Interactive and Variable Visualisations of Aggregate Motion

Depicting Movements of Simulated Flocks

Szun Kidd Choi

B. Thomas Golisano College of Computing and Information Sciences, Rochester Institute of Technology

sc4020@rit.edu

According to Aristotle, the whole is greater than the sum of its parts. Craig Reynolds's 1986 boids simulation was developed based on the assumption made that the overall motion of a flock of organisms in nature could be expressed as a cumulation of decisions and behaviours of individual flock members themselves. This paper presents a top-down, graphical approach to visualising the complex emergent interactions that Reynolds's simple bottom-up solution generates. Several modifications and additions are made to the original framework, with the intent of emphasising, and heightening appreciation of, resultant observable group behaviours.

Additional Keywords and Phrases: flocking algorithm, swarm intelligence, swarming, simulation, collective intelligence, user interactivity

ACM Reference Format:

Szun Kidd Choi. 2022. Interactive and Variable Visualisations of Aggregate Motion. Depicting Movements of Simulated Flocks. For IGME 690: Seminar on Intelligent Systems in the Arts, January 10 – May 4, 2022, Rochester, NY. ACM, New York, NY, USA, 24 pages.

1 INTRODUCTION

There were two sources of inspiration behind this system. The first hailed from the nature of air shows, an example of flocking, albeit man-made. The coordinated motions of squadrons of fighter jets flying together in tight proximity is a sight to behold. This is further enhanced when colourful smoke is deployed, carving out streaks of vibrancy in the sky. The second was simply the coordinated group motion that a swarm of insects tends to have.

This system aims to provide a visceral depiction of the collective interactions of members within a simulated flock. The overall “theme” of the environment can be swapped during runtime between “artistic” and “plain” colour palettes. On one hand, in artistic mode, individual agents produce a coloured history of their trajectories behind them in the form of a trail. In this way, the decisions of the overall group become rendered as lines drawn onscreen. On the other hand, plain mode removes all trails and instead displays the results of the flocking algorithm without any visual supplements.

Within this system, agents adhere to the original three flocking rules in Reynolds’s boids simulation (cohesion, alignment, and separation), with the addition of five more: lifespan, speed limitation, environment retention, obstacle avoidance, and tendency toward user interaction locations. These additional rules do not overrule any of the original three, but instead work together to produce modified interactions between agents and their environment.

Creating art by leveraging the motions of individual flock members is a process that has been well-attempted and thoroughly documented online. Nevertheless, existing implementations all lack the combination of interactivity and an extensive, user-exposed customisation system with respect to the behaviour of individual agents.

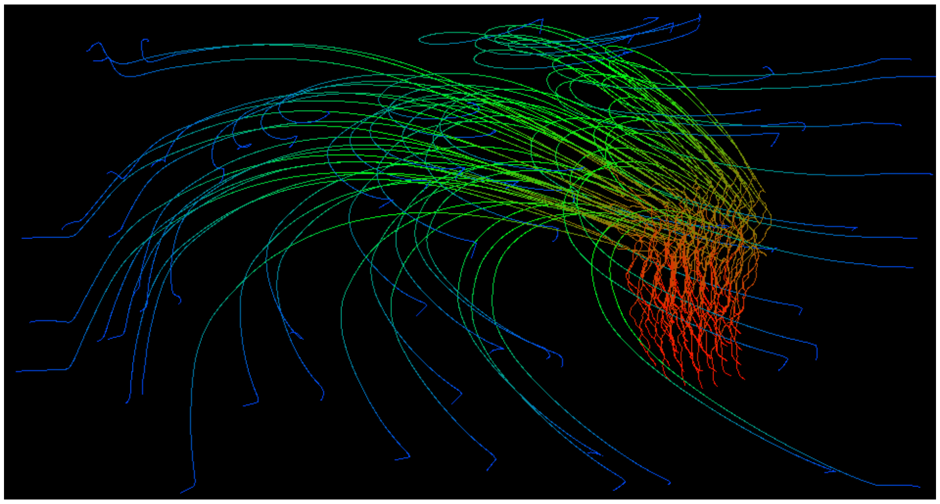


Fig. 1. An example of the output the system can generate. This was created using a combination of both emergent agent behaviour and user interaction.

2 BACKGROUND AND RELATED WORK

The flocking algorithm was created in 1986 by Craig Reynolds [1], who published a paper on the subject in 1987 [2]. He named the agents within his system “boids”. In the simulation, agents move together, and any changes in direction made by members at the forefront are propagated throughout the flock. Reynolds’s system is considered an example of swarm intelligence; each agent is its own entity, capable of making decisions; however, the behaviour of the resulting formation arises from the collective movements of the agents themselves. [3]

When in motion, an agent’s movements are governed by three “rules”: flock centring, velocity matching, and collision avoidance. More recent texts refer to these rules as cohesion, alignment, and separation, respectively. The influence of each rule contributes to a steering force that is applied to each agent.

Since its creation, further rules have been developed over time to extend Reynolds’s original boids implementation. Parker [4] has described several possible modifications. These include scattering the flock in response to external stimuli and responses to forces, such as a gust of wind or an underwater current. Some of these modifications have been used in the presented system to achieve flexibility and customisability.

As mentioned in the previous section, there are numerous examples of artwork, both digital and physical, that visualise flocking artistically. There also exist sites online that draw the results of such aggregate motion to the screen in real-time.

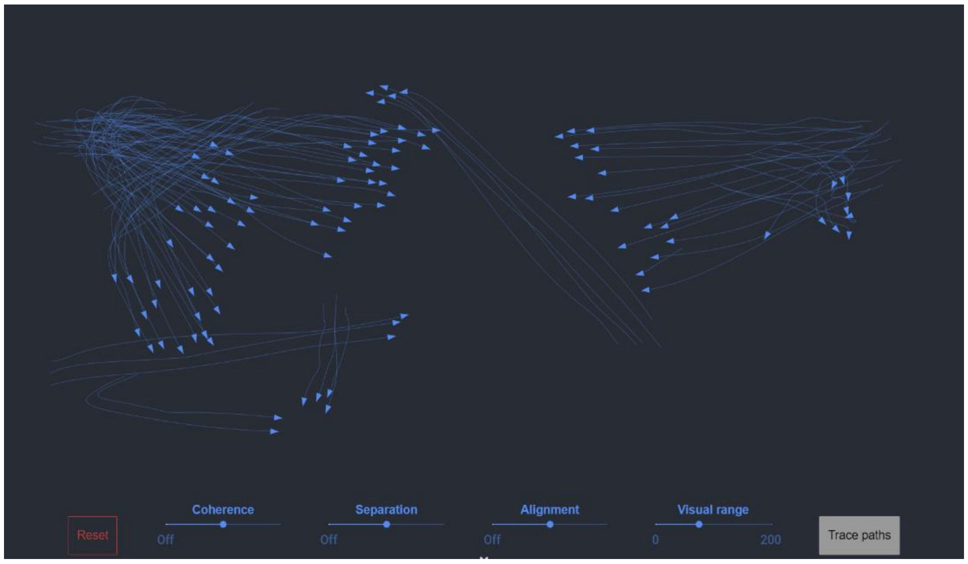
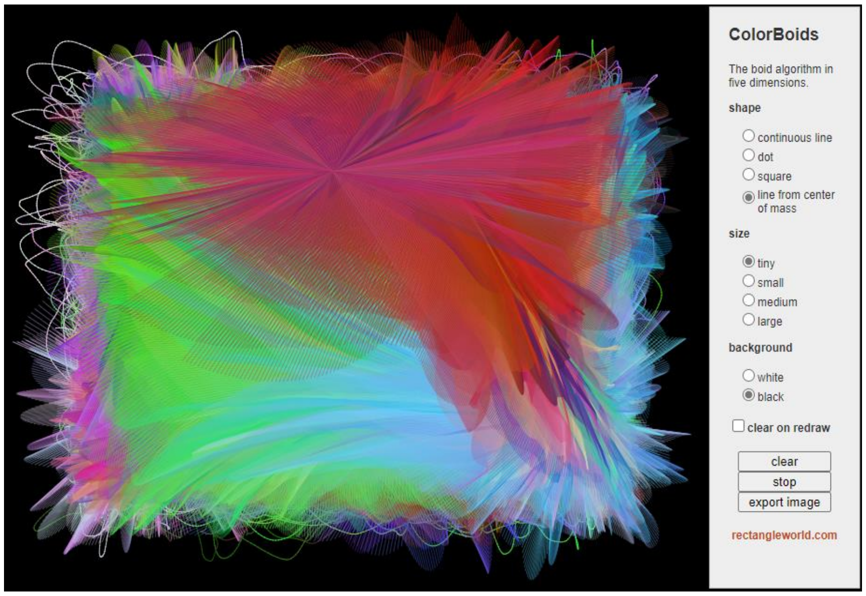


Fig. 2. Eater’s [6] implementation of boids. Four rules whose values the user can customise are exposed. Besides toggling trails, the system’s appearance cannot otherwise be modified.



ColorBoids

The boid algorithm in five dimensions.

shape: continuous line, dot, square, [✓ line from center of mass]

size: [✓ tiny], small, medium, large

background: white, [✓ black]

clear on redraw

Fig. 3.

Another implementation of boids. [14] This version lets users customise the way agents' movement is drawn to the screen. The simulation run shown here is using a mixture of continuous lines and a line drawn from the centre of mass of the entire flock. It is unclear what the influence of each rule is here.

3 SYSTEM/PROJECT DESCRIPTION

The entire system was created within Unity, with scripting done in C#. as that meant not needing to write code handling the updating and drawing of the agents to the screen. The engine provides inbuilt support for raycasting, which was useful for implementing collision avoidance. It is worth noting that Unity's physics library was not used in this implementation of the system, and that, as a result, all code for agent movement was written manually.

