Effects of Population Size on Food-collecting Efficiency in Simulated Swarms of Pheromone-guided Robots

N. A. Dahlfors[1] and **E. Lundell**[1]

[1] *Norra Real, Stockholm*

ABSTRACT

In swarm robotics the goal is that many simple robots should be able to complete advanced tasks by working together. In order to do this many types of communication have been proposed, one of which is inspired by the laying of pheromone trails seen in ant colonies. To design a system with a specific function in mind, information about how the number of agents might affect the outcome is essential. We studied the effect of group size on the efficiency with which pheromone-guided robots gathered food in different environments through computer simulations. Our results showed that the swarm's efficiency first increased non-linearly with group size, but after reaching an apex decreased and even went below that of robots lacking any means of communication. This highlights challenges in designing and scaling swarm robotics systems.

Contents

1. INTRODUCTION

1.1 Background

Accordning to Brambilla et al. (2013), swarm robotics is a field of robotics inspired by behavioural properties of social animals such as ants and bees. Dorigo et al. (2020) states that it is a relatively new field, having gained prominence only in the 21-st century, but with the goal of making multi-robot systems more viable than single-robot ones in a number of aspects. Dorigo et al. (2020) lists many potential uses of the technology, predicting the first use of swarm robotics in space missions to take place between 2030 and 2040, allowing for greater search areas and the possibility of constructions on other planets, and Navarro & Matía (2013) see possibilities in areas such as demining.

One of the main characteristics of social animals that swarm robotics tries to emulate is that of scalability, which Brambilla et al. (2013) describes as the ability to perform a task well in differing group sizes. This property is of course desirable when the task in question can damage a robot (as space, and especially mines, can do), since the removal of individual robots, henceforth called **agents**, does not hinder the general performance of the group. It also makes robot swarms more flexible than single robot systems when conditions change. Dorigo et al. (2020) state that some robot swarms should consist of millions of individuals and they note the work done by Rubenstein et al. (2014) and Slavkov et al. (2018) in the area of managing large swarms, implying that this should be possible. However, for many approaches to swarm behaviour in robotics, research on pure scalability is hard to find.

This is the case for the method of **Ant colony optimisation**. Ant colony optimisation, or **ACO**, is a method of finding optimal paths inspired by ants. Ants lay down pheromone trails when they find paths to food or other good things that other ants can follow. When other ants follow the path, they also lay down pheromone which reinforces good paths. This, according to Dorigo et al. (2006), is more or less the way ACO works, but they also state that there are lots of variations.

Garnier et al. (2007) tested both real and simulated systems of 1, 2, 3, 4, 5 and 10 agents with pheromone communication. They concluded that an optimal number of robots was required to achieve an effective collective path choice, but it is worth noting the small number of agents in the study. Fujisawa et al. (2014) performed similar simulations but with a larger number of agents, ranging from 1 to 40. They measured the number of pheromone secretion events and found that these increased when the swarm size increased. They also found that the increase seemed to diminish as the number of agents increased. They then ran five simulations with agent sizes between zero and 250, that confirmed the diminished increases on this interval. According to Fujisawa et al. (2014) this is a usual phenomenon in swarm robotics which Krieger et al. (2000) believe to be the result of overcrowding.

Fujisawa et al. (2014) conclude their paper by emphasising that non-linear increase in performance must be taken into account when dealing with pheromone-guided agents. They also speculate that this could be a source of emerging complex swarm intelligence and is worthy of further investigation.

1.2 Purpose

The purpose of this experiment is to explore behaviour in pheromone-guided swarms. Because intercommunication between agents is essential for complex behaviour and larger swarm sizes have exponentially increasing interactions, there might be unforeseen consequences when swarms reach critical sizes. Exploring patterns related to this is therefore imperative in understanding when these systems are viable and applicable to real use cases.

1.3 Research Question

How does the number of agents in a swarm affect the efficiency with which it collects food in various environments using pheromone communication?

1.4 Method

The experiment was carried out as a simulation of agents existing on a map of square cells, each cell being a pixel in an image. There were no collisions between agents, meaning that multiple agents could inhabit the same cell simultaneously without affecting each others trajectories. The positions of the agents were represented by floating point values, leading to multiple possible agent positions per cell. Each cell could be either empty, a wall cell, a food cell, or a home cell. An empty cell represented open space and was capable of containing agents and pheromone. A wall cell represented an impassable obstacle, these cells neither contained agents nor pheromone. A food cell contained one bit of food. If an agent walked into this cell it became an empty cell and the agent picked up the food. A home cell acted as an empty cell except when an agent carrying food stepped on it, in which case the agent returned to a state of not carrying food and a counter of collected food increased by one, while leaving the home cell unchanged.

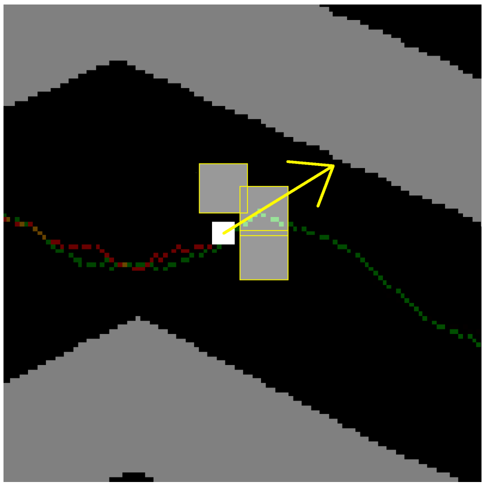


Fig. 1: An agent surrounded by walls that is carrying food, leaving a red pheromone trail behind it, and following a green pheromone-trail left by another agent. The detection-zones are highlighted with yellow outlines and the forward-direction is marked with an arrow.

The agents were modelled as infinitely small dots, either carrying food or not, and moving at constant speed in a forward direction defined by an angle relative to the horizontal axis. Each time an agent updated it left a pheromone trail behind it, green if the agent was not carrying food and red if the agent was carrying food, which strength on the map reduced over time. The initial strength of the pheromone trail left behind the agent was also linearly reduced from full strength to zero during 5000 simulation steps after the agent touched either a food or home cell. When the agent touched a food or home cell this timer was reset and the pheromone was laid with full strength again. The strength of the pheromone was reduced over time both on the map, and with which is was laid down.

To update the direction, in each simulation step, the agent sampled the pheromone concentration in three areas in front of it, as seen in Fig. 1: forward-left, forward-middle, and forward-right separated by $\frac{\pi}{3}$ radiance. Agents only focused on one type, green pheromone if the agent was carrying food and red pheromone if the agent was not carrying food, with walls registered as large negative pheromone values and food or alternatively home as large positive values depending on the agent's food status. They then did one of two things depending on where the highest concentration was found. If the highest concentration was found right in front of the agent, in the forward-middle zone, the agent did not change its forward direction. If the highest concentration was found in the forward-left or forward-right zone the agent turned in the relevant direction by a random amount between 0 and $\frac{\pi}{2}$ radiance. The agent then, independently of the pheromone, applied a random offset to its angle between -0.2 and 0.2 radiance. When the simulation was run without pheromones all zones detected zero pheromones and only the negative wall pheromone and the random offset factor affected the direction of an agent. The update was done with the following lines of code. The code has been simplified to become clearer, the full version can be found in Appendix C. The random function used was gold noise which generated values between 0.0 and 1.0. All trigonometry functions used radians.

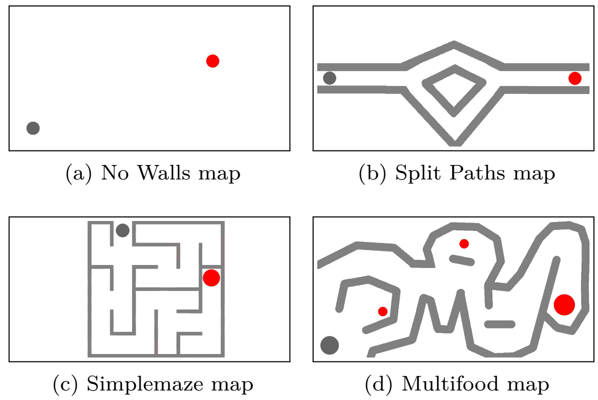
```
// Move by current angle and speed
agent.position += vec2(
    cos(agent.angle), sin(agent.angle)
  ) * 1.5;

// Sence feromone concentration
vec4 left = sence(agent.position +
  vec2(
    cos(agent.angle + PI / 3.0),
    sin(agent.angle + PI / 3.0)
  ) * senceDistance);
vec4 middle = sence(agent.position +
  vec2(
    cos(agent.angle),
    sin(agent.angle)
  ) * senceDistance);
vec4 right = sence(agent.position +
  vec2(
    cos(agent.angle - PI / 3.0),
    sin(agent.angle - PI / 3.0)
  ) * senceDistance);

// Extract relevant pheromone color
float f_left =
  (agent.hasFood == 1 ? left.g   : left.r);
float f_right =
  (agent.hasFood == 1 ? right.g  : right.r);
```

```
float f_middle =
  (agent.hasFood == 1 ? middle.g : middle.r);

// Decide which direction to turn
float f_turning = 0.0;

if (f_middle > f_left && f_middle > f_right)
{
    f_turning = 0.0;
}

if (f_left > f_middle && f_left > f_right)
{
    f_turning = random() * PI * 0.5;
}

if (f_right > f_middle && f_right > f_left)
{
    f_turning = -random() * PI * 0.5;
}

// Turn agent
agent.angle += f_turning +
    (random() - 0.5) * 0.4;
```

The simulation was written in C++ and GLSL (OpenGL Shader Language). OpenGL is a cross plattform API usually used for rending 2D and 3D graphics but was here utilised to dispatch compute shaders to the GPU (Graphics Processing Unit). The GLEW library was employed to load the available OpenGL extensions. The OpenGL context, map, and agents were initialized in C++. First, the map was loaded from an image where specific pixel values represented different objects according to the table below.

