Master Thesis:
Pedestrian Simulations with the Social Force Model

Anders Johansson[2,1]

[1]Institute for Transport & Economics,
Dresden University of Technology,
Andreas-Schubert Str. 23, 01062 Dresden, Germany

[2] Department of Physical Resource Theory,
Chalmers University of Technology,
41296 Göteborg, Sweden

Supervisor: Prof. Kristian Lindgren[2]
2^{nd} Supervisor: Prof. Dr. rer. nat. Dirk Helbing[1]

November 16, 2004

Abstract

In order to fully understand and analyze how the infrastructure in public facilities as well as in urban regions affects the overall characteristics of pedestrian crowds within these areas, computer simulations need to be performed. An implementation of the Social Force Model is discussed, which is suitable for both small-scale pedestrian simulations as well as computationally fast large-scale pedestrian simulations. Simulations are presented which are used to test proposed design solutions for maximizing the throughput and avoid peaks of density and pressure at critical locations. Among the applications one can find; Pilgrim streams on the Jamarat bridge, evacuation of soccer arenas and theaters. Also, self-organization phenomena such as lane formation, stripe formation and oscillations at bottlenecks are discussed.

Contents

Chapter 1

Introduction

This thesis is presenting a model for simulation of pedestrians together with a description of an effective implementation and some applications with corresponding results and analysis.

The model which is used is based on the *Social Force Model* (Helbing, 1991, 1995, 1997; Helbing and Molnár, 1995; Helbing and Vicsek, 1999; Helbing *et al.*, 2000; Molnár, 1996a, 1996b).

There is always a tradeoff between correctness and computational speed. Therefore the aim of this project is to make the software as flexible as possible. It may be used to simulate really simple pedestrian scenarios with almost no planning at all, but it may also be used to simulate sophisticated pedestrians that perform advanced planning and so on. The core of the simulation is simple and optimized to be as fast as possible and then several modules can be invoked for tasks like:

- Path planning with arbitrary precision.
- Statistics of velocity, density and other pedestrian properties at different areas in the simulation.
- The simulation core is not dependent of any graphics output, but there are modules that can be attached to show the simulation at every n:th time step. Currently a 2D GUI in SDL, a simple 3D GUI in OpenGL and a Flash Export tool exists.

The flexibility also includes that all sourcecode is written in standard ISO C++ and all code has been successfully tested both on Debian Linux and Windows XP.

The simulation itself can be specified in XML format and given as input to the simulator. This also makes it easy to convert CAD files to suitable

input to the simulator.

Another choice that can be made is if the simulation shall be started as fast as possible or if the simulation should be as fast as possible when started. Information about static forces and pedestrian routes may be precalculated or calculated on-the-fly.

Chapter 2

The Model

The model itself consist of some primitives:

- *Agent*: A pedestrian with a set of properties.
- *Obstacle*: A wall that will produce forces onto agents in the vicinity.
- *ForceField*: A polygon area with a constant force within. Can be used to model stairs or sloping floors.
- *Goal*: A location where agents want to go.

The pedestrian movement is built up by different forces:

- Obstacle forces: Repulsive forces are applied to all pedestrians in the vicinity of the obstacle with an increasing magnitude when the distance is decreasing. The force from the wall, B, to the pedestrian α is specified as:

$$\vec{f}_{\alpha B}(\vec{r}_\alpha) = -A_o \nabla_{\vec{r}_\alpha} e^{\frac{-||\vec{r}_{alpha} - \vec{r}_B||}{B_o}} \tag{2.1}$$

  where the point $\vec{r}_B$ is the point along obstacle B which is closest to the current pedestrian position, $\vec{r}_\alpha$.

- Pedestrian forces: Between all pedestrians there are repulsive forces that increase in magnitude when the distance is decreasing. The force has two parts, one physical part and one social part, because people like to have a personal sphere around them, a territorial effect. The forces between pedestrian α and pedestrian β are specified as:

$$\vec{f}^{ph}_{\alpha\beta}(t) = A_\alpha e^{\frac{r_\alpha + r_\beta - d_{\alpha\beta}}{B^{ph}}} \vec{n}_{\alpha\beta} \tag{2.2}$$

$$\vec{f}^{soc}_{\alpha\beta}(t) = A_\alpha e^{\frac{r_\alpha + r_\beta - d_{\alpha\beta}}{B_\alpha}} \vec{n}_{\alpha\beta}(\lambda_\alpha + (1-\lambda_\alpha)\frac{1 + cos(\varphi_{\alpha\beta})}{2}) \tag{2.3}$$

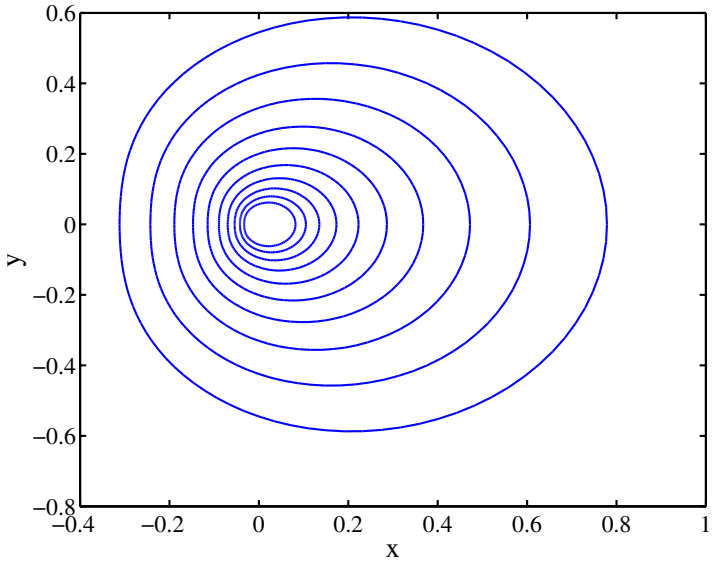

Figure 2.1: Unisotropic forcefield

where $\vec{n}_{\alpha\beta} = \frac{\vec{r}_\alpha - \vec{r}_\beta}{||\vec{r}_\alpha - \vec{r}_\beta||}$, $\vec{e}_\alpha = \frac{\vec{v}_\alpha}{||\vec{v}_\alpha||}$ (the desired direction of movement), $\varphi_{\alpha\beta}$ is the angle between $\vec{n}_{\alpha\beta}$ and $\vec{e}_\alpha$, A_α is the interaction strength, B_α is the range of the interaction.force and λ_α is a value in the range $[0, 1]$ specifying the shape of the unisotropic forcefield around the agents. A value $\lambda_\alpha < 1$ results in a behaviour that makes the agent react stronger in events that happens in front of her (see figure 2.1).

- Driving forces: Each pedestrian has a desire to go to some location and this is modelled by a force pointing towards that location, $F_d\alpha$

When all forces are calculated they are added into one resultant force, $\vec{f}_\alpha$, which is then used to update the velocity and location of the pedestrian according to the following movement functions:

$$\frac{d\vec{r}_\alpha(t)}{dt} = \vec{v}_\alpha(t) \tag{2.4}$$

$$\frac{d\vec{v}_\alpha(t)}{dt} = \vec{f}_\alpha(t) + \vec{\xi}_\alpha(t) \tag{2.5}$$

$$\vec{f}^0_\alpha = \frac{1}{\tau_\alpha}(v^0_\alpha \vec{e}_\alpha - \vec{v}_\alpha) \tag{2.6}$$

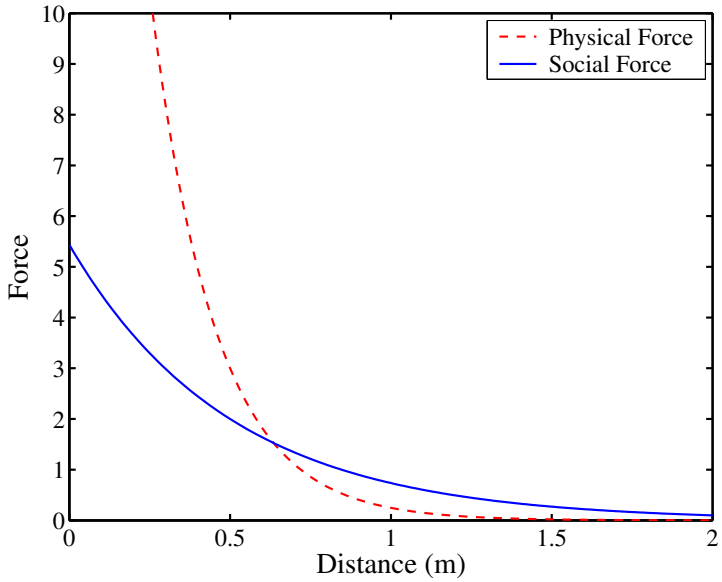

Figure 2.2: Forces

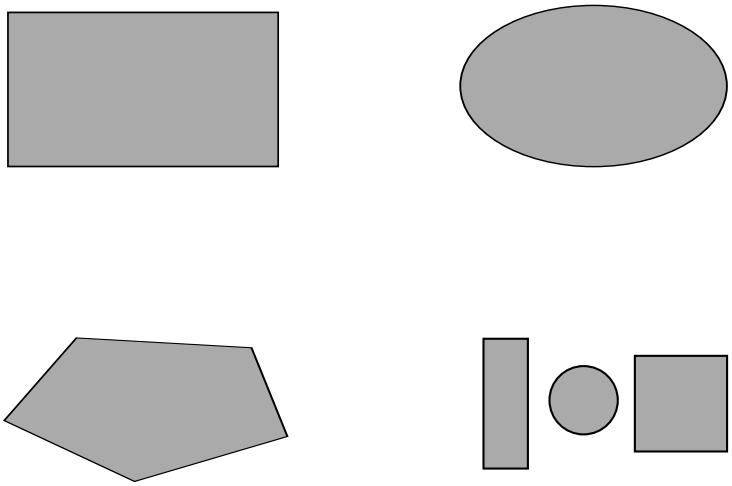

Figure 2.3: Goal Types: Rectangular, Ellipse, Polygon and Complex Area

where $\vec{\xi}_\alpha(t)$ is a small amount of stochasticity added to the model to smoothen up the movement.

See figure 2.2 as an example of the inter-pedestrian forces as a function of the distance between the two pedestrians.

The simulation core is simple but generic, which makes it possible to attach modules for all kind of behaviours and simulation features. Pedestrians are introduced in the simulation inside an area (see figure 2.3) and pedestrians are not treated as single indivuduals but as different groups with a set of macroscopic quantities attached to them, such as

- maximal number of pedestrians,
- rate of birth,
- a list of tasks to perform,
- parameters for normal distributed velocities.

2.1 Additions to the Force Model

2.1.1 Avoidance of Congested Areas

Most of the pathfinding and statical forces may be precalculated before the simulation, but it may happen that some areas become really crowded and in some cases even can be considered a serious obstacle to pass. Therefore an addition to the model is needed to consider dynamical obstacles aswell. Two main constraints need to be considered:

- It should not slow down the simulation more than marginally.
- It should be precise enough to fulfill its purpose.

The most obvious way to solve this would be to add an additional long range social force which build up a superpositioned force strong enough to lead pedestrians around crowded areas. One major drawback with this method is however that it cost a lot of computational time if the area of interaction around each pedestrian is enlarged. Also, it would not result in the desired functionality that pedestrians are repelled from high densities only, but also increase the inter-pedestrian distances in low density areas. Therefore an additional density grid is set up, that has a slightly repulsive effect on pedestrians. The densities are both low-pass filtered,

$$\rho_i{}^{low-pass} = (1-k) * \rho_i{}^{low-pass} + k * \rho_i{}^{instant},$$

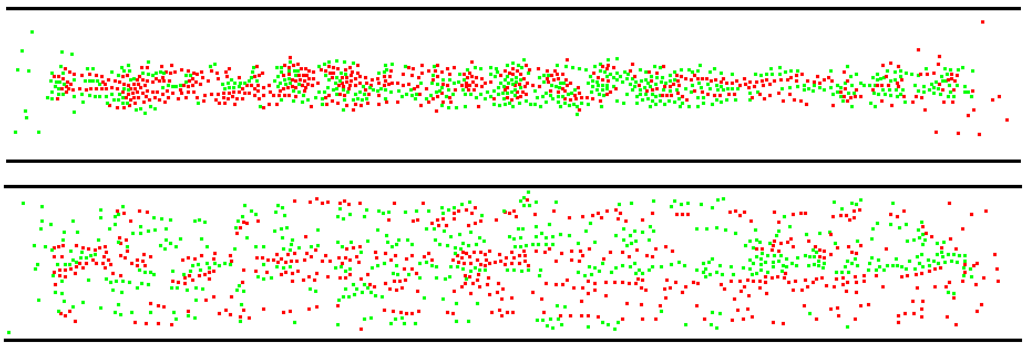

Figure 2.4: Simulation of corridor

as well as distributed,

$$\rho_i = (k-1) * \rho_i + \frac{k}{8} * \sum_{j \neq i} \rho_j,$$

to the nearest neighbors. The outcome is that the pedestrian routes will converge to an optimal usage of space over time. See figure 2.4 for an example of bi-directional streams in a corridor, both with and without the density grid.

2.1.2 Force Separation

One problem with the Social Force Model is that there is no consideration of the areas between pedestrians when the forces are calculated. For example, if there is a group of pedestrians separated by a wall, then the forces between pedestrians are equal both between pedestrians at the same side aswell as between pedestrians at opposite sides. To solve this a grid is created that keeps track of the relative positions to the obstacles.

The only information needed to know in each position in the grid is:

- which obstacles are in the neighborhood,
- what the relative positions to these obstacles are; left side or right side.

To bookkeep the data two bitmasks are used:

- `a`, affected, where each bit, i, denotes if the obstacle number i is within the interaction distance d,

- `o`, orientation, where each bit, i, denotes if the current grid cell is to the left (0) or to the right (1) of the obstacle i.

The grid is created with these steps:

1. Create a grid of size n x m where the size of the cells are a few times smaller than the size of the pedestrians.
2. Update the affected list, `a`, and set the bits i where the obstacle i is within the grid cell.
3. Distribute the information of the affected list to all neighbor cells up to the interaction distance d.
4. Set the orientation lists, `o`, to the corresponding left/right value. In cases where the point is outside the endpoints of the obstacle, set the corresponding `a` value to 0.

See figure 2.5 for an example of the four steps.

