ID2209 Assignment 1

Conor Gallagher, Edwin Sundberg, Benedict Wolff

November 13, 2025

1 Running Instructions

No additional inputs are required to run the program- open `main.gaml` in GAMA, and begin the experiment.

2 General Overview

This assignment involves a basic simulation of a festival scenario. Guests who become hungry or thirsty report to an information center to receive directions to the nearest food or water store. We have modelled this with three types of agents: `Guest`, `InformationCenter`, and `Store`. The interactions between these agents are detailed in the following section.

3 Base Functionality

3.1 Explanation

The `Guest` species has three primary attributes which control their behavior: `food` and `water` (both floats between 0-100), and `targetStore`:

```
species Guest skills: [moving] {
    // randomize hunger/thirst levels to start in range [10, 100]
    // for each guest, and decrease at varying rates
    float food <- rnd(50.0, 100.0) update: food - rnd(0.1, 1.0);
    float water <- rnd(50.0, 100.0) update: water - rnd(0.1, 2.0);
    Store targetStore <- nil;
```

As is visible, their food/water levels are decreased by a varying amount at each cycle. We define an agent as hungry or thirsty when their respective value falls below 20, which will trigger them to seek out a suitable store for replenishing their attribute. If they're already on the way to a store (`targetStore != nil`), they will continue, else they will visit the information center to find out where the nearest store is. In contrast, if they're neither hungry nor thirsty, they will wander around:

```
reflex move {
    if (isHungry() or isThirsty()) {
        if (targetStore = nil) {
            // guest is hungry/thirsty and hasn't gotten location of target
            // store yet from InformationCenter or cache
            if (distance_to(self, infoCenter) < 1.0) {
                // guest is within range to ask InformationCenter for nearest store
                targetStore <- askForTargetStore();
```

```
                do goto target: targetStore;
            }
            // guest isn't within range to ask, keep moving towards InformationCenter
            else {
                do goto target: infoCenter;
            }
        } else {
            do goto target: targetStore;
        }
    } else {
        do wander;
    }
}
```

Note that `infoCenter` is an attribute of the global species, making it accessible to all agents. The `askForTargetStore` action of the Guest species communicates with the information center to find out where the nearest food/water store is, depending on the need of the guest:

```
Store askForTargetStore {
    Store store <- nil;
    ask InformationCenter {
        if (myself.isHungry() and myself.isThirsty()) {
            store <- self.getNearestStore(myself);
        } else if (myself.isHungry()) {
            store <- self.getNearestFoodStore(myself);
        } else {
            store <- self.getNearestWaterStore(myself);
        }
    }
    return store;
}
```

Once a `targetStore` is set, the guest will make their way to the store, replenishing the attribute when they get close:

```
reflex eat when: targetStore != nil and distance_to(self, targetStore) < 1.0
    and targetStore.hasFood {
        food <- 100.0;
        targetStore <- nil;
 }

reflex drink when: targetStore != nil and distance_to(self, targetStore) < 1.0
    and targetStore.hasWater {
        water <- 100.0;
        targetStore <- nil;
    }
```

3.2 Implementation

The entire implementation, including code only for this base functionality, is shown below:

```
model festival

global {
    int guestNumber <- 20;
    int foodStoreNumber <- 2;
    int waterStoreNumber <- 2;
    InformationCenter infoCenter;

    init {
        create Guest number: guestNumber;
        create InformationCenter {
            infoCenter <- self;
        }
        create Store number: foodStoreNumber {
            hasFood <- true;
        }
        create Store number: waterStoreNumber {
            hasWater <- true;
        }
    }
}

species Guest skills: [moving] {
    // randomize hunger/thirst levels to start in range [10, 100] for each guest, and decrease at
    // varying rates
    float food <- rnd(50.0, 100.0) update: food - rnd(0.1, 1.0);
    float water <- rnd(50.0, 100.0) update: water - rnd(0.1, 2.0);
    Store targetStore <- nil;

    bool isHungry {
        return food < 20;
    }

    bool isThirsty {
        return water < 20;
    }

    reflex move {
        if (isHungry() or isThirsty()) {
            if (targetStore = nil) {
                // guest is hungry/thirsty and hasn't gotten location of target store yet from InformationCenter or cache
                if (distance_to(self, infoCenter) < 1.0) {
                    // guest is within range to ask InformationCenter for nearest store
                    targetStore <- askForTargetStore();
                    do goto target: targetStore;
                }
                // guest isn't within range to ask, keep moving towards InformationCenter
                else {
                    do goto target: infoCenter;
                }
            } else {
                do goto target: targetStore;
            }
        } else {
            do wander;
        }
    }

    reflex eat when: targetStore != nil and distance_to(self, targetStore) < 1.0 and targetStore.hasFood {
        food <- 100.0;
        targetStore <- nil;
    }

    reflex drink when: targetStore != nil and distance_to(self, targetStore) < 1.0 and targetStore.hasWater {
        water <- 100.0;
        targetStore <- nil;
    }

    Store askForTargetStore {
        Store store <- nil;
        ask InformationCenter {
            if (myself.isHungry() and myself.isThirsty()) {
                store <- self.getNearestStore(myself);
            } else if (myself.isHungry()) {
                store <- self.getNearestFoodStore(myself);
            } else {
                store <- self.getNearestWaterStore(myself);
            }
        }
        return store;
    }

    aspect base {
        rgb guestColor <- #green;
        if (isHungry() and isThirsty()) {
            guestColor <- #red;
        } else if (isHungry()) {
            guestColor <- #orange;
        } else if (isThirsty()) {
            guestColor <- #yellow;
        }

        draw circle(1) color: guestColor;
    }
}
```