Object	Red	Green	Blue
Food	255	0	0
Home	100	100	100
Wall	128	128	128
Empty	0	0	0

The agents were then created on a random home cell with a random direction and the OpenGL context was created. The code then moved into the update loop where a series of compute shaders simulated the agents' behaviour and weakened the pheromones on the map. The agents were simulated as described above, and the pheromone (with strength represented by floating point values between 0.0 and 1.0) was linearly reduced with a speed of 0.000667 strength/simulated step, resulting in a reduction from full strength (1.0) to zero in 1500 simulated steps. See Appendix B and C for the source code.



(a) No Walls map (b) Split Paths map

(c) Simplemaze map (d) Multifood map

Fig. 2: The four maps tested

The collection of data was handled by the C++ code. Every 240th simulated step a new line was added to a log file containing the current timestamp and the total food *collected* at that point. Only food returned to the home was counted as collected, it was not enough for an agent to pick it up. Either when the agents had collected more than 99.5% of the food or when the simulation had run for more than 163200 updates, the simulation finished. The complied program can be modified using command-line arguments detailing what map to use and how many agents to create. A batch file, see Appendix D, was used to automate the process of running the program multiple times with different parameters. The simulation was run for four different maps, seen in Fig 2, and each map was run 40 times for each agent count. The number of agents analysed were 2^1, 2^2, 2^3, 2^4, 2^5, 2^6, 2^7, 2^8, 2^9, 2^{10}, 2^{11}, 2^{12}, 2^{13}, 2^{14}, and 2^{15}. This resulted in a total of $4 \times 40 \times 15 = 2400$ simulations, which took an approximate of 80 hours to finish, but the time required will vary depending on the computer on which it is being run. Since the simulation did not use real time but simulated steps, it did not matter how much real time each update took. Thus the computing power of the computer running it only affected the time required to collect the data and not the data itself.

To answer the research question, two values, called *collexiency* - see definition 1 - and *relative collexiency* - see definition 2 - were measured. It was the total amount of food collected divided by the number of simulated steps it took, scaled up with a factor of 100. If the agents collected more than 99% of all

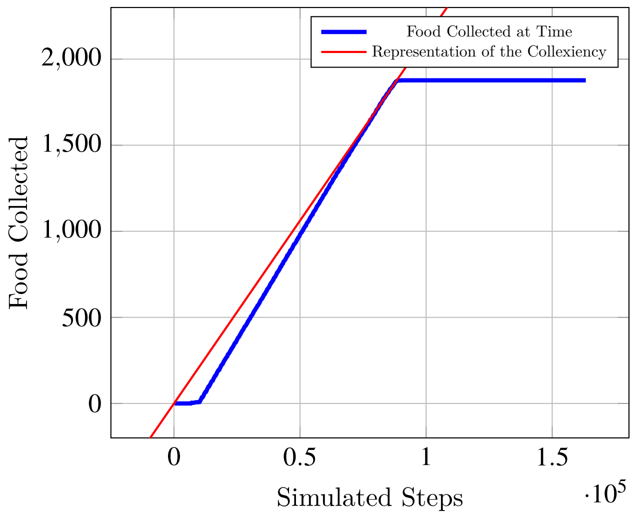


Fig. 3: The food collected over time on the map Split Paths with 32 agents from one of the simulations and a linear graph with the calculated collexiency as inclination.

the food on a specific map, the collexiency would be 99% of the food divided by the number of simulated steps needed to reach that point. This way the collexiency was less affected by the last few agents struggling to find the home after the pheromone trail has collapsed. As seen in Fig 3 where only the relevant part of the simulation and food collection was included in the collexiency. The relative collexiency is the collexiency per agent, i.e. the collexiency divided by the number of agents.

Definition 1. Let the *collexiency*, (denoted ✱), of a run of the simulation be the inclination of the linear graph showing food collected over time, up to the first 99% of the food on the map.

Definition 2. Let the *relative collexiency*, (denoted ℟), of a run of the simulation be the *collexiency* divided by the number of agents.

To calculate the collexiency and relative collexiency for all the runs of the simulation and to calculate the average values and standard deviations for each map and agent count, a python script was used. It calculated the collexiency and relative collexiency according to the definition above and compiled it into a LaTeX-friendly format. See Appendix D for the python script.

2. RESULTS

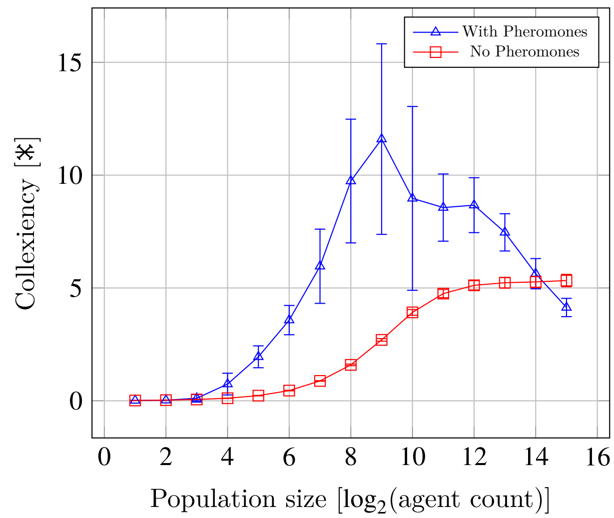


Fig. 4: Average collexiency and standard deviation per population size for pheromone- and non-pheromone-seeking agents on the map No Walls.

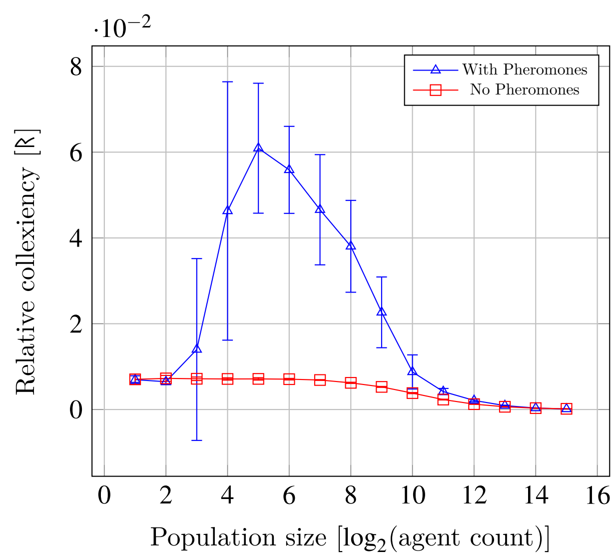


Fig. 5: Average relative collexiency and standard deviation per population size for pheromone- and non-pheromone-seeking agents on the map No Walls.

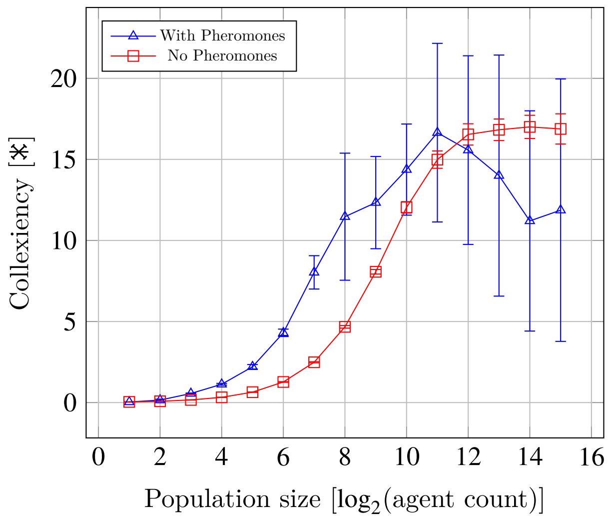


Fig. 6: Average collexiency and standard deviation per population size for pheromone- and non-pheromone-seeking agents on the map Split Paths.

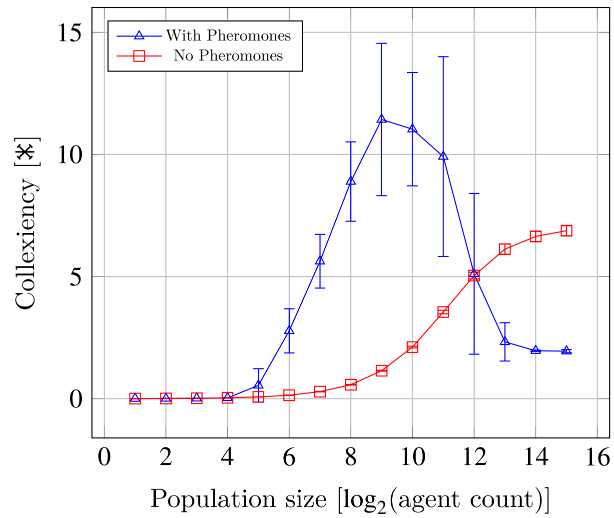


Fig. 8: Average collexiency and standard deviation per population size for pheromone- and non-pheromone-seeking agents on the map Simplemaze.

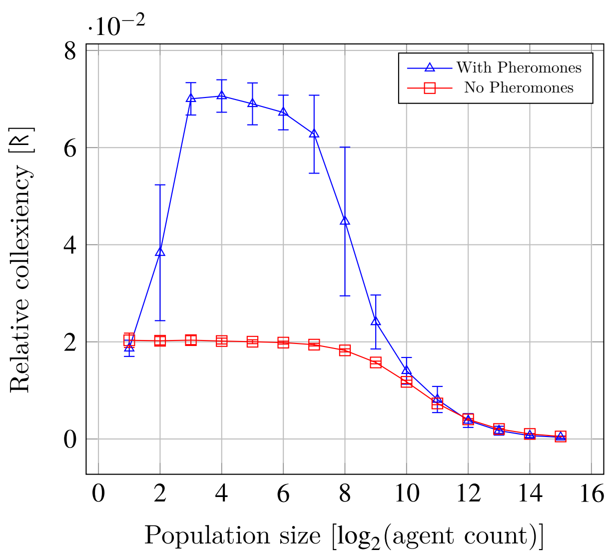


Fig. 7: Average relative collexiency and standard deviation per population size for pheromone- and non-pheromone-seeking agents on the map Split paths.

