
Chapter 3

Algorithmic composition: swarm music

3.1 Introduction

In algorithmic composition, a composer creates or utilises abstract processes in order to generate sequences of musical events. As such, algorithmic composition can be described as a form of ‘meta-composition’ (Taube 2004, Xenakis 1992), because the composer engages in composition at a higher, or more abstract, level than might otherwise be the case. To contrast algorithmic composition with non-algorithmic composition, a composer not working with algorithmic techniques will typically create a composition by explicitly specifying an arrangement of notes, perhaps by producing a musical score or by entering musical notes into a computer-based sequencer. The key point is that the composer of non-algorithmic music is working directly at the level of musical notes – they may write down a melody that they have imagined in their head, or that they have created by improvising with a musical instrument. In contrast, a composer using algorithmic techniques is less concerned about specifying the exact arrangement of musical events in time, and instead focuses attention towards higher level abstract processes. It is these abstract processes that are responsible for generating the musical sequences that are heard as the music. Composing music using algorithms opens up many fascinating avenues of exploration; it also presents many challenges.

It is important to remember that the use of abstract processes and algorithms in musical composition is not a new idea, and certainly predates the invention of the computer. The role of abstract processes in the creation of music dates back to the Middle Ages, with the **canon** being one of the earliest examples (Harley 2009, p. 109). In the fourteenth-century, composers of the Ars Nova style developed a technique now known as **isorhythm** (Bent 2010). This process involves the combination of a rhythmic pattern (called a **talea**) with a melodic phrase (called a **color**), which are typically of different lengths, in order to generate musical variation.

Music throughout the twentieth-century has significantly expanded the range of process-based compositional techniques; for example, those of the serialist and aleatoric traditions. It is also important to remember that even when a computer is used to execute algorithmic processes, the resulting composition can still be performed by human musicians (see Harley (2009) for an in-depth survey). For example, the *Illiac Suite* for string quartet (1955–56), programmed by Lejaren Hiller and Leonard Isaacson, relied on a computer to generate the score, employing a range of innovative algorithmic techniques. Similarly, during the composition of the string quartet *ST/4,1-080262*, Iannis Xenakis used a computer to carry out stochastic calculations in order to generate the musical material (Griffiths 1995, p. 207).

When a computer is used to realise an algorithmic work, a considerable range of possibilities opens up. As well as using algorithmic processes to determine notes,

pitches, rhythms, dynamics and instrumentation, the whole palette of sound synthesis techniques becomes amenable to algorithmic control. There has been a rapid development in dedicated music programming languages and environments over recent decades offering the composer easy access to this exciting new world. Widely used software includes: Chuck¹, Common Music², CSound³, Impromptu⁴, Max/MSP⁵, Pure Data⁶, and SuperCollider⁷. One aspect of the design of these and other similar software is to meet the needs of algorithmic composers. The following quotation from James McCartney, the original creator of SuperCollider, describes the key objectives behind the software design.

Motivations for the design of SuperCollider were the ability to realize sound processes that were different every time they are played, to write a piece in a way that describes a range of possibilities rather than a fixed entity, and to facilitate live improvisation by a composer/performer. (McCartney 2002, p. 61)

Many music programming systems today are highly dynamic and allow the composer to interact in real time with the generative process. This has allowed composers to go one step further and take algorithmic composition into live performance. The practice of live coding brings together algorithmic composition and live performance (Collins et al. 2003). Live coding involves writing and interacting with algorithm in front of an audience in order to generate music or visual animation. Typically, a performer projects their screen simultaneously to help give the audience some insight into the abstract processes they are creating. Live coding not only places tough demands on the performer (they might introduce some code that causes their system to crash!), but also on the programming language itself. A system must be robust and efficient, but importantly must also be flexible enough to allow the performer to concisely express their musical ideas. Live coding is a rich application domain for research into language design and also the psychology of programming (Blackwell & Collins 2005).⁸

As Robert Rowe notes, the range of techniques employed in composition is much wider than that used to analyse and understand music (Rowe 2001, p. 201). Explanatory models are required to be grounded in more general terms, in the case of music, often supported by cultural, perceptual, cognitive or musicological theories. Whereas compositional techniques require no such external validation, and Rowe gives the example of a composer utilising a telephone directory in the act of composition. As such, the evaluation of algorithmic composition techniques need only rest on 'whether they output structures that make musical sense' (Rowe 2001, p. 7). The methodological considerations of algorithmic composition are also discussed in Section 1.2 of this study guide.

