Imperial College London – Department of Computing

MSc in Computing Science

580: Algorithms

Assessed Coursework
Event Driven Simulation

Groups

This is a group exercise. You may work in groups of up to three people. Each of you has been given a code respository, but you should choose a group leader and submit your work via their repository. **Only the group leader should submit**.

Introduction

Computer simulations have extensive applications in science and engineering. In this exercise you will be implementing a Java event driven simulation. This type of simulation works by maintaining a queue of *events*. The key component is this queue, which must work very hard. You will be implementing the queue, the events that are placed into the queue, and completing the simulation controller itself. The rest of the framework is provided for you.

Event Driven Simulations

In an event driven simulation, events are continually being created. When an event is created it is assigned a time at which it will happen. The events then queue up, ordered by time, waiting to be processed. Time within the simulation is measured in *ticks*, and only progresses when events happen. So, given the following queue:

$$[E_1(t = 1.67), E_2(t = 1.73), E_3(t = 2.33), E_4(t = 2.65)]$$

the time will jump from 1.67 ticks to 1.73 ticks, then 2.33, then 2.65 as events E_1 to E_4 happen. Since there are no events between these times it does not matter that the system jumps like this. This method of advancing the clock is why the process is called *event driven*.

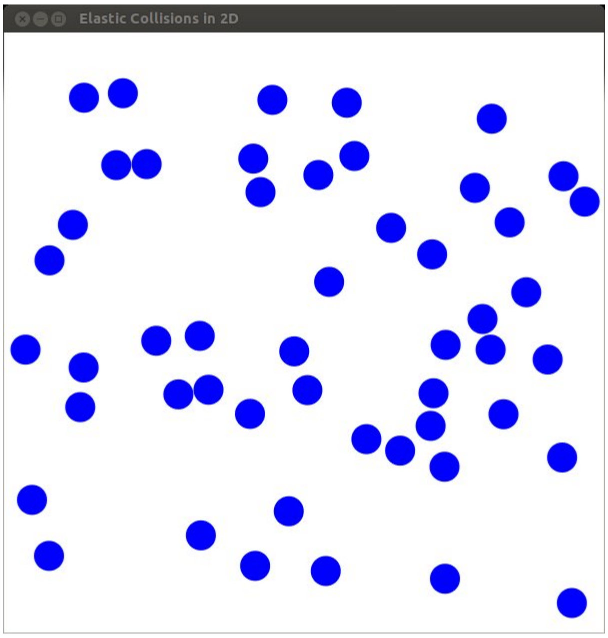

Figure 1: A particle simulation.

The Simulation

Specifically, you will be completing a program that implements a two-dimensional simulation of the movement of particles undergoing elastic collisions. You have been provided with the following parts of the simulation framework, some of which need to be completed:

Particle a `Particle` object represents a single particle, which has a position, a velocity, a mass, a diameter and a colour. A `Particle` can be made to *move* for some amount of time *dt*. The class also provides static methods that compute collision events based on a particle's current trajectory, and update a particle object when a collision happens.

Simulation Model A `ParticlesModel` contains all the particles for a simulation, and provides methods that move all the particles, and find the next collisions for all the particles.

Graphical Display The `ParticlesView` class provides a graphical display of a `ParticlesModel`. The class provides an `update()` method to update the display.

Simulation Clients The classes

- `SimpleSimulation`,
- `DiffusionSimulation`, and
- `BrownianSimulation`

set up and run specific simulations. They create a specific set of particles, put them into a model, pass this model into a new simulation and run it. You can try running these programs once your system is complete (or even during testing).

Event types a set of interfaces and classes defining a hierarchy of events (see below).

Simulation Controller The `ParticleSimulation` class controls the simulation. It maintains the queue, determines the time, and processes events.

What To Do

Obtain the Exercise Files

You will need to clone the skeleton respository to your DoC home directory in order to work on it. Later you will need to *push* your changes back to the server.

- You can get your skeleton repository by issuing the following command (all on one line, replacing the occurrence of *login* with your college username):

  ```
  git clone https://gitlab.doc.ic.ac.uk/lab1718_spring/580particlesimulation_login.git
  ```

  Or, if you have set up ssh key access you can use:

  ```
  git clone git@gitlab.doc.ic.ac.uk:lab1718_spring/580particlesimulation_login.git
  ```

- This will create a new directory called `580particlesimulation_`*login*. Inside you will find the following files / directories:
  - `src/simulation` — this directory contains Java source files of classes in the `simulation` package.
  - `src/utils` — this directory contains Java source files of classes in the `utils` package.
  - `.git` and `.gitignore`

Complete The Program

There are several classes for you to implement as part of the exercise.

The Priority Queue (70%)

The file `src/utils/MinPriorityQueue.java` contains the skeleton of a min priority queue class which you should complete. The order of the objects in the queue will be determined by comparing the objects themselves, rather than associating them with separate keys. This is done as follows. `MinPriorityQueue` is a generic class that can provide a queue of objects of any (single) type that is `Comparable`. In other words, any class that implements the `Comparable` interface. This interface is part of the `java.lang` library package and contains a single method:

```
public int compareTo(Object o)
```

that must be provided by all `Comparable` types. This method allows objects of the same type to be compared. So, given a `Comparable` object `comp1`, of type `T`, calling `comp1.compareTo(comp2)`, where `comp2` is also of type `T`, will return:

- 0 if the objects are 'equal'

- a negative integer if `comp1` is ‘less than’ `comp2`
- a positive integer if `comp1` is ‘greater than’ `comp2`

The definition of ‘less than’, ‘greater than’ or ‘equal’ depends on how the type `T` has implemented `compareTo`. Whatever the definition, your queue can use the `compareTo` method to decide how the different `T` objects that it contains should be ordered.

The important public interface methods of `MinPriorityQueue` are:

- `void add(T elem)` – which adds a new element to the queue.
- `T remove()` – which returns the ‘smallest’ element, removing it from the queue.

These methods will both need to run in $O(\log_2 N)$ time or your program will not be able to keep pace with all the events. You may want to design the class to contain other methods too.

Events (20%)

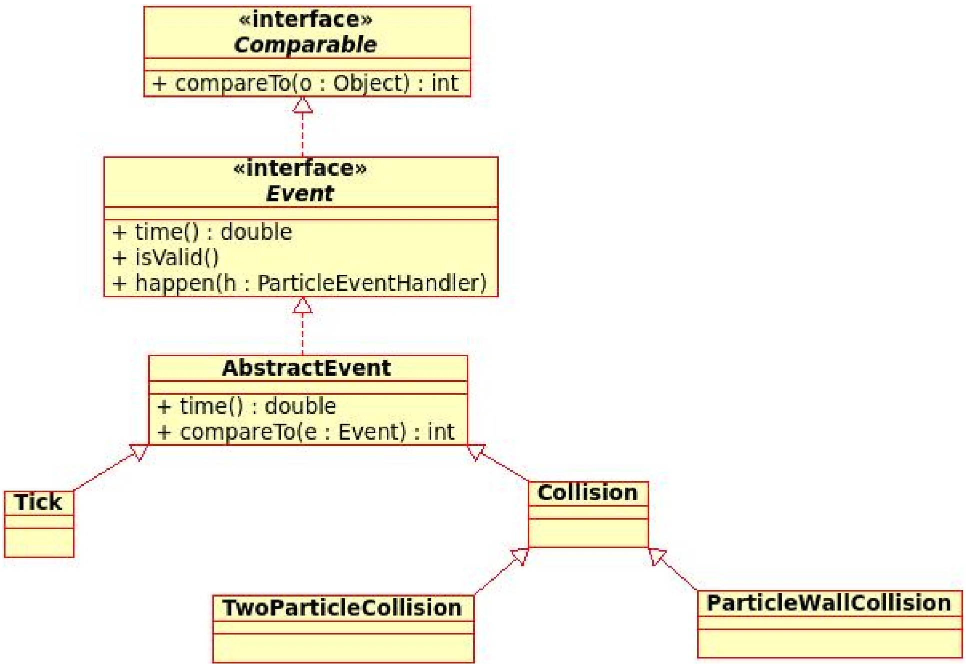

Figure 2: The Event classes.

The objects that are put into the priority queue are various types of `Event`. The hierarchy of `Event` types is shown in Figure 2. At the top is the `Comparable` interface. All `Events` must be `Comparable` so that they can be placed into the queue. Next is the `Event` type itself, which is also an interface. `Event` defines three methods:

- `public double time()` – returns the time that this event will occur,
- `public boolean isValid()` – returns true if this event can occur,
- `public void happen(ParticleEventHandler h)` – makes the event happen;

that must be implemented by all concrete event types. The `happen()` method warrants a little explanation. This method will be called by the simulation controller. When it is called the `Event` has two things to do. First it should update any objects that are part of the event (like particles). Once this is done, it must pass on the message that it has now occurred. This is where the `ParticleEventHandler` parameter comes in.

The Event Handler

The last thing an `Event` should do when it happens is to let an event handler know. The event handler then performs any other actions that are necessary as a result of the event. (Hopefully it will become clear what this means!) `ParticleEventHandler` is another interface (supplied), and defines two methods:

- `public void reactTo(Tick t)` – reacts to a `Tick` event
- `public void reactTo(Collision c)` – reacts to a `Collision` event

A `ParticleEventHandler` must be passed to the `Event` as a parameter of `happen()`. The event should complete its job by calling `reactTo()` on the event handler and pass itself as a parameter. The `reactTo()` method that is called will be determined by the type of event. This is an example of the *visitor* design pattern (the event handler being the visitor).