This section will be broken down into six subsections, each describing a core component of the overall system. It will begin with Section 3.1, an overview of program states using a finite state machine (FSM) diagram, before proceeding to detail the logic that all agents run each frame during a simulation run. A brief look at the components making up individual agents in the editor is then supplied in Section 3.2.

As an overview, however, agents begin by looking at all their flockmates, keeping track of only the ones within a certain visibility region (their "neighbours"). This is detailed in Section 3.3. Next, using the neighbours obtained in the previous step, rule calculations take place. The original three rules are detailed in Section 3.4, and additional rules are discussed in Section 3.5.

Finally, interactivity and exposed user-configurable options are covered in Section 3.6.

3.1 Program Architecture

The following is an FSM describing all possible defined states the program can be in.

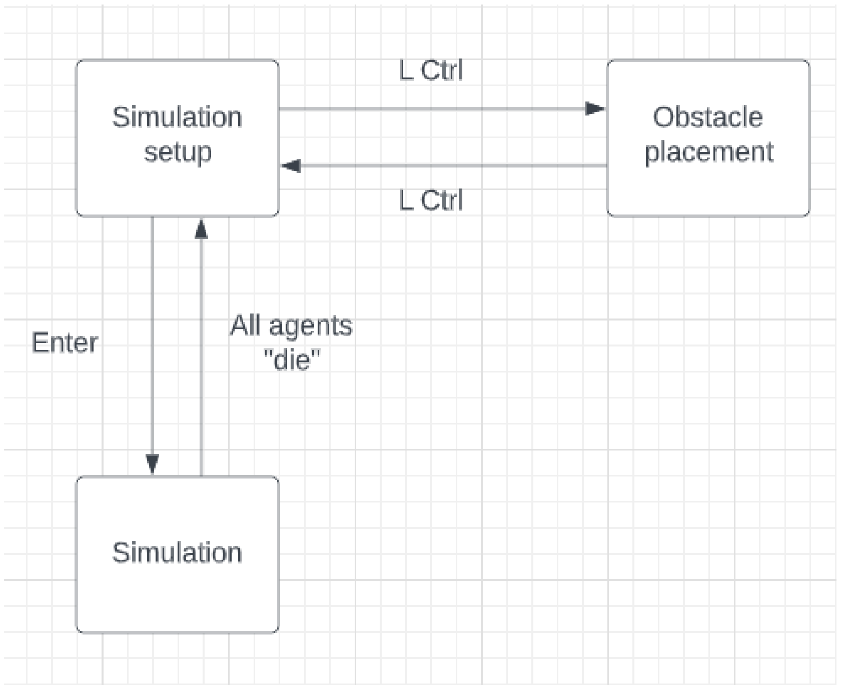


Fig. 4. FSM of program states

The simulation setup screen is where variable values (as highlighted in Section 3.6) can be altered, as well as trail colours. Pressing Left Control on that screen brings users to the obstacle placement screen, and another press of the same key brings the user back to the simulation setup screen. From that same screen, a press of the Enter key begins the simulation with the user-defined variable values, trail colours, and obstacle locations, spawning agents at random locations on-screen. Only when all agents “die” (i.e., the number of seconds specified by the value of the “lifespan” slider has elapsed) is the user returned to the simulation setup screen once again.

3.2 Agent Architecture

Because the system does not leverage Unity’s physics library, individual agents do not have rigid body components. They are, instead, comprised of a sprite, a trail renderer, a collider, and a script to control movement. The dimensions – and, therefore, bounds – of an agent, then, are defined directly by its sprite.

3.3 Neighbour-Checking

Before any of the calculations for each of the three rules can take place, an agent needs to know about its immediate neighbours. The following diagram illustrates the local visibility region of an agent.

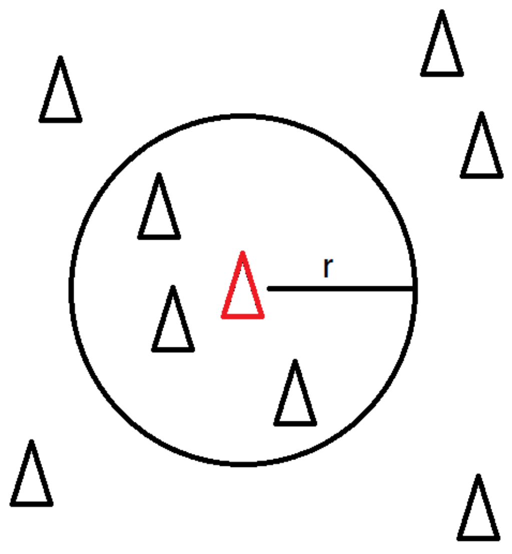


Fig. 5. The visibility region of an agent.

In the system, an agent’s visibility region is a circle whose size is described by a radius, r. Flockmates within this circle are “visible” and considered neighbours. They are used when applying the flocking rules. All the other agents are ignored. As the value of r increases, the agents can “see” more flockmates, resulting in a more cohesive flock.

The flock is constantly moving, and so the simulation must update each agent’s view of the world each frame, to acquire each agent’s unique perspective.

Simplifying the formula proposed by Seeman and Bourg [5], the Euclidean distance is used to test which of an agents flockmates is to be considered a neighbour that frame:

$$d = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2}$$

where x_a and x_b are the x-coordinates for the current agent and the current flockmate, respectively. Similarly, y_a and y_b are the y-coordinates for the current agent and the current flockmate, respectively. If

$$d \leq r$$

then the flockmate under consideration is counted as a neighbour. Additionally, d is expressed in terms of the length of the agent’s sprite. This was done to allow the system to scale dynamically should the size of the agents be modified in future iterations.

3.4 The (Original) Three Rules

3.4.1 Cohesion

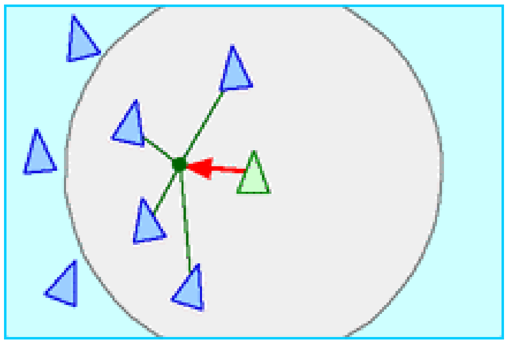


Fig. 6. An illustration of the cohesion rule. [1] Here, the current agent (green) must steer left (red arrow) toward the average position of its neighbours (green dot).

Cohesion is a rule that prevents agents from breaking off from the flock. To achieve this, each agent steers toward the average position of its neighbours in a method like that proposed by Eater [6], such that

$$s_{avg} = \frac{s_{total}}{n}$$

where s is a position vector and n is the number of neighbours the current agent has. s_{total} is a running total of the positions of all an agent's neighbours. The current agent's velocity is then updated via:

$$v_{new} = v_{curr} + (s_{avg} - s_{curr}) * F_{cohesion}$$

where s_{curr} is the agent's position vector and $F_{cohesion}$ is a constant that determines the influence of the cohesion rule.

The difference between s_{avg} and s_{curr} forms a relationship that is directly proportional: the greater the difference, the larger the corrective force required to return a straying agent to its flock. [5]

3.4.2 Alignment

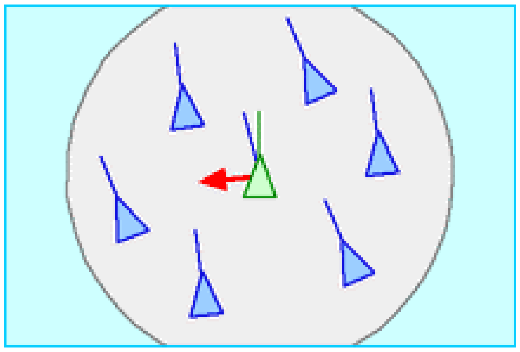


Fig. 7. An illustration of the alignment rule. [1] The current agent (green) must steer left (red arrow) to match the average velocity of its neighbours (blue line drawn from the current agent).

Alignment is a rule that forces agents to move in the same general direction. To do this, each agent's velocity is adjusted such that it matches the average velocity of its neighbours in a method like that proposed by Eater [6], such that

$$v_{avg} = \frac{v_{total}}{n}$$

where v is a position vector. v_{total} is a running total of the positions of all an agent's neighbours. The current agent's velocity is then updated via:

$$v_{new} = v_{curr} + (v_{avg} - v_{curr}) * F_{alignment}$$

where $F_{alignment}$ is a constant that determines the influence of the alignment rule.

Like cohesion, the difference between v_{avg} and v_{curr} forms a relationship that is directly proportional.

3.4.3 Separation

Separation states that agents should stay some minimum distance apart to avoid collisions, despite the cohesion and alignment rules bringing them together. In this system, this minimum distance may be different from the radius of the agent's visibility region! Based on Eater [6], a running total of the distance vectors between the agent and neighbours that are

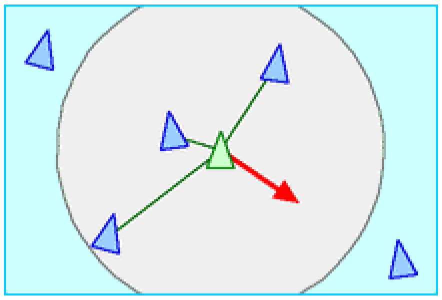


Fig. 8. The separation rule. [1] The current agent (green) must steer right (red arrow) to distance itself from its neighbours.

too close is obtained by using the Euclidean distance obtained in Section 3.2. Then, the agent's velocity is updated via:

$$v_{new} = v_{curr} + d_{total} * F_{separation}$$

where d_{total} is the vector sum of all distance vectors between the agent and neighbours that present a potential collision, and $F_{separation}$ is a constant that determines the influence of the separation rule.