When the grid is created it is a simple task to check whether two pedestrians, α and β, should interfer with each other or not. Since it is only the obstacles which are within distance d from both pedestrians that are of importance, a combined bitmask are created with use of the logical `and` operator.

$$a_{combined} = a_\alpha \texttt{ and } a_\beta,$$

and then the test to check if the two pedestrians interfer is:

$$(o_\alpha \texttt{ and } a_{combined}) = (o_\beta \texttt{ and } a_{combined})$$

Figure 2.6 shows how it looks both with and without the force separation.

Note that because of limitations in the operating system and the computer architecture the bitmasks can not be longer than typically 32 or 64 bits. This is however no problem because the information can easily be saved in a series of variables and then one test for each block need to be performed. Also, normally not all of the obstacles in the simulation are relevant to test for force separation so those who are important may be marked with a flag and the other obstacles will be excluded from the test.

2.2 Optimization

To gain computational speed while losing as little of the precision and correctness as possible a few tricks are used.

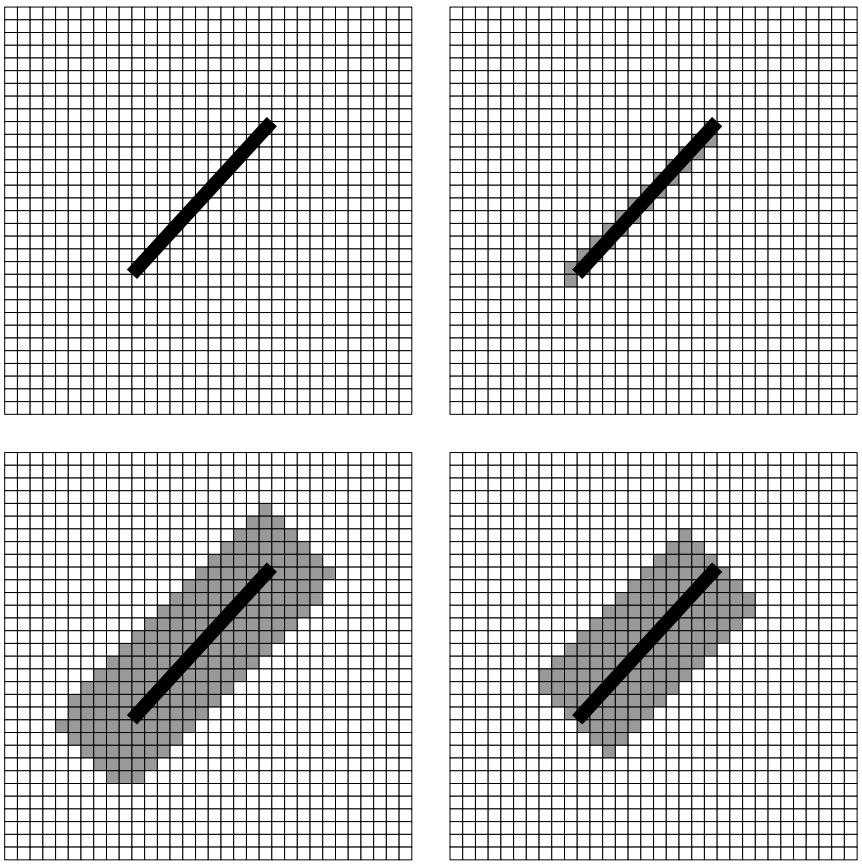

Figure 2.5: The four steps in the creation of the force separation grid

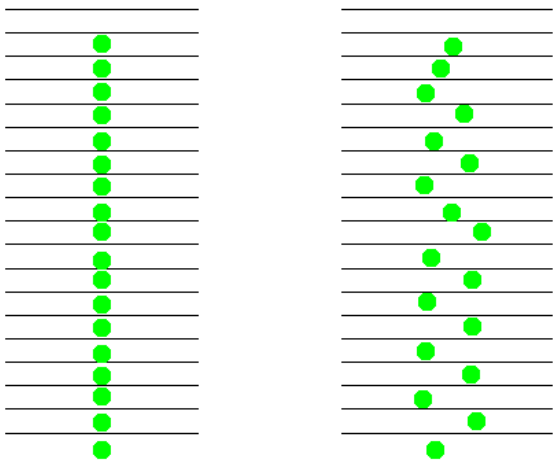

Figure 2.6: The same simulation - both with and without the force separation

2.2.1 Interaction Grid

A grid is wrapped around the environment and each grid cell has a dynamical list that keeps track of which pedestrians are located on this grid cell. Then only the pedestrians in the grid cell itself and the eight surrounding cells are considered when the inter-pedestrian forces are calculated. This process is the most time consuming one and the speed complexity is changed from $O(n^2)$ to $O(n)$ with this trick (assuming constant density), which improves the speed significantly for simulations with many pedestrians. The only drawback is that the amount of memory needed is now increasing proportionally to the number of grid cells, and not a constant one, as in the unoptimized case.

2.2.2 Statical Force Grid

Another grid with higher resolution is wrapped around the environment and each cell has a vector that is the summation of all statical forces (i.e. obstacle forces) in that cell. All cells also have a Boolean value that tells whether the force has been calculated or not. All static forces may be pre-calculated before the simulation starts, or if they are not they will simply be pre-calculated on-the-fly as soon as a pedestrian is entering a cell with an uncalculated force. The geometry of this grid is however slightly different from the previous one, since the size of the grid cells need to be very small around obstacles, but not

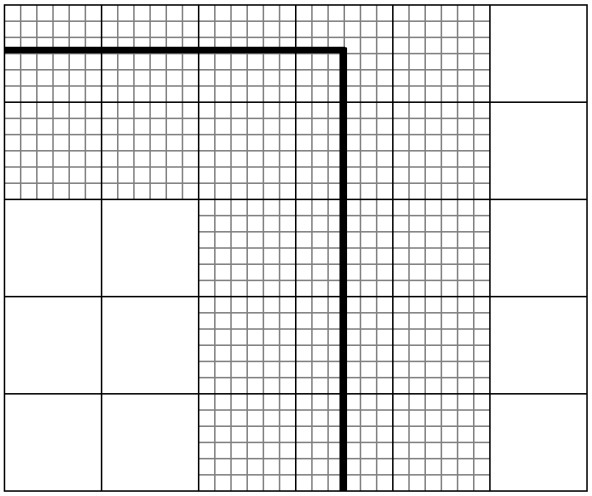

Figure 2.7: Topological Grid

in the middle of empty areas, this grid is structured in a topological manner. The organization is made in two steps: one regular grid, and each cell may either contain a single force, or a new forcefield. Figure **??** shows an example of this grid around an obstacle.

2.2.3 Layers

To be able to simulate more than statical environments, another feture is implemented that arranges the obstacles onto an arbitrary number of layers. One separate statical force grid is used at each layer and then the resultant statical force is the addition of the local forces in each layer. With help of this structure a scheduling object is implemented that is able to:

- Enable or disable a certain layer at a specific time.
- Show or hide a certain layer at a specific time.
- Start or stop moving a certain layer with a constant velocity at a specific time.

With help of the layers and the scheduling object more advanced dynamic may be simulated, such as trams and crossings with traffic lights. Note also that the scheduling has two parts, one offset time and one periodic time, which makes it possible to schedule events every T seconds.

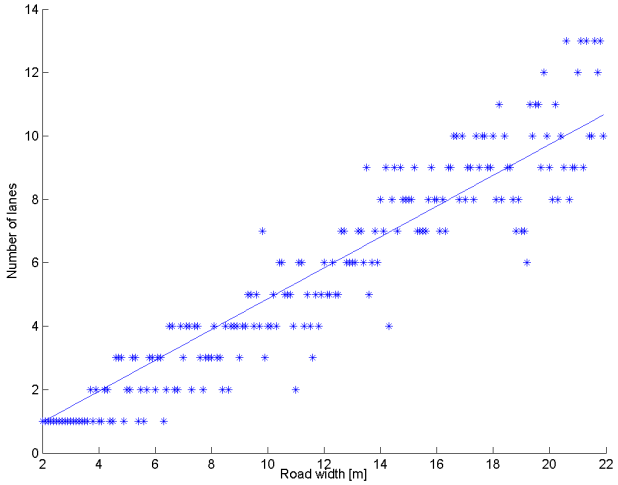

Figure 2.8: Lane formation

2.2.4 Agent Forces

One of the most computational expensive part in the simulation is to calculate the forces between pedestrians. This can however be precalculated by defining a number of n personalities (with a corresponding setup of parameters) and then create a number of $n * n$ matrices which will consist of the forces between two pedestrians at different distances dx, dy.

2.3 Calibration and Self-organization Phenomena

In order to get a useful simulation tool, it needs to be calibrated to empirical data before it can be used to simulate new untested scenarios. This section will show simulation results of some self-organization phenomena together with a few simple simulations which has been tried with real humans and where empirical data has been collected.

2.3.1 Lane Formation

One macroscopic property in bi-directional pedestrian flows is that they will self-organize into distinct lanes. Figure 2.8 shows the result of several simulations of a bi-directional corridor with the number of lanes as a function of

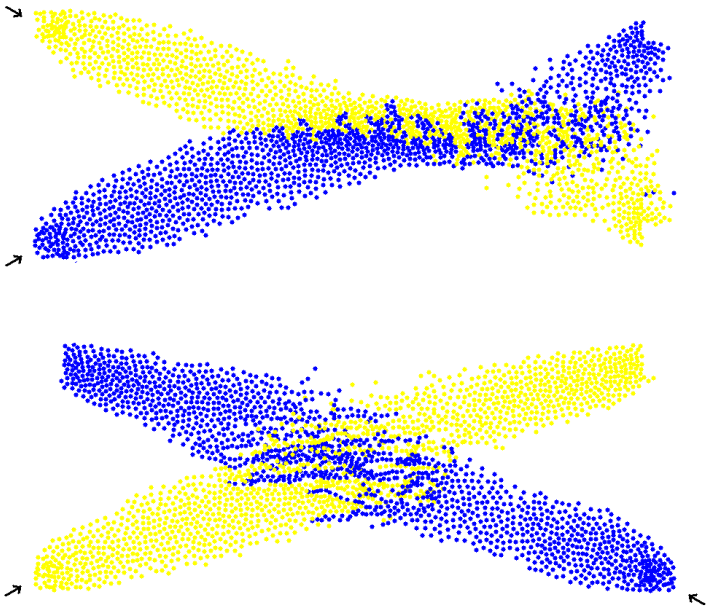

Figure 2.9: Stripe formation

corridor width. The linear regression is very close to previous results (Helbing 1997). The reason for the lane formation is that it is more efficient to move inside a trail which is heading in the same direction, than to move inside an area with mixed walking directions or even worse, and area where pedestrians have complete opposite walking direction.

2.3.2 Stripe Formation

When two pedestrian streams are moving not in parallel with opposite directions but with another angle in between, another self-organized phenomena called stripe formation occur (Ando *et al.*, 1988). The stripes are not stationary but propagating along the direction of the average of the two crowds (Dzubiella and Löwen, 2002). This gives rise to an effective two-part movement scheme, where pedestrians move forward with the stripes and sidewards within the stripes. See figure 2.9 for a simulation of two different angles. The stripes are horizontal in one case and vertical in the other case.

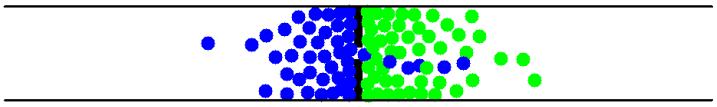

Figure 2.10: Corridor with a bottleneck

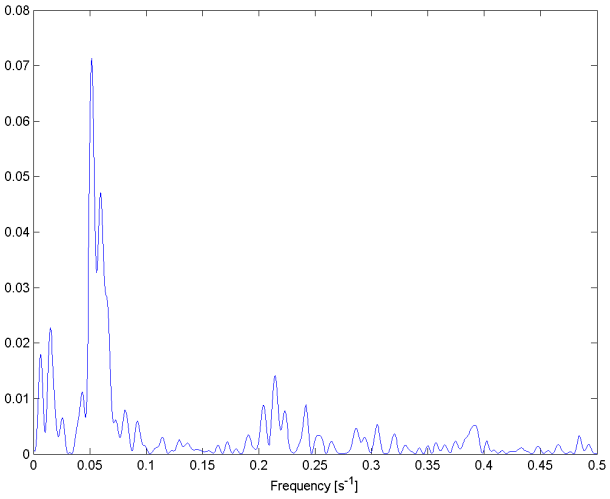

Figure 2.11: Frequency distribution of oscillations at bottleneck

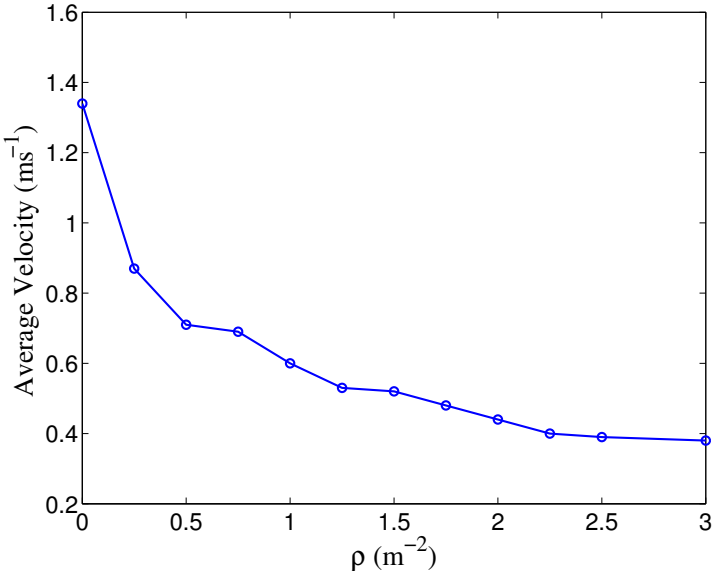

Figure 2.12: Velocity-Density

2.3.3 Oscillations at Bottlenecks

Another simulation is performed in a corridor but with a bottleneck added in the middle, as figure 2.10 shows. A bottleneck with a narrow gap give rise to oscillations when pedestrians are coming from both sides. This can be explain with evaraging of pressures: when the pedestrians are moving through the bottleneck in one direction it is easier for the pedestrians on the corresponding side to move along the stream, than for the pedestrians on the other side to walk against the stream. However, after some time there are more pedestrians waiting on the other side and a pressure is building up. When the pressure is high ebough these pedestrians are able to change the pedestrian flow through the bottlenceck in the other direction. These two phenomena give rise to the oscillations. To show that these oscillations really occur, a frequency distribution of the velocity oscillations over time is calculated from the simulation, and figure 2.11 clearly show a peak which is the main oscillation frequency.

