Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Lecture with Computer Exercises: Modelling and Simulating Social Systems with MATLAB

Project Report

train jamming

Katja Briner & Marcel Marti & Thomas Meier

Zürich
16.12.2011

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of Originality

This sheet must be signed and enclosed with every piece of written work submitted at ETH.

I hereby declare that the written work I have submitted entitled

_____

is original work which I alone have authored and which is written in my own words.*

Author(s)

Last name	First name
_____	_____

Supervising lecturer

Last name	First name
_____	_____

With the signature I declare that I have been informed regarding normal academic citation rules and that I have read and understood the information on 'Citation etiquette' (http://www.ethz.ch/students/exams/plagiarism_s_en.pdf). The citation conventions usual to the discipline in question here have been respected.

The above written work may be tested electronically for plagiarism.

Place and date	Signature
_____	_____

*Co-authored work: The signatures of all authors are required. Each signature attests to the originality of the entire piece of written work in its final form.

Print form

Agreement for free-download

We hereby agree to make our source code of this project freely available for download from the web pages of the SOMS chair. Furthermore, we assure that all source code is written by ourselves and is not violating any copyright restrictions.

Katja Briner Marcel Marti Thomas Meier

Table of content

1 Abstract

This project's goal was to simulate the process of passengers exiting and boarding a train. The focus of interest was the influence of people's aggression level, i.e. how much they push others away in order to get straight to their target area.

Therefore we implemented a map depicting different types of areas, intraversable surfaces as walls, areas which slow passengers down as stairs and space where they can move freely. Onto this map we projected a variable number of layers representing different groups of people including their starting and target points.

Using a vector field calculated from several forces passengers were then moved according to their status (waiting, walking, arrived at target), and environments (walls, other passengers, target).

The time until all passengers have reached their target was measured. The aggression level was then varied and the difference in time clearly confirmed our expectations: the higher the level of aggression, the longer it takes for all passengers to reach their target. While some individuals decrease their walking time, others are slowed down considerately.

2 Individual contributions

Katja Briner: report, revision of source code
Thomas Meier: input map and layer files, testing, photos
Marcel Marti: adapting base project source code, additional source code, debugging

3 Introduction and Motivations

Every day thousands of people travel from their home to their working place and the same way back in the evening. A big part of these people do this by train, including the members of our group. During rush hours, train stations get crowded with people waiting for their train. Naturally, no one wants to travel a long distance without having a seat. Commuters cluster around the train's doors and most of them are polite and wait until the people inside have left the train before entering. However some impatient commuters start pushing into the train before everyone has left. Often, this behaviour leads to a jammed entrance area causing the time it takes for all people to get in or out to increase considerately. Since each member of our group comes across this situation almost every day, we wanted to investigate how the level of aggression affects the time needed until the train is ready to depart. We try to simulate a rush hour situation for the entrance area of an IC with different levels of aggression. The higher the aggression of an individual, the higher the chance it tries to enter the train too soon. The aggression level also influences how often a person tries to jump the queue. All mentioned parameters are united in the social force model, which we used in our simulation.

Fundamental Questions

Following questions are to be answered through simulations:
How much more time does it take until the train is ready to depart with different levels of aggression? Does it take longer at all? Is there a difference between people entering and leaving considering the change in time each individual needs to get in or out, when aggression level changes?

Following questions about the simulations are important for the feasibility of our results:
Is the social force model suitable for situations described in the introduction or should we apply changes to make it more reliable? Example: Do people wait in corners before attempting a second push into the seat direction? Does the model fail in very small rooms like an entrance area of a train? Can the model in our case result in a social deadlock, where no person can make any further move and the situation is stuck?
Can we design areas pushy persons can use to evade deadlocks?

Expected Results

By experience we can tell that pushy persons increase the amount of time depending on how many people try to push at the same time. This means one pushy person won't make that much of a difference in an Intercity entrance area whereas five pushy persons in the same entrance area cause a jam. In Intercity tilting trains the situation is even worse since there is not as much space available for evasion manoeuvres as in a normal two-floored Intercity train. We expect our model to reflect these experiences.
Also we expect situations without pushy persons or with a strategy to resolve faster than situations with pushy people.

Research Methods

Basing our work on the “Airplane Evacuation Simulation” by Philipp Heer and Lukas Bühler, we used continuous modelling by implementing the social force model with a defined time step **dt**. The simulation ends as soon as every passenger involved has reached his, target area. The running time of each simulation is measured and can later be used to compare different situations.

4 Description of the Model

Our goal was to implement an easy model of the entrance area of a train in which jamming situations can be simulated. Reworked plans of entrance areas can be imported into MATLAB. These plans are used to generate the simulation environment. Different elements on a map are identified by colours. Unlike in the airplane evacuation situations, we did not have the same target areas for every person on the map. Target and starting areas can overlap. To solve this problem, extra layers can be imported into MATLAB. Each layer represents one group of passengers with their start and target areas marked on the layer. Given this file structure, we get our main map with walls and special areas and an arbitrary number of group layers identifying the start and end areas for each group.
Given this input, MATLAB first generates the shortest path from any point to designated exit zones for each layer. Out of these results, MATLAB generates a vector field for each layer in the space of the train entrance area. Each passenger then follows the path given by these vector fields leading to his target.
The main simulation is then performed with an implementation of the social force model as described in [1].
Passengers tend to follow the shortest path to their target. They will not, however, always be able to follow the shortest path due to several factors affecting their direction. First, people cannot walk through walls. This will be implemented introducing a physical force radiant from the walls. Also, passengers cannot walk through each other unless they are ghosts, which we didn't take into account in this sheet. To prevent a situation where one passenger walks through another passenger, we introduce a second force pushing people away from each other. Finally there is the social force which reflects social interactions and, in our case, the behaviour of a passenger according to its aggression level.

5 Implementation

The simulation can be split into two parts. The first part initializes all variables and resources needed for the computation which is done in the second part. The first phase is packed into a file named “Setup.m” and is a MATLAB script which can be run without any arguments passed to it. When the setup script has finished executing, the simulation is ready for the second phase which is represented in a file named “Simulation.m”. This file again is a script. Execution will be cancelled if the setup script hasn’t been run yet. In order to facilitate the use of our simulation, “Setup.m” is directly called up first place in “Simulation.m”.