The very open-ended range of possibilities available to the algorithmic composer presents opportunities and challenges. The composer is free to explore arbitrarily complex abstract processes, which might generate surprising and high quality musical results. However, processes that include elements of randomness may vary considerably in the quality of their output if the indeterminacy is not tightly controlled. Control itself may be an issue, for if the algorithmic process is too complex, the composer may lose sight of how the process relates to the music it

¹<http://chuck.cs.princeton.edu/>

²<http://commonmusic.sourceforge.net/>

³<http://csound.sourceforge.net/>

⁴<http://impromptu.moso.com.au/>

⁵<http://cycling74.com/>

⁶<http://puredata.info/>

⁷<http://supercollider.sourceforge.net/>

⁸Also see the TOPLAP website for more information on live coding: <http://www.toplap.org/>

generates. In other words, the process could become a 'black box' instead of a transparent process that the composer is able to engage with – however, this might not necessarily be a bad thing, and may even be an artistic aim. One under researched topic in algorithmic composition is the issue of larger scale musical form. Algorithms used to generate musical material are often constrained to producing one particular kind of musical patterning or texture, and lack a means of generating structured variation over larger time frames (Collins 2009).

Algorithmic composition also raises issues for listeners and musicologists. To consider listeners: is it necessary that a listener should be able to hear and understand the process that generated an algorithmic work in order to fully appreciate it? See Lerdahl (1992) for one argument on this subject from the point of view of a cognitive scientist. A similar issue is raised concerning the musicological analysis of generative music: how is it possible to analyse a piece of music that might be different each time it is performed (Collins 2008)? Issues of authorship are also brought into question by the practice of algorithmic composition. In most cases it seems reasonable to assume that even if an entire musical work has been generated by an algorithm, we would still attribute authorship to the composer since they created the algorithm. However, does the situation become different when an algorithm becomes extremely complex, or has the ability to learn and modify its own behaviour to the extent that it could be said to be creative (Wiggins 2006)?

The remainder of this chapter explores swarm process for algorithmic composition. Swarm music is an example of a class of algorithmic techniques that are inspired by behaviour observed in the natural world.

3.2 Swarms, music and improvisation

Swarming locusts, buffalo herds and anchovy schools are just three examples of the many kinds of groupings of social animals. The incredible and seemingly choreographed behaviour of starling flocks, filmed at Otmoor, Oxfordshire, UK⁹ is a particularly striking and beautiful example.

This synchronised, co-ordinated dance seems so perfectly executed that we can only assume a leader bird has choreographed the display. However the swirling, darting patterns are believed to occur spontaneously, deriving from simple and local rules that do not in themselves contain features of the overall, global behaviour of the group. Flocks, and systems displaying similar behaviour, are said to **self-organise** (Bonabeau et al. 1999).

Self-organisation (SO) has been observed in many systems in physics and chemistry as well as in market economics, motorway traffic flow, and in biological systems such as animal groups and immune systems. In flocks, the collective order arises from the motions of individual birds that only react to the motions of nearby neighbours. Individual starlings do not have a sense of the overall flock. In general, self-organising systems display emergent structures that are not imposed by external ordering influences, but result from local interactions and without reference to the global pattern.

This chapter explores the idea that music may be viewed as a self-organising system. At first glance, music does not seem to be a candidate for emergence.

⁹<http://www.youtube.com/watch?v=XH-groCeKbE>

Systems that self-organise are also usually **decentralised**. Antibodies in an immune system mobilise against the pathogen in a way that is quite unlike our own centralised way of organising armies, with top down control flowing through a command hierarchy, headed by the General (read *Turtles, Termites and Traffic Jams* (Resnick 1997) for a convincing and heartfelt account of decentralisation).