2.3.4 Velocity - Densitiy

Another simulation is performed where a pedestrian is trying to move in a crowded room. The average walking speed as a function of the pedestrian density is shown in figure 2.12. This result is close to the one presented by

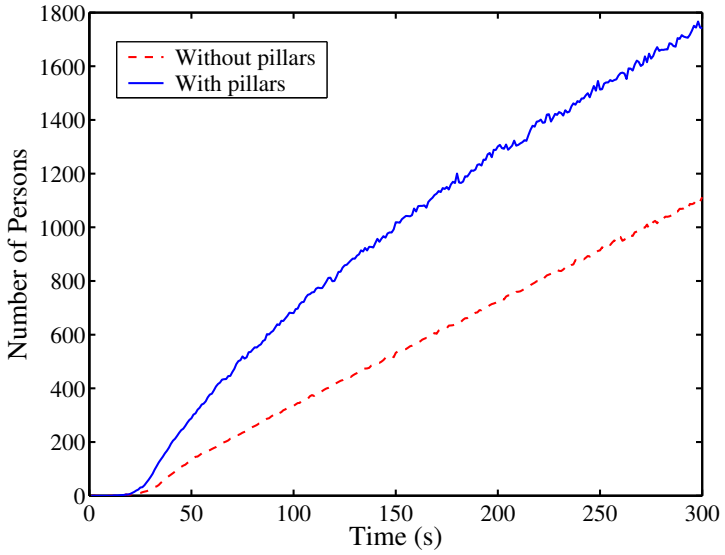

Figure 2.13: Corridor with and without pillars

AlGadhi *et al.*, 2002.

2.3.5 Pillars Added for Stabilization of Flows

For bi-directional corridors where the density of pedestrians is high and the desired velocities are widespread, the lane formation is not very effective and the lanes are short-lived due to the heavy perturbations. To avoid this problem a row of pillars can be added to the middle of the corridor, to separate the two directions of movement. The pillars are not separating the two sides completely but they are more of a psycological obstacle, and also, even though the total walking area is decreased by introducing the pillars, the throughput is increased as figure 2.13 shows.

Chapter 3

Path Finding

To be able to analyze complex pedestrian scenarios it is not always enough with the driving force to move the pedestrians. Therefore, path finding algorithms need to be constructed to be able to route the pedestrians around the environment. There are two kinds of path findings, one long term route planning and one short term planning to avoid obstacles and other pedestrians. The idea with the *Social Force Model* is that as little effort as possibe should be invested in the short term planning because the repulsive forces will take care of that automatically.

The path finding is subdivided into three different types:

1. Short Distance Routes: Prevent crashing into meeting pedestrians. The social force model is taking care of this.
2. Medium Distance Routes: Routes within the same room.
3. Long Distance Routes: Routes between rooms.

3.1 Long Distance Routes

The idea here is the same as for the rest of the simulator - precalculate as much as possible so that the simulation will be as fast as possible, after initialization. To be able to find a good path planning algorithm the first step is to find a good representation of the paths. Two kind of primitives are used - rooms and gates. This is what is needed:

- Flexibility. Not even the mathematically best path should be found but all possible detours and the fitness of each path will determine the probability of this path to be choosen of the agent.

- Memory issue - as little redundancy data as possible should be saved but still, not too less data to be able to find all paths should be saved.

The outcome of the needs is two matrices:

- One matrix with size $n * n$, where n is the number of rooms and each element is a list of gates connecting the two rooms together, plus a fitness value saying how good each gate is assumed to fulfill its purpose.
- One other matrix of same size as the first one but where each element (i, j) is a list of the intermediate rooms that connects room i to room j. If more than one intermediate room is needed the element will instead consist of a series of rooms needed to go through from room i to room j. In case two a Boolean variable *longTour* is set to *true* to denote that the element is of the second type.

With this internal representation all possible tours between all rooms are determined before the simulation starts. Note however that for paths longer than three rooms, all possible paths will not be specified but only one possible path. To fill the second matrix with data an iterative process will find empty elements and try to find tours via a third room. Figure 3.1 shows an example of routes between rooms. Note that different pedestrians have different goal points. That is why some of them are leaving the main trail at a few locations.

3.2 Medium Distance Routes

Now when the pedestrians have the tools required to navigate between rooms, another part, called medium distance planning from now on, is needed to navigate inside the rooms. The idea here is still to minimize calculation time primarily and memory usage secondarily but still keep the accuracy high enough so that the algorithm will be useful. To quantify the empty space (floor free from obstacles) a number n of way-points (navigation points) are placed around the free space. Then an $n * n$ sized matrix is created where each element (i, j) has the members *distance*, *path* and *directConnection*. If the sight from way-point i to way point j is free from obstacles the *directConnection* is set to *true* and the *distance* is set to the distance between the two way-points. Then an iterative process is processing each empty route trying to find a detour via another matrix element. If multiple connections exist, the connection with lowest distance is choosen. This algorithm is based on the *Floyd-Warshall algorithm*.

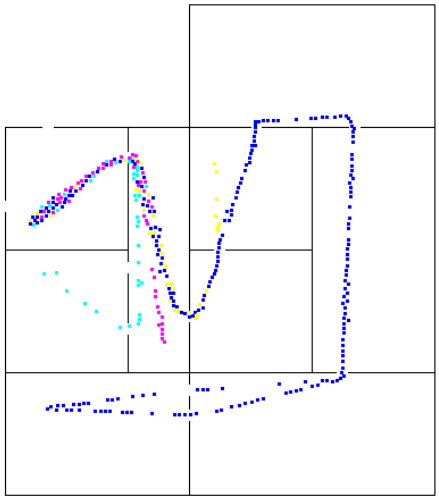

Figure 3.1: Path Finding - Finding routes between rooms

The procedure to find a path from point p_0 to p_1 is then constructed as following:

1. Find way-point p_{start} which is the point nearest to p_0.
2. Find way-point p_{end} which is the point nearest to p_1.
3. Add all way-points in the *path* list for element (p_{start}, p_{end}) in the matrix to the route.
4. If not free sight from current point to p_{end} goto 3.

The distance above is the *Manhattan Distance*, $|x_1 - x_0| + |y_1 - y_0|$. This means that the choosen point may be a few percentages further away than the nearest one, but this small lack of accuracy is an easy tradeoff because a lot of computational time is saved. Note that the computational speed complexity of the algorithm is $O(n)$, where n is the numer of way points. It is rather fast since no hard computation occurs, because it is only lookups in pre-calculated matrices.

A non-trivial part of the algorithm is how to place the way points and how many to use. A first naive approach would be to place them evenly spaced onto an $n * n$ sized grid but this is obviously a waste of memory and computational time if there are big areas of free space. Instead a method based on the force model itself is used:

1. Place an arbitrary number of way points at random locations in the room

2. Update the way point locations, wp, with a schematic updating rule:

$$wp_i = wp_i + \sum_{j=1}^{n_{obstacles}} n(i,j) * \frac{1}{[distance(i,j)]^2} + \sum_{j=1}^{n_{waypoints}} n(i,j) * \frac{1}{[distance(i,j)]^2} \tag{3.1}$$

$distance(i,j)$ is the distance between the objects i and j and $n(i,j)$ is the normalized vector pointing from object j to object i. The equation above should however be considered a schematic one because the forces are not summarized over all objects but instead the nearest intersecting distances in a number of directions are used as input to the force calculations.

Note that the force here is proportional to the inverse of the squared distance and not exponential as for the force model. The reason for this is that the uniformly distribution requires that the influence from other objects is reaching far away and is not a spike when objects are nearly hitting each other, as for the force model.

Figure 3.3 shows the outcome of a way point distribution example, to show how it scales to the number of obstacles, and figure 3.4 shows the outcome of a path finding example.

Another non-trivial thing is to choose the number of way points to use. Some simple examples show that the theoretical number of way points is roughly proportional to the number of obstacle clusters in the room. Figure 3.2 shows examples of way points to span the space for different number of obstacles. Table 3.1 shows that the free space theoretically can be quantified with $2(n+1)$ numer of way-points, where n is the number of separated obstacles. However, these points are not easy to find, so in reality a little more way points will be needed to span the free space.

3.3 Quantifying the Environment

For some simulations it may be important not only that the pedestrians find their ways but also what kind of motion scheme they are following. If this is the case the algorithms described earlier have a tendancy to give a

n	experimental	$2 * (n + 1)$
1	4	4
2	6	6
3	8	8
4	9	10
5	12	12
6	14	14

Table 3.1: Number of way points as a function of number of separated obstacles, n

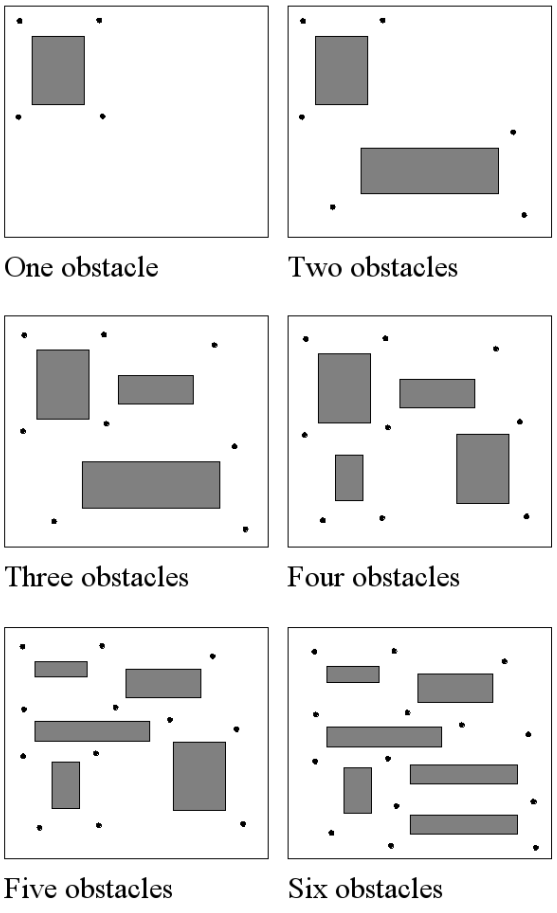

Figure 3.2: Path Finding - Way Points uniformly distanced in space

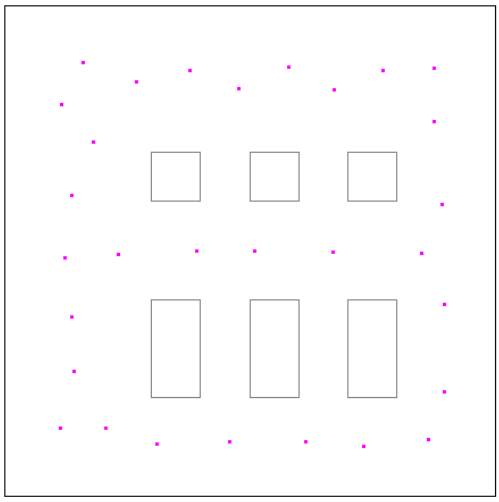

Figure 3.3: Path Finding - Way Points uniformly distanced in space

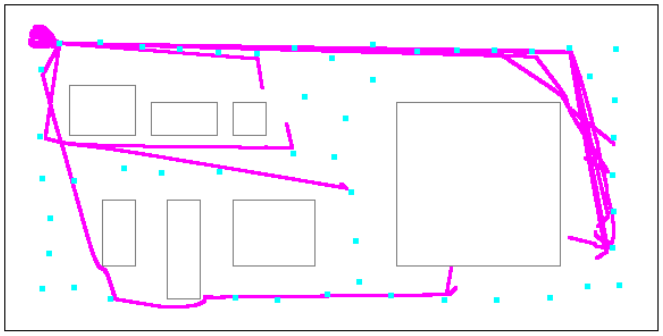

Figure 3.4: Path Finding - An example of finding point (10, 10) from random starting locations

higher density of pedestrians around the way point locations - especially if the number of way points is low. To solve this problem an alternative algorithm is made which smoothens up the motion schemes and also makes the path finding faster,

1. Iterate about 100 000 times:
   - Get a random starting point (s_x, s_y).
   - Get a random goal point (g_x, g_y).
   - Use the Floyd-Warshall or similar path finding algorithm to find the route.
   - Log path to a file.
2. Use an artificial neural network (ANN) with the setup:
   - 4 input nodes: s_x, s_y, g_x, g_y.
   - 2 output nodes f_x, f_y (driving force).
   - 8-32 hidden nodes.
3. Train the ANN about 100 000 times
4. Now the ANN can be used by giving the current location (s_x, s_y) and some desired goal point (g_x, g_y) as an input to the ANN and the output will be the x and y components of the force that will take the pedestrian there.

The characteristics of this algorithm is:

- About one minute of processing time to get samples and train the ANN.
- Then, to get a direction from a given point (s_x, s_y) to a given point (g_x, g_y) it is a process of about 100 multiplications, which gives about 10^{-7} seconds of computation time on a state-of-the-art PC.

To verify that the algoritm works, three different environments are tested according to figure 3.5.

The testing is performed as: use $30x30$ uniformly spaced starting points and one goal point (marked with a red cross). For each starting point the output from the ANN is shown as a force vector arrow. The outcome is shown in figures 3.6 to 3.8.