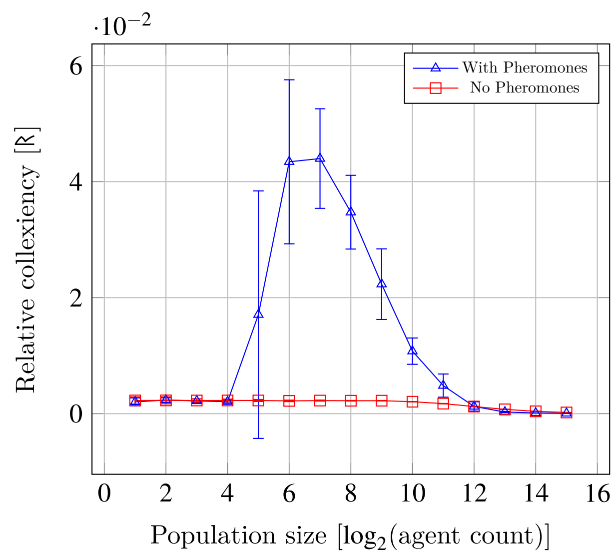


Fig. 9: Average relative collexiency and standard deviation per population size for pheromone- and non-pheromone-seeking agents on the map Simplemaze.

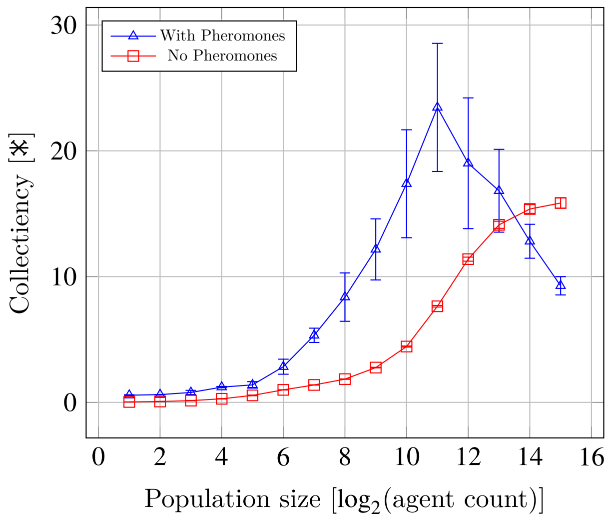


Fig. 10: Average collexiency and standard deviation per population size for pheromone- and non-pheromone-seeking agents on the map Multifood.

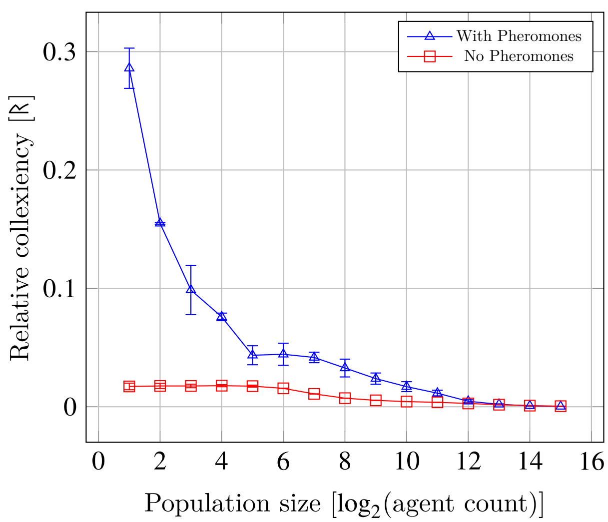


Fig. 11: Average relative collexiency and standard deviation per population size for pheromone- and non-pheromone-seeking agents on the map Multifood.

2.1 No Walls Data

On the No Walls map, pheromone-seeking agents had an average collexiency of approximately zero up until a population of 2^3, after which the average collexiency increased monotonically until reaching a maximal value when the population was 2^9 as seen in Fig 4. The figure also shows that the average collexiency then decreased, attaining a lower average collexiency than that of non-pheromone-seeking agents when the population size was 2^{15}.

The average relative collexiency of pheromone-seeking agents increased steadily until reaching a maximum with a population of 2^5, after which it decreased and reached similar values as those of non-pheromone-seeking agents when the population went above 2^{11} as seen in Fig. 5.

2.2 Split Paths Data

On the Split Paths map, pheromone-seeking agents achieved strictly increasing values of average collexiency until reaching a maximum with a population of 2^{11} as Fig 6 illustrates. Fig 6 also shows that the average collexiency then decreased up to a population of 2^{14}, dropping below that of non-pheromone seeking agents for populations of 2^{12} and above.

The relative collexiency of the pheromone-seeking agents initially increased and then maintained an approximately constant value for populations between 2^3 and 2^7, with a maximum value at 2^4 as Fig. 7. It then decreased monotonically and assumed values similar to that of non-pheromone-seeking agents around a population of 2^{11}.

2.3 Simplemaze Data

On the Simplemaze map, Fig 8 shows that the average collexiency stayed close to zero for population sizes up to 2^4, after which it strictly increased until reaching a maximum at a population of 2^9. The value then decreased, going below that of non-pheromone-seeking agents at the population size 2^{13}, after which it stayed approximately constant.

The average relative collexiency was also zero for populations of sizes up to 2^2, then increasing and attaining maximal value at population sizes of 2^6 and 2^7 as seen in Fig. 9. For larger populations it decreased monotonically up until 2^{12}, after which its value remained constant at around zero along with that of non-pheromone-seeking agents.

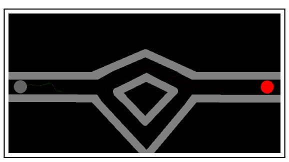
2.4 Multifood Data

On the Multifood map, pheromone-seeking agents achived an average collexiency of about zero until the population size reached 2^5, after which the average collexiency steadily increased, reached a maximum at a population of 2^{11} and then steadily decreased, going below the value of non-pheromone-seeking agents for populations of 2^{14} and 2^{15} as Fig 10 illustrates.

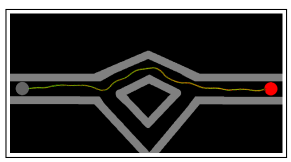
The pheromone-seeking agents attained maximal average relative collexiency for a population size of 2^1, after which the average relative collexiency steadily decreased as Fig. 11 shows.

2.5 Main Results

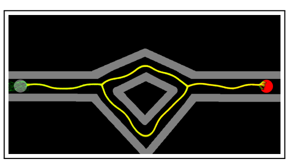
No technical problems were encountered during the simulations and it could clearly be observed that the agents were capable of forming paths, as seen in Fig. 12 and 13.



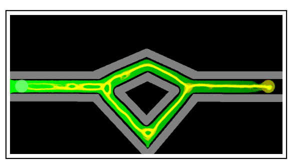
(a) Split Paths with 2^2 agents



(b) Split Paths with 2^4 agents



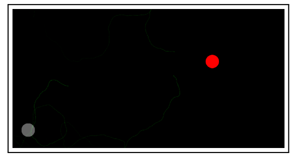
(c) Split Paths with 2^8 agents



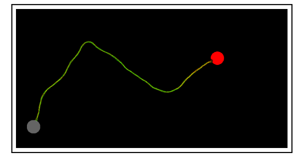
(d) Split Paths with 2^{15} agents

Fig. 12: Typical situations of simulations with different number of agents on the map Split Paths.

Our findings show that, across all maps tested, there existed maximal values of collexiency for pheromone-seeking agents after which the collexiency reduced as agent population increased. They also show that, across all maps tested, for large enough agent populations the collexiency values of pheromone-seeking agents went below those of non-pheromone-seeking agents. This general behaviour was also seen in the relative collexiency across all maps, with the distinction of maximal ᚱ-values occurring for smaller agent populations than maximal ᚼ-values, and differences in ᚱ-values between pheromone-seeking and non-pheromone-seeking agents being negligible for large populations.

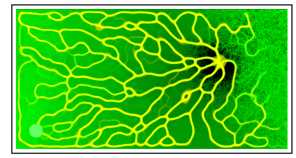


(a) No Walls with 2^2 agents



(b) No Walls with 2^4 agents

(c) No Walls with 2^8 agents



(d) No Walls with 2^{15} agents

Fig. 13: Typical situations of simulations with different number of agents on the map No Walls.

3. DISCUSSION

3.1 Evaluation of Results

From our results we can conclude that both the collexiency and relative collexiency increased with population for low population sizes, but that they decreased for sufficiently large populations. Where this change took place differed from map to map, but the general trend could be observed across all maps. If one accepts collexiency as a good representation of food-collecting efficiency, we believe our results to be significant and relevant in the context of the original research question posed. However, to gain a better understanding of the generality of our results, tests with different values for parameters such as pheromone-trail length, strength and diffusion along with different map sizes are needed. It should be noted that relative collexiency will not be discussed in great detail throughout the rest of the discussion, since its main purpose was to give insight into the trends of the standard collexiency. We do however believe that measurements similar to relative collexiency could be of interest in future studies focusing on maximising performance per cost in robot swarms.

Apart from the existence of collexiency peaks, our most noteworthy results were the phenomenon of diminishing returns observed by Fujisawa et al. (2014), and the fact that the collexiency of pheromone-guided agents not only decreased for large populations, but even went below that of non-pheromone-seeking agents.

