

Stampede

student's name: WU Yiyou

student number: 101731278

degree program: computational engineering (English bachelor)

year of studies: first year

date: 24 / 02 / 2024

General description

I implemented the project on the difficult level, that is, to create a program that simulate a rushing situation where all individuals in a room want to leave the room through its only small door at the same time and as quickly as possible.

And their behavior obeys the following rules:

1. Seek. all the members in the room are always adjusting their velocity to get through a fixed position(where the door is), they won't slow down in front of the door(different from "arrive" which will stop right before the destination)
2. Brake. A member in the room can detect another individual in front of them and their speed, they will slow down if the other individual is slower than themselves.
3. Avoidance. Members in the room avoid walls. When a wall is present in the front of an individual within certain distance, a steering force will be applied to avoid collision.
4. Separation. A member in the room can detect others in a nearby area, and depending

on the distance between them, a combined force will be applied to steer it away from them.

Combination of those behavioral mode

1. linear combination: linearly combine the (weighted) steering force vector together.
2. assign priority: decide the most important action of the moment, and calculate the steering force of that action, then do some other calculations to decide to what extent the next force should be applied.(or omit them all at once)

I implemented both modes in my project, but only the linear combination one is presented in the GUI.

User interface

the Main Menu:

After launching the program, the user will enter the main menu window, he/she will see the following buttons:

new Sim: this button will launch a new simulation with the default parameters.

Load Sim: pressing this button will give the user a list of existing saves (if any). The user can launch one of them by clicking it.(It may take a few seconds for the buttons to be responsive)

Continue: If the user returns to the main menu from the simulation, they can go back to it by clicking this button.

The Simulation:

Any of the buttons from the main menu will lead the user to this page where the simulation takes place.

There are a couple components on this page that the user can use to control the simulation.

In the section on the left:

The use for the **sliders** is very self-evident, to adjust the values the parameters in the

simulator: I used abbreviation for simplicity.

Just for clarification: avoi -> avoidance, rad -> radius, agl -> agility, mass -> mass, sek -> seeking, mxs -> max speed, mxf -> max force, sep -> separation

And the numbers in display mean the power of ten, so don't drag them too much every time. :)

For the **buttons** down below,

Initiate button means to paint the simulation, every time you launch a simulation you need to click it to make the simulation appear. The same applies when you return to the simulation from the main menu.

Pause/ Resume button will pause/ resume the simulation (click it once and the text on the button will change). The user needs to press it after initiating the map for the people to start moving.

Add/Kill button simply add/ remove 50 people in the room, add people will result in 50 people appearing at random positions in the room and kill people will remove the people that are added last.

Tick button advances the positions of the members in the room manually by a bit. It is usually used when the simulation is paused, so the user can observe the behavior of the people in the room at their own pace.

In the **canvas section** on the right, the user can:

Enable the **door moving mode** by pressing the **control button(ctrl)** on their keyboard, after that a line of red text will appear on the top left corner on the canvas indicate that the door moving mode the is enabled.

When the door moving mode is enabled:

Dragging the mouse with the left button moves the door, regardless of if the simulation is paused or not.

When the door moving mode is disabled:

Dragging the mouse with the left button draw a line indicate where you want to build a wall. release the button and a confirmation window will pop up, confirm if you want to build a wall where the line is.

Dragging the mouse with the right button draw a rectangle within which the walls will be destroy, a similar confirmation window will also appear to make sure you make the right move.

For the non-interactive part of the canvas

A blue bubble is a person, the line in from of the bubble indicates the velocity. The purple rectangle is the door. If a person gets in the door the person will be removed from the

