Writ of possession california

Language: Suomexes in English svenska norsk Korean fran'ais Spanish catal'japanese chinese(trad) Chinese (simplified) davvisámegiella hindi magyar Introduction Sonic Pi is an open source programming environment, designed to explore and teach program concepts through the process of creating new sounds. It is a free live synthesizer programme for all, created by Sam Aaron in a computer lab at the University of Cambridge. You can use Sonic Pi for the program, composing and interpretation of a range of styles from classical to contemporary, from Canon to Dubstep. This tutorial will walk you through the basics and more Sonic Pi. At the end of this guide, you'll be able to create things like this: Or something like this: Sonic Pi is trying to explore. No mistakes, they're just discoveries. And most of all, it's a matter of good time. So remember: have fun, investigate and hack! Open Sonic Pi If you don't have Sonic Pi installed, visit sonic-pi.net, download and install. Available for Windows, OS X, and Linux operating systems. Then turn on Sonic Pi! Let's see what it's like. This is the Sonic Pi interface; has three main windows. The biggest is to write your code and we can call it the Program Board. There is also a registration board that displays information about your program when it starts. When you click the Help button at the top of the window, a third pane appears at the bottom that displays help documentation. It contains language information for Sonic Pi programming, as well as various sound synthesizers, sound examples, and more. There are also plenty of ready examples to try and use! Sonic Pi interface Let's touch note Let's start by programming Sonic Pi to play notes. Select Buffer 0 and type: play 60 Press Run from the top left corner. Can you hear the signal? Try different values. Write, for example, play 50 or play 70. How does the sound change? Now try typing pley 60 and press run. What's wrong? Here's an example of an error in your code. In the activities below, if the error panel displays text, you will know that you have an error that you need to fix. It could be that you wrote a bad word like game. The numbers you used are MIDI notes. MIDI is a convenient way to compose and is a handy tool for quickly testing notes and customizing them by reducing values (making your note lower) or increasing it (which increases height). Sonic Pi is familiar with numerical notation in MIDI (with values between 0 and 127) and traditional musical note (such as: :C4, :Eb3 or :G5). Page 2 Language: suomexes in English svenska norsk Korean fran'ais Español Japanese Chinese(trad) Chinese (Simplified) Davvisámegiella Hindi Magyar Introduction Write the following in buffer and press run: play 60 play 67 play 69 It didn't sound like a melody, did it?. Instead of playing them one after the other, Sonic Pi played all the notes at once (and so chords can really be written). If you want Sonic Pi to touch every note in a row, you need to tell the software to pause between notes. Try typing sleep 1 under each note like this: play 60 sleep 1 play 67 sleep 1 play 69 sleep 1 tells Sonic Pi to wait a while. You can try larger or smaller numbers. The lower the sleep value, the shorter the duration between the game commands and the opposite. If you are familiar with musical notes, so are the different musical notes in Sonic Pi: As we have said before, you can write notes in MIDI, which are basically numbers between 0 and 127 (67, 80, 22) or as musical notes (:G4, :Ab5, :Bb), as you prefer. Here we have a chart showing notes and their corresponding values in MIDI: Try using master scale C notes (72, 74, 76, 77, 79, 81, 83, or :C5 :D 5:E5 :F5 :G5 :A5 :B5) to create a melody. Use sleep with different values to distinguish between paces. At first, use_bpm can be added to make your melody faster or slower. The acronym BPM comes from Beats Per Minute (pulses per minute). For example: use_bpm 120 play 72 sleep 0.25 play 76 sleep 0.25 play 76 sleep 0.25 play 72 sleep 0.5 play 83 sleep 0.25 play 74 sleep 0.25 play 83 sleep 0.25 play 79 sleep 0.25 play 84 Listen to the example above Now make your own tune! Welcome to Sonic Pi. I hope you're as excited to start making crazy noises as I am to show you. It's going to be a really fun ride where you learn all about music, synthesis, programming, composition, performance and more. But wait, how rude of me! Let me introduce myself - I'm Sam Aaron - the guy who created Sonic Pi. You can find me @samaaron twitter and I'm happy to welcome you. You might also be interested to learn more about my live coding performances where I code with Sonic Pi live in front of an audience. If you have any thoughts, or ideas to improve Sonic Pi - please pass them on - the feedback is so useful. You never know, your idea could be the next big feature! This guide is divided into group sections by category. Although I wrote it to make simple progress in learning from start to finish, feel free to conspire and exit the sections as you see fit. If you think something's missing, let me know and I'll consider it for the future version. Finally, watching other live code is a really great way to learn. I regularly stream live on so please come by, say hello and ask me questions :-) Okay, let's start... 