We were able to see that, even when the collexiency increased, it did so with smaller and smaller increments in relation to the number of agents added. This is the same observation reported by Fujisawa et al. (2014), although they used a different metric than collexiency. Our findings thus strengthen the research of Fujisawa et al. (2014), and show that the trends they observed do not seem to be unique to their measured parameters or their limited population sizes.

The fact of pheromone-guided agents performing worse than non-pheromone-guided ones for large populations was more unexpected, since it to our knowledge has not previously been reported. We believe both it and the more general decrease in collexiency to be the result of the maze-like structure of trails occurring for the large populations, se Fig. 13. These could make agents travel in loops with only a small probability of choosing the right sequence turns to make it home. More research is needed in order to confirm that these structures can indeed account for the remarkable drops in performance, but we believe that it is possible to build models of its effects through, for example, random walks in weighted graphs.

We believe that the results we have discussed are reliable, since a large number of runs were conducted and the phenomenons all could be observed in differing environments. It is however worth noting that in some situations, especially for the larger populations on the Split Paths map, the standard deviations were quite high. We do not believe that this compromises our results since the general trends tended to be much larger than the standard deviation intervals, and we note that the large standard deviations often occurred in conjunction with rapid changes in collexiency. This could be explained if there were transition points between two different emerging trail structures yielding widely different collexiency values, were both of them occurred in some noteworthy capacity. If that is the case it would give much insight into the impact of trail-structure on behaviour and is thus a prime target for further research. One could for example investigate if the collexiency values for specific population sizes are clustered around specific values, or if they are distributed in other ways.

3.2 Evaluation of Method

There are three areas in which we deem there to be a high risk of systemic errors. These are the way we measure efficiency, the lack of collision between agents - and therefore lack of collision handling -, and the large number of arbitrarily chosen parameters where only a small change in a key variable can yield widely different swarm behaviour, notable parameters are the trail-length, dictated by pheromone diffusion speed, and the agent turning speed.

The way we chose to measure efficiency, with collexiency - the derivative of the linear approximation how much food is carried from the food to the home, see definition 1 -, does, as mentioned earlier, differ from metrics used in previous research in this field by Fujisawa et al. (2014) who measured the number of pheromone strengthening events to determine the efficiency. Based on our results where we could see the same falloff in efficiency as the agent count increased as Fujisawa et al. (2014) we conclude that collexiency is a valid measurement of efficiency. We also believe that it, in addition to being valid, is easier to adapt to different simulation environments and agent algorithms, since it directly measures how well the task in performed. It would for example not be possible to measure the efficiency of the non-pheromone following agents if the variable to be analysed was pheromone interactions, even though the non-pheromone agents might solve the task in an adequate way. There seems to also be an correlation between relative collexiency - the collexiency per agent, see definition 2 -, and the number of pheromone-reinforcing events when normalised to the number of foraging events, another variable that increase linearly with agent count, found by Fujisawa et al. (2014). Because our work was purely computational no real meaning come from the size of the collexiency and relative collexiency, speeding up or slowing down the simulation, changing what is considered one second in simulated time, will scale the resulting values. Should this work have be redone care should be put into choosing realistic values for agent movement and turning speed in m/s. The maps would consequently have to be realistic sizes in m. The food should either be measured in kg or in units of what one agent can carry.

The lack of collision detecting in our simulation makes it more difficult to directly connect the results to the real world. Especially for the runs with more agents, where much efficiency would be lost due to the large number of agents not having enough space in the narrow pheromone paths formed. As the simulation was carried out it is comparable to swarm-robotics with very small agents or flying robots that can exist on top of or very close to each other, together with a well functioning collision handling system that allows interactions to go smoothly. The reduced complexity of the simulation without collisions compared to one with collisions can also be beneficial when calculating optimal performance, assuming that collisions cannot increase the swarms performance. Although, collisions might induce different behaviour, such as forcing the agents to use wider paths which might make them find the local minimum faster, since more neighbouring options will be tested and the fastest option will be reinforcing more, but time will also be wasted on worse paths and the capacity of the local-optimal path found will be limited. A future study comparing collision handling systems and their effects on agent behaviour is necessary to determine whether such a system that yield similar results in efficiency to what we found exists or not.

Another important parameter in the simulation that play a large role in determining the swarms efficiency and behaviour is pheromone diffusion. Faster diffusion would lead to shorter trails, which combined with for example slower turning speed would make the agents more unlikely to form paths and thus require a higher number to increase the chance of a path forming. Similarly to how the speed must be normalised using SI units the diffusion of a real life pheromone material should either be measured and then implemented, or, if the behaviour of the real life material used can be adjusted, realistic values be chosen in the simulation or optimal parameters be found in a realistic range through tests.

3.3 Conclusions

We have illustrated problems that might emerge when pheromone guided swarm systems are scaled up to large population sizes. We have also highlighted the importance of appropriate knowledge of each problem, because of the wild variation depending on the environmental situation. This can hopefully lead to better decisions in future designing of swarm robot systems and inspire further research in scalability and efficiency measurements.

4. SOURCES

Brambilla, M., Ferrante, E., Birattari, M., & Dorigo, M. (2013). Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence*, *7*(1), 1–41.

Dorigo, M., Birattari, M., & Stutzle, T. (2006). Ant colony optimization. *IEEE computational intelligence magazine*, *1*(4), 28–39.

Dorigo, M., Theraulaz, G., & Trianni, V. (2020). Reflections on the future of swarm robotics. *Science Robotics*, *5*(49).

Fujisawa, R., Dobata, S., Sugawara, K., & Matsuno, F. (2014). Designing pheromone communication in swarm robotics: Group foraging behavior mediated by chemical substance. *Swarm Intelligence*, *8*(3), 227–246.

Garnier, S., Tache, F., Combe, M., Grimal, A., & Theraulaz, G. (2007). Alice in pheromone land: An experimental setup for the study of ant-like robots. In *2007 ieee swarm intelligence symposium* (pp. 37–44).

Krieger, M. J., Billeter, J.-B., & Keller, L. (2000). Ant-like task allocation and recruitment in cooperative robots. *Nature*, *406*(6799), 992–995.

Navarro, I., & Matía, F. (2013). An introduction to swarm robotics. *International Scholarly Research Notices*, *2013*.

Rubenstein, M., Cornejo, A., & Nagpal, R. (2014). Programmable self-assembly in a thousand-robot swarm. *Science*, *345*(6198), 795–799.

Slavkov, I., Carrillo-Zapata, D., Carranza, N., Diego, X., Jansson, F., Kaandorp, J., … Sharpe, J. (2018). Morphogenesis in robot swarms. *Science Robotics*, *3*(25).

5. APPENDIX

5.1 A. Project Setup

The entire project is uploaded to github.com where all of the code, maps and binaries are included, aswell as build configuration for Visual Studio 2019. To run the project please clone it from there and compile it from within Visual Studio 2019. A graphics card with support for OpenGL 4.5 or higher is required.

Github: `https://github.com/logflames/ant-simulation`

File Struture

```
/ant-simulation
├── Dependencies
│   ├── GLEW
│   ├── GLFW
│   └── STB
└── Ant Simulation
    ├── res
    │   ├── exportedcsvs
    │   ├── shaders
    │   └── textures
    └── src
```