Figure 1: Base implementation (1)

```
species InformationCenter {
    action getNearestStore(agent guest) {
        Store nearest <- closest_to(Store, guest);
        return nearest;
    }

    Store getNearestFoodStore(Guest guest) {
        list<Store> foodStores <- Store where (each.hasFood);
        return closest_to(foodStores, guest);
    }

    Store getNearestWaterStore(Guest guest) {
        list<Store> waterStores <- Store where (each.hasWater);
        return closest_to(waterStores, guest);
    }

    aspect base {
        draw hexagon(3) color: #salmon;
        draw "InfoCenter" color: #black at: location + {-3, 3};
    }

}

species Store {
    bool hasFood <- false;
    bool hasWater <- false;

    aspect base {
        draw triangle(2) color: (self.hasFood ? #darkgoldenrod : #darkblue);
        draw self.hasFood ? "food" : "water" color: #black at: location + {-1, 2};
    }
}

experiment festivalSimulation type:gui {
    output {
        display festivalDisplay {
            species Guest aspect:base;
            species InformationCenter aspect:base;
            species Store aspect:base;
        }
    }
}
```

Figure 2: Base implementation (2)

3.3 Demonstration

We first show the initial state of the simulation in Figure 3: an information center, two stores of each type, and 20 guests are present. All guests are green, meaning they are neither hungry nor thirsty:

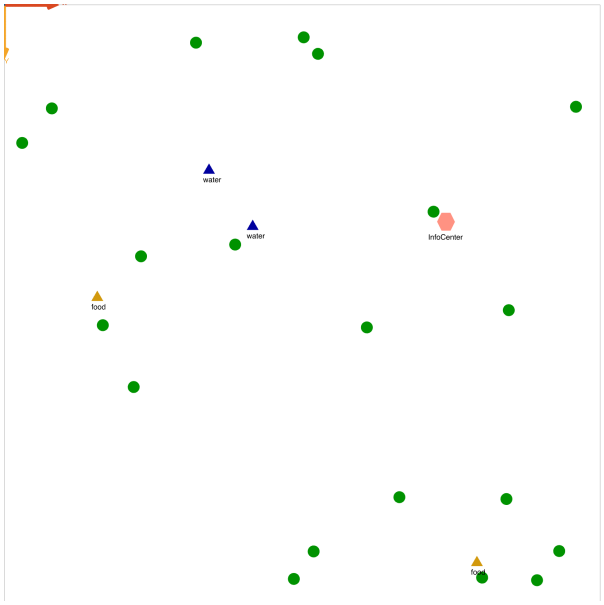


Figure 3: Starting state of simulation

Secondly, we show the state of the simulation after 500 cycles in Figure 4: guests are much less spread out as they fall into paths of shortest routes between stores and the information center, and they have varying levels of hunger (orange), thirst (yellow), or both (red).

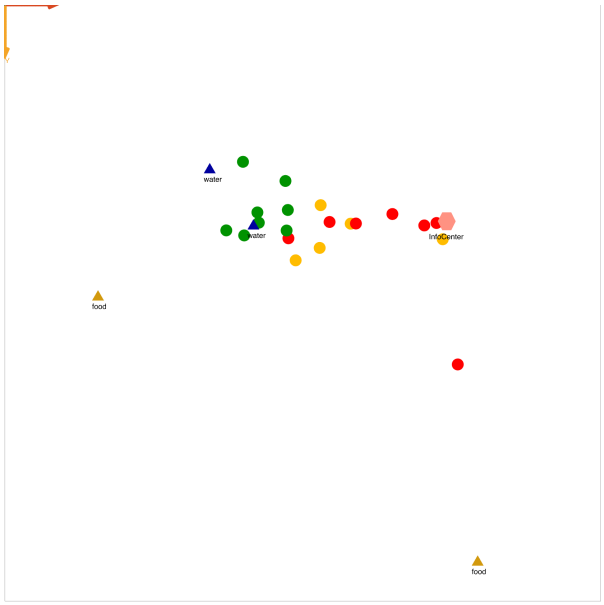


Figure 4: Intermediate state of simulation

4 Challenges

4.1 Challenge 1: Memory of Agents - Small Brain

4.1.1 Explanation

This challenge was about adding memory/"a small brain" to the agents, allowing them to remember the locations of visited shops, but with a chance to forget occasionally. To achieve this, we added a cache to guest agents where they store locations of stores, as well as a step counter for tracking how many productive steps (non-wandering movement) the guest takes in total to allow distance comparison between agents with/without memory. An attribute `useCache` allows us to enable/disable this memory on a per-agent basis.