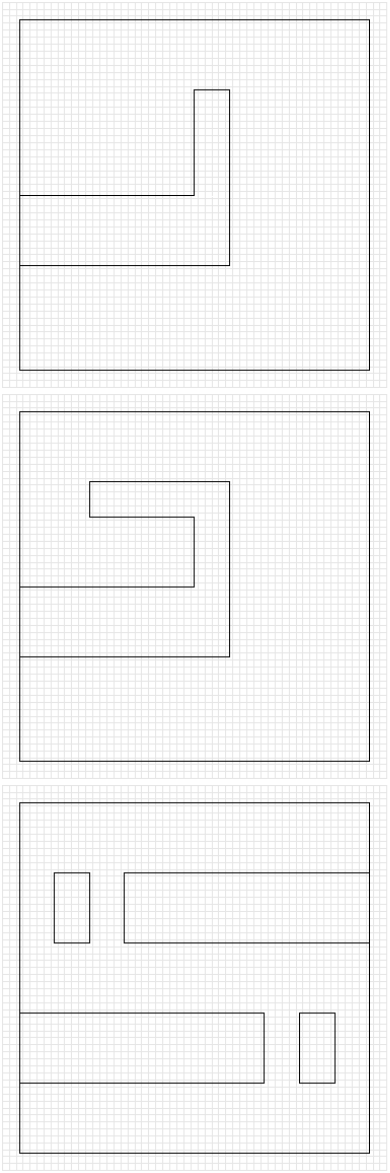

Figure 3.5: Test designs - Design 0, 1, 2

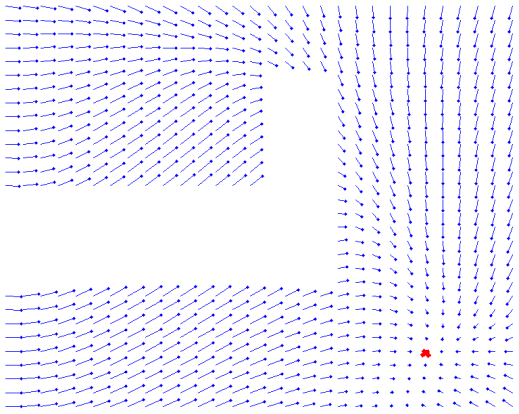

Figure 3.6: Test result - Design 0

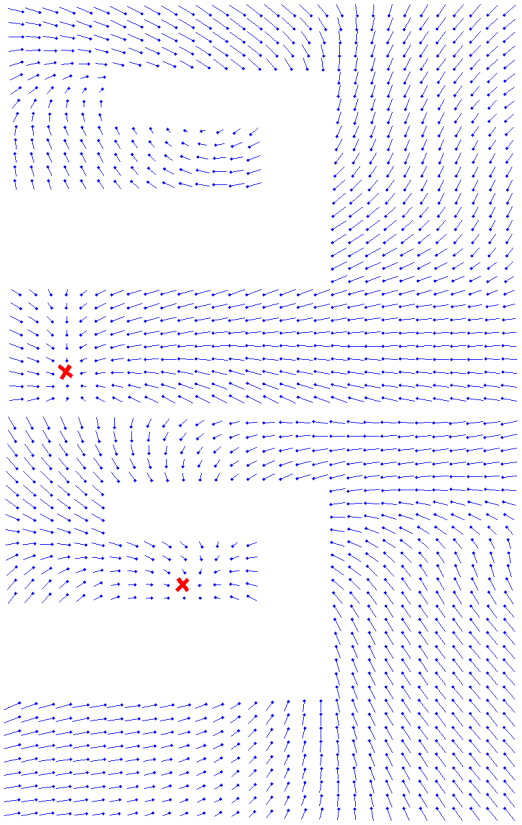

Figure 3.7: Test results - Design 1

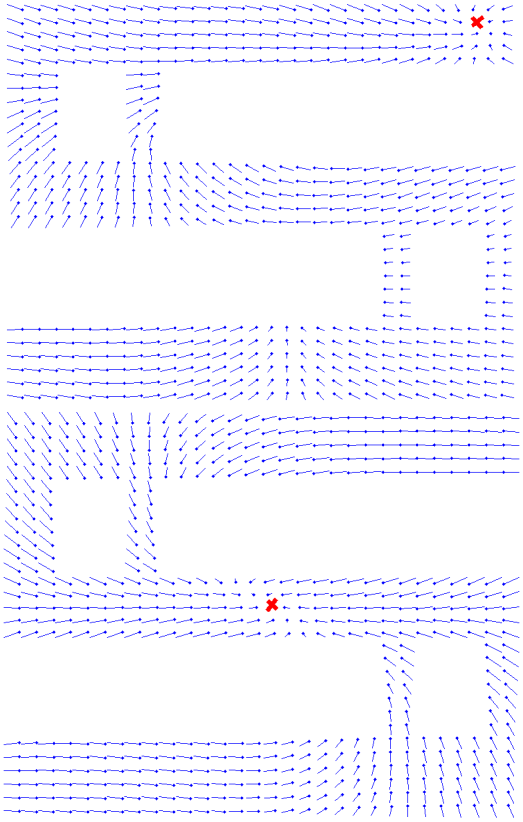

Figure 3.8: Test results - Design 2

3.3.1 ANN Parameters

There are basically two important parameters for back propagated artificial neural networks; the η parameter and the number of hidden neurons. The number of hidden neurons is quite easy to analyze, and is not described in detail here. It is rather easy to test how many hidden neurons are needed until the ANN fitness is reaching a saturated state. The η parameter on the other hand is a bit more tricky because it may have a constant value as well as a value that changes over time following a linear or quadratic scheme.

The η parameter is telling how much the neuron weights shall be changed proportionally to the error. Often a slightly higher value of η is used in the beginning of the training process, since the weights are only random values and need a lot of tuning, and then the value of η is decreased when the weights are nearing their desired values. In this case however the problem seems to be complex enough so that it is far from trivial to measure the current average error, so instead some empiric tests are performed to determine the fitness of different schemes of η. See figure 3.9 for the outcomes of the tests. In all tests the number of hidden neurons is 16, which seems to be a good value.

For the kind of environments used here, 16 hidden neurons and a constant value 0.8 of η seems to be a good parameter setup. This may however not be the case for all kinds of environments, so if maximizing the fitness is important this kind of analysis needs to be done separately for each scenario.

3.4 Potential Grids

If the number of goals is low and it is important to calculate the paths fast and accurate, another method is proposed, based on potential grids. For each goal do:

1. Create a grid of size $n * n$ and wrap around the infrastructure.
2. For all cells, i, within the current goal area:
$$V_i = 0$$
3. For all cells, i,
   For all eight nearest neighbor cells, j:
$$V_i = min(V_i, V_i + \Delta V_{ij})$$
4. While changes made in step 2), goto 2).

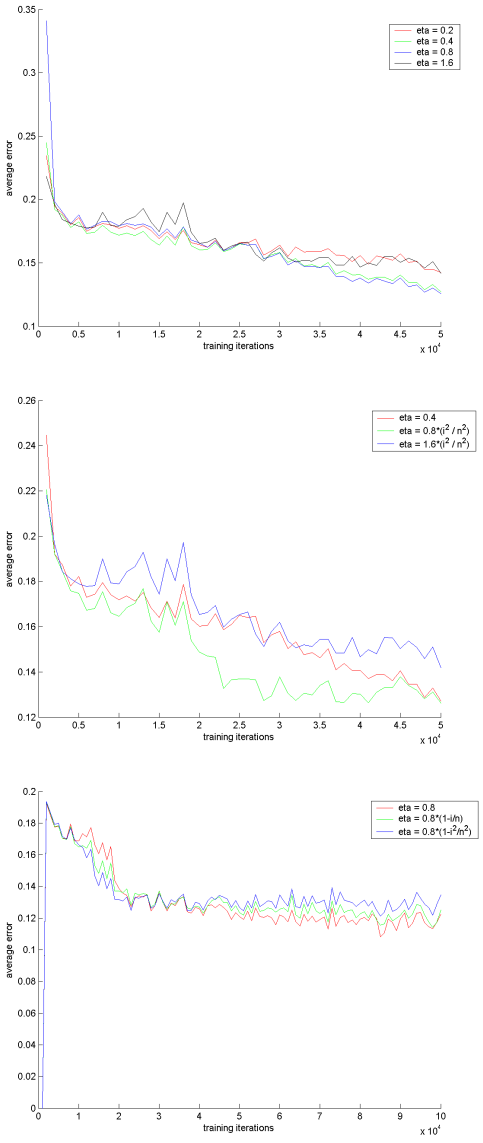

Figure 3.9: η parameter tests

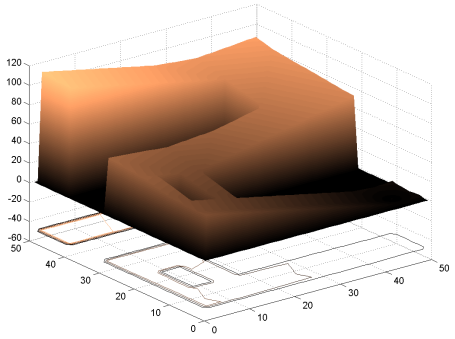

Figure 3.10: Potential Grid

where ΔV_{ij} has three possible values:

- ∞, if there is an obstacle between cells i and j.
- 1, for neighbors above, below, to the left and to the right.
- $\sqrt{2}$, for the four diagonal neighbors.

After the grids are created the procedure to find the goal areas is just to follow the steepest descent in the potential grid. An example of a potential grid is shown in figure 3.10.

Note also that the process can be speeded up significantly by first creating a matrix that keeps track on which neighborcells are connected and which have obstacles in between.

3.5 Summary

A few different path finding approaches have been discussed. It is however impossible to pick one and discard the others since different approaches are best suited for different scenarios. For example if a large and complex infrastructure is to be simulated and the number of goals are low (not more than 100), the potential fields are the best approach. On the other hand if the number of goals are high, or maybe even infinite (arbitrary picking of goal positions in empty space) the approach based of the artificial neural networks may be a better suited one. If the initialization time must be kept short the Floyd-Warshall-based algorithm may be the best one, and finally, for very simple infrastructures, some times path finding is not even needed,

so it is enough with just the driving force pointing to the desired endpoint of the trip.

Chapter 4

Real-Life Applications

4.1 Soccer Stadia

Evacuation of soccer stadia may be dangerous since they are often constructed in a way that creates peeks in the distribution of density and pressure at specific locations (figure 4.1). Therefore an improved construction is proposed according to figure 4.2. The purpose of the zig-zag pattern is to even out the pressures at different locations during long stairways.

Two different kind of exits are evaluated according to figures 4.1 and 4.2.

4.1.1 Evolved Setups

In order to make an objective investigation of how the infrastructure parameters are influencing the overall fitness, two evolutionary processes are started. The changable parameters are the width distributions and the sideways displacement of the stairways sections. One of the processes has the task to maximize the throughput flow and the other one tries to minimize the density calculations shown above, i.e. to even out the density as much as possible. The outcome of the single best evolved setup from each evolutionary process are shown in figures 4.3 and 4.4. It is obvious that the zig-zag pattern is a well-choosen safety-increasing solution. The straight stairway has a slightly better throughput, but on the other hand if people are starting to panic, it is a well known property of crowds that high densities together with panic is a worst case scenario also for the throughput. Therefore it makes sense to choose the zig-zag solution even if throughput are the most important factor.

Starting with the improved setup in figure 4.2, two evolutionary processes are started:

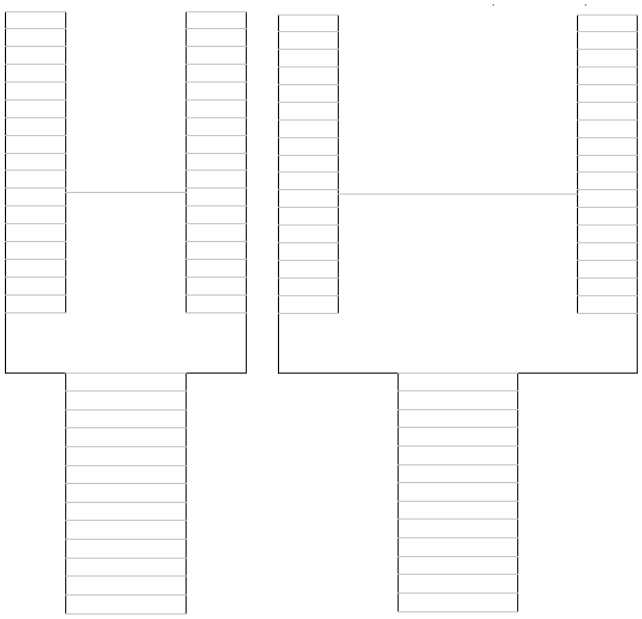

Figure 4.1: Conventional setups

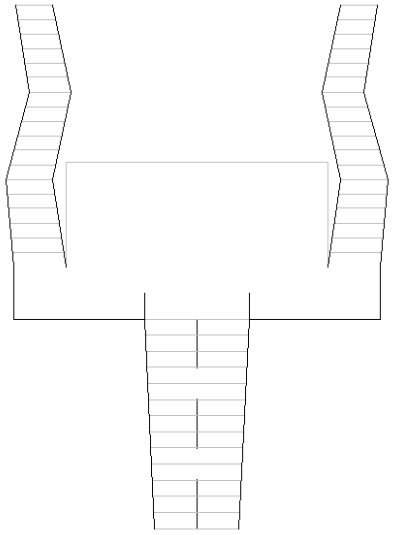

Figure 4.2: Improved setup

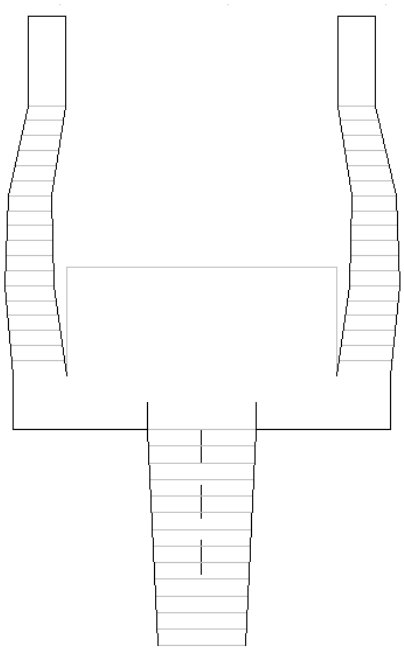

Figure 4.3: Best evolved infrastructure for maximum flow

1. One trying to find the parameters according to the maximum flow setup.
2. Another one trying to even out the density.

Maximum Flow

The only parameters changed are the ones corresponding to the downward zig-zag stairways at the sides of the exit. This gives a total of six parameters; three horizontal displacements and three widths. Symmetry is used since it makes no sense to evolve different kind of stairways on different sides of the exit.

The first evolved infractructure setup is trying to find out how to maximize the pedestrian flow out of the exit. The best result is shown in figure 4.3, and not very suprisingly the fastest stairways are very straight.

Well Distributed Density

As an approach to a pedestrian safety parameter the following scheme is used:

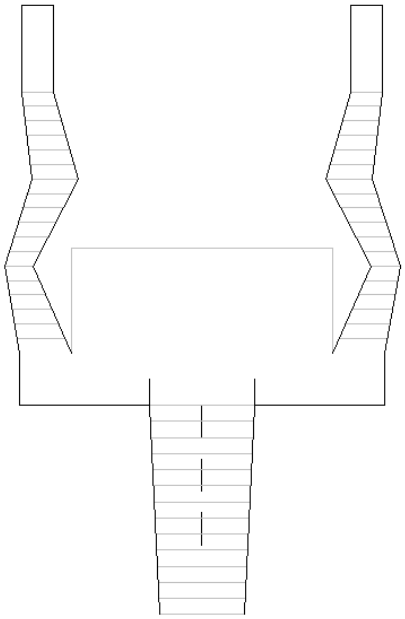

Figure 4.4: Best evolved infrastructure for even density

- A grid of 100x100 cells is wrapped around the environment.
- A function, f, is averaged over all nonempty cells i according to:

$$f_i = {\rho_i}^2 \tag{4.1}$$

$$f_{mean} = \frac{1}{n} \sum_{i=1}^{n} {\rho_i}^2$$

  Where ρ_i is the local density in cell i.

If f_{mean} is low it means that the density is evenly distributed. If the density is really high at some places, the square function will result in a bad total fitness of that setup and the evolutionary process will discard that solution.

The best evolved setup with the new density measure is shown in figure 4.4, and the zig-zag is more pronounced this time, as expected.

Statistics of Evolutions

The evolutionary history of the fitness as a function of the number of tournaments in the Genetical Programming process is shown in figure 4.5.

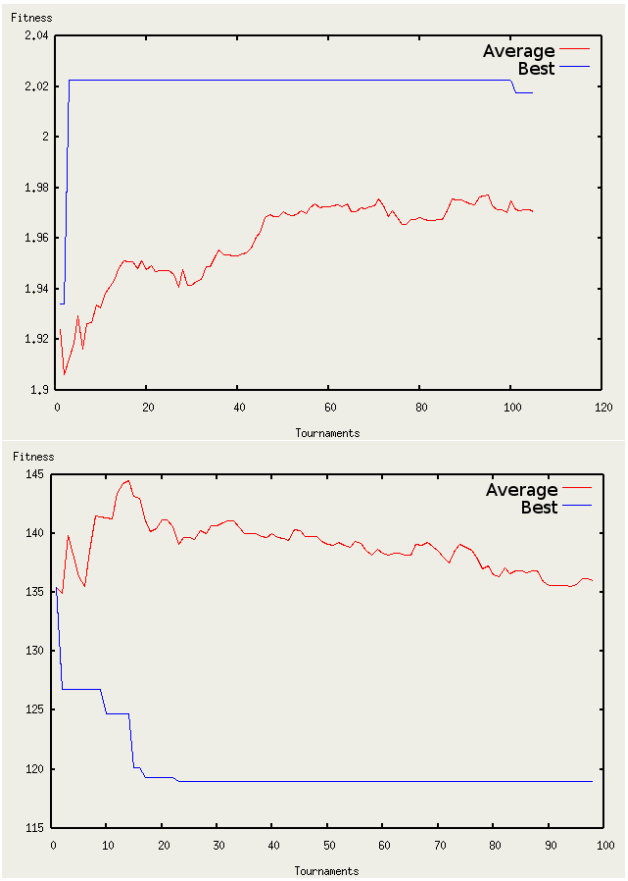

Figure 4.5: Evolution of: 1) Flow, 2) Density

A Large-scale Simulation

A large-scale simulation is made with about 3000 pedestrians to be evacuated via one single exit (figure 4.6). First the throughput of the exit are measured for a regular evacuation (figure 4.7a). Then a new simulation is made but with an obstacle placed at the endpoint of one of the side exits, to investigate what will happen if one of the sideways are blocked for some reason during an evacuation. This time it is more obvious why the zig-zag shapes are needed, since the pressures at the end of the corridor for the improved infrastructure is decreased by a factor 2. The reason for this is that the pressure is focused of each of the five corners in the corridor and not at one single point in the end of the corridor as in the conventional case. Figure **??**b shows the average pressure at the endpoint of the blocked corridor as a function of time.

4.2 Jamarat Bridge

Every year several millions of Pilgrims are walking over the Jamarat bridge in Mekka for the 'stoning the devil' ritual which is about throwing peddles at each of three different pillars. Due to the heavy usage of this bridge under a short period of time, several people are crushed to death every year. To investigate how this can be solved one can simulate the stoning process on the Jamarat to be able to analyze it. Figure 4.8 shows a simulation of the bridge where the coloring denotes the pressure level. Not very surprisingly the pressure is much higher near the pillars.

4.3 Urban Simulation

A simple CAD program is constructed (figure 4.9) which makes it possible to load a background image file and then outline the borders manually. This makes it easy to simulate for example an urban part of a city (figure 4.9). Obstacles are drawn in form of lines where the walls of the buildings are located and then a second functionality makes it possibly to add gates at the buildings just by clicking the mouse near the border. When the CAD program is terminated the current map is converted into XML format and exported to a file. After this process the XML data can be used to perform simulations. For example to investigate the impact of the city core when a new shopping mall is added or an old one is removed.

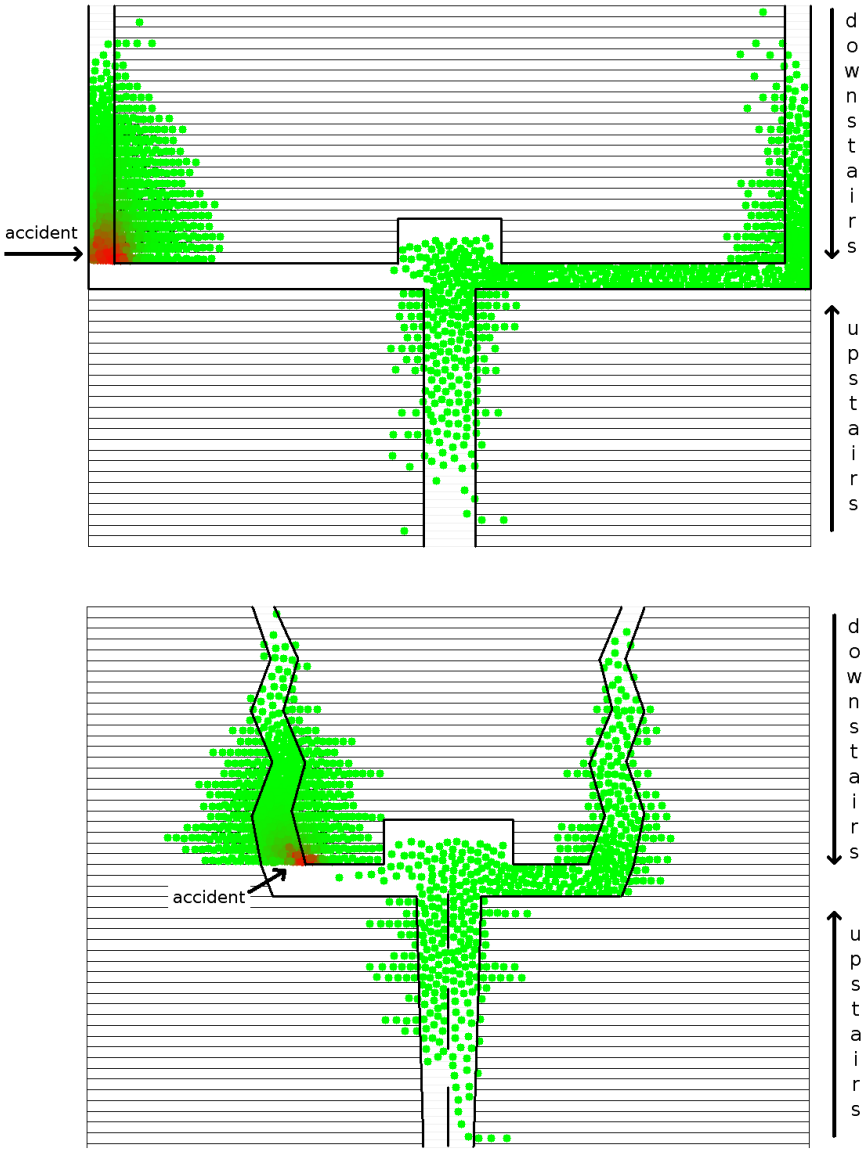

Figure 4.6: Soccer stadia evacuations - Top: Conventional infrastructure. Bottom: Improved infrastructure

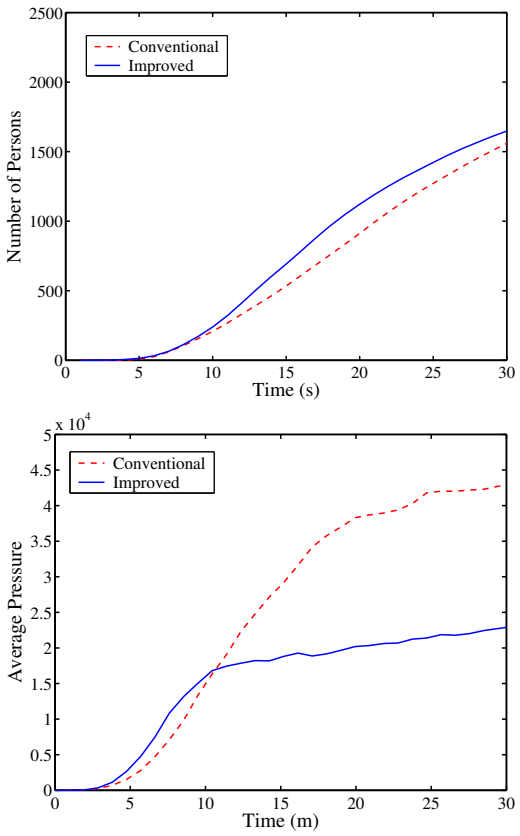

Figure 4.7: Soccer stadia evacuation - Top: Throughput. Bottom: Pressure

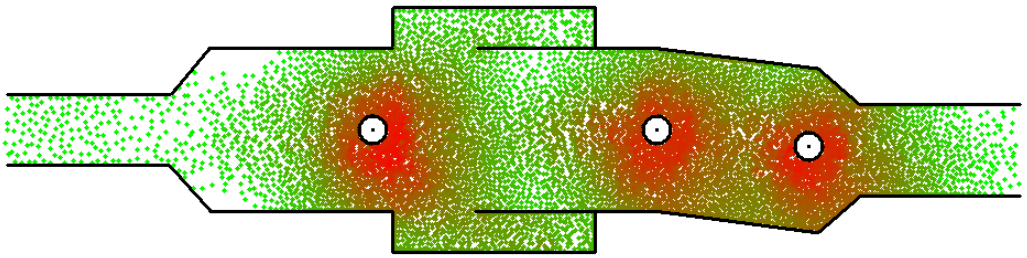

Figure 4.8: Jamarat Bridge

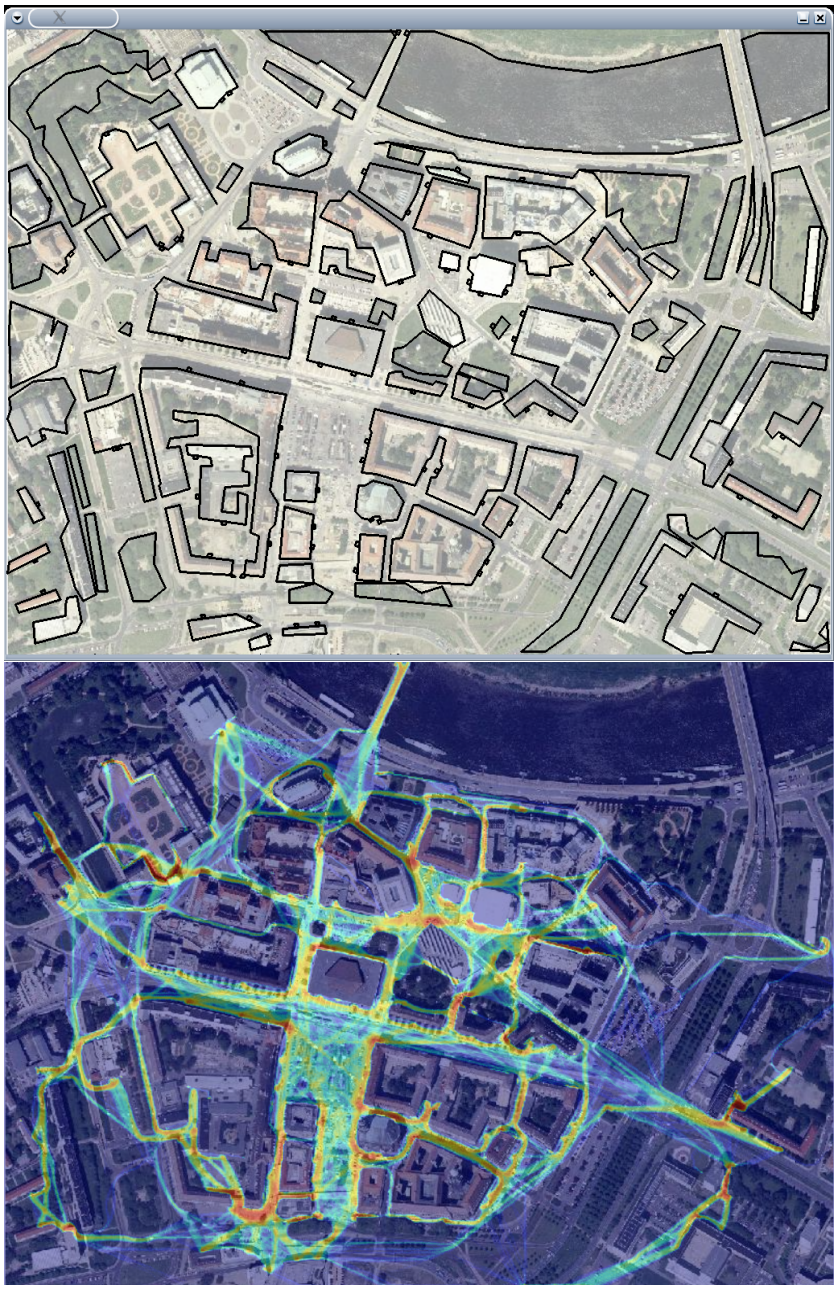

Figure 4.9: Urban simulation of Dresden: Top: CAD program. Bottom: Density distribution

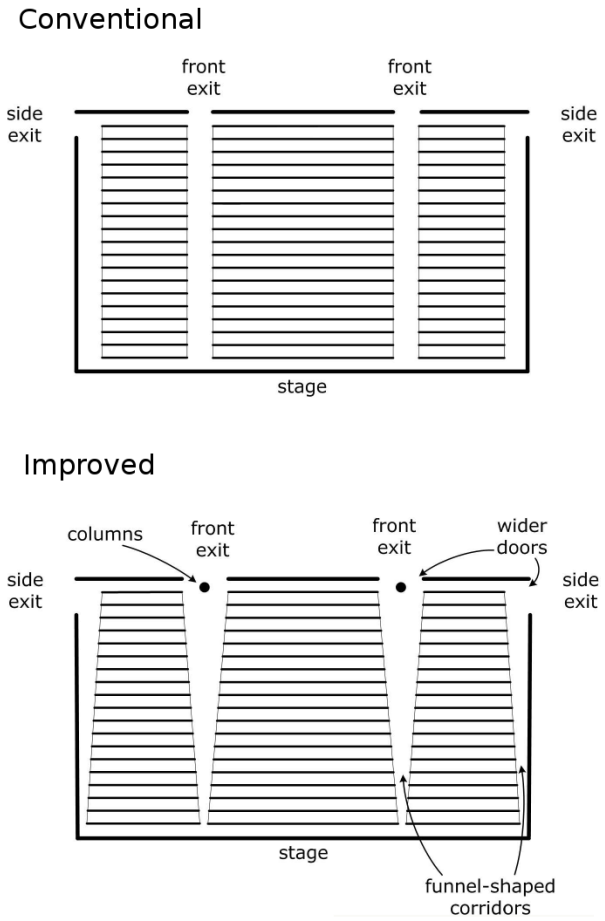

Figure 4.10: Theater infrastructure

4.4 Theater

A theater is simulated and analyzed both with its current conventional infractructure as well as a preposition for an improved infrastructure (figure 4.10). The preposition for an improved infrastructure results in faster evacuations as shown in figure 4.11.

A new batch of simulations are made where the leaving times are measured for each pedestrian and the leaving times are shown on a geometric map connecting the leaving times to the original seating position of the pedestrians. Figures 4.13 and 4.14 shows the results for the conventional and the improved infrastructure respectively. The purpose of these simulations is to see how the leaving time is correlated to the seating row number. It turns

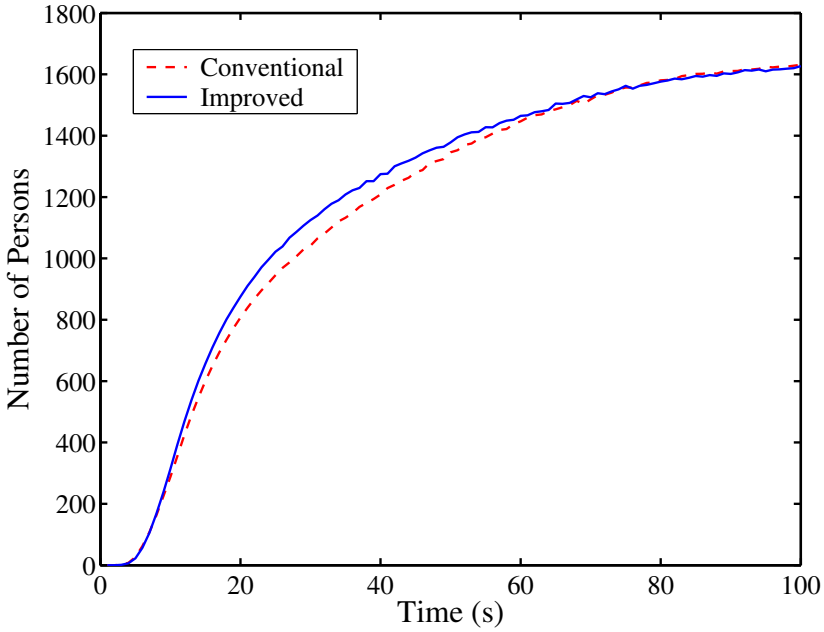

Figure 4.11: Theater evacuation

out that the leaving time is proportional to the seating row, in both cases. Since the main bottlenecks are the end-points of the seating rows, the evacuation time as a function of the row number is approximately linear in both cases. It is also clear from observing the simulations that once a pedestrian has reached the corridor it is fast to just follow the flow out of the room, but the tricky part is to get from the seating rows to the dense corridors. Note also that the pedestrians in the back (high row numbers) have less problem of getting to the corridor since the pedestrian densities in the corridors are decreasing for increasing distances from the exits.

4.5 Pedestrian Crossing

A conventional crossing for pedestrians is shown together with a preposition for an improvement, in figure 4.15.

4.5.1 Railings

A first step of improvement is to add railings to decrease the level of chaos occuring for counterflows with high density. The improved crossing is sim-

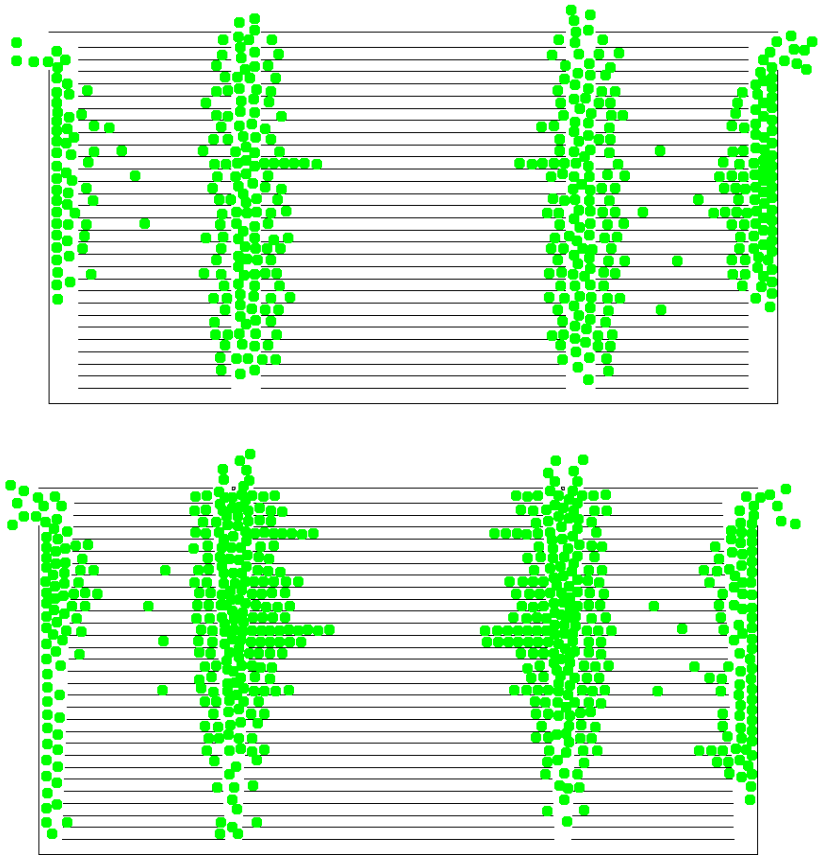

Figure 4.12: Theater simulations - Top: Conventional infrastructure. Bottom: Improved infrastructure

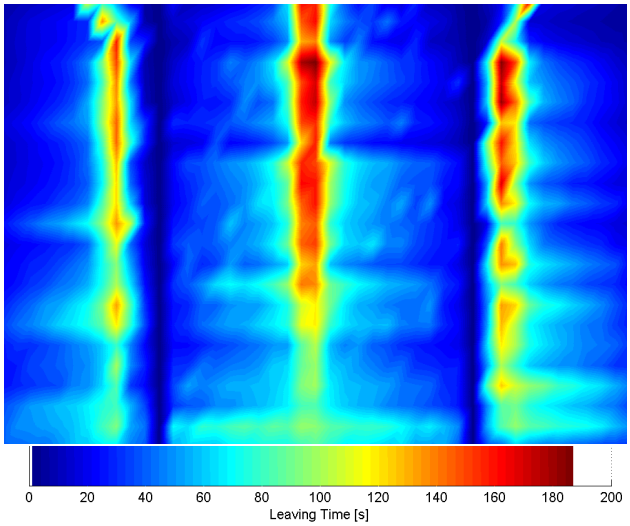

Figure 4.13: Leaving times - Conventional setup

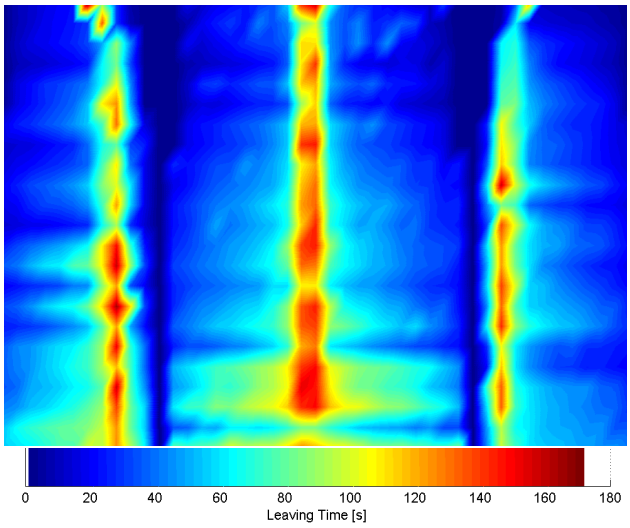

Figure 4.14: Leaving times - Improved setup

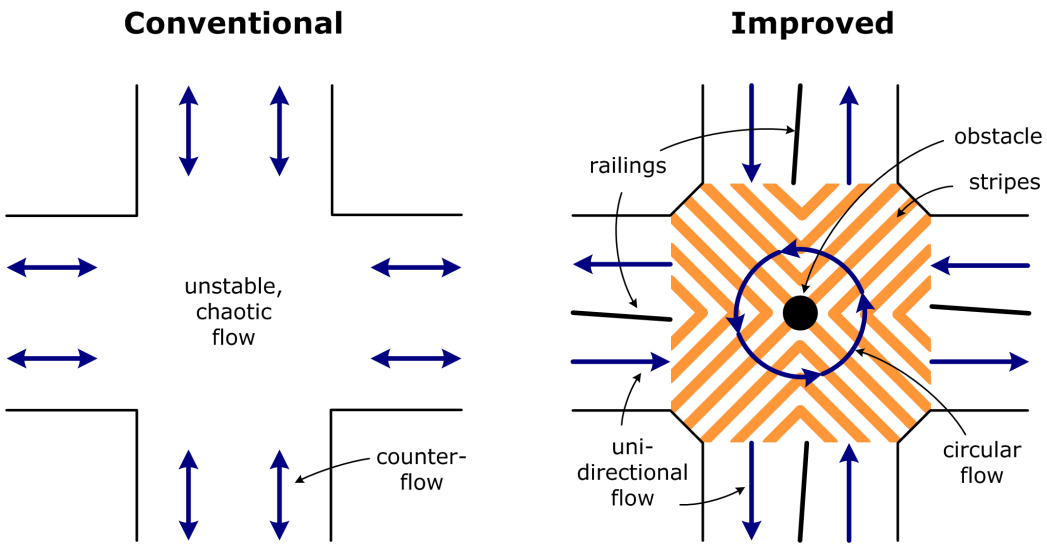

Figure 4.15: Pedestrian crossings

ulated during 120 seconds (in simulation time domain) and the number of successful pedestrian passings per second is used as a fitness measurement. The standard parameter setup is shown in table 4.5.1.

Variable	Value	Description
w	4.0 m	Road Width
r	0.4 m	Central Pillar Radius
dx	0.7 m	Middle section enlargement
dw	0.5 m	Railing displacement

Three different simulations are made: One with railings, one without railings and one with permeable railings (five stripes per railing). 32 different values of the railing displacement are iterated to find the optimal value. The outcome of the simulations is shown in figure 4.16 and the three different simulations are shown in figure 4.17.

4.5.2 Radius

Next, the radius of the pillar, in the center, is changed to find an optimal value. The outcome of this test is shown in figure 4.18.

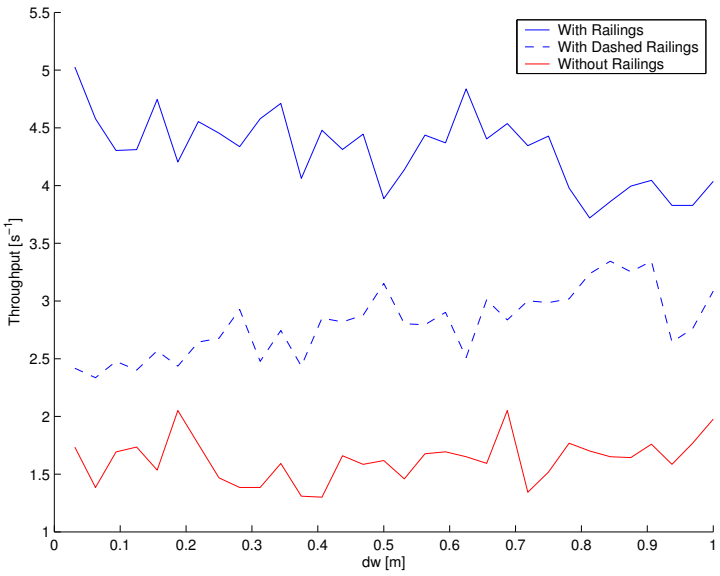

Figure 4.16: Railings

4.5.3 dx

The enlargement of the middle section, dx, is analysed next, with simulations of 64 different values. The results are shown in figure 4.19.

4.5.4 Both r and dx

To verify the previous results a long range of simulations are started with iterations of both dx and r simultaneously. The results are shown in figure 4.20. The road width in the middle section seem to be the important parameter, according to figure 4.21

4.5.5 Summary

Not very surprisingly the railings is an effective improvement since the densities are high and the railings are decreasing the level of chaos. Also, the efficiency is increased by increased widths of the central section. However, it is clear from figure 4.21 that it is a bad idea to make the routes too wide in the central section since the chaos level will be increased and at some point the total throughput is actually decreased by increasing the middle section further.

