

University Of Wales Trinity Saint David

Tileyard Education

MA Commercial Music Producer
2018/2019

Alessandro Favero
Student Number 1808638

Term 2, Development: Element 2, Part B

Module Lecturer: Helen Leigh

Introduction

In this essay we will discuss the design, production and employment possibilities of Spaceship Journey, a pre-written set of functions through coding in Sonic Pi.

Sonic Pi is a software-based live coding music synth (Sonic Pi website). The creator Sam Aaron designed it to be accessible to anyone that is willing to learn a few controls, with no need for real discipline. In fact, he encourages experimentation in the first place (Sonic Pi built-in manual, section 1).

The main principle of Sonic Pi is to have a blank canvas and a computer waiting for instructions. These instructions can be written, strictly in a form that the computer can interpret, with no time constraints. Sonic Pi contains built-in functions, sounds, samples and synths, that are all more or less music related. As soon as the code is ready, it can be run with a button on the screen or with a keyboard shortcut. The computer then starts executing the instructions and will ignore any input until the *Run* button is pressed again. When that happens, the previous run will not be interrupted, the computer will just execute the new script, which is the main principle that makes live coding possible. This means that instructions and functions can be updated and thus the sound being played back with them.

Part of the goal in producing the instrument for the Development Assignment is to achieve an instrument that can be played by anyone, including people that are not necessarily musical. The problem with Sonic Pi is that it is based purely on formally structured code. This means that either you know what the structures are, or you do not. Some functions are very intuitive to someone that knows the terminology of music synthesis, but even then the performance would end very quickly when the computer encounters a piece of code that does not know how to interpret and stop the run to prevent further random errors.

Spaceship Journey tries to work around that problem, and opens the doors for possibly unforeseen employment of the Sonic Pi environment.

Instructions for using Spaceship Journey are included in Appendix A. The product is designed so that instructions are as few as possible in order to make it work, but still leave the freedom to experiment.

The instrument

Spaceship Journey is a pre-written set of functions through coding in Sonic Pi. The main principle is that with minimal instruction, anyone can go through a musical journey, which is descriptive of a spaceship: the scenes are starting the journey, entering the spacecraft, starting the engine, launching it into space, travelling through space, arriving to the destination and landing. The level of possible customization of the journey depends on the players ability to code and use Sonic Pi.

Spaceship Journey sets a frame into which some musical scenes are launched by the player, and can be modified even further through live coding. In other words, it feels

interactive for players at any level. Some of the potential applications will be discussed at the end of this essay.

The process

The initial idea was not what became the end product. It included having a color palette of melodic, harmonic and rhythmic phrases and patterns that could always musically work together, much like Madeon's Adventure Machine (<https://www.madeon.fr/adventuremachine>), that could be triggered by a midi controller. The lack of advanced MIDI implementation capabilities of Sonic Pi and its predisposition to live coding made it much more complicated, and also a bit pointless, as it would have been an approximation of what could more easily and effectively be done with other software (e.g. C++, Ableton Live, etc).

In particular, there is no efficient way in Sonic Pi of pre-programming changes that are time dependant - it could be done only with complex loop structures that would affect the whole script, not just one sample - or by means of a MIDI controller, that would easily control live instruments but not as easily trigger new samples or new loops, like Ableton Live would.

The consequent step then was working around it, and instead of coming up with an idea first, trying to find Sonic Pi's unique attractive features and use them. Many friends and family members, musical and non musical, were put in front of a Sonic Pi demonstration. The result of this test was that the naked code in different colors, and the log and the oscilloscope made it look, to the majority of subjects, more real, pure and live, than other DAWs, which, as an analogy, resemble more of a piano, where you immediately understand what happens when you press the keys, but there is no way of seeing how it is actually happening until the cover is lifted. The analogy comes from the researcher's experience in piano teaching to children and adults, and the observation of the how students reacted with surprise, interest and excitement whenever the piano cover was lifted and they could observe the entire action of the sound-creation process. Another thing that leads to the sense of realism and thus interest is the possibility to shape the sound or the performance by changing parameters in normal written form, instead of the more music-related music notation or MIDI piano roll. The process of successfully change one parameter leads to the curiosity of trying to change another, and sets a challenge to the player that quickly starts learning and being rewarded with more and more satisfaction by the newly acquired skills. This is the foundation of Sonic Pi, as Sam Aaron writes in many occasions:

No matter how experienced you are - from a complete beginner to a seasoned live coder, you'll run some code that has a completely unexpected outcome. It might sound insanely cool - in which case run with it. However, it might sound totally jarring and out of place. It doesn't matter that it happened - what matters is what you do

next with it. Take the sound, manipulate it and morph it into something awesome.