4.1.2 Implementation

The most important piece of new logic in the Guest species is the action for applying the cache, and when to occasionally forget it. `applyCache` checks if the agent is hungry/thirsty, and if they have their cache enabled and don't currently have a target store, will apply whatever stores may be in their memory. This action is executed before the normal "go to ask" flow so that a `targetStore` will be set if a location is accepted. Similarly, at each cycle we define that there is a small probability of the guest forgetting their cached food/water store.

```
bool useCache <- false;
int steps <- 0;
Store cachedFood <- nil;
Store cachedDrink <- nil;
...
// small chance to forget
reflex forget when: flip(0.01) {
    string forgets <- one_of(["food", "water"]);
    if (forgets = "food") {
        cachedFood <- nil;
    } else {
        cachedWater <- nil;
    }
}

action applyCache {
    if (targetStore != nil or useCache = false) {
        return;
    }
    if (isHungry() and cachedFood != nil) {
        targetStore <- cachedFood;
    }
    else if (isThirsty() and cachedWater != nil) {
        targetStore <- cachedWater;
    }
}
```

The second relevant snippet is for storing locations of stores in its cache/memory once it has visited a store:

```
action cacheStore(Store store) {
```

```
    if (store.hasFood) {
        cachedFood <- store;
    }
    if (store.hasWater) {
        cachedDrink <- store;
    }
}
```

4.1.3 Demonstration

Use Case 1 In Figure 5, we can see that guest agents with memory cover significantly less distance on average traveling to stores or the information center than those without (1485 vs 1901 after 2000 cycles). This is because in many cases, they can travel the shorter, direct distance to the nearest store of their need without having to first travel to the information center.

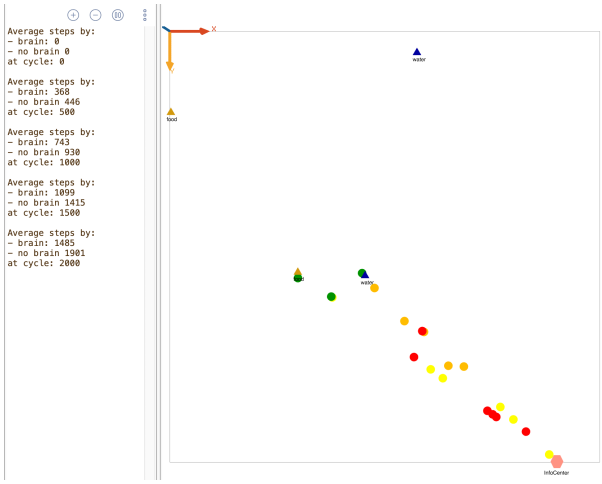


Figure 5: Difference in steps with/without brain

Showing another aspect, in Figure 6 guests with a brain can be seen traveling directly between stores, without having to go to the information center as an intermediate step; this would not be observed without a memory.

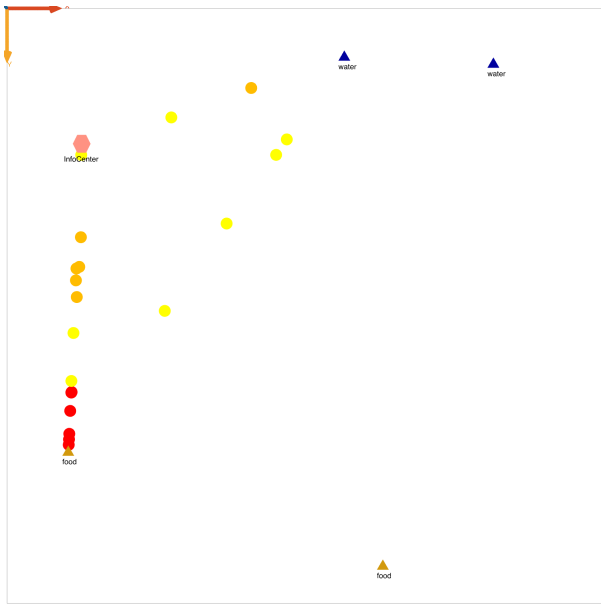
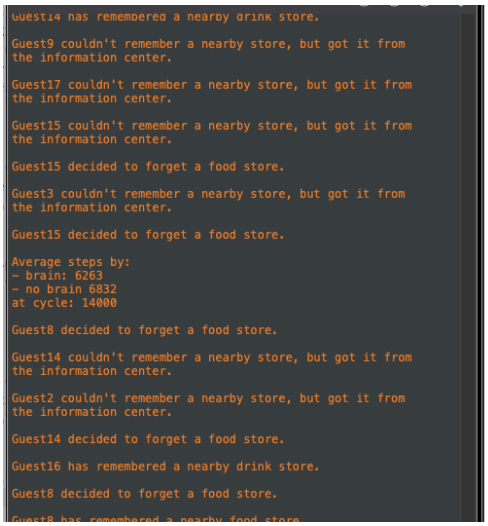


