Sound And Music Computing – Scratch examples

CAS Wiltshire Hub, Kingdown School, Warminster, 14 November 2012.

Due to some time restrictions, not all these examples were demonstrated in the session.

A key idea here is that sound can be used to demonstrate and explore fundamental CS principles. The examples suggest approaches to algorithmic composition that could be useful in a music class. They may also find uses in supporting visually impaired students. There are also aspects relevant to maths classes.

Note that for all these examples, and generally for audio and music work, it is necessary to set Turbo Speed (Edit->Set Single-Stepping), in order to generate notes with the expected timing and tempo.

Examples 1-4 present various aspects of SMC. Examples 5 and 6 demonstrate some limitations and a possible benign bug (or "undocumented feature") in Scratch, suggesting opportunities for research. The programs include a number of embedded comments and suggestions for further exploration.

Some SMC programming idioms.

These examples demonstrate some programming idioms central to Sound and Music Computing. These idioms appear in one form or another in several of the examples.

- *Concurrency*. Music is a highly concurrent activity. A chord consists of a number of notes played simultaneously (if they are adjacent, such as the notes of a scale, we would call the result a "cluster"). Most music is *polyphonic* – multiple voices or instruments playing together.
- *Modulo addressing*, *modulo arithmetic*. The one data structure supported in Scratch is the *list*. It is supported not only for creation with Scratch, but also for file input and output. When the list consists only of numbers, we can also call it an *array* or a *table*. While the basic procedure is to read a list from start to end (e.g. if the list represents a melody), it is very common in SMC to "wrap around" back to the beginning and continue reading, potentially indefinitely (a simple example of a *generative algorithm*). Computationally, it is performed by incrementing a value (i.e. the position in the table) and modifying it *modulo* the table length.
- *Time-varying* or *dynamic* control of parameters. This may be done manually ("interactively"), e.g. by means of a slider, or computationally. The latter case will often be referred to as *modulation*. For example, musical vibrato, where the pitch of a note is varied up and down more or less periodically, would be an example of pitch (or frequency) modulation.

Example 1: *starthere.sb*

This starter script introduces a possible initial approach to audible repetition and looping.

The script contains small independent routines, beginning with an elementary *repeat* loop – in effect a brief metronome. As repetition already lies at the heart of music-making, as it stands this is sonically unremarkable. To work as a metronome the repetition would need to be of indefinite length, and be controllable for speed. Possible algorithmic extensions could include accenting the first of every N beats to indicate the first beat of a "bar" where N might be any of 2,3,4,6,8, etc[1]. This can immediately incorporate nesting and modulo counting, for example to accent the first of each group of 4, to give the standard 16-step machine pulse. Ways to create the accent include increasing the volume for that note, or using a different pitch and/or voice. For use as a metronome, unpitched sounds such as drums will usually be needed.

The second example in this script extends the first to generate a plain musical scale. It introduces some variables. In pure CS terms, these are explained with reference to storage in memory, types, etc. However, to the musician something that is "variable" has special significance as something that changes over time ("dynamically"), either algorithmically or under direct user control. Scratch enables this aspect to be explored very easily.

Hence, in the third example, the tempo is made dynamically variable using a slider[2]. The variables are incremented on each passage through the original scale. Thus the pitch gradually rises, and the instrument sound used also changes. This then becomes a "scan" through all the sounds available for MIDI notes 1 to 128. The way the loop changes should be easy to hear.

This example introduces the use of a *wraparound* computation, so that when the pitch reaches an upper limit, it resets to its starting value. This is a fundamental technique in SMC. It may sometimes take the form of a simple modulo calculation, or it may mean stepping repeatedly through a table of values. On a more abstract level, the example demonstrates the *sonification* of an algorithm – the use of loops and the changing of variables etc. can be clearly heard.

Example 2: *flight.sb*

This demonstrates a simple example of *data sonification* – the rendering of non-audio data as sound. The script sonifies an inverted parabola, as might be explored in a maths class. Beyond its pure mathematical formulation the parabola is associated with the physical behavior of projectiles under the influence of

---

[1] When generated by an audio workstation as a playing guide, such a sequence is often called a "click track". See e.g. http://www.protoolsproduction.com/creatingaclicktrack.

[2] Any numeric variable can be displayed by name on the stage, where it can be configured as a horizontal slider, for which the minim and maximum values can be easily set.

gravity. The sonification in this form can therefore be related very intuitively to "real-world" experience[3].

