Eurockéennes simulation
VI51 project

Adrien BERTHET, Isaac CHIBOUB, Thibault MICLO and Camille MOUGIN

SPRING 2014

Summary

1 Requirement Analysis

1.1 Authors of the project

This project's group is composed of four I2RV students : Adrien Berthet, Isaac Chiboub, Thibault Miclo and Camille Mougin.

1.2 Project's context

The Eurockéennes de Belfort is a large music festival happening in the beginning of July. It lasts three days and hosts a hundred thousand visitors each year. It takes place in the Malsaucy peninsula - just a few miles away from the city of Belfort.

Attendants can enjoy world-famous bands playing on four different stages. On-site stands are offering food, drinks, clothes and accessories. Obviously men and women bathrooms can also be found all over the festival. The entrance ticket includes an access to the camp site to sleep at night.

1.3 Goals of the project

The project globally consists in simulating people's actions during one day of festival. Our goals can be listed as follows.

- Model the site environment : stages, stands of different kind, bathrooms, trees, barriers, exit/entrance.
- Simulate crowd movements among stands and stages during the day, according to each people particular need.
- At a given moment, allow simulation user to drop a bomb inside the festival area. This event must lead to the evacuation of the site in a fast and efficient way.

2
Design part

2.1 Description of the design architecture

2.1.1 Agents

We can define 3 types of agents :

Attendant, somebody who is here to attend concert
Vigil, someone who keep an eye on people. We can have regular vigils, and specialized vigils like medics, who help injured people
Shopkeepers, sell goods to attendants people

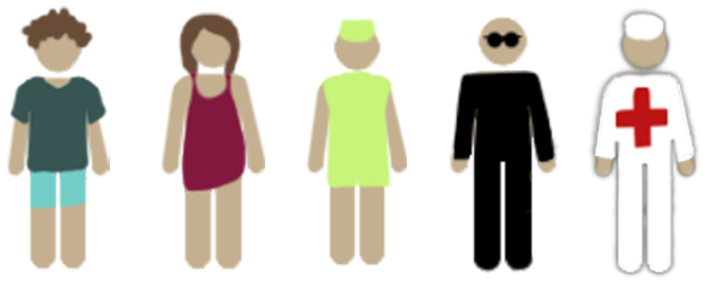


FIGURE 2.1 – Agents representation

2.1.2 Environment

The environment will be represented in 2D. The map is a simple area, filled with fixed entities and different kind of people. A single entrance/exit is present, with emergency exits all around the area.

Entities can be listed as follows :

Stages, where the bands play, and people gather around it
Obstacles, trees, barriers, etc...
Toilets, distinction between men's and women's
Food stands, where people go eat and drink
Shopping stands, where people can shop souvenirs
One entrance/exit + emergency exit, Where you can enter or leave the festival

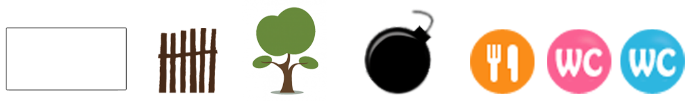


FIGURE 2.2 – Entities representation

2.1.3 GUI

The GUI part is in both sub-packages, framework and motion. It is called from the launch of the MainProgram class. It contains all the references to graphical data (sprites), and it manages the capture of the mouse to set the bomb somewhere on the map. Apart from that, it just draws the scene at each step of the program, when all agents have performed their actions.

2.1.4 Behaviours

For each agent, we can specified differents behaviour :

Attendants each person has a list of concert he wants to see, he will try to go to each of them. Different schedules can be made :

— Regular schedule, one concert each hour, no conflict, the schedule will only be interrupted by natural needs (food, toilets, drink)
— Groupie schedule, one group in particular, skip the previous concert to be in the first row, then regular schedule
— Conflicted schedule : some concerts are in conflict, so the person will see x% of the first one and 100-x% of the second one

Outside of schedules, people can act differently :

— Regular people, drink a bit, eat a bit, use the restrooms, go to concerts from the beginning to the end
— Drugged people, slow motion, can slow the evacuation process
— Angry people, various reasons (alcohol, pogos, fights), can hurt others people. People hurted should be evacuated by vigils as soon as seen
— A few, randomly selected can be heavy drinkers and start to act totally randomly (should be stopped by vigils as soon as seen by one)
— Idiot people, take the evacuation as an opportunity to be on the front row