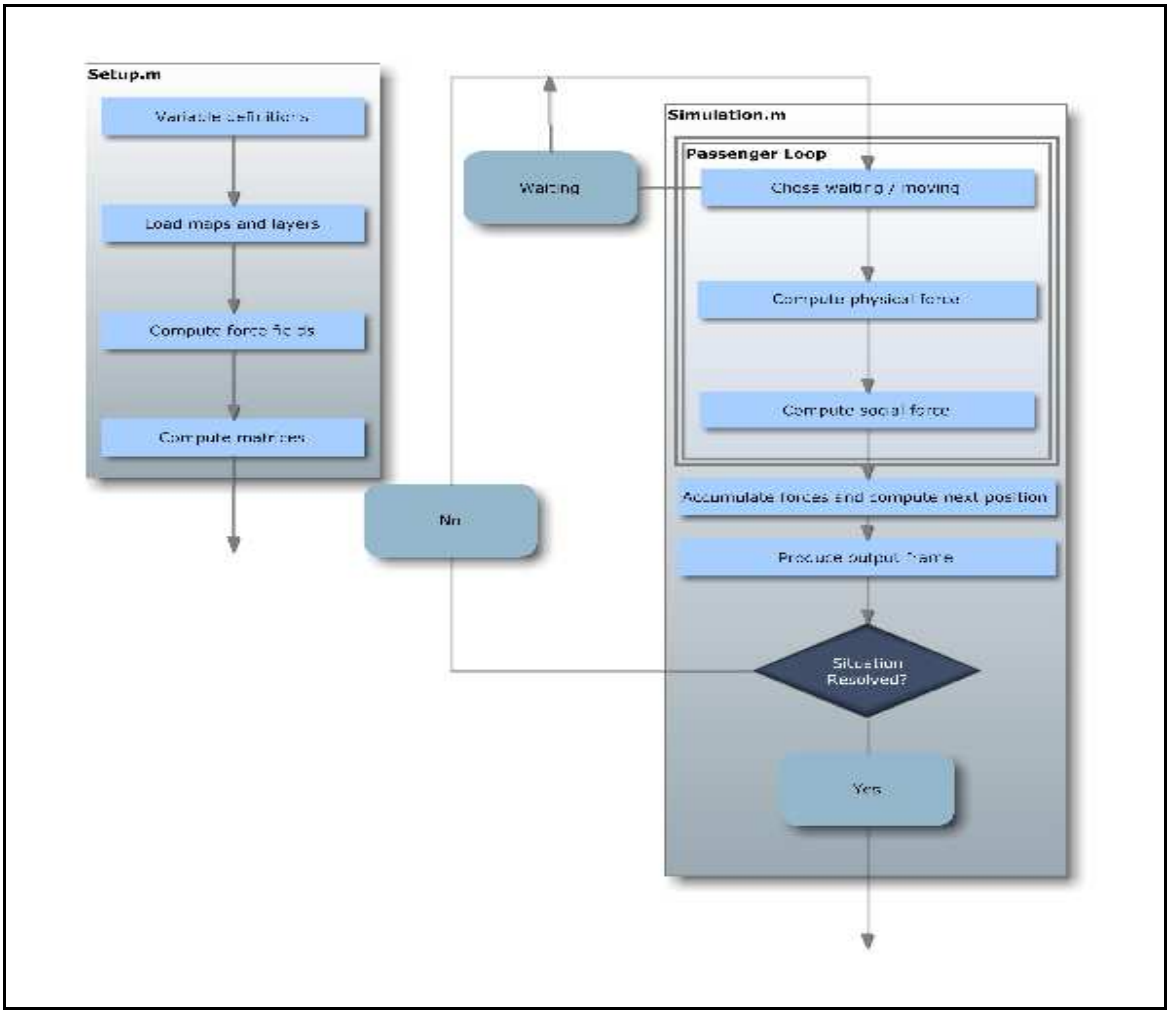


image [1]: simulation flowchart

Setup

The setup of a situation is done in multiple steps. First, the image of the area and the images of the paths are loaded and processed by loadSituation.m. It generates a three–dimensional matrix. In the computeVF.m file, this matrix is processed again. The shortest paths on every target layer are generated. Passengers will try to follow this field.

Loading a Situation

A situation consists of a map of a train and at least one layer. The map stores all walls and special areas, whereas the layers store the entry and exit points for different groups. The layers must have the same size as the map. A layer is not restricted to contain only one entry and one exit. It can contain arbitrary many entries and exist, but passengers will randomly spawn at an entry and then follow the shortest path to an exit available. If you wish to simulate a situation with fixed starting positions, use a different entry area for each passenger and set the spawn count to one. The spawn count is a variable indicating the number of passengers generated in a specific layer.

To keep the maps realistic, we asked the SBB for floor plans of both the Intercity train and the Intercity tilting train. They did not provide us with floor plans of the tilting train, but with a poorly resolved map of the Intercity train. We adapted this image in order to limit the content to the most important details. The primary use of the floor plan was to keep the dimensions realistic, what the plans were sufficiently accurate for. The adapted image could then be imported into MATLAB, which offers a function to convert an image file into a matrix with every entry representing a pixel of the input image.

To identify the different elements of maps and layers, a colour code is used. In the loadSituation.m file, every black pixel encodes a wall, every green pixel signalises a starting point and every red pixel represents an exit point. A yellow pixel indicates an area in which passengers have to slow down. In the Intercity example, the stair is coloured yellow. Pixels coloured white are interpreted as free space. The additional file findColor is used to find pixels of a specific colour.

If the input to loadSituation.m consists of one map and k layers, the output will be a matrix containing different numbers for different elements and a three dimensional matrix which consists of k two dimensional matrices representing the layers.

image [2]: Intercity train

image [3]: entrance area of an intercity train

Path Finding Algorithm

Every passenger tries to take the shortest way to an exit or target point. To guide passenger, we used a vector field. A fast marching algorithm is used to calculate the distance from every point on a map to the nearest target point. Thus the problem of finding the shortest path from every point on a map to the nearest exit is similar to a fluid spreading towards the drain. The file computeVF.m, based on P. Heer and L. Bühler's project source code, is adapted to our needs.
We decided to take the same toolbox for this kind of problem as did P. Heer and L. Bühler in their project our work is based on. The toolbox can be found on Mathworks:
http://www.mathworks.com/matlabcentral/fileexchange/6110
The provided fast marching algorithm generates a potential field in the area, which contains the distance of every point to the next target.
Starting from the potential field, gradientField.m computes a gradient field, base of the vector field.

Simulate

In the Simulate.m file, the new position of every passenger is calculated using basic physical laws. Forces are exerted on every passenger, depending on their current positions, resulting in an acceleration and thus movement.

Spawning Passengers

Our model includes a spawn system which makes passengers appear at specified spawn positions. This way, passengers entering the map from outside the area can be simulated. This system is applied to both the entering and the leaving group. The spawns check for every inactive passenger if there is enough space to let him spawn. If there is enough space, the passenger is spawned and taken into consideration in the following calculations.

Finished passengers

A passenger has reached its target if he gets near enough. As soon as this happens, the passenger is deactivated. He isn't visualized anymore and is ignored in any further calculation of forces.

Calculating forces

Wall force

The wall force deters passengers from moving through walls. This force has the highest influence on passengers since it will not only prevent them from doing movements which are physically impossible, but it will also prevent them from running off the map. Its influence range is very low tough. Passengers should not act as if they fear the walls, however they have to be kept from moving through fixed obstacles.

The wall force is an exponential force. The nearer a passenger gets to the wall, the stronger the force exerted on the passenger. Expressed in MATLAB-code the formula for the wall force exerted by one wall element is

WallForce = ForceStrength * exp(-Distance / ForceRange) * Normal

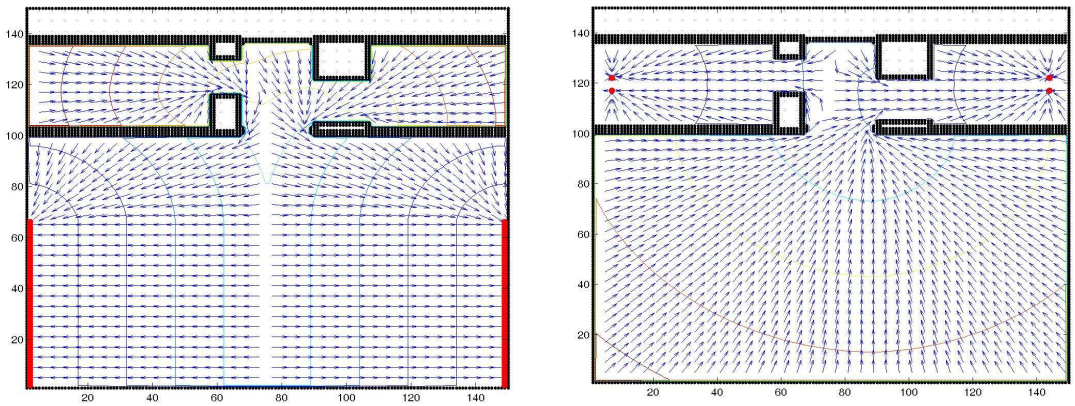
where “Normal” is the unit vector in normal direction from the wall.

Field force

The field force leads passengers onto their shortest path. It should be strong enough to make passengers appear to move directed towards a target.

The field force is never directed towards a wall. People tend to follow their shortest path, while the other forces can push them away from the path they are following and lead them onto another path leading them to another target.

What’s special about the field force is the aggression level coming into play here. The more aggressive a passenger is, the harder he will try to follow his shortest path and therefore push other passengers away.



image[4]: field forces

Rejection force

The rejection force is the physical force between passengers. Obviously passengers are not supposed to be able to walk through each other. However this force must not be too strong since passengers should not appear to be afraid of coming near to each other. It should be strong enough to push them in another direction and not into a passenger, but the interaction range of this force should be fairly low.

Also, aggression is taken into account in the rejection force. A very aggressive passenger pushes other passengers out of his way. In other words, the rejection force exerted on a less aggressive person is stronger than the force exerted on an aggressive person.

Social force

The social force defines the behavior of every passenger. Its range of interaction is quite large. Unlike every other force, the social force is not isotropic. It is however axis–symmetric to the axis given by the movement vector. Another factor, lambda, is taken into account. Lambda weights the angle between the movement vector and the vector between two passengers. Increasing lambda will result in a passenger reacting stronger on intrusions in front of him. Analogical to the wall and rejection force, the social force increases when two passengers move nearer to each other.

The formula for the social force between two passengers expressed in MATLAB–code is:

SocialForce = ((Lambda + (1 – Lambda)*(1 + cos(Phi))/2) * ForceStrength …

***exp(– Distance / ForceRange) * Direction;**

Calculating positions

The new position of every passenger is calculated using the physical law of motion. The acceleration resulting from the total force is calculated through division by the weight of the concerned passenger. Then the acceleration vector is multiplied with the time step and added to the current position.

6 Simulation Results and Discussion

Tested situations

We tested three different situations to compare the amount of time needed in every situation.