Western art music is highly centralised, headed by the composer. However, music outside this tradition, for example jazz and folk music world-wide, is highly improvisational. Much of the musical content is not planned ahead of the performance but is generated afresh and in real time. Improvised music is much more akin to the decentralised, bottom up, structuring that we see in flocks, swarms and motorway bunching.

Imagine replacing starlings by musical notes (crotchets, quavers etc.) and letting these notes 'fly' across a sheet of musical manuscript: what might this sound like? Examine (almost) any musical score you can find. If the score has been composed, you will find a tight structure, governed by statement, repetition and variation. However, if you look at a transcription of an improvisation (the transcriptions of the alto saxophonist Charlie Parker¹⁰ provide exemplary examples) you will notice a looser, more organic flow. Perhaps a swarm of interacting notes might generate such a solo?

Let us examine in more detail the local concerns of individuals in a swarm. The overriding concern is surely not to collide! Beyond this is the need to fly towards other individuals, especially for those individuals on the outside of the swarm. (We will focus on swarms rather than flocks because they are simpler. In dynamical terms, individuals in flocks attempt to match flight direction, as well as attract, and avoid collisions.) Further, notes in music also avoid collisions, or else the music would be too bland. Notes do not wander aimlessly across the staff with large jumps separating them; they clump together as if mutually attracted.

Apart from attraction and collision avoidance, there are a couple of other noteworthy parallels. The spatial patterns produced by a swarm resemble previous patterns, but are unlikely to be exact repetitions. This is also a feature of improvised music, where strict repetition is unusual, if not undesirable, yet some degree of similarity is apparent.

A further property of SO is the 'amplification of fluctuations'. In a swarming example, neighbouring individuals might become attracted towards a wayward straggler (a fluctuation); and then neighbours of the neighbours are diverted, eventually causing the entire population to fly towards the outlier (amplification). Similarly, improvised music can take on unpredictable twists and turns with the entire ensemble spontaneously moving in new musical directions, as if carefully planned.

These analogies are suggestive of links between SO and improvised music that might be made at some level of analysis. The question for the computer musician is this: can we use principles of SO to generate sequences of notes that, although unfamiliar and improvisational in character, we might hear as musical?

¹⁰*The Charlie Parker Omnibook*, Jamie Aebersold, pub. Criterion (1976) ISBN 0769260543

3.3 Project: MIDI synthesiser

This chapter contains a number of programming projects culminating in a swarm-based improvisational system. It is a simple version of the author's own Swarm Music (Blackwell 2007).

First, you will need to build a synthesiser for the production of sound. A MIDI synthesiser will be sufficient for our purpose. We will use Java since it has convenient MIDI and graphics libraries (the latter will become important when coding the swarm simulation) and because you are familiar with this language.

MIDI allows messages between devices to be transmitted over one of 16 channels. Each channel may be set to a different instrument (piano, guitar, etc.) by sending a program change message. Notes are played by sending note-on messages to a particular channel, along with the number of the note to play and its velocity (how hard a key was pressed for example, which is usually interpreted by a synthesiser as loudness, although it may also affect the timbre of the sound). MIDI note numbers increase in semitones and range from 0 to 127. Velocity values also range from 0 to 127, with 0 corresponding to the quietest level, and 127 to the loudest. Sounding notes are terminated by sending note-off messages.

Channels also respond to control messages. These do not play notes but alter the instrument characteristics in some way. For example 'control change 10 72' will set control number 10 – which corresponds to pan (stereo position) – to a value of 72. There are 128 controls, and each has a value in the range 0–127.

The Java `javax.sound.midi` package contains several classes which represent channels, synthesisers, messages and other MIDI technology that need not concern us here. You should now take a look at this package in the Java API, and in particular study the `MidiChannel`, `MidiSystem` and `Synthesizer` classes.

The following program shows a single channel MIDI synthesiser. A `Synthesizer` object is obtained from the `MidiSystem` class, which interacts directly with your machine. Notice that the synthesiser must be opened, and a `close` method shuts the synthesiser down. This method should be called when the program is terminated. This synthesiser has a single channel.