A helicopter sprite is used to suggest flight; it could also (and most appropriately) be a form of projectile such as a cannon-ball, or a tennis ball lob.

A table is used to store computed values of the parabola. We can therefore call this a *sampled* form of the function, in contrast to the smooth (quasi-analogue) nature of the pure mathematical function. Consequently the sound is granular – the shape is represented by discrete notes. For convenience (use of integer steps), the x axis ranges between -100 and + 100, but other ranges can be freely chosen. As the comments hint, note that in such cases a few changes may need to be made to code for Sprite 2. In principle, all such numbers should be specified via variables, rather than being "hard-wired" in this way. This change is left as an exercise.

Sprite 2 demonstrates the core sonification technique of *data mapping* – the conversion of numbers from one range to another. Specially, there is a *normalization* computation to map the numbers linearly to the range 0 to 1. These values are in turn mapped *dynamically* to suitable MIDI notes and volume values, and to screen co-ordinates.

The example also demonstrates *quantisation* of values – the necessary rounding to integer in order to produce a MIDI note value. The effect here is that some notes are repeated. This is most easily heard at a slow tempo. We can thus directly hear the two primary aspects of digital sampling - the sampling of time and quantization of value.

Note that in this and other related examples we are making a virtue out of necessity. To create a smooth frequency sweep we would need to be able to program at the level of individual samples (e.g. using Python).

Example 3: *sinesounds.sb*

This is a more involved sonification example, where the subject is a sine wave. Rather than play a sine wave as a musical tone (as we can easily do using Audacity), this sonifies the shape of the waveform. Or, rather, it sonifies the sequence of samples which represent the sine wave shape.

The single most important property of a sound wave is that the waveform is *bipolar* – it cycles more or less symmetrically over positive and negative values, with the central zero line corresponding to silence. The project reminds us of this by representing the zero line as a fixed frequency, around which the notes oscillate. The **range** parameter sets the maximum excursion above and below the zero line.

A second sprite is used for the centre note, initiated via a *broadcast* instruction. This can be used either to define a named subroutine or procedure (*broadcast with*

[3] See e.g. http://en.wikipedia.org/wiki/Projectile_motion

wait), or to implement a degree of concurrency (parallel computation). In this case, it so happens that selection of MIDI voice is maintained independently per sprite; so the centre tone uses a different voice and a lower volume than the main sprite rendering the sinewave.

Four variables, including **range**, are controlled via sliders, and the program can in the first instance simply be explored freely as a simple musical automaton.

The Scratch *sin* function computes angles in degrees, hence the *mod*(360) instruction at the end of the loop. Clearly, with a very small **stepsize**, each cycle will be represented by a large number of notes (in the limit, 360 per cycle), and the sine wave shape will be very clear. Again, since this is a sampled waveform, and MIDI notes are confined to integer values (hence the *round*() instruction for **thisnote**), there will be many repeated samples.

The medium speeds and step sizes are of particular interest – if **stepsize** is an integral factor of 360, the notes will be identical in each cycle and therefore repeat exactly. For other values, it will be noticed, sooner or later, that the extreme pitches (which will dominate, sonically) tend to vary from cycle to cycle. This mimics very clearly the varying sample values of a true digital sinusoid (as might be viewed in Audacity at high Zoom). Yet, as the *sin* function is computed for each note, we know that each value must be "correct", within the limits of numerical precision.

A few variables have been given especially long names – this is the only way to create a long slider permitting fine adjustments. The tempo parameter emulates the underlying pseudo sample rate.

An advanced mathematics exercise would be to modify the project to perform a "true frequency" computation (relative to the sample rate), calculating **stepsize** as an internal variable, and using a slider-controlled variable named **frequency**. As supplied, **stepsize** is confined to integer values, but in a music-oriented digital synthesiser, this will only very rarely be the case (e.g. a frequency of 441 Hz at the 44100 Hz sample rate). Most of the time these step sizes (which in engineering parlance would usually be referred to as *phase increments*) will have floating-point fractional values. A pure digital sinusoid, free of distortion, and even allowing for quantisation errors, may well be represented by sample values that never repeat exactly.

There is therefore a natural progression from this example to the programming of a sine wave synthesizer in Python or other suitable language.

Example 4: *notesort.sb*

This presents a simple sonification of an algorithm, in this case the bubble sort. A table is filled with random numbers in a convenient range for MIDI notes (on receipt of the **init** message), and on each sorting pass, the table is "played" as a note sequence. A short pause is incorporated, to indicate each pass. There is a slightly more extensive use of *broadcast* here than in previous examples. Each pass of the bubble sort is performed in Sprite 2 in response to a **sort** message *broadcast* (with *wait*) from Sprite 1.