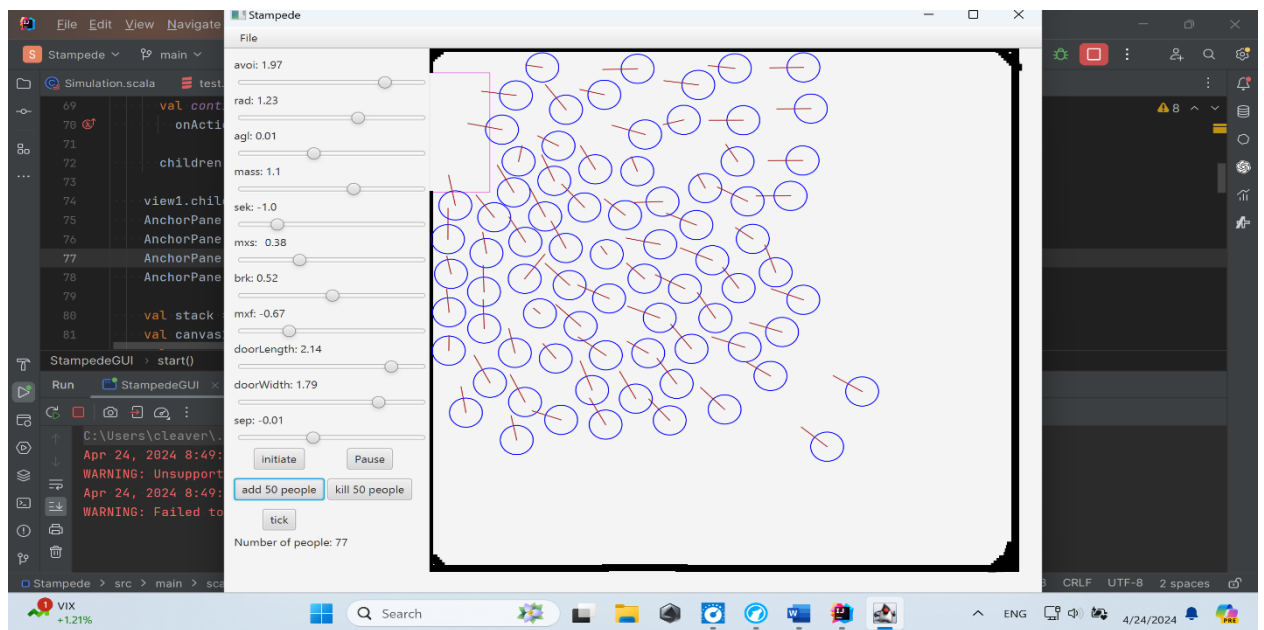
room. The many small black squares are walls.

For the **menu bar** on the top

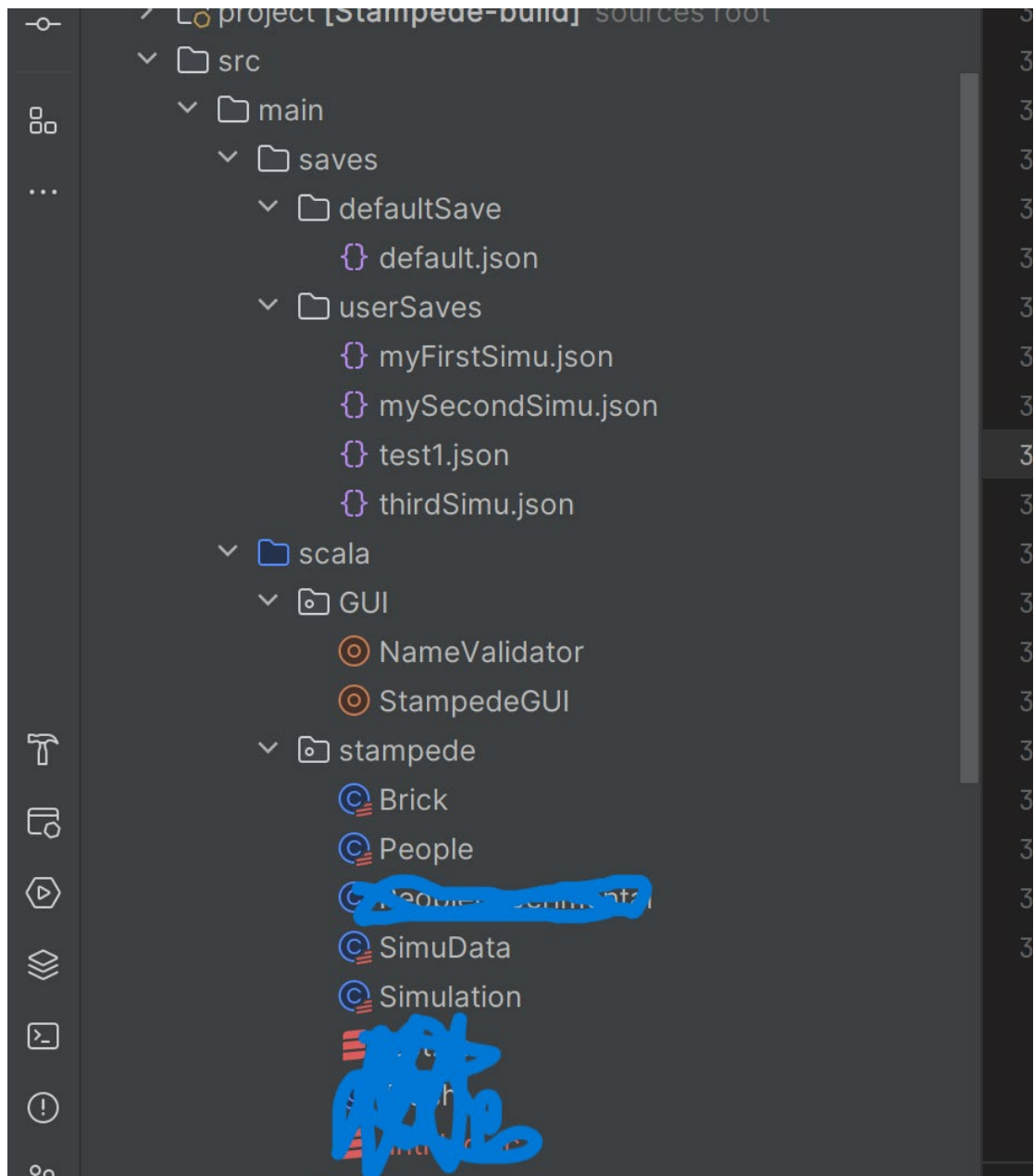
You can see two menu items upon clicking it.

