Agent-Based Music Live Coding: Sonic adventures in 2D

GERARD ROMA

School of Computing and Engineering, University of West London, London, UK
Email: gerard.roma@uwl.ac.uk

This article describes agent-based music live coding, an approach for music performance and composition based on programming a set of agents in a 2D plane. This style of programming draws from the tradition of agent-based models and facilitates interactive algorithmic control of data-driven sound synthesis methods such as wave terrain synthesis or corpus-based concatenative synthesis. The main elements are a 'terrain', which may be used to access different types of data, a set of agents and their trajectories, and a set of synthesis functions associated to agents. An implementation using the SuperCollider language is demonstrated.

1. INTRODUCTION

Live coding (Collins, McLean, Rohrhuber, and Ward 2003) is nowadays an established practice for laptop performance, as well as a fertile ground for experimentation with different ways to express and control music processes. A common issue, both for live coders and audiences, is understanding the state of the ongoing processes as a result of executing different lines of code. This has inspired research into different ways to visualise, or to enhance the visualisation, of the code (McLean, Griffiths, Collins and Wiggins 2010). Beyond visualisation of code itself, one approach is to use an intermediate model with a straightforward graphical representation, which can be used to control music processes.

Agent-based modelling (ABM) often uses visualisations to facilitate understanding of complex phenomena. This model is well suited for music, as it is common to create music using ensembles of computational processes and autonomous agents.

This paper explores agent-based music live coding (ABMLC) as an application of ABM to music live coding for performance and composition. Beyond the benefit of the visualisation as an explanatory complement to the sound and code, using these models allows to quickly create complex algorithmic musical structures and textures by aggregation of simple behaviours.

2. BACKGROUND

ABMLC stems from different traditions in music and graphics programming. This section provides an overview of related traditions that inform the practice of coding spatial sonic trajectories.

2.1. Vector and wave terrain synthesis

The idea of using two dimensions to control sound synthesis can be traced back to the use of joysticks in early synthesisers, such as the EMS VCS 3. The joystick was particularly instrumental to vector synthesis, as shown in the Prophet VS synthesiser (released in 1986). Vector synthesis extended wavetable synthesis by allowing the control of the interpolation of multiple wavetables using a 2D plane. Like any synthesis parameters, vector synthesis trajectories are typically amenable to automation.

It can be argued, however, that vector synthesis made limited use of the 2D space. In this sense, linear crossfading between wavetables could be seen as a nice addition, rather than a significant departure from wavetable synthesis. Wave terrain synthesis (Mitsuhashi 1982) extended the idea of a one-dimensional wavetable to two dimensions. While wavetable synthesis can be used with waveforms inspired by physical instruments, wave terrain synthesis is a more abstract technique, as there is no direct interpretation of the terrain in terms of physical sound production. The technique was later extended by introducing several functions that could be used to define 2D trajectories (Borgonovo and Haus 1986). While both vector synthesis and wave terrain synthesis can now be seen as classic forms of digital sound synthesis, the 2D control associated with them has continued to evolve. For example, the Yamaha SY22 synthesiser applied the concept of vector synthesis to FM synthesis, and many hardware synthesisers have continued to offer joystick control. With respect to wave terrain synthesis, the idea of traversing a data terrain was extended to different synthesis techniques (James and Hope 2011), mostly using input devices to control the traversal. In this article, a similar framework is proposed with respect to the generality of the terrain generation, but with a focus on multiple agents and live coding.

2.2. Feature spaces

Two- and three-dimensional spaces have been used in psychoacoustics to analyse the perception of musical instrument sounds (Caclin, McAdams, Smith and Winsberg 2005). Spectral analysis of audio is used to obtain multidimensional representations of audio data that relate to how it is perceived, but for interactive applications, they are often reduced to two dimensions. Options for interacting with sound databases are either selecting relevant descriptors or using dimensionality reduction algorithms. Corpus-based concatenative synthesis systems such as CataRT (Schwarz, Beller, Verbrugghe and Britton 2006) have traditionally been based on 2D scatterplots using scalar descriptors as axes. Several systems have explored dimensionality reduction and layout mapping for descriptor-based visualisation of sound collections in two dimensions (see Roma, Xambó, Green and Tremblay 2021, and references therein). These systems have generally been controlled through input devices such as mice, tablets, or other sensors. Some systems (Garber, Ciccola, and Amusategui 2020; Roma et al. 2021) allow recording trajectories that are re-played in a loop. Visualisations of sound collections can be seen as a way of creating a terrain for live-coded sound-generating agents. In this case, the agent can be used to control a synthesis algorithm that plays or processes the sample assigned to the current position. This idea has been explored in previous work using a multiagent system with CataRT (Eigenfeldt and Pasquier 2011).

