Emergency exit simulation

Project Document

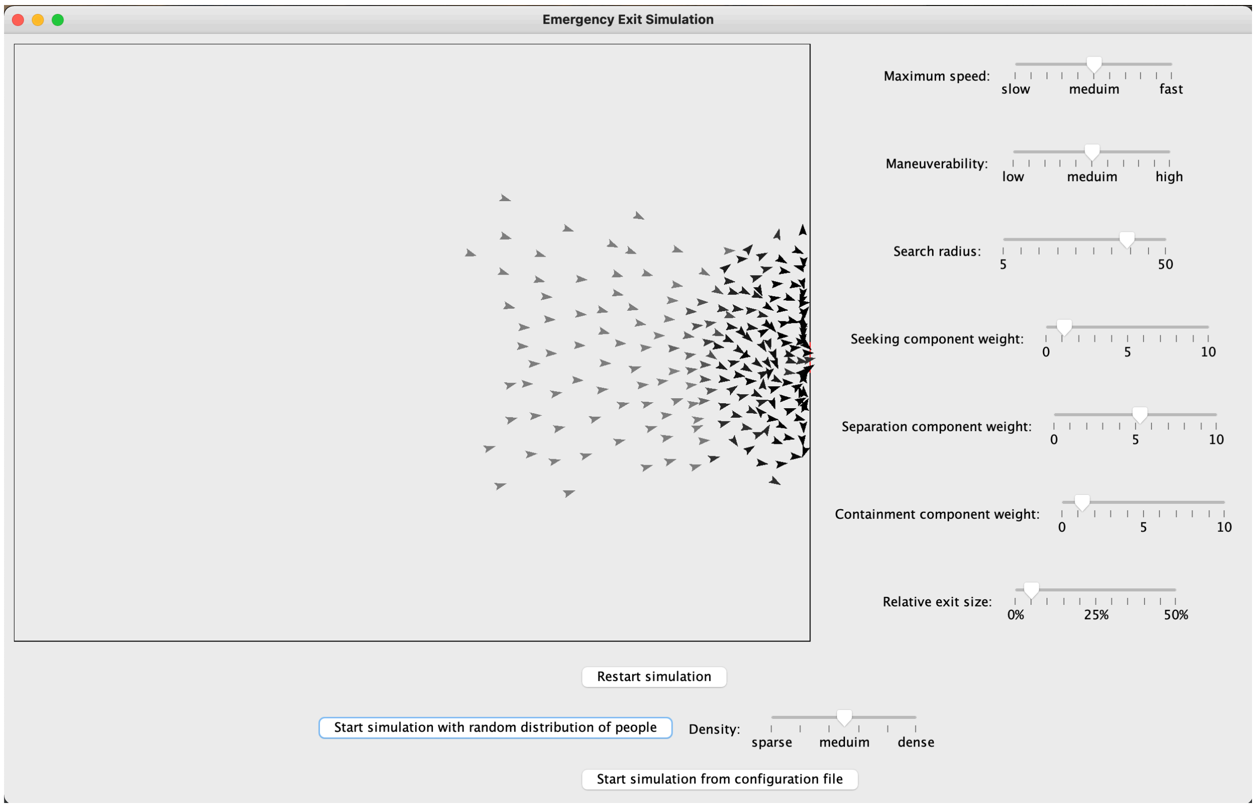
Project by Georgy Ananov (890980)
Bachelor's program in Data Science
First year
26.04.2020

General description

The created application simulates a crowd of people exiting a rectangular room through a doorway in an emergency exit situation. The simulation Is presented through an interactive graphical user interface, through which the user can view the current positions of the simulated people, as well adjust the various parameters of the simulation. The initial state of the simulation can either be generated randomly, or imported from a configuration file.
The logic of the simulation is based on Steering Behaviors described in the article "*Steering Behaviors For Autonomous Characters*" (Reynolds, 1999).

The application meets all of the requirements stated on the project topic page (congestion simulation). The program was created with the "challenging" difficulty level in mind.