Main menu: pause the simulation and return to main menu.

Save: If it's a new Simulation a window will pop up and ask you to name the simulation before saving, otherwise it just saves the simulation.



Program structure



neighborRadius: everyone within this radius would be considered neighbors and the people in the room would try to get away from their neighbors. It is calculated based on agility and radius (size of the circle).

CylinderLen: calculate the cylinder length mentioned above, it is used to detect obstacles.

Move(): change the person's velocity and position based on the steering force calculated by other methods and the parameters from the Simulation class

isOut: returns true the person gets in the door.

Seeking: return a force vector toward the part of the target door that is closest to the person.

AvoidanceBrakingAndSeparation: return three force vectors represent avoidance, braking, and separation, I put them together because there are some shared calculation for these three forces: putting them together is to avoid repeated calculation.

Other methods are simple helper methods for calculating the forces.

Case class **Brick** is to model the walls in the simulation with small circles.

Path: it is the vector from the start to the end of the wall.

fillBrick: return a matrix of the position of those small circles on the path, these would be added to the actualWall variable in the class Simulation.

Case class **SimuData**

This class stores the necessary information from a Simulation to facilitate file handling. It derives ReadWriter from upickle library (a library for handling Json file in Scala).[1]

The reason I choose this way of partition is quite straightforward:

the main logic of the program is the class Simulation.

Two main things in the Simulation: Walls and People, and walls are simpler, so they get a case class.

And the class SimuData is for file handling.

Alternative solutions:

1. Model door as a case class like Brick.

Reason why I didn't use it: I didn't intend to make the door in my program very complicated (such as multiple doors with different shapes) so it's not worth it.

2. Model all people in the class People using a matrix of position and velocity.

Reason why I didn't use it: The linear algebra library I use has bad documentation and the readability of the program would suffer.

• Algorithms

Algorithms are the major challenges in the project, and they will be developed following the article Steering Behaviors for Autonomous Characters by Craig W. Reynolds.[2]

1. People's movement

This part will be done by implementing the Simple Vehicle Model from the article mentioned above.

The change of position and velocity in 2d:

$\text{steering_force} = \text{truncate}(\text{steering_direction}, \text{max_force})$

$\text{acceleration} = \text{steering_force} / \text{mass}$

$\text{velocity} = \text{truncate}(\text{velocity} + \text{acceleration}, \text{max_speed})$

$\text{position} = \text{position} + \text{velocity}$

The formula I copied from the article is quite straightforward. I will do some simple explanations:

The steering_force here will be obtained from the linear combination of the steering force produced from the four behavioral modes. Note that all the quantities here are vectors(2d), hence the operations are vector operation.

The change of orientation in 2d:

$\text{new_forward} = \text{normalize}(\text{velocity})$

Here because we are operating on the flat plane, the orientation is represented by a single vector aligned with the velocity. Normalize means scale the size of the vector to one.