The crowd will go *wild*.

(from section 1.3 “Learning through Play”, Aaron 2019)

In order to use the appealing features of Sonic Pi, then, Spaceship Journey was born. The player can transition between one scene and the next one.

In order to achieve that, different *buffers* are used. Each scene is coded inside each buffer. This allows for different parts of code to be executed when the player decides to. The player can also take the controls and change the parameters through live coding. In Spaceship Journey, that is steering the ship.

The code

In Sonic Pi there are two main ways of producing sound: synthesis and sample playback. Synthesis is achieved through basic functions like *play* followed by a number or a note expressed in scientific pitch notation, which both are played by an internal synthesizer, the sound of which can be determined by the command *use_synth*; sample playback is achieved with the command *sample* followed by the name of the samples included in the built-in Sonic Pi sound library, or the root path to an external audio file. Spaceship Journey use both of the technique. In particular, it uses synths for the melodic and harmonic parts, as different notes will always sound the same even if the player should decide to change the bpm. The rest of the sound journey is made of samples, as their role is not pitch-dependent, and the sound is more rich and complex. The reason why samples would change in case of a change in bpm is that most of them, in this program, are stretched to fit particular durations, expressed in beats. The *beat_stretch* algorithm of Sonic Pi is based on varispeed: the sample maintains all the original frames and it is simply played back at different speed to fit the specified duration, thus changing the pitch. Some of the external samples were recorded with a field recorder, some others came from Apple loops, some other from Noiiz (noiiz.com).

Playback-wise, the elements of the code can be divided in two broad categories: the one-shot events and the live loops. One shot events are triggered once each time the code is run, whereas live loops are triggered when the code is run, but repeat indefinitely, even if the code is rerun, until the *stop* explicit instruction is reached. One shots are used mainly as transitional sounds between one scene and the next one, whereas live loops build the majority of the actual music. The reason is more evident when the concept of *sync* is explained.

As any computer, the machine only executes what it is being instructed inside the code. This means that as soon as the *Run* button is pressed, the computer will execute the first line, then the next one, until the end of the script (that is unless there are looped structures). Whenever a second part of the script needs to be introduced and run, this will happen again, at the moment of pressing the button. Needless to say, this would produce the two scenes to be very easily out of time with each other.

The *sync* command is then crucial to build a performance based on many live loops that are activated at different times. Any loop can be synchronized to each other, but the structure of Spaceship Journey, where different loops start at different times and sometimes substitute each other, and often have different duration, synchronizing one loop with another is not the most reliable option. The most efficient option is to attach every loop to a common pulse in the background, generated by the function *cue*. When a cue is created, the computer generates cue points based on the instructions received. In the case of Spaceship Journey a cue point is generated every 16 beats (the default bpm is used in this case, which is 60, and the beats are equal to fourth notes). Once the cue is started, a function can be triggered by the cue points by using the *sync* function, that tells the current thread to wait to be executed until the next cue point is detected. Since the same cue function is used throughout the program, all the new threads of a new part of code with the embedded *sync* function will start at the same time. Every single thread of Spaceship Journey, excluding the very first one, that also starts the cue signal, is synced to the same internal cue points.

In total compliance with Sam Aaron principle's, the code was written bit by bit, keeping the ones that worked, and changing the ones that did not. This trial and error process, together with logical thinking, led to some mistakes. Some of them were fixed, some other were not, as they turned out to be good mistakes. The management of the *sync* function in particular is probably the most challenging one in Spaceship Journey. In fact, its line position is crucial, and determines different results when at different lines, with respect to the rest of the code. The reason being that the functions before it will be executed anyway; the functions following *sync* will have to wait until the next cue point. Some of these mistakes made some functions react all together in a new scene, but made some other react half or one entire loop later. The musical shift in changes on different instruments makes the Journey flow a bit more naturally and a bit less predictably, especially for first time players.