Here, the corrective steering force is inversely proportional to the actual separation distance. This will make the steering correction force greater the closer an agent gets to a neighbour. [5]

3.5 Additional Rules

3.5.1 Lifespan

The lifespan of an agent determines how long it is alive for i.e., how long it is active in the Scene in seconds. It therefore determines the duration of the current simulation run altogether.

3.5.2 Tendency Toward User Interaction Location

Agents are "attracted" to the location of the cursor while the user holds down the left mouse button. This was inspired by the mobile games FROST [7] and Lifelike [8], as well as the interactive physical installation SWARM [9], and was included to allow for added user interactivity. Based on the implementation proposed by Parker [4], the following accomplishes this:

$$v_{new} = \frac{s_{cursor} - s_{curr}}{m}$$

where s_{cursor} is the position of the cursor in world coordinates, and m is the number of frames in total needed to move the agent towards the cursor. As m increases, then, so does the time it takes for the agents to reach the cursor.

3.5.3 Environment Retention

This was quickly deemed necessary, since flocks, due to their impromptu decision-making, tend to move off-screen quite quickly. Using an implementation drawn from that proposed by Eater [6] and Parker [4], a correctional force is applied in the opposite direction of an agent's movement if it approaches a bounding margin of a set distance from any of the four edges of the screen via

$$v_{x,new} = v_{x,curr} + F_{retention}$$

$$v_{x,new} = v_{x,curr} - F_{retention}$$

if the agent approaches the left and right edges of the screen, respectively, and

$$v_{y,new} = v_{y,curr} + F_{retention}$$

$$v_{y,new} = v_{y,curr} - F_{retention}$$

if the agent approaches the lower and upper edges of the screen, respectively.

The individual x- and y-components of the agent's velocity are modified directly in this implementation, and $F_{retention}$ is the amount of force by which to coerce the agent back on-screen. As $F_{retention}$ increases, the more closely agents respect the predefined margins, but also the more abrupt – and, therefore, the more unrealistic – the corrective turns they tend to make. This approach assumes the use of a standard Cartesian coordinate plane for world coordinates, with the origin O(0, 0) at the screen's centre.

3.5.4 Obstacle Avoidance

Placing user-defined constraints on the motions of flocks in the form of obstacles agents cannot fly through was included for two reasons. First, it provided a greater deal of customisability and provided an additional layer of interactivity. Second, the placement of such constraints causes emergent behaviour that effectively draws "stencilled out" shapes to the canvas.

The implementation is akin to that provided by Bevilacqua [10], which utilises circles for obstacles, and outfits each agent with two "feelers", one which is half the length of the other. These feelers are components separate to that of the radius, r, defining an agent's visibility region.

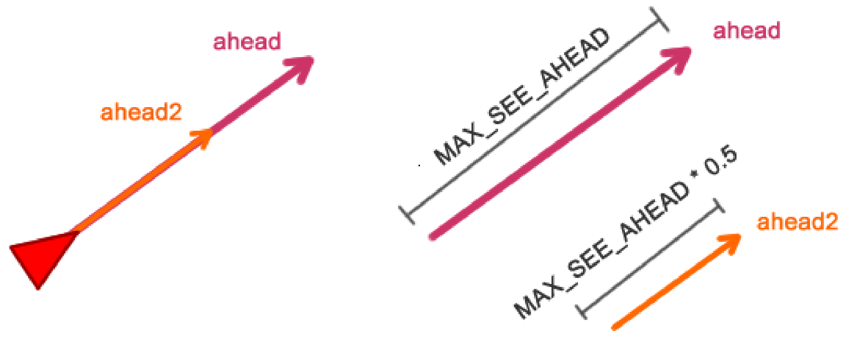


Fig. 9. The two feelers, as described by Bevilacqua [10]. MAX_SEE_AHEAD* 0.5 is half the length of MAX_SEE_AHEAD. Both point in the same direction.

Subsequently, a raycast is used to determine whether either of the two feelers are intersecting with an obstacle ahead. Bevilacqua notes in their implementation that it is necessary to handle cases where multiple objects may present potential collisions, since only the closest, "most threatening" obstacle need be considered.

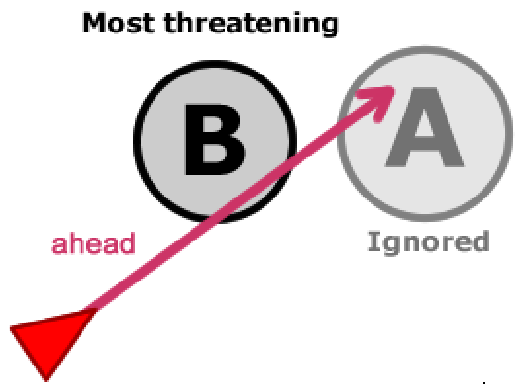


Fig. 10. Bevilacqua [10] states that only the closest or "most threatening" obstacle must be used for calculations. Unity's raycast system eliminates the need for this, since only the first object the ray intersects with is returned.

However, as Unity's raycast method provides an overload where only the first object intersected with gets returned [11][12], the system does not need to account for this.

If either feeler intersects with an obstacle, then the following is used to compute an avoidance force vector:

$$d = A - C$$

$$a = \hat{d} \times F_{avoidance}$$

where A is a vector representing the longer of the two feelers, C is the position of the centre of the obstacle, d is the distance vector between A and C, $\hat{d}$ is the value obtained after normalising d, and $F_{avoidance}$ is the amount by which to scale the avoidance force. Finally, the agent's velocity is updated:

$$v_{new} = v_{curr} + a$$

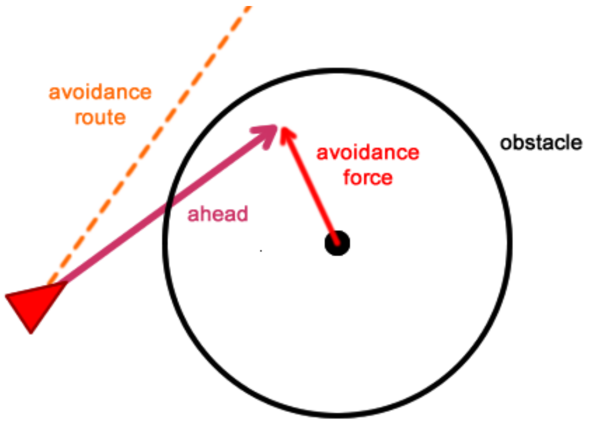


Fig. 11. An illustration of the obstacle avoidance rule being applied. [10] The calculations above in this case result in the projected avoidance route (orange dotted line).

3.5.5 Speed Limitation

It can be observed in the definitions of previous rules that, to move an agent in the desired direction, the steering force is applied by means of multiple vector sums. This can cause agents to travel at unrealistically high speeds, which is in line with Parker's own discoveries [4]. By limiting the magnitude of each agent's velocity, the system is kept under control; this rule works hand-in-hand with environment retention such that agents are prohibited from escaping the boundaries of the canvas.

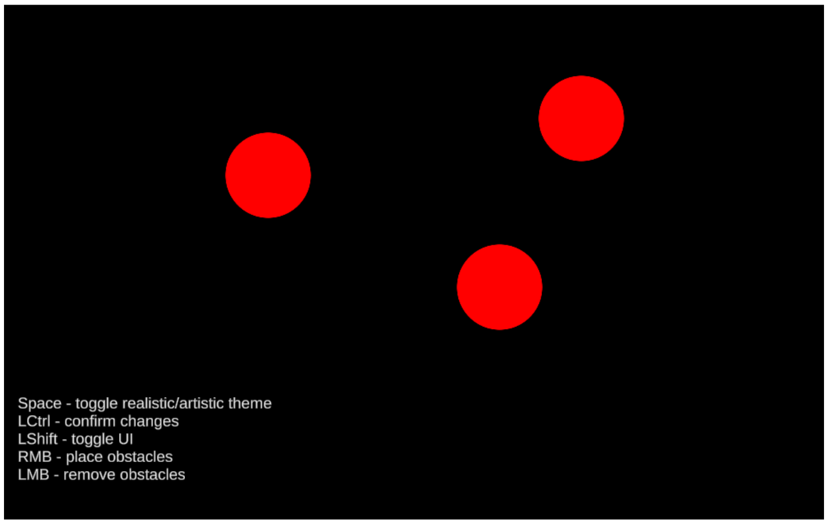
In an implementation similar to that proposed by Parker, each agent's speed is obtained by calculating the magnitude of its velocity vector. If this value is larger than a prescribed maximum, it is limited by means of

$$v_{new} = \hat{v}_{curr} * S_{max}$$

where $\hat{v}_{curr}$ is the value obtained after normalising v_{curr} and S_{max} is the maximum allowed speed. Note the use of an uppercase S to denote speed in contrast to the lowercase s used earlier to denote position.

3.6 Interactivity

Aside from user-defined obstacle placement and making agents flock to the user's cursor while the left mouse button is held down, the system also exposes several of the previously mentioned variables.

