Live Coding Sonic Flocks

Gerard Roma
University of West London
gerard.roma@uwl.ac.uk

ABSTRACT

This paper describes a system for interactively coding the behaviour of flocks of artificial sound-making agents. The system is based on an animated two-dimensional visualization where the background represents acoustic or visual information, and multiple agents make sound by traversing the space. The main interface is a domain-specific language that allows for composing agent behaviours from elementary actions. The language is described through several basic examples. The system is demonstrated through Brunzit, a standalone C++ application that supports real-time music live coding using two different synthesis engines. While the system is still in early development, two musical examples are used to demonstrate its potential.

1 Introduction

Interactive music systems have been traditionally considered as collections of artificial agents (Rowe 2004). In this context, a computer is often used to automate some parts of a music performance, using several algorithms, sometimes mimicking the role of human performers. In this sense, the view of the computer as a support of artificial agents can be seen as a general paradigm for computer-aided music performance. Live coding (Collins et al. 2003) is no exception. In this practice, multiple code snippets typically result in multiple concurrent processes spawned and/or steered by a programmer in real time.

Agent-based models (ABM) are a popular take on multi-agent systems, used in many disciplines to model and study the emergent properties of real-life complex systems. A common application is artificial life (ALife). ABMs often feature 2D visualizations that help in understanding the process or system being modelled. While the application of multi-agent systems is quite natural for computer-aided music performance, the use of 2D visualizations is not very widespread. Two- and three-dimensional spaces are ubiquitous in graphics-heavy interactive applications such as video games, but there is no obvious universal mapping to common music practices. Yet there are a few traditions, such as wave terrain synthesis (Borgonovo and Haus 1986) or concatenative synthesis (Schwarz et al. 2006) that take advantage of 2D spaces. Previous work (Roma 2023) has explored agent-based models as a live coding interface for interacting with sonic data terrains, with a focus on coding individual agent behaviours. This paper further develops the idea of agent-based music live coding by exploring live coding of collective agent behaviours. While ABMs often try to model specific real-world systems, some algorithms are widespread. For example, the *Boids* algorithm was proposed in the computer graphics community to simulate bird flocks by Reynolds (1987). This algorithm has been used in interactive music systems (Blackwell 2003; Unemi and Bisig 2004). The present study proposes a domain-specific language for controlling sonic flocks through live coding. The following section quickly reviews the languages used for agent models and related work in interactive music. Section 3 describes the language and general data model. An implementation is described in Section 4, and some examples are demonstrated in Section 5. Sections 6 and 7 discuss the main contribution and future work.

2 Related Work

2.1 Languages for agent-based modelling

Agent-based models are computational models used to create ALife systems and study real-world complex phenomena. Early developments, such as Sugarscape (Epstein 1996) were centered around specific models. More flexible software platforms then evolved to support the development of different models. Many classic open source platforms and models were reviewed by Standish (2008). While several platforms have been developed in conventional object-oriented languages such as C++ or Java, Netlogo (Tisue and Wilensky 2004) continued the tradition of the Lisp-based Logo language, with a focus on education. This focus may have led some to regard it as a toy language. However, a formal comparison by Railsback, Lytinen, and Jackson (2006) highlighted NetLogo's ease of use. This platform offers some capabilities in the direction of live coding: as an interpreted language, it can be programmed interactively using a REPL-style text field known as the "observer". While NetLogo is based on visualization and offers some limited music functionality, it is generally not designed for the arts, so its capabilities are too limited for music or graphics live coding.

2.2 Music With Particle Systems

Boids (Reynolds 1987) is a popular algorithm that appeared in computer graphics, where particle systems are commonly used for different applications. The model is based on a few simple rules (separation, alignment, cohesion) that allow simulating the flocking behaviour of birds and insects. Beyond modelling real-world systems, Boids is often used to experiment with ALife in artistic practice. Swarm Music (Blackwell 2003) is an early example. This system implemented the Boids algorithm in a particle animation, which was then interpreted to generate MIDI messages to a software instrument. A later system, Swarm Granulator (Blackwell and Young 2004) used granular synthesis as a more natural mapping to sound. Both systems were autonomous but interacted with human musicians. Following Swarm Granulator, the proposed system implements granular synthesis as a prominent sound mapping of flocks, but using live coding to create and control multiple flocks.