Vigils As presented above, there is two types of vigils :
— Regular vigils : stay at the same place all the day, have a sight of seeing to detect drunk people and angry people, maybe rotation of given people at given posts
Passive stationed near his post (stage supervisory, entry body check)
Active motion until in range of troublemakers then warning or neutralization, back to original post after operation success

— Medics : same as vigils but for injured people

Shopkeepers sells goods to people, stay behind the stand the all night and evacuate when needed

Apart from those specific behaviors for each type of agent, there is also a panic behavior, which is triggered by the user (events defined below).

— Augmentation of maximal speed
— Reduction of perception field
— Following crowd direction until perception of emergency exit

2.1.5 Tools

The application will be written in Java, in a Maven project. For the interface, it will be in 2D (aerian view), using Swing. To manage the multi-agent aspect of the project, Janus 1.0 will be used. The teamwork was optimized with the help of the SCM Git.

Sources can be found here : https://github.com/zarov/EurockSimulation

2.2 Class diagram

Given the important size of the simulation, and to be coherent between everything, we designed a class diagram as follow :

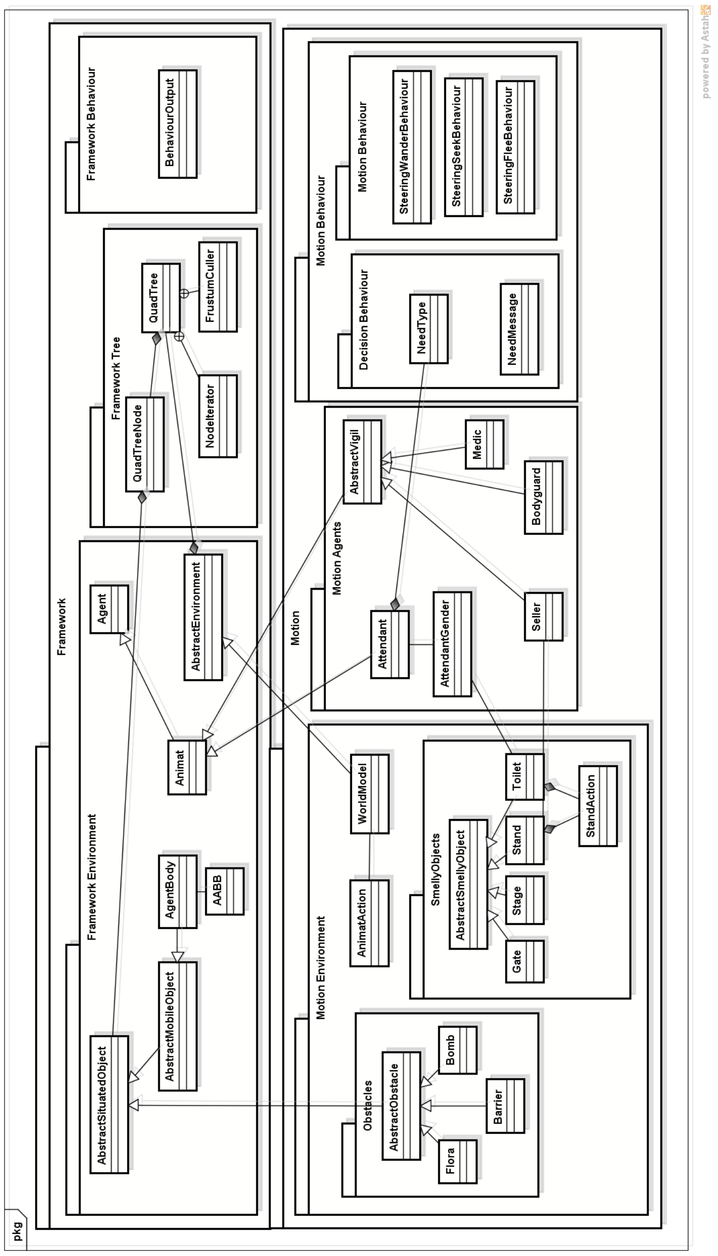


FIGURE 2.3 – Class diagram

2.3 Design choice and explanations

2.3.1 Decision making

Attendants are the only agents that follow a real decision making process. Attendant movements are guided by their instant need. Implemented needs are thirst, hunger, seeing gig need and going to bathroom need. Each attendant moves in direction of the object that will satisfy its higher need ; once satisfied, another need takes over that lead to a new movement target.