5.2 B. Application Code

The parts of the code that are solely used for debugging and running are marked with a faded background. These parts can be removed without affecting the data collection.

```cpp
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <stb_image.h>

#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <vector>

#define ASSERT(x) if (!(x)) __debugbreak();
#define GLCall(x) GLClearError();\
    x;\
    ASSERT(GLLogCall(#x, __FILE__, __LINE__))
#define LOG_ERROR() logError(__FILE__, __FUNCTION__, __LINE__)

#define PI 3.14159265358979f

int AGENT_COUNT = 3000;

std::string MAP_PATH = "res/textures/testmap_simplemaze_1024x512.png";

#define EXPORTED_CSVS_FOLDER "res/exported_csvs/"
```

```cpp
std::string logFileName = "results.csv";
#define SAVE_DATA_EVERY_N_ROUNDS 240
#define END_SIMULATION_AFTER_N_ROUNDS 163200

#define FOLLOW_GREEN_FEROMONE "true"
#define FOLLOW_RED_FEROMONE "true"
#define AVOID_WALLS "true"

bool SAVE_DATA = false;

/* If this variable is one the simulation will end when all the food has been collected (or the maximum number of rounds has been reached)
   If this variable is greater than one the simualtion will end according to the END_SIMULATION_AFTER_N_ROUNDS variable (since it is impossible for the ants to collect more food than is on the map) */
const double END_SIMULATION_AFTER_FOODP_COLLECTED = 0.995;

static void GLClearError()
{
    while (glGetError() != GL_NO_ERROR);
}

static bool GLLogCall(const char* function, const char* file, int line)
{
    while (GLenum error = glGetError())
    {
        std::cout << "[OpenGL Error] (" << error << "): " << function << " " << file << ":" << line << std::endl;
        return false;
    }
    return true;
}

inline void logError(const char* file, const char* func, int line)
{
    std::cout << "Error: " << func << " " << file << ":" << line << std::endl;
}

struct ShaderProgramSource {
    std::string VertexSource;
    std::string FragmentSource;
};

static ShaderProgramSource ParseShader(const std::string& filepath)
{
    std::ifstream stream(filepath);

    enum class ShaderType {
        NONE = -1, VERTEX = 0, FRAGMENT = 1
    };

    std::string line;
    std::stringstream ss[2];
    ShaderType type = ShaderType::NONE;
    while (getline(stream, line))
```

```
    {
        if (line.find("#shader") != std::string::npos)
        {
            if (line.find("vertex") != std::string::npos)
            {
                type = ShaderType::VERTEX;
            }
            else if (line.find("fragment") != std::string::npos)
            {
                type = ShaderType::FRAGMENT;
            }
        }
        else
        {
            if (type != ShaderType::NONE) {
                ss[static_cast<int>(type)] << line << "\n";
            }
        }
    }

    return { ss[0].str(), ss[1].str() };
}

static std::string ReadFile(const std::string& filepath) {
    std::ifstream stream(filepath);

    if (!stream.good())
    {
        std::cout << "File " << filepath << " does not exists" << std::endl;
    }

    std::string line;
    std::stringstream ss;
    while (getline(stream, line)) {
        ss << line << "\n";
    }

    return ss.str();
}

void findReplaceAll(std::string& data, std::string search, std::string replaceString) {
    while (data.find(search) != std::string::npos) {
        data.replace(data.find(search), search.size(), replaceString);
    }
}

static unsigned int CompileShader(unsigned int type, const std::string& source)
{
    GLCall(unsigned int id = glCreateShader(type));
    const char* src = source.c_str();
    GLCall(glShaderSource(id, 1, &src, nullptr));
    GLCall(glCompileShader(id));

    int result;
    GLCall(glGetShaderiv(id, GL_COMPILE_STATUS, &result));
```

```
    if (result == GL_FALSE)
    {
        int length;
        GLCall(glGetShaderiv(id, GL_INFO_LOG_LENGTH, &length));
        char* message = (char*)_malloca(length * sizeof(char));
        GLCall(glGetShaderInfoLog(id, length, &length, message));
        std::cout << "Failed to compile " <<
            (type == GL_VERTEX_SHADER ? "vertex" : "fragment") <<
            " shader" << std::endl;
        std::cout << message << std::endl;
        GLCall(glDeleteShader(id));
        return 0;
    }

    return id;
}

static unsigned int CompileComputeShader(const std::string& source) {
    GLCall(unsigned int id = glCreateShader(GL_COMPUTE_SHADER));
    const char* src = source.c_str();
    GLCall(glShaderSource(id, 1, &src, nullptr));
    GLCall(glCompileShader(id));

    int result;
    GLCall(glGetShaderiv(id, GL_COMPILE_STATUS, &result));
    if (result == GL_FALSE) {
        int length;
        GLCall(glGetShaderiv(id, GL_INFO_LOG_LENGTH, &length));
        char* message = (char*)_malloca(length * sizeof(char));
        GLCall(glGetShaderInfoLog(id, length, &length, message));
        std::cout << "Failed to compile compute shader" << std::endl;
        std::cout << message << std::endl;
        GLCall(glDeleteShader(id));
        return 0;
    }

    return id;
}

static unsigned int CreateShader(const std::string& vertexShader, const std::string& fragmentShader
    )
{
    GLCall(unsigned int program = glCreateProgram());
    unsigned int vs = CompileShader(GL_VERTEX_SHADER, vertexShader);
    unsigned int fs = CompileShader(GL_FRAGMENT_SHADER, fragmentShader);

    GLCall(glAttachShader(program, vs));
    GLCall(glAttachShader(program, fs));
    GLCall(glLinkProgram(program));
    GLCall(glValidateProgram(program));

    GLCall(glDeleteShader(vs));
    GLCall(glDeleteShader(fs));

    return program;
```

```cpp
}

struct Agent {
    float position[2];
    float angle;
    int hasFood;
    int foodLeftAtHome;
    float timeAtSource;
    float timeAtWallCollision;
    int special;
};

std::vector<Agent> agents;

int main(int argc, char** argv)
{
    time_t randomSeed = std::time(NULL);
    std::cout << "Using random seed: " << randomSeed << std::endl;
    std::srand(randomSeed);

    if (argc > 1)
    {
        if (argc != 4)
        {
            std::cout << "Unexpected number of arguments. Expected 'map path' num_ants export_name";
            return 1;
        }

        SAVE_DATA = true;

        std::string map = argv[1];
        std::cout << map << std::endl;

        int agent_count = std::stoi(argv[2]);

        std::string export_name = argv[3];

        MAP_PATH = map;
        AGENT_COUNT = agent_count;

        logFileName = "result_" + std::to_string(randomSeed) + "_ac" + std::to_string(AGENT_COUNT) + "_" + export_name + ".csv";
        std::cout << logFileName << std::endl;
        std::cout << "Running map: " << MAP_PATH << std::endl << "    with " << AGENT_COUNT << " number of agents." << std::endl;
    }

    agents.reserve(AGENT_COUNT);

    float currentTime = 0.0f;
    float lastTime = 0.0f;
    float deltaTime;

    if (SAVE_DATA) {
```

```cpp
        std::ofstream logFile;

        logFile.open(EXPORTED_CSVS_FOLDER + logFileName, std::fstream::app);
        std::time_t now = std::time(0);
        char* dt = std::ctime(&now);
        logFile << "Started at: " << dt << std::endl;
        logFile << "Using random seed: " << randomSeed << std::endl;
        logFile << "Map: " << MAP_PATH << std::endl;
        logFile << "FOLLOW_GREEN_FEROMONE: " << FOLLOW_GREEN_FEROMONE << std::endl;
        logFile << "FOLLOW_RED_FEROMONE: " << FOLLOW_RED_FEROMONE << std::endl;
        logFile << "AVOID_WALLS: " << AVOID_WALLS << std::endl;
        logFile << "AGENT_COUNT: " << AGENT_COUNT << std::endl;
        logFile << "END_SIMULATION_AFTER_FOODP_COLLECTED: " << END_SIMULATION_AFTER_FOODP_COLLECTED << std::endl;
        logFile << "time,total_gathered_food,gathered_food_since_last_entry,number_of_ants_carrying_food" << std::endl;
        logFile.close();
    }

    GLFWwindow* window;

    /* Initialize the library */
    if (!glfwInit()) {
        LOG_ERROR();
        return -1;
    }

    glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
    glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 5);
    glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);

    /* Create a windowed mode window and its OpenGL context */
    window = glfwCreateWindow(1600, 900, "Ants", NULL, NULL);
    if (!window)
    {
        glfwTerminate();
        LOG_ERROR();
        return -1;
    }

    /* Make the window's context current */
    glfwMakeContextCurrent(window);

    glfwSwapInterval(1);

    if (glewInit() != GLEW_OK) {
        LOG_ERROR();
    }

    std::cout << "OpenGL Version: " << glGetString(GL_VERSION) << std::endl;
    std::cout << "Device Hint: " << glGetString(GL_RENDERER) << std::endl;

    /* Load map */

    stbi_set_flip_vertically_on_load(true);
```

```

    int mapWidth, mapHeight, mapNrChannels;
    const char* constMapPath = MAP_PATH.c_str();
    unsigned char* mapData = stbi_load(constMapPath, &mapWidth, &mapHeight, &mapNrChannels, STBI_rgb_alpha);

    if (!mapData) {
        std::cout << "Failed to load map texture." << std::endl;
        glfwTerminate();
        return 0;
    }

    int screenWidth = mapWidth;
    int screenHeight = mapHeight;

    int foodOnMap = 0;

    /* Find home pixels */
    std::vector<int> homePixels = {};
    for (int i = 0; i < mapWidth * mapHeight; i++) {
        if (mapData[i * 4 + 0] == 100 &&
            mapData[i * 4 + 1] == 100 &&
            mapData[i * 4 + 2] == 100 &&
            mapData[i * 4 + 3] == 255) {
            homePixels.push_back(i);
        }
        if (mapData[i * 4 + 0] == 255 &&
            mapData[i * 4 + 1] == 0 &&
            mapData[i * 4 + 2] == 0 &&
            mapData[i * 4 + 3] == 255) {
            foodOnMap++;
        }
    }

    /* Initialize agents */
    for (unsigned int i = 0; i < AGENT_COUNT; i++)
    {
        int startPositionPixelIndex = std::rand() % homePixels.size();
        float x = homePixels[startPositionPixelIndex] % mapWidth;
        float y = homePixels[startPositionPixelIndex] / mapWidth;

        float distance = std::sqrt(static_cast<float>(std::rand()) / static_cast<float>(RAND_MAX)) * static_cast<float>(std::min(mapWidth, mapHeight) / 2);
        float angle = static_cast<float>(std::rand() % static_cast<int>(2.0f * PI * 1000.0f)) / 1000.0f;
        agents.push_back({ { x, y }, angle + 1.0f * PI / 3.0f, 0, 0, 0.0f, -1000.0f, 0 });
    }

    agents[0].special = 1;

    /* Texture */
    unsigned int tex_TrailMap;
    GLCall(glActiveTexture(GL_TEXTURE0));
    GLCall(glGenTextures(1, &tex_TrailMap));
    GLCall(glBindTexture(GL_TEXTURE_2D, tex_TrailMap));
```

```
    GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE));
    GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE));
    GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST));
    GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST));
    GLCall(glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, mapWidth, mapHeight, 0, GL_RGBA, GL_FLOAT,
    nullptr));

    GLCall(glBindImageTexture(0, tex_TrailMap, 0, GL_FALSE, 0, GL_READ_WRITE, GL_RGBA32F));

    unsigned int tex_Agents;
    GLCall(glActiveTexture(GL_TEXTURE1));
    GLCall(glGenTextures(1, &tex_Agents));
    GLCall(glBindTexture(GL_TEXTURE_2D, tex_Agents));
    GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE));
    GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE));
    GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST));
    GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST));
    GLCall(glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, mapWidth, mapHeight, 0, GL_RGBA, GL_FLOAT,
    nullptr));

    GLCall(glBindImageTexture(1, tex_Agents, 0, GL_FALSE, 0, GL_READ_WRITE, GL_RGBA32F));

    unsigned int tex_Map;
    GLCall(glActiveTexture(GL_TEXTURE2));
    GLCall(glGenTextures(1, &tex_Map));
    GLCall(glBindTexture(GL_TEXTURE_2D, tex_Map));
    GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE));
    GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE));
    GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST));
    GLCall(glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST));
    GLCall(glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, mapWidth, mapHeight, 0, GL_RGBA,
    GL_UNSIGNED_BYTE, mapData));

    GLCall(glBindImageTexture(2, tex_Map, 0, GL_FALSE, 0, GL_READ_WRITE, GL_RGBA8UI));

    /* We use this memory later */
    //stbi_image_free(mapData);

    /* SSBO */
    unsigned int ssbo;
    GLCall(glGenBuffers(1, &ssbo));
    GLCall(glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo));
    GLCall(glBufferData(GL_SHADER_STORAGE_BUFFER, agents.size() * sizeof(Agent), static_cast<void*>
    (agents.data()), GL_DYNAMIC_DRAW));
    GLCall(glBindBufferBase(GL_SHADER_STORAGE_BUFFER, 3, ssbo));
    GLCall(glBindBuffer(GL_SHADER_STORAGE_BUFFER, 0));

    int work_grp_cnt[3];

    glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_COUNT, 0, &work_grp_cnt[0]);
    glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_COUNT, 1, &work_grp_cnt[1]);
    glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_COUNT, 2, &work_grp_cnt[2]);

    printf("max global (total) work group counts: x:%i y:%i z:%i\n", work_grp_cnt[0], work_grp_cnt
    [1], work_grp_cnt[2]);
```

```

    int work_grp_size[3];

    glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_SIZE, 0, &work_grp_size[0]);
    glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_SIZE, 1, &work_grp_size[1]);
    glGetIntegeri_v(GL_MAX_COMPUTE_WORK_GROUP_SIZE, 2, &work_grp_size[2]);

    printf("max local (in one shader) work group counts: x:%i y:%i z:%i\n", work_grp_size[0],
    work_grp_size[1], work_grp_size[2]);

    int work_grp_inv;

    GLCall(glGetIntegerv(GL_MAX_COMPUTE_WORK_GROUP_INVOCATIONS, &work_grp_inv));

    printf("max local work group invocations %i\n", work_grp_inv);

    float vertices[] = {
        /* positions */   /* texture coords */
         1.0f,  1.0f,      1.0f, 1.0f, /* top right */
         1.0f, -1.0f,      1.0f, 0.0f, /* bottom right */
        -1.0f, -1.0f,      0.0f, 0.0f, /* bottom left */
        -1.0f,  1.0f,      0.0f, 1.0f  /* top left */
    };

    unsigned int indices[] = {
        0, 1, 2,
        2, 3, 0
    };

    unsigned int vao;
    GLCall(glGenVertexArrays(1, &vao));
    GLCall(glBindVertexArray(vao));

    unsigned int vbo;
    GLCall(glGenBuffers(1, &vbo));
    GLCall(glBindBuffer(GL_ARRAY_BUFFER, vbo));
    GLCall(glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW));

    GLCall(glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, 4 * sizeof(float), nullptr));
    GLCall(glEnableVertexAttribArray(0));

    GLCall(glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, 4 * sizeof(float), (void*)(2 * sizeof(
    float))));
    GLCall(glEnableVertexAttribArray(1));

    unsigned int ebo;
    GLCall(glGenBuffers(1, &ebo));
    GLCall(glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, ebo));
    GLCall(glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(indices), indices, GL_STATIC_DRAW));


    std::string computeSource = ReadFile("res/shaders/Compute.shader");
    findReplaceAll(computeSource, "FOLLOW_GREEN_FEROMONE", FOLLOW_GREEN_FEROMONE);
    findReplaceAll(computeSource, "FOLLOW_RED_FEROMONE", FOLLOW_RED_FEROMONE);
    findReplaceAll(computeSource, "AVOID_WALLS", AVOID_WALLS);
```

```
    unsigned int computeShader = CompileComputeShader(computeSource);

    GLCall(unsigned int computeProgram = glCreateProgram());
    GLCall(glAttachShader(computeProgram, computeShader));
    GLCall(glLinkProgram(computeProgram));
    GLCall(glValidateProgram(computeProgram));
    GLCall(glDeleteShader(computeShader));

    GLCall(glUseProgram(computeProgram));

    GLCall(int timeLocation = glGetUniformLocation(computeProgram, "u_Time"));
    //ASSERT(timeLocation != -1);

    GLCall(int computeTextureSizeLocation = glGetUniformLocation(computeProgram, "u_TextureSize"));
    ASSERT(computeTextureSizeLocation != -1);
    GLCall(glUniform2f(computeTextureSizeLocation, static_cast<float>(mapWidth), static_cast<float>(mapHeight)));

    GLCall(int arrayOffsetLocation = glGetUniformLocation(computeProgram, "u_ArrayOffset"));
    ASSERT(arrayOffsetLocation != -1);
    GLCall(glUniform1i(arrayOffsetLocation, 0));

    std::string fadeSource = ReadFile("res/shaders/Fade.shader");
    unsigned int fadeShader = CompileComputeShader(fadeSource);

    GLCall(unsigned int fadeProgram = glCreateProgram());
    GLCall(glAttachShader(fadeProgram, fadeShader));
    GLCall(glLinkProgram(fadeProgram));
    GLCall(glValidateProgram(fadeProgram));
    GLCall(glDeleteShader(fadeShader));
    GLCall(glUseProgram(fadeProgram));

    std::string clearSource = ReadFile("res/shaders/Clear.shader");
    unsigned int clearShader = CompileComputeShader(clearSource);

    GLCall(unsigned int clearProgram = glCreateProgram());
    GLCall(glAttachShader(clearProgram, clearShader));
    GLCall(glLinkProgram(clearProgram));
    GLCall(glValidateProgram(clearProgram));
    GLCall(glDeleteShader(clearShader));
    GLCall(glUseProgram(clearProgram));

    ShaderProgramSource quadSource = ParseShader("res/shaders/Basic.shader");
    unsigned int quadProgram = CreateShader(quadSource.VertexSource, quadSource.FragmentSource);
    GLCall(glUseProgram(quadProgram));

    GLCall(int trailTextureLocation = glGetUniformLocation(quadProgram, "u_TrailTexture"));
    ASSERT(trailTextureLocation != -1);
    GLCall(glUniform1i(trailTextureLocation, 0));

    GLCall(int agentTextureLocation = glGetUniformLocation(quadProgram, "u_AgentTexture"));
    ASSERT(agentTextureLocation != -1);
    GLCall(glUniform1i(agentTextureLocation, 1));

    GLCall(int mapTextureLocation = glGetUniformLocation(quadProgram, "u_MapTexture"));
```

```
    ASSERT(mapTextureLocation != -1);
    GLCall(glUniform1i(mapTextureLocation, 2));

    GLCall(int screenSizeLocation = glGetUniformLocation(quadProgram, "u_ScreenSize"));
    ASSERT(screenSizeLocation != -1);
    GLCall(glUniform2f(screenSizeLocation, static_cast<float>(screenWidth), static_cast<float>(screenHeight)));

    GLCall(int textureSizeLocation = glGetUniformLocation(quadProgram, "u_TextureSize"));
    ASSERT(textureSizeLocation != -1);
    GLCall(glUniform2f(textureSizeLocation, static_cast<float>(mapWidth), static_cast<float>(mapHeight)));

    int roundsCounter = 0;
    int roundsPerFrame = (SAVE_DATA ? 512 : 0);
    int gatheredFood = 0;

    bool spacePressedLastFrame = false;

    float time = 0.0;

    bool runningWindow = true;

    while (runningWindow)
    {
        if (glfwWindowShouldClose(window))
        {
            runningWindow = false;
            break;
        }
        currentTime = time;
        deltaTime = currentTime - lastTime;
        lastTime = currentTime;

        glfwGetWindowSize(window, &screenWidth, &screenHeight);
        GLCall(glViewport(0, 0, screenWidth, screenHeight));

        GLCall(glClear(GL_COLOR_BUFFER_BIT));

        for (int i = 0; i < roundsPerFrame; i++) {
            time += 0.01f;
            currentTime = time;

            {
                GLCall(glUseProgram(fadeProgram));
                GLCall(glDispatchCompute(mapWidth / 16, mapHeight / 16, 1));
            }

            GLCall(glMemoryBarrier(GL_SHADER_IMAGE_ACCESS_BARRIER_BIT));

            {
                GLCall(glUseProgram(clearProgram));
                GLCall(glDispatchCompute(mapWidth / 16, mapHeight / 16, 1));
            }

```

```
        GLCall(glMemoryBarrier(GL_SHADER_IMAGE_ACCESS_BARRIER_BIT));

        {
            for (int round = 0; round < (AGENT_COUNT / 1 / work_grp_cnt[0]) + 1; round++)
            {
                GLCall(glUseProgram(computeProgram));
                GLCall(glUniform1f(timeLocation, currentTime));
                GLCall(glUniform1i(arrayOffsetLocation, round * work_grp_cnt[0]));
                GLCall(glDispatchCompute(std::min(AGENT_COUNT / 1 - work_grp_cnt[0] * round, work_grp_cnt[0]), 1, 1));
            }
        }

        GLCall(glMemoryBarrier(GL_SHADER_IMAGE_ACCESS_BARRIER_BIT));

        roundsCounter++;

        if (SAVE_DATA && roundsCounter % SAVE_DATA_EVERY_N_ROUNDS == 0)
        {
            int gatheredFoodTheseRounds = 0;
            int numberOfAntsCarryingFood = 0;
            GLCall(glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo));
            GLCall(glGetBufferSubData(GL_SHADER_STORAGE_BUFFER, 0, agents.size() * sizeof(Agent), static_cast<void*>(agents.data())));

            for (int i = 0; i < AGENT_COUNT; i++) {
                while (agents[i].foodLeftAtHome > 0) {
                    agents[i].foodLeftAtHome--;
                    gatheredFood++;
                    gatheredFoodTheseRounds++;
                }

                if (agents[i].hasFood == 1) {
                    numberOfAntsCarryingFood++;
                }
            }

            GLCall(glBufferSubData(GL_SHADER_STORAGE_BUFFER, 0, agents.size() * sizeof(Agent), static_cast<void*>(agents.data())));

            std::ofstream logFile;

            logFile.open(EXPORTED_CSVS_FOLDER + logFileName, std::fstream::app);
            logFile << time << " " << gatheredFood << " " << gatheredFoodTheseRounds << " " << numberOfAntsCarryingFood << std::endl;
            logFile.close();
        }

        if ((roundsCounter >= END_SIMULATION_AFTER_N_ROUNDS && END_SIMULATION_AFTER_N_ROUNDS != -1 && SAVE_DATA) || (gatheredFood / foodOnMap >= END_SIMULATION_AFTER_FOODP_COLLECTED && SAVE_DATA))
        {
            runningWindow = false;
        }
    }
```

```

        glfwPollEvents();

        if (GLFW_PRESS == glfwGetKey(window, GLFW_KEY_ESCAPE))
        {
            glfwSetWindowShouldClose(window, 1);
        }

        if (GLFW_PRESS == glfwGetKey(window, GLFW_KEY_0))
        {
            roundsPerFrame = 0;
        }
        else if (GLFW_PRESS == glfwGetKey(window, GLFW_KEY_1))
        {
            roundsPerFrame = 1;
        }
        else if (GLFW_PRESS == glfwGetKey(window, GLFW_KEY_2))
        {
            roundsPerFrame = 2;
        }
        else if (GLFW_PRESS == glfwGetKey(window, GLFW_KEY_3))
        {
            roundsPerFrame = 4;
        }
        else if (GLFW_PRESS == glfwGetKey(window, GLFW_KEY_4))
        {
            roundsPerFrame = 6;
        }
        else if (GLFW_PRESS == glfwGetKey(window, GLFW_KEY_5))
        {
            roundsPerFrame = 8;
        }
        else if (GLFW_PRESS == glfwGetKey(window, GLFW_KEY_6))
        {
            roundsPerFrame = 10;
        }
        else if (GLFW_PRESS == glfwGetKey(window, GLFW_KEY_7))
        {
            roundsPerFrame = 12;
        }
        else if (GLFW_PRESS == glfwGetKey(window, GLFW_KEY_8))
        {
            roundsPerFrame = 16;
        }
        else if (GLFW_PRESS == glfwGetKey(window, GLFW_KEY_9))
        {
            roundsPerFrame = 512;
        }

        if (GLFW_PRESS == glfwGetKey(window, GLFW_KEY_SPACE) && !spacePressedLastFrame)
        {
            spacePressedLastFrame = true;

            GLCall(glBindBuffer(GL_SHADER_STORAGE_BUFFER, ssbo));
```

```
            GLCall(glGetBufferSubData(GL_SHADER_STORAGE_BUFFER, 0, sizeof(Agent) *agents.size(),
                  static_cast<void*>(agents.data())));

            for (int i = 0; i < AGENT_COUNT; i++)
            {
                agents[i].special = 0;
            }

            agents[std::rand() % AGENT_COUNT].special = 1;

            GLCall(glBufferSubData(GL_SHADER_STORAGE_BUFFER, 0, sizeof(Agent) *agents.size(),
            static_cast<void*>(agents.data())));

        }
        else
        {
            spacePressedLastFrame = false;
        }

        /* Render to the screen */
        {
            GLCall(glUseProgram(quadProgram));
            GLCall(glUniform2f(screenSizeLocation, static_cast<float>(screenWidth),
                   static_cast<float>(screenHeight)));
            GLCall(glBindVertexArray(vao));
            glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_INT, 0);
            GLCall(glDrawArrays(GL_TRIANGLES, 0, 3));
        }


        /* Swap front and back buffers */
        glfwSwapBuffers(window);
    }

    if (SAVE_DATA) {
        std::ofstream logFile;
        logFile.open(EXPORTED_CSVS_FOLDER + logFileName, std::fstream::app);
        logFile << std::endl << std::endl;
        logFile.close();
    }

    glfwTerminate();
    return 0;
}
```