2.3. Agent-based models

Programming paths in 2D spaces can be traced back to the Logo language (Solomon et al. 2020). Logo was conceived as a programing language for learning and famously allowed experimentation with computer graphics by moving an agent around the screen (also known as *turtle graphics*). Beyond its general influence in computer graphics and education, Logo was the inspiration for ABM software such as StarLogo (Resnick 1996) and Netlogo (Tisue and Wilensky 2004). ABM allows the understanding of emergent phenomena by defining local rules of multiple agents, an idea that can be traced back to cellular automata (CA). Agent-based models and CA are fundamental tools in the study of artificial life (A-life). These techniques have been extensively used in music composition (Miranda and Todd 2003). Experiments with CA and A-life can actually be linked to the popularisation of LED button grids in computer music. An early example is Toshio Iwai's 'Music Insects' (1992), where a group of animated agents played notes in reaction to paths of dots painted by the user. Iwai was later involved in the development of Tenori-on (Nishibori and Iwai 2006), a commercial musical instrument based on a grid of illuminated buttons released around the same time as the Monome controller (Dunne 2007). Developed around the same time, Dave Griffiths's Al-Jazari (Griffiths 2008) allowed a user to make music by controlling a set of robots in terrain made of cubes that triggered sounds. The robots were controlled by coding in a basic visual language using a gamepad.[1]

General-purpose ABM software has also been used for music. Netlogo includes some music capabilities, although very limited. NetMusic (Anderson and Anderson 2020) is a Netlogo-based system for education using traditional music concepts. Osc-netlogo (Cadiz and Colasso 2012) is an Open Sound Control library for NetLogo that allows controlling sound generators using this language.[2]

2.4. Live coding

Many live coding languages and environments have emerged during the last few years (All things live coding 2022). Music live coding approaches could be classified into 'synthesis-oriented' and 'event-oriented' (Roma 2016). Synthesis-oriented music live coding leverages the potential for rapid creation of real-time audio signal graphs offered by languages such as SuperCollider. Event-oriented live coding focuses on generating events, often triggering samples or controlling pre-defined synthesisers. Both approaches are sometimes combined. The approach proposed in this article may qualify as event-oriented, but unlike many event-oriented languages, it does not focus on rhythmic patterns or musical notes (although it is possible to generate these kinds of events). In event-oriented live coding, a control process is typically associated with a sound-generation process (e.g., a sampler or a synthesiser). This combination can be seen as an agent. The idea of artificial agents as a metaphor for music live coding processes was used prominently in ixi lang (Magnusson 2011).

In this article, agents are defined as visual objects moving on a 2D surface, where the events are defined by the positions and the objects found in the environment. Agents are associated with a synthesis function created in advance. In addition to agent modelling and music live coding, an inspiring system is tixy.land (Kleppe 2022) a CA-like live coding environment where the size and colour of a dot in a matrix can be defined as a function of its position, index and time.

[1]Thus, Al-Jazari could in fact be considered the first implementation of ABMLC.

[2]The system presented in this article was inspired by early experiments with osc-netlogo by the author and colleagues at the Barcelona SuperCollider user group.

3. AGENT-BASED MUSIC LIVE CODING

Agent-based models have found widespread use many domains, notably in social sciences. As outlined in section 2.3, a particularly interesting lineage evolved from the Logo language in the context of education research. The idea of a parallel Logo, with improved agent senses and the addition of 'patches' as a way to represent information in the environment, was described as a conceptual model, implemented in StarLogo, for creating 'explorations of microworlds', often loosely based on real-world phenomena such as ant colonies or traffic jams (Resnick 1994).

The view of a real-time computational process as an autonomous agent is ubiquitous in computer music. When computers with real-time capabilities started being available, the idea of interactive music systems as a dialogue between a human and an ensemble of agents was developed by Rowe (1992).

Live coding systems are also easily described in terms of agents: live coders execute short code snippets that result in an ensemble of real-time processes, each typically assigned to a variable. This is usually implemented in the form of domain-specific languages and libraries that allow the programmer to focus on a limited number of abstractions.

In this context, and by analogy to agent-oriented programming as a specialisation of object-oriented programming (Shoham 1993), ABMLC can be described as a *live coding paradigm*: just as some languages and environments focus specifically on note patterns or on signal-processing graphs, in ABMLC the focus is on coding individual agents moving in an abstract 2D space. This model has several implications. On the one hand, focusing on the visualisation means that there are two prominent parameters (associated with the two dimensions) controlled by each agent. On the other, while agents can be thought to possess some degree of intelligence, the emphasis tends to be on collective behaviour, and thus complex results can be obtained by aggregation of simple behaviours.

This section describes a framework for ABMLC inspired by the StarLogo and NetLogo conceptual model. An implementation using the SuperCollider language is presented in section 5. Instead of coding complete programs, here the goal is to support rapid coding of agent behaviours. With this aim, many important features of ABMs, such as interactions between agents, or the ability to spawn and destroy agents, are left for future work.

An overview diagram of the model is shown in Figure 1. An agent is defined by a code snippet that modifies its position (from now on, the 'agent function'). The agent function is called repeatedly at a certain rate. An agent lives in a 'terrain', a grid of cells that may contain different kinds of information. The agent has the same size as one of the cells. Each agent is associated with a sound processor, and thus both its function and the sound processor can make use of the information in the cell currently occupied. Agents may also get information about other agents.

3.1. Generating the terrain

In a way, the idea of using an agent model for sound production can be seen as a metaphor that evokes physical sound production by human agents. The terrain can be seen as a shared instrument, or a pre-composed material, which can be used to influence the sound generation of each agent in different ways. The fundamental aspect of the terrain is that it is common to all agents. The terrain may also be static (such as in, e.g., wave terrain synthesis), although in many agent-based models it can also be modified by agents, or even evolve on its own.