A more recent example is Tölvera (Armitage, Shepardson, and Magnusson 2024), a Python library for composing with ALife. The system allows composing different agent behaviours and offers an Open Sound Control (OSC) interface. Tölvera has been used in several performances, including interaction with live coding systems. Like Tölvera, the proposed system allows composing agent behaviours but adds the ability to do so in real time through a live coding interface. Also, while Tölvera focuses mostly on larger-scale behaviours and models, the proposed system allows fine-grained control of individual particles as well as larger swarms.

Many ALife systems are mostly focused on the behaviours of agents. In addition to agent behaviours, the system proposed in this paper also allows defining a terrain, in the tradition of graphical ABMs. A terrain is a dataset that is associated and visualized in the 2D space navigated by the agents. In ABMs, the terrain can be used to modify the behaviour of the agents. In the proposed system, the terrain is used more specifically as a source of data for sonification.

3 Language

Previous work has shown the potential of flocking algorithms for improvised interactive music. Since complex behaviour emerges from the rules programmed into the agents, they are perceived to have a life of their own, while still reacting to input from the real world. However, in previous work using flocking and ALife algorithms, the behaviour of agents needs to be programmed beforehand. This paper proposes a system for live coding agent behaviours in real time. In the spirit of music live coding, this allows improvising algorithms that generate music, in this case with the support of an ALife model.

The system is based on an animation that is updated periodically with a given frame rate. Agents are grouped into flocks (flocks may contain a single agent) which navigate a data terrain. Data from the terrain is used to produce sound using different synthesis engines. Each agent is mapped to a single voice. Agents are controlled by *behaviours*. A behaviour contains a list of actions that are executed sequentially for all agents in the flock at each frame.

A custom domain-specific language is used to create behaviours and attach them to agents. The language is loosely inspired by Logo, but at the moment has a very simple syntax. In particular, procedures are not supported, so code is basically used to compose calls to existing functions. The general syntax follows this form:

```
name: action p1 p2 p3, action p1 p2 p3, action p1 p2 p3
```

This is designed to fit in a single long text field, so a single statement is executed at a time. Here, *name* is a user-defined identifier, *action* is the name of an existing action available in the system, and *p1...pn* are parameters of the action. A global agent named *world* is used to create other agents and configure the environment. A number of actions are supported, which can be grouped into world actions, agent actions and flocking actions. Since identifiers are assigned to flocks, both agent and flocking actions are executed on flocks. Table 1 shows currently implemented actions.

Table 1: Supported actions.

Name	Type	Meaning
background	world	change background colour
map	world	set up the terrain
make	world	create a new flock or agent
go	agent	advance in current direction
stop	agent	keep direction but stop moving
turn	agent	change direction (relative angle)
up	agent	steer to absolute angle (up)
down	agent	steer to absolute angle (down)
left	agent	steer to absolute angle (left)
right	agent	steer to absolute angle (right)
seek	agent	steer to specific point
wander	agent	change to random direction
die	agent	leave the terrain and disappear
volume	flock	set the sound volume of the flock
join	flock	steer towards group (boids cohesion)
avoid	flock	steer away from group (boids separation)
align	flock	steer towards group (boids alignment)

Combining individual behaviours with collective behaviours in a live coding environment involved several design compromises. First, the use of a data terrain commonly leads to a learning process where the performer learns to predict the different sounds found in different locations (Roma 2023). In this context, it is useful to be able to direct agents to specific zones. However, flocking algorithms critically rely on relative geometry, also known as *turtle geometry* (Abelson and DiSessa 1986), where the agent is controlled by a heading direction, which is changed algorithmically. A compromise was found in implementing the model using turtle geometry, but with helper functions that relate to absolute coordinates (*up, down, left, right, seek*). It is also worth noting that the terrain is typically continuous, i.e. agents reaching the right border will appear at the left border. It is left to the user to design the data terrain to avoid discontinuities or embrace the ones appearing when agents cross the edges.