2. Calculating the steering force

Seeking

$\text{desired_velocity} = \text{normalize}(\text{position} - \text{target}) * \text{max_speed}$

$\text{steering} = \text{desired_velocity} - \text{velocity}$

The "desired velocity" is a vector in the direction from the character to the target. The target position here will be the position of the part of the door that is closest to the person. Which is achieved by comparing the person's position (both x and y) with the door's position at both end (the door is modelled with a rectangle). For example, if the door's highest x is 100, and lowest x is 50, then if the person's x is between 50 and 100 then it's target x would be itself, but if it's lower than 50 then the target x would be 50, if it's higher than 100 the target x would be 100. The same for y.

Avoidance

As in the picture below, The goal of the behavior is to keep an imaginary cylinder (rectangle in the case of 2d) of free space in front of the character. The cylinder has the same radius as the character and the length is calculated based on the character's speed and agility. The wall is approximated by a line of small circles (Brick class) The character ignores any brick outside of the cylinder, The brick which intersects the forward axis nearest the character (in the picture marked as red) is selected as the "most threatening." Steering to avoid this obstacle is computed by negating the (lateral) side-up projection of the obstacle's center.

Particularly, I first convert the object's coordinate into the agent's local coordinate (for simpler calculations) using a rotating matrix, then draw a vector from the agent's position to the center of the object's bounding sphere. If the x component of that vector is greater than the rectangle's width (rectangle's x component), we know the two shapes are not intersecting, otherwise there is a potential intersection, in which case we compare the y component of each. [3]

Separation

To compute steering for separation, first a search is made to find other.

characters within the specified neighborhood. The radius of the search is specified by the variable in class People as neighborRadius. For each nearby character, a repulsive force is computed by subtracting the positions of our character and the nearby character, normalizing, and then applying a $1/r$ weighting.

Steering1 = normalize (position - neighbor1Position)

Steering2 = normalize (position – neighbor2Position)

...

SteeringN = normalize(position – neighborNPosition)

brakingForce = steering1 / r1 + steering2 / r2 + ... + steeringN / rN

where r is the distance between the character and the neighboring

Braking

Braking is very similar to avoidance: we will have the same imaginary cylinder, In this case, only the people that are considered neighbor are under the radar of the cylinder to save computational resource, if any of them have a speed lower than the person itself, a braking force(opposite to its velocity) will be applied, it's magnitude will be based on the person's own velocity.

• Data structures

I use ArrayBuffer for storing the members in a simulation. It is mutable because I need to modify the size of the collection. The program needs to add/remove people which will use apply and append, according to the collection performance chart in the scala

documentation, ArrayBuffer is very fast in these two operations.

I use DenseVector and DenseMatrix from breeze library to store the position, velocity, force, and all the other vector values because it offers a natural way to calculate vectors. Additionally, matrix operation proved to be very efficient when calculating the braking and avoidance forces.

I use mutable hashmap to store the parameters, since it will only have a few elements, the speed doesn't really matter here, I need it to be mutable so that the user can change the weights.

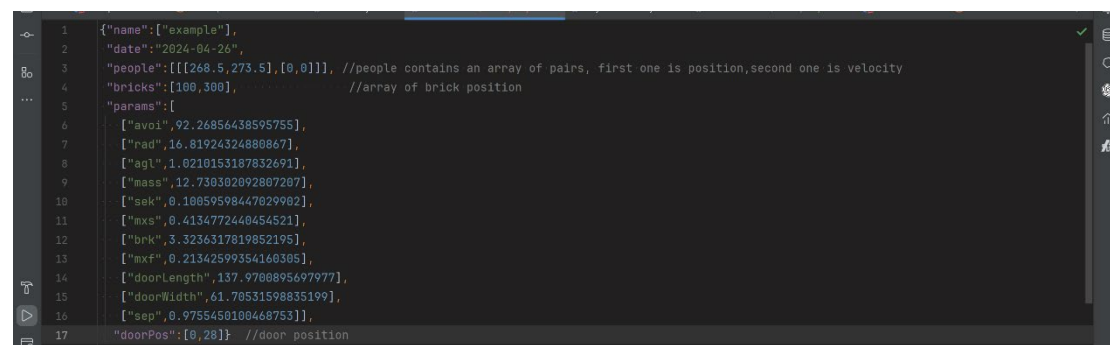
• Files and Internet access

I created a folder named saves under src/main, with two folders inside which: defaultSave and userSaves

They are of the same format. When the user launch a new Simulation the file default.json in defaultSave will be read, when the user load an old simulation a save in userSaves will be read.