User interface


Emergency Exit Simulation
Maximum speed:
slow
meduim
fast
Maneuverability:
low
meduim
high
Search radius:
5
50
Seeking component weight:
0
5
10
Separation component weight:
0
5
10
Containment component weight:
0
5
10
Relative exit size:
0%
25%
50%
Restart simulation
Start simulation with random distribution of people
Density:
sparse
meduim
dense
Start simulation from configuration file

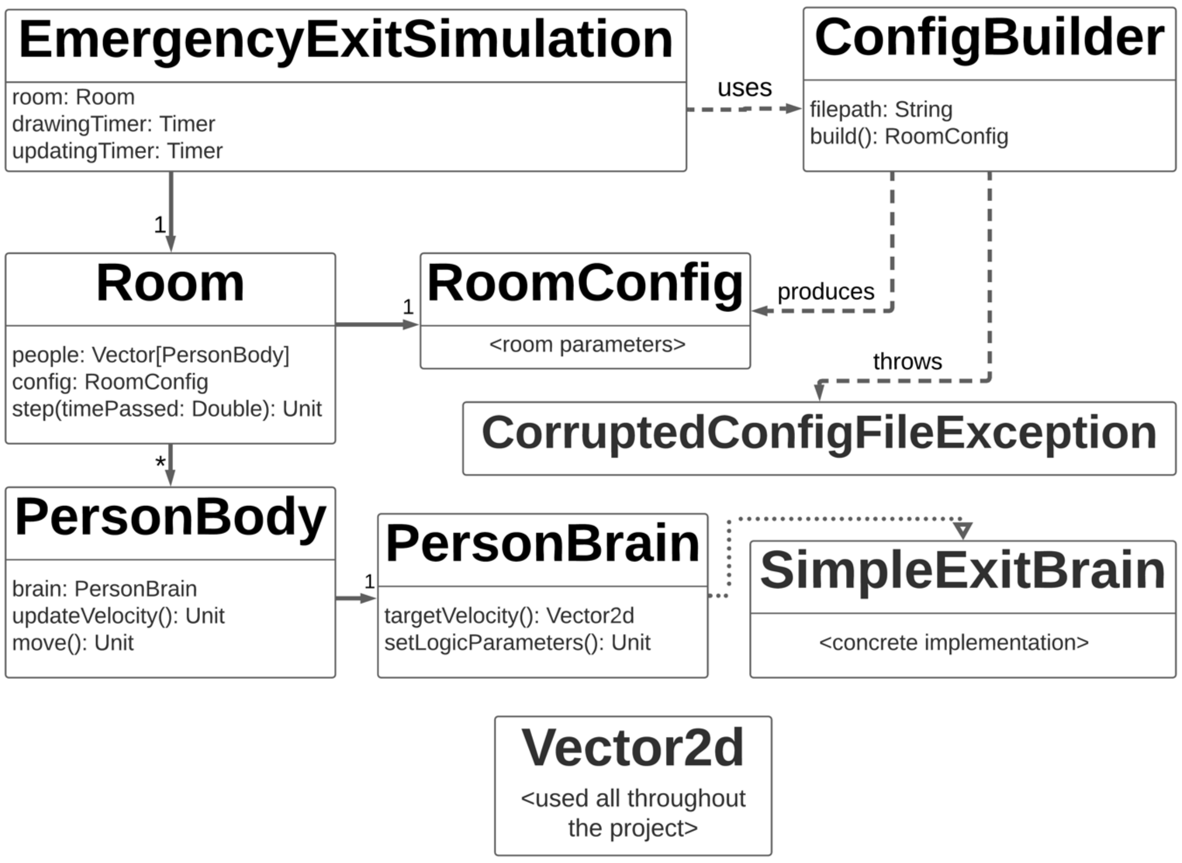
After starting the program, the user is presented with a graphical user interface. The window is split into three main areas: the simulation panel (top left portion of the screen), the parameter adjustment panel (right side of the screen), and the simulation controls panel (bottom of the screen).

Within the simulation panel, each simulated person is represented by an arrow shape. The coloring of the person depends on how fast they are moving compared to their maximum speed – the darker, the slower. The exit of the room is represented by a red line, while the walls are represented with black lines.

The simulation panel initially represents an empty room with default dimensions. The user can start the simulation in one of two ways – either by clicking the “Start simulation with a random distribution of people” button or clicking the “Start simulation from configuration file”. The former option populates the room with people in random locations, while the latter prompts the user to select the desired configuration file within a file chooser window. After either of the options is selected, the movement of the people commences and runs until the user chooses to reset the simulation with the buttons mentioned above or with the “Restart simulation” button. The “Restart simulation” button offers the user the option to replay the simulation from the start (preserving the exact starting conditions).

At any point during the simulation, the user can adjust the parameters that govern the motion of the simulated people. This is done using the sliders located to the right of the simulation panel. Hovering over the labels of each of these sliders or buttons reveals a tooltip containing additional information about the function of the UI pieces.

Program structure



Above is a simplified structure plan of the project with the most important variables, methods and class interactions highlighted.