A second issue relates to time. While agent behaviours are evaluated for each frame, some actions only make sense when executed once. Generally, some degree of control over the frequency of execution is necessary. This was implemented through a *frequency* parameter which is available for most actions. Possible values are: *once, sometimes, often, always*, where *sometimes* and *often* are defined in terms of fixed frequencies, *once* executes the action at the next frame only, and *always* executes the action for every frame as long as the behaviour is active. Crucially, actions have appropriate default frequencies, so the parameter is not needed in many cases.

A similar problem appears from the way flocking behaviours are typically implemented. As these behaviours depend on the distance between agents, they are typically modelled as forces, with force being proportional to acceleration. In contrast, individual behaviours modify a velocity vector. Different behaviours, composed together by the user and with different frequencies, can contribute to the decision of where the agent is heading. In addition, it is also useful in some cases to have agents that do not advance. As a result, most actions are used only to compute the velocity vector, which indicates the direction of the agent. The only action that modifies the position is *go* (equivalent to *forward* in Logo). This action also applies a final multiplier that determines the speed of travel, given the velocity computed from all the previous actions. This means that most of the time, the statements need to explicitly conclude with a *go* action.

Several examples of the language are shown in Figures 1-6. Here, the implementation described in Section 4 was used at a small resolution with a white background and the input text box hidden for demonstration purposes.

After setting the background colour, a set of agents is created (Figure 1):

```
world: make agents 50 arrowhead black
```

This creates 50 agents, using the *arrowhead* Unicode character in black (some names are provided in the language, but the Unicode character can also be used directly as text), and assigns the flock the name "agents", which is used in subsequent commands. Agents are then instructed to seek the center of the screen:

```
agents: seek 400 300, go
```

Figure 2 shows the initial result, with agents moving towards the centre. The process converges with all the agents at the centre (Figure 3). After this, they are sent to travel right, but with some randomness (Figure 4):

```
agents: right often, wander 0.8 sometimes, go
```

In this case, the *wander* function causes some of the agents (randomly with probability 0.8) to diverge *sometimes*, but then *often* turning right again, which causes the flock to navigate towards the right while expanding. The next command implements traditional *boids* flocking behaviour:

```
agents: join 20 0.1, align 30 0.5, avoid 10 0.7, go
```

Each of the flocking actions (*join, align, avoid*) has a *radius* parameter, which determines which neighbours will be taken into account, and a *force* parameter, which can be used to relatively balance the effect of each action. The result is a more cohesive flock (Figure 5). Finally, Figure 6 shows a less predictable variation:

```
agents: join 20 0.5, align 10 0.1, avoid 10 0.5, turn 1 often, go
```

Here, the agents turn by one degree *often*, which in interaction with the flocking actions makes them draw arcs but also disperse and in some cases form smaller flocks.

4 Implementation

The proposed language has been implemented into *Brunzit*, a multi-platform desktop application using the Cinder C++ library [1]. The application presents a text field used for executing one statement of the described language at a time. The rest of the screen shows the data terrain and agent animations.

The user may create any number of flocks (which may be individual agents) to navigate the terrain. As seen in Section 3, agents are rendered as Unicode icons, which allows defining the icon and colour from the language, with some icons available as keywords (e.g., triangle, square, circle, arrow, arrowhead, smiley). Colours are specified as strings from the names in the SVG standard as available in Cinder [2]

For the sonification, two synthesis engines are implemented at the time of this writing, which are associated with different types of terrain.

An additive synthesis engine is implemented for image terrains. Additive synthesis works by adding multiple sine waves at different frequencies and amplitudes. In *Brunzit*, if the user chooses an image file with the *map* action, the image (converted to grey scale) is used to provide a value for each pixel. Each agent is associated with a sine oscillator where the frequency is determined by the current pixel value.

A granulation engine is used with audio terrains. Granular synthesis works by combining (often overlapping) small segments of sound. In granulation, the segments come from an existing audio sample. In this case, following previous work (Roma 2023), if the user specifies a sound file in the *map* command, the file is split into segments, and the resulting corpus is analyzed using the FluCoMa library (Tremblay, Roma, and Green 2021). The framework proposed in (Roma et al. 2021) is used to map the segments into a 2D grid, where the amplitude envelope of each segment is visualized. Each agent emits grains with a Hann window envelope. The actual grains are smaller than the sound segments so that each grain can start at a random position. Compared to the SuperCollider implementation in (Roma 2023), using C++ allows scaling to larger corpora and numbers of agents. However, the sound capabilities are obviously much more limited.

5 Examples

Brunzit is still under development and has not yet been used in live performance. Regardless, initial experiments with the system showed a great potential for interactive sonic exploration. This section describes two examples. The code and links to videos can be found in the github repository.[3]

[1] https://libcinder.org/
[2] https://en.wikipedia.org/wiki/Web_colors#X11_color_names
[3] https://github.com/g-roma/Brunzit

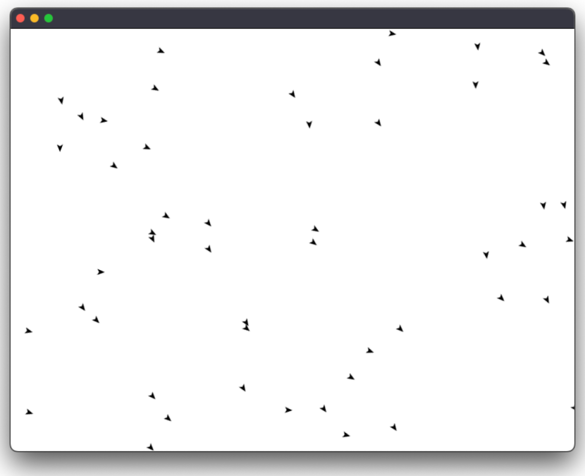


Figure 1: 50 agents are created

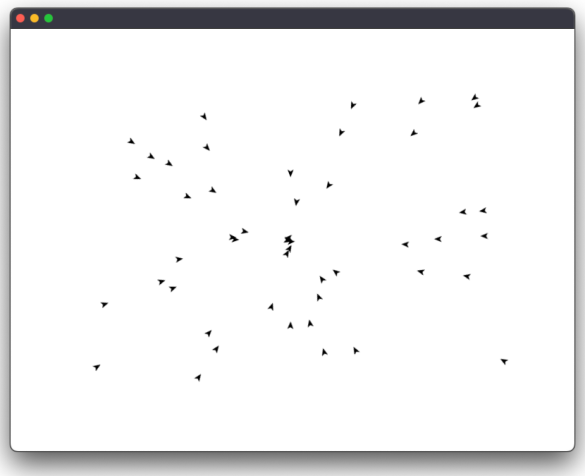


Figure 2: Steer towards centre using *seek*

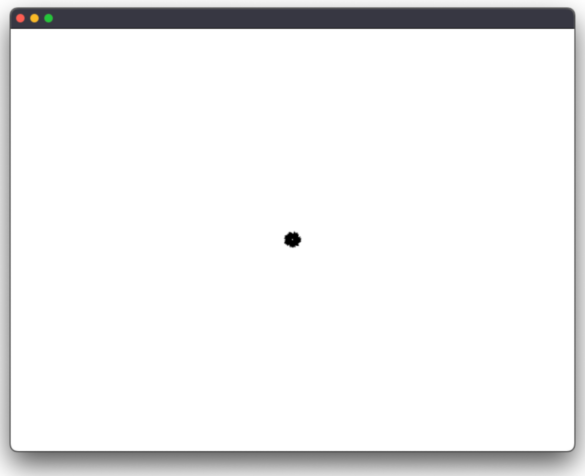


Figure 3: All agents at the centre (after some time)

Figure 4: Move right with some *wander* behaviour

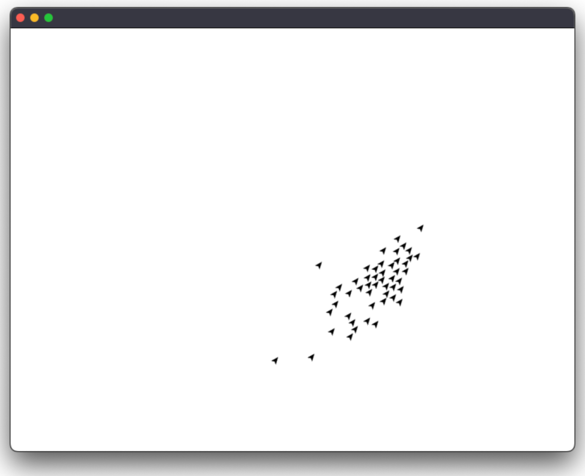


Figure 5: Using composed boids (*join, align, avoid*) algorithm

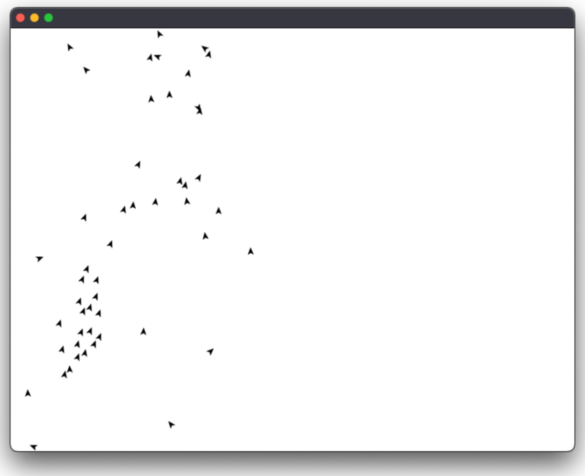
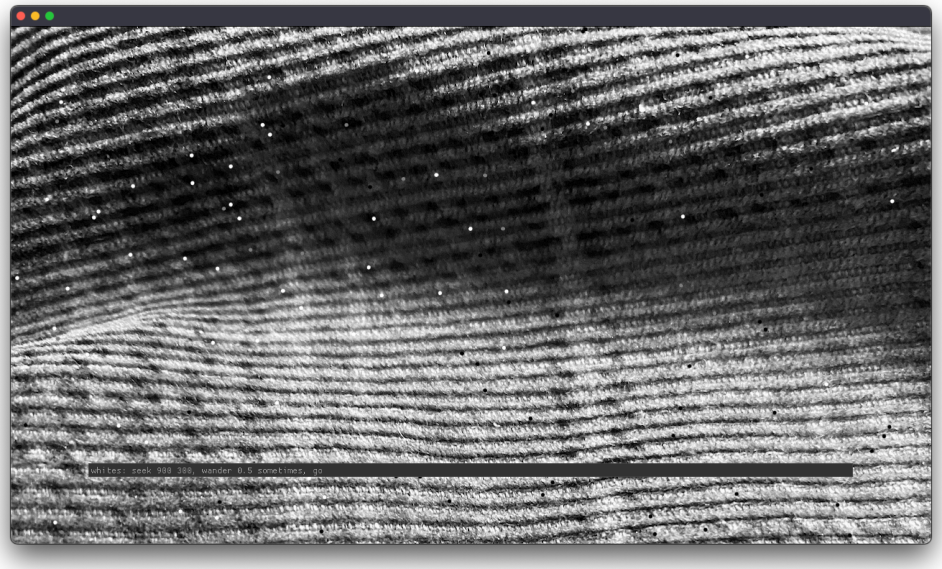


Figure 6: Adding *turn* behaviour to boids



```
whites: seek 900 300, wander 0.5 sometimes, go
```

Figure 7: *Example of the Brunzit interface using a picture as terrain.*

5.1 Chasing Sweet Spots

The first example was created with a picture of patterned fabric as terrain, using the additive sound engine (Figure 7). The stripes in the image created quick changes in the frequencies of the agents. An interesting result was obtained by sending three flocks towards specific points using *seek*. This causes the flocks to concentrate around one point, producing a cluster of sound, sometimes alternating between two frequencies. Stopping some times freezes all modulation producing an inharmonic drone. The flock is then sent to another spot, combining *seek* with flocking actions.

5.2 Death by Flocking

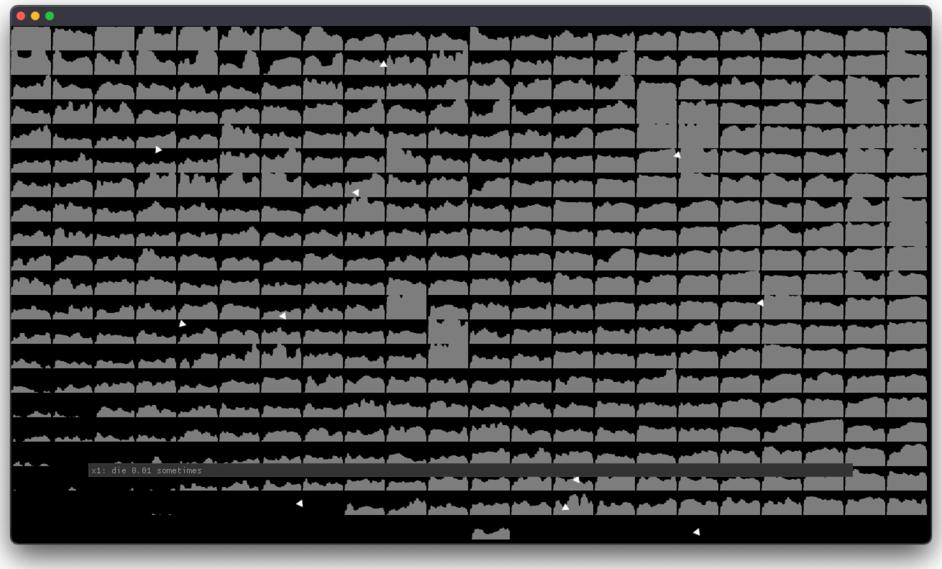
The second example is based on a sound file, in this case a field recording containing bell sounds and traffic noise from Freesound[4]. The interface is shown in Figure 8. The sound was chosen to be used with the granulation engine as it contained an interesting variety of sounds. For this case, it was found that sending small flocks of agents to wander the terrain randomly was better than using many agents. The resulting textures were then more nuanced as the agents consumed different sounds. The flocking behaviour was then used to fade out, letting the agents slowly die during the process. Repeating the same process multiple times could be used as a musical structure, leading to different sonic results each time, but with all iterations sharing a common form.

6 Discussion

Artificial life, particle systems and flocking algorithms are commonly used in computer graphics applications and have also been used for interactive music systems. The system described in this paper introduces the ability to improvise compositions of agent behaviours in real-time. By combining the agent models with data terrains, a great variety of complex sonic behaviours and textures can be obtained, and the animations enhance the understanding of the algorithmic sonic generation process.

While the use of ALife animations often evokes the intelligence of different life forms beyond humans (Armitage, Shepardson, and Magnusson 2024), there is clearly a strong human component in the design and composition of the behaviours. Live coding emphasizes this element by allowing the performer to visually demonstrate their thinking process. As commonly seen when improvising with AI systems, Using *Brunzit* involves finding the right balance between trying to control and steer the performance, and embracing the unexpected behaviours emerging from the interaction rules.

