Simulating the Festival

Digital Urban Visualization. People as Flows.

12.10.2015

iA

zuend@arch.ethz.ch

treyer@arch.ethz.ch

Crowds at the Festival

image: zvv.ch

Crowds at the Festival

image: zvv.ch

Legende

- Bar
- Food & Getränke
- Food
- Waren / Info
- Kühlwagen
- Musik ab Tonträger
- mit DJ / Live
- Gas
- Fluchtwege 4 m
- Sanität
- Meeting Point
- Eingang
- WC/Pissoir
- Pressmulden
- Container

20 YEARS LATIN MUSIC FESTIVAL

CALIENTE!

3 - 5 JULIO 2015

WWW.CALIENTE.CH

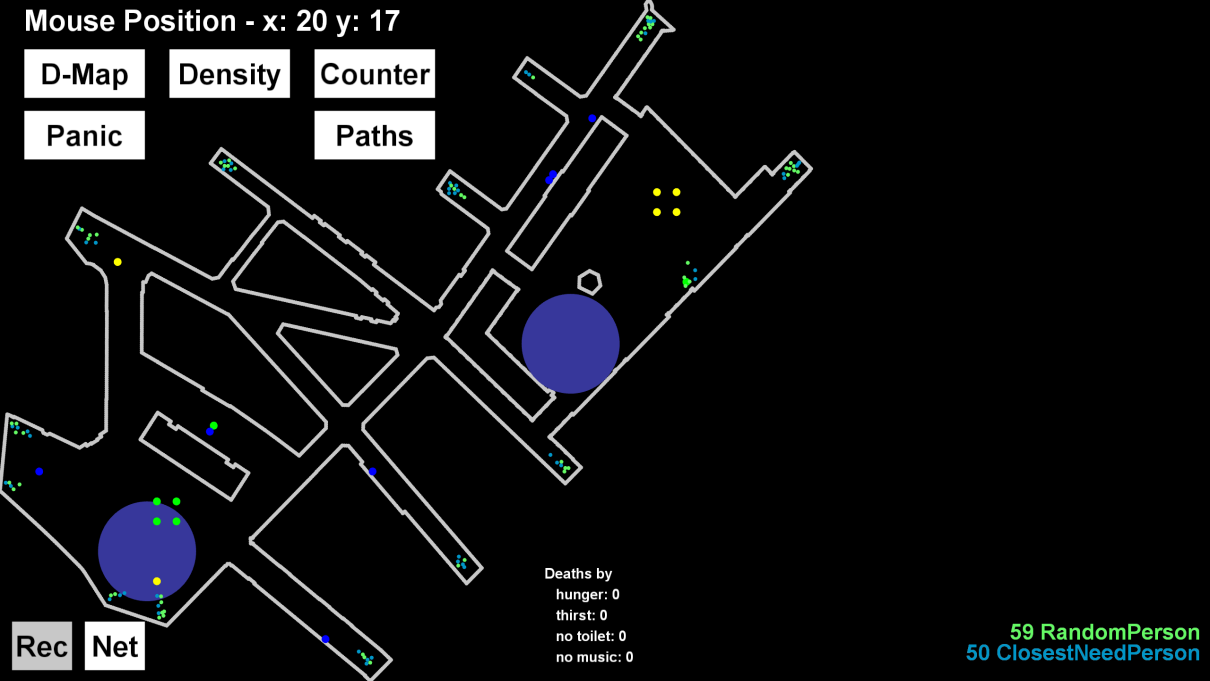
LANGSTRASSE

WC PISSOIR

VOLKSHAUS

Tische

Tische


Mouse Position - x: 20 y: 17
D-Map
Density
Counter
Panic
Paths
Deaths by
hunger: 0
thirst: 0
no toilet: 0
no music: 0
Rec
Net
59 RandomPerson
50 ClosestNeedPerson


Mouse Position - x: 20 y: 17
D-Map
Density
Counter
Panic
Paths
Deaths by
hunger: 0
thirst: 0
no toilet: 0
no music: 0
Rec
Net
934 RandomPerson
932 ClosestNeedPerson

Framework

what does it do?

The framework is able to simulate any kind of crowd simulations. We set it up to simulate the Caliente Festival in Zürich.

The simulation consists of different people types walking around. They have some needs to fulfill or otherwise they will die.