Another principle that helps things work, and players to modify what is already running, is that there can never be two live loops running with the same name (Sonic Pi's built-in tutorials by Sam Aaron). This allows live loop's properties to be changed in real time by giving new instructions about the same live loop, through its identifier (name). That is, of course, if the new instructions are compatible with the next ones - i.e. same parameter. If they are, old instructions will be updated. If new properties are added but old ones are not modified, the old ones will remain unchanged and will carry on indefinitely, while the new ones will be implemented independently. Take for example *live_loop :rain* inside Spaceship Journey. It is used in three different buffers in order to modify its properties at every scene: the first buffer starts it and give it properties like beat stretch and sample release (fade out); the second filters off the high frequency, the third one stops it altogether so that it does not interfere with the next scenes. This property is the reason why some live loops in Spaceship Journey are close to other ones, but they are terminated through the stop function, and a slightly modified version of them is started at the same time, with a different name. Changing all the compatible parameters and explicitly reset default ones would have been more complicated than copying and pasting a modified version of the previous one, with a different name.

A similar example to *live_loop :rain* are *live_loop :drumfull* and *live_loop :bipbip*.

The latter two are identical, if not for a new parameter that was added or changed in *:bipbip*: the parameter *amp*, which determines the volume of the sample being played. The sound changes entirely, as only one element is retained from the previous drum pattern. This is for a simple reason that includes a complex pattern, which is the main drums pattern in Spaceship Journey. It is one of the very few things that weren't coded from scratch, as it was taken by Sonic Pi's tutorials, where it is used as an example (section 5.2 of Sonic Pi's tutorial, paragraph 3, *Nesting Iterations*).

This is what it looks like:

```
live_loop :drumsfull do

  sample :ambi_lunar_land, amp: 0.6

  4.times do
    4.times do
      sample :drum_bass_hard
      2.times do
        sample :elec_blip2, rate: 2, amp: 0.8, pan: rrand(-0.8, 0.8)
        sleep 0.25
      end
      sample :drum_cymbal_closed
      sample :elec_snare
      4.times do
        sample :drum_tom_mid_soft
        sleep 0.125
      end
    end
  end
end
```

What happens there is there is a loop that repeats four times. Inside that loops there is another four times repetition. Inside that loop there is a two times repetition and another four times repetition, with different sleep times. The sleep times determines how long the computer waits for before going to the next instruction (in this case, since it is the last string of a loop block, it determines how long the computer waits for before restarting the loop or exiting it). The internal sleep times of $2 \times 0.25 + 4 \times 0.125$ makes 1 beat, which is the duration of the entire drum loop than repeats four times. That four-times-repeated drum loop is then repeated four times, to make it 16, which is a scene duration in Spaceship Journey. The reason to add that external four times loop is to be able to add the sample *ambi_lunar_land* every time the scenes starts again after 16 beats.

Since the drum loops relies on internal loops with different sleep times to create a pattern, when the journey gets to Buffer 6 and only the sample *elec_blip2* can be heard, the code of the rest of the loop cannot be eliminated, or the pattern would be changed. As a quick fix for this, the entire code is unchanged and all the samples are being played back. The substantial difference is that all the samples but *elec_blip2* are played back with their levels down to 0, which is complete silence. Another way of sparing some resources and code lines would be to bypass, by commenting, or delete the lines that play the sample back, leaving only the sleep function active. This would be more efficient, but

would undermine the player's possibility of interacting again with the new scene, by bringing back some samples and customizing their parameters at his or her will, steering the ship through the journey.

Ultimately, in order to achieve independent blocks that are executed simultaneously, the *in_thread do/end* blocks are extensively used throughout the program. The main application for these blocks is of allowing for simultaneous endless loops. In fact, a simple *loop do/end* block without further conditional instruction - e.g. *2.times* - once entered, cannot be exited because there are no instructions to do so. Furthermore, even if a condition was included, two independent loops that were written in sequence would be executed one after the other: the computer would run the first loop, complete it, exit the loop and enter the next one.