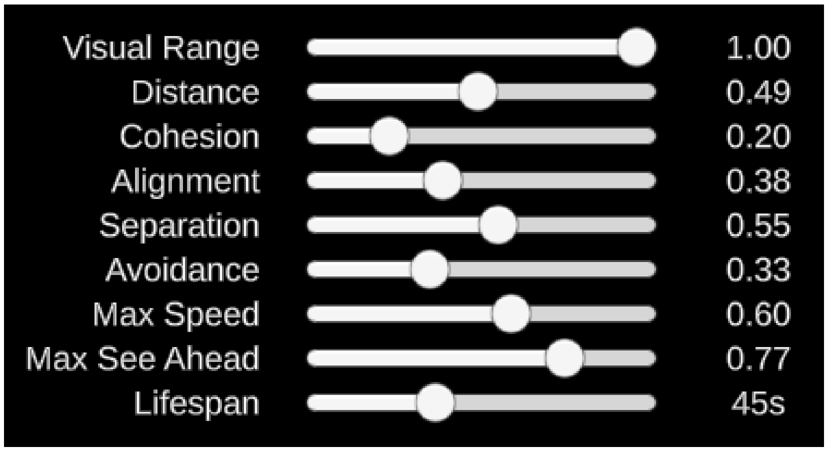


Space - toggle realistic/artistic theme
LCtrl - confirm changes
LShift - toggle UI
RMB - place obstacles
LMB - remove obstacles

Fig. 12. The obstacle placement screen. Clicking the right mouse button places an obstacle, and clicking the left while hovering over an existing obstacle removes it. Up to a maximum of 10 obstacles can be placed.

The variables exposed were chosen based on whether they had the potential to "break" the simulation. For instance, $F_{retention}$ was not included, simply because altering that value might result in agents leaving the screen entirely. The margin – and, thus, the canvas size altogether – was also not exposed for similar reasons.

Users are also permitted to modify the colours of agents' trails. By dividing the trails up into start, middle, and end segments, different colours can be selected for each. Automatic



Visual Range	1.00
Distance	0.49
Cohesion	0.20
Alignment	0.38
Separation	0.55
Avoidance	0.33
Max Speed	0.60
Max See Ahead	0.77
Lifespan	45s

Fig. 13. The exposed parameters whose values, and thus influence, can be adjusted prior to a simulation run using sliders.

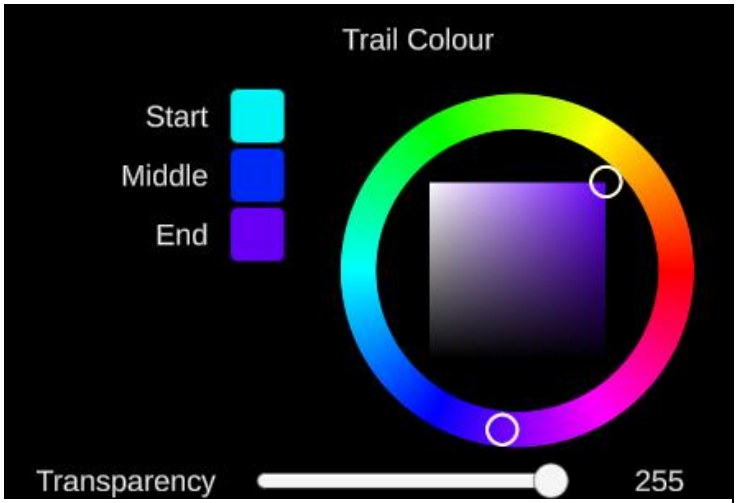
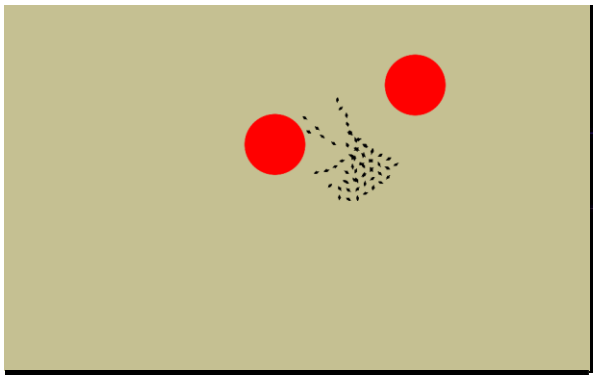
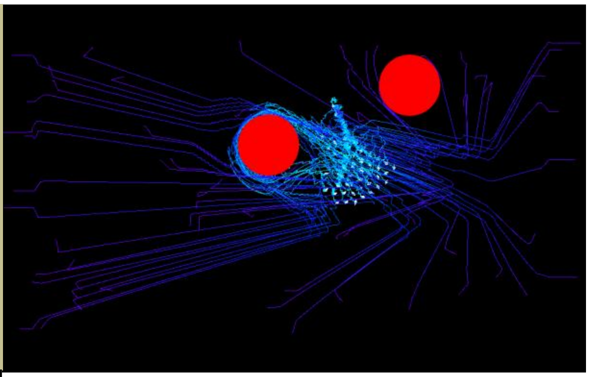


Fig. 14. System UI where the colours of agents' trails can be adjusted on a per-segment basis. Transparency is also supported, and mapped directly to the alpha channel of each colour.

interpolation of colours in between each of the segments illustrates the evolution of an agent's path over time, and creates an illusion of depth, particularly if high-contrast colours are selected. The colour picker was obtained via a third-party library [13].

Throughout the program, pressing the spacebar switches between plain and artistic modes. This was a stylistic choice, to create a contrast between how flocks appear in nature, versus the system's "artistic" eye. Additionally, to draw attention to the impact of obstacle placement, pressing Right Shift toggles visibility of all obstacles in the Scene.





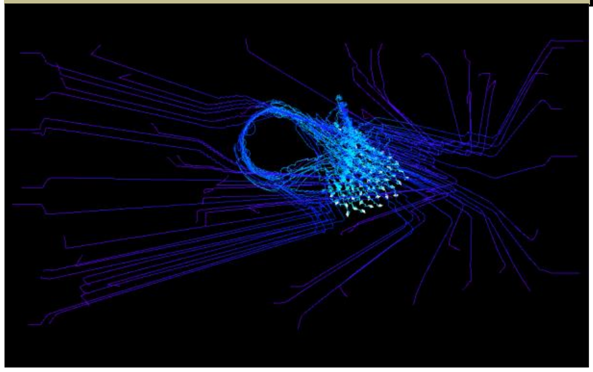


Fig. 15 (upper left). The simulation rendered using artistic mode. Agents display trails.

Fig. 16 (upper right). The same simulation rendered using plain mode. Agents do not display trails.

Fig. 17 (lower left). The same simulation rendered in artistic mode, with obstacle visibility turned off.

4 EXPERIMENTS AND RESULTS

Screenshots of selected output runs are included here, alongside slider values used, where appropriate.

The experiment was broken up into two parts. In the first, data for each run consisted of keeping two of the three “original” rules at 100%, with the third being set to 0%, to highlight the significance of each rule in its absence. This was carried out for lifespans of 5s, 10s, and 20s. This was necessary because of the sheer number of possible value combinations. The remaining slider values were kept constant at the 50% mark. Obstacles were also not used.

In the second, user testing was carried out with no restrictions, and the results were recorded.

Those interested in either viewing the results of a simulation run in real-time or interfacing with the system itself can do so at https://juuu-jiii.github.io/Stork2D/build.html.

4.1 Experiment Data, Part 1

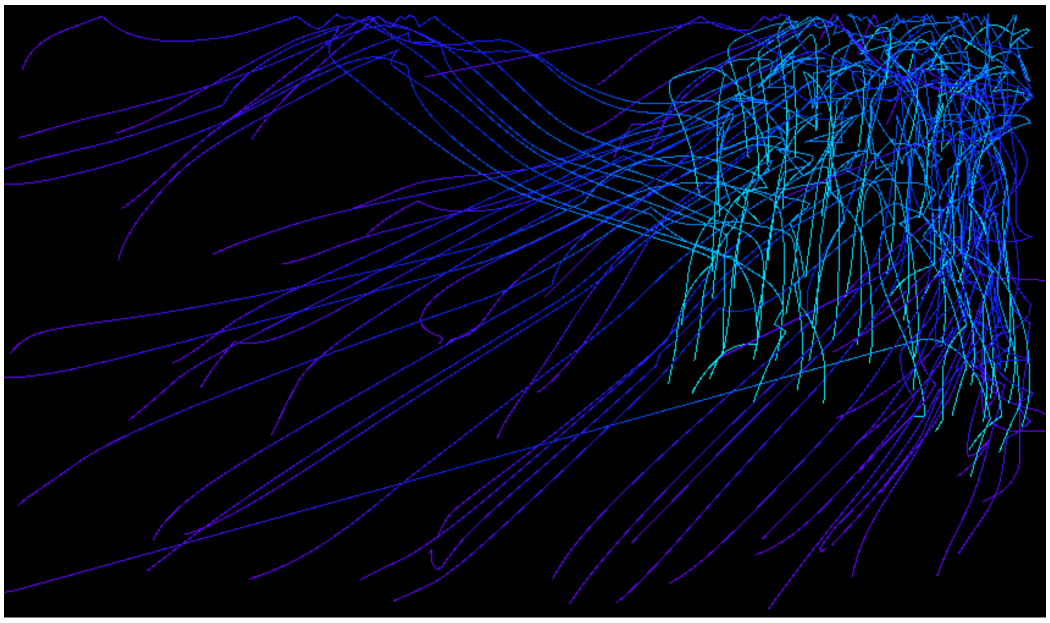


Fig. 18. Cohesion = 0, alignment = 1, separation = 1, and lifespan = 5s.