Add some code where indicated to play a few notes. You will have to pause the program for a while in between sending note-on messages. This can be achieved by asking the program thread to 'sleep' for a given time, as measured in milliseconds.

Try to pan the sound so that low notes are towards the left and high notes are towards the right. (You will have to send a `controlChange` message to the `MidiSynth.channel`.)

3.3.1 Simple MIDI synthesiser

```
import javax.sound.midi.MidiChannel;
import javax.sound.midi.MidiSystem;
import javax.sound.midi.MidiUnavailableException;
import javax.sound.midi.Synthesizer;

public class SimpleMidiSynth {

    Synthesizer synthesizer = null;
    MidiChannel channel;

    SimpleMidiSynth() {

        try {
            synthesizer = MidiSystem.getSynthesizer();
            synthesizer.open();
        } catch (MidiUnavailableException e) {
            e.printStackTrace();
        }
        MidiChannel[] channels = synthesizer.getChannels();
        channel = channels[0];
    }

    void close() {
        synthesizer.close();
    }

    public static void main(String[] args) {

        MidiSynth synth = new SimpleMidiSynth();

        // Play some notes by calling
        // noteOn(int noteNumber int velocity)
        // on this midi synth's single available channel.
        // Remember to call noteOff(int noteNumber)
        // before sending the next noteOn.

        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {
        }

        synth.close();
    }
}
```

3.4 Rules of swarming

There are many ways of formulating the swarming rules alluded to in the Introduction. The idea is to find a set of rules that describes a swarm **at some level**, and, when coded, produces a swarm-like animation. The correct level of description for our purposes is in terms of a **dynamical system**.

In the dynamical system model, swarm individuals are represented as **particles**. Particles are simple dynamic entities of zero size, moving in space, and subject to forces due to inter-particle interaction. In other words they are described by position (\mathbf{x}), velocity (\mathbf{v}) and acceleration (\mathbf{a}) vectors. They have no internal state or any other property.

Time is necessarily discrete in this model. Each particle is updated by calculating an acceleration vector and adjusting velocity and position. The **dynamical state** at time $t + 1$ is obtained from the state at t according to a discrete particle kinematics:

$$\begin{aligned}\mathbf{v}(t + 1) &= \mathbf{v}(t) + \mathbf{a}(t) \\ \mathbf{x}(t + 1) &= \mathbf{x}(t) + \mathbf{v}(t)\end{aligned}$$

An animation proceeds by a sequence of frames; each frame is a snapshot of the swarm at time t . The animation is paused for a short time between frames, the swarm is updated and the new frame is displayed.

The acceleration vector can be calculated in various ways, but here we specify a method that leads to good results and has its origins in biological modelling of actual swarming animals.

Each particle is considered to have two zones of perception, an inner and an outer zone. The particle cannot 'see' any other particle that is outside the outer zone. This means that the interactions are **localised**. Particles are repelled from the centre of mass of any particles within the inner zone, and attracted towards the centre of mass of any particles within the outer zone, but outside the inner zone. The Figures 3.1 and 3.2 below illustrate the intention.

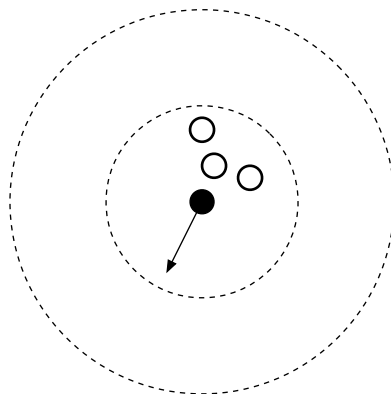


Figure 3.1: Repulsion away from three neighbours within the inner zone.

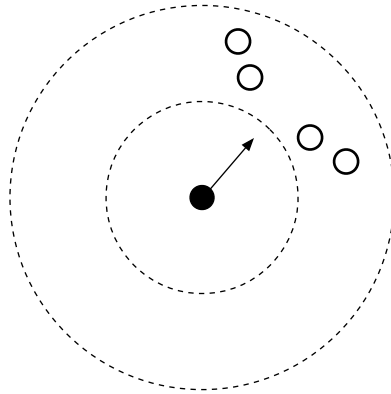


Figure 3.2: Attractions towards four neighbours within the outer zone.