By wrapping the first loop in an *in_thread do/end* block we tell Sonic Pi to run the contents of the do/end block at *exactly* the same time as the next statement after the do/end block (which happens to be the second loop).

(Section 5.4 of Sonic Pi's tutorial, paragraph 2, *Threads to the Rescue*, Aaron 2019)

This is the reason why almost every piece of code in Spaceship Journey is embedded in its own *in_thread* block.

Main Issues

The majority of issues that were encountered in making Spaceship Journey were due to not understanding right away the purpose of Sonic Pi, and thus not foreseeing the lack of some features thereof, that one would usually expect from a digital audio workstation. As discussed above, the product had to be rethought and redesigned in order to employ the most characteristic features and capabilities, i.e. having an internal sound engine, supporting synthesis and sampling features, as well as the possibility to update instructions live.

One major setback due to the same misunderstanding of the Sonic Pi platform happened at the very end of the coding, while trying to fix a minor issue that was occurring on the last buffer: Sonic Pi's *Save* and *Load* functions are different from the majority of commercial software. The software does not create a file that can be opened in its entirety; it only saves, as plain text, the buffer that is displayed when the *Save* button is pressed. The same way, the *Load* function substitutes the code currently written on the buffer that is displayed when the *Load* button is pressed with the saved code. On the other end, whenever Sonic Pi is closed, it maintains the code in every buffer when opened again.

What happened is that, trying to fix a small part of code, many changes were made throughout the script, using the empirical approach suggested by Sam Aaron, knowing that everything could be set back to the last saved, working version. When changing the rest of the code failed, the last saved version was loaded, only to find out that the buffer that was displayed at the moment had been completely wiped and substituted with the script of a different buffer from the last saved version. Not only was the modified code not fixable, two buffers of working code were completely lost (the procedure was done twice as the first time it looked like it didn't work).

This time there were no happy mistakes, and the code of those buffers had to be re-figured out and rewritten entirely.

This is a clear example of Sonic Pi not being built for long lasting projects, but only for live applications. A buffer code can be saved so that many independent scripts can be saved in a folder for faster recall during a performance.

Results

After the whole process of designing, redesigning, addressing issues and changing direction many times, Spaceship Journey is a product that works but contains inefficiencies and flaws mainly caused by the partial incompatibility between the idea and Sonic Pi's coding environment. Buffers are interdependent, so restructuring would mean tearing down entire blocks and rewriting them. In other words, it is not an efficient template to effortlessly create different journeys from. The ship can be steered, parameters can be modified. Spaceship Journey thus works by itself and can be quite enjoyable, but does not represent a general baseline for replication. This can be considered a partial failure for an instrument, but not necessarily for a piece of art.

Possible applications

As discussed above, the end result of the project is not very efficient as an instrument, the main reason being that Spaceship Journey really is a project built inside and by means of an instrument. It resembles more of a composition (piece of art) for an orchestra (instrument). This is not a real issue, though, as instruments can be built or coded with more appropriate tools (e.g. Max MSP, Native Instruments' Massive). It makes sense then to consider, once again, what would make this project unique.

Applications for this project can vary very broadly, depending on the mind that sees fit. A couple of commercial applications will be considered to narrow down the possibilities. The principle of Sonic Pi being naked code, with visible logs, computational operations and analysis tools is not a common one these days in technology. A good example of that would probably be Apple computers, which manufactures extremely powerful and complex machines that work effortlessly thanks to a very complex but hidden software. The code is all there, but is invisible to the end consumer who has no interest but using it as a tool to make something else. The same goes for websites, little pieces of technologies, and even instruments, virtual or hardware, digital or analog.

Modular synths, exposed point-to-point hardware units and such are a rarity these days. One that is surrounded by curiosity of those who happen to see it. This is true of the NASA aesthetics as well, as well as every piece of technology that is created as a tool, and not a commercial product. The main reason why spacecrafts are not aesthetically refined is because of the possibility of manual control over them and because aesthetics are not of any concern to those who are making science and can invest money more efficiently. As Robert Frost, Instructor and Flight Controller at NASA says:

[...] But just like in my man-cave where once it is built, it's easier to run a cable down a wall rather than inside the wall, on ISS it isn't practical to remove all of the panels and pull out all of the racks to install new cables. There are also complexities involved in doing so with regards to safety analysis. The inside of a rack has thermal control equipment designed to handle a certain heat load. Randomly putting power supply block and power converters back there would derail any of the previous analysis. [...] (Frost, 2017)

Spaceship Journey tries to take advantage of the "exposed wires" aesthetics that shares with NASA, to create that sense of curiosity and interest.

It could be part of an interactive installation in a science museum, or an art exhibition like the Biennale, where the cockpit of a space shuttle is recreated and images are projected on the screen to emulate a journey through space. The computer in the cockpit could be an instance of Spaceship Journey, through which the player can steer the shuttle and create different musical landscapes. The sense of real-time realism of the log and the oscilloscope, and the live formal coding could bring music and art into a scientific concept, like outer space and interplanetary travels.

Also, thanks to Raspberry Pi, Spaceship Journey could be controlled via MIDI controllers inside the cockpit, to make it even more immersive.

The concept of art installations with interactive sound is not only already popular, but is actually particularly popular these days, in exhibitions like the Venice Biennale, Sydney Biennial, and art galleries like Vienna's Kunsthalle and Frankfurt's MMK (Buffenstein, 2016).

On a less artistic side, Spaceship Journey could be published as part of Sonic Pi's tutorials and examples, to show how limits of a live performance can be pushed towards a pre-set composition that still remains interactive.

References

Aaron, S. (2019). *Sonic Pi - Tutorial*. [online] Sonic-pi.net. Available at: <https://sonic-pi.net/tutorial.html> [Accessed 28 Mar. 2019].

Frost, R. (2017). *Why does the inside of the ISS and the space shuttle look so cluttered with wires and devices?*. [online] Quora. Available at: <https://www.quora.com/Why-does-the-inside-of-the-ISS-and-the-space-shuttle-look-so-cluttered-with-wires-and-devices> [Accessed 30 Mar. 2019].

Buffenstein, A. (2016). *12 Sound Artists Changing Your Perception of Art - artnet News*. [online] artnet News. Available at: <https://news.artnet.com/art-world/12-sound-artists-changing-perception-art-587054> [Accessed 3 Apr. 2019].

Appendix A

Instructions

Note: Since external samples were used, the player needs to modify the path inside the code before running it. Download the samples folder onto your desktop, and in the code, on the green lines, just change “alessandrofaverio” to your computer user name.

Open Sonic Pi. Go to buffer 0 and load “Spaceship 1 buffer 0.rb”, in the folder “Spaceship Journey/Buffers”.

Press the *Run* button or the shortcut cmd+R to run the scene.

When you are ready, go to buffer 1 and load “Spaceship 1 buffer 1.rb” in the same folder.

Press the *Run* button or the shortcut cmd+R once again.

Whenever you are ready to transition to the next thing, do the same with “Spaceship 1 buffer 2” on buffer 2, and so on.

Feel free to try and change bits to see what happens.

Enjoy!

Appendix B

Full Code

Buffer 0

#PRESS CMD+R TO START THE JOURNEY

```
with_fx :reverb, room: 1 do
  sample :ambi_lunar_land, amp: 0.5, beat_stretch: 16
  sample :ambi_drone, amp: 0.5, beat_stretch: 4, release: 8
  sample :ambi_drone, amp: 0.2, beat_stretch: 4, release: 12
  sample :ambi_sauna, amp: 0.2, beat_stretch: 4, release: 8
end
```

```
live_loop :fx do
  play :c2, release: 4
  play :c1, release: 4
```

```
  sleep 16
end
```

```
in_thread do
  loop do
    cue :sixtbars
    cue :raincue
    sleep 16
  end
end
```

```
live_loop :rain do
```

```
  with_fx :reverb, room: 1 do
    sample "/Users/alessandrofavero/Desktop/Sonic Pi
Project/CrispyRain.wav", beat_stretch: 18, release: 2#, lpf: 80

  end
  sync "/cue/raincue"
  ##| sleep 16
end
```