Given that it is a digital representation, the terrain will necessarily be a discrete matrix (from here on $P \in \mathcal{R}^{N \times M}$ for N rows and M columns). However, different applications, ranging from wave terrain or corpus-based synthesis to the generation of musical events, can be implemented depending on the resolution and density. Following ABM, the elements of P are called 'patches'. An agent is always located on a specific patch. The elements of P may be encoded in different data types depending on the application. At the same time, the data in P can be used to visualise the terrain in user interfaces. The different underlying data types can be used to survey some uses of the terrain.

The most basic use would be to represent P as a matrix of booleans, which can be represented as a bitmap. This can be used to trigger events in the synthesiser associated with each agent, that is, while traversing the terrain, the agent would start a new sound when running over a patch with a positive value. Thus, in this case, the proportion of positive patches would be associated with the density of sound events.

More generally, P can be a matrix of floating-point numbers (Figure 2). This can of course include zero and thus it can also be used to trigger sound events with additional information (e.g., amplitude). This kind of terrain can be used to implement wave terrain synthesis.[3] A floating-point matrix can be understood as a 3D volume, which can be visualised in 2D as a grayscale map. So, in general, the patch value can be used as a third parameter for the synthesiser in addition to the horizontal and vertical positions.

[3]It is worth noting that, for wave terrain synthesis, the rate of the traversal of the terrain usually needs to be much faster than the control signal generated by the agent-based model, so a separate mechanism needs to be implemented in the synthesis function.

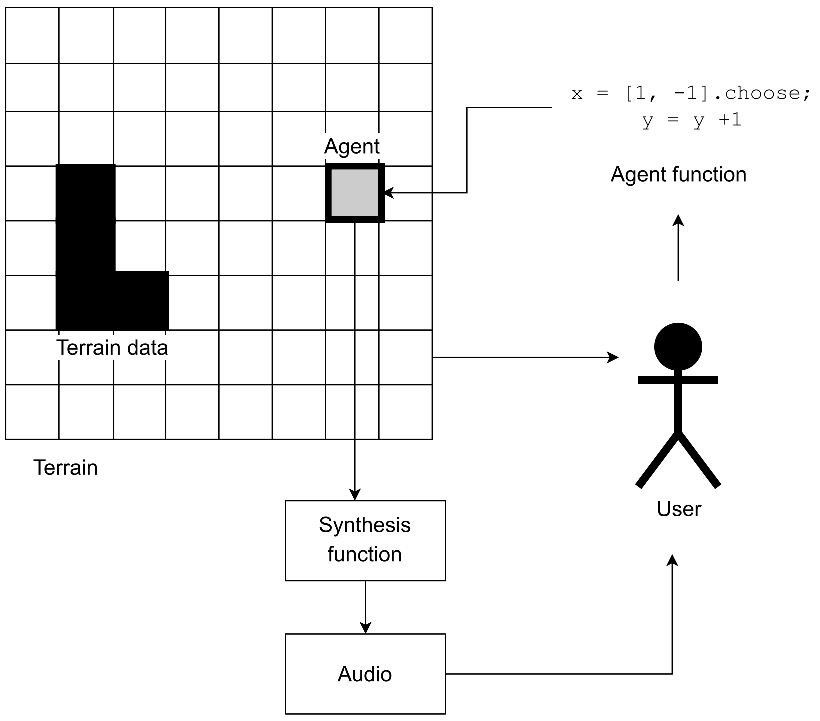


Figure 1. Agent-based music live coding overview.

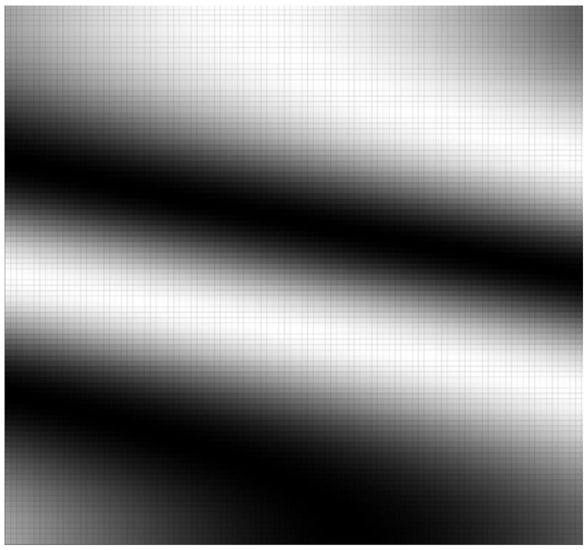


Figure 2. Grayscale terrain generated from a function.

Finally, the elements of P can be encoded as integer numbers. Integers can generally be used to index data collections or discrete scales. For example, integers can point to audio samples in a database. In this case, a terrain can be used as an interface for a potentially large ($N \times M$) audio database. This allows exploring a rich palette of sounds, in the spirit of corpus-based synthesis, but using live-coded agents instead of input devices or pre-defined target sequences. If patches are arbitrarily mapped to any sample in the collection, it would be difficult to make sense of the trajectory of an agent. This can still work for small collections; for example, a drum machine can be implemented in a small grid, where agent behaviours are coded to represent patterns. Agent functions would often require knowledge of the sound associated with each patch, so scaling to larger databases would be challenging. For large databases, the mapping of samples to the terrain should follow some logic. This can be based on audio descriptors, either directly (e.g., as in Schwarz et al. 2006) or using dimensionality reduction (as in Roma et al. 2021). Audio descriptors can also be used to visually represent each of the patches. An example is shown in Figure 3, where the loudness of each sample is displayed as a waveform, and the colour (in this case grey level) represents the average spectral centroid. Another common use of integer terrain matrices would be indexing discrete scales, such as pitch classes or other discrete parameters. A simple visualisation for integer-based terrains can be implemented by using a discrete colour palette.