Figure 6: Guests with brain traveling direct between stores

Use Case 2

In Figure 7 Agent 8 decides to forget their memorized food store and will instead return to the information center to get new information on where he can find a food store once hungry.



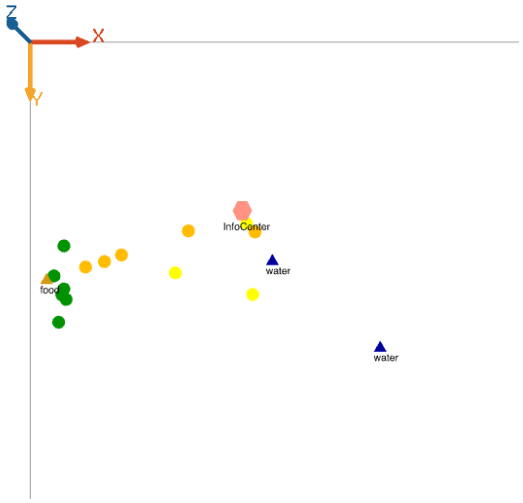


Figure 7: Displays the log output of agent which decides to forget the location of their memorized food-store and will instead ask for this information again

Use Case 3

Similarly to Case 2, Figure 8 shows another agent who decides to forget where to get drinks and therefore will go to the information center to ask for a new location once they are thirsty.

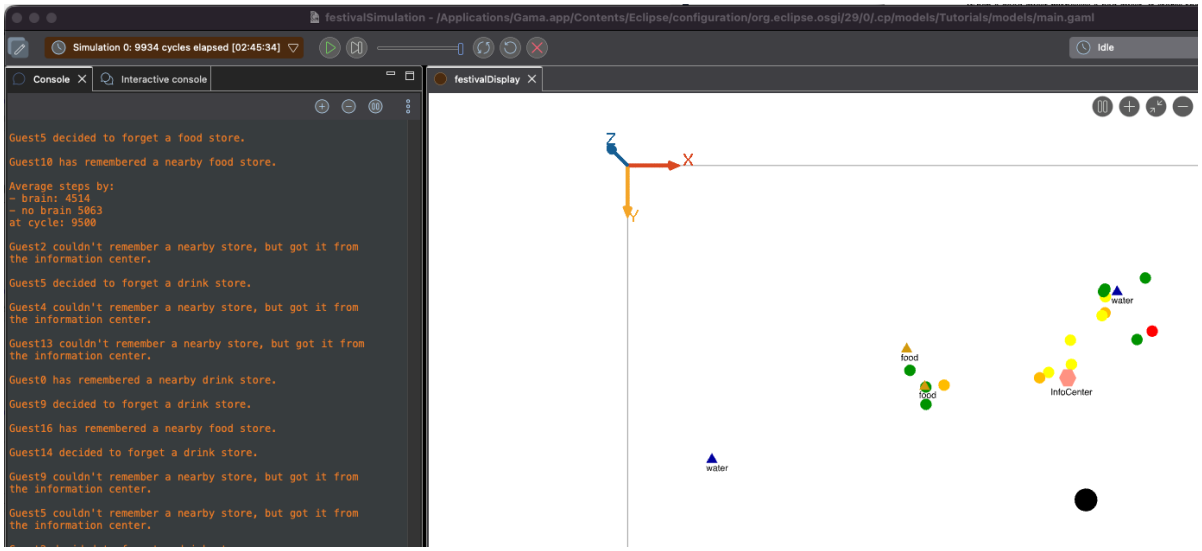


Figure 8: Displays agents which decides to forget location of drink stores and subsequently events when they went to the info-center to collect this information.

Use Case 4

Also shown in Figure 8, the agents with memory enabled attempted remember a nearby store, but if they couldn't they logged that they had to go to the information center for directions.

4.2 Challenge 2: Removing Bad Behavior Agents

4.2.1 Explanation

As part of this challenge, bad guests were introduced that can be reported by good guests. For that, a good guest needs to be close to a bad guest. The good guest will then take note of the bad guest and go to the information center to report the bad guest. The information center then calls a guard and informs him about the bad guests, at which point the guard will track them down to arrest them.

