**5. Implementation**

As mentioned above are we using the code from a previous project because of its high efficiency and therefore good performance.

Of course we had to make several changes to the code so that it is suitable for our simulation in order to answer our question setting.

In this chapter we will explain these changes. At least the most significant. For the exact understanding of the original code and the process of optimization, please refer to the documentation, “Modeling of Evacuation Siutations in a multi-level building”, chapter 5 and 9 **[2]., or our code in the Appendix.**

**5.1 Changes**

**5.1.1. exits are not on lowest floor**

Reason:

The first change which was necessary was due to the fact, that the exits in our simulation are not simply on the lowest floor.

Assumption:

Here we assume, that the agents above the exit floor only move down, and those under it only upstairs.

Function: applyForcesAndMove.m,

new varables:

🡪 To define the floor in which the exits are we introduce a new variable „floor\_exit“ in the config file.

Modifications:

Since we know now in which floor the exits are, we can modify the code easily by splitting the loop, in which we calculate the forces and move the agents, in two parts.

First we loop over all floors higher than the exit floor, in which all agents only have to use the stairs down, or in case they reached the exit floor, the exits.

And second, we do the same for all the agents in floors lower than the exit floor, now only checking if an agent reached a staircase up or an exit.

This modification keeps the code fast, since we do not have to loop twice over all floors. We can also keep the simple concept of a vector of booleans.

For each agent there is one number:

🡪if agent reached a staircase and therefore changes floor: 1

🡪else 0

The assumption also is a great simplification for the pictures since we don’t have to mess with the problem of having „overlapping“ stairs.

**5.1.2.** **having different exits**

Reason:Since the exits model lifeboats, which can only hold a limited number of agents, we had to find a way, to differ the exits from each other and to assign a specific number to every exit.

Assumption:

Function:loadConfig.m

new varables:

🡪*exit\_count,* to define the number of exits

🡪for each exit k: *exit\_k\_nr,* to define the number of agents it can hold

🡪To store how many agents can exit in one specific exit, we introduced a matrix exit\_nr matrix, where the number of agents that can exit is indicated for each pixel

Modifications:

🡪We had to make some changes in the decoding of the pictures. The aim was to change as little as possible to the original code. It was clear that we are going to need as many different colors as we have exits, to be able to distinguish them during the simulation. By defining every pixel which is red to value=0, blue value=0 and green value unequal to zero, we can define a lot of different colored exits by using green values from 256 to 256-*exit\_count*.

-->how we implement the matrix…different green values for different exits

🡪we implemt a matrix *exit\_nr* similar to the already existing one *img\_exit*, in which we store a count to number the different exits. The number is defined by the green value of the pixel we are at.

%make a zeroes matrix as big as img\_exit

config.exit\_nr=zeros(size(config.floor(config.floor\_exit).img\_exit));

%build the exit\_nr matrix

config.exit\_nr = config.exit\_nr + e\*( img\_build(:, :, 1) == 0 & img\_build(:, :, 2) == (256-e) & img\_build(:, :, 3) == 0 ) ;

**5.1.3. closing exits during simulation**

Reason: To close an exit as soon as it let a specific number of agents in, we have to keep track of the number of agents that already used this exit.

Assumption:

Function: loadConfig.m, applyForcesAndMove.m

new varables: exit\_left

Modifications:

For this purpose, we defined the matrix exit\_left, in which we store the number of agents who can exit for every exit (defined by his number).

%make a zeros vector as long as **exit\_count**

config.exit\_left = zeros(1,config.exit\_count);

%loop over all exits

for e=1:config.exit\_count

%build the exit\_nr matrix

config.exit\_nr = config.exit\_nr + e\*( img\_build(:, :, 1) == 0 & img\_build(:, :, 2) == (256-e) & img\_build(:, :, 3) == 0 ) ;

**%build the exit\_left matrix**

**config.exit\_left(1,e) = config.(sprintf('exit\_%d\_nr', e));** 🡪save the number of agents the exit can hold

end

In the loop where the forces are calculated and the agents moved, (in function apllyForcesandMove.m) we added a piece of code, which updates the exit\_left matrix at every time-step.

First, we get the number of the current exit we are at

%save current exit nr

data.current\_exit = data.exit\_nr(round(newp(1)), round(newp(2)));

then we update the exit\_left matrix by counting down the number of agents allowed to exit by 1.

%update exit\_left

data.exit\_left(1,data.current\_exit) = data.exit\_left(1,data.exit\_nr(round(newp(1)), round(newp(2)))) - 1;

If the allowed number of agents exited the number is 0. Now we have to close the current exit, by changing it into a wall. Therefore we have to update the img\_wall matrix.

%close exit if there is no more free space

if data.exit\_left(1,data.current\_exit) < 1

%change current exit to wall

data.floor(data.floor\_exit).img\_wall = data.floor(data.floor\_exit).img\_wall == 1 ...

| (data.exit\_nr == (data.current\_exit));

data.floor(data.floor\_exit).img\_exit = data.floor(data.floor\_exit).img\_exit == 1 ...

& (data.exit\_nr ~= (data.current\_exit));

**5.1.4. controlled evacuation**

Reason: With the goal of a faster evacuation, we tried to control the agents to go to specific exits.

Assumption: We realised that the biggest problem-zones are the stairs. So, in order to get the agents as fast as possible away from the stairs, once they changed the floor, we decided to split the agents in two groups.

One group only reaches one half of the exits and the other group for the others.

Function: init\_agents.m , addDesiredForce.m, initEscapeRoutes\_even.m, initEscapeRoutes\_odd.m, addDesiredForce.m, initialize.m, initAgents.m, **loadConfig.m**

(you can find the changes in the code-files by searching for the comment „control exit“)

new varables:

🡪*numbr*, each agents gets a number ( 0 or 1, selected randomly)

🡪*control\_exit*, in the config file you can decide wether or not you want to have controlled exits during your simulation

Modifications:

Essentially, we had to modify the calculation of the force dragging a specific agent to the nearest exit. Therefore we wrote two new functions to init the escape routes.

🡪 initEscapeRoutes\_even.m, this function only considers the exits which are identified by a even number

temp1=double(mod(data.exit\_nr,2)); %matrix in which every number which is even turns to zero, odd turns to one

temp2=logical((data.floor(i).img\_exit)-(temp1));

boundary\_data(temp2)=-1; %boundary\_data considers only the exits with even numbers --> -1

🡪initEscapeRoutes\_odd.m, this function only considers the exits which are identifies by a odd number

temp=logical(mod(data.exit\_nr,2)); %matrix in which every number which is even turns to zero, odd turns to one

boundary\_data(temp) = -1; %boundary\_data considers only the exits with odd numbers

From this functions, we get two different directions to the next exit.

🡪 [data.floor(i).img\_dir\_x\_odd, data.floor(i).img\_dir\_y\_odd]

🡪 [data.floor(i).img\_dir\_x\_even, data.floor(i).img\_dir\_y\_even]

Those, we use to calculate the force dragging the specific agent to his nearest exit.

The agents are splitted in “even-agents” and “odd-agents” defined by the randomly added number (0 or 1).

%even agents

if numbr==0;

% get direction towards nearest exit

ex = lerp2(data.floor(fi).img\_dir\_x\_even, p(1), p(2));

ey = lerp2(data.floor(fi).img\_dir\_y\_even, p(1), p(2));

e = [ex ey];

% get force

Fi = m \* (v0\*e - v)/data.tau;

% add force

data.floor(fi).agents(ai).f = data.floor(fi).agents(ai).f + Fi;

end