Framework

what does it do?

Currently, the people in the simulation have four needs, eating, drinking, going to the toilet, and listening to music.

These are provided by different types of need-providers: bars, food stalls, toilets, and stages.

During the simulation, people go to these locations, thus walk around.

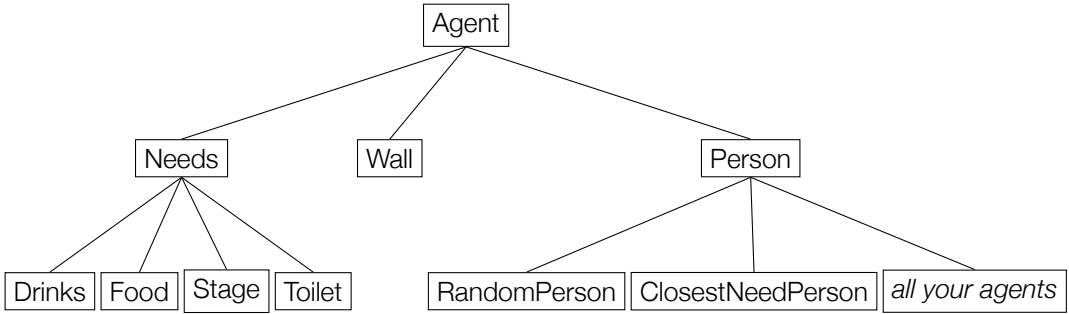
Types in the Framework

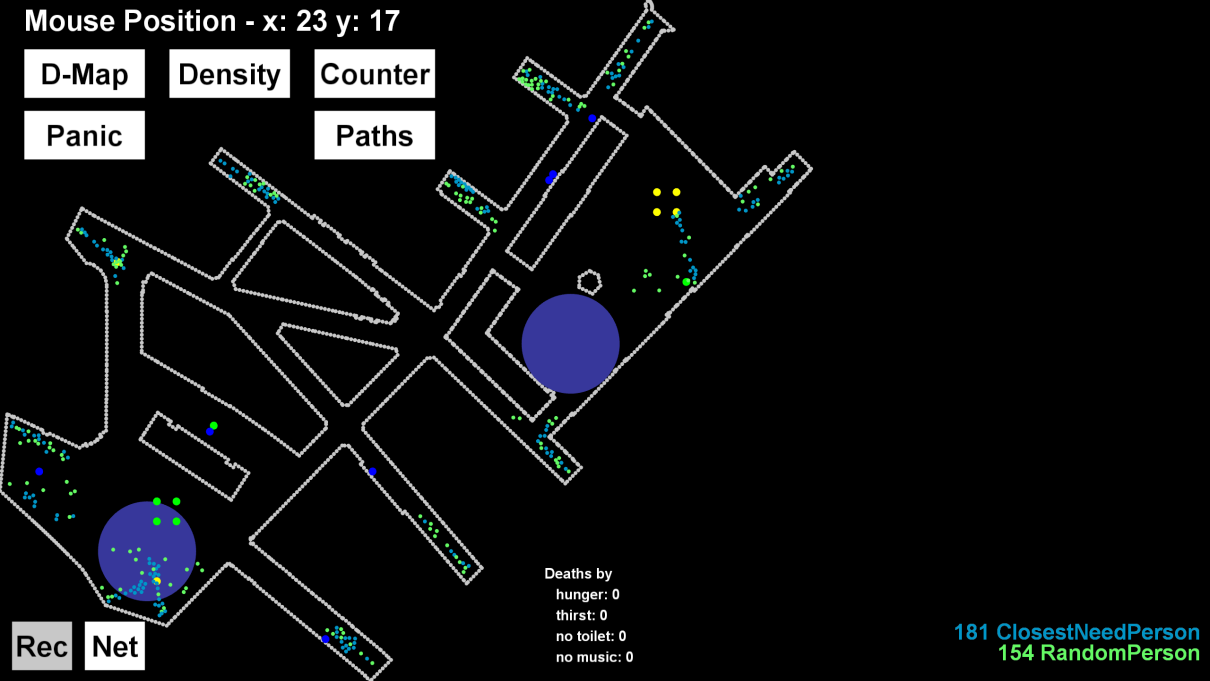
everything is agents

All actors in the simulation are so called agents. The following main types are all subclasses of the type *Agent*:

Needs: All the stalls in the simulation. Food stalls, bars, toilets, and stages inherit from this class.
Person: All the people in the simulation are a subtype of *Person*. They are the objects that move around.
Wall: The walls of the simulated area consists of agents of the class *Wall*.