1. one group of 40 passengers tries to enter the train while another group of 40 passengers tries to exit it at the same time.
2. one group of 40 passengers tries to exit the train while two very aggressive passengers try to enter it.
3. one group of 40 passengers tries to exit the train while eight very aggressive passengers try to enter it.
4. one group of 40 passengers tries to exit the train without a group trying to enter the train at the same time.

Situation one is the extreme case of every passenger being an egoist. We expected massive jamming and a very high amount of time needed to resolve the situation.

Situation two is the case when many passengers wait next to the doors to let people exit the train, nevertheless two aggressive passengers already try to enter.

Situation three is a situation where a group of younger people tries to get seats before anyone else.

Situation four is the optimal case of all the people waiting outside the train, letting people exit without attempting to enter the train at the same time.

The map

The input map is based on a plan received from the SBB customer support. The original map was extremely blurry. We reworked it and extracted the dimensions of an Intercity train. We did not model the seats. The part which interests us is the entrance area including the stairs leading down from the top floor, the entrance door, and the passage to the lower floor seats. Behind those key positions puffer zones were added in order to give spawning passengers enough space. Doing otherwise, the jamming situations would end in a deadlock, where no passenger can make any progress, which is neither realistic nor producing usable results.

The factors

The biggest challenge was finding values for every factor such that they made sense.

Empirically we decided that the following values resulted in the most realistic situations:

Factor	**Value**
Lambda	0.65
Exit Radius	1.5
Rejection force influence area	1
Wall force influence area	2
Social force influence area	6

Situation 1

As we expected the situation ended in massive jamming. It took quite a long time until a passenger could leave or enter the train. Most interesting, once a passenger managed to escape the situation, other passengers with the same target followed. This way three paths appeared during the simulation. The middle path was used by passengers entering the train, while the outer two paths were used by passengers trying to leave the train.

One passenger leaving the train seemed to be more aggressive than others, since he started pushing away less aggressive people. This passenger came from the left side inside the train and pushed out of the train along the wall. It was this passenger which created the first canal. The actions of this passenger resulted in a chain reaction which is visualized in the following image.

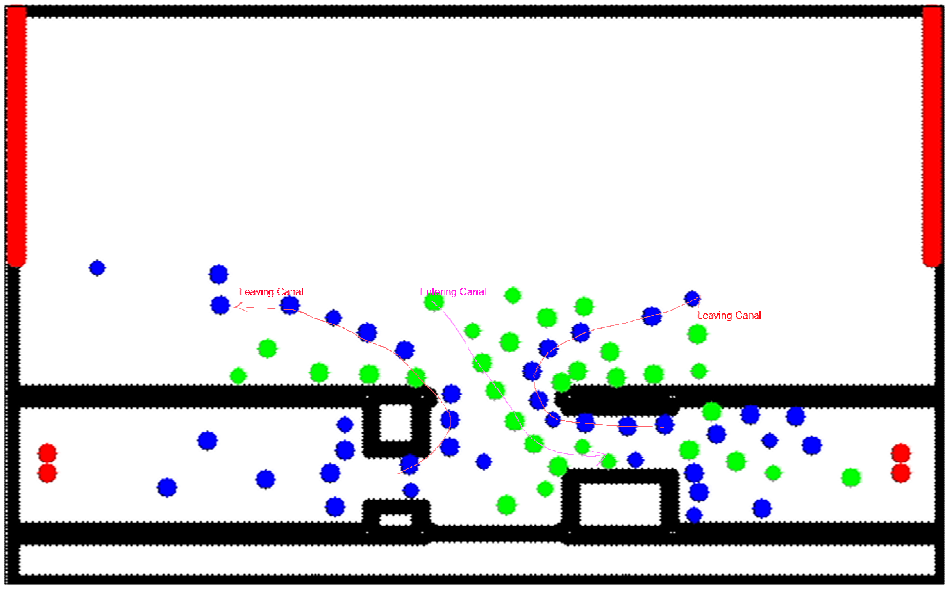


image [5]: Situation 1

By pushing away every entering passenger, the right side area of the train became much more attracting for entering passengers, since they were pushed nearer to it. This resulted in very few entering passengers trying to enter the train on the left side. We had the simulation run this situation for 4000 frames (dt = 0.1, T = 400) and the situation did not resolve completely. However the situation could be called resolved after 3800 frames. The four passengers left in the entrance area of the train did not make a serious attempt to reach their target. The forces applied to them just let them tremble in their positions which we did not intend, but randomly happened. We call this situation a live-lock. Every passenger on its own still makes some progress, but the situation over all does not.

Situation 2

This situation ended in a quick resolve. After 1050 calculated frames, every passenger wanting to leave the train had left. As expected, the aggressive two passengers pushed away every other passenger and got to their target exits very quick. The big group of leaving passengers had to let them pass and was slowed down a little, but when the aggressive guys disappeared, they could leave the train very fast.

Situation 3

This situation is a nearly perfect example for the effects of the social force and the aggression factor.

The aggressive passengers entering the train run very fast and again push away all the passengers leaving the train. The social force results in them following each other and thus reaching their target very fast. The last of them was cut off and then did not have a chance to get into the train before most of the leaving passengers were out of the train. He was simply pushed away by the mass of leaving passengers.

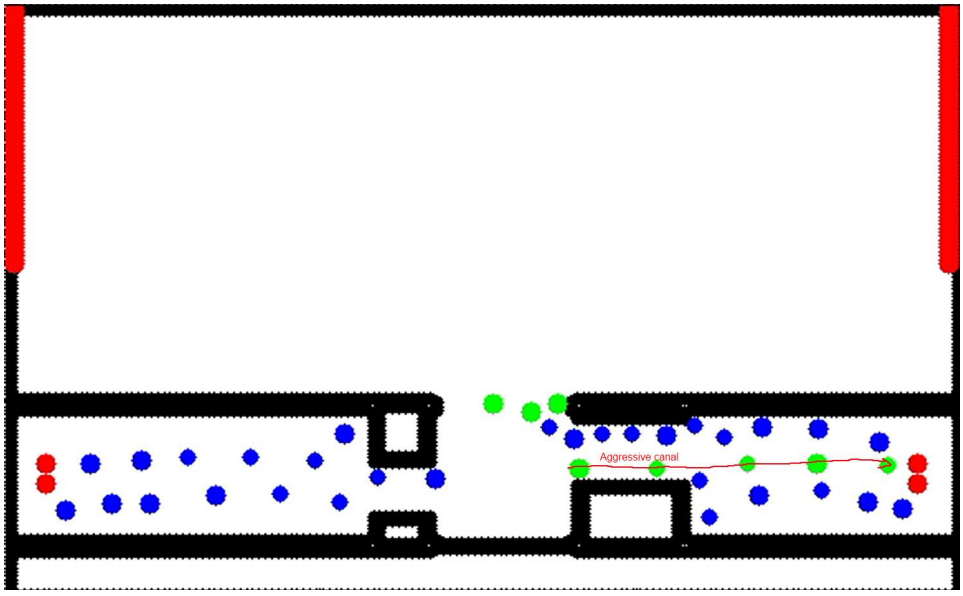


image [6]: situation 3

Situation 4

As expected, this situation resolved the fastest. The leaving passengers did not have any obstacles except for their mates and walls.

Result conclusion

Situation	Frames needed
Situation 1	4000+
Situation 2	1050
Situation 3	1200
Situation 4	970

Comparison with real life

The simulation went as expected. Overall the results matched our experience. However, there were some major issues which we did not foresee. Passengers in our simulation did not behave very intelligent. They just acted according to the defined rules. There was one situation, where the majority of entering passengers was stuck in the entrance area, trying to get to the right target while the path to the left target was completely free.

If there are just some small groups of passengers which try to enter the train while others are exiting, the conflicts resolve very fast. The situation would be different, if the entering people were not aggressive. They would be pushed out of the way by the pure mass of passengers leaving the train.

7 Summary and Outlook

The fundamental question of this project, whether higher aggression leads to prolonged time the train has to wait for all passengers to reach their target area, has been answered: Some individuals manage to reduce their own walking time by pushing others away, while other passengers are slowed down by these individual's behaviour.

Individual social force

A possible extension could be to use different social forces for every group of passengers. That way, passengers could react more aggressive on other groups. There would be much more possible outcomes of a simulation by customizing the social force for every passenger.

Model

The simulation showed that the model fits well for this kind of situations. However the social force as we implemented it suboptimal. Entering passengers and leaving passengers tried to have a certain safety area in front of them which resulted in situations where nearly no to no progress was made.