3.2. Agent trajectories

Agent trajectories are the main representation of the proposed model. Even with no terrain, an interface for controlling a small number of agents through 2D trajectories can be a productive musical instrument. In

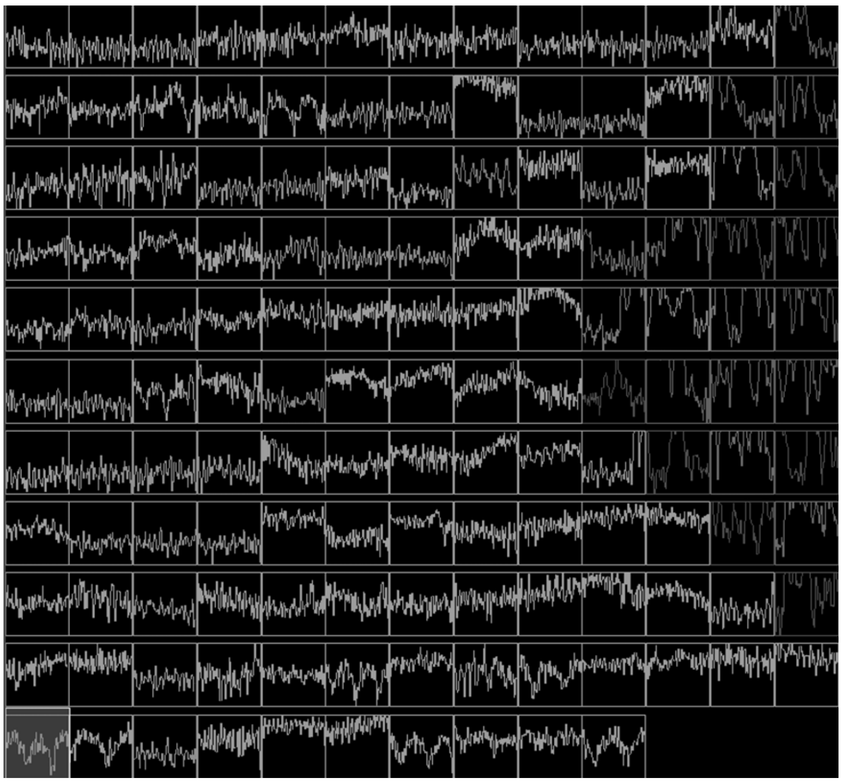


Figure 3. Terrain generated from a database of audio samples.

the proposed framework, the functions for specifying the trajectories are the user interface. Formally, the agent function can be seen as a 2D recurrence relation, which returns the coordinates for the next step given the current step:

$$\begin{bmatrix} x_{t+1} \\ y_{t+1} \end{bmatrix} = f(x_t, y_t, t, P, A)$$

where x_t, y_t are the coordinates of the agent at the current time and x_{t+1}, y_{t+1} are the coordinates at the next time, t is the time step (which can be simply a counter), P is the terrain matrix, and A is the list of agents. This model generally allows accessing any agent and point of the terrain, although in many cases it may be preferable to access only agents and patches in the neighbourhood of x_t, y_t. Each agent can be characterised by its position, and $P[[x_t],[y_t]]$ (where $[x]$ represents the rounded version of x) is the patch at the current agent position. In this article, we are mostly concerned with basic trajectories defined by simple mathematical functions. Some examples are described in section 4. Complex behaviours such as flocking typically require significant amounts of code and are less amenable to live coding, although they could be provided as functions.

The trajectory function is executed at each time step following some rate r. The rate can be user-defined, but it corresponds to the speed at which the visualisation is updated. Thus, common rates could be typically in the range of video frame rates; for example, between 10 and 60 frames per second. Obviously, the rate has a direct effect on the speed of the agents, depending on the code used.

One important aspect of the trajectory function is how to deal with border effects. Depending on the terrain and synthesis method, wrapping around either axis when the border is surpassed may be acceptable or not. It may be desirable to use some convenience functions, or a configuration parameter, for dealing with different options.

3.3. Sound synthesis

In addition to the trajectory function, the agent produces sound through a synthesis function. While the rate of the trajectory function can be controlled by the user, the rate of the synthesis function needs to work for real-time audio generation, typically either sample or block-based. Thus, in terms of computer music systems, the agent trajectory can be seen as a control rate, while the synthesis function runs at audio rate. In practice, the synthesis function can be any synthesiser (e.g., even hardware), as long as the parameters can be updated periodically. All the scalar parameters of the agent trajectory function, including additional mappings (e.g., the values of P may be mapped to a buffer for sample-based sound generation), can be passed to the synthesis function (typically the current patch is sent, as opposed to the whole of P). While in early experiments by the author the synthesis function was live-coded along with the agent function, reducing the live coding activity to the agent trajectory function makes the system easier to use. Both functions deal with different domains (i.e., x, y position or audio signal graph) which involves switching from one mindset to the other, for each of the agents. Focusing on the trajectory code provides a convenient high-level interface for performance. An interesting variation could be to fix the agent trajectory functions and live code the synthesis function.