Implementing the Events

Abstract Events `AbstractEvent` is the next class for you to complete. The source file is `src/simulation/AbstractEvent.java` (this class is part of the `simulation` package). `AbstractEvent` should implement `Comparable`, but only partially implement `Event` (making it abstract). To implement `Comparable`, `AbstractEvent` must define the `compareTo` method. `AbstractEvent`s should be compared according to their *times*. So, an `AbstractEvent` will also need to have a time attribute (a double). This attribute will be set when the event is created, and will not change. You should also implement the `time()` method, which should return the value of the time attribute. `AbstractEvent` should not implement `isValid()` or `happen()`.

Ticks Next we come to the first concrete `Event` type, `Tick`. A `Tick` is a very simple event, but it drives the whole simulation. A `Tick` event happens every 1.0 *ticks*. You must create a `Tick` class (none is provided), placing it into the `simulation` package. As shown in Figure 2, `Tick` should extend `AbstractEvent`. It must also implement `isValid()` and `happen()`. This is simple since a `Tick` is always valid, and when a `Tick` happens the only thing it must do is make a *callback* to the `ParticleEventHandler` (see above).

Collisions A `Collision` is the other direct subclass of `AbstractEvent`. Unlike `Tick`, `Collision` itself is also abstract. A skeleton source file is supplied in `src/simulation/Collision.java`. A collision can occur between a `Particle` and a `Wall`, or between two `Particle`s. It always involves particles though, which are set when the `Collision` is created. `Collision` should also define the `isValid()` method from the `Event` interface. A collision will be valid *only if* the particles involved have not been through any other collisions since it was created. When a collision is created it is a type of prediction. The collision might happen if the particles continue on their current paths for long enough. However, if they hit other particles, or walls, between the time the prediction is made (when the `Collision` object is created) and the time the collision is supposed to occur, the collision becomes invalid. A `Particle` object (class supplied) keeps track of how many collisions it has been through, so a `Collision` must check for any differences when `isValid()` is called.

Particle–Wall Collisions Finally, we come to the concrete collision classes. The first is `ParticleWallCollision`. You must create this class, which extends `Collision`. A `ParticleWallCollision` must know what `Particle` and what `Wall` it will involve, and what time it is supposed to happen. Like `Tick` it should also define the `happen()` method. When a `ParticleWallCollision` happens it must use the `Particle` class to update the state of its particle (via one of the `collide()` methods) and then tell the `ParticleEventHandler` to react.

Particle–Particle Collisions The final type of event is a `TwoParticleCollision`. This is equivalent to a `ParticleWallCollision` but, you guessed it, involves two particles.

The Simulation Controller (10%)

The simulation is controlled by the `ParticleSimulation` class. The skeleton code for this is in `src/simluation/ParticleSimulation.java`.

- When a `ParticleSimulation` is created it is supplied with a `ParticlesModel`.
- The `ParticleSimulation` must create a `ParticlesView`, telling it what name to display and what data model to represent.
- The `ParticleSimulation` class maintains the queue of events. This queue will be an instance of your min priority queue class. It is initialised by adding a single `Tick` event, set for time 1, and predicted collision events for all particles in their initial state.
- The `ParticleSimulation` class is in charge of time (ticks). It should maintain a clock (an attribute of type double) which is updated as events occur.
- The `ParticleSimulation` has a `run()` method that is called to start the simulation. The method begins by creating a new thread for the display to run in. This code has been supplied. Following this the method should start processing events and keep running as long as there are more events available. Its operation is as follows:
  - Remove an event from the queue
  - *If the event is valid*
    - Update the current time to the time of the event
    - Move the particles for the amount of time elapsed since the previous event
    - Tell the event to happen

- The last repsonsibility of `ParticleSimulation` is to act as event handler. The class implements `ParticleEventHandler`, so it must provide the two methods that are part of that interface. When these methods are called the simulation should:
  - If a `Tick` has happened:
    - Pause for `FRAME_INTERVAL_MILLIS` milliseconds by calling the `Thread.sleep()` method. (This is the only point at which 'real' time is relevant.)
    - Tell the display to update.
    - Add a new `Tick` event to the queue.
  - If a `Collision` has happened:
    - Add new collisions to the queue for the particles that just collided.

Testing

You should test your classes' methods as you develop them, and ultimately try running the three client programs. Before submitting please check the test output on LabTS to ensure your code compiles and runs without errors in the test environment.

Submission

Submit By 1900, 19th February 2018

1. **Make a Group on CATE**. The group leader should go to the hand in page and create a group. Group members should sign the declaration.
2. **Push To Gitlab.** Use `git add`, `git commit` and `git push` to update the *group leader's* repository on the Gitlab server with the changes you have made to the skeleton code. Use `git status` to confirm that you have no local changes you have not pushed, and then inspect the files on Gitlab: `https://gitlab.doc.ic.ac.uk`.
3. Click the appropriate **Submit To CATe** button.

Assessment and Feedback

The marks for the exercise are assigned as follows:

MinPriorityQueue	70%
Event classes	20%
ParticleSimulation	10%

This exercise will be marked and returned by **5th March 2018**. Feedback on your solution will be given on the returned copy.