Conclusion

We focused our simulation on the Intercity entrance area. Since we could not get plans for an Intercity tilting train, we could not test the outcome with specially assigned exits and entries. However our simulation showed that the difference in time needed between the situations extremely depends on the aggression of one of the groups. If one group is clearly more aggressive, it will find its way through the crowd fast. If both groups are at the same level of aggression, paths can't get stable and people tend to push themselves and their partners for– and backwards without making that much progress.
Our conclusion is that pushing into the train early does have influence on the whole crowd. Also our model did not take into account that people tend to get annoyed. You won't make friends with pushing too early and aggressive.

8 References

[1]: “Pedestrian Dynamics Airplane Evacuation Simulation“ by Philipp Heer and Lukas Bühler, May 2011

image [1]: Marcel Marti 2011
image [2]: Thomas Meier 2011
image [3]: Thomas Meier 2011
image [4]: Marcel Marti 2011
image [5]: Marcel Marti 2011
image [6]: Marcel Marti 2011
image [7]: Katja Briner 2011

9 Appendix: source code

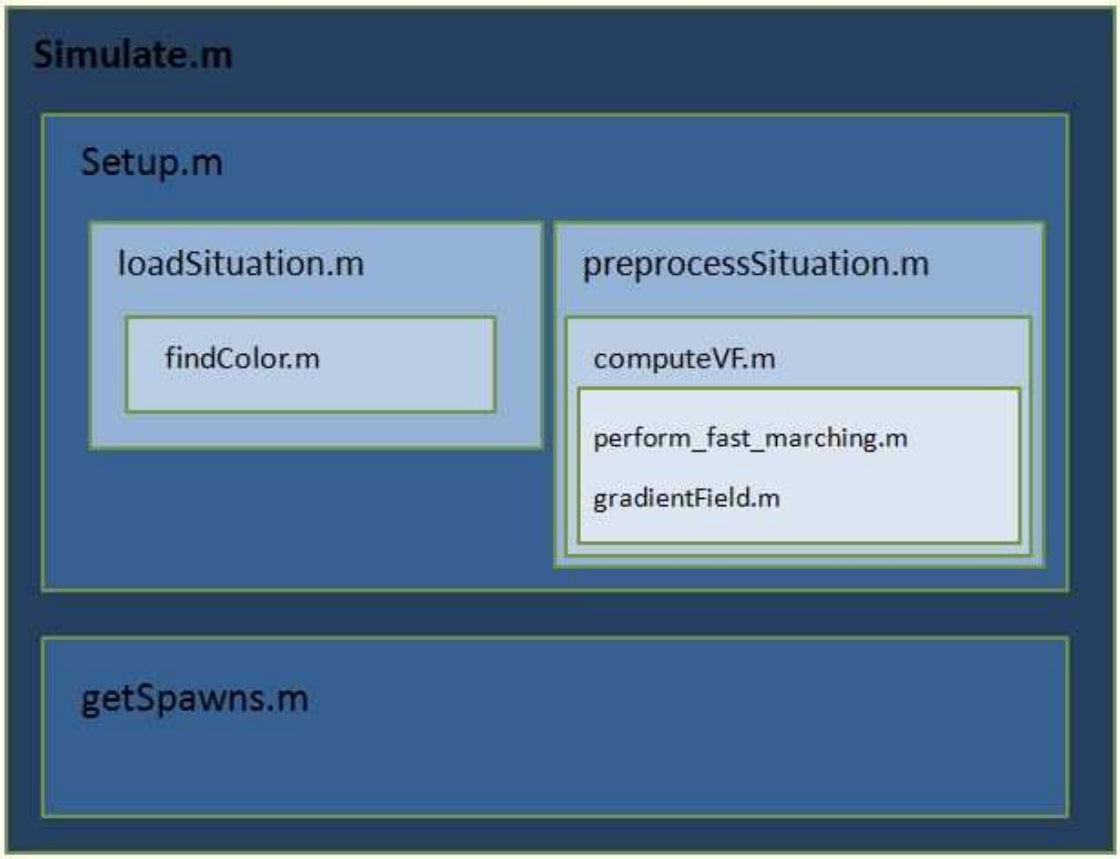


image [7]: structure of the source code files

```
%   The acutal simulation of the train entrance.
%
%   First runs the setup script which loads the data needed by the
%   simulation to run, then runs the simulation for a given time.
%
%   Variables needing to be defined:
%
%   Variable                |   Description
%   ------------------------------------------------------------------------
%   Map                     |   Map, containing information about walls and
%                           |   special zones.
%   Layers                  |   Contains the initial information of the k
%                           |   layers loaded in the setup phase.
%   Vectorfields            |   Contains the k vectorfields used to lead
%                           |   passengers onto their shortest path.
%   m                       |   The height of the map.
%   n                       |   The width of the map.
%   Lambda                  |   This scalar factor influences how much the
%                           |   angle between the movement vector of a
%                           |   passenger a and the normalized vector between
%                           |   a and another passenger b is weighted.
%   nGroups                 |   The number of groups.
%   T                       |   The duration of the simulation.
%   dt                      |   The timestep in the simulation.
%   pInfArea                |   The physical influence area for passengers.
%   wInfArea                |   The wall influence area.
%   sInfArea                |   The social influence area for passengers.
%   nPassengers             |   The number of passengers in each group.
%   fField                  |   The factor applied to the field force.
%   fForceStretch           |   The factor applied to the total force.
%   SpawnSecurityFactor     |   The additional radius for the spawn area.
%   nExits                  |   The number of exits.
%   nSpawns                 |   The number of spawn points.
%   nTotalPassengers        |   The total number of passengers.
%   eps                     |   The predefined epsilon value.
%   Passengers              |   Structure providing information about every
%                           |   passenger. The detailed structure is
%                           |   explained in Setup.m.
%   Groups                  |   Structure providing information about every
%                           |   group.
%   Walls                   |   Structure providing information about every
%                           |   wall element.

run Setup;

close all;

movieCount = 1;
Movie = avifile(['Output' num2str(movieCount) '.avi'], 'compression', 'None');
gcf;
set(gcf, 'visible', 'off', 'units', 'normalized', 'outerposition', [0 0 1 1]);

%   Measure time in full steps
time = 0;
```

```
%   Run simulation for the specified time with a specified frequency.
for t = 1:dt:T,

    timeold = time;
    time = int16(t);
    if timeold ~= time,
        disp([num2str(time) ' of ' num2str(T)]);
    end

    for pNo = 1:nTotalPassengers,
        %   Check, if the passenger has started.
        if Passengers(pNo).Started == 0,
            %   Check if there is a free slot for him to start.
            Spawns = getSpawns(Passengers, Groups, Walls);
            nStarts = length(Spawns(Passengers(pNo).Group).Starts);
            for sNo = 1:nStarts,
                Radius = Passengers(pNo).Radius;
                Start = Spawns(Passengers(pNo).Group).Starts(sNo);
                if Spawns(Passengers(pNo).Group).Starts(sNo) > Passengers(pNo).Radius↙
+ spawnSecurityFactor,
                    %   The passenger can start at this position.
                    Passengers(pNo).Position = Groups(Passengers(pNo).Group).Starts↙
(sNo).Position;
                    Passengers(pNo).OldPosition = Passengers(pNo).Position +↙
[(unidrnd(2*1e3)-1e3)/1e6; (unidrnd(2*1e3)-1e3)/1e6];
                    Passengers(pNo).Started = 1;
                    break;
                end
            end

            clear nStarts Spawns sNo;
        end
    end

    %   Calculate forces for every passenger.
    for pNo = 1:nTotalPassengers,

        %   Check, if the passenger is moving. If not, stop here.
        if Passengers(pNo).Started == 0 || Passengers(pNo).Finished == 1,
            continue;
        end

        %   Check if the passenger has finished.
        Group   = Passengers(pNo).Group;
        Ends    = [Groups(Group).Ends.Position];
        nEnds   = length(Ends(1,:));
        for i = 1:nEnds,
            Direction   = (Passengers(pNo).Position - Ends(:,i));
            Distance    = norm(Direction);
            Direction   = Direction./Distance;

            if Distance < ExitRadius + Passengers(pNo).Radius,
                Passengers(pNo).Finished = 1;
            end
        end
```

```
        %   Now re-check, if the passenger is moving. If not, stop here.
        if Passengers(pNo).Started == 0 || Passengers(pNo).Finished == 1,
            continue;
        end

        %   From this point on, we can assume the passenger has started and
        %   not finished.

        %*****************************************************************%
        %                          FORCES                                 %
        %*****************************************************************%

        %   First, reset forces.
        Passengers(pNo).WallForce    = [0; 0];
        Passengers(pNo).SocialForce  = [0; 0];
        Passengers(pNo).FieldForce   = [0; 0];
        Passengers(pNo).RejectForce  = [0; 0];

        %   1.  Vectorfield force
        %
        %   This force is used to make the passenger follow his shortest
        %   path. How much the passenger will be pushed towards his
        %   shortest path depends on his aggression level.
        Position = Passengers(pNo).Position;
        row = int16(Position(2));
        col = int16(Position(1));

        if row <= 0 || col <= 0 || row > m || col > n,
            Movie = close(Movie);
            error('Passenger walked to an invalid position.');
        end

        Field = Vectorfields{Passengers(pNo).Group};
        Passengers(pNo).FieldForce = [Field(row, col, 1); Field(row, col, 2)] *↙
(Passengers(pNo).Aggression + fField);

        clear Position Field;

        %   2.  Wall force.
        %
        %   The Wall force is exerted to a passenger by a wall
        %   element in his range. The range is determined by 'wInfArea'.
        %   The nearer the passenger gets to the wall, the stronger the
        %   force.
        for wNo = 1:nWalls,
            Distance = norm(Passengers(pNo).Position - Walls(wNo).Position);
            %   Check if the passenger is within the influence area of the
            %   wall element.
            if Distance < Passengers(pNo).Radius + wInfArea,
                WallIntStrength = Passengers(pNo).Interactionstrength.Wall;
                WallIntRange    = Passengers(pNo).Interactionrange.Wall;

                Direction = Passengers(pNo).Position - Walls(wNo).Position;
```

```