Buffer 1

```
in_thread do
  sync :raincue
  sample "/Users/alessandrofaverio/Desktop/Sonic Pi
Project/DoorClose.wav"
end

in_thread do
  live_loop :approach do
    sync :sixtbars
    with_fx :reverb, room: 1, mix: 0.5 do
      sample "/Users/alessandrofaverio/Desktop/Sonic Pi
Project/RainDropArp.wav", beat_stretch: 16, amp: 0.5, lpf: 80
    end
  end

  live_loop :rain do
    with_fx :reverb, room: 1 do
      sample "/Users/alessandrofaverio/Desktop/Sonic Pi
Project/CrispyRain.wav", beat_stretch: 17, release: 2, lpf: 100
    end
    sync :raincue
  end
end
```

Buffer 2

```
in_thread do

  in_thread do
    live_loop :approach do

      with_fx :reverb, room: 1, mix: 0.5 do
        sample "/Users/alessandrofavero/Desktop/Sonic Pi
Project/RainDropArp.wav", beat_stretch: 16, amp: 0.5, lpf: 80,
lpf_release: 4, release: 16#, attack: 10
      end
      sync :sixtbars
    end

    live_loop :rain do
      with_fx :reverb, room: 1 do
        sample "/Users/alessandrofavero/Desktop/Sonic Pi
Project/CrispyRain.wav", beat_stretch: 17, release: 16, lpf: 100#,
lpf_release: 4
      end
      sync :raincue
    end
  end

  in_thread do
    sync :raincue

    live_loop :chord2x do

      ##| 4.times do

        with_fx :echo,decay: 6 do
          with_fx :reverb, room:1, mix: 0.7 do
            play rrand(50, 95), amp: 0.1, pan: rrand(-1, 1)
          end
        end

        with_fx :reverb, room:1, amp: 0.7 do

          use_synth :fm
          play :c3, pan: 0, attack: 0.9, release: 1.5
```

```
play :eb, pan: -0.7, attack: 0.9, release: 1.5

in_thread do
  use_synth :sine
  play :g
end

use_synth :fm
play :bb, pan: 0.8, amp: 0.5, attack: 0.9, release: 1.5

sleep 2

play :bb2, pan: 0, attack: 0.9, release: 1.5

play :d, pan: -0.7, attack: 0.9, release: 1.5

in_thread do
  use_synth :sine
  play :f
end

use_synth :fm
play :bb, pan: 0.8, amp: 0.5, attack: 0.9, release: 1.5

sleep 2

play :g2, pan: 0, attack: 0.9, release: 1.5

in_thread do
  use_synth :sine
  play :d, pan: -0.7, attack: 0.9, release: 1.5
end

use_synth :fm
play :f

play :bb, pan: 0.8, amp: 0.5, attack: 0.9, release: 1.5
sleep 2
```

```
    play :ab2, pan: 0, attack: 0.9, release: 1.5

    in_thread do
      use_synth :sine
      play :eb, pan: -0.7, attack: 0.9, release: 1.5
    end

    use_synth :fm

    play :f
    play :bb, pan: 0.8, amp: 0.5, attack: 0.9, release: 1.5

  end

  sleep 2

end

end

end
```

Buffer 3

```
sync :raincue
sample "/Users/alessandrofaverio/Desktop/Sonic Pi
Project/TakeOff.wav", release: 2
sample "/Users/alessandrofaverio/Desktop/Sonic Pi
Project/Injection.wav", release: 1, amp: 0.8
sleep 16

in_thread do
  live_loop :approach do
    stop
    sleep 16
  end

  in_thread do
    live_loop :rain do
      stop
      sleep 16
    end
  end

end
end
in_thread do
  live_loop :drums do
    4.times do
      sample :drum_bass_hard
      sleep 0.5
    end
  end
end
end
```

Buffer 4

```
in_thread do
  sync :raincue
  live_loop :drums do
    stop
  end
end

in_thread do

  sync :raincue

  live_loop :drumsfull do

    sample :ambi_lunar_land, amp: 0.6

    4.times do
      4.times do
        sample :drum_bass_hard
        2.times do
          sample :elec_blip2, rate: 2, amp: 0.8, pan:
rrand(-0.8, 0.8)
          sleep 0.25

        end
        sample :drum_cymbal_closed
        sample :elec_snare
        4.times do
          sample :drum_tom_mid_soft
          sleep 0.125
        end
      end
    end
  end
end
end
```