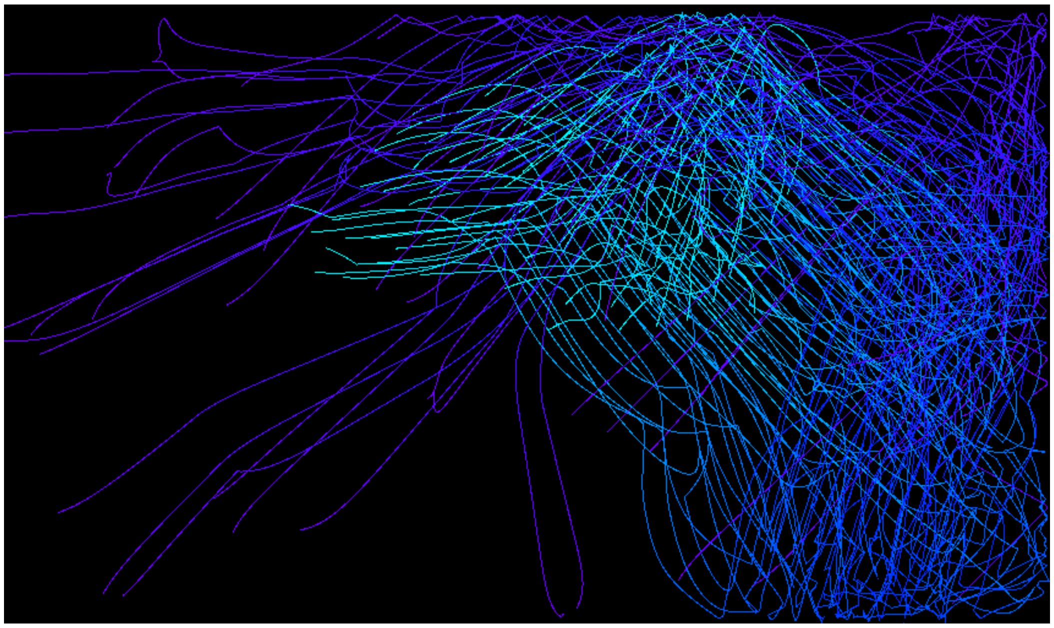


Fig. 19. Cohesion = 0, alignment = 1, separation = 1, and lifespan = 10s.

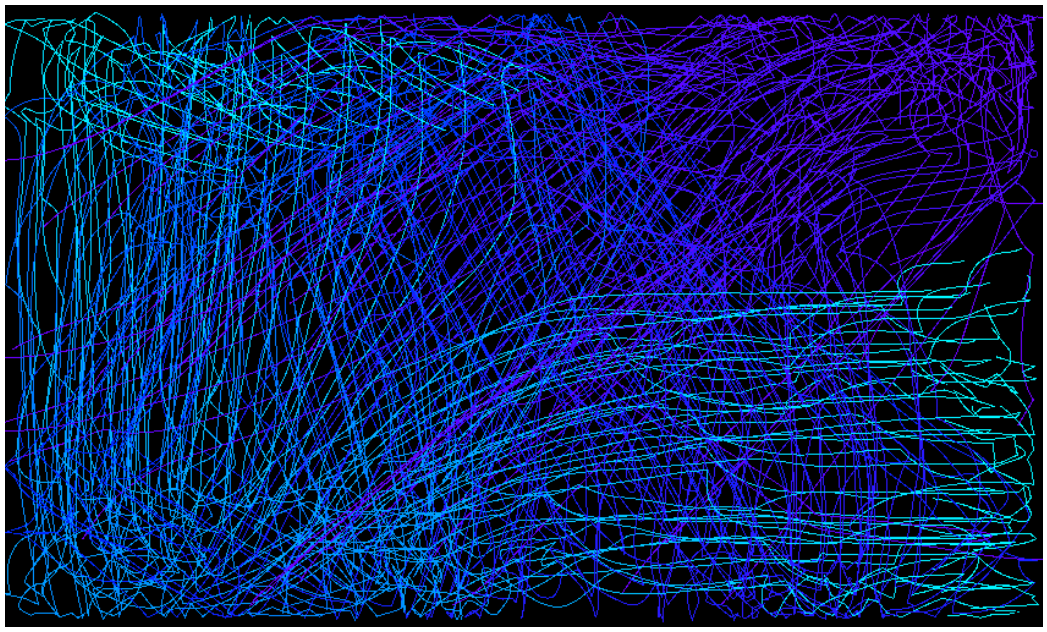


Fig. 20. Cohesion = 0, alignment = 1, separation = 1, and lifespan = 20s.

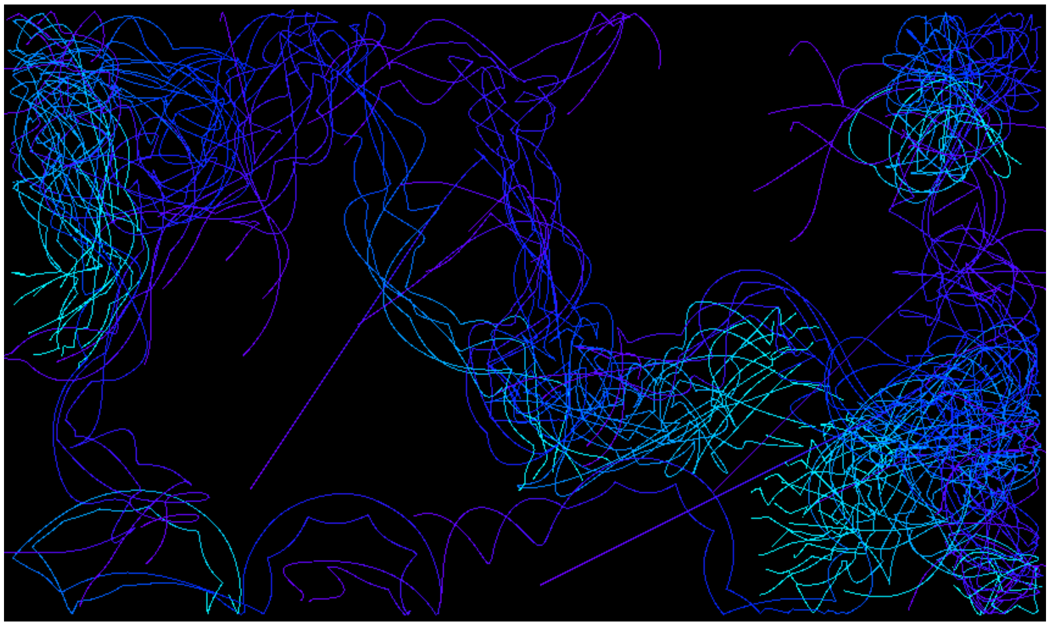


Fig. 21. Cohesion = 1, alignment = 0, separation = 1, and lifespan = 5s.

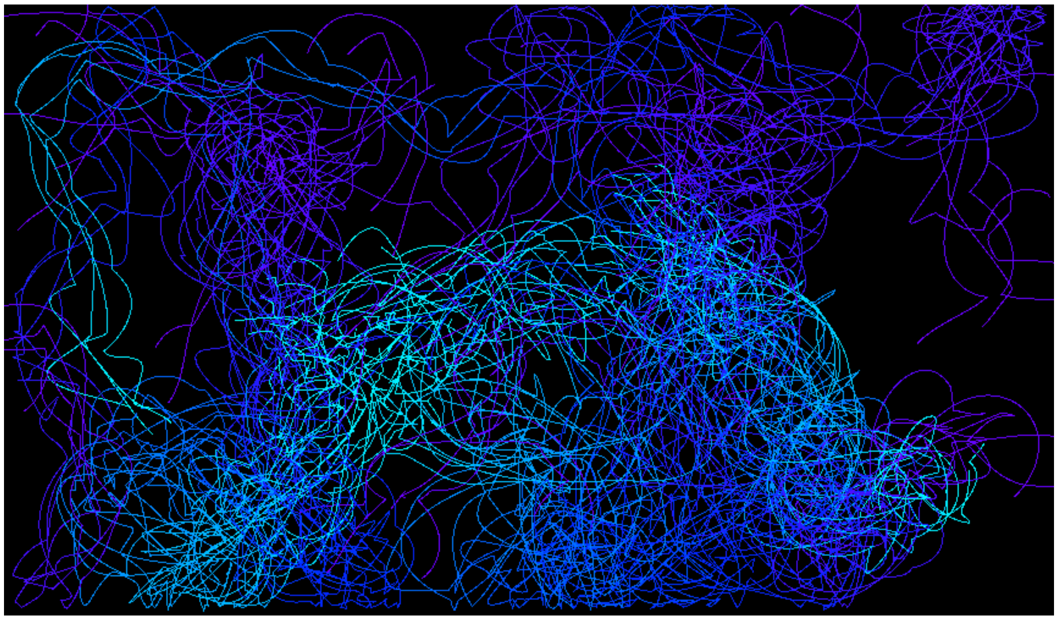


Fig. 22. Cohesion = 1, alignment = 0, separation = 1, and lifespan = 10s.

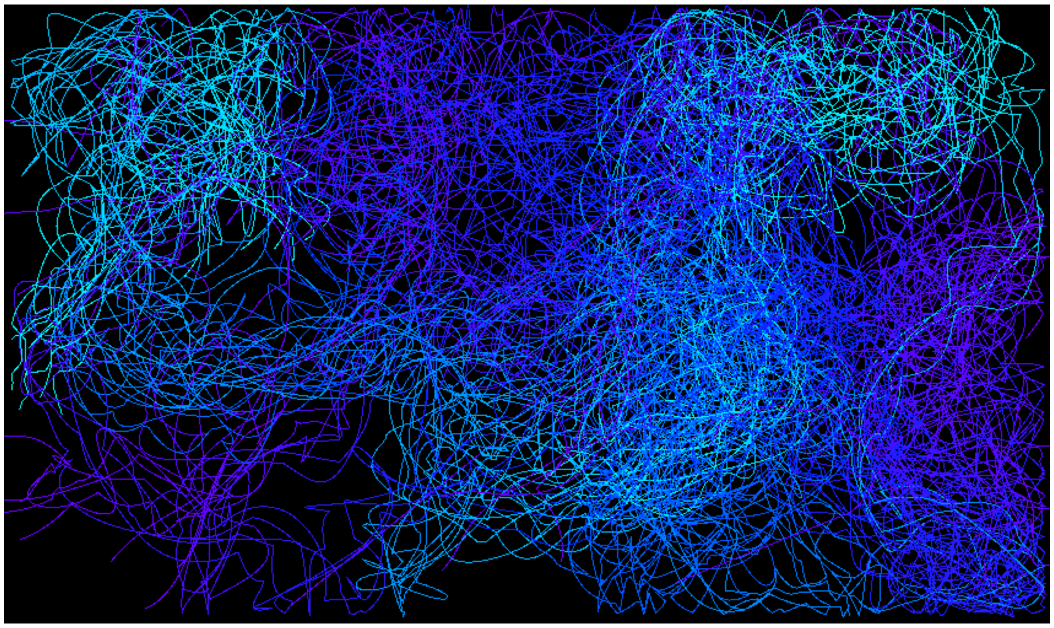


Fig. 23. Cohesion = 1, alignment = 0, separation = 1, and lifespan = 20s.