                Direction = Direction./Distance;

                ForceStrength = WallIntStrength * exp(-Distance/WallIntRange);

                %   Check passenger position.
                if  Passengers(pNo).Position(1) > Walls(wNo).Position(1) - 0.5 &&...
                    Passengers(pNo).Position(1) < Walls(wNo).Position(1) + 0.5 ,
                    yPos = sign(Direction(2));
                    %   1  -> Passenger is below wall.
                    %   -1 -> Passenger is above wall.
                    Passengers(pNo).WallForce = Passengers(pNo).WallForce +↙
ForceStrength * [0; yPos];
                elseif Passengers(pNo).Position(2) > Walls(wNo).Position(2) - 0.5↙
&&...
                       Passengers(pNo).Position(2) < Walls(wNo).Position(2) + 0.5,
                   xPos = sign(Direction(1));
                   %    1  -> Passenger is on the right side of the wall.
                   %    -1 -> Passenger is on the left side of the wall.
                   Passengers(pNo).WallForce = Passengers(pNo).WallForce +↙
ForceStrength * [xPos; 0];
                end
            end
        end
        clear Direction Distance ForceStrength WallIntStrength WallIntRange xPos↙
yPos;

        %   3.  Passenger physical force.
        %
        %   The passenger physical force prevents
        %   passengers from running through each other. The range is
        %   determined by 'pInfArea'. The nearer passengers get to each
        %   other, the stonger the force.
        %   Additionally, aggressive passengers push less aggressive
        %   passengers away far stronger.
        %
        %   4.  Passenger social force.
        %
        %   The passenger social force influences the
        %   behaviour of a passenger.
        for opNo = 1:nTotalPassengers,
            %   No interaction should be calculated between a passenger and
            %   himself and whith inactive passengers.
            if opNo == pNo || Passengers(opNo).Started == 0 || Passengers(opNo).↙
Finished == 1, continue; end

            Distance = norm(Passengers(pNo).Position - Passengers(opNo).Position);
            if Distance < pInfArea + Passengers(pNo).Radius + Passengers(opNo).↙
Radius,

                %   Calculate the physical force.
                RadiusA = Passengers(pNo).Radius;
                RadiusB = Passengers(opNo).Radius;
                AggressionA = Passengers(pNo).Aggression;
                AggressionB = Passengers(opNo).Aggression;
                AggressionSummand = AggressionB - AggressionA;
```

```
                if AggressionSummand < 0, AggressionSummand = 0; end
                ForceStrength = Passengers(pNo).Interactionstrength.Physical;
                ForceRange = Passengers(pNo).Interactionrange.Physical;
                Direction = (Passengers(pNo).Position - Passengers(opNo).Position)./Distance;

                Passengers(pNo).RejectForce = Passengers(pNo).RejectForce...
                    + (AggressionSummand + ForceStrength)...
                    *exp((RadiusA + RadiusB - Distance)/ForceRange) * Direction;
                %   Clear trash.
                clear RadiusA RadiusB AggressionB ForceStrength ForceRange Direction;
            end


            if Distance < sInfArea + Passengers(pNo).Radius + Passengers(opNo).Radius,

                %   Calculate the social force
                Direction = (Passengers(pNo).Position - Passengers(opNo).Position)./Distance;
                Move = (Passengers(pNo).Position - Passengers(pNo).OldPosition);
                MoveNorm = norm(Move);
                Move = Move./MoveNorm;
                Phi = acos(dot(Direction, Move));

                Passengers(pNo).SocialForce = Passengers(pNo).SocialForce...
                    + (Lambda + (1 - Lambda)*(1 + cos(Phi))/2)...
                    *Passengers(pNo).Interactionstrength.Social...
                    *exp(1 - Distance/Passengers(pNo).Interactionrange.Social)*Direction;

                %   Clear trash.
                clear Direction Move MoveNorm Move Phi;
            end
        end

        %   Sum up forces
        SocialForce         =   Passengers(pNo).SocialForce;
        WallForce           =   Passengers(pNo).WallForce;
        RejectForce         =   Passengers(pNo).RejectForce;
        FieldForce          =   Passengers(pNo).FieldForce;
        Passengers(pNo).TotalForce          =   WallForce + FieldForce + SocialForce + RejectForce;
        Passengers(pNo).TotalForce          =   Passengers(pNo).TotalForce.*fForceStretch;

        %   Clear trash.
        clear SocialForce WallForce RejectForce FieldForce Direction Distance nEnds nEnds Group;
    end

    %   Calculate new Position
    for pNo = 1:nTotalPassengers,
         %   If the passenger has not started or finished, stop here.
        if Passengers(pNo).Started == 0 || Passengers(pNo).Finished == 1,
```

```
            continue;
        end

        %   Else, calculate new Position.

        %   F = m*a -> a = F/m
        Weight              =   Passengers(pNo).Weight;
        Acceleration        =   Passengers(pNo).TotalForce/Weight;

        %   Store old position.
        Passengers(pNo).OldPosition     =   Passengers(pNo).Position;
        OldPosition                     =   Passengers(pNo).OldPosition;

        %   Calculate new position.
        Passengers(pNo).Position        =   dt*Acceleration + OldPosition;
        Position                        =   Passengers(pNo).Position;
    end

    %   Plot the whole situation.
    %   Plot walls.
    WallPositions       = [Walls.Position];
    plot(WallPositions(1, :), m - WallPositions(2, :) + 1, '.k', 'MarkerSize', 20);

    %   Plot Exits.
    ExitPositions       = zeros(2, nExits);
    MatrixPosition      = 1;
    for i = 1:nGroups,
        Ends = [Groups(i).Ends.Position];
        nEnds = length(Ends(1,:));
        ExitPositions(:,MatrixPosition:MatrixPosition+nEnds-1) = Ends;
        MatrixPosition = MatrixPosition + nEnds;
    end
    hold on;
    plot(ExitPositions(1, :), m - ExitPositions(2, :) + 1, '.r', 'MarkerSize', 30);

    %   Plot Passengers
    for i = 1:nTotalPassengers,
        Started     = Passengers(i).Started;
        Finished    = Passengers(i).Finished;
        if Started == 1 && Finished == 0,
            if CustomMarkers,
                plot(Passengers(i).Position(1), m- Passengers(i).Position(2) + 1,↙
Groups(Passengers(i).Group).Marker, 'MarkerSize', 20 + 3*Passengers(i).Radius);
            else
                plot(Passengers(i).Position(1), m - Passengers(i).Position(2) + 1, '.↙
bl', 'MarkerSize', 20 + 3*Passengers(i).Radius);
            end
        end
    end
    xlim([0 n+1]);
    ylim([0 m+1]);
    title(num2str(t));

    %   Add Frame to movie.
    Movie = addframe(Movie, gcf);
```

```
    %   Clear trash.
    clear WallPositions ExitPositions MatrixPosition i Ends nEnds Spawns nStarts↙
SpawnPositions StartedMatrix FinishedMatrix Started Finished;
    if sum([Passengers.Finished]) == nTotalPassengers,
        %   Add another 25 frames without any passenger moving to fade the
        %   result out.
        for i = 1:25,
            Movie = addframe(Movie, gcf);
        end
        break;
    end

    %   Clear figure;
    clf;

    %   Start new movie if the first one gets too big.
    if mod(t, 201) == 0,
        Movie = close(Movie);
        movieCount = movieCount + 1;
        Movie = avifile(['Output' num2str(movieCount) '.avi'], 'compression',↙
'None');
    end
end

Movie = close(Movie);
clear all;
```

```
%Modelling and Simulating Social Systems
%Project: Pedestrian Dynamics,
%Train Jamming by Marcel Marti, Thomas Meier, Katja Briner
%
%file Setup.m

%*************************************************************************%
%   About the Positioning problem:                                        %
%                                                                         %
%   First, the y axis start at the top with index 1 and is directed       %
%   downwards with increasing inidces.                                    %
%                                                                         %
%   y                                                                     %
%   |                                                                     %