4.2.2 Code

We added a Guard species that can move around the festival, take reports from the information center, and arrest festival visitors. The boolean attribute `isCalled` controls whether there are reported guests at the information center to find out about, while the list of bad guests to remove is stored in `targets`. If this list is non-empty, the guard travels to the first guest in the list with the intent of arresting them.

```
species Guard skills: [moving] {
    bool isCalled <- false;
    list<Guest> targets <- [];

    reflex move {
        // prioritise handling existing bad guests over getting new reports
        if (!empty(targets)) {
            Guest target <- targets[0];
            do goto target: target;
            if (distance_to(self, target) < 1.0) {
```

```
                do arrest(target);
            }
        } else if (isCalled) {
            do goto(target: infoCenter);
        } else {
            do wander;
        }
    }

    reflex getReports when: distance_to(self, infoCenter) < 1.0 {
        ask InformationCenter {
            myself.targets <- union(myself.targets, self.reportedGuests);
        }
        isCalled <- false;
    }

    action arrest(Guest target) {
        write target.name + " has been arrested by the guard.\n" ;
        ask target {
            do die;
        }

        // remove target from targets
        targets >> target;

        ask infoCenter {
            do handleGuestArrest(target);
        }
    }

    aspect base {
        draw circle(2) color: #black;
    }
}
```

In addition, good Guest agents were equipped with the ability to witness and report bad guests. When a good guest witnesses a bad guest, it stores that bad guest in its `guestsToReport` list and heads to the information center. Once it arrives, it shares its findings.

```
reflex witness when: !isBad {
    // when a good guest gets close to a bad guest, report them
    list<Guest> badGuests <- Guest where (each.isBad and distance_to(each, self) < 3.0
        and !(guestsToReport contains each));

    guestsToReport <- guestsToReport + badGuests;
}

reflex report when: distance_to(self, infoCenter) < 1.0 and !empty(guestsToReport) {
    ask InformationCenter {
        do handleGuestsReport(myself.guestsToReport);
    }
    guestsToReport <- [];
```

```
}
```

Lastly, the information center takes those findings, and adds them to its reports (taking care to avoid duplicate reports from different guests). These reports are then shared with a guard after they are called and arrive at the center. When the guard successfully arrests one of the bad guests, it calls the `handleGuestArrest` action of the information center to allow for the removal of that guest from the target list.

```
action handleGuestsReport(list<Guest> badGuests) {
    // while guest was travelling to infoCentre to report, bad guest could have been arrested
    badGuests <- badGuests where (!dead(each));

    list<Guest> newBadGuests <- badGuests - reportedGuests;
    if (!empty(newBadGuests)) {
        write "Guests reported: " + collect(newBadGuests, each.name) + "\n";
    }

    // avoid duplicates when multiple guests report same bad guests
    reportedGuests <- reportedGuests union badGuests;
    ask Guard {
        self.isCalled <- true;
    }
}

// remove arrested guest from reportedGuests
action handleGuestArrest(Guest guest) {
    reportedGuests >> guest;
}
```

4.2.3 Demonstration

The initial state of the simulation solution to challenge 2 is shown in the following figure 9. Good guests are circles, the guard is displayed as a larger circle, and bad guests are represented as squares.