At the core of the entire project lies the *EmergencyExitSimulation* object. This object is primarily responsible for interacting with both the user and the simulation logic. For the user, this object draws the GUI, listens to the user interacting with buttons and sliders. For the simulation, this object sends the instructions to update the simulation parameters when necessary and requests the simulation to move forward. To that extend, the *EmergencyExitSimualtion* object utilizes two Timer instances – one for redrawing the GUI and another one for requesting a simulation update by calling the *step()* method of the *Room*.

Linked to the *EmergencyExitSimualtion* is an instance of the *Room* class. This class holds together everything that is needed for the simulation to run: the list of people, the methods that are necessary for those people to be aware of their surroundings, and a link to an instance of *RoomConfig* data structure. The *RoomConfig* data structure is responsible for storing the parameters of the simulation, such as the room dimensions, exit location and size, movement logic parameters and so on. This data structure also plays an important role in the process of preserving the simulation parameters between instances of the *Room* class. In order to facilitate the creation of the *RoomConfig* from a configuration file, the *ConfigBuilder* class is implemented. *ConfigBuilder's* sole purpose is to process a configuration file at a given filepath, and produce a ready-to-use *RoomConfig* instance.

Moving on to the logic of the simulation, the *PersonBody* and *PersonBrain* classes work in tandem to determine the location of one specific simulated person at all times. The abstract *PersonBrain* class is responsible for determining the desired direction of movement of the person using steering behaviors. The *SimpleExitBrain* class proves concrete implementations of the methods defined in *PersonBrain*. In order to calculate the desired velocity, *PersonBrain* requires some information about the state of the simulation, such as the locations and velocities of nearby people and the proximity to the walls of the room. The *PersonBody* is responsible for working with the Room class to provide the *PersonBrain* with all the information it needs. *PersonBody* is also in charge of keeping track of the physical constraints of a person, such as the speed and acceleration limits as well as the room walls.

Finally, the *Vector2d* is a data structure used by almost every class in the project. *Vector2d* helps streamline all of the vector arithmetic that needs to happen as the program runs.

The structure of the project emphasizes the modularity and expandability of the project. Many of the elements of the simulation can be changed or swapped out without changing the surrounding infrastructure. New behaviors for people can be added by simply inheriting from the abstract *PersonBrain* class. New room shapes can be implemented by changing the *getBoundaryNormal* method in the *Room* class. A simpler approach to designing the class structure of the project, such as one that combines all of the logic into one object would have perhaps been a little quicker to implement, but much of the modularity would have been lost.

Algorithms

The core algorithm of the simulation is based on the steering behaviors (C.W. Reynolds, 1999). Each individual person's behavior is modeled by several basic rules, that determine the direction in which the person strives to move. Below are the 4 rules used in this simulation:

1. Seeking. The seeking component vector is aimed at a point slightly behind the middle of the exit doorway.
2. Containment. In order to avoid finding themselves facing directly into a wall, a person tests the space ahead of them with several probe points. If any of these probes happen to be outside of the room walls, a normal vector from each relevant probe to the wall of the room is calculated. These normals are then added together to form the containment component.
3. Separation. A person surveys the surrounding area within a certain radius, looking for other people. For every person found, a repelling vector is calculated and scaled with the inverse of the squared distance between the two people. The repelling vectors are summed up to produce the separation component.
4. Braking. A person "looks" ahead and slows down upon detecting a person in front of them moving slower than itself. The action of slowing down is modeled through calculating a breaking coefficient, and then applying it to the distance of travel at every step of the simulation. This ensures that no person moves faster than the people ahead of them.

The velocity components obtained from these steering behaviors are combined linearly, which outputs a resulting desired velocity vector. The desired velocity vector is then compared with the current velocity of a person, the difference is scaled and added to the velocity for the next step of the simulation.

These computations are carried out for each individual person, after which the whole simulation advances one step forward, and all of the people move according to their newly calculated velocities.

Data structures

For the purpose of storing the *PersonBody* instances in the *Room* class I have chosen a simple *Buffer*. Initially I was using a *Vector*, but during development the need to alter the contents of the data structure became apparent, so I switched to *Buffer*.

For the majority of the development process I kept the simulation parameters, such as room dimensions and speed limit, as simple variables in the *Room* class. This has caused me great pain, as every time I needed to create a new *Room* instance but keep some of the parameters, I had to manually copy the parameters over. All of these issues were solved then I implemented a data structure for storing simulation parameters – the *RoomConfig* class. With this data structure in hand, I was able to implement much of the GUI-related functionality without much trouble.

Finally, the *Vector2d* data structure proved to be of immense help when it came to vector arithmetic. I had to expand its functionality several times as I made progress with developing the application. Early in the development I did consider using a pre-made structure for this purpose, but ended up deciding against it, as some of the methods I required were not implemented in the structures I was able to find.