%   |                                                                     %
%   |                                                                     %
%   |                                                                     %
%   v                                                                     %
%   ---------------------------------------> x                           %
%                                                                         %
%   This results in the following problems:                               %
%                                                                         %
%   1.  Map Matrices must be accessed like this: Map(y, x).               %
%   2.  Positions must be accessed like this Position(1) = x,             %
%       Position(2) = y.                                                  %
%*************************************************************************%

clear all;
clc

%   Load a situation
[Map, Layers]       =   loadSituation;
Vectorfields        =   preprocessSituation(Map, Layers);
[m, n, nGroups]     =   size(Layers);

%   Define variables needed for the simulation.
T                   =   20;
dt                  =   0.1;
Lambda              =   0.65;
ExitRadius          =   1.5;
pInfArea            =   1;
sInfArea            =   10;
wInfArea            =   2;
fField              =   10;
fForceStretch       =   5;
spawnSecurityFactor =   1.5;


%*************************************************************************%
%   Default Values:                                                       %
%                                                                         %
%   Change these values to adjust default behaviour.                      %
%*************************************************************************%
Defaults.Interactionstrength.Physical   =   20;
```

```
Defaults.Interactionstrength.Social     =   60;
Defaults.Interactionstrength.Wall       =   300;
Defaults.Interactionrange.Physical      =   pInfArea;
Defaults.Interactionrange.Social        =   sInfArea;
Defaults.Interactionrange.Wall          =   wInfArea;
Defaults.Weight.Minimum                 =   50;
Defaults.Weight.Maximum                 =   80;
Defaults.Weight.Heavy                   =   1e10;
Defaults.Radius.Minimum                 =   2;
Defaults.Radius.Maximum                 =   6;
Defaults.Aggression.Maximum             =   50;
Defaults.Aggression.Minimum             =   10;
Defaults.nPassengers                    =   20;
%*************************************************************************%
%   Explanation:                                                          %
%                                                                         %
%   The interactionstrength is a factor which influences the forces       %
%   directly. A bigger interactionstrength means that the specific        %
%   force will be increased by that factor.                               %
%                                                                         %
%   The interactionrange is in the divisor of the exponential quotient    %
%   in the specified forces. A bigger interactionrange flattens the       %
%   increase of the resulting force. In the same distance, the resulting  %
%   is smaller the higher the interactionrange.                           %
%*************************************************************************%

%   Description of the group array format:
%   ---------------------------------------------------------------------
%
%   Field Name  |   Description
%   ---------------------------------------------------------------------
%   Starts      |   Stores every starting point of the group.
%   Ends        |   Stores every target point of the group.
%   isFinished  |   'true', if every passenger of that group has reached
%               |   a target point, 'false' else.
%   nSpawned    |   The number of passengers of that group that have
%               |   already started walking.
%   pIndex      |   Index in the passenger array where the group starts.
Groups(nGroups).isFinished    =   0;
Groups(nGroups).nSpawned      =   0;

%   Extract every starting and target area from the k layers.
Groups(nGroups).Starts(1).Position  = [0; 0];
Groups(nGroups).Ends(1).Position    = [0; 0];

%   If you want to add custom information, set this value to 1 for a group.
Groups(nGroups).Custom = 0;
Groups(nGroups).Aggression.Minimum = 0;
Groups(nGroups).Aggression.Maximum = 0;
Groups(nGroups).Radius.Minimum = 0;
Groups(nGroups).Radius.Maximum = 0;
Groups(nGroups).Weight.Minimum = 0;
Groups(nGroups).Weight.Maximum = 0;
Groups(nGroups).nPassengers = 0;
```

```
%   Count exits and spawns.
nSpawns = 0;
nExits = 0;

for i = 1:nGroups,
    Layer = Layers(:,:,1);
    [StartRow,  StartCol,   V] = find(Layer == 2);
    [EndRow,    EndCol,     V] = find(Layer == Inf);
    nStarts = length(StartRow);
    nSpawns = nSpawns + nStarts;
    nEnds   = length(EndRow);
    nExits = nExits + nEnds;
    Groups(i).Starts(nStarts).Position      = [0; 0];
    Groups(i).Ends(nEnds).Position          = [0; 0];
    %   Convert Matrix to structure array.
    for j = 1:nStarts,
        Groups(i).Starts(j).Position = [StartCol(j); StartRow(j)];
    end
    for j = 1:nEnds,
        Groups(i).Ends(j).Position   = [EndCol(j); EndRow(j)];
    end

    Groups(i).Aggression.Maximum = Defaults.Aggression.Maximum;
    Groups(i).Aggression.Minimum = Defaults.Aggression.Minimum;
    Groups(i).Radius.Maximum = Defaults.Radius.Maximum;
    Groups(i).Radius.Minimum = Defaults.Radius.Minimum;
    Groups(i).Weight.Minimum = Defaults.Weight.Minimum;
    Groups(i).Weight.Maximum = Defaults.Weight.Maximum;
    Groups(i).nPassengers = Defaults.nPassengers;
end

%   Load custom group information.
run customGroups;

%   Get nTotalPassengers.
nTotalPassengers = 0;
for i = 1:nGroups,
    nTotalPassengers = nTotalPassengers + Groups(i).nPassengers;
end

%   If no specific markers should be used, set CustomMarkers to 0.
%
%   Custom markers are the marker settings used to draw a group.
CustomMarkers = 1;
for i = 1:nGroups,
    Groups(i).Marker = '.b';
end
Groups(2).Marker = '.g';

%   Description of the wall array format:
%   ---------------------------------------------------------------------
%
%   Field Name  |   Description
%   ---------------------------------------------------------------------
%   Position    |   The position of the wall element.
```

```
%   Weight      |   The weight of the wall element.
%   Unneeded    |   'true' if the wall element is unneeded, 'false' else.
[WallRow, WallCol, V] = find(Map == 0);
nWalls = length(WallRow);

Walls(nWalls).Position      =   [0;     0];
Walls(nWalls).Weight        =   Defaults.Weight.Heavy;
Walls(nWalls).Unneeded      =   0;

for i = 1:nWalls,
    Walls(i).Position       =   [WallCol(i) + 0.5; WallRow(i) + 0.5];
    Walls(i).Weight         =   Defaults.Weight.Heavy;
    Walls(i).Unneeded       =   0;
end

%   Extract unneeded walls:
%
%   We can be sure, that this case can be extracted:
%   x W x
%   W t W
%   x W x
%
%   Where an 'x' means that it does not matter if there is a wall or not,
%   'W' indicates a wall element, and 't' is the target wall element we
%   need to extract.
%
%   The walls could have been extracted in the upper loop, but this
%   way it looks cleaner for the eye.
for i = 1:nWalls,
    x = Walls(i).Position(1) - 0.5;
    y = Walls(i).Position(2) - 0.5;
    if y == 1 || x == 1 || y == m || x == n,
        continue;
    end
    if Map(y - 1, x) == 0 && Map(y + 1, x) == 0 && Map(y, x - 1) == 0 && Map(y, x +↵
1) == 0,
        Walls(i).Unneeded = 1;
    end
end
%   The unneeded marker is used, because we would not extract every
%   unneeded wall correctly if we deleted a unneeded wall directly.

[t, order] = sort([Walls.Unneeded], 'descend');
Walls = Walls(order);
Walls = Walls(find([Walls.Unneeded] == 0));
nWalls = length(Walls);

%   Description of the passenger array format:
%   ------------------------------------------------------------------------
%
%   Field Name  |   Description
%   ------------------------------------------------------------------------
%   Position    |   2x1 - Vector containing the actual position.
%   OldPosition |   2x1 - Vector containing the previous position.