4. CODING SONIC TRAJECTORIES

As seen in section 3.2, agent trajectories can be described by recurrence relations, with additional inputs such as the terrain or the list of agents. First-order relations are functions where the next value is generated from the previous value. While higher-order relations (where the next value is computed from a larger number of previous values) could be implemented, a few agents implemented using first-order functions strike a good balance between the simplicity required at the user interface side (functions that are easy to code live) and the potential for complexity. In this sense, thinking in terms of simple mathematical functions is useful for live coding, since the behaviour needs to be coded in a short amount of time. This section describes some elementary examples. The possibilities are obviously unlimited. For example, multiple functions can be combined, and a different function can be used for each axis.

4.1. Straight lines

Straight lines are possibly the most trivial functions, generally defined as:

$$\begin{bmatrix} x_{t+1} \\ y_{t+1} \end{bmatrix} = \begin{bmatrix} x + d_x \\ y + d_y \end{bmatrix},$$

where d_x and d_y control the speed in each axis with respect to r. Agents will reach the end of the terrain, and typically either a modulo operator or a change of sign will be used to avoid losing them. This generally leads to looping behaviour, although some randomness can always be added. Systems based on straight lines will often lead to generative music with loops of different sizes depending on the starting point of the agent and the values of d_x and d_y. While the terrain is discrete, values smaller than 1 are useful to implement slower and smooth trajectories. Also, the interpolation is relevant for controlling the underlying synthesis function.

4.2. Random walks

Random walks can be used to explore the terrain while avoiding repetitive/predictable behaviour. There is of course some predictability in the fact that the terrain is a limited resource. The most basic random walk can be implemented by sampling d_x and/or d_y from a sequence of values $[z_1, z_2]$, such as $[-1, 1]$. Generally, if z_1 is different from z_2, the trajectory will be biased and more predictable. Smaller values encourage exploration of the current location, and larger values encourage wandering.

4.3. Oscillations and orbits

Since agent trajectories operate within a control rate defined by the visual frame rate, they can be naturally thought of as low-frequency oscillators (LFO). Many LFO functions are also trivial to write; for example, a sawtooth oscillator with offset o, amplitude a and period k can be written simply as

$$\begin{bmatrix} x_{t+1} \\ y_{t+1} \end{bmatrix} = \begin{bmatrix} x_t + d_x \\ o + \frac{a}{k} \bmod (t,k) \end{bmatrix}.$$

Similarly, a sine oscillator of the y axis can be written as $y_{t+1} = o + asin(\omega t)$. A 2D orbit can then be written as

$$\begin{bmatrix} x_{t+1} \\ y_{t+1} \end{bmatrix} = \begin{bmatrix} o_x + a_x cos(\omega_x t) \\ o_y + a_y sin(\omega_y t) \end{bmatrix}.$$

Orbits can be used to create loops without wrapping around the borders, depending on the amplitudes and offsets. These can in turn be changed dynamically to create more complex trajectories.

4.4. Sequences

Since the terrain is composed of a finite number of patches, the agent can be seen as a finite-state machine where each patch is a different state. Patches can be assigned specific values (such as, e.g., pitch values or samples). Thus, the agent trajectory can be used to create discrete patterns by specifying sequences of states. For example, the following function would alternate between four arbitrary points at a speed controlled by k:

$$a = (a_1, a_2, a_3, a_4)$$

$$b = (b_1, b_2, b_3, b_4)$$

$$\begin{bmatrix} x_{t+1} \\ y_{t+1} \end{bmatrix} = \begin{bmatrix} a\left[mod\left(\frac{t}{k}, 4\right)\right] \\ b\left[mod\left(\frac{t}{k}, 4\right)\right] \end{bmatrix}.$$

4.5. Interactions

More complex behaviours can be created by creating functions that depend on the value of the patch or neighbouring patches. For example, the value of patches can be used to create walls. Similarly, interactions with other agents could be added. Common ideas easily lead to code that spans more than a few lines so they will typically be implemented as helper functions in advance.

5. IMPLEMENTATION

The conceptual model presented in section 4 can be used to implement ABMLC in any language. In fact, as mentioned in section 3, many live coding systems could be seen to implement some form of ABMLC. The full interface proposed here includes the notion of a 2D surface that agents navigate and the corresponding visualisation. Thus, it can be implemented in languages that allow programming both graphics and sound synthesis, or sound control output.

Mob is a SuperCollider program developed by the author to experiment with the different elements of ABMLC. The code is open source.[4] Figure 4 shows the user interface. The system is designed to be extensible. The general idea is that synthesis functions (i.e., synth definitions in SuperCollider), along with functions attached to different objects are coded in advance, but not hard-coded in the program. The user modifies a script containing synth definition functions and auxiliary functions that is loaded at runtime. During a performance, live coding focuses on the functions that define the trajectories of agents. The program is instantiated from the SuperCollider interpreter. The main parameters of a *Mob* instance are the width and height of the terrain, the frame rate, and optionally a terrain specification.