Sliders are used to set the tempo and the length of the table. At slow speeds the sorting algorithm can be followed note by note. As the standard procedure to sort into ascending order is used, high notes will propagate towards the end of the sequence, culminating in an ascending scale pattern – the notes almost literally bubble to the top.

In principle any algorithm that transforms input data to output data could be sonified in the same way.

The principle of deriving note information from a table is entirely general. For example, data can be imported to a table from an external text file. This will however involve some extra code, to determine the range of the numbers (minimum and maximum values), before they can be mapped to an appropriate note range.

Each complete run of the program will generate a new set of random numbers and hence a new result. This is therefore an example of *destructive processing.* Once sorted the data remains in that state. Since the main control variable **sorted** is initialized in the **init** block, directly re-triggering the **play** block will have no effect. One possible extension to the program would be to add a mechanism to copy the initial data to a second list, and some code (run independently) to reload it, to enable the process to run again with the same data[4]. Scratch supports saving and loading of the contents of a list to/from an external text file; this offers a further non-robust solution.

Another extension would be to add a count of the number of sort iterations performed in a run. As the data is random numbers, it should not be assumed that the same number of passes will be required for a given data length. Results of multiple runs could be accumulated in a separate list, enabling some observation and analysis (including further sonification) to be made of the nature of random distributions.

An interesting yet simple development, which could form part of a Mathematics session on random distributions and statistics, is to use the average of two random numbers, giving a “triangular probability density function” (TPDF)[5]. This will lead to a noticeably different pattern of pitches in the sonification.

---

[4] In most general-purpose languages, there is the option to “seed” the internal pseudo random number generator (RNG) with a particular value, causing the same sequence of numbers to be generated on each run. Where this is not desired, a common solution is to seed the RNG from the system clock. For a long sequence of numbers (e.g. to generate audio noise) this issue is generally of no consequence, but for a very short run as used here, “random” numbers can exhibit significant “local structure” and character.

[5] When used for digital audio, a common practice when reducing the resolution of a sound (e.g. from 24bits to 16 bits or less) is to “dither” the samples by adding very low-level random numbers affecting just the lowest one or two bits. This technique subjectively improves the quality of the sound. By default, *Audacity* dithers 16bit audio when written to disk, so that even a nominally silent track will contain low-level non-zero samples. This facility can be turned off via Audacity Preferences (Quality).

Example 5: *clockrock.sb*

This presents a quasi-algorithmic rendering of Bill Haley's classic "Rock Around the Clock". It demonstrates the use of the *broadcast* instruction (with *wait*) to run processes in parallel – in this case, the separately voiced melody and accompaniment. The latter uses the pattern familiar to classical musicians as the "Alberti bass", generally associated with the piano sonatas of Haydn and Mozart, in which a three-note chord (triad) is presented in a sustained rapid repeating pattern – a style which clearly has an algorithmic dimension. The melody is rendered by a dedicated second sprite. This procedure is especially useful for music as in Scratch the MIDI voicing (choice of instrument, and Volume) is maintained independently for each Sprite.

However, this also demonstrates the scale of the musical timing errors to which Scratch is liable, especially in the context of such concurrency. We have to remember that despite the parallel processing, all the note messages are eventually merged into a single MIDI stream that is then fed to the synthesis engine provided by the operating system (e.g. Media Player on Windows, Quicktime on Apple OS X, and Soundfont-based players on Linux).

The first version of this program simply ran all four verses together as a single looped routine, but the timing errors were extreme – the melody would finish very much earlier than the accompaniment. In the revised version presented here, *broadcast* is used to resynchronise melody and accompaniment at the end of each verse, so that timing deviations are confined to a single verse. In effect, the melody and accompaniment wait for each other to finish before starting the next verse. It almost goes without saying that such strategies are very standard in multi-threaded programming.

Even without formal musical training, our ears are very sensitive to such timing errors (which are far from subtle), whereas similar timing discrepancies in a GUI may be unremarked upon, if they are noticed at all. Arguably only the critical demands of music performance reveal such limitations in the software.

A revealing and even less subtle aspect becomes apparent as soon as any attempt is made to move the Scratch window while the program is running (it is sufficient simply to click on the title bar of the window). This completely suspends computation while the mouse key is held down. However, any notes that have started will continue to sound, since they are managed by the external MIDI synthesizer. A MIDI note is defined by a NOTE ON command followed (some user-defined time later) by a matching NOTE OFF command. If the latter is not received, the note plays for ever.