In detail, the algorithm is:

- if the inner zone of the particle at \mathbf{x} is occupied, find the neighbour centre of mass \mathbf{y}_{CM} where \mathbf{y} is the position of each neighbour. The acceleration is then given by $\mathbf{a} = \mathbf{x} - \mathbf{y}_{CM}$. The centre of mass of a set of points $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_n$ is $\mathbf{y}_{CM} = \sum_{i=1}^n \mathbf{y}_i$
- else if the outer zone of the particle at \mathbf{x} is occupied, find the neighbour centre of mass \mathbf{y}_{cm} where \mathbf{y} is the position of each neighbour. The acceleration is then given by $\mathbf{a} = \mathbf{y}_{CM} - \mathbf{x}$
- else $\mathbf{a} = 0$.

Notice that avoidance always has precedence: if both zones are occupied, a single avoidance acceleration is calculated. Finite accelerations may be 'clamped' to a fixed magnitude a_0 .

In practice the velocity is also clamped to a constant. Individuals in a swarm or flock tend to move at a steady speed. Velocity clamping prevents both high speeds and stagnation, neither of which are observed in nature. Velocity clamping ensures that particles always move at the same speed; accelerations instigate turning, not speeding or slowing.

The complete particle update kinematics are specified by the following steps:

$$\mathbf{a} = \begin{cases} \mathbf{x} - \mathbf{y}_{CM} \\ \mathbf{y}_{CM} - \mathbf{x} \\ 0 \end{cases} \quad (3.1)$$

$$\mathbf{a} \leftarrow a_0 \frac{\mathbf{a}}{|\mathbf{a}|} (|\mathbf{a}| \neq 0) \quad (3.2)$$

$$\mathbf{v}(t+1) = \mathbf{v}(t) + \mathbf{a}(t) \quad (3.3)$$

$$\mathbf{v}(t+1) \leftarrow v_0 \frac{\mathbf{v}_{t+1}}{|\mathbf{v}_{t+1}|} \quad (3.4)$$

$$\mathbf{x}(t+1) = \mathbf{x}(t) + \mathbf{v}(t) \quad (3.5)$$

3.5 Project: vector maths

Although not necessary, the development of a vector maths library will keep your code tidy when you begin to implement the full swarm animation.

We can identify the following operations:

- vector addition and subtraction
- multiplication of a vector by a scalar
- finding the length of a vector
- clamping a non-zero vector to a scalar magnitude
- finding the distance $|\mathbf{a} - \mathbf{b}|$ between points at positions \mathbf{a} and \mathbf{b} .

Implement these operations using static methods and place them in a class `VecMaths`. Write a test program to verify that the methods are behaving as expected. The following class `SimpleVecMaths` provides a skeleton. For our purposes a vector is an array of `floats`. The animation package that we will use draws to single precision accuracy (easily sufficient resolution on a computer screen). Working to single precision accuracy avoids excessive casting later on.

There are a few method prototypes with three arguments. The first argument is a pre-defined vector, used to carry the result of the calculation. This saves the method from creating a new array at each call. A reference to the result is also returned, enabling the method call to be used in an expression.

3.5.1 Simple vector maths

```
import java.util.Arrays;

public class SimpleVecMaths {

    static float length(float[] vec) {

        float mag = 0;
        for (int d = 0; d < vec.length; d++)
            mag += vec[d] * vec[d];
        return (float) Math.sqrt(mag);
    }

    static void clamp(float[] vec, float length) {
        // Place your code here.
    }

    static float[] add(float[] result, float[] vecA, float[] vecB) {
        // Place your code here.
        return result;
    }

    static float[] subtract(float[] result, float[] vecA, float[] vecB
    ) {
        // Place your code here.
        return result;
    }

    static float[] scalarMult(float[] result, float scalar, float[]
    vec) {
        // Place your code here.
        return result;
    }

    static float dist(float[] vecA, float[] vecB) {
        // Place your code here.
        return 0;
    }

    public static void main(String[] args) {

        float[] vec = new float[] { 3, 4 };
        System.out.println("length of " + Arrays.toString(vec) + " is "
            + length(vec));
    }
}
```

3.6 Project: particle animation

We are almost ready to begin coding the animation. We shall make use of *Processing*, a Java animation and graphics environment which you have used extensively in previous units. This will enable us to ignore many details of Java graphics and threading, and concentrate on what we want to do, namely program a swarm animation, and turn it into music.