1.1 - Live Coding One of the most exciting aspects of Sonic Pi is that it allows you to write and change the code live to make music, just as you could perform live with a guitar. This means that given some practices you can take Sonic Pi on stage and concert with him. Free your mind Before we go into real detail about how Sonic Pi works in the rest of this guide, I would like to give you an experience of what it's like to live code. Don't worry if you don't understand much (or any) of this. Just try to keep your seats and enjoy... Live loop Let's start, copy the following code to the empty clipboard above: live_loop:flibble to sample :bd_haus, rate: 1 sleep 0.5 end Now, press the Run button and you'll hear a nice fast bass drum hitting. If you want to stop the sound at any time, just press the Stop button. Although don't guess yet... Instead, follow these steps: Make sure the bass drum sound is still running Change sleep value from 0.5 to something more like 1. Press the Run button again Notice how the drum speed has changed. Finally, remember this moment, this is the first time you've lived coded with Sonic Pi and it's unlikely to be your last... Okay, that was simple enough. Let's add something else to the mix. Above the pattern :bd_haus add a line pattern :ambi_choir, rate: 0.3. Your code should look like this: live_loop:flibble up sample :ambi_choir, rate: 0.3 sample :bd_haus, rate: 1 sleep 1 end Now, play around. Change rates - what happens when you use high values, or low values or negative values? See what happens when you change the rate: the value for :ambi_choir sample is only slightly (say at 0.29). What happens if you choose a really small sleep value? See if you can do it so quickly that your computer will stop making an error because it can't keep up (if it does, just select a higher sleep time and press Run again). Try to comment on one of the sample lines by adding # to the beginning: live_loop:flibble to sample :ambi_choir, rate: 0.3 # sample :bd_haus, rate: 1 sleep 1 end Notice how he tells the computer to ignore it, so we don't hear it. It's called a comment. At Sonic Pi, we can use comments to remove and add things to the mix. Finally, let me leave you something fun to play with. Take the code below and copy it to the backup clipboard. Now, don't try to understand it too much except to see that there are two loops - so two things go around at the same time. Now, do what you do best - experiment and play. Here are some suggestions: Try changing the blue speed: values to hear the pattern change. Try to change your sleep time and hear that both loops can spin at different rates. ussh off the pattern line (remove #) and enjoy the sound of the guitar playing backwards. Try changing any of the blue blends: values for between 0 (not in the mixture) and 1 (completely in the mixture). Remember to press Run and you'll hear a change the next time the loop is rounded. If you end up in a sour pickle, don't worry - hit Stop, delete the code on the clipboard and pour a fresh copy and you're ready to get stuck again. Making mistakes is how you learn the fastest... live_loop :guit to with_fx :echo, mix: 0.3, stage: 0.25 to sample :guit_em9, rate: 0.5 end # pattern :guit_em9, rate: -0.5 sleep 8 end live_loop :boom to with_fx :reverb, room: 1 to sample :bd_boom, amp: 10, foot: 1 end of sleep 8 end Now, keep playing and experimenting until your curiosity about how it all actually works begins and you start to wonder what else you can do with it. Now you are ready to read the rest of the tutorial. What are you waiting for... 1.2 - Sonic Pi Interface Sonic Pi has a very simple interface for coding music. Let's spend some time researching it. A - Play Controls B - Editor Controls C - Info and Help D - Code Editor E - Prefs Panel F - Log Viewer G - Help System H - Scope Viewer A. Play Controls These pink keys are the main controls for starting and stopping sounds. There is a Run button to run the code in the editor, Stop stopping all boot codes, Save to save code to an external file, and Record to create a recording (WAV file) of playing sound. B. The editor controls these orange buttons allow you to manipulate the code editor. Size + and size - buttons allow the text to be larger and smaller. These blue buttons give you access to information, help and preferences. The Info button opens an information window that contains information about Sonic Pi itself - the core team, history, coworkers and community. The Help button includes Help (G), and the Prefs button includes a settings window that lets you manage some basic system parameters. D. Code Editor This is the area where you will write your code and compose/perform music. It is a simple text editor where you can write code, delete it, cut and paste, etc. Think of it as a very basic version of Word or Google Docs. The editor will automatically color