As noted above the program uses two sprites, to perform the accompaniment and the melody. Clicking ("mouse down") on one sprite suspends its performance, while allowing the other to continue. This will of course completely break the synchronisation; releasing the mouse resumes performance for that sprite. This "blocking" behaviour of the mouse is a consequence of the way Scratch is implemented.

Example 6: *madchords.sb*

One of the more significant limitations of Scratch is the absence of a simple way to play chords. Since the one available note command *play note* requires a duration, a sequence of notes naturally results in a monophonic pattern (melody). As shown in Example 5, to play multiple notes together we have to use *broadcast* to send concurrent "play" messages to multiple instances of *play note*[6]. This example emerged unexpectedly while experimenting. It is almost certainly an "undocumented feature" (i.e. a bug). It may however be very useful! This leads to a question strangely familiar to computer musicians – should the bug be fixed or not?

Starting the program runs the first code block, which sends multiple *play* messages in a short *repeat* block using *broadcast* (without *wait*). We would expect the result to reflect the specified duration value for each note – here a mere 0.1 beat – so either a rapid scale or arpeggio (i.e. fast successive notes), or a very short chord. Instead, all the notes except the final one sustain indefinitely.

From our knowledge of MIDI, we can reasonably speculate that somehow, the NOTE OFF messages that should terminate each note are (presumably on account of the speed of the multiple broadcasts) are being deleted or over-written, and hence not reaching the synthesizer. Scratch does not provide an explicit command to turn notes off, other than by terminating the whole script. Can we find a way to stop them programmatically?

The solution demonstrated in the script relies on some low-level knowledge of the MIDI specification. NOTE ON and NOTE OFF are defined as distinct MIDI messages, using different numbers. Attached to each message is a "Velocity" value indicating the desired volume of the note. Scratch does provide the *set volume* instruction. While the documentation does not go into such details, we can speculate that this command works by modifying the Velocity byte attached to the NOTE ON message. The MIDI number range is between 0 and 127 (which it appears Scratch has remapped to a simple percentage); and, as it happens, attaching a Velocity value of 0 to a NOTE ON message has the same effect as sending an explicit NOTE OFF message. This is entirely by design, and is stated in the official documentation for MIDI. The bottom block in the script exploits this to stop the notes by setting volume to 0 and broadcasting the same **play** messages as before. This block is activated when the space bar is pressed.

We are in somewhat unexpected territory here – this behaviour is undocumented, and for all we know is unknown even to the Scratch developers. Rather than learning and using Scratch to implement known algorithms and techniques, we are now experimenting with the operation of Scratch itself, and possibly also of the synthesiser. We are engaged in a research project! Possible questions include:

- How many notes can we play simultaneously?
- Is the effect stable – runs on all platforms and computers? (Or, put another way, is it "machine dependent" in any way?)

---

[6] There are many music examples on the Scratch site demonstrating the level of concurrency required. See e.g. http://scratch.mit.edu/projects/realSAB/685670

- Can we add other code to the broadcast block without breaking this behavior?
- Can we tidy up the chord – i.e. without the top note dropping out?
- Can we make this play a single note indefinitely?
- How far can we reasonably develop and exploit this in Scratch?
- Are there other commands in the Sound panel which may have related undocumented behavior? (Hint – test the *wait* command)

The key point about this exercise is that it is all speculation. Short of reporting the behavior to the developers and asking for an explanation, we can't be absolutely sure our speculation (or explanation) is correct. However it is a very reasonable speculation, and might well be helpful to the developers in tracking down the bug (if it is agreed to be a bug).

We have therefore now moved from being *users* of Scratch to *testers* of Scratch. Many users volunteer to be "beta testers" of software. They have a chance to try out and use the software before it is formally released, but accept that there may well be bugs or design flaws which need to be reported back to the developers as clearly as possible. They may also be in a privileged position to suggest changes or enhancements to the development team, and thus directly influence the final form of the product.

One longer-term consideration in this respect is the fact that Scratch 2 is under development and is being implemented using Flash[7]. The developers report that as Flash does not directly support MIDI sound synthesis, they will have to write their own synthesis engine. There is no reason to suppose that this "bug" will propagate to the new version. Or that it won't. The question is – do we want it to?

Richard Dobson (http://people.bath.ac.uk/masrwd/)
Composers Desktop Project (www.composersdesktop.com)

[7] See http://wiki.scratch.mit.edu/wiki/Scratch_2.0