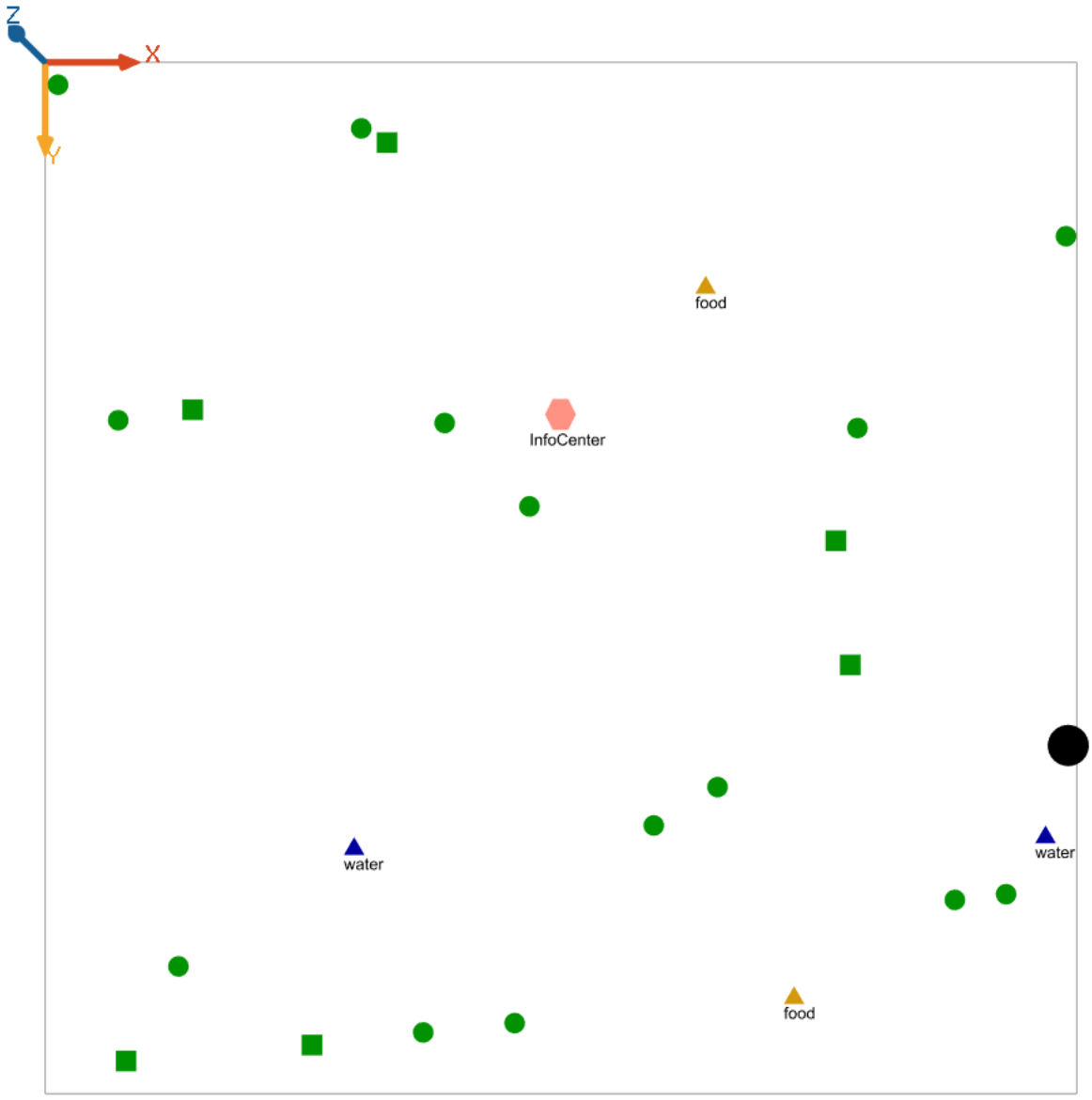


Figure 9: Initial state of challenge 2 simulation

The following four use cases highlight the feature-completeness of the given implementation in regard to the requirements defined as part of the second challenge. The input is the same for all four use cases:

- 20 guests
- 20% chance for a guest being bad
- 2 food stores
- 2 water stores

- 1 information center
- 1 guard

Use Case 1

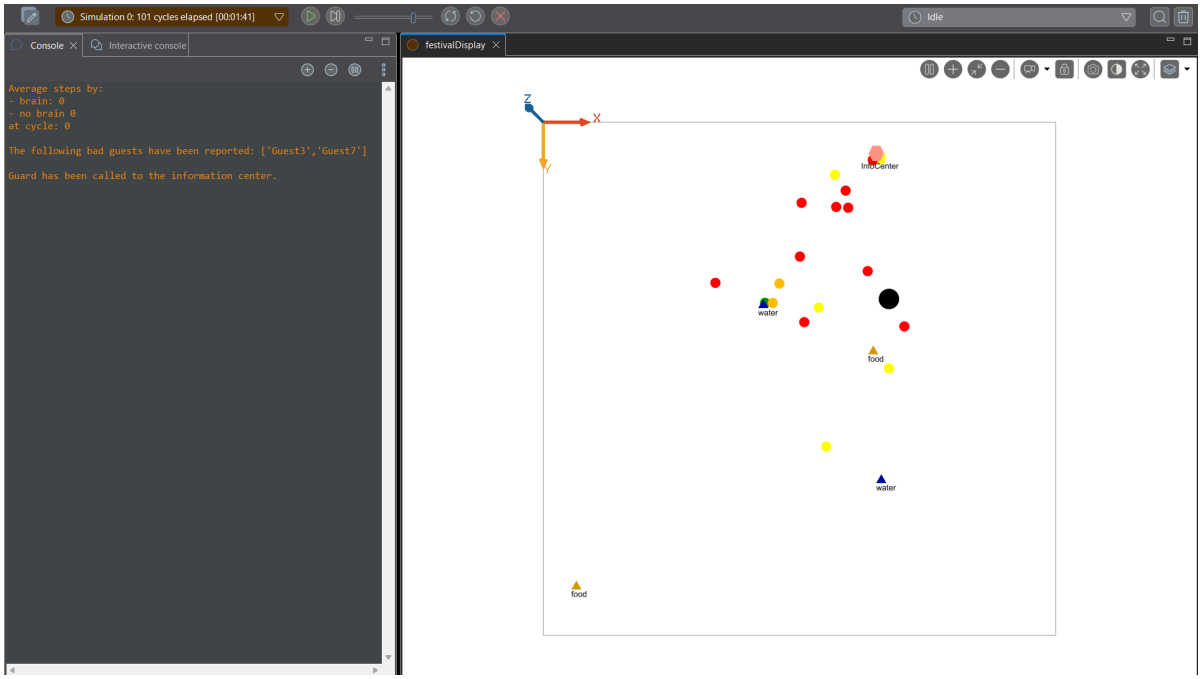


Figure 10: A good guest reports two bad guests to the information center, which calls the guard

Use Case 2

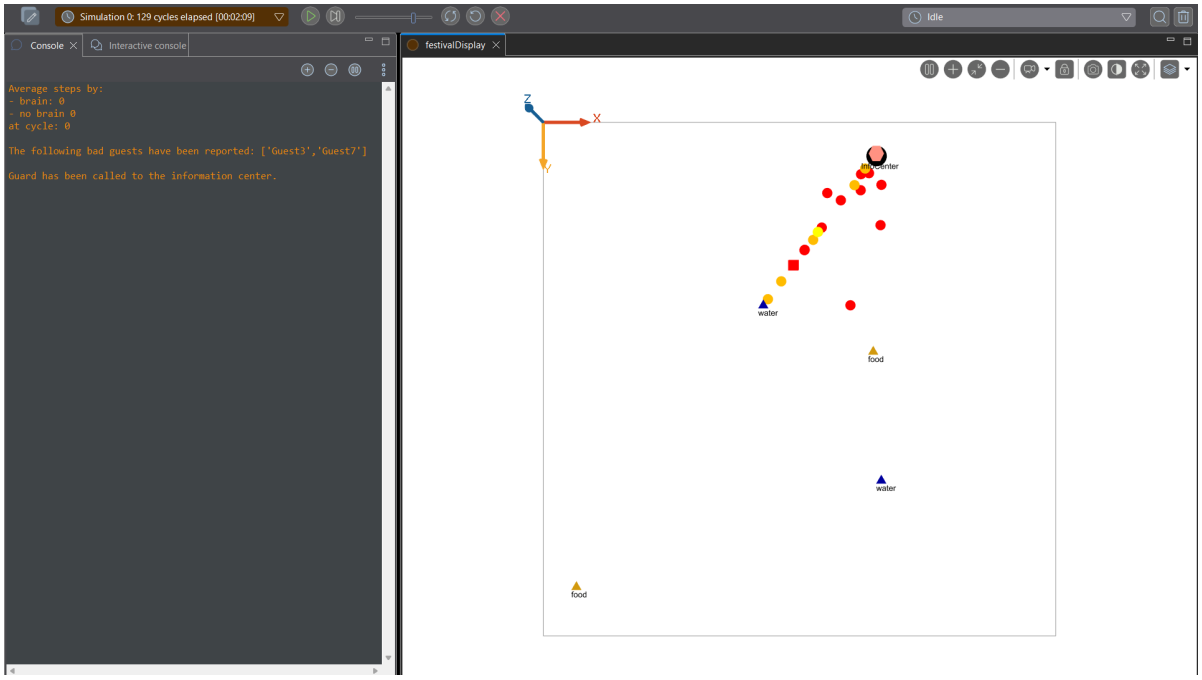


Figure 11: The guard reports to the information center and receives its two targets

