line 64 and assume the queue has already $k > 0$ events. Estimate the time complexity of a single iteration of the `while`-loop, assuming the priority queue is implemented as a sorted vector. Then, repeat the analysis for a priority queue implemented as a binary heap. Use Big-O notation and justify your answer.

## Presenting part 1 and deadlines

The exercises in this lab part are compulsory and you should demonstrate your solutions during the lab session *Lab3 RE*. Read the instructions given in the labs webpage and consult the course schedule.

Necessary requirements for approving your lab are given below.

- Present the implementation a priority queue as required in exercise 1. Make sure you can explain the logic behind the code.
- The code must be readable, well-indented, and use good programming practices.
- Present and clearly justify your answers to the questions in exercise 2. This will be assessed through an oral examination conducted by the lab assistant.

If your code for lab3/part1 has not been approved in the scheduled lab session *Lab3 RE* then lab 3 is considered a late lab. Late labs can be presented provided there is time in a another RE lab session. All groups have the possibility to present one late lab on the extra RE lab session scheduled in the end of the course.