%   Weight      |   Positive scalar value indicating the passenger's
```

```
%               |   weight.
%   Aggression  |   The aggression is a scalar value indicating how much a
%               |   passenger will insist on following his shortest path.
%   Finished    |   Either 'true', if the passenger has reached its target,
%               |   or 'false', if not.
%   Started     |   Either 'true', if the passenger is walking or 'false',
%               |   if the passenger is still waiting.
%   FieldForce  |   2x1 - Vector indicating the force exerted on the
%               |   passenger by its force field.
%   SocialForce |   2x1 - Vector indicating the social force exerted on the
%               |   passenger.
%   RejectForce |   2x1 - Vector indicating the force exerted on the
%               |   passenger by other passengers.
%   WallForce   |   2x1 - Vector indicating the force exerted on the
%               |   passenger by wall elements.
%   TotalForce  |   2y1 - Vector indicating the total force exerted on the
%               |   passenger.
%   Group       |   Scalar value, 0 < Group <= k, indicating the group to
%               |   which the passengers belongs.
%
%   Additionally the passenger contains information about force strengths
%   and ranges:
%
%   'Interactionstrength' is a structure containing:
%       -   Physical
%       -   Social
%       -   Wall
%       -   Aggression
%   'Interactionrange' is a structure containing:
%       -   Physical
%       -   Social
%       -   Wall
%       -   Aggression
%
%   Additional explanation to the aggression level:
%
%   The higher the aggression level of a passenger, the more he pushes
%   other passengers away.

%   Initialize last passenger.
Passengers(nTotalPassengers).Position     =   [0;     0];
Passengers(nTotalPassengers).OldPosition  =   [eps;   0];
Passengers(nTotalPassengers).Finished     =   0;
Passengers(nTotalPassengers).Started      =   0;
Passengers(nTotalPassengers).Weight       =   0;
Passengers(nTotalPassengers).FieldForce   =   [0;     0];
Passengers(nTotalPassengers).SocialForce  =   [0;     0];
Passengers(nTotalPassengers).RejectForce  =   [0;     0];
Passengers(nTotalPassengers).WallForce    =   [0;     0];
Passengers(nTotalPassengers).TotalForce   =   [0;     0];
Passengers(nTotalPassengers).Group        =   0;
Passengers(nTotalPassengers).Radius       =   0;
Passengers(nTotalPassengers).Aggression   =   0;

pNo = 1;
```

```
for i = 1:nGroups,
    nPassengers = Groups(i).nPassengers;
    for j = 1:nPassengers,
        %   Create random weight for every passenger.
        Passengers(pNo).Weight    =   unidrnd(Groups(i).Weight.Maximum - Groups(i).↙
Weight.Minimum) + Groups(i).Weight.Minimum;
        %   Determine group for every passenger.
        Passengers(pNo).Group     =   i;
        %   Determine random aggression for every passenger.
        Passengers(pNo).Aggression    =    unidrnd(Groups(i).Aggression.Maximum -↙
Groups(i).Aggression.Minimum) + Groups(i).Aggression.Minimum;
        %   Initialize every field.
        Passengers(pNo).Position      =   [0;     0];
        Passengers(pNo).OldPosition   =   [eps;   0];
        Passengers(pNo).Finished      =   0;
        Passengers(pNo).Started       =   0;
        Passengers(pNo).FieldForce    =   [0;     0];
        Passengers(pNo).SocialForce   =   [0;     0];
        Passengers(pNo).RejectForce   =   [0;     0];
        Passengers(pNo).WallForce     =   [0;     0];
        Passengers(pNo).TotalForce    =   [0;     0];

        Passengers(pNo).Interactionstrength.Physical     =   Defaults.↙
Interactionstrength.Physical;
        Passengers(pNo).Interactionstrength.Social       =   Defaults.↙
Interactionstrength.Social;
        Passengers(pNo).Interactionstrength.Wall         =   Defaults.↙
Interactionstrength.Wall;
        Passengers(pNo).Interactionrange.Physical        =   Defaults.↙
Interactionrange.Physical;
        Passengers(pNo).Interactionrange.Social          =   Defaults.↙
Interactionrange.Social;
        Passengers(pNo).Interactionrange.Wall            =   Defaults.↙
Interactionrange.Wall;

        pNo = pNo + 1;
    end
end

%   Find the maximum radius for every group.
Spawns = getSpawns(Passengers, Groups, Walls);
for i = 1:nGroups,
    Starts = Spawns(i).Starts;
    MaxRadius = max(Starts) - spawnSecurityFactor;
    if MaxRadius > Groups(i).Radius.Maximum,
        MaxRadius =  Groups(i).Radius.Maximum;
    end
    Groups(i).MaxRadius = MaxRadius;
    if Groups(i).MaxRadius <= Groups(i).Radius.Minimum,
        clear all;
        error('Maximum Radius is smaller then group minimum radius. Change map↙
please.');
    end
end
```

```
%   Give every passenger a valid radius.
for i = 1:nTotalPassengers,
    Passengers(i).Radius = mod(rand, Groups(Passengers(i).Group).MaxRadius - Groups
(Passengers(i).Group).Radius.Minimum) + Groups(Passengers(i).Group).Radius.Minimum;
end

%   Sort accordng to group index.
[t, order] = sort([Passengers.Group]);
Passengers = Passengers(order);

%   Clear out trash.
clear V Layer x y t nStarts nEnds order Defaults StartRow StartCol EndRow EndCol
WallCol WallRow i j Starts MaxRadius Spawns;
```

```
function [Map, Layers] = loadSituation()
%loadSituation: Conversion of k + 1 bmp-images into a map-matrix and a
%               three dimensional matrix consisting of k two dimensional
%               matrices representing the given subsequent bmp-images.
%
%   Only the following colors with their interpretations are allowed:
%
%   Color               Hex         Description
%   White               FFFFFF      Free space. Passengers can freely walk within
%                                   white areas.
%   Black               000000      Wall.
%   Red                 FF0000      Target area for a group.
%   Green               0000FF      Starting area for a group.
%   Yellow              FFFF00      Slow areas (stairs etc.).
%
%   The output map-matrix contains:
%   0 -> Wall, 1 -> Free space, 2 -> slow area
%
%   The output layer-matrices contain:
%   1 -> Free space, 2 -> starting point, Inf -> ending point


%Open text file containing all situation files.
[filename, pathname] = uigetfile('*.txt', 'Please select input text file');
f = fopen([pathname,filename]);
%Fetch first line.
tline = fgetl(f);
%Parse every line.
S = {};
i = 1;
while ischar(tline)
    S{i, 1} = [pathname,tline];
    i = i + 1;
    tline = fgetl(f);
end
fclose(f);
nFiles = length(S);
nLayers = nFiles - 1;


%Every file path is now stored in S.
%The first file S{1,1} is considered to be the map.

%Generate map.
rawMap = imread(S{1,1});
walls = findColor(rawMap, 0, 0, 0);
space = findColor(rawMap, 255, 255, 255);
slow  = findColor(rawMap, 255, 255, 0);

[lines, columns, depth] = size(rawMap);
if (length(walls) + length(space) + length(slow)) ~= lines*columns,
    error('Invalid input map.');
end

Map = zeros(lines, columns);
```

```
Map(walls) = 0;
Map(space) = 1;
Map(slow)  = 2;

%Add a wall around the map.
Map(:, 1)       = 0;
Map(1, :)       = 0;
Map(:, columns) = 0;
Map(lines, :)   = 0;

if nLayers == 0,
    Layers = [];
else
    %Add a wall around every layer.
    Layers = zeros(lines, columns, nLayers);
    for i = 2:nFiles,
        rawLayer = imread(S{i,1});
        space  = findColor(rawLayer, 255, 255, 255);
        starts = findColor(rawLayer, 0, 255, 0);
        ends   = findColor(rawLayer, 255, 0, 0);

        if length(space) + length(starts) + length(ends) ~= lines * columns,
            error(['Invalid input layer: Layer ', num2str(i-1)]);
        end

        layer  = zeros(lines, columns);
        layer(space)  = 1;
        layer(starts) = 2;
        layer(ends)   = Inf;

        Layers(:,:,i-1) = layer;
    end
end
end
```

```
function [Entries] = findColor(Image, R, G, B)
%findColor: Finds all entries in an image of a specified RGB color.
[m,n,t] = size(Image);
if R > 255 | R < 0 | G > 255 | G < 0 | B > 255 | B < 0 | t ~= 3,
    error('Input error in function findColor');
end

search_px = [R;G;B];

Entries = [];
for i = 1:m,
    for j = 1:n,
        px = [Image(i,j,1); Image(i,j,2); Image(i,j,3)];
        if px == search_px,
            Entries = [Entries; i + j*m - m];
        end
    end
end
end
```

```
function [Vectorfields] = preprocessSituation(Map, Layers)