Use Case 3

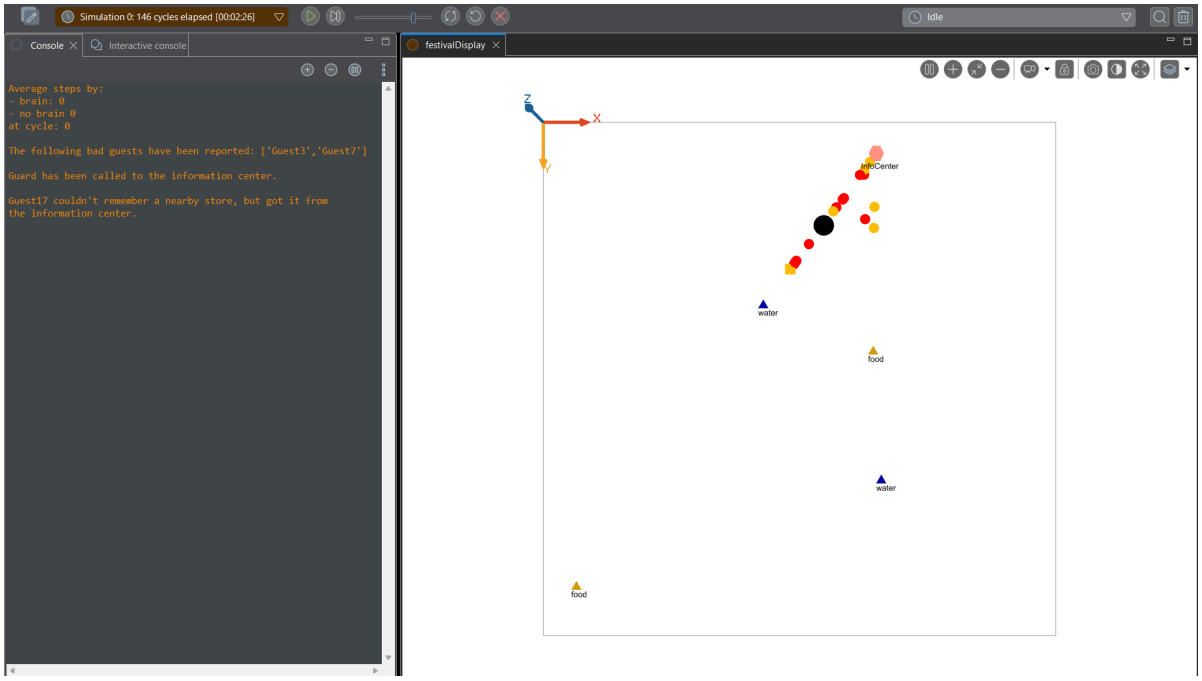


Figure 12: The guard makes his way to arrest the two bad guests at once

Use Case 4

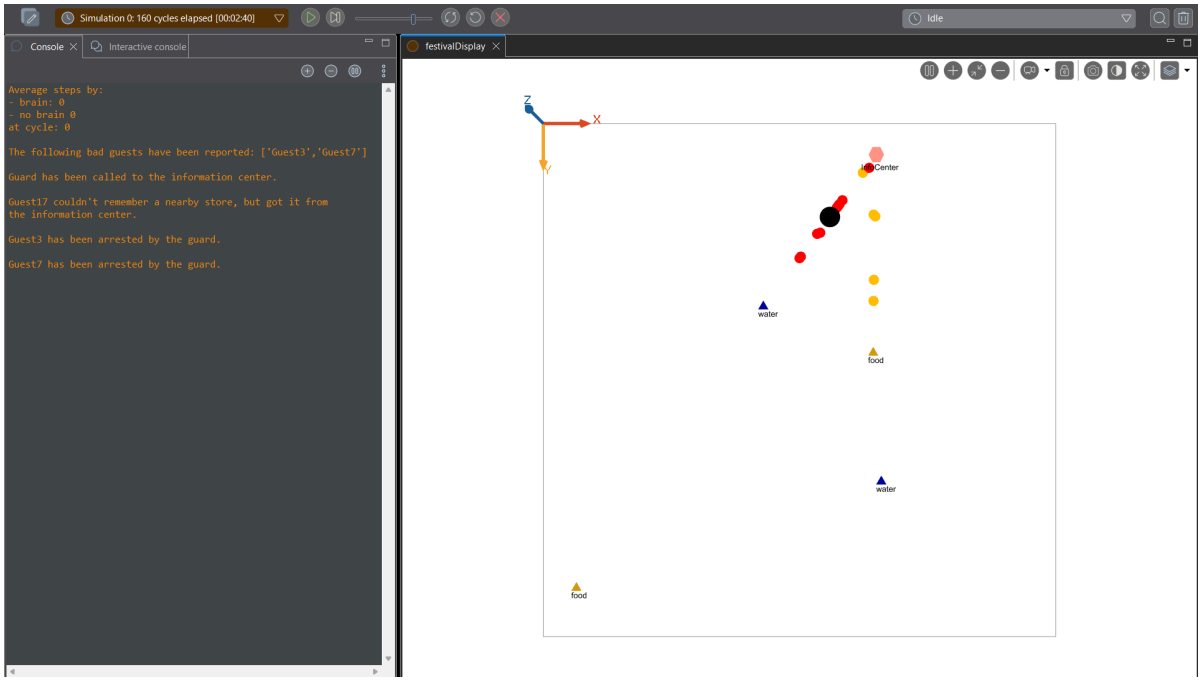


Figure 13: The two bad guests have been arrested by the guard

5 Final Remarks

In conclusion, we believe that we have effectively addressed all parts of the requirements, including the additional challenges. The implementation has given us exposure to many different and important aspects of the GAMA platform, which has proved surprisingly easy to learn and interact with. We particularly appreciate the built-in handling of much of the GUI, which provides a very nice feedback loop for iterating on solutions when desired behavior is not being seen. We do not see any clear shortcomings/limitations of our approach and believe it to be quite robust.