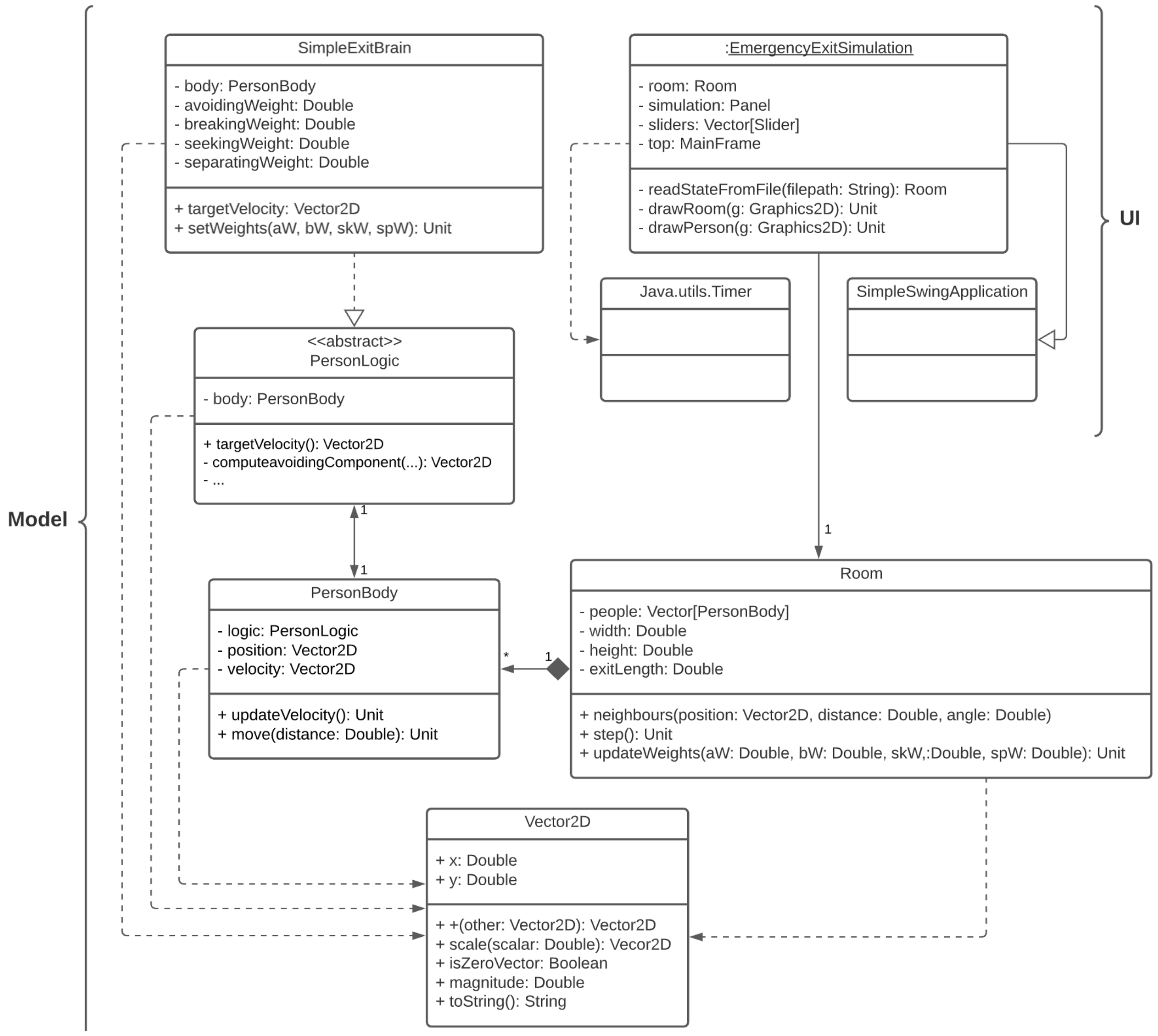
Emergency exit simulation

Technical project plan

Project by Georgy Ananov (890980)
Bachelor's program in Data Science
First year
16.02.2020

1 Class structure

The structure of this project can be broken down into two main components: the model and the user interface. The model is responsible for running the simulation, while the UI is in charge of presenting the current state of the simulation in way that is easy to visually interpret for a human user, as well as allowing the user to interact with the simulation parameters.