%preprocessSituation:   Takes a map and k layers and computes the
%                       corresponding vectorfields. Then plots the results.
%
%   The input is a matrix, containing the map, and a three dimensional
%   matrix, containing information about the k layers.
%
%   The output is a four dimensional cell array containing the vector fields
%   for every layer.
[m, n, k] = size(Layers);

Walls = find(Map == 0);
Vectorfields = cell(k, 1);
%For every Layer, compute the vector field and store it in a cell array.
for i = 1:k,
    %Import the walls into every layer.
    Layer = Layers(:,:,1);
    Layer(Walls) = 0;
    [VFX, VFY] = computeVF(Layer, sprintf('Layer%d.jpg', i));
    VF = zeros(m, n, 2);
    VF(:,:,1) = VFX;
    VF(:,:,2) = VFY;
    Vectorfields{i} = VF;
end


end
```

```
function [VFX, VFY] = computeVF(M, File)
%computeVF:    Computes the vectorfield of a map for
%              given starting and target areas.
%
%
%   The input is a m*n-matrix containing information about walls, spaces
%   and target positions. The output are two matrices containing the x
%   and y components of the vectors of a vectorfield in every point of the
%   input matrix. These vectors represent the direction leading a passenger
%   onto the shortest path to the nearest target.

%   As defined in the loadSituation.m file, the codes we need are:
%   Wall = 0, Space = 1, Exit = 3
%
%   Note:   If a file name is specified, the function will store the
%           appearance of the provided matrix into that file.
if nargin < 2,
    File = -1;
end


[m, n] = size(M);

%   Find walls in the input matrix.
F = ones(m , n);
Walls = find(M == 0);
F(Walls) = 0;

%   Find exits.
[ExitRows, ExitCols, V] = find( M == Inf );

%   Generate exit vector.
nExits = length(V);
Exits = zeros(2, nExits);
Exits(1, :) = ExitRows;
Exits(2, :) = ExitCols;

%   Apply fast marching and gradient.
options.nb_iter_max = Inf;
[D, S] = perform_fast_marching(F, Exits, options);
[VFX, VFY] = gradientField(D);

%   Plot if needed.
if File ~= -1,
    fig = figure('visible', 'off');

    %   Plot contours.
    D(D == inf) = 0;
    contour(D);
    hold on;

    %   Plot Vectorfield
    quiver(VFX, VFY, 2);

    [WallRows, WallCols, V] = find(F == 0);
```

```
    %   Plot Walls
    p = plot(WallCols, WallRows, '.k');
    set(p, 'MarkerSize', 10);

    %   Plot Exits
    p = plot(ExitRows, ExitCols, '.r');
    set(p, 'MarkerSize', 20);

    %   Print to file
    print(fig, '-djpeg', File);
end

end
```

```
function [FFX, FFY] = gradientField(M)
%gradient:  Calculate the gradient basing on a matrix M and ignore entries which
%           are infinitely large.
%
%   Input is a matrix M with a potential field and entries set to 'Inf'.
%   The infinite entries represent walls. This function generates a vector
%   field representing the field of M, ignoring all infinite entries.
%
%   This function is based on the gradient_special function used in the
%   base project.
[m, n] = size(M);

FX = zeros(m, n);
FY = zeros(m, n);
FFX = zeros(m ,n);
FFY = zeros(m, n);

%   Check every element of the input matrix.
for i = 1:m
    for j = 1:n
        %   Is M(i, j) part of the wall?
        if M(i, j) == Inf,
            %   Is M(i, j) on the border of the map?
            if j > 1 && j < n,
                %   M(i, j) is not on the border.
                %   Check the following cases:

                %   1. WÃ°W
                if M(i, j - 1) == Inf && M(i, j + 1) == Inf,
                    FX(i, j) = 0;
                %   2. WÃ°Ã°
                elseif M(i, j - 1) == Inf,
                    FX(i, j) = M(i, j) - M(i, j + 1);
                %   3. Ã°Ã°W
                elseif M(i, j + 1) == Inf,
                    FX(i, j) = M(i, j - 1) - M(i, j);
                %   4. Ã°Ã°Ã°
                else
                    FX(i, j) = 0.5*(M(i, j - 1) - M(i, j + 1));
                end

            elseif j > 1,
                %   M(i, j) is on the right border.
                %   Check the following cases:

                %   1. WÃ°
                if M(i, j - 1) == Inf,
                    F(i, j) = 0;
                %   2. Ã°Ã°
                else
                    F(i, j) = M(i, j - 1) - M(i, j);
                end

            else
                %   M(i, j) is on the left border.
```

```
        %   Check the following cases:

        %   1. Â°W
        if M(i, j + 1) == Inf,
            FX(i, j) = 0;
        %   2. Â°Â°
        else
            FX(i, j) = M(i , j) - M(i, j + 1);
        end
    end

    %   is M(i, j) on the border of the map?
    if i > 1 && i < n,
        %   M(i, j) is not on the border.
        %   Check the following cases:

        %   1.  W
        %       Â°
        %       W
        if M(i - 1, j) == Inf && M(i + 1, j) == Inf,
            FY(i, j) = 0;
        %   2.  W
        %       Â°
        %       Â°
        elseif M(i - 1, j) == Inf,
            FY(i, j) = M(i, j) - M(i + 1, j);
        %   3.  Â°
        %       Â°
        %       W
        elseif M(i + 1, j) == Inf,
            FY(i, j) = M(i - 1, j) - M(i, j);
        %   4.  Â°
        %       Â°
        %       Â°
        else
            FY(i, j) = 0.5*(M(i - 1, j) - M(i + 1, j));
        end
    elseif i > 1,
        %   M(i, j) is on the bottom border.
        %   Check the following cases:

        %   1.  W
        %       Â°
        if M(i - 1, j) == Inf,
            FY(i, j) = 0;
        %   2.  Â°
        %       Â°
        else
            FY(i, j) = M(i - 1, j) - M(i, j);
        end
    else
        %   M(i, j) is on the top border.
        %   Check the following cases:

        %   1.  Â°
```

```
                %       W
                if M(i + 1, j) == Inf,
                    FY(i, j) = 0;
                %   2.  Â°
                %       Â°
                else
                    FY(i, j) = M(i, j) - M(i + 1, j);
                end
            end
        else
            %   M(i, j) is part of the wall.
            FX(i, j) = 0;
            FY(i, j) = 0;
        end

        %   Normalize vector
        if FX(i, j) ~= 0 && FY(i, j) ~= 0,
            FFX(i, j) = FX(i, j)/(sqrt( FX(i, j)^2 + FY(i, j)^2 ));
            FFY(i, j) = FY(i, j)/(sqrt( FX(i, j)^2 + FY(i, j)^2 ));
        elseif FX(i, j) ~= 0,
            FFX(i, j) = FX(i, j)/abs( FX(i, j) );
            FFY(i, j) = 0;
        elseif FY(i, j) ~= 0,
            FFX(i, j) = 0;
            FFY(i, j) = FY(i, j)/abs( FY(i, j) );
        end
    end
end
end
```

```
function Spawns = getSpawns(Passengers, Groups, Walls)
%getSpawns:     Returns a structure array of Spawn-entries.
%               A spawn entry consists of the group index, the starting
%               point index and the radius which is available for spawn.
%
%   Inputs are three structure arrays, one containing information about the
%   passengers, the second one containing information about the groups and
%   the last one containing information about the wall elements.
nGroups     = length(Groups);
nWalls      = length(Walls);
nPassengers = length(Passengers);

Spawns(nGroups).Starts      = 0;

%   Check every group.
for i = 1:nGroups,
    %   Check every starting area of every group.
    l = length(Groups(i).Starts);
    Spawns(i).Starts(l) = 0;
    for j = 1:l,

        %   If every passenger position is [0; 0], don't check against
        %   passengers. It would return awkward results.
        checkPassengers = 0;
        if [Passengers.Position] ~= zeros(2, nPassengers),
            checkPassengers = 1;
        end

        %   Check against every passenger.
        Pos = [Passengers.Position] - repmat(Groups(i).Starts(j).Position, 1,↙
nPassengers);
        radius = norm(Pos(:, 1));
        for k = 2:nPassengers,
            t = norm(Pos(:, k));
            if t < radius,
                radius = t;
            end
        end

        %   Check against walls.
        Pos = [Walls.Position] - repmat(Groups(i).Starts(j).Position, 1, nWalls);
        startIndex = 1;
        if checkPassengers,
            radius = norm(Pos(:,1));
            startIndex = 2;
        end
        for k = startIndex:nWalls,
            t = norm(Pos(:, k));
            if t < radius,
                radius = t;
            end
        end

        Spawns(i).Starts(j) = radius;
    end
```

```
end

end
```