When the user saves their simulation in the GUI, a new file will be created in the userSave or an old file will be updated.

An example of the file structure is described below in JSON.



```
1  {"name":["example"],
2  "date":"2024-04-26",
3  "people":[[[268.5,273.5],[0,0]]], //people contains an array of pairs, first one is position,second one is velocity
4  "bricks":[100,300], //array of brick position
5  "params":[
6    ["avoi",92.26856438595755],
7    ["rad",16.81924324888867],
8    ["agl",1.0210153187832691],
9    ["mass",12.738382892887287],
10   ["sek",0.18859598447829982],
11   ["mxs",0.4134772448454521],
12   ["brk",3.3236317819852195],
13   ["mxf",0.21342599354168395],
14   ["doorLength",137.978895697977],
15   ["doorWidth",61.78531598835199],
16   ["sep",0.9755450188468753]],
17  "doorPos":[0,28]} //door position
```

• Testing

I ended up not using any unit test during the implementation since the goal of the program was not very clear and it was not worth the time to write a unit test.

I mainly tested the program using the features in the GUI and observed whether the people exhibit desirable behavior in different case.

I did much less test than planned before mainly because first, I didn't encounter many errors during the implementation nor did I have enough time to do all the testing.

• Known bugs and missing features

Bugs:

1. the people in the room could sneak out of the room through corners when the speed is high enough. This is due to the algorithm I used for calculating avoidance. This issue is also reported before [4], as stated in this article, to handle these cases one can either hard code logic for specific cases, or use a more complicated path-finding algorithm.

2. I would not call this a bug, but different parameters can result in different behaviors in the people, for example if the weight of the avoidance is too low then a person could penetrate the wall. If Separation is too low, then they bump into each other very often.

3. When the radius of people is set to be very small then it can also go through the wall, this is actually expected but may come across as a bug. A easy fix would be link the radius of the brick circle with the radius of the people. But that would look strange in the GUI(the wall being to thin), and the program is not intended to simulate very small characters anyway.

4. this only happens in one of my laptops: if I leave the program running in the back stage for too long, sometimes it crashes.

Missing features

1. pop up error window in case of file corruption. If given more time I would create some test class for the files and create try-and-catch structure in the program to make it more fixable.

2. reminders to save the simulation upon quitting the program, somehow, I could not achieve this by onCloseRequest in scalafx, I need to do more research about the library.

3. a help button in the GUI for the user to read the manual.

4. enable the user to delete saves

- 3 best sides and 3 weaknesses

3 best sides:

1. a highly customizable room, the users can conveniently add/ remove people and walls wherever they want. (you can use the door as an eraser for people)

2. a working save system

3. reasonably efficient algorithm so the room can accommodate hundreds of people even in a modest computer.

3 weaknesses.

1. little error handling is presented in the programs

2. code not very well commented

3. the GUI class is too clunky and should be divided into sub-parts

- **Deviations from the plan, realized process and schedule**

The plan did not match very well with the reality as I am typing this on the last date of the deadline. I didn't do much during the early stage of the implementation, spent quite some time on some failed algorithm, and only in the last week did I start to adding the functionalities to the GUI. The total amount of time I spent on (about 40 hours) the project is considerably less than planned. I probably should have spent more time planning and made an everyday routine to do the project.

- **Final evaluation**

The process of implementing this program taught me a lot about how to implement a simple GUI, save system and the importance of external libraries. It also motivates me to learn more about linear algebra and to get better at time management.

In terms of the program itself, it successfully displays all the desired behaviors given correct parameters. And it is also fun to experiment on because of the customizability it offered.

If I am asked to improve this program, I will strive to implement a better path finding algorithm and use threads to make it more efficient.

- References and code written by someone else

[1] <https://com-lihaoyi.github.io/upickle/>

[2] <https://www.red3d.com/cwr/steer/gdc99/>

[3] <https://sldsdo.github.io/steering-behaviors/>

[4] this is from the same article as [3]

All the code in this program is written by me.

- **Appendices**

<https://version.aalto.fi/gitlab/wuy26/stampede.git>