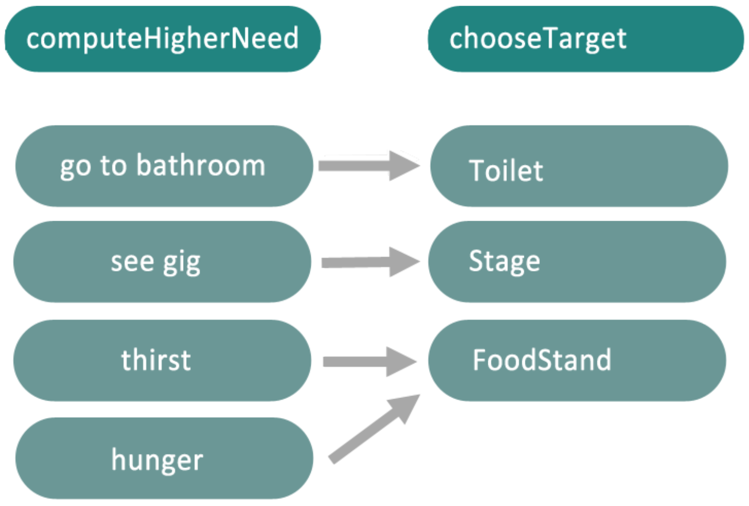


FIGURE 2.4 – Decision making graph

2.3.2 Classical and panic behaviors

Attendant Behaviour

— Update his need function of messages received
— Optional : increase his need randomly
— Get perception function of its frustum
— Choose a seek target function of its higher need and perceptions
— Choose a flee target (closer obstacle) for collision avoidance
— Run both flee and seek behavior function of the target

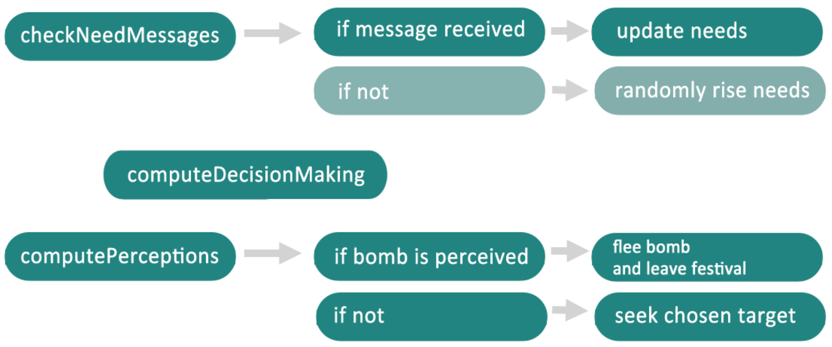


FIGURE 2.5 – Attendant behavior graph

Seller Behaviour

— Get perception
— Check if bomb is seen
— Else send a message to the attendant on to of the stand stack

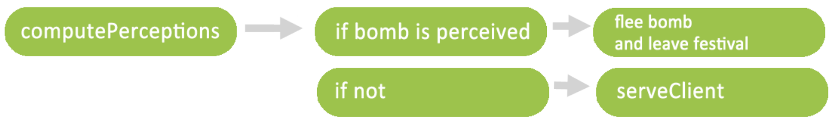


FIGURE 2.6 – Seller behavior graph

Medic Behaviour

— Get perception
— Check if an attendant is hurt
— In case one is, rush to him until he his healed or transported to safety

Bodyguard Behaviour

— Get Perception
— Stay near stage
— Check if an attendant has a drunk behaviour
— Guide him out of the gig far from the stage
— Tag team with medic to protect hurted attendant from the crowd

2.4 Used techniques

2.4.1 Quadtree structure

To store environment's objects, we could use a Grid, but this structure is too basic and can not contain all the data of the environment. If the simulation needs to fit reality, there will be at least 80.000 attendants, and a lot of other people as vigils, medics etc... So a lot of data needs to be handle, and a simple Grid can't do that. A Grid is also for discrete movements, and there isn't a need for such movements.

