Northeast Journal of Complex Systems (NEJCS)

Volume 3
Number 1 *Special Issue: NERCCS 2021 Papers* Article 6

March 2021

Flocc: From Agent-Based Models to Interactive Simulations on the Web

Scott Donaldson
Open Set, scott@openset.tech

Recommended Citation
Donaldson, Scott (2021) "Flocc: From Agent-Based Models to Interactive Simulations on the Web," *Northeast Journal of Complex Systems (NEJCS)*: Vol. 3 : No. 1 , Article 6.
DOI: 10.22191/nejcs/vol3/iss1/6
Available at: https://orb.binghamton.edu/nejcs/vol3/iss1/6

Flocc: From Agent-Based Models to Interactive Simulations on the Web

Scott Donaldson[1*]
[1] Open Set, Washington DC, USA
[*] scott@openset.tech

Abstract
Agent-based modeling (ABM) is a computational technique wherein systems are represented through the actions and interactions of many individual entities ('agents') over time. ABM often attempts to elucidate the unpredictable, high-level behavior of systems through the predictable, low-level behavior of actors within the system. There are currently few software or frameworks for ABM that allow modelers to design and build interactive models on the web, for a wide audience as well as a scientifically literate audience well-versed in complexity, models, and simulations. Flocc is a novel framework for agent-based modeling written in JavaScript, the lingua franca programming language of the web (which can also run on servers or one's machine). In this paper, we present Flocc's main features and show how it can be used by scientists, data journalists, web developers, and others to create web-based simulations able to be viewed and interacted with by anyone with a modern web browser. By lowering the barrier to entry to complexity science, we contend that Flocc shows promise as a pedagogical tool as well as a software for exploring complex systems.

1. Introduction

Agent-based modeling (ABM) is an approach toward modeling complexity across domains, from political science and economics to population ecology, epidemiology, and more. In ABM, individual entities ('agents') are represented as distinct data representations, along with rules for their behavior (actions and interactions). A simulation takes place over many discrete time steps, and a modeler may hope to gain insight into the high-level behavior of the system being modeled through the logic of its parts. Simulations (including ABM), as philosopher Manuel DeLanda argues, give conceptual legitimacy to the notion that complex behavior can emerge from interactions between the elements of a system at all: "Simulations can play the role of laboratory experiments in the study of emergence[,] complementing the role of mathematics in deciphering the structure of possibility spaces" [1]. By designing, running, and analyzing agent-based models, we can gain valuable insights into complex systems.

1.1 Agent-based modeling and complex systems

A canonical example of an agent-based model exhibiting emergent behavior is the flocking model [2] designed by artificial life researcher Craig Reynolds. In the flocking model (from which Flocc takes its name), individual agents — affectionately nicknamed ‘boids’ — are modeled spatially in two dimensions, and follow three simple rules:

1. Avoid collisions with nearby boids
2. Attempt to match velocity with nearby boids
3. Attempt to stay close to nearby boids

In a typical run of the flocking model, boids begin randomly dispersed both in position and direction. However, as the simulation advances in time and boids move in space, they begin to form groups (‘flocks’) that emerge and disperse, identifiable and persisting over time but not reducible to their constituents. As writer M. Mitchell Waldrop describes, “What is striking about these rules is that none of them said ‘Form a flock’... the rules were entirely local, referring only to what an individual boid could do and see in its own vicinity… And yet flocks *did* form, every time” [3]. As an example of emergence, flock formation is intuitively graspable, and the computational implementation demonstrates how well-defined, simple rules that individuals follow can lead to such group-level behavior.

While Reynolds’ original program for the flocking model was written in Lisp, most if not all programming languages today can be used to write ABM. And, while Reynolds’ program was bespoke, using a few packages for graphics and geometry, there are currently several software libraries with the sole purpose of designing and running agent-based models.

1.2. Software for ABM

A popular ABM software is NetLogo [4], a desktop application that allows users to write model code (in its own custom scripting language) and run and visualize simulations. NetLogo includes many features for designing and developing models, but is difficult to interface with other software and requires downloading the application in order to run models. There is an attempt to port NetLogo models to the web, but it does not contain all of the features of the desktop application and the website states, “The desktop version of NetLogo is recommended for most uses” [5].