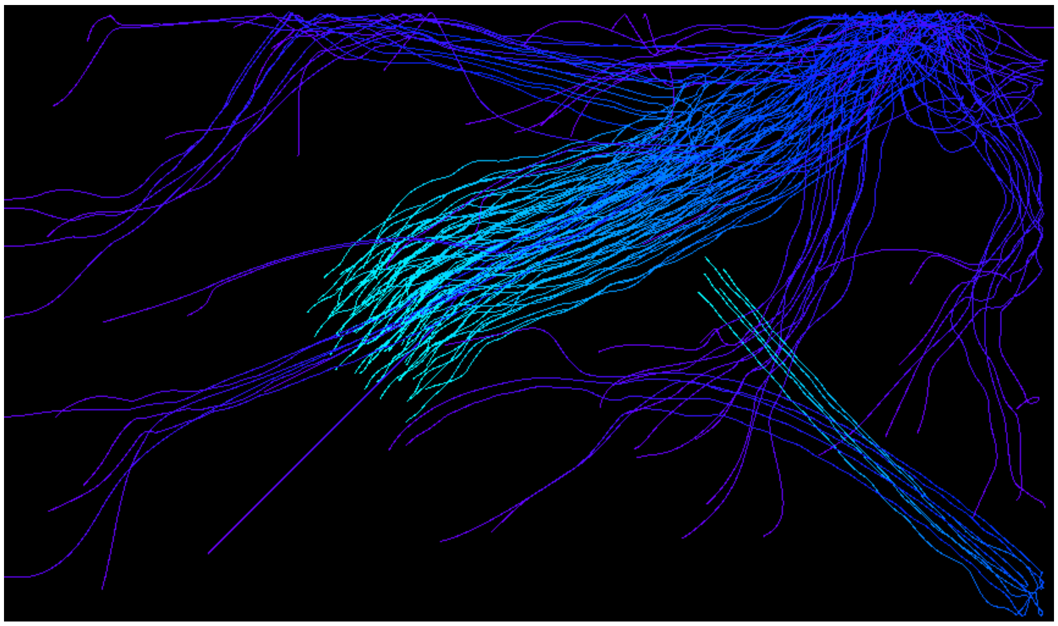


Fig. 24. Cohesion = 1, alignment = 1, separation = 0, and lifespan = 5s.

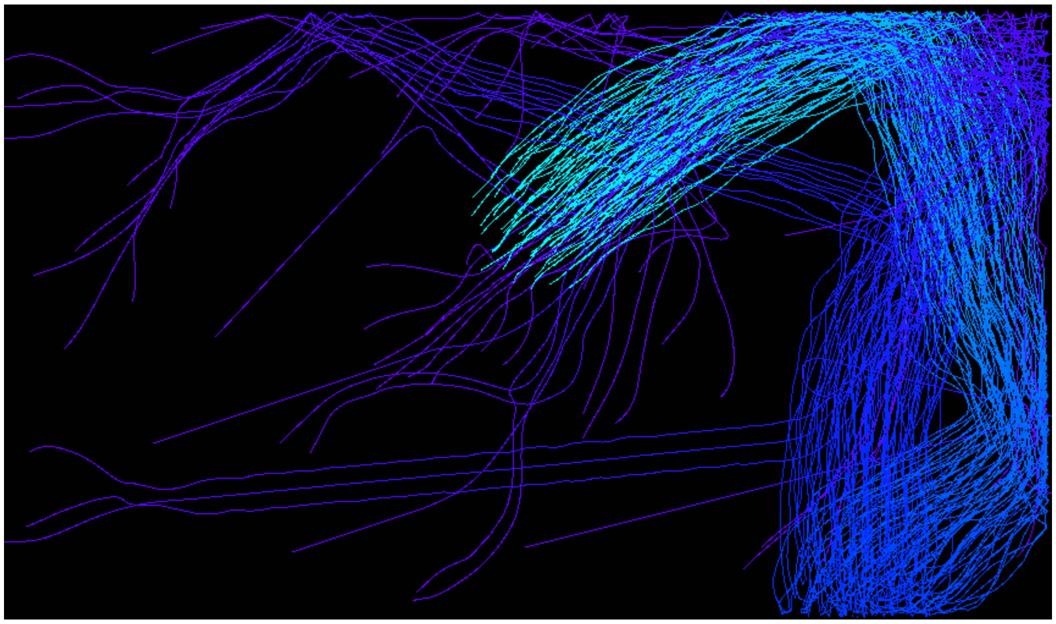


Fig. 25. Cohesion = 1, alignment = 1, separation = 0, and lifespan = 10s.

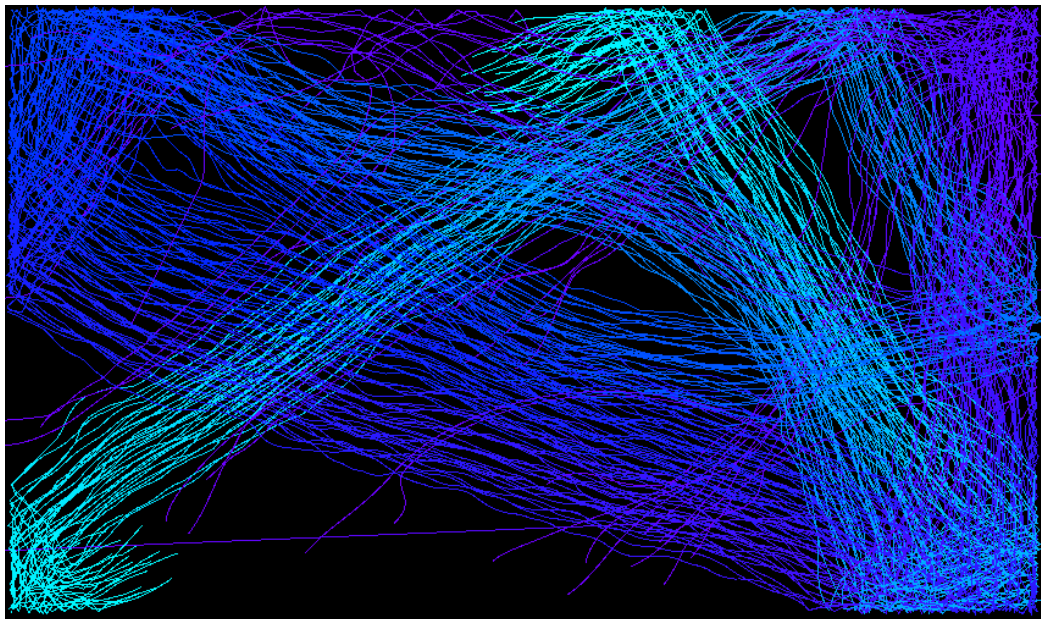
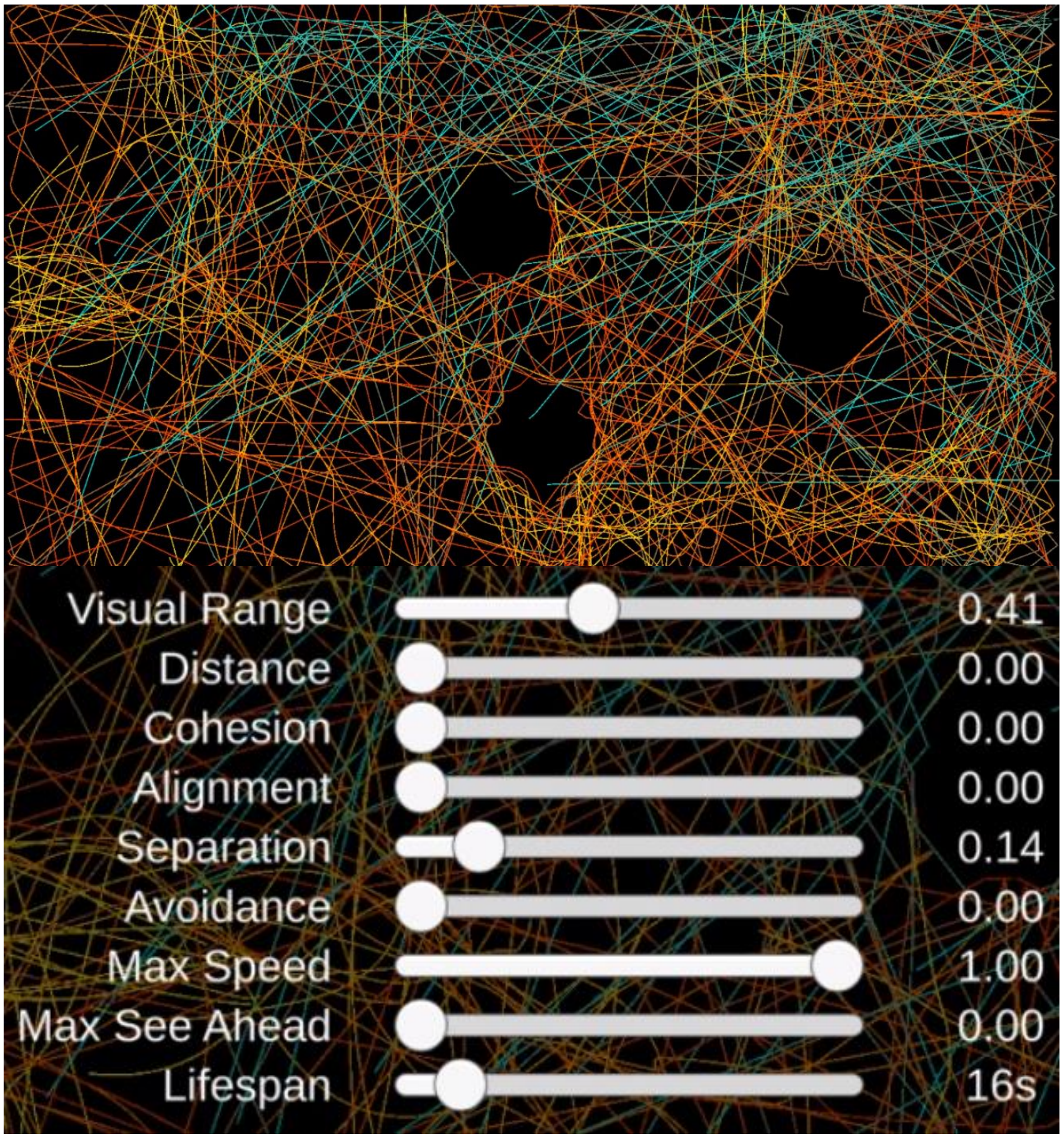