Files

The application uses a custom text file format to store the parameters of the simulation. Using files allows the user to directly input the exact parameters of the room they wish to simulate, as well as the exact coordinates of the people within it. The file format is human-readable and editable. The file is separated into 11 blocks that can be placed in arbitrary order, each of which starts with a header line and contains information about one of the simulation parameters. This information must be presented as a single decimal number, or as a list of pairs of decimal numbers in the case of the *initial_coordinates* parameter. Each pair must be on its own line and the numbers within the pair must be separated by a comma. No extra lines or values must be included in the file.

Description of the parameters:

- *#room_width* – the horizontal dimensions of the room
- *#room_h*eight – the vertical dimensions of the room
- *#relative_exit_size* – the ratio of the exit size to the length of the right room wall
- *#exit_location* – the vertical coordinate of the top point of the exit (the zero coordinate corresponds with the top of the room)
- *#max_speed* – the maximum speed the people will be allowed to reach
- *#max_acceleration* – the maximum acceleration the people will be allowed to reach
- *#search_radius* – how far apart two people must be to not be considered neighbors
- *#seeking_weight* – the weight coefficient of the seeking behavior
- *#separation_weight* – the weight coefficient of the separation behavior
- *#containment_weight* – the weight coefficient of the containment behavior
- *#initial_coordinates* – the list of coordinates where people will be placed at the start of the simulation

Sample configuration file:

```
#room_width
  500.0
#room_height
  600.0
#relative_exit_size
  0.01
#exit_location
  30.0
#max_speed
  0.05
#max_acceleration
  0.0001
#search_radius
  25.0
#seeking_weight
  20.0
#separation_weight
  120.0
#containment_weight
  30.0
#initial_coordinates
  2.0   , 2.0
  10.5  , 28.1
```

Testing

Most of the testing has been carried out through the user interface, as the simulated behavior is difficult to test precisely. Another argument against the use unit tests for the simulation process is that the primary goal of the simulation is to achieve realism rather than absolute precision, which makes visual observation more meaningful than automated tests.

That said, unit tests have proved useful in testing the *Vector2d* data structure and some of the methods in the *Room* class. Since both of these classes have received extensive expansions and reworks, the unit tests were indispensable in making sure the classes still work in the expected way after the changes.

Generally, whenever I would add a new feature or an element of the GUI, I would test it through the user interface before moving on with developing the application further. The file interactions and error handling have been tested manually.

Known bugs and missing features

The application does not contain any noteworthy bugs that I am aware off – I have squashed any existing ones to the best of my ability. That said, there are a couple behaviors that may be interpreted as inconsistent. Among them is the behavior of the GUI sliders in the scenario where the values inputted by the user through the configuration file fall outside the range of the sliders. In that case, the sliders technically would display incorrect information (for example, if the user inputs 200 as the separation weight, the slider would display 100 instead, as that is the maximum value it can display). This issue only concerns the displayed values, the actual parameters are interpreted correctly.

Another aspect of the user interface that is worth pointing out is that no safeguards are put in place to prevent the user from creating and loading a configuration file that results in impossible rooms. For example, the exit may end up outside of the room bounds if the *exit_location* parameter exceeds the height of the room, or the doorway might be created too small for even one person to fit through. As the parameters are fairly straight-forward, and the application does not crash as a result of impossible room configurations, I did not feel that putting such safeguards in place should be high on the priority list. Nevertheless, In the future this feature definitely should be implemented for an optimal user experience.

Finally, I should mention that the adjustment of the exit location is only available through the configuration file. As the use of the application without the configuration file is intended to be mostly a space for experimenting with different combinations of steering component weights, I did not feel the need to include this adjustment in the GUI sliders. In contrast, using the application through the configuration file is intended for actual simulations and testing of room plans, so the exit location adjustment is a lot more relevant in that use case.

Best sides and weaknesses

In my opinion, the strongest aspect of the project is the modularity and expandability I was able to achieve. The project can be transformed into a simulation of a different kind (such as the flocking simulation or path following simulation) with minimal effort. Furthermore, the different kinds of simulated behaviors can even be combined within the same simulation space. For instance, one could create a room where some people are trying to reach the exit, while others try to form a line to protect the exit doorway.

Another feature I am pretty proud of is the mechanic of representing people with differently shaded icons depending on their current speed. This mechanic wasn't particularly difficult to implement, but, in my opinion, it brings a lot when it comes to analyzing the flow of people in the room. At a glance the user can tell where people are crowding up, and where they are able to move at full speed.