Another library, Mesa [6], is close to Flocc in spirit, but uses the Python programming language. Like Flocc, it can be used with 3rd-party libraries (there is a robust and growing collection of Python-based tools for data science and analysis). One of Mesa’s strengths (which Flocc does not aim to replicate) is its ability to integrate with 3rd-party Python-based tools for data analysis. Mesa also affords model and data visualization in a web browser using JavaScript. However, to place a Mesa model on a web server and visualize it on a web page is a non-trivial task, making it an unattractive tool when considering publishing models for a wide audience.

Two more software, AgentScript [7] and AgentBase [8], are written in JavaScript (or have JavaScript APIs) with the purpose of building and sharing browser-based models. Both make it relatively easy to create interactive simulations that can be viewed by a large audience. However, each focuses primarily on visualizing models themselves, providing few tools for visualizing data about models. Since many agent-based models lack a spatial or network dimension, this makes it difficult to meaningfully represent such models.

We introduce Flocc to address the gaps in the above ABM software.

Flocc is an open-source, MIT-licensed JavaScript library for designing and building agent-based models. Flocc models can easily be distributed on the web, making it possible for anyone with a modern web browser to view and interact with them. Additionally, those distributing/publishing models can use free platforms, managed services, or self-hosted web pages on which to place the models. Since it is a JavaScript library, Flocc models can integrate with any 3rd-party code written in JavaScript, as well as interface with APIs. Lastly, to explore the parameter spaces of models and analyze the results of simulations, Flocc can also run in a Node.js environment on a remote server or one's machine. A key benefit is that the same code can be used to produce interactive simulations for a lay audience as well as to perform exploration and data analysis of models.

2. Overview of Flocc

Flocc emphasizes customizability, flexibility, and a relatively unopinionated approach to developing computational models. As with agent-based modeling and programming more generally, while there are more straightforward and suggested approaches, there is no 'right' or 'wrong' way to build a Flocc model. There is usually more than one way to build a given model feature, both syntactically and procedurally, using built-in Flocc components, extending them, or drawing from the larger JavaScript ecosystem.

The types of objects that Flocc provides can be divided into roughly two categories — those used for modeling/analysis and those used for visualization. Modeling objects include the Agent, Environment, Network, Terrain (a 2-dimensional grid for cellular automata), and other classes. Visualization objects include the CanvasRenderer (a spatial representation of an Environment with Agents), renderers for line charts, histograms, and heatmaps, and a tabular view of data. There is also a separate, add-on library called Flocc UI[1] that can be used to build interactive interfaces in the browser (similar to the popular dat.GUI library). It provides components such as sliders, inputs, and buttons to allow users to update model variables and parameters, reducing the friction in creating explorable simulations. Flocc UI is still in beta development and will not be covered in this paper, but we expect to publish it in the future.

2.1. Environment

The core of any Flocc model is the Environment class. It provides functionality for both the space and time in which a simulation occurs. A model typically only has a single environment, which is instantiated using the `new` keyword:

```
const environment = new Environment();
```
[2]

An Environment can be configured with a number of options, such as a `width` and `height` for spatial environments, and a boolean value for `torus` (whether or not a 2-dimensional spatial environment wraps from one edge to the other).

```
const environment = new Environment({
  "width": 500,
  "height": 500,
  "torus": false
});
```

An empty Environment is of little value to an *agent-based* model — in general, models rely on the Agent class to add individual agents to the simulation.

2.2. Agents

Like all other classes in Flocc, Agents are instantiated with the `new` keyword:

```
const agent = new Agent();
```

Agent data can be added or removed at any time, and does not have to adhere to the same data type once it is set (although for code maintenance this is discouraged). Values can be anything from simple types such as booleans, numbers, and strings,

1 https://github.com/o-p-e-n-s-e-t/flocc-ui
2 Throughout this paper, we will use modern JavaScript syntax, such as `const` and `let` for variable declaration. In early versions of JavaScript, only `var` could be used to declare variables, and older web browsers may not support or recognize newer syntax. Flocc itself makes few concessions to older browsers, assuming that in general, most browsers are automatically updated to the latest version.

to more complex types such as arrays and objects, to other agents in the environment and beyond[3]. Data may be set using the `set` method:

```
agent.set("favorite_food", "pizza");
agent.set("hunger",                                    60);
```

The `set` method can also set multiple key-value pairs at once by passing a JavaScript object, so the above code could be rewritten:

```
agent.set({
  "favorite_food": "pizza",
  "hunger": 60
});
```

Any of these values can be retrieved using the `get` method (ex. calling `agent.get("hunger")` will return the number 60).

The `set` method also allows for functions as values, which return dynamically calculated whenever they are retrieved. This can specify relationships between the agent's pieces of data, or between agent and environment data (an environment is itself a special type of agent, and can call the `set` and `get` methods to allow for high-level, global variables). For example, by passing a function for `favorite_beverage`, this agent's drink of choice can return different values depending on its `favorite_food`:

```
agent.set("favorite_beverage", function(agt) {
  if (agt.get("favorite_food") === "pizza") {
    return "soda";
  } else {
    return "water";
  }
});
```

Note that the function that is passed takes a single parameter, which refers to the agent on which it is being called. Usually, the function is declared separately and used for all agents when initially populating the environment:

```
function favorite_beverage(agt) { … }
agent.set("favorite_beverage", favorite_beverage);
```

3 In the case of complex types, the usual warnings about object reference apply. If two agents a and b both have a key-value pair that refers to the same object, and a piece of data on that object changes (say, by agent c), then it will be changed for both a and b.

A special, reserved key-value pair is the `tick` value. This must be a function, and describe the behavior of the agent over time. There are few restrictions on `tick` functions otherwise. They may run synchronously and/or asynchronously. Synchronous `tick` functions ensure that any changes in data occur simultaneously across all agents, while in asynchronous functions they occur sequentially, either in the order agents were added to the environment or in a randomized order (see section 2.3. below). `tick` functions can update the agent's own data, that of other agents, or even environment variables; that is, the restrictions on `tick` functions come more from JavaScript as a programming language than anything else. As in the above example, a typical `tick` function will be declared separately and then added to all agents as they are instantiated and added into the environment:

```
function tick(agent) {
  if (agent.get("hunger") > 100) {
    if (agent.get("favorite_food") === "pizza") {
      agent.decrement("hunger", 10);
    } else {
      agent.decrement("hunger", 5);
    }
  } else {
    agent.increment("hunger", 3);
  }
}

environment.addAgent(new Agent({
  "favorite_food": "pizza",
  "hunger": 60,
  "tick": tick
});
```

The above is an example of asynchronous updating. As soon as `decrement` or `increment` are called, that agent's `hunger` value is immediately updated. If code is added later in the function to reference the agent's `hunger` value, it will return the updated value, just as it will if another agent references it. However, the same function could be rewritten to run synchronously:

```
function tick(agent) {
  let hunger = agent.get("hunger");
  if (agent.get("hunger") > 100) {
    if (agent.get("favorite_food") === "pizza") {
      hunger = hunger - 10;
    } else {
      hunger = hunger - 5;
    }
  } else {
      hunger = hunger + 3;
  }

  return { "hunger": hunger };
}
```

In synchronous updating, instead of calling an agent's `` `set` ``, `` `increment` ``, or `` `decrement` `` methods within the function, an object of key-value pairs should be returned, which will update the agent's values only after the `` `tick` `` function has run for all agents in the environment in a given time step. Unlike asynchronous updating, calling `` `agent.get("hunger")` `` within this function, or referencing another agent's `hunger` value will always return the value at the start of the time step. This helps to avoid runaway patterns where an agent updates based on another's changes, which in turn are effected by another's changes, etc.

Note, however, that in a model where one agent is able to make changes to another agent's data, these changes must run asynchronously, by referencing the other agent and calling its `` `set` ``, `` `increment` ``, or `` `decrement` `` methods. In fact, the concept of synchronous updating introduces a problem here — if agent a updates both agent b's and its own data, and agent b updates both agent a's and its own data, then the priority of these changes must be dependent on the order of activation (that is, it is necessarily asynchronous).

2.3. Running a model

Once agents have been added to an environment (usually adding many at a time with a `` `for` `` loop), the model is ready to run over time. To do this, the program should call `` `environment.tick()` ``, which will advance the model forward by one discrete time step, calling each agent's `` `tick` `` function once. By default, the environment loops over every agent in the order they were added. However, the order of agent activation can have significant effects on the emergent behavior of the system [9]. For example, in a spatial model, if agents reference nearby agent data, and updates occur sequentially, it is easy for cascading patterns to emerge that are an artifact of the order of activation, rather than from the behavior itself. By passing a configuration object, it is possible to override this default[4]:

```
environment.tick({
  "randomizeOrder": true
});
```

4 Most ABM software defaults to randomized order, and Flocc will shift to this regime in the release of version 0.6.0 (so users will need to opt into, not out of, sequential activation).

Alternatively, a modeler may not want to uniformly activate each agent in one time step. For example, uniform activation is more relevant to physical simulations, where motion is guaranteed to occur deterministically, than social simulations, where individuals act sporadically or unpredictably. To randomly activate only one or a few agents, the configuration object should contain these keys:

```
environment.tick({
  "activation": "random",
  "activationCount": 5
});
```

In the above examples, the environment progresses one time step. However, a single step will likely not reveal much about the emergent behavior of the model, as complexity arises from the continuous interaction among parts of a system over time. An environment can progress several time steps at once by either passing a number as the single parameter to `` `environment.tick` ``, or by passing it as part of a configuration object:

```
environment.tick(100);

environment.tick({
  "count": 100
});
```

The above examples are identical, both advancing the environment by 100 time steps. However, in these examples, assuming there is a visual representation of the model (a renderer, described in the next section), the progression over time will not be depicted. Instead, 100 time steps will appear to have instantaneously passed, and any visualizations will now show the state of the environment and agents after the fact.

Often, it is useful to show progression and evolving system behavior over time. Rather than advancing the environment by a large number of time steps, a built-in JavaScript function can be used to ensure updates occur regularly over time and are rendered each time `` `environment.tick` `` is called. To do this, `environment.tick` is usually wrapped in an outer function:

```
function run() {
  environment.tick();
  requestAnimationFrame(run);
}
run();
```

The built-in browser function `` `requestAnimationFrame` `` takes as a parameter a callback function, which will be run on the next available frame for the browser to be 're-painted' (i.e. the display's refresh rate). As the name suggests, this can be used to produce relatively smooth animations. In the above example, once `` `run` `` is called for the first time, since it calls itself (via `` `requestAnimationFrame` ``), the animation will begin running and never stop. It can be useful to 'shortcut' and stop animating under certain conditions, as below:

```
function run() {
  environment.tick();
  if (environment.time < 1000) requestAnimationFrame(run);
}
run();
```

The above example will begin running but stop once the environment has progressed 1,000 time steps (on a 60 Hz display, this will take 16-17 seconds).

2.4. Environment helpers

There are three classes that provide additional functionality for agents and environments. They can be considered 'plugins' in the sense that they aren't strictly necessary for many models but can be indispensable for some. Briefly, the three helpers are:

1. *Terrain*: Extends the environment to allow for space- and time-efficient cellular automata-based models (CA). Rather than populating an environment with an agent for each cell of a CA, terrains come pre-populated and follow slightly different update rules than agents in a non-CA model. For example, since terrains are expected to be visualized by a CanvasRenderer (see below), their cells contain numeric values for red, green, blue, and alpha channels (or a single value, if running in grayscale mode). However, by using multiple terrains, cells can contain arbitrary data as well.
2. *Network*: Networks provide a means for storing and representing relationships between agents. Relative to libraries built specifically for network analysis such as cytoscape.js [10] and webweb [11], the functionality of networks in Flocc are limited to the basics of undirected edges between agents. However, it is possible to extend networks to include edge attributes (including weights and/or directionality), and to generate different network structures, from random graphs to small world networks to preferential attachment networks and beyond. Upcoming releases of Flocc will include built-in methods for wiring networks in a given structure.

3. *KDTree*: For spatial environments where agents have limited visibility, operations such as finding the nearest neighbor(s) of an agent can be inefficient, typically involving looping over every other agent in the environment and filtering to only those within a certain distance. Over a single time step this has O(n^2) algorithmic efficiency. By using a KDTree, these spatial operations can be performed at O(n log n) efficiency, a significant improvement when there are large numbers of agents in a model. The difference may be negligible in a spatial environment with 500 or fewer agents, but as that number increases, to maintain relatively smooth visualization/animation, a KDTree becomes indispensable.

2.5. Visualizing the model and data about the model

Especially for interactive simulations where numerical output is less important than the first-hand experience of witnessing complex, emergent behavior, it is important to represent models visually. The CanvasRenderer class can be used to visualize environments and agents in two-dimensional space, and several other renderers can be used to visualize aggregate data in other ways. The other renderers are the LineChartRenderer, Histogram, TableRenderer, and Heatmap.

All renderers follow a similar pattern for instantiation — they take the environment as the first parameter, and can take a configuration object as the second. All renderers also have a `` `mount` `` method, which states which DOM element on the web page should be populated with the renderer's visual output. Since a CanvasRenderer attempts to provide a complete visual representation of agents in an environment, it has the most configuration options:

```
const renderer = new CanvasRenderer(environment, {
  "width": 600,
  "height": 300,
  "background": "blue",
  "scale": 2,
  "origin": { "x": 150, "y": 75 }
});
```

The above options will draw a 300 x 600 canvas with a blue background, scaled up 2x with the origin (the upper-left corner, as in computer graphics) set to the coordinates `(150, 75)` (with the lower-right corner, then, showing the coordinates `(450, 225)`). CanvasRenderers will draw agents in the environment differently depending on what agent data they encounter. As with the above `` `tick` `` key-value pair, there are certain pieces of data that have significance for rendering.

```
new Agent({
  "x": 200,
  "y": 125,
  "size": 6,
  "color": "yellow",
  "shape": "arrow",
  "vx": 0,
  "vy": 1
});
```

An agent with the above data would be represented by a yellow triangle at the point `(200, 125)` (for the above CanvasRenderer, this would actually be drawn at `(100, 100)`), 6 pixels wide at the base, pointing due south (following the `` `vx` `` and `` `vy` `` data).

3. Example Model: Race to the Center

Having provided an overview of the basics of the Flocc library, we will now present a model, called "Race to the Center," with more interesting emergent behavior. A highly idealized model of preferential voting dynamics in a two-party system, it attempts to show how candidates from opposing parties might shift their positions in order to attain a greater share of the vote. The full version can be viewed online on the Flocc website: https://flocc.network/examples — here we focus on the code relevant to the behavior of the model, as opposed to its presentational aspects.

This model includes two types of agents, voters and candidates. There are 201 voters (an odd number to prevent a tie vote) and 2 candidates representing opposing political parties. Both voters and candidates are initialized with an `` `x` `` value, which represents their ideological position along a political spectrum (as noted in section 2.5. above, this also allows them to be visualized on a CanvasRenderer). This code adds the voters:

```
for (let i = 0; i < 201; i++) {
  environment.addAgent(
    new Agent({
      "x": utils.gaussian(50, 25),
      "tick": vote
    })
  );
}
```

The voter's `` `x` `` value is drawn from a Gaussian distribution with mean value 50 and standard deviation 25 (most values fall between 0 and 100, clustered around 50). The two candidates begin with `` `x` `` values closer to 0 and 100, respectively, to represent opposing political ideologies:

```
left = new Agent({
  "x": utils.random(0, 25),
  "votes": 0,
  "direction": utils.sample([1, -1]),
  "tick": shift
});

right = new Agent({
  "x": utils.random(75, 100),
  "votes": 0,
  "direction": utils.sample([1, -1]),
  "tick": shift
});
```

The candidates, left and right, are declared in the global scope so that voters can reference them in their `` `vote` `` functions. They are also initialized with 0 votes (this is updated in the `` `vote` `` function), and a `` `direction` ``, either 1 or -1, which will be the direction they choose to move in when they receive a minority of votes.

The `` `vote` `` function is straightforward, incrementing the `` `votes` `` value of the candidate who is closer (in this 1-dimensional ideological space) to the agent in question:

```
function vote(agent) {
  let choice;
  if (utils.distance(agent, left) < utils.distance(agent, right))
{
    choice = left;
  } else if (utils.distance(agent, left) > utils.distance(agent,
right)) {
    choice = right;
  } else {
    choice = utils.sample([left, right]);
  }
  choice.increment("votes");
}
```

In the event both candidates are equidistant from the voter, they essentially flip a coin to choose.