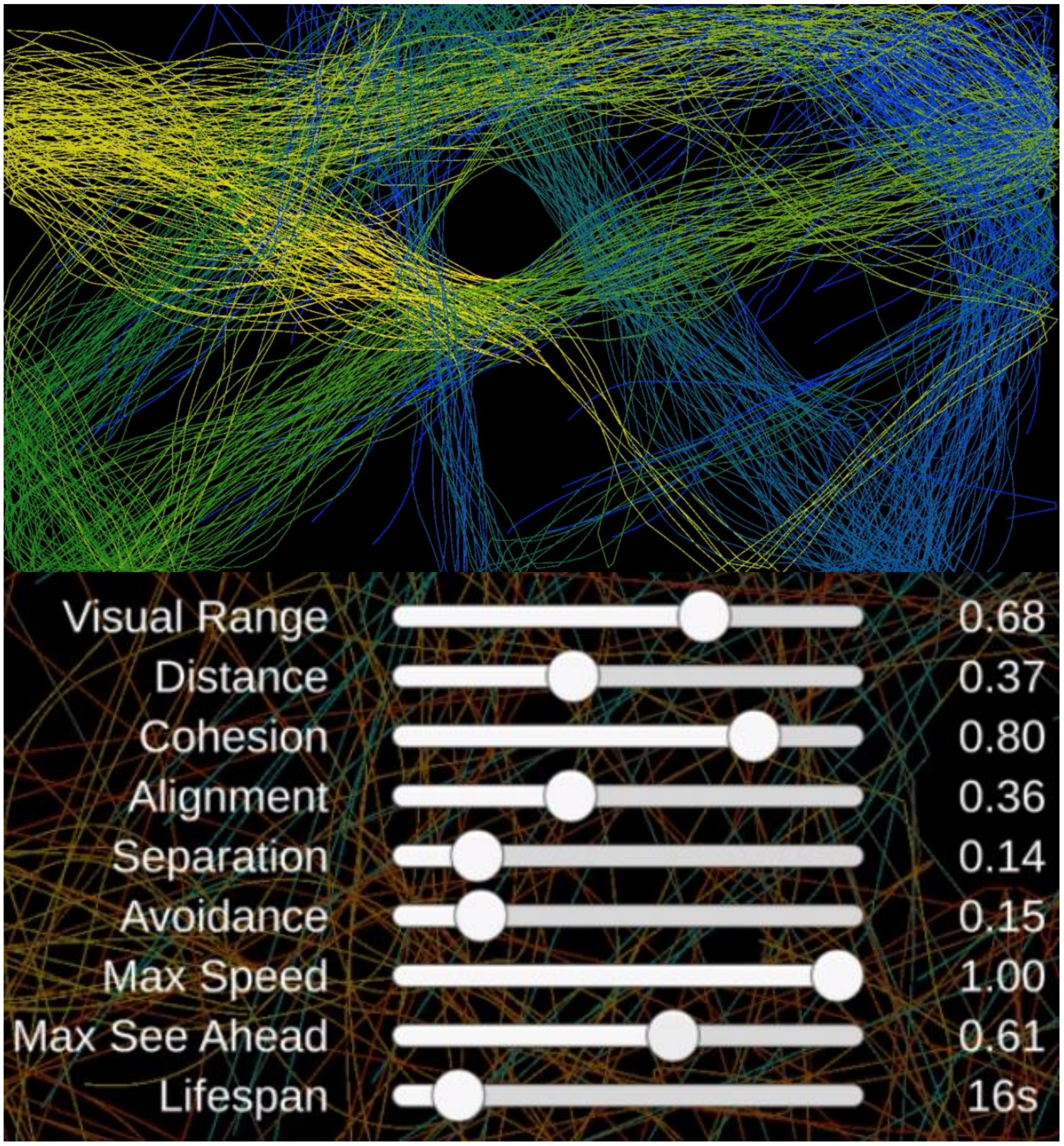
Fig. 26. Cohesion = 1, alignment = 1, separation = 0, and lifespan = 20s.

4.2 Experiment Data, Part 2



Visual Range	0.41
Distance	0.00
Cohesion	0.00
Alignment	0.00
Separation	0.14
Avoidance	0.00
Max Speed	1.00
Max See Ahead	0.00
Lifespan	16s

Fig. 27. User test 1. Note the placement of the three obstacles.



Visual Range	0.68
Distance	0.37
Cohesion	0.80
Alignment	0.36
Separation	0.14
Avoidance	0.15
Max Speed	1.00
Max See Ahead	0.61
Lifespan	16s

Fig. 28. User test 2, using the same three obstacles as in user test 1.

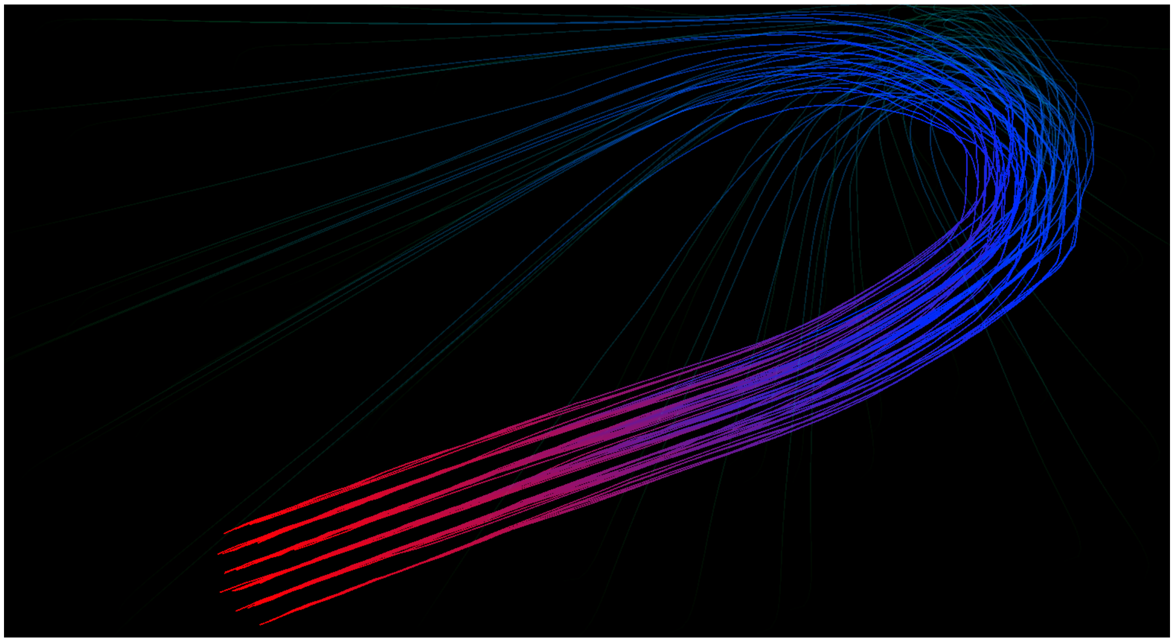


Fig. 29. User test 3. 5s lifespan. The left mouse button was clicked on the upper-right corner and dragged, in an arc, down to the lower-right corner.

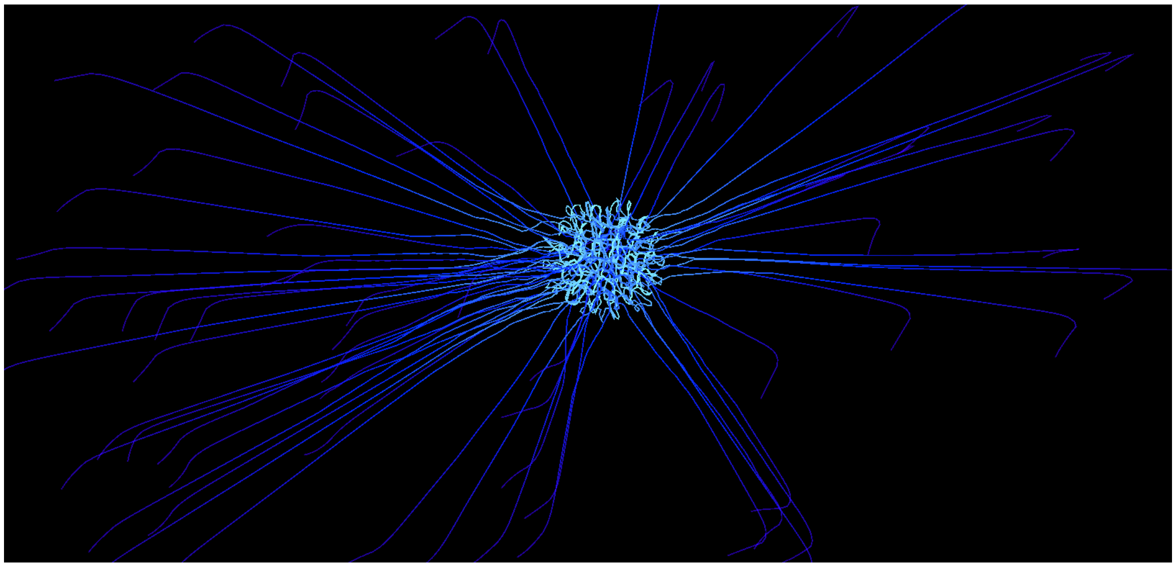


Fig. 30. User test 4. 5s lifespan. The left mouse button was clicked and held in the centre of the screen for the entire duration of the simulation run.