SimpleExitBrain
- body: PersonBody
- avoidingWeight: Double
- breakingWeight: Double
- seekingWeight: Double
- separatingWeight: Double
+ targetVelocity: Vector2D
+ setWeights(aW, bW, skW, spW): Unit
:EmergencyExitSimulation
- room: Room
- simulation: Panel
- sliders: Vector[Slider]
- top: MainFrame
- readStateFromFile(filepath: String): Room
- drawRoom(g: Graphics2D): Unit
- drawPerson(g: Graphics2D): Unit
UI
Java.utils.Timer
SimpleSwingApplication
<<abstract>> PersonLogic
- body: PersonBody
+ targetVelocity(): Vector2D
- computeavoidingComponent(...): Vector2D
- ...
Model
1
1
1
PersonBody
- logic: PersonLogic
- position: Vector2D
- velocity: Vector2D
+ updateVelocity(): Unit
+ move(distance: Double): Unit
*
1
Room
- people: Vector[PersonBody]
- width: Double
- height: Double
- exitLength: Double
+ neighbours(position: Vector2D, distance: Double, angle: Double)
+ step(): Unit
+ updateWeights(aW: Double, bW: Double, skW,:Double, spW: Double): Unit
Vector2D
+ x: Double
+ y: Double
+ +(other: Vector2D): Vector2D
+ scale(scalar: Double): Vecor2D
+ isZeroVector: Boolean
+ magnitude: Double
+ toString(): String

1.1 Model

The central two classes in the model are the PersonBody and PersonBrain. PersonBody class represents the physical body of a simulated human, its interactions with the environment of the room and the bodies of other people. The body is capable of moving within the room, changing its velocity according to the directions of the brain, as well as surveying its surroundings and supplying the acquired information to the brain.

PersonBrain's main task is determining the direction in which the person desires to go, according to the rules of the simulation. As the need for simulating people with varying steering behaviors (more on those later) may arise, PersonBrain is declared abstract, so that classes with specific implementations (such as SimpleExitBrain) may extend this class. I have considered serval ways to implement the way a specific subclass of PersonBrain might operate in terms of class structure. Since the algorithm that brain is based on calculates its output by first computing several independent components, and then combining them in some way, I thought there might be opportunities for meaningful division of labor between specialized classes or traits. So far, I can conclude that such a division would ne unnecessary, due to a highly limited number of different descendants of PersonBrain. The current plan is to implement component-calculating methods in the abstract class, while implementing specific methods of combining components is left for descendants of the abstract.

The people's bodies exist within a room. The Room is then responsible for aggregating the people and helping them in navigating around the room. Many of the steering behaviors require information about some aspects of the simulated room (such as the location of the nearest wall, the direction towards the exit, or the velocities of nearby persons). The Room class has functionality that helps PersonBody instances obtain this information, so that their brains have all the data they need to follow the rules of the simulation. The Room class is also responsible for advancing the simulation with the step() method, which includes updating the velocity of each person and advancing the position of each person.

All of these classes make use of the java.vecmath.Vector2d data structure, which represents a simple two-dimensional geometric vector or a point on a 2D plane. The structure includes some basic vector arithmetic methods, such as summing and scaling vectors.

1.2 User Interface

In this project, the User Interface is responsible for essentially bringing the model to life by initializing the starting state, repeatedly calling its step function, and displaying a graphical representation of the room and the people inside of it. The design of the interface is currently at its early stages, as I need to do more research into concurrent programming in Scala. The way I envision the structure currently is that the UI application will maintain three threads – one for advancing the simulation, one for drawing it on the screen, and one for listening to user input.

I plan to utilize two different Java.utils.Timer class instances for running the simulation and drawing threads, as this class provides a fairly straight-forward way to run simulation updating code and redrawing code at even time intervals. An alternative way to go about running the simulation would be to look for a way to run it at full speed (without artificial delays introduced by the Timer). This path would require some way of measuring the time passed between each two concurrent updates of the simulation, so that the distance of each person can be adjusted in accordance with the time passed (similar to how it is often implemented in Unity2D games for instance). Without this adjustment the movement might become uneven and jittery.

2 Use case

A fire safety inspector is reviewing the proposed architectural blueprints of a new cinema theatre. As part of the review process, the inspector needs to make sure that the cinema is equipped with sufficient fire exists, so that in case of an emergency, the people inside the can evacuate before the fire becomes a threat to their safety.

The inspector launches the Emergency Exit Simulation Application. A menu pops up, propping the user to either specify a filepath that leads to a file containing information about the cinema hall, or enter the parameters of the room manually into dedicated text fields. The inspector chooses to input the parameters manually, fills the text fields, and presses an “OK” button. At this point the inputted string values are evaluated, and, if some of the fields contain data that does not conform to the expected format (for instance, the inspector inputted a decimal value into the “number of people” field), a message pops up, propping the user to reenter the data.

After the initial state parameters have been accepted, the application creates a Room object using those parameters. Since no data for initial positions of people in the room have been supplied, the UI application passes a vector of given length with random positions to the Game constructor. After that the simulation thread and the drawing thread are started. Every few milliseconds the simulation thread calls the step() method of the Room, which causes all of the PersonBody instances to request an updated desired velocity from their brains. After all of the desired velocities have been updated, each PersonBody moves slightly in the new direction using the move() method. The positions of the PersonBody instances are now updated. In the meantime, the drawing thread goes through the list of PersonBody instances in the Room instance, and adds them to a temporary graphic2D object. Once all of the persons are drawn, the graphic is displayed in the GUI window. The user can now see people moving on the screen.

While observing the simulation, the inspector decides that, due to the pandemic, people might try to prioritize keeping a safe distance slightly higher than usual. The user bumps up the weight coefficient of the separation steering behavior using a slider in the GUI. At this point, the event listening thread in the UI application captures the change, and calls the updateWeights(…) method of the Room instance. In turn, the Room instance propagates the request to all of the PersonBody.brain instances. People now behave in a slightly different manner on the user’s screen. This process is repeated until the user is satisfied with the way people behave.

Now the inspector restarts the simulation with a “restart” button in the GUI. Simulation thread and drawing thread are stopped, a new Room instance is created in place of the old one, with the same room parameters and starting state. Simulation thread is resumed.

The inspector times the simulation and concludes that the evacuation was not completed quickly enough. The user restarts the simulation again, this time adjusting the size of the exit doorway. This time the inspector is satisfied with the simulated evacuation time and closes the program. Use case complete.

3 Algorithms

The simulation is based on steering behaviors described in the article ”Steering behaviors for autonomous characters” by Craig Reynolds. Each individual person is modeled by several basic rules, that determine the direction in which the person will strive to move. Below are 4 of these rules, also referred to as “steering behaviors”, that are used in emergency exit simulation:

1. Braking. A person will slow down upon observing people in front of them moving slower than itself.
2. Seeking. A person is attracted to the doorway.
3. Avoiding. A person will try to avoid the walls of the room.
4. Separating. A person will try to avoid getting too close to other people.

The velocity components obtained from these steering behaviors are combined in some way, which outputs a resulting desired velocity vector. The exact formulas for each of the behaviors, as well as the rule according to which the components are combined, are currently not finalized. The simulation application itself will serve as a testing ground for trying out different ways to implement the exact mechanics of these behaviors and the combining rule. Here are some ideas for building the first iteration of each of these unknowns:

1. Braking. A person will “look” at all other persons that are located within a specific distance and specific angle of itself. Based on how many of such persons are detected, and how slow they are moving, a scaling factor is determined, which is applied to the final desired velocity vector.

2. Seeking. This one is fairly straight-forward. The seeking component vector is aimed at a point within the exit doorway. The exact point of aim can be determined by, for example, looking at the ratio of people above the person to the number of people below the person.

3. Avoiding. This rule prevents people from facing into walls. The mechanism of this steering behavior is described thoroughly in the article, I will quote it here: “The vehicles test the space ahead of them with probe points <…> When a probe point touches an obstacle, it is projected <…> to the nearest point on the surface of the obstacle <…> and the normal to the surface at that point is determined <…> Steering is determined by taking the component of this surface normal which is perpendicular to the vehicle's *forward* direction.

4. Separating. This component can be calculated by simply surveying the neighboring area for other people, and applying a repelling vector for each person found. The magnitude of the repelling vector scales with 1/R, where R is the distance between the two persons.

At last, for the combination rule I will use a simple weighted sum and clipping of the resulting vector to a desired maximum magnitude. Now that we have the desired velocity, all we need to do to obtain the actual acceleration of a person is to take the difference between the current and the desired velocities, and clip the result with some maximum acceleration.

4 Data structures

One external data structure that I will be using in this project is java.vecmath.Vector2d. The use of this structure simplifies the many vector calculations that are performed throughout the simulation. I was unable to find a suitable data structure for Scala specifically, so I am using a java class.

Another noteworthy question is whether it might be worth it to switch the structure used for storing the persons in the Room from Vector to Buffer. There are some benefits to be gained from the ability to, for example, delete persons who successfully left the room, as this might lighten the simulation. However, after careful consideration, I concluded that those benefits can be obtained without switching to a mutable data structure.

5 Schedule

The development of this project can be divided into several phases.

5.1 Basic GUI

As the simulation is very difficult to test without some basic GUI, the first stage of the project is to implement a basic canvas, and learn to draw simple shapes on said canvas. After that is complete, it would make sense to implement a “skeleton” of the simulation classes Room and PersonBody. We can then move on to developing the code for drawing the persons on the canvas. Once we have that complete, we can move on to further developing the UI application.

I estimate the workload to up to 8 hours. I would hopefully have this done by the first bi-weekly checkup.

5.2 Threads

Here we need to delve deeper into concurrent programming with Scala, and develop the part of the UI responsible for periodically calling the step() method of the Room class (the method itself does not need to be fully implemented for now), as well as periodically redrawing the canvas. Simple placeholders for the step() method can be implemented here for basic testing.

I expect a lot of debugging at this phase, so the workload could easily go up to 16 hours including research. I hope to make good progress on this phase by the second bi-weekly checkup, and have this phase completely finished by the third checkup at the latest.

5.3 Implementing the PersonBrain

This is likely going to be the most challenging and the most meaningful part of the project. Here I will need to implement the 4 rules that govern the behavior of each person. Many different implementations will need to be experimented with.

Workload estimation - many sleepless nights. I hope to have this built to a passable state by the end of March.

5.4 Finishing and testing the GUI

Here I will need to implement the on-the-fly adjustments of the simulation with sliders, as well as functionality connected with reading from a file. A lot of testing anticipated.

I estimate the workload to be about 15-20 hours. Should be done by the mid-April checkup.

5.5 Polishing, adding extra features

By the time the project reaches this stage, I will likely be out of time. Nevertheless, if it so happens that it's not presenting time yet, there is plenty of additional features to be added (listed in the general plan). Improvements to the visual component of the GUI can also be implemented at this stage.

6 Testing plan

Much of the testing of the logic of the simulation would have to be done visually through the GUI. As there is no "true" answer as to what exact outputs the core logic methods (such as targetVelocity() in PersonBrain) should output, the main testing criteria will be achieving a realistic simulation.

That said, a lot of the methods that serve to support the core logic of the simulation can be tested effectively. Especially methods that help the HumanBody obtain information about its surroundings. Here I will unitize unit testing, as it is absolutely crucial that these methods work as intended. Errors here may cause undebuggable faults in the way the simulation runs.

Some components of the UI can also be put through unit tests. As far as I know there exist libraries that specifically aid testing the User Interface, which would come in very handy in making sure that all parts of the UI work as intended. File-reading functionality can also easily be put through unit tests.

7 References

- ”Steering behaviors for autonomous characters” by Craig Reynolds

  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/
- Vector2d class documentation https://docs.oracle.com/cd/E17802_01/j2se/javase/technologies/desktop/java3d/forDevelopers/j3dapi/javax/vecmath/Vector2d.html
- ScalaTest library https://www.scalatest.org/