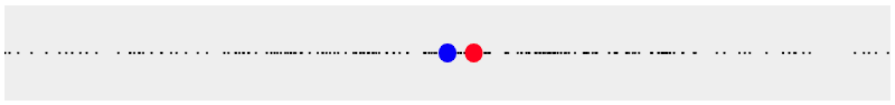
As noted in section 2.2., since voters update the candidates' data, this function must run asynchronously (calling `` `increment` ``). However, because both candidates' `` `tick` `` functions run after all the voters have voted, they are guaranteed to count all votes.
Meanwhile, the two candidates follow a slightly more involved rule:

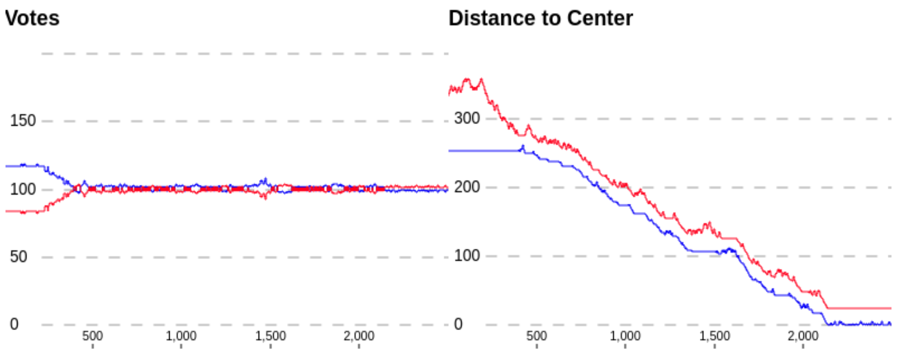
```
function shift(agent) {
  if (agent.get(“votes”) < 101) {
    agent.increment("x", agent.get("direction"));

    const min = agent === left ? 0 : 50;
    const max = agent === right ? 50 : 100;
    agent.set("x", utils.clamp(agent.get("x"), min, max));

    const choices = [1, -1];
    for (let i = 0.4; i > 0; i -= 0.1) {
      if (agent.get(“lastVotes”) !== null) {
        choices.push(
          agent.get("lastVotes") > agent.get(“votes”) ?
          -1 :
          1
        );
      }
    }
    agent.set("direction", agent.get("direction") *
utils.sample(choices));
  }
  agent.set("lastVotes", agent.get(“votes”));

  return {
    “lastVotes”: agent.get(“votes”),
    “votes”: 0
  };
}
```

The bulk of this function only runs when the candidate has received less than 101 (half of the total) votes. If so, the candidate moves in the `direction` they have set — shifting their position in the hopes of gaining more votes in the future. The next lines ensure that they always remain within the ideological bounds of the given space and within their own ‘lanes’ (i.e. the left candidate can never occupy the right half of the space and vice-versa). After this, they may update their `direction`, in a rudimentary model of learning from past mistakes. Based on comparing the previous amount of votes they received to the number they received this time, they will create an array (or list) of either mostly 1s or mostly -1s. If their previous vote count was higher, it will be mostly -1s, and if their previous vote count was lower, it will be mostly 1s. Then, they update their `direction` by multiplying it by a value randomly chosen from this array — so if they gained votes this round, the value will likely be 1, maintaining the direction they’re moving in, and if they lost votes, it will likely be -1, changing course and beginning to move in the opposite direction in order to gain more votes. The possibility of not changing direction means that candidates are not merely deterministically following the path toward more voters. The function finishes by returning the values that will be used for `` `lastVotes` `` and `` `votes` `` in the next round.

Although this is a simple model, the dynamics over time demonstrate the incentive for both candidates to shift positions ever closer to the center of the political space. Below, the figure shows the spatial layout (with candidates, blue and red, slightly larger than voters), along with the number of votes received over time and the candidates' distances from the center at 2,500 time steps elapsed.

The chart with the number of votes over time shows that the left candidate begins with a greater share of the vote (and, not coincidentally, also begins closer to center). However, the right candidate shifts their position until they both have close to half of the vote, around time 400. From there, they are deadlocked, each shifting closer and closer to the center until, slightly after time 2,000, left is at the center and right just a short distance away — this particular simulation's equilibrium state.





The above model, including the code, parameters, and visualization can be embedded on a web page, able to be viewed and interacted with by a wide audience. With a few more lines of code, using Flocc UI, parameters such as the population count and the candidates' learning rate could be easily manipulated by viewers, allowing them to explore how varying these parameters affects the outcome.

4. Current and Future Work with Flocc

The library of examples on the Flocc website (https://flocc.network/examples) shows the broad domains which can be modeled using ABM and Flocc — from the social and political sciences ("Tribute Model," "Iowa Caucuses") to population ecology ("Predator-Prey") to linguistics ("Language Evolution") and beyond. These examples represent adaptations of existing models, as well as original work by the author and others. More models are added to the website regularly in order to demonstrate the expansive world of complexity.