5 EVALUATION AND DISCUSSION

Because the initial motivation behind the creation of this system was to create an interactive approach to appreciating flocking as an art, it was decided that the system would be evaluated in two parts: based on how well the movement of flocks were conveyed with different parameters, and how users interacted and interfaced with the system.

However, beyond this, it was challenging to ascertain, define, and standardise methods by which to evaluate the output generated by the system. With the sheer number of variables, it was necessary to restrict the dataset to a select few, hence the conditions set for the first part of the experiment. This was evaluated based on how well the significance of each rule was highlighted. In the second part of the experiment, it was determined that the values of the sliders were not as important as users' actions themselves. In turn, said actions were monitored closely to see which interactive features of the experience users paid most attention to. Any side remarks or comments were also noted.

5.1 Interpretation of Experiment Data, Part 1

5.1.1 Cohesion = 0, Alignment = 1, Separation = 1

This output appeared to be very disorderly, and this lack of organisation did not seem to improve over time. The influence of the alignment and separation rules are very apparent, however; most agents travelled in similar directions, as evidenced by the near-parallel lines of similar hues throughout. Distance was maintained between each agent as well.

5.1.2 Cohesion = 1, Alignment = 0, Separation = 1

Here, output appeared squiggly and doodle-like. As time passed, multiple flocks merged into a single flock, which then travelled together. This is expected, as alignment controls the movement of agents such that they attempt to match velocities with their neighbours. Cohesion and separation appear to work against each other, here, as there are some moments where agents come very close to each other, and others where they move noticeably far apart.

5.1.3 Cohesion = 1, Alignment = 1, Separation = 0

Output in this scenario appeared very linear and uniform. Most - if not all - agents moved in the same general direction, very close to one another, regardless of time elapsed. Again, this is expected, since separation is the only rule amongst the three that works to keep flock members apart from each other. Its absence here is very apparent.

5.2 Interpretation of Experiment Data, Part 2

Users, upon first interfacing with the system, spent a decent amount of time configuring slider values and obstacle placements before running the simulation. This was perhaps because of the large amount of data and adjustable parameters presented on the simulation setup screen, which, in turn, possibly led to cognitive overload. After the first run, however, they were quick to experiment with further values in subsequent runs.

Although obstacle placement and theme switching were options that oftentimes were quickly forgotten, users appeared to find delight in interacting with agents using the mouse. The images on the previous page were generated solely using user interaction. One user made the unexpected remark that the system presented potential to act as a source of, and starting point for, artwork inspiration, “especially with the click-and-drag feature.”

In line with the comment in the previous paragraph, it was observed that giving users some degree of control over the creative process while letting an underlying algorithm deal with the heavy lifting in the drawing department produced markedly positive responses. The behaviour arising from agents’ interactions amongst one another seemed to allow users to focus on immersing themselves in, and even enjoying, the experience of interfacing with the system and creating custom pieces.

5.3 Discussion, Weaknesses, and Next Steps

A notable observation made during the experimentation process was the role of the lifespan rule. In general, longer simulations (longer than 20 seconds) resulted in noisier data; agents tended to draw over existing paths due to limited canvas space. However, selection of colours with high contrast appeared to mitigate this issue, since the differences gave an illusion of depth. Shorter lifespans (5 to 20 seconds) naturally resulted in more minimalist results with less opportunity for emergent behaviour to reveal itself. Nevertheless, it appeared that user interaction under this circumstance produced controlled, yet visually compelling results.

The current implementation does not account for an agent’s field of view. As detailed in Section 3.2, an agent’s visibility region is a circle. This is not very physically accurate, however, for organisms in a flock tend to not have the capability to turn their necks around fully. This means that narrower flock formations, such as that of a trail of ants, are not currently supported. A future iteration could include modifications to the calculations determining the visibility region such that it becomes more of a visibility arc defined by both a radius and an angle instead.

As noted earlier, the system does not leverage Unity’s inbuilt physics library, and instead relies on manually-programming scripts to handle agent movement. As sources consulted to create the simulation described implementation details primarily in terms of pseudocode, this choice was deliberate, to ensure maximum reproducibility both inside and outside of Unity. A downside that was created because of this is the fact that collision avoidance and handling is not perfect. This is illustrated in Figures 27 and 28; observe that the paths agents take do not fully respect the boundaries of obstacles placed. They move into the obstacle for a short distance, before turning and heading in a different direction. The accuracy of future iterations, then, could potentially benefit from replacing the existing implementation with one that fully leverages rigid bodies and Unity's physics library, albeit at the cost of reproducibility.

A factor not included in any of said evaluation criteria was that of aesthetics. Simply put, it was difficult to decide what elements tended to lead to visually pleasing results. Part of the reason for this was, again, due to the large number of customisable parameters. Removing individual rules was undoubtedly a good start, however, since it highlighted the contributions of each. Perhaps next steps might include running further tests and removing other rules from the equation, before obtaining feedback from human judges to determine which rules are most influential in generating appealing output.

It might be worth investing some time into investigating further the potential use of this system as a tool in creating artwork. For this, additional user studies will need to be conducted, with questions geared more toward the system interface's ease of use, as well as overall functionality. Questions aimed at evaluating the aesthetics of generated output could also be asked.

6 CONCLUSION

The system presented in this paper was developed as an approach for depicting visually and interactively the complex motions that Reynolds's boids simulation produces. Based on the results obtained from experiments, it can be concluded that the system does convey various types of flock movements based on supplied parameters, despite some physical inaccuracies caused by implementation and design choices. It can also be concluded that, regardless of the fact some features were given more attention by users than others, the interactivity and customisability enhances the overall appreciation of emergent observable group behaviours produced by the flocking algorithm. Future work could involve improving the physical accuracy of the simulation, conducting further experiments to determine the parameters – or combinations thereof – that produce aesthetically pleasing results, and investigating the potential of the system to be used as an art tool.

REFERENCES

[1] Craig W Reynolds. 1995. Boids (Flocks, Herds, and Schools: a Distributed Behavioral Model). Boids. Retrieved from https://www.red3d.com/cwr/boids/

[2] Craig W. Reynolds. 1987. Flocks, herds and schools: A distributed behavioral model. Proceedings of the 14th annual conference on Computer graphics and interactive techniques - SIGGRAPH (1987). DOI:https://doi.org/10.1145/37401.37406

[3] Sahil Gupta. 2021. Introduction to Swarm Intelligence - GeeksForGeeks. GeeksForGeeks. Retrieved from https://www.geeksforgeeks.org/introduction-to-swarm-intelligence/

[4] Conrad Parker. 2007. Boids Pseudocode. Boids Pseudocode. Retrieved from http://www.kfish.org/boids/pseudocode.html

[5] David M Bourg and Glenn Seemann. 2004. AI for Game Developers. O'Reilly.

[6] Ben Eater. Boids algorithm demonstration. Ben Eater. Retrieved from https://eater.net/boids

[7] FROST - A Mesmerizing iOS Game. FROST - A Mesmerizing iOS Game. Retrieved from http://frost-game.com/

[8] Lifelike by kunabi brother. Lifelike by kunabi brother. Retrieved from http://www.lifelikegame.com/

[9] SWARM - Interactive Installation - Sometimes | Design Studio. sometimes. Retrieved from https://sometimes.design/projects/swarm-installation/

[10] Fernando Bevilacqua. 2013. Understanding Steering Behaviors: Collision Avoidance. Envato Tuts+. Retrieved from https://gamedevelopment.tutsplus.com/tutorials/understanding-steering-behaviors-collision-avoidance--gamedev-7777

[11] Scripting API: Physics2D.Raycast - Unity. Unity Documentation. Retrieved from https://docs.unity3d.com/ScriptReference/Physics2D.Raycast.html

[12] Scripting API: RaycastHit2D - Unity. Unity Documentation. Retrieved from https://docs.unity3d.com/ScriptReference/RaycastHit2D.html

[13] Unity Color Picker - UnityList. UnityList. Retrieved from https://unitylist.com/p/qes/Unity-Color-Picker

[14] Dan Gries. 2013. ColorBoids: The boid algorithm in five dimensions | Rectangle World. Rectangle World. Retrieved from http://rectangleworld.com/blog/archives/952

APPENDIX A: ACCESSING THE SYSTEM

A site summarising paper details has been deployed at: https://juuu-jiii.github.io/Stork2D/. The system is also live and accessible at: https://juuu-jiii.github.io/Stork2D/build.html.

NOTE: Due to the way resolution is handled, ensure that the maximise button on the lower-right corner of the frame is clicked before the program finishes loading.

APPENDIX B: SOURCE CODE

Source code for this project can be found at: https://github.com/juuu-jiii/Stork2D.