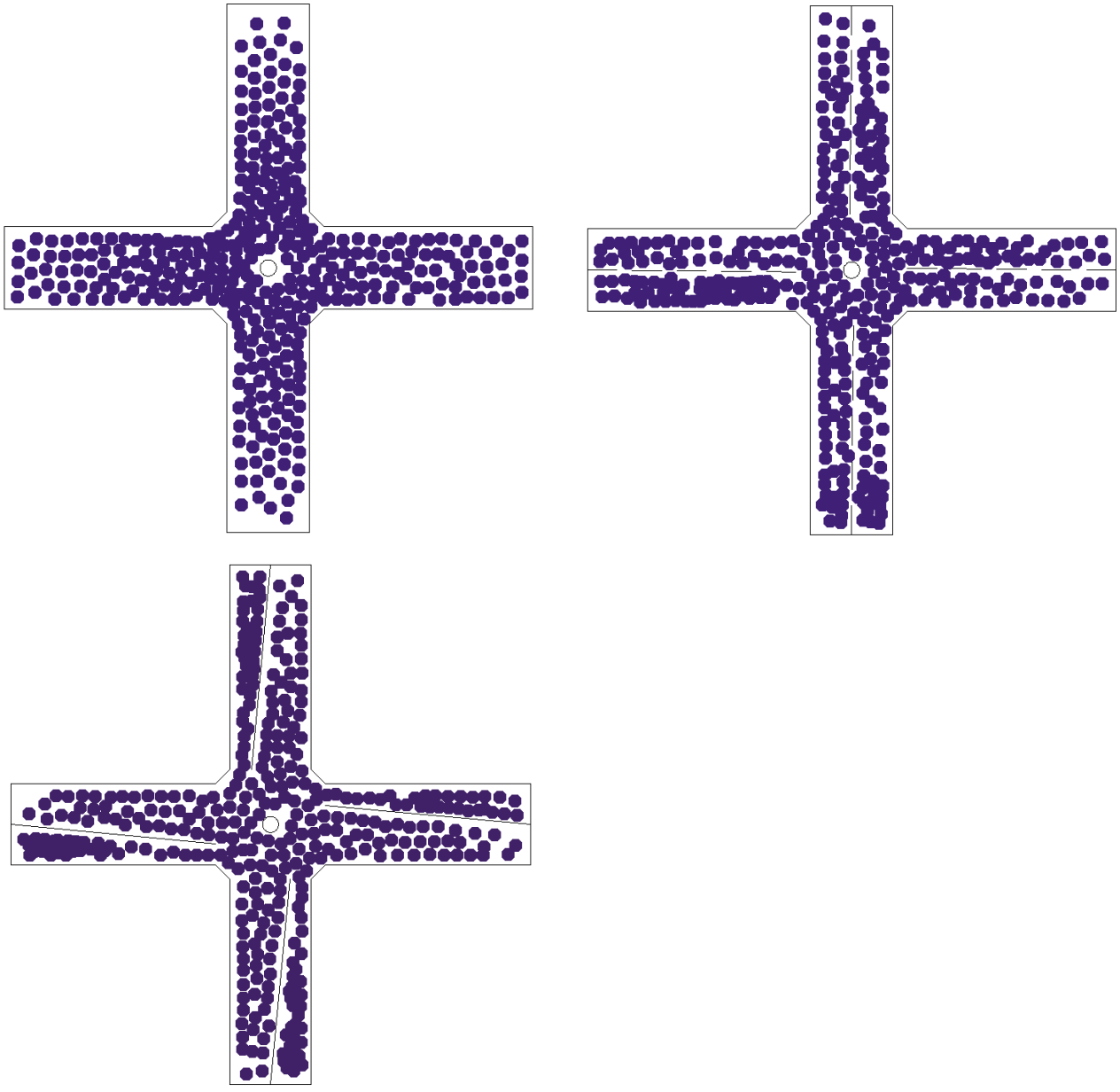

Figure 4.17: Different infrastructures: Without railings, with permeable railings and with normal railings

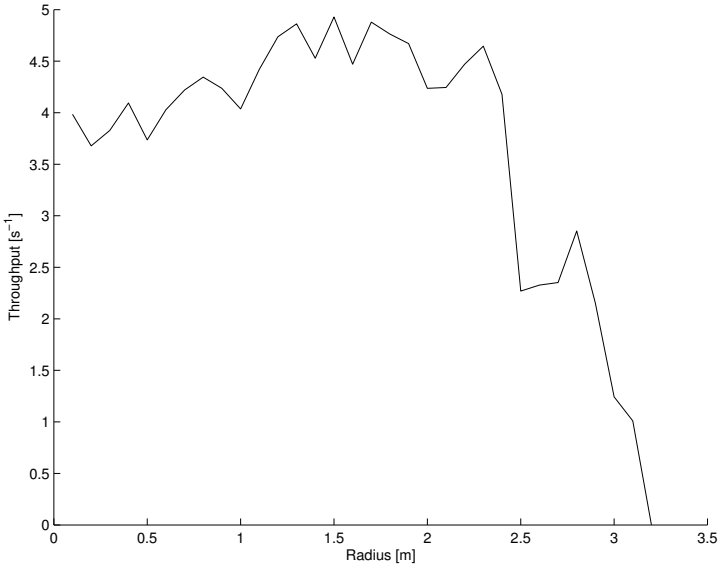

Figure 4.18: Radius iteration

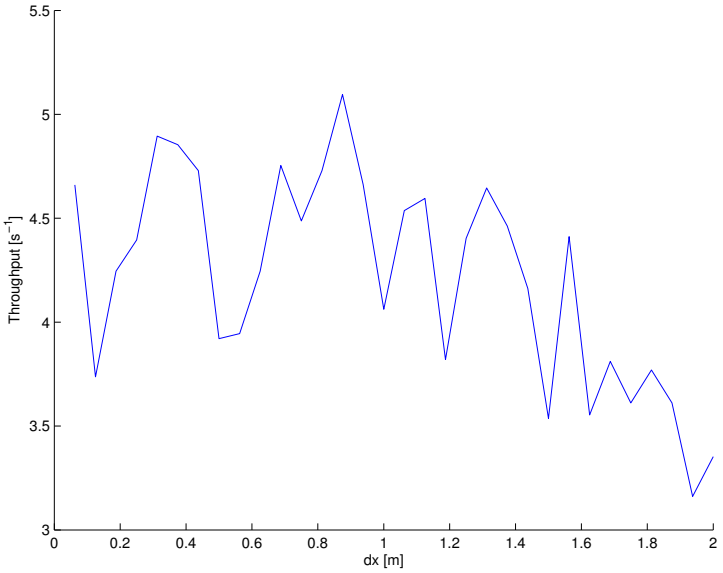

Figure 4.19: Iteration of the size of the middle section

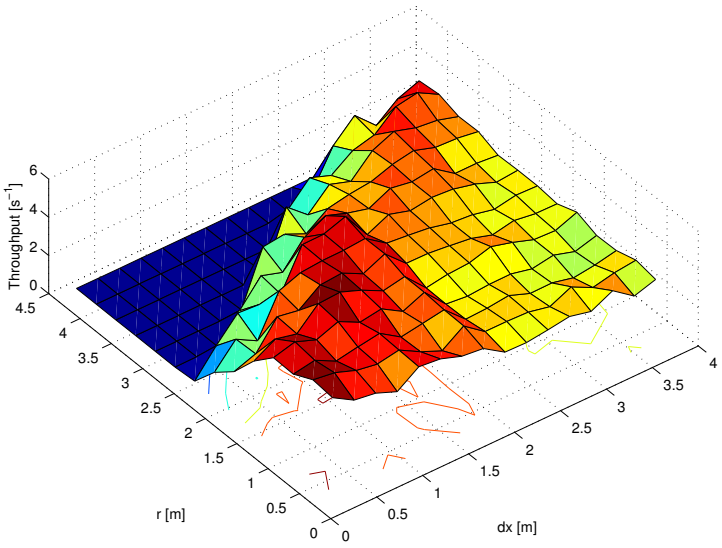

Figure 4.20: Iteration of r and dx

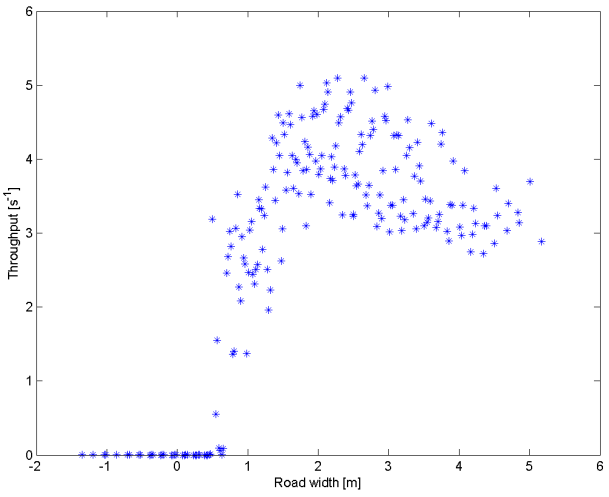

Figure 4.21: Iteration of r and dx

Chapter 5

User Interface

An important component of the simulation software is of course the user interface and the basic demands of this are:

- Simplicity - easy to define and test new simulation scenarios.
- Scalability - the same user interface has to work for basic simulations as well as for large and complex ones.
- Automation - It has to be possible to define series of simulations and measure the fitness according to some parameter.

The choosen way to create the user interface is to use XML files as input. XML is a wide-spread standard and there are a lot of free and open-source parsers to use. Another positive thing with XML is that it simplifies the process of exporting CAD files to environments suitable for pedestrian simulations. The root item is called `<simulation>` and has the following properties:

- *description*: Defines a descriptional text of the simulation.
- *max_time*: Defines for how many seconds (in simulation domain) the simulation should be performed.
- *dt*: Redefines the timestep. The default value is 0.05 s.
- *fps*: Specifies how many frames per seconds (in simulation domain) to show. Default value is 20.

The other defined basic items are:

- `<obstacles>`
  with subitem `<obstacle>` with elements $x0$, $y0$, $x1$, $y1$, which defines a line obstacle from point $(x0, y0)$ to point $(x1, y1)$.

- `<rooms>`
  with subitem `<room>` with elements $x0$, $y0$, $x1$, $y1$, which defines an area.

- `<agents>` with subitem `<agent>` with elements *advanced*, *id*, $freq$, $x0$, $x1$, $y0$, $y1$, *task*, *color*, v
  where $x0, y0, x1, y1$ specifies an area where the agents are born, *task* is the moving pattern, $freq$ specifies the average number of born agents per second, *advanced* determines whether the agent will use a path finding algorithm or simply an attractive force towards the goal area. If $x0$, $y0$, $x1$, $y1$ is omitted the property *born* may be added which specifies that the agent shall be born within the corresponding goal area instead. *id* is used for the statistics to separate different groups of agents. *color* may have the values r, g, b (red, green blue in interval [0..255]), *pressure* (green for low pressure and red for high pressure) or *nervousness*, for similar color coding of the nervousness. v is the desired velocity, either in form of a scalar value or $N(mean, variance)$ for a normal distribution.

- `<gates>`
  with subitem `<gate>` with elements $x0$, $y0$, $x1$, $y1$, which defines a line where a gate is created between two rooms.

- `<goals>`, with any of the subitems:

  - `<goal>` or `<rectangle>`, with elements $x0$, $y0$, w, h, nx, ny, *type*, *border*, which defines $nx * ny$ goal points on a grid, with a border of strength *border* in range [0..1]. The *type* is specifying an id number that is used in the *task* property for routing the agents.
  - `<circle>`, with elements x, y, r, which specifies a circular goal area with origo (x, y) and radius r.
  - `<polygon>`, with elements x, y, which specifies a polygon shaped goal area within the corner points specified as comma-separated lists.
  - `<complex>`, which is a complex goal area which may have any of the other goal types as subitems.

- `<layers>`
  with subitem `<layer>` with elements *name*, *enabled*, *visible*.

- `<schedule>`
  with subitem `<event>` with elements *object* (name of a layer), *action* (with one of *hide*, *show*, *enable*, *disable*, *set_location*, *set_velocity*), *time* (time offset to execute event) and *interval* (time period when event is occuring again). In case of *set_location*, parameters x and y specifies the new location and in case of *set_velocity*, parameters vx and vy specifies the new velocity in which the layer is starting to move.

5.1 Advanced Items

5.1.1 Agent Task

The `task` property, for the `agent` item, is of the form `a@b c@d e@f` where `b`, `d`, `f` are the goal types and `a`, `c`, `e` are the number of goals to visit. An example: `task="1@2 70%@5 all@8"` means

1. Visit one goal of type 2
2. Visit 70% of the goals of type 5
3. Visit all goals of type 8

Additional parameters may be added to each task, in brackets:

- *mode*, with values *nearest* (chose the nearest goals) or *random* (choose goals randomly).
- *wait* with values $N(mean, variance)$ or *scalar*, which means that the agents should wait at the specific goal for a specific time; either a fixed time or time picked from a a normal distribution.

For example to go to the nearest goal of type 3 and wait for ten seconds, `task="1@3[mode=nearest,wait=10]"`.

5.1.2 Parameters

The `<parameters>` section with subitem `<parameter>` is a constant with properties *name* and *value*, that can be used to simplify the process of rescaling and generalizing the environment. The parameters can be used as they are or in equations for all other values in the XML file. For example if a parameter *height* is specified, then the $x0$ property of an obstacle can be set to $height/2 + 10$.

5.1.3 Statistics

The `<statistics>` item specifies the different kinds of statistics to extract from the simulation. All statistics is created in a format that can be understood by Matlab and Octave, which means that the statistics may easily be plotted as it is and also be used for more complex analysis after the simulation is completed. The available statistics types are:

- *velocity*: Measures the velocity of pedestrians along the x and y axes.
- *density*: Measures the density of pedestrians.
- *count_lanes*: Counts the number of lanes of pedestrians.

All statistical elements are sharing the same properties:

- $x0$, $y0$, $x1$, $y1$: The area where the statistics is performed.
- $filename$: The filename where the statistics will be saved.
- *resolution*: A value that determines the length of the time cycle to average the statistics over.
- *index_start*, *index_stop*: Specifies a range of pedestrians to apply the statistics on.

5.1.4 Iteration

Often it is interesting to see how the fitness of a specific environment depends on a specific parameter. Therefore the *loop* item is used to iterate some parameters. The properties are:

- *parameter*: The parameter to be iterated
- *steps*: Number of steps n, to run the simulation. The parameter will have values 0..n-1 during the simulation batch.

There is one subitem `<parameter>` with properties:

- *name*: Name of parameter.
- *value*: Value of parameter, which is recalculated for every loop step.