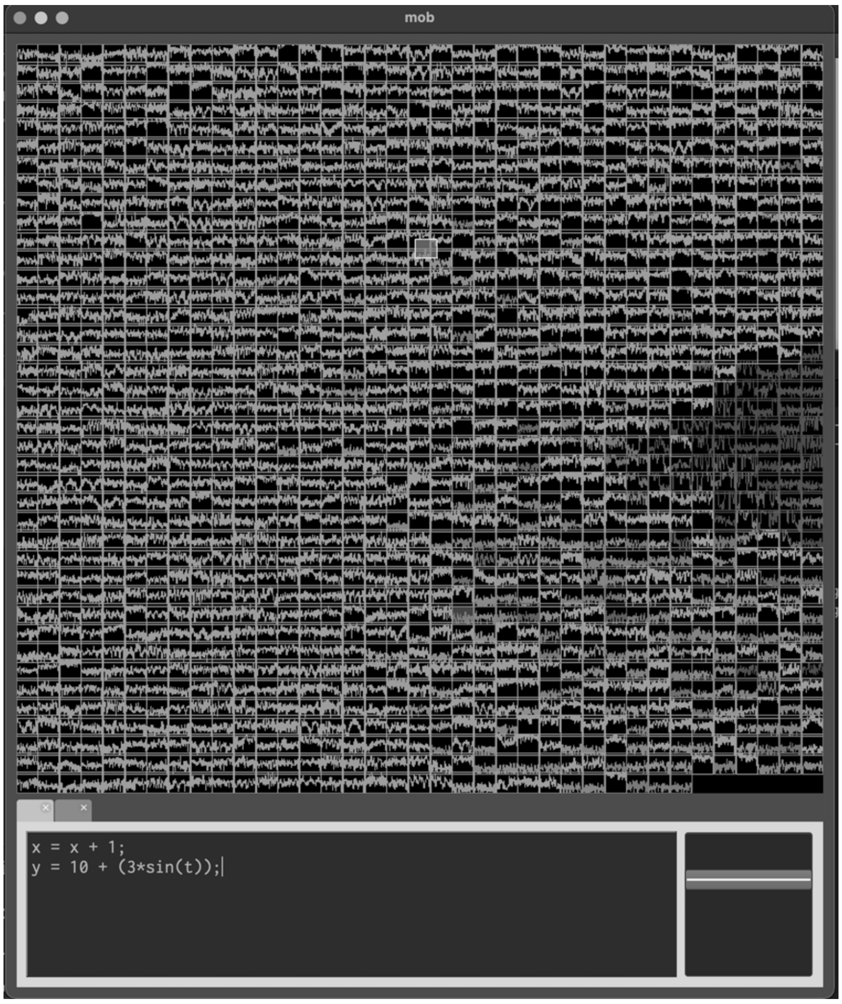
The terrain consists of a grid of patches defined by the user-specified dimensions. The system thus maintains a mapping of the terrain to the screen resolution. The rate defines the speed at which the visualisation is updated (in SuperCollider this works through UserView and the *animate* functionality). At the same time, the live-coded functions are executed at each frame refresh. Thus, faster rates and larger numbers of patches will result in smoother animations and parameter updates, whereas lower values can be used for events and patterns. The timing in the live-coded functions is controlled by the rate parameter.

An extra parameter can be used to provide data for the terrain, specified as a dictionary with 'type' and 'value' keys. There are three types. The first type, 'function', assumes the value is a function of the form $z = f(x, y)$ that will define an intensity value for each patch depending on the x and y coordinates. In this case, the program will execute the function for each position and generate the terrain data and visualisation upon startup. The terrain data are also stored in a buffer, which can be used with the WaveTerrain UGen available in the SLUGens package (Collins 2022). If the type is 'file', the value is assumed to be a text file. The file follows a simple CSV-like format with a custom header declaring the values to be binary, integer or float, as described in section 3.1. This can be used to define arbitrary shapes as the terrain. A third type of terrain is defined as 'samples'. The value is then assumed to be a folder of audio samples. The samples are analyzed using the FluidCorpusMap library (Roma et al. 2021). This library performs analysis and dimensionality reduction of the audio features and maps them to a grid using an assignment algorithm. When either a text file or a samples directory is provided, the width and height parameters are obtained from these, and any user-specified values are overridden.

Agents are represented as a coloured rectangle over one of the patches in the terrain visualisation. It is in theory possible, while usually not desirable, for an agent to be out of the terrain. Each agent is programmed through a code snippet, shown in a tab below the agent-terrain visualisation (Figure 4). Tabs can be created and closed manually, which results in the agent being spawned or removed. The code snippet is compiled using a keyboard shortcut. Before compiling, the code is wrapped in a template code string, which includes all needed variables: x, y, t, p, a (see section 3.2). The code snippet is expected to modify x and y, and, potentially, p. If the code is compiled successfully, the compiled function is attached to the agent and used to determine the next position at each frame. An extra control for volume is provided. This assumes that each agent corresponds to a voice, which may be triggered by objects in the terrain, but can also be playing all the time. The addition of a volume slider allows the performer to have direct control over the mix of the agents. The slider is mapped to an amplitude parameter sent to the synth and can be mapped to a MIDI controller. The x and y position values can then be used to control other parameters of the synth.

While this system allows controlling musical processes with terse code snippets, there are always recurrent issues that can be solved by reusable helper functions. One example is the need to wrap around the terrain. While this is necessary in order to maintain the agent in the terrain and provide an understandable visual representation of the agent model, wrapping around may also mean a jump in the parameters controlled by x or y. Therefore, in some cases, other strategies may be desirable. Another example is dividing the rate, which is necessary for scheduling events. In order to allow for user-defined functions with a compact syntax, functions provided in the

[4]https://github.com/g-roma/Mob.



```
x = x + 1;
y = 10 + (3*sin(t));
```

Figure 4. Mob user interface.

auxiliary script are attached as methods to the agent, the terrain or the time parameters.

In addition to the *x* and *y* parameters, the agent code snippet also needs to define the name of its synth definition. This is a symbol corresponding to the name of one of the synth definitions provided in the auxiliary file. As explained in section 3.3, synthesisers also receive the position parameters, which are updated for each video frame. A particular problem is the range of *x* and *y*, since the specific value of these parameters may be useful (e.g., when representing musical notes), but often needs to be mapped to another range (e.g., frequencies). For this reason, the synthesiser is passed the raw values plus parameters for the width and height, so that it is up to the synth definition to normalise the value if desired.

6. INITIAL EXPLORATIONS

Mob has been developed and used by the author in several public performances, using different audio corpora as terrain. The system has been extended to implement the more general framework presented in section 3. Some examples of improvisation with the functions described in section 4 and different synthesis techniques are provided as additional material for this article (Video Examples 1–4).

When used with audio corpora, the system has proved to be a very powerful way to explore the sounds contained in the corpus and different sound forms that can be made with it, allowing the combination of systematic and random exploration. In comparison with other corpus-based approaches, typically using input devices to control trajectories in the corpus visualisation, live coding of agents allows finer control, and a wider range of possibilities such as randomness, algorithmic automation, or trajectories that cannot be performed via gestures with a given sensor. The use of the FluidCorpusMap library also facilitates the creation of polyphonic mixtures, as the sounds are evenly spread in the visualisation, and different locations correspond to different sound qualities. Performing with pre-recorded audio corpora is typically a sweet spot in the continuum between composition and performance, as the curation of audio materials can be used to orient the overall range of sounds produced in the performance. Curating the synthesis algorithms in advance further constrains the live coding activity to control structures, in the spirit of event-based live coding. Given the potential of the dialogue with the corpus for creating a diversity of sounds, this is a good compromise, as coding the synthesis process can at times be time-consuming.

The extension of the model to other types of terrains and synthesis techniques shows promise for ABMLC as a general interaction paradigm for music live coding. Using simple 2D functions such as the ones described in section 4, the agents are merely autonomous, with a continuous behaviour. In comparison with other event-based live coding strategies, this seems to lead to the creation of more continuous structures (although coding functions with jumps is of course also possible). This is also helped by the use of spatial position as a parameter, in the sense that when a new agent function is compiled, the agent is still at the same place. Despite the simplicity of the individual agents behaviour, merely adding multiple instances quickly allows creating programs that exhibit a strong artificial agency, in the sense that the result is not easily predictable by the live coder. This ease for serendipity can be advantageous both for performance and composition.

The visualisation is a generally welcome complement to the code, both for the performer and the audience. Feedback from the performances has been positive in this respect. The animation can be seen as a mediator between the live code and the sound, which facilitates the understanding of the code and its effect on the resulting sound. From the point of view of the performer, this helps being in control and improving the ability to quickly understand and modify the code. In comparison to other live coding approaches, having a 'big picture' visualisation paired with physical control of each agent via midi supports risk-taking, which is quite helpful for experimental music. From the point of view of the audience, the visualisation of the terrain and agent trajectories along with the code enhances the readability of the performance.

The implementation is limited in several ways, which prevents further exploration of the ABMLC framework. One example is the manual creation and destruction of agents. When experimenting with agent models, it is clear that being able to programmatically create and remove agents would add great potential for music structure. For example, it would allow starting several sounds at once. The current interface leads to a progressive building style by starting the agents one by one. This is a common feature of live coding when starting from scratch, but here decoupling the code of the agent models from the text snippet interface would allow more flexible control of agent groups. Similarly, the manual control (and the link to hardware MIDI controllers) limits the number of agents to small numbers (e.g., around eight), while there are many interesting musical ideas that could be realised with larger numbers of agents.

Another limitation is the link between graphics and model updates. This is common in ABM software, where real-time is usually not a big concern, but here the rate has a very strong impact on the coding of agent functions and the resulting sounds. The ideal rate really depends on the kind of music. The rates used in the experiments so far are similar to typical control rates in computer music systems, but the rate could also be defined in terms of musical tempo for rhythmic music. Being able to run each agent at a different rate would clearly be much more flexible and easier to code. In any case, decoupling the rate from the graphics updates would be beneficial for both performance and usability reasons.

Finally, a lot of uncharted territory remains for ABMLC. The present study has focused mostly on the control of trajectories, independent of other agents, and using the terrain mostly to feed the synthesis process. ABM typically exploits local interactions between agents and between agent and environment to study emergent behaviour such as swarming. Self-organisation algorithms are common in computer music composition (Blackwell and Young 2004). The simplified model proposed here is focused on functions that can be coded quickly in a performance situation. An open question is thus what the ideal interface and level of granularity would be for creating and controlling agent swarms through live coding.

7. CONCLUSIONS

This article proposes ABMLC as a live coding paradigm for music performance and composition, and introduces an extensible open source implementation. The framework emerged from a practice in corpus-based performance and has been used mostly with 2D representations of audio corpora. In this context, unlike existing approaches, ABMLC allows real-time control of algorithmic explorations of the corpus.