Over the past year, Flocc has been presented at workshops to web developers and to graduate students enrolled in a computational design program. It is also currently being used by researchers at Aarhus University to translate a NetLogo model to the web in order to build an interactive simulation to facilitate remote learning in high schools. In both workshops and in the classroom, relative to existing software, Flocc is useful in that it requires no special software to be downloaded and can be run in a web browser. We are encouraged by these projects and hope to see more student-based work as Flocc finds more usage.

Flocc is an open-source library, and we hope that as more people (across fields and specialties) build models using Flocc, they will identify areas for improvement and provide feedback in the form of bug reports, feature requests, and code contributions. Like all software, Flocc's success will be an emergent, collaborative effort. However, we do have some concrete plans to add new features to Flocc in the near future:

- Officially releasing Flocc UI as an add-on library for parameter-based interfaces
- Adding more network modeling capabilities, including automatic generation of certain network structures, having both directed and undirected networks, incorporating edge weights, and more
- Allowing for server-side (Node.js) based visualization (currently only the browser is supported)
- Lowering the barrier to entry of publishing models by providing the ability to create one's own models on the Flocc website

Finally, a software library is only as usable as its documentation is useful. The Flocc website (https://flocc.network) attempts to provide a comprehensive overview of the library API, but would certainly benefit from more documented examples and tutorials, as well as audience-specific sections for those with more or less experience with programming, web development, agent-based modeling, or domain-specific backgrounds.

5. Conclusion

Agent-based models and simulations are increasingly prominent in the world of interactive data visualization and digital storytelling. In 2020, *The Washington*

Post's most viewed article [12] was an interactive ABM showing how social distancing and quarantine measures could slow the spread of diseases like coronavirus. As opposed to modeling through mathematical parameters (like the coefficients of an SIR model), simulations and ABM in particular provide a more embodied experience, showing through animation and visualization how the parameters, rules, and interactions between individuals in a simulation give rise to complex, system-level behavior.

Flocc provides a robust framework for designing and building ABM, taking advantage of the web as a medium to allow a wide audience to interact with models. It easily integrates with 3rd-party JavaScript libraries, such as Three.js for 3-dimensional visualization, React for complex user interfaces, TensorFlow for leveraging machine learning models, and more. Since the same Flocc model can run in a browser or on a server, it is possible for the same code to be used for interactive digital storytelling as well as computationally intensive parameter sweeps and data analysis. We are proud to present it, and we believe that Flocc can be a valuable tool for computational scientists, researchers, and modelers as well as data journalists, artists, and students of complexity.

References

1. DeLanda, Manuel. *Philosophy and simulation: The emergence of synthetic reason.* Bloomsbury Publishing, 2011.
2. Reynolds, Craig W. "Flocks, herds and schools: A distributed behavioral model." Proceedings of the 14th annual conference on Computer graphics and interactive techniques. 1987.
3. Waldrop, Mitchell M. *Complexity: The emerging science at the edge of order and chaos.* Simon and Schuster, 1993.
4. Wilensky, Uri. "NetLogo." Evanston, IL: Center for Connected Learning and Computer-Based Modeling, Northwestern University, 1999.
5. Weintrop, D., Hjorth, A., Brady, C., & Wilensky, U. *NetLogo Web: Bringing Turtles to the Cloud.* https://netlogoweb.org/
6. Masad, David, and Jacqueline Kazil. "MESA: An agent-based modeling framework." 14th PYTHON in Science Conference. 2015.
7. Densmore, Owen, "AgentScript." RedfishGroup LLC. http://agentscript.org/
8. Wiersma, Wybo. "Agentbase: Agent-based modeling in the browser." Advances in Social Simulation 2015. Springer, Cham, 2017. 451-455.
9. Comer, Kenneth W. "Who Goes First? An Examination of the Impact of Activation on Outcome Behavior in Agent-Based Models." George Mason University, 2014. http://ebot.gmu.edu/handle/1920/9070
10. Franz, Max, et al. "Cytoscape. js: a graph theory library for visualisation and analysis." Bioinformatics 32.2 (2016): 309-311.
11. Wapman, K., and Daniel Larremore. "webweb: a tool for creating, displaying, and sharing interactive network visualizations on the web." Journal of Open Source Software 4.40 (2019).
12. Stevens, Harry. "Why outbreaks like coronavirus spread exponentially, and how to "flatten the curve"." *The Washington Post* (2020).