5.3 C. Shaders

The parts of the code that are solely used for debugging and running are marked with a faded background. These parts can be removed without affecting the data collection.

Fade Shader This shader reduces the value of each pixel by a constant amount, and clamps it between 0.0 and 1.0. If one wishes to blur the texture, the commented lines can be used to achieve a weighted average of all the neighbours.

```
#version 450 core

layout(local_size_x = 16, local_size_y = 16) in;

layout(rgba32f, binding = 0) uniform image2D img_TrailMap;

void main() {
    ivec2 pixel_coords = ivec2(gl_GlobalInvocationID.xy);

    vec4 pixel = 1.0 *   imageLoad(img_TrailMap, pixel_coords);
    /*
    pixel     += 0.0 * imageLoad(img_TrailMap, pixel_coords + ivec2(1, 0));
    pixel     += 0.0 * imageLoad(img_TrailMap, pixel_coords + ivec2(-1, 0));
    pixel     += 0.0 * imageLoad(img_TrailMap, pixel_coords + ivec2(0,  1));
    pixel     += 0.0 * imageLoad(img_TrailMap, pixel_coords + ivec2(0, -1));
    pixel     += 0.0 * imageLoad(img_TrailMap, pixel_coords + ivec2(1,  1));
    pixel     += 0.0 * imageLoad(img_TrailMap, pixel_coords + ivec2(-1,-1));
    pixel     += 0.0 * imageLoad(img_TrailMap, pixel_coords + ivec2(-1, 1));
    pixel     += 0.0 * imageLoad(img_TrailMap, pixel_coords + ivec2(1, -1));
    */

    pixel = vec4(clamp(pixel.rgb - vec3(1.0 / (100.0 * 15.0)), 0.0, 1.0), 1.0);
    imageStore(img_TrailMap, pixel_coords, pixel);
}
```

Clear Shader This shader clears a texture.

```
#version 450 core

layout(local_size_x = 16, local_size_y = 16) in;

layout(rgba32f, binding = 1) uniform image2D img_Agents;

void main() {
    ivec2 pixel_coords = ivec2(gl_GlobalInvocationID.xy);

    vec4 pixel = vec4(0.0);

    imageStore(img_Agents, pixel_coords, pixel);
}
```

Render Shader This shader combines multiple textures and renders them to the screen.

```
#shader vertex
#version 450 core

layout(location = 0) in vec4 position;
layout(location = 1) in vec2 textureCoords;

```

```
out vec2 TexCoord;

uniform vec2 u_TextureSize;
uniform vec2 u_ScreenSize;

void main()
{
    vec4 newPosition = position;

    float textureAspect = u_TextureSize.x / u_TextureSize.y;
    float screenAspect = u_ScreenSize.x / u_ScreenSize.y;

    if (textureAspect > screenAspect)
    {
        newPosition.y *= screenAspect / textureAspect;
    }
    else
    {
        newPosition.x *= textureAspect / screenAspect;
    }

    gl_Position = newPosition;
    TexCoord = textureCoords;
};

#shader fragment
#version 450 core

layout(location = 0) out vec4 color;

in vec2 TexCoord;

uniform sampler2D u_TrailTexture;
uniform sampler2D u_AgentTexture;
uniform sampler2D u_MapTexture;

uniform vec2 u_TextureSize;

void main()
{
    vec4 agentColor = texture(u_AgentTexture, TexCoord);
    vec4 trailColor = texture(u_TrailTexture, TexCoord);
    vec4 mapColor = texture(u_MapTexture, TexCoord);

    vec3 added = mapColor.rgb * mapColor.a + trailColor.rgb * trailColor.a + agentColor.rgb *
    agentColor.a;
    added.r = clamp(added.r, 0.0, 1.0);
    added.g = clamp(added.g, 0.0, 1.0);
    added.b = clamp(added.b, 0.0, 1.0);

    color = vec4(added, 1.0);
};
```

Compute Shader This shader contains all of the logic for each individual agent.

```
#version 450 core
layout(local_size_x = 1, local_size_y = 1) in;

struct Agent
{
    vec2 position;
    float angle;
    int hasFood;
    int foodLeftAtHome;
    float timeAtSource;
    float padding;
    int special;
};

layout(rgba32f, binding = 0) uniform image2D img_TrailMap;
layout(rgba32f, binding = 1) uniform image2D img_Agents;
layout(rgba8ui, binding = 2) uniform uimage2D img_Map;

layout(std430, binding = 3) buffer agentsLayout
{
    Agent agents[];
};

uniform float u_Time;
uniform vec2 u_TextureSize;
uniform int u_ArrayOffset;

// These will be replaced by the main script to either true or false
// FOLLOW_RED_FEROMONE;
// FOLLOW_GREEN_FEROMONE;
// AVOID_WALLS;

const float PI = 3.141592653589793238;
const float PHI = 1.61803398874989484820459;
const float TIME_LAYING_TRAIL = 50.0;
const float TIME_ERASING_FEROMONES_AFTER_WALL_COLLISION = 0.10;

const float senceDistance = 10.0;

float gold_noise(vec2 xy, float seed, float seed_counter);
vec4 sence(ivec2 pos, int special);

void main() {
    float seed_counter = 0.0;

    uint ind = gl_GlobalInvocationID.x + u_ArrayOffset;
    Agent agent = agents[ind];

    ivec2 pixel_coords = ivec2(agent.position);

    // Move by current angle and speed
    agent.position += vec2(cos(agent.angle), sin(agent.angle)) * 90.0 / 60.0;
    ivec2 new_pixel_coords = ivec2(agent.position);