The model can be extended beyond corpus-based performance to accommodate different live coding practices, including discrete events and continuous parameter control of arbitrary sound synthesis and sampling techniques. The focus on rapid coding of multiple agents easily leads to unpredictable collective behaviours. The use of a 2D visualisation as a mediator improves the understanding of live coding improvisation for both the performer and the audience.

Further work on the interaction between agents and between agents and terrain will help fully realise the potential of the framework.

SUPPLEMENTARY MATERIAL

To view supplementary material for this article, please visit https://doi.org/10.1017/S1355771823000274

REFERENCES

All things live coding. 2022. github.com/toplap/awesome-livecoding (accessed 16 January 2023).

Anderson, S. and Anderson, S. D. 2020. Coding and Music Creation in a Multi-Agent Environment. *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*. New York: ACM, 527–8.

Blackwell, T. and Young, M. 2004. Self-Organised Music. *Organised Sound* **9**(2): 123–36.

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

Caclin, A., McAdams, S., Smith, B. K. and Winsberg, S. 2005. Acoustic Correlates of Timbre Space Dimensions: A Confirmatory Study Using Synthetic Tones. *The Journal of the Acoustical Society of America* **118**(1): 471–82.

Cadiz, R. F. and Colasso, M. 2012. Osc-Netlogo: A Tool for Exploring the Sonification of Complex Systems Using Netlogo. *Proceedings of the 2012 International Conference on Computer Music*. Ljubljana: ICMA, 379–82.

Collins, N. 2022. SLUGens. composerprogrammer.com/code.html (accessed 16 January 2023).

Collins, N., McLean, A., Rohrhuber, J. and Ward, A. 2003. Live Coding in Laptop Performance. *Organised Sound* **8**(3): 321–30.

Dunne, J. 2007. Monome 40h Multi-Purpose Hardware Controller. *Computer Music Journal* **31**(3): 92–4.

Eigenfeldt, A. and Pasquier, P. 2011. A Sonic Eco-System of Self-Organising Musical Agents. *European Conference on the Applications of Evolutionary Computation*. Torino: Springer, 283–92.

Garber, L., Ciccola, T. and Amusategui, J. 2020. AudioStellar, an Open Source Corpus-Based Musical Instrument for Latent Sound Structure Discovery and Sonic Experimentation. *Proceedings of the 2021 International Conference on Computer Music*. Santiago: ICMA, 62–7.

Griffiths, D. 2008. Al-Jazari. www.pawfal.org/dave/index.cgi?Projects/Al%20Jazari (accessed 16 January 2023).

James, S. and Hope, C. 2011. Multidimensional Data Sets: Traversing Sound Synthesis, Sound Sculpture, and Scored Composition. *Proceedings of the 2011 Australasian Computer Music Conference*. Auckland: ACMA, 60–6.

Kleppe, M. 2022. Tixy Land. tixy.land (accessed 16 January 2023).

Magnusson, T. 2011. The IXI Lang: A Supercollider Parasite for Live Coding. *Proceedings of the 2011 International Computer Music Conference*. Huddersfield: ICMA, 503–6.

McLean, A., Griffiths, D., Collins, N. and Wiggins, G. 2010. Visualisation of live code. *Electronic Visualisation and the Arts*. London: EVA, 26–30.

Miranda, E. R. and Todd, P. M. 2003. A-Life and Musical Composition: A Brief Survey. *Proceedings of the 9th Brazilian Symposium on Computer Music*. Campinas: SBC.

Mitsuhashi, Y. 1982. Audio Signal Synthesis by Functions of Two Variables. *Journal of the Audio Engineering Society* **30**(10): 701–6.

Nishibori, Y. and Iwai, T. 2006. Tenori-on. *Proceedings of the 2006 International Conference on New Interfaces for Musical Expression*. Paris, 172–5.

Resnick, M. 1994. *Turtles, Termites, and Traffic Jams: Explorations in Massively Parallel Microworlds*. Boston: MIT Press.

Resnick, M. 1996. StarLogo: An Environment for Decentralized Modeling and Decentralized Thinking. *Conference Companion on Human Factors in Computing Systems*. Vancouver: ACM, 11–12.

Roma, G. 2016. Colliding: A Supercollider Environment for Synthesis-Oriented Live Coding. *Proceedings of the 2016 International Conference on Live Interfaces*. Brighton, 58–64.

Roma, G., Xambó, A., Green, O. and Tremblay, P. A. 2021. A General Framework for Visualization of Sound Collections in Musical Interfaces. *Applied Sciences* **11**(24): 11926.

Rowe, R. 1992. *Interactive Music Systems: Machine Listening and Composing*. Boston: MIT Press.

Schwarz, D., Beller, G., Verbrugghe, B. and Britton, S. 2006. Real-Time Corpus-Based Concatenative Synthesis with Catart. *9th International Conference on Digital Audio Effects*. Montreal, 279–82.

Shoham, Y. 1993. Agent-oriented Programming. *Artificial intelligence* **60**(1): 51–92.

Solomon, C., Harvey, B., Kahn, K., Lieberman, H., Miller, M. L., Minsky, M., et al. 2020. History of Logo. *Proceedings of the ACM on Programming Languages* 4(HOPL), New York, 1–66.

Tisue, S. and Wilensky, U. 2004. Netlogo: A Simple Environment for Modeling Complexity. *International Conference on Complex Systems*. Boston: NECSI, 16–21.