5.1.5 Evolutionary Algorithms

An item with functionality close to *loop* is defined with name *evolve* which performes genetical programming trying to find the optimal configuration of a set of parameters.

The properties of `<evolve>` are:

- *data*: A file name for saving the data (for example to continue the evolution at some other time if it is stopped).
- $fitness$: Determines how to measure the fitness of the current setup. Two different values are valid: $satisfied$, the number of pedestrians that have performed all their tasks successfully, or $density_spread$, a value denoting how well the density is distributed.
- *method*: With values *min* or *max* which determining whether a big or a low fitness value is the best.
- *population*: The number of individuals in the gene pool.
- *genome*: Number of genes (parameters).
- *initiate*: *yes*/*no*; Determins whether the gene pool should be reset or loaded from the data file on startup.

The subitem `<parameter>` defines a gene in the gene pool with the properties:

- *name*: Name of the parameter.
- *min*: Minimum value.
- *max*: Maximum value.

The parameter *min*, *max* values can also be dependent on the previously defined genes.

5.1.6 Modules

The item `<modules>` makes it possible to attach several modules to add functionality to the simulations. These subitems are defined:

- `pathfinding`: with properties *enabled* ($true$ or $false$), *type* (*grid* or *network*), *congestion_avoidance* (to add repulsiveness of densed areas with a certain magnitude). If $type = grid$, nx and ny specifies the size of the grid.

- `force_separation`: with property *enabled*.
- `output`: with property *type* (*png* or $flash$), which exports the simulation either as single frames in PNG format or as a Flash movie.
- `behaviour`: with property *type*:
  - *panic*. To add stochasticity.
  - *nervousness*. To add nervousness, which makes pedestrians increase their desired velocity if they are getting stuck for a while.
  - *herding*. To add herding behaviour.
  - *immobility*. To add functionality that makes the pedestrians more immobile as a function of the pressure on them.
  - $social\_force$. Which is used to change the magnitude of the social force.

  All behaviours have the property *magnitude* which is a scalar that tells how much influence the behaviour should have.

5.1.7 Macros

To be able to specify complex scenarios without needing to hardcode every part, an item `<macros>` is used to automate the creation of agents and obstacles. The only parameters defined are $i = n_i..m_i$, $j = n_j..m_j$, $k = n_k..m_k$. The subitems may be defined, of type `agent` and `obstacle` which are basing their parameters on the i, j and k values, which is one, two or three nested loops. Only i is needed to be specified and then j and k are optional.

Chapter 6

Pedestrian Behaviour in Urban Regions

6.1 Introduction

Traditionally, pedestrian simulations most often is based on one specific task; evacuate a building as fast as possible etc. To model a shopping mall, a market-place and similar, pedestrians behave a lot less irrational and to model this a more sophisticated scheme than shortest path finding to some fixed targets in space is needed.

To be able to mimic the scenario above, some semi-contradictional points need to be concidered:

- The pedestrians want to be where the other pedestrians are but do not want to be in too crowded environments.
- The pedestrians may have specific goals but may still be able to find other attractions during the way.
- The pedestrians want to explore new areas but when they have been at a place for some time they get bored and want to explore some other places.
- And at last; probably the contradiction most hard to solve is the one between an accurate model and also something that is relatively fast to compute to be able to do fast simulations with a few thousand pedestrians.

A model is created which takes all these points into consideration.

6.2 Model

An approach is made influenced by the technique of *Cellular Automata*, but with a few important differences. Initially a group of matrices are initialized, all of them mapped to a grid spanning the space where the simulation takes place:

- *attractionsGrid*: A matrix where each element tells how attractive the location is; i.e. how many shops, kiosks, information desks, etc, are located in the region.
- *densityGrid*: A matrix with pedestrian densities.
- *pathFindingGrid*: n matrices, one for each goal, that consist of a potential grid to find each goal (Hoogendoorn and Bovy, 2002). See figure 6.1 for an example how it is used.

Also, two matrices are set up to speed up the calculations:

- *distancesGrid*: Holding the distances between all cells to all other cells.
- *directionsGrid*: Normalized vectors that points from each cell to each other cell.

Each pedestrian has a property *desiredDensity* which is the density of the environment that is optimal for the pedestrian.

The *bias* is a parameter which can be tuned to make the pedestrians more or less restless. With a high bias the pedestrians can be a long time at the same location without getting bored, and vice versa. The division by $distancesGrid_{i,j}$ is to make cells further away less interesting and the $|densityGrid_j - desiredDensity|$ part is to make the areas with desired amount of people more interesting.

Each pedestrian may have one or several goals that need to be visited with different priorities. The update scheme for the main part of the driving force is:

- Choose the goal in the list of goals that minimizes $distance(goal_i)/priority(goal_i)$
- Walk towards the negative gradient from the current position on the pathfinding grid corresponding to the active goal.

If some of the tasks have deadlines so that the pedestrian need to arrive at the goal before a certain time, this can be modeled by increasing the priority of that goal, with a maximum priority when the theoretical walking time is

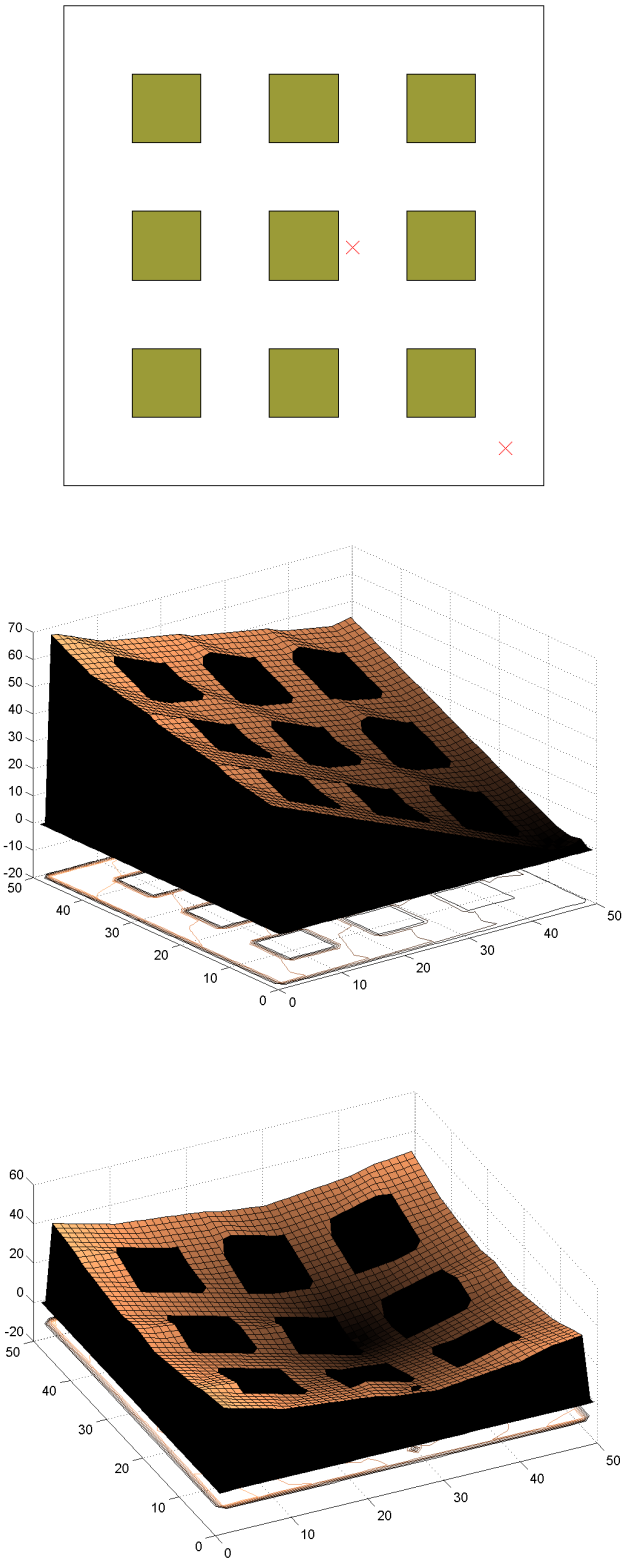

Figure 6.1: Pathfinding with Potential Fields

near the remaining time to the deadline. This will have the effect that if a pedestrian is walking towards goal A and the urgency to visit the higher prioritized goal B is increased above the threshold value, he may turn around completely before reaching A and go directly towards goal B.

The characteristics of the function to compute the utility gain of moving from cell i to cell j, looks like:

$$f(i,j) = directionsGrid_{i,j} * \frac{attractionsGrid_j * e^{k|densityGrid_j - desiredDensity|^2}}{distancesGrid_{i,j}}$$

The choice of which cells to average over to update cell (x, y) can be done in different ways:

- The conventional Cellular Automata way: Use the nearest neighbors $(x, y) \in [x - n..x + n, y - n..y + n]$: This approach is slow and also unrealistic because the area of attention for a pedestrian is far bigger than just the very nearest surroundings.
- Find the direction corresponding to the maximum utility, by evaluating all cells: This is an exact method but it is even slower.
- The third method is to just pick n numer of cells at random and average over these. This method is the most general one and also creates fair distributions of choices while keeping the computational speed high.
- If the simulated pedestrians are considered to be rational and the ability to visit the desired goals as fast as possible is the only important thing it is however suitable to only go towards the steepest downhill gradient in each step, and the movements will be deterministic until the highest prioritized goal is changed for some reason.

The last important task is to find a good scheme of updating the matrices. The trick is to smoothen the matrices so that the discretization of space will become less obvious. For example if the pedestrian has been a lot in one grid cell he probably got tired of the nearest neighbor cells as well. figure 6.2 shows the positions of some attractions and the corresponding smoothened attraction grid.

6.3 Discussion

It is important that the ability to search for optimal crowdedness is a minor part of the total driving force. If this is not the case unrealistical heavy turbulence may occur.

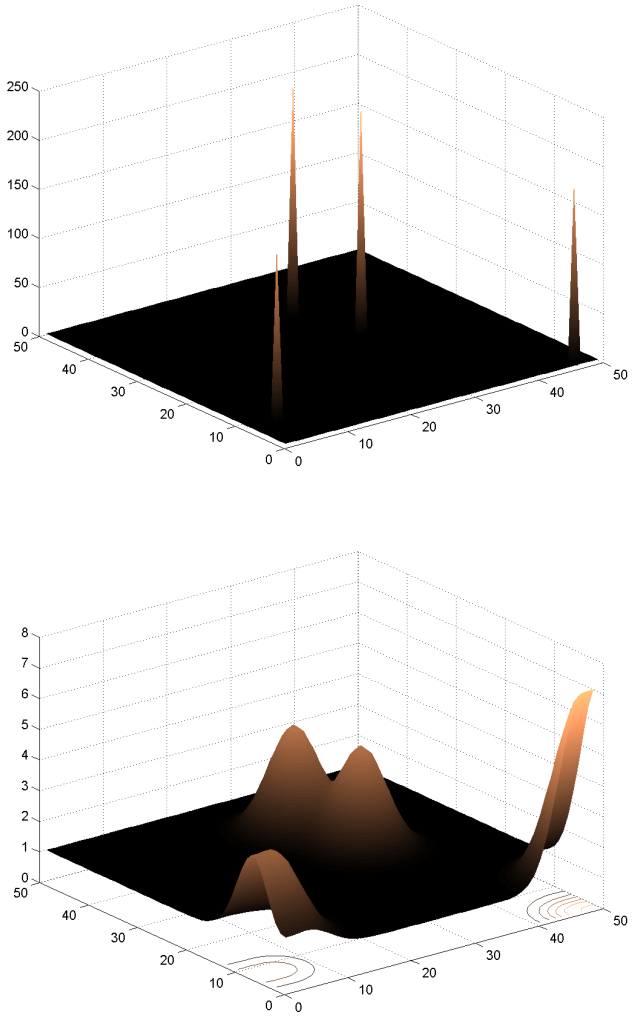

Figure 6.2: Attractions Grid - Top: Original. Bottom: Low-passed

Acknowledgements

The author is thankful to Dirk Helbing for his supervising and proposed design solutions for soccer arenas, theaters and crossings. Further credits goes to Martin Treiber for fruitful scientific discussions of the implementation of the model, to Mikael Persson for his help with LaTeXand related software and to Stefan Lämmer for the illustrations of the theaters and crossings. Finally, special thanks goes to Karl Walkow for his help during the implementation of the simulation software.

Bibliography

[1] S. A. H. AlGadhi, H. S. Mahmassani and R. Herman, "A speed-concentration relation for bi-directional crowd movements with strong interaction", in Pedestrian and Evacuation Dynamics, 3-20, M. Schreckenberg and S. D. Sharma (eds.), Springer, Berlin, 2002.

[2] K. Ando, H. Oto and T. Aoki, Forecasting the flow of people [in Japanese], Railway Research Review 45 (8), 8-13 (1988).

[3] J. Dzubiella and H. Löwen, "Pattern formation in driven colloidal mixtures: tilted driving forces and re-entrant crystal freezing", J. Phys.: Cond. Mat. 14, 9383-9395 (2002).

[4] D. Helbing, "A mathematical model for the behavior of pedestrians", Behavioral Science 36, 298-310 (1991).

[5] D. Helbing, Quantitative Sociodynamics. Stochastic Methods and Models of Social Interaction Processes, Kluwer Academic, Dordrecht (1995).

[6] D. Helbing, Verkehrsdynamik [Traffic Dynamics], Springer, Berlin (1997).

[7] D. Helbing, I. Farkas, and T. Vicsek, "Simulating dynamical features of escape panic", Nature 407, 487-490 (2000).

[8] D. Helbing and P. Molnár, "Social force model for pedestrian dynamics", Physical Review E 51, 4282-4286 (1995).

[9] D. Helbing and T. Vicsek, "Optimal self-organization", New Journal of Physics 1, 13.1-13.17 (1999).

[10] S.P. Hoogendoorn, P.H.L. Bovy, *Pedestrian route-choice and activity scheduling theory and models*, 2002

[11] P. Molnár, Modellierung und Simulation der Dynamik von Fußgängerströmen, Shaker, Aachen (1996a).

[12] P. Molnár, "Microsimulation of pedestrian dynamics", in Social Science Microsimulation, J. Doran, N. Gilbert, U. Mueller and K. Troitzsch (eds.), Springer, Berlin, 1996b.