Open the *Processing* editor. Type `ParticleApplet.java` directly into the editor and save the 'sketch' as `ParticleApplet`. A *Processing* sketch is actually a Java **Applet**, i.e. a Java program which runs inside a webpage. *Processing* will do the embedding for you if you click on the 'export' button.

`ParticleApplet` refers to two other classes, `Particle` and `VecMaths`. `Particle` conveniently holds position, velocity vectors and the particle size (the size is used purely for rendering and takes no part in the dynamics). The class definition is given below; `VecMaths` has already been defined (by you!). Both of these classes need to be loaded into the *Processing* editor.

Click on the arrow button on the top right of the tab menu bar. Select 'New Tab' and type (or copy and paste) the sourcecode for `Particle` into the blank editor pane. Save the new file (this dialogue appears at the foot of the editor) as `Particle.java`. It is very important to remember the `.java` ending. Repeat the procedure with `Vecmaths`.

The `PApplet`'s `setUp()` method is performed when the applet is launched, rather like a constructor. Here, position and velocity vectors (floating point arrays) are initialised. The velocity is clamped, and a `Particle` object is created. The call to `smooth` asks the graphics to perform antialiasing so that edges are sharp, and `noStroke` means that shapes are drawn without an outline.

`PApplet` repeatedly calls `draw` at a default rate of 60 times per second. This is known as the frame rate. In this case `draw` paints the entire frame black,

```
background(0),
```

and prepares to draw and fill with white any requested shape,

```
fill(255),
```

and finally draws a circle where the particle is located,

```
ellipse((particle.position[0], particle.position[1], particle.diam,
particle.diam).
```

If all goes to plan, when you click on run, you will see a black background with a stationary white disc. We have drawn the particle, but it does not move, because the position does not change.

Add your own code where indicated to make the particle move. The update is simple. In the absence of interactions, just add the velocity to the position.

After a while the particle will leave the frame. Your code will therefore need to take action when this happens. One possibility is that the particle is reflected, like a

billiard ball; another possibility is that the particle reappears on the opposite edge.

3.6.1 Particle applet

```
import processing.core.PApplet;

public class ParticleApplet extends PApplet {

    private static final long serialVersionUID = 1L;

    int D, L;
    float speed, particleDiam;

    Particle particle;

    public void setup() {

        size(L, L);

        float[] x = new float[D];
        float[] v = new float[D];

        for (int d = 0; d < D; d++) {

            x[d] = (float) (L * Math.random());
            v[d] = (float) (Math.random() - 0.5f);
        }
        VecMaths.clamp(v, speed);
        particle = new Particle(x, v, particleDiam);

        smooth();
        noStroke();
    }

    public void draw() {

        background(0);

        // Place your code here.

        fill(255);
        ellipse(particle.position[0], particle.position[1], particle.
            diam,
            particle.diam);
    }

    public static void main(String args[]) {

        PApplet.main(new String[] { "SimpleSwarmMusic" });
    }
}
```

3.6.2 Particle

```
public class Particle {
    float[] position, velocity;
    float diam;

    Particle(float[] x, float[] v, float d) {
        position = x;
        velocity = v;
        diam = d;
    }
}
```

3.7 Project: swarm animation

Now that you have succeeded in animating a single particle, it is time to implement a swarm of interacting particles. Since a swarm is a collection of particles, it makes sense to introduce a *Swarm* class to store the particle objects. *SimpleSwarm* shows an outline class. The associated *PApplet*, *SimpleSwarmAnimation*, transfers the drawing to the swarm object by calling `swarm.draw`, and passing a reference to itself. Your code for reflection or doubling-back should be pasted in the corresponding empty methods. The hardest part is to code the particle interactions, as defined in the algorithm above, and to choose the constants so that the animation proceeds at a reasonable pace (not too fast or too slow), and gives interesting swarm behaviour.