[4]https://freesound.org/people/bruno.auzet/sounds/530590/



```
x1: die 0.01 sometimes
```

Figure 8: *Example of the Brunzit interface using a sound file as terrain.*

7 Conclusions and Future Work

This paper has described a system for composing behaviours of sonic agent models using live coding. The user interface is based on a domain-specific language that allows combining multiple actions into behaviours, with different event frequencies.

The system is in early development and several features are planned. On one hand, the language is still very limited and offers no mechanisms for abstraction. Following the Logo tradition, the ability to create new procedures should be added so that interesting combinations of actions can become new actions.

Another area of improvement concerns the sound engines and musical capabilities. While initial development has been directed to a mostly continuous sound aesthetic that goes well with the flow of agents, adding more sound-related actions will allow playing rhythmic patterns, envelopes and other more traditional musical structures.

Finally, while visual animations are currently used mostly as a tool, more work on the graphics would allow algorithmic design of data terrains, and ultimately a better balance between sound and visuals in flock live coding.

References

Abelson, Harold, and Andrea DiSessa. 1986. *Turtle Geometry: The Computer as a Medium for Exploring Mathematics.* MIT press.

Armitage, Jack, Victor Shepardson, and Thor Magnusson. 2024. "Tölvera: Composing With Basal Agencies." In *Proc. New Interfaces for Musical Expression*. Utrecht, NL.

Blackwell, Tim. 2003. "Swarm Music: Improvised Music with Multi-Swarms." *Artificial Intelligence and the Simulation of Behaviour, University of Wales* 10: 142–58.

Blackwell, Tim, and Michael Young. 2004. "Swarm Granulator." In *Workshops on Applications of Evolutionary Computation*, 399–408. Springer.

Borgonovo, Aldo, and Goffredo Haus. 1986. "Sound Synthesis by Means of Two-Variable Functions: Experimental Criteria and Results." *Computer Music Journal* 10 (3): 57–71.

Collins, Nick, Alex McLean, Julian Rohrhuber, and Adrian Ward. 2003. "Live Coding in Laptop Performance." *Organised Sound* 8 (3): 321–30.

Epstein, Joshua M. 1996. *Growing Artificial Societies: Social Science from the Bottom up.* The Brookings Institution Press.

Railsback, Steven F, Steven L Lytinen, and Stephen K Jackson. 2006. "Agent-Based Simulation Platforms: Review and Development Recommendations." *Simulation* 82 (9): 609–23.

Reynolds, Craig W. 1987. "Flocks, Herds and Schools: A Distributed Behavioral Model." In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, 25–34.

Roma, Gerard. 2023. "Agent-Based Music Live Coding: Sonic Adventures in 2D." *Organised Sound* 28 (2): 231–40.

Roma, Gerard, Anna Xambó, Owen Green, and Pierre Alexandre Tremblay. 2021. "A General Framework for Visualization of Sound Collections in Musical Interfaces." *Applied Sciences* 11 (24): 11926.

Rowe, Robert. 2004. *Machine Musicianship.* MIT press.

Schwarz, Diemo, Grégory Beller, Bruno Verbrugghe, and Sam Britton. 2006. "Real-Time Corpus-Based Concatenative Synthesis with Catart." In *9th International Conference on Digital Audio Effects (DAFx)*, 279–82.

Standish, Russell K. 2008. "Open Source Agent-Based Modeling Frameworks." *Computational Intelligence: A Compendium*, 409–37.

Tisue, Seth, and Uri Wilensky. 2004. "Netlogo: A Simple Environment for Modeling Complexity." In *International Conference on Complex Systems*, 21:16–21. Citeseer.

Tremblay, Pierre Alexandre, Gerard Roma, and Owen Green. 2021. "Enabling Programmatic Data Mining as Musicking: The Fluid Corpus Manipulation Toolkit." *Computer Music Journal* 45 (2): 9–23.

Unemi, Tatsuo, and Daniel Bisig. 2004. "Playing Music by Conducting BOID Agents-a Style of Interaction in the Life with a-Life." *Proceedings of A-Life IX*, 546–50.