```

```

    // Sence feromone concentration
    vec4 left =   sence(ivec2(agent.position + vec2(cos(agent.angle + PI / 3.0), sin(agent.angle + PI / 3.0)) * senceDistance), agent.special);
    vec4 middle = sence(ivec2(agent.position + vec2(cos(agent.angle           ), sin(agent.angle           )) * senceDistance), agent.special);
    vec4 right =  sence(ivec2(agent.position + vec2(cos(agent.angle - PI / 3.0), sin(agent.angle - PI / 3.0)) * senceDistance), agent.special);

    float f_left =   (agent.hasFood == 1 ? left.g   : left.r);
    float f_right =  (agent.hasFood == 1 ? right.g  : right.r);
    float f_middle = (agent.hasFood == 1 ? middle.g : middle.r);

    if (agent.hasFood == 1 && !FOLLOW_GREEN_FEROMONE)
    {
        f_left = 0.0;
        f_right = 0.0;
        f_middle = 0.0;
    }
    else if (agent.hasFood == 0 && !FOLLOW_RED_FEROMONE)
    {
        f_left = 0.0;
        f_right = 0.0;
        f_middle = 0.0;
    }

    float f_turning = 0.0;

    if (f_middle > f_left && f_middle > f_right)
    {
        f_turning = 0.0;
    }

    if (f_left > f_middle && f_left > f_right)
    {
        seed_counter++;
        f_turning = gold_noise(agent.position, u_Time, seed_counter) * PI * 0.5;
    }

    if (f_right > f_middle && f_right > f_left)
    {
        seed_counter++;
        f_turning = -gold_noise(agent.position, u_Time, seed_counter) * PI * 0.5;
    }

    // Turn by anglechange
    seed_counter++;
    agent.angle += f_turning * 1.0 + (gold_noise(agent.position, u_Time, seed_counter) - 0.5) * 0.4;
    //agent.angle += gold_noise(agent.position, u_Time) - 0.5;


    // Bounce on map borders - horizontal
    if (agent.position.x < 0.0)
    {
```

```
         agent.position.x = 0.0;

         seed_counter++;
         agent.angle = (gold_noise(agent.position, u_Time, seed_counter) - 0.5) * PI;
    }
    else if (agent.position.x >= u_TextureSize.x)
    {
        agent.position.x = u_TextureSize.x - 1.0;

        seed_counter++;
        agent.angle = (gold_noise(agent.position, u_Time, seed_counter) + 0.5) * PI;
    }

    // Bounce on map borders - vertical
    if (agent.position.y < 0.0)
    {
        agent.position.y = 0.0;

        seed_counter++;
        agent.angle = (gold_noise(agent.position, u_Time, seed_counter)) * PI;
    }
    else if (agent.position.y >= u_TextureSize.y)
    {
        agent.position.y = u_TextureSize.y - 1.0;

        seed_counter++;
        agent.angle = (gold_noise(agent.position, u_Time, seed_counter) + 1.0) * PI;
    }

    // Interactions with map
    float xDiff = new_pixel_coords.x - pixel_coords.x;
    float yDiff = new_pixel_coords.y - pixel_coords.y;
    float maxDiff = max(abs(xDiff), abs(yDiff));

    for (float step = 1; step <= maxDiff; step++) {
        ivec2 intermediate_pixel_coords = ivec2(pixel_coords.x + xDiff * step / maxDiff,
    pixel_coords.y + yDiff * step / maxDiff);

        uvec4 mapColor = imageLoad(img_Map, intermediate_pixel_coords);
        if (mapColor == uvec4(128, 128, 128, 255))  // Wall collision
        {
            agent.position = vec2(pixel_coords);

            seed_counter++;
            agent.angle += PI + ((gold_noise(agent.position, u_Time, seed_counter) - 0.5) * PI);
            break;
        }
        else if (mapColor == uvec4(255, 0, 0, 255) && agent.hasFood == 0) // Food collision
        {
            imageStore(img_Map, intermediate_pixel_coords, uvec4(0));
            agent.hasFood = 1;
            agent.timeAtSource = u_Time;
            agent.angle += PI;
            break;
        }
```

```
        else if (mapColor == uvec4(100, 100, 100, 255)) // Home collision
        {
            agent.timeAtSource = u_Time;
            if (agent.hasFood == 1) {
                agent.foodLeftAtHome++;
                agent.hasFood = 0;
                agent.angle += PI;
            }
            break;
        }
    }

    // Show agent on map

    vec4 pixel = vec4(0.0, 1.0, 0.0, 1.0);
    if (agent.hasFood == 1)
    {
        pixel = vec4(1.0, 0.0, 0.0, 1.0);
    }

    ivec2 final_pixel_coords = ivec2(agent.position);

    // Show on agent map

    imageStore(img_Agents, final_pixel_coords, pixel);
    if (agent.special == 1)
    {
        for (int x = -2; x <= 2; x++) {
            for (int y = -2; y <= 2; y++) {
                imageStore(img_Agents, final_pixel_coords + ivec2(x, y), vec4(1.0));
            }
        }
    }

    // Add feromone-trail
    vec4 oldValue = imageLoad(img_TrailMap, final_pixel_coords);
    pixel *= max(1 - (u_Time - agent.timeAtSource) / TIME_LAYING_TRAIL, 0);
    vec4 newValue = vec4(clamp(oldValue.rgb + pixel.rgb / 2.0, 0.0, 1.0), 1.0);
    imageStore(img_TrailMap, final_pixel_coords, newValue);


    agents[ind] = agent;
}

float gold_noise(vec2 xy, float seed, float seed_counter)
{
    seed = fract(seed) + seed_counter;
    xy += vec2(1.0);

    return fract(sin(distance(xy * PHI, xy) * seed) * xy.x);
}

vec4 sence(ivec2 pos, int special)
{
    vec4 averageColor = vec4(0.0);
```

```
    for (int x = -5; x <= 5; x++)
    {
        for (int y = -5; y <= 5; y++)
        {
            vec4 trail = imageLoad(img_TrailMap, pos + ivec2(x, y));

            uvec4 map = imageLoad(img_Map, pos + ivec2(x, y));
            if (map == uvec4(100, 100, 100, 255))
            {
                trail.g = 10000.0;
            }
            else if (map == uvec4(255, 0, 0, 255))
            {
                trail.r = 10000.0;
            }
            else if (map == uvec4(128, 128, 128, 255) && AVOID_WALLS)
            {
                trail.rg = vec2(-100.0, -100.0);
            }
            averageColor += trail;

            if (special == 1)
            {
                imageStore(img_Agents, pos + ivec2(x, y), vec4(1.0));
            }
        }
    }
    averageColor /= 25.0;

    return averageColor;
}
```

5.4 D. Data Collection Code

This batch script goes through the relative paths of the maps to the run location of the batch file in the project. It then goes over the number of agents that should be simulated and then does every simulation 40 times.

```
@echo off

FOR %%m IN (.\res\textures\testmap_no_walls_1024x512.png
            .\res\textures\testmap_simplemaze_1024x512.png
            .\res\textures\testmap_split_path_1024x512.png
            .\res\textures\testmap_multifood_1024x512.png) DO (
  FOR %%a IN (2 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 32768) DO (
    ECHO Calculating map %%m
    FOR /L %%n IN (1, 1, 40) DO (
      ECHO     Running %%n
      "..\\bin\\Win32\\Release\\Ant Simulation.exe" %%m %%a %%n
    )
  )
)
PAUSE
```

5.5 E. Data Analysis Code in Python

```python
import collections
import csv
import os
import statistics
import math
path = r"path/to/folder/with/data/files]"
#PARAMETRAR:
simulationsPerAgentNumber = 40
relativeCollexiency = False

mapDictionary = collections.defaultdict(dict)

for simulationName in os.listdir(path):
    agentNumber = int(simulationName.split("_")[2][2:])
    with open(path + "\\" + simulationName, 'r') as csv_file:
        csv_reader = csv.reader(csv_file, delimiter = ' ')
        lineCounter = 0
        for line in csv_reader:
            if lineCounter < 10:
                if lineCounter == 3:
                    if "no_walls" in line[1]:
                        currentMap = "no_walls"
                        totalFood = 1877
                    elif "split_path" in line[1]:
                        currentMap = "split_path"
                        totalFood = 1877
                    elif "multifood" in line[1]:
                        currentMap = "multifood"
                        totalFood = 6887
                    else:
                        currentMap = "simplemaze"
                        totalFood = 3241
                elif lineCounter == 4:
                    if "false" in line[1]:
                        currentMap = currentMap.upper()  #UPPERCASE = simulation without pheromones
                lineCounter += 1
                continue
            if int(line[1]) >= totalFood * .99 or line[0] == "1602.05":
                collexiencyValue = int(line[1]) / float(line[0])
                if relativeCollexiency:
                    collexiencyValue /= agentNumber

                if mapDictionary[currentMap].get(agentNumber) is None:
                    mapDictionary[currentMap][agentNumber] = [collexiencyValue]
                else:
                     mapDictionary[currentMap][agentNumber].append(collexiencyValue)

                     if len(mapDictionary[currentMap][agentNumber]) == simulationsPerAgentNumber:
                         collexiencyValueList = mapDictionary[currentMap][agentNumber]
                         averageCollexiency = sum( collexiencyValueList ) / simulationsPerAgentNumber
                         collexiencyStandardDeviation = statistics.pstdev( collexiencyValueList )
```

```
                         mapDictionary[currentMap][agentNumber] =
                                 (averageCollexiency, collexiencyStandardDeviation)
                break
print( "\n".join([f"{math.log2(k)} " + " ".join(map(lambda x: "{:.20f}".format(x), v[:]))
                  for k, v in mapDictionary["simplemaze".upper()].items() ]))  # print collexiency
```