The kinematic parameters and other constants affecting the animation and swarm behaviour are:

f	frame rate
a_0	acceleration magnitude
v_0	speed
r_{avoid}	radius of inner avoidance zone
r_{attr}	outer radius of attraction zone
N	number of particles
D	dimension of space
L	size of space in each dimension

Additionally we note the two algorithms:

- particle kinematics steps (1)–(5)
- boundary rules.

To help you decide on parameter values, note that v_0 is the speed in pixels/frame. The speed in pixels/second is obtained by multiplying v_0 by the frame rate; the actual speed in cm/sec is found by multiplying this by the size of the pixels on your display. You can calculate the pixel size by dividing the resolution of your display by the linear measurement of your screen. A speed of a few cm/sec is about right.

3.7.1 Simple swarm

```

import java.util.ArrayList;
import processing.core.PApplet;

public class SimpleSwarm {

    int D, L;
    float speed, particleDiam;

    private ArrayList particles;

    public SimpleSwarm(int numParticles, int dim, int xMax) {

        D = dim;
        L = xMax;

        particles = new ArrayList();
        for (int p = 0; p < numParticles; p++) {

            float[] x = new float[D];
            float[] v = new float[D];

            for (int d = 0; d < D; d++) {

                x[d] = (float) (L * Math.random());
                v[d] = (float) (Math.random() - 0.5f);
            }
            VecMaths.clamp(v, speed);
            particles.add(new Particle(x, v, particleDiam));
        }
    }

    void wrap(float[] vec) {
        // Code to make the particle reappear on the opposite edge.
    }

    void reflect(float[] pos, float[] vel) {
        // Code to make the particle reflect at the edge.
    }

    void draw(PApplet pApplet) {

        for (int p = 0; p < particles.size(); p++) {

            Particle particle = (Particle)particles.get(p);

            for (int n = 0; n < particles.size(); n++) {

                if (n == p)
                    continue;
                Particle neighbour = (Particle)particles.get(n);

                // Particle interaction code goes here.
            }

            wrap(particle.position);
            // reflect(particle.position, particle.velocity);
        }
    }
}

```

```

        pApplet.ellipse(particle.position[0], particle.position[1],
            particleDiam, particleDiam);
    }
}

public Particle getParticle(int p) {
    return (Particle)particles.get(p);
}

public int getNumParticles() {
    return particles.size();
}
}

```

3.7.2 Simple swarm applet

```

import processing.core.PApplet;

public class SimpleSwarmApplet extends PApplet {

    private static final long serialVersionUID = 1L;

    int D, L, numParticles;
    SimpleSwarm swarm;

    public void setup() {

        size(L, L);

        swarm = new SimpleSwarm(numParticles, D, L);

        smooth();
        noStroke();
    }

    public void draw() {

        background(0);

        fill(255);
        swarm.draw(this);
    }
}

```

3.8 Project: swarm music

Hopefully you will now have a swarm animation running. It is now time to interpret particle positions as notes. Once more, keep your code tidy by defining a new class, an `Interpreter` class with a method `void interpret(Swarm swarm)`. This class will contain a `MidiSynth` object. The purpose of `interpret` is to take particle positions and form notes from the position components. A simple and direct scheme is to extract a note number from one coordinate, and a note velocity from the other.

Suppose the particles are flying in a box $[0, L]^2$. A linear interpretation would imply

$$n = n_{min} + \frac{x}{L}(n_{max} - n_{min}) \quad (3.6)$$

$$l = l_{min} + \frac{y}{L}(l_{max} - l_{min}) \quad (3.7)$$

where n and l are midi note numbers and midi velocities corresponding to a position (x, y) . Notes and velocities are scaled between maximum and minimum values which can be adjusted as the user wishes.

Since `draw` will be called 60 times a second, and since your swarm may have 10 (or more) particles, you certainly don't want to interpret each particle. Instead, you will have to **sample** the swarm, interpreting a particle every δt seconds where δt is the required duration of the note. Particles can be chosen in turn. Clearly we wish to sample the swarm often enough so that the interpretation is a good representation of the configuration of the particles. It is helpful to draw a blob at each interpreted position; erase and redraw a new blob when the next particle sounds.