As for the weaknesses, I feel that the simulation is still running fairly slowly, despite the fact that the heavier calculations are rigged to run in parallel. At the moment, my application can comfortably simulate up to 300 people in real time on a 4-year-old laptop. Bumping the number of people up beyond that starts causing visible stuttering and lag. I do have a solution to this problem in mind, which relies on partitioning the space of the room into multiple regions. This would enable the program to only examine certain parts of the simulation space when looking for neighbors for any one person, instead of having to loop over the entire collection of people. This partition can also be combined with a dynamic search radius (for example, when a person detects more than 10 neighbors at any one step of the simulation, we can instruct this person to shrink its search radius for the next step). I have unfortunately been unable to find time for implementing these optimizations in the development schedule.

I am also not entirely satisfied with the appearance of the GUI. The *scala.swing* library proved to be somewhat irritating to work with due to significant lapses in documentation. As a result, the GUI I produced is not exactly pleasing to the eye, especially in terms of alignment of buttons and sliders. In my opinion, the best solution to this issue would be to move the entire GUI to a different framework, such as the more modern *scalaFX* library. Fortunately, the logic of the simulation would not need to be changed, since the code responsible for creating and running the simulation is entirely separate from the user interface code.

One feature that the application is lacking at the moment is a timer that would measure the total evacuation time and display it to the user. Implementing this feature is a little trickier than it might seem at first glance, since one has to consider the potential slowdowns of the simulation thread due to large number of simulated individuals, or other processes competing for the computational resources. Since a timer was not listed as a mandatory feature, I decided against prioritizing it over some of the other critical features, but, if given more time, I would definitely try to implement this feature as soon as possible.

Deviations from the plan, realized process and schedule

In general, the development proceeded very closely to what I had initially planned. I started off by implementing the *Vector2d* data structure, as without it I could not create even the most basic skeleton of the class structure. After the *Vector2d* was completed and thoroughly tested with unit tests, I proceeded to create some basic logic for the simulation, omitting any of the “brains” of the people. Once I had implemented the essentials of the Room and *PersonBody* class, I began working on the graphic interface, so that I could see the classes I was working on in action. From that point onward, I was working intermittently on the logic of the simulation, the UI and on expanding the *Vector2d*. Finally, near the end of the development timeframe, I implemented the classes and methods responsible for working with files and put final touches on some of the UI elements.

My schedule ended up looking something like this:

Weeks 1-2:	Skeletal class structure, *Vector2d*, basic GUI
Weeks 3-4:	Seeking and separation logic, more work on *Vector2d*
Weeks 5-6:	Containment and braking logic, sliders in the GUI, more work on *Vector2d*
Weeks 7-8:	Finalization of the UI, configuration files, code cleanup

My workload estimations were mostly on point, with two notable exceptions: working on researching and implementing threads took significantly less time that I anticipated, and working on the UI took quite a bit more time than I was expecting. The overall workload turned out to be around 60-65 hours.

All throughout the development process, I felt the immense importance of starting out with a solid plan and a schedule. At no point did I find myself guessing what part of the project to take on next. This not only sped up the process, but also made it seem a lot less intimidating and more manageable.

Final evaluation

Two months ago, I set out to build a visual application for simulating the movements of a large number of people in an emergency exit situation. I have completed this goal by creating a finished application with all of the required features and some additional ones. Although the project still has room to expand, I am more than satisfied with the progress I've made, especially since this is my first experience in fully designing, planning, implementing and finalizing a software project.

Among the design choices I've made, the one I am least confident in the choice of the GUI library. If I were to start this project from scratch, I would probably move away from scala.swing, as this library proved to be somewhat outdated and poorly documented.

The application can be further developed in two main directions – expanding the core simulation functionality with new behaviors or improving the graphical user interface. As mentioned above, the former can be done with very little refactoring of the existing code. The latter may require migration to a different library, which quite obviously involves a lot of refactoring, but only the GUI class would have to be reworked, since the model is designed to work completely independently if the exact implementation of the user interface.

References

- C. Reynolds (1999) ”Steering behaviors for autonomous characters” http://www.red3d.com/cwr/steer/gdc99/
- Scala-swing library https://github.com/scala/scala-swing
- O1 section on GUIs https://plus.cs.aalto.fi/o1/2020/w12/ch03/
- ScalaTest library https://www.scalatest.org/
- Sebastian Lague's video on steering behaviors https://www.youtube.com/watch?v=bqtqltqcQhw

Special thanks to Sergey Zakuraev, who helped me with many of the issues along the way.