Type Hierarchy




Mouse Position - x: 23 y: 17
D-Map
Density
Counter
Panic
Paths
Deaths by
hunger: 0
thirst: 0
no toilet: 0
no music: 0
Rec
Net
181 ClosestNeedPerson
154 RandomPerson

Types

agent type

This is the super-type of all actors in the simulation. It provides the basic interface of the minimal functionalities an actor has to provide.

It defines that an agent must have a **color** for when it is displayed and a **radius** of interaction with other agents. It also defines that all agents must have a **force field** and that it must be possible to get **their position**. Additionally an agent must have a function called **step** which changes the state of the agent for the next iteration.

Types

needs type

This is the super-type of all the different locations people can satisfy their needs.

In the current case, we have four different *Needs* types, food stalls, bars, toilets and stages.

They all have a certain range of interaction. This means that if an agent is close enough, he can fill up with, e.g., drinks.

Types

wall type

Their only job is to repel moving agents, to keep them in the area.

Types

person type

This is the super-type of all moving things, they have many additional state variables and also additional functions. For example they have to calculate the next location they want to go, as well the path to that location.

Objects of type *Person* are initialized with a certain value for the amount of thirst and hunger, and need for music and to go to the toilet. In every step they make, all the values worsen a little bit.

Where to go?

force fields

To have quasi-realistic behavior of the crowd, it is important that they cannot walk through each other. In our framework we use force fields to accomplish that.

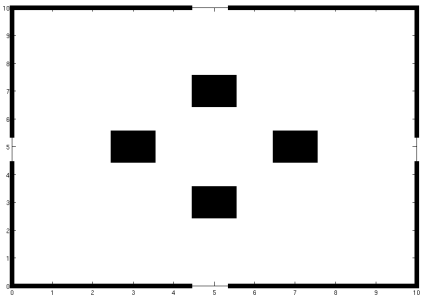
The force field is described by the distance of agents to each other, the closer they get, the more they are repelled from each other.

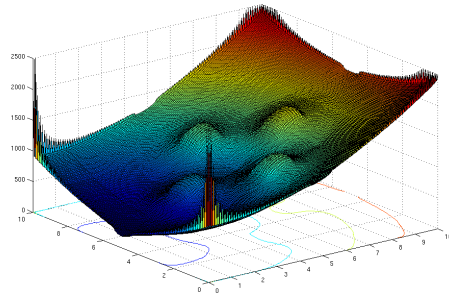
The opposite accounts for the location the agent decided to go to. He is attracted to that location.

Since we actually calculate the force field from potentials, the agent can be pictured as a ball rolling down a landscape.

Where to go?

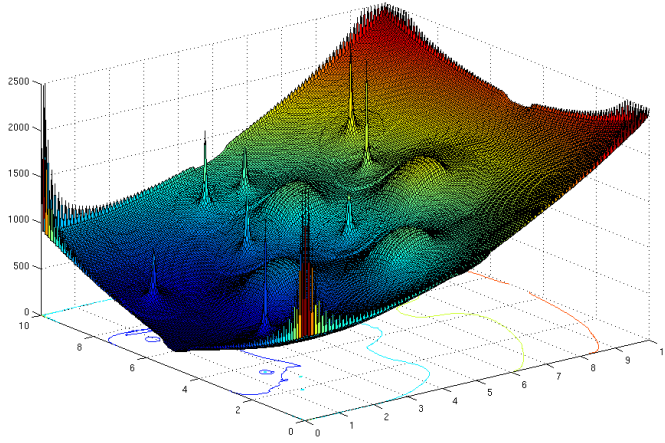
potential landscape





Where to go?

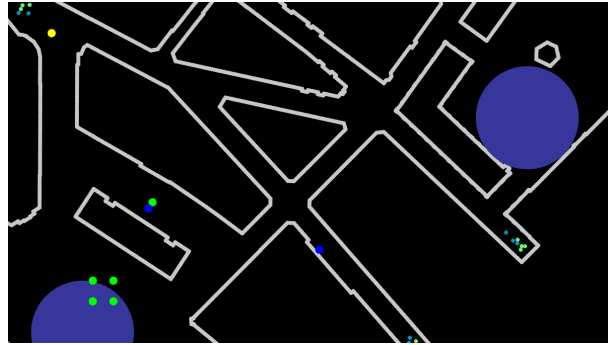
potential landscape



Where to go?

walking along a graph

If a person would take the direct path to the desired location, he/she might get stuck in certain situations.



Where to go?

walking along a graph

If a person would take the direct path to the desired location, he/she might get stuck in certain situations.

So we introduced a “walking”-network along which the people walk.

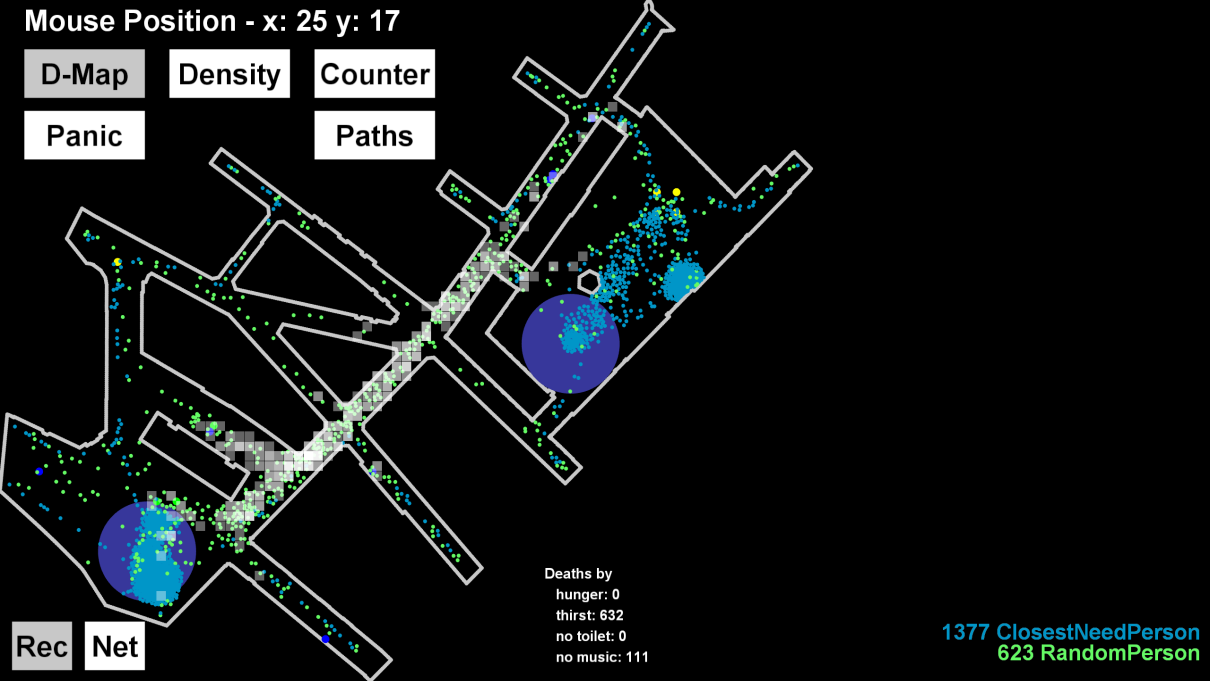
Interface

analysis helper

The interface has different helpers to analyse the current simulation. They are:

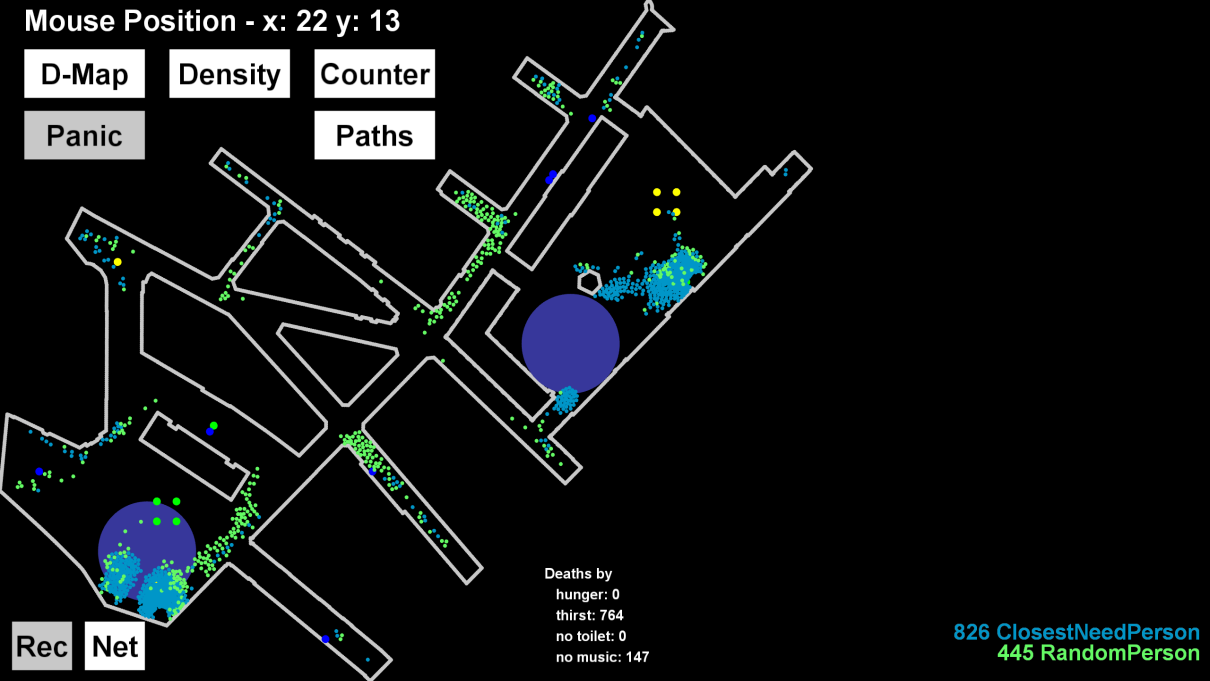
Density: draws a heatmap of the density per square meter, 5 and more people per m^2 is critical and full red.
D-Map: draws a heatmap of the number of people who died inside that square.
Counter: shows the number of happy customers per needs-provider.
Paths: additional draws the last 50 locations per person.
Panic: this button sets all people into panic mode and they try to exit at the closest exit as fast as possible.


Mouse Position - x: 23 y: 17
D-Map
Density
Counter
Panic
Paths
Deaths by
hunger: 0
thirst: 0
no toilet: 0
no music: 0
Rec
Net
1009 ClosestNeedPerson
991 RandomPerson


Mouse Position - x: 25 y: 17
D-Map
Density
Counter
Panic
Paths
Deaths by
hunger: 0
thirst: 632
no toilet: 0
no music: 111
Rec
Net
1377 ClosestNeedPerson
623 RandomPerson


Mouse Position - x: 16 y: 11
D-Map
Density
Counter
Panic
Paths
16
13
2063
1403
31
205
161
11
21
17
4411
3443
423
453
13
Deaths by
hunger: 0
thirst: 0
no toilet: 0
no music: 0
Rec
Net
1009 ClosestNeedPerson
991 RandomPerson


Mouse Position - x: 24 y: 20
D-Map
Density
Counter
Panic
Paths
Deaths by
hunger: 0
thirst: 0
no toilet: 0
no music: 0
Rec
Net
1009 ClosestNeedPerson
991 RandomPerson


Mouse Position - x: 22 y: 13
D-Map
Density
Counter
Panic
Paths
Deaths by
hunger: 0
thirst: 764
no toilet: 0
no music: 147
826 ClosestNeedPerson
445 RandomPerson
Rec
Net

Interface

other information

The other two buttons in the bottom left corner are:

Rec: stores an image of the simulation every 25 iterations into the frames folder of your project.
Net: shows the graph along which people walk to their desired location.

At the bottom of the canvas are some minimal statistics of how people died.

At the right hand side are the sub-types of *Person* listed that are active in the current simulation, together with the number of them.


Mouse Position - x: 20 y: 17
D-Map
Density
Counter
Panic
Paths
Deaths by
hunger: 0
thirst: 0
no toilet: 0
no music: 0
Rec
Net
934 RandomPerson
932 ClosestNeedPerson

Exercise

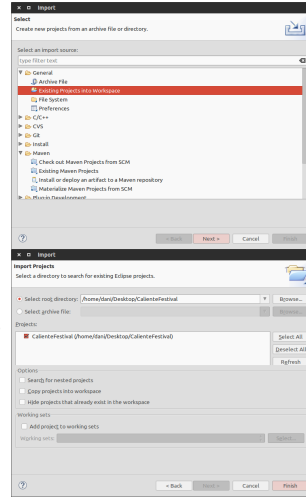
import the project

Download the zip file “calienteCrowds.zip” from the website and extract it at your desired location.

In Eclipse, go to *File* → *Import…*

Under *General* choose *Existing Projects into Workspace*.

Next to *Select root directory:* click *Browse* and select the *CalienteFestival* where you have extracted it. Click *Finish*. The project should be imported now.



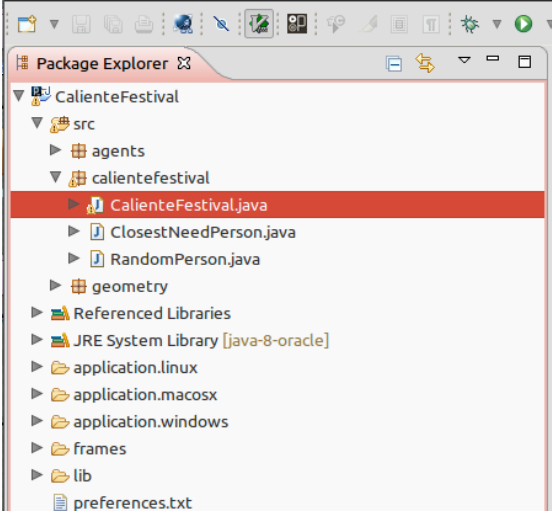
Exercise

start the simulation

To start the simulation, run the following file

in project *CalienteFestival* → *src* → *calientefestival* → *CalienteFestival.java*

with a right click and the *Run As* → *2 Java Application*.



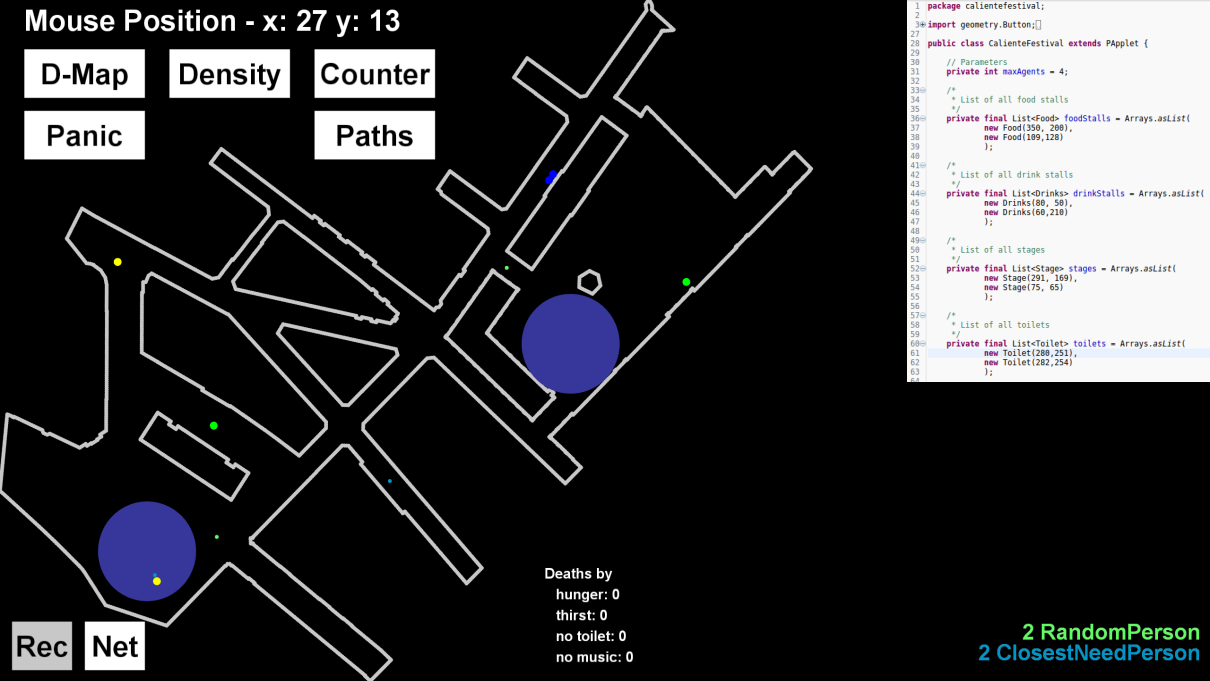
Exercise

basic setup and parameters

The basic parameters for the simulations are all within the first few lines of the *CalienteFestival.java* class.