3.9 Swarm music as a live algorithm

We close this chapter by considering how a musician might interact with the music system that you have just built. The version of swarm music described in the previous pages is an example of a **generative** music system. In other words, the output depends only on the choice of internal parameters, the sequence of random numbers, and the details of the algorithm. The music is generated as if by machinery.

Swarm music¹¹ was conceived as an interactive application. The intention is that musicians play along with the system just as if taking part in a duet with another (human) musician. Such systems have been termed **Live Algorithms**.¹² The term is meant to convey the impression that the system behaves autonomously, almost as if alive, and also alludes to the frequent use of artificial life algorithms (which includes swarming simulations) in this context.

Swarm music employs an interactive model that is inspired by the indirect interaction known to social entomologists as **stigmergy**. A stigmergetic interaction

¹¹www.timblackwell.com for further information, including downloads

¹²Swarming and Music. In Miranda E. and Biles A. (eds.): *Evolutionary Computer Music*. Springer Verlag 2007 ISBN 978-1-84628-599-8

is mediated by the environment rather than directly by the near (or actual) contact of interacting individuals. A chance encounter with a disturbance made sometime in the past influences future behaviour. A well known example is the laying and then following of pheromone trails by ants.

In swarm music, the musical output of the musician is mapped onto particle movement within the state space of the system. Notes are interpreted as positions within the internal space. The musician leaves a trail of particles behind her, just as if she were actually moving in the internal space. Each deposited particle then becomes an attractor for swarm particles. A particle will experience an acceleration towards any attractors that are within perceptual range; the swarming rules can be easily extended to include this scheme.

The field of live algorithm research is still developing but some desirable properties are immediately apparent.

- The system must be able to **reflect** musically what it hears. The output must bear some resemblance to the playing of the human partner. At the simplest level, a direct echo would suffice, although this would become tedious.
- The system must also be able to **innovate**. Its improvisations should suggest novelty; 'ideas' that the human partner can engage with. A quite random output would appear to satisfy this criterion but would again become tedious because of the relative lack of predictability and patterning.
- **Autonomy**, as opposed to automation, implies that the system may play, or be quiet, at any moment: this is not contingent on the activity of the interacting partner. Automatic responses will lead to predictability and musical tedium. On the other hand, apparently random responses become unpredictable and uninteresting.
- The system should have a degree of **transparency**; its internal patterning should be apparent in the interpreted sonic output, and the relation between system response and musical input should not be too obscure.

The stigmergetic model attempts to satisfy reflection, innovation and transparency. The musical output of a musician will surely lie in some region of state space, and this region will shrink somewhat, perhaps expand, and move around. In fact it appears to be a swarm (compare with the earlier remarks concerning SO and music). If the internal swarm is able to track the swarm-like image of the external musician, the output of the system will reflect what it hears. Conversely, if the swarm, or perhaps a breakaway subswarm, flies to an unrelated region of state space, the output will bear less resemblance to the input, and might be regarded as novel. Of course the musician must be able to infer some regularity concerning when this behaviour might happen; the tracking should not be too predictable or the breakaway too random.

The requirement of autonomy is satisfied to some extent in swarm music by including time-related parameters such as note duration and time between event onsets as additional dimensions in the space.

3.10 Project: interacting with swarm music

Although you could develop your program to include MIDI input, it can be quite tricky to set up the hardware, and to get Java to recognise this hardware. However, you can interact with the swarm by using the mouse. Attractors can be deposited by mouse clicks, or by pressing the mouse button and moving the cursor across the screen. The *Processing* API reveals two methods, `mouseClicked()` and `mouseDragged()` which you can override in your `PApplet`. In either case, the variables `mouseX` and `mouseY` yield the co-ordinates of the cursor. You will need to create attractor objects (these are just particles, except that they have zero velocity, and a different appearance) and store them in a variable. You should consider the lifetime of the attractors. Do they get consumed after a certain number of visits, or is there a fixed number, with the oldest being replaced by the newest?