A Tree is more adapted, especially a QuadTree, that can divide the space in 4 in each of his node. Compared to a Grid, it is way faster, because the search can be focused on one corner of a chosen node at each level. On a theoretical level, a Grid has a complexity of $O(n)$ while a Tree as one of $O(\log n)$. It is the final choice made to manage the environment.

The implementation of the QuadTree is an adaptation of the QuadTree seen in courses. The structure is, of course, the same as every QuadTree : each node has a set of children (maximum four), and each child has a unique parent. The root of the tree is the entire map of the Eurockéennes. There is various methods within the tree. First, a classic Iterator has been implemented, which traverses the node in a depth-first iteration. There is also a find function which takes as an argument an object of the world, based on the box of elements and nodes. As AABB boxes are used, it is possible to tell if a box is inside a node. The algorithm is as follows :

```
while iterator has next do
    node <= iterator next if not(node.box contains object) then
        /* as the node doesn't contain the object, we can remove children
           of the node                                                  */
        iterator remove 4 next
    else
        if node is leaf then
            break
        end
    end
end
```

2.4.2 Frustum culler

In the tree, there is also a function that allow to get the objects in the perception field of an agent. And if we want to be more efficient, it can remove objects that the agent doesn't see. So basically it works the same as the find function, but instead of looking if a node contains an object, it's searching if the object is in the frustum, a AABB box to represent the perception field, after being sure that the frustum at least intersects with the node in which it's taking a look at. So at the end the algorithm is as this :

```
while iterator has next do
    node <= iterator next if frustum intersects node.box AND contains node.object then
        add object to percepts
    end
end
```

2.4.3 Agents communication

In order to communicate between agents, we used the mailbox system offered by Janus library. This system allowed us to realize interactions between people in one particular case : attendant's food delivery. Indeed once a hungry festival attendant reached a stand to get some food, he places himself in the stand queue. Stand seller then peek the first client in the queue and send him several messages containing actions that will apply on his needs, for example -7 on drinking need and +3 on going to bathroom need.

2.4.4 Collision avoidance

One of the main aspect of the UV is the collision management between agents, themselves and environment. But as we were focused on managing the needs and behaviours of each agent, we didn't take time to implement correctly a good collision avoidance. Hence there is only collision avoidance with stages and the gate.

The algorithm is simple : a stage has an influence zone which repulse agents that are in a determined radius. If an agent has a stage in his perception field, a flee target is set, to avoid the collision with the object. After the behaviours are computed, the flee target is added to the computed movement. The final movement of the agent is then adapted to the collision.

2.4.5 Bomb implantation

The bomb is considered as an obstacle, Agents will then flee away from it as soon as they see it. A timer is set in the GUI, which will be displayed on the screen. The WorldModel will take care of decreasing this timer, and once it arrives at 0, the WorldModel will hurt the Agents in range to be hurt and kill the ones closer to the bomb. Both ranges are defined as a propriety of the bomb class.

2.5 Encountered problems

2.5.1 AgentBody/AgentAddress

Find an body knowing its animat address, in laboratory work examples, was quite easy to realize thanks to a Map structure linking both classes. Since in our case, only SituatedObjects are stocked in the QuadTree structure, finding the body corresponding to an address without going through the entire QuadTree and test every of the bodies became a problem.

2.5.2 The toilet issue

Solution used to act on attendant needs from a foodstand was to delegate the task of applying need changes to the seller belonging to this stand. Indeed as an agent it was able to send a message to the attendant. In the case of toiletstand though, no agent was there to send the message to the agent allowing him to go elsewhere.

The only way we have think of to solve this issue was to create a unique invisible Agent to which is assigned a set of all toilets of the map and whose unique live behaviour is to check the queue of every toilet stand and send a message to each attendants on top of those queues.

2.5.3 Smelly Object

Our first intuition about how to guide agents to the object in adequation to their need was to use the concept of smelly objects. The principle is that interesting objects emits a smell decreasing function of the distance. People interested in finding one object will detect the object proper smell and go in direction of the strongest smell.

At last this technique involved too much complications, and according to attendant perceptions ranges and the numerous stands, agents are able to find their way to the appropriate stand thanks to their perceptions only.

2.6 Benchmark

2.6.1 QuadTree

So as to be certain of the structure reliability we run JUnit tests on the structure and its different iterators. That method was very helpful in a way that we could check at any new code inclusion if something has been damaged, especially if we consider that iterators use AgentBody's attributes which can potentially be modified during the project development.

In these tests, the QuadTree supports really well the insertion of thousands of entities (environment and agent) with no problem. It was tested with 100.000 of trees (as flora) without impact on performance (a few seconds). It is ok to have such results, because it is the initialization of the simulation, so it doesn't impact afterward.

The find function and the frustum culler have more weight on performance. Even if the data structure is a QuadTree, which gives a high gain in computation time, if an object to search for is far in the Tree, it can take a lot of time to get to it. It is the main impact on the simulation, because when the number of agents increases, the number of computation inside the Tree increases too. It isn't a problem from the data structure itself, but more from the algorithm of search. If some criterias and parameters are added inside the object of the Tree, it may improve the computation. In all cases, it is a problem of optimization.

2.6.2 Agents

In real life, there is approximately between 80.000 and 100.000 attendants a day at the festival. We can count also between 1.000 and 2.000 workers during the three days. So in theory, the simulation need to work with these numbers. Let's try with different number of people.

For these tests, workers are alone on their own stand. If we increase the numbers of workers, while staying at a ratio 1 worker for 50 attendants (same as reality), it doesn't influence very much (a few ms), because a worker has few computation to do.

Results of benchmark :

200 attendants $\rightarrow$ step computed between 52 and 80 ms
500 attendants $\rightarrow$ step computed between 171 and 246 ms
1000 attendants $\rightarrow$ step computed around 600 ms
5000 attendants $\rightarrow$ step computed around 11 and 13 s

3 User guide

3.1 Installation and use

3.1.1 Agents creation

The Main program allows the user to control most of the simulation parameters. From the spawning loop, the user chooses the number of agents involved for each type, the most important parameter for the simulation remaining the number of Attendants which deeply influences the performances :

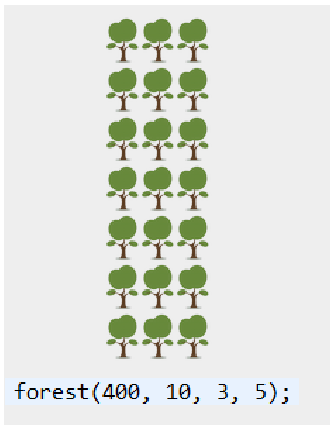
```
for(int i = 0; i < 100; i++) {
    FrameworkLauncher.launchAgent(new Attendant(AttendantGender.MAN));
    FrameworkLauncher.launchAgent(new Attendant(AttendantGender.WOMAN));
}
```

Apart from that, the user can decide to more specific agents as Stand Owners or Medics for a more realistic simulation, for example we usually spawn a seller for each stand as we limited their size in order not to overload the map with unnecessary occupied space.

```
for(Stand elem : standSet) {
    FrameworkLauncher.launchAgent(new Seller(elem));
}
```

3.1.2 Objects creation

The class WorldModel is at work here. It basically simulates the whole environment in the program. The method build() is where the code which creates and positions the objects on the map. Subfonctions are called inside the function build for each object category that allows the user to place a number n of objects at the coordinates x,y and arrange them in a rectangle of height h and width w.



```
forest(400, 10, 3, 5);
```

FIGURE 3.1 – Example of a forest

3.2 GUI control documentation

The GUI presents 2 controls, one to plant a bomb and one to exit the simulation. in order to plant a bomb in the festival, you just need to click somewhere on the map. A timer will then starts on the screen and people should start panicking if everything is done correctly.

Conclusion

Performances

With this project we can now understand the optimization structures, because in this project, slowness is the main issue. In a plateform which compute this much and contain a lot of entities to manage, it is essential to have an optimized plateform.

Critical analysis

We completed our main objectives. We have create an environment including stages, obstacles, an entry gate, stands and toilets, in which several types of agents can evolve. Each agent has a specific behaviour during the simulation. Additionally the interactions between agents and stands are working pretty well and if we let simulation running we can really observe crowd movements. Finally, when the user set the bomb, each agent reaction changes end everyone tries to reach the gate avoiding the bomb.

With a little more time we could have improved the bomb explosion, the role of bodyguard and medics, and the collision avoidance between agents and between agents and obstacles.