Buffer 5

```
in_thread do
  sync :sixtbars

  live_loop :chord2x do
    stop
  end
end

in_thread do
  sync :raincue

  live_loop :chord2xRev do

    with_fx :echo,decay: 6 do
      with_fx :reverb, room:1, mix: 0.7 do
        play rrand(50, 95), amp: 0.1, pan: rrand(-1, 1), amp: 0
      end
    end

    with_fx :reverb, room: 1, mix: 1, amp: 0.2, lpf: 50 do

      use_synth :fm
      play :c3, pan: 0, attack: 0.9, release: 1.5

      play :eb, pan: -0.7, attack: 0.9, release: 1.5

      in_thread do
        use_synth :sine
        play :g
      end

      use_synth :fm
      play :bb, pan: 0.8, amp: 0.5, attack: 0.9, release: 1.5

      sleep 2

      play :bb2, pan: 0, attack: 0.9, release: 1.5
```

```

    play :d, pan: -0.7, attack: 0.9, release: 1.5

    in_thread do
      use_synth :sine
      play :f
    end

    use_synth :fm
    play :bb, pan: 0.8, amp: 0.5, attack: 0.9, release: 1.5

    sleep 2

    play :g2, pan: 0, attack: 0.9, release: 1.5

    in_thread do
      use_synth :sine
      play :d, pan: -0.7, attack: 0.9, release: 1.5
    end

    use_synth :fm
    play :f

    play :bb, pan: 0.8, amp: 0.5, attack: 0.9, release: 1.5
    sleep 2

    play :ab2, pan: 0, attack: 0.9, release: 1.5

    in_thread do
      use_synth :sine
      play :eb, pan: -0.7, attack: 0.9, release: 1.5
    end

    use_synth :fm

    play :f
    play :bb, pan: 0.8, amp: 0.5, attack: 0.9, release: 1.5

    end
    sleep 2

  end
end

```

Buffer 6

```
in_thread do
  sync :raincue
  live_loop :drumsfull do
    stop
  end
end

in_thread do
  sync :raincue
  live_loop :chord2xRev do
    stop
  end
end

in_thread do
  sync :raincue
  live_loop :bipbip do

    sample :ambi_lunar_land, amp: 0.6

    4.times do
      4.times do
        sample :drum_bass_hard, amp: 0
        2.times do
          sample :elec_blip2, rate: 2, amp: 0.8, pan:
rrand(-0.8, 0.8)
          sleep 0.25

          end
          sample :drum_cymbal_closed , amp: 0
          sample :elec_snare , amp: 0
          4.times do
            sample :drum_tom_mid_soft , amp: 0
            sleep 0.125
          end
        end
      end
    end
  end

in_thread do
  sync :raincue
```

```
live_loop :transition do
  with_fx :slicer, phase: 0.125 do
    sample :loop_drone_g_97, beat_stretch: 8
  end
  2.times do
    sample :loop_electric, beat_stretch: 4
    sleep 4
  end
  sleep 0
end
end
```

Buffer 7

```
in_thread do
  sync :raincue
  live_loop :transition do
    with_fx :slicer, phase: rrand(0.09, 0.651) do
      sample :loop_drone_g_97, beat_stretch: 8
    end
    2.times do
      sample :loop_electric, beat_stretch: 4
      sleep 4
    end
    sleep 0
  end
end
end
```

Buffer 8

```
in_thread do
  sync :raincue
  live_loop :transition do
    stop
    sleep 0
  end
end
```

```
in_thread do
  sync :raincue
  live_loop :bipbip do
    stop
    sleep 0
  end
end
```

```
in_thread do
  sync :raincue
  live_loop :fx do
    stop
    sleep 0
  end
end
```

```
in_thread do
  sync :raincue
  sample :loop_drone_g_97, beat_stretch: 16, release: 16
  with_fx :echo, decay: 16 do
    sample :loop_electric, beat_stretch: 4, release: 4
  end
end
sleep 16
```