The maximum number of agents can be set at line 31 with the private variable *maxAgents*.

The locations of the basic needs providers are defined right after this. They have to be defined as new objects of the specific type, having its coordinates as construction parameters.




```
package calientefestival;

import geometry.Button;

public class CalienteFestival extends PApplet {

    // Parameters
    private int maxAgents = 4;

    /*
     * List of all food stalls
     */
    private final List<Food> foodStalls = Arrays.asList(
            new Food(350, 200),
            new Food(109,128)
            );

    /*
     * List of all drink stalls
     */
    private final List<Drinks> drinkStalls = Arrays.asList(
            new Drinks(80, 50),
            new Drinks(60,210)
            );

    /*
     * List of all stages
     */
    private final List<Stage> stages = Arrays.asList(
            new Stage(291, 169),
            new Stage(75, 65)
            );

    /*
     * List of all toilets
     */
    private final List<Toilet> toilets = Arrays.asList(
            new Toilet(280,251),
            new Toilet(282,254)
            );
```

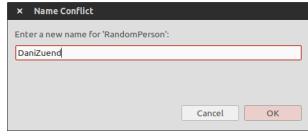
Exercise

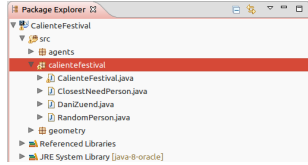
program your own agent

The easiest way to start with your own agent is to copy and paste for example the *RandomPerson.java* file into the same package and then rename it.

Right click on *RandomPerson.java* → click *Copy*
Right click on the package *calientefestival* → click *Paste*
When the dialogue pops up, put in your name, without whitespaces.

After clicking *OK* there should be a class with your name in the *calientefestival* package.





Exercise

program your own agent

To make your agent visually distinguishable from other agents, set up a unique color. This is done by opening your agent and set the *RGB* values for your agent in the *setColor* function.

```
@Override
protected void setColor() {
    this.r = 100;
    this.g = 250;
    this.b = 100;
}
```

To add the agent to the whole simulation, one last step has to be taken. You have to register the agent in the *CalienteFestival.java* class. Put the name of your class, e.g. *"DaniZuend"*, to the *classNames* list.

```
/*
 * List of all different person classes.
 */
private final List<String> classNames = Arrays.asList("ClosestNeedPerson", "RandomPerson");
```

→

```
/*
 * List of all different person classes.
 */
private final List<String> classNames = Arrays.asList("ClosestNeedPerson", "RandomPerson", "DaniZuend");
```


Mouse Position - x: 12 y: 174
D-Map
Density
Counter
Panic
Paths
Deaths by
hunger: 0
thirst: 0
no toilet: 0
no music: 0
Rec
Net
674 ClosestNeedPerson
666 RandomPerson
660 DaniZuend

Exercise

exercise 4 preview

In the exercise after the next, you will have to implement the location choice function for your agent.

We will then run all the different implementations together to see who programmed the most persistent behavior.

The winner will win a price!!

CalienteFestival.java | Agent.java | Person.java | Rando

```
1  package calientefestival;
2
3  import java.util.ArrayList;
8
9  public class DaniZuend extends Person {
10
11     public DaniZuend(Agent curr) {
12         super(curr);
13         this.setColor();
14     }
15
16
17     public DaniZuend(double x, double y) {
18         super(x,y);
19         this.setColor();
20     }
21
22     @Override
23     protected Needs locationChoice(ArrayList<Agent> others) {
24         Needs n;
25         ArrayList<Needs> needs = this.getNeeds(others);
26         n = needs.get(rGen.nextInt(needs.size()));
27         return n;
28     }
29
30
31     @Override
32     protected void setColor() {
33         this.r = 200;
34         this.g = 200;
35         this.b = 100;
36     }
37
38 }
39
```