the words based on their meaning in the code. This may seem strange at first, but it will soon be very useful to you. For example, you know something's a number because it's blue. E. The Sonic Pi Prefs Panel supports a number of tweaked preferences that can be accessed by attaching the prefs button in the Info and Help button set. This will change the visibility of the Prefs panel which includes a number of options that need to be changed. Examples are forcing mode, stereo increment, toggling log output transparency, and also a volume slider and audio selector on Raspberry Pi. F. Log Viewer When you run your code, information about what the program does will be displayed in the viewer log. By default, you will messages for each sound you create with the exact start time of the sound. This is very useful for correcting code correction and understanding what your code does. Mr. Help System One of the most important parts of the Sonic Pi interface is the help system that appears at the bottom of the window. This can be turned on and off by clicking the blue Help button. The help system contains help and information on all aspects of Sonic Pi including this tutorial, a list of available synths, samples, examples, FX and a complete list of all the functions Sonic Pi provides for music encoding. H. Scope Viewer scope allows you to see the sound you hear. You can easily see that the wave of the saw looks like a saw and that the basic beep is a curved sinus wave. You can also see the difference between loud and silent sounds by line size. There are 3 scopes for the game - the default setting is the combined scope for the left and right channels, there is a stereo scope that attracts a separate scope for each channel. Finally, there is a Lissajous curve scope that will show the phased relationship between the left and right channels and allows you to draw beautiful images with sound (. 1.3 - Learning through Play Sonic Pi encourages you to learn about computing and music through play and experimentation. The most important thing is to have fun, and before you realize it you will accidentally learn to code, compose and perform. No mistakes While we're on this topic, let me give you just one piece of advice I've learned over the years of live kodiing with music - no mistakes, just opportunities. It's something I've often heard in relation to jazz, but it works just as well with live coding. No matter how experienced you are - from a complete beginner to an experienced 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 may sound completely jarring and out of place. It doesn't matter what happened - it's what you do next with it that matters. Take the sound, manipulate it and turn it into something awesome. The crowd will go wild. Start simply when you're studying, it's tempting now to want to do amazing things. However, just hold that thought and look at it as a distant goal to achieve later. For now, instead of thinking about the simplest thing you could write that would be fun and useful, it's a small step towards the amazing thing you have in your head. Once you have an idea of this simple step, try to build it, play with it and then see what new ideas it gives you. Soon you will be too busy with fun and real progress. Just make sure you share your work with others! 2 - Synths OK, quite a few introductions - let's go into some sound. In section we will cover the basics of starting and manipulating synths. Synth is short for synthesizer which is a fancy word for something that makes sounds. Usually synthesizers are quite complicated to use - especially analog synthesizers such as Eurorack modules connected by a mess of wires. However, Sonic Pi gives you a lot of that power in a very simple and affordable way. Don't be fooled by the immediate simplicity of the Sonic Pi interface. You can get very deep into very sophisticated sound manipulation, if that's your thing. Hold on to your hats... 2.1 - Your first beeps See the following code: play 70 This is where it all starts. Go ahead, copy it and pour it into the code window at the top of the app (large white space under the Run button). Now, press Run... Beep! Intense. Press him again. And again. And again... Woah, crazy, I'm sure he could keep working all day. But wait, before you get lost in the endless course of beeps, try changing the number: play 75 Can you hear the difference? Try a lower number: play 60 So the lower numbers make the beeps set lower, and the larger numbers make the larger the beps set. Just like on the piano, the keys on the lower part of the piano (left side) play the lower notes, and the keys on the higher part of the piano (right side) play higher notes. In fact, the numbers actually refer to notes on the piano. Playing 47 actually means playing the 47th piano note. Which means game 48 is one note up (next note on the right). It just so happens that on October 4th, 1945, Go ahead and play: Play 60. Don't worry, if this doesn't mean anything to you, it didn't mean anything to me when I started out. All that matters now is that you know that low figures make for lower beps and high numbers make for bigger beps. Chords Playing notes is pretty fun, but playing many at the same time can be even better. Try: Play 72 play 75 to play 79 Jazzy! So when you write multiple plays, they play everything at the same time. Try it yourself - which numbers together sound good? Which ones sound terrible? Experiment, explore and find out for yourself. Melody So playing notes and chords is fun - but how about a melody? What if you want to play one note after another and not at the same time? Well, it's easy, you just need to sleep between notes: play 72 sleep 1 play 75 sleep 1 play 79 How nice, a little arpeggio. So, what does 1 mean in sleep 1? That means the duration of sleep. It actually means sleeping for one rhythm, but for now we can think of it as sleeping for 1 second. So, what if we wanted to make our arpeggio a little faster? Well, we have to use shorter sleep values. What about half and 0.5, respectively: play 72 sleep 0.5 play 75 sleep 0.5 play 79 Notice how the game plays faster. Now, try it yourself, change the time - use different times and notes. One thing to try is notes such as game 52.3 and play 52.63. There is no need to stick to the standard whole notes. Play and have fun. Traditional note names For those of you who already know some musical notation (don't worry if you don't - you don't need it to have fun) you might want to write a melody using note names like C and F# instead of numbers. Sonic Pi is covering for you. You can do the following: play :C sleep 0.5 play :D Sleep 0.5 game :E Don't forget to put your colon: in front of your name notes so it goes pink. You can also specify an octave by adding a number by note name: play :C3 sleep 0.5 play :D 3 sleep 0.5 play :E4 If you want to make a note sharp, add with after-name notes such as play :Fs3 and if you want to make a note flat, add b such as play :Eb3. Now you go crazy and have fun making your own songs. 2.2 - Synth Options: Amp and Pan, as well as allowing you to control which note to play or which pattern to run, Sonic Pi provides a full range of options for making and controlling sounds. We will cover many of them in this guide and there is extensive documentation for everyone in the assistance system. However, for now we will present two of the most useful: amplitude and pan. First, let's look at what the options really are. Options Sonic Pi supports the term option (or opts for short) for your synths. Opts are the controls you pass for gaming that change and control aspects of the sound you hear. Each synth has its own set of opts for fine-tuning their sound. However, there are common sets of opts shared by many sounds such as amp: and envelope opts (covered in the second section). Opts have two main parts, their name (control name) and their value (the value to which you want to set the control). For example, you may have an opt called cheese: and you want to set it with a value of 1. Optovi are transferred to play calls using a comma and then the name opta such as amp: (do not forget the colon :) and then the space and value of the opta. For example: play 50, sir: 1 (Have this cheese for example: not a valid opt, we just use it as an example). You can pass multiple opts by separating with a comma: play 50, cheese: 1, beans: 0.5 The order of the charges does not matter, so it is identical: play 50, beans: 0.5, cheese: 1 Optovi that are not recognized synthom are just ignored (like cheese and beans that are obviously funny opt names!) If you accidentally use the same opt twice with different values, the last one wins. For example, beans: here it will have a value of 2, not 0.5: play 50, beans: 0.5, cheese: 3, eggs: 0.1, beans: 2 Many things in Sonic Pi accept optoves, so just spend a little time learning how to use them and you will be set! Let's play with our first opt: amp:. Amplitude Amplitude is a computational representation of the loudness of sound. High amplitudes loud sound and low amplitude produces a silent sound. Just as Sonic Pi uses numbers to represent times and notes, it uses numbers that represent amplitudes. Amplitude of 0 is silent (you will not hear anything), while amplitude of 1 is of normal volume. You can even amplitude more to 2, 10, 100. However, you should note that when the overall amplitude of all sounds becomes too high, Sonic Pi uses what is called a compressor to crush them all to make sure things are not too loud for your ears. This can often make the sound muddy and strange. Therefore, try using low amplitudes, that is, ranging from 0 to 0.5 to avoid compression. Amplify it To change the sound amplid, you can use the amplifier: make up your mind. For example, play at half amplitude pass 0.5: play 60, amp: 0.5 Play on double ampplitude pass 2: play 60, amp: 2 Amplifier: decide just changing the call to play is associated with. So, in this example, the first call for the game is at half the volume, and the second returns to the default (1): play 60, amp: 0.5 sleep 0.5 play 65 Course, You can use different amplifiers: values for each game call: play 50, amp: 0.1 sleep 0.25 play 55, amp: 0.2 sleep 0.25 play 57, amp: 0.4 sleep 0.25 play 62, amplifier: 1 Scroll Another fun to decide to use is pan : which controls the movement of sound into the stereo. Moving the sound to the left means you hear it from the left speaker, and scrolling to the right means you hear it from the right speaker. For our values, we use -1 to represent the entire left, 0 to represent the center, and 1 to represent the entire right in the stereo field. Of course, we are free to use any value between -1 and 1 to control the exact positioning of our sound. Let's play the beep from the left speaker: let's play 60, pan: -1 Now, let's let it out of the right speaker: let's play 60, pan: 1 Let's finally play it back from the center of both (default position): let's play 60, pan: 0 Now, go and have fun changing amplitudes and moving your sounds! 2.3 - Switching Synths So far we've had plenty of fun making beeps. However, you are probably starting to get bored with the basic beep noise. Is that all Sonic Pi has to offer? Surely there's more to live coding than just playing beeps? Yes there is, and in this section we will explore some of the exciting ranges of sounds that Sonic Pi has to offer. Synths Sonic Pi has a number of different instruments called synths (which is short for synthesizers). While the patterns represent prerecorded sounds, synths are capable of generating new sounds depending on how you control them (which we will explore later in this guide). Sonic Pi synths are very powerful and expressive and you will have a lot of fun exploring and playing with them. First, let's learn how to choose the current synth to use. Buzzy chicken and prophets Fun sound is wave saw -- use_synth :saw play 38 sleep sleep Play 50 sleep 0.25 play 62 Let's try another sound - Prophet: use_synth :p horn play 38 sleep 0.25 play 50 sleep 0.25 play 62 How about combining two sounds. First one after the other: use_synth :saw play 38 sleep 0.25 play 50 sleep 0.25 use_synth :p rofet play 57 Now more sounds at the same time (not sleeping between consecutive Game calls): use_synth:tb303 play 38 use_synth :d in the picture play 50 use_synth :p rofet play 57 Notice that the use_synth command only affects the following play calls. Think of it as a big switch - new game calls will be played by any synth it currently points to. You can move the switch to a new synth with use_synth. Discover synths to see which synths Sonic Pi has for you to play with see the Synths option in the menu at the bottom of this Help screen (between examples & Fx). There are more than 20 to choose from. Here are some of my favorites: :p can :d with had:fm :tb303 :p and get toy with switching synths during your music. Have fun combining synths to make new sounds, as well as using different synths for different parts of your music. 2.4 - Envelope duration In the earlier section, we looked at how we can use the sleep command to control when to start our sounds. However, we have not yet been able to control the duration of our sounds. To get a simple but powerful means of controlling the duration of our sounds, Sonic Pi provides the term ADSR amplitude envelopes (later in this section we will cover what ADSR means). Amplitude Envelope offers two useful aspects of the control: control over the duration of sound control over audio amplitude Duration is the length of sound lasts. Longer duration means you hear a sound longer. Sonic Pi sounds all have a controlled amplitude envelope, and the total duration of that envelope is the duration of the sound. Therefore, by controlling the envelope, you control the duration. The ADSR amplitude envelope not only controls the duration, but also gives you fine control over the sound amplitude. All sound sounds begin and end quietly and contain some non-silent part in between. Envelopes allow you to slide and hold the amplitude of non-silent parts of the sound. It's like giving someone instructions on how to turn up and lower the volume of the guitar amplifier. For example, you can ask someone to start with silence, slowly move to full volume, hold it a little, and then quickly fall back into silence. Sonic Pi allows you to program exactly this behavior with envelopes. Just to repeat, as we have already seen, amplitude of 0 is silence and amplitude of 1 is a normal volume. Now let's look at each of the envelope parts. Release stage The only part of the envelope used by default is the release time. This is the time it takes to a sound that will fade. All synths have a release time of 1 which means that by default they have a duration of 1 beat (which is at the default BPM of 60 is 1 second): play 70 Notes will be heard 1 second. Go ahead and time :-) This is a short hand for a longer more explicit version: play 70, release: 1 Notice how it sounds exactly the same (the sound lasts one second). However, it is now very easy to change the duration by changing the release value: decide: play 60, release: 2 The sound of synths we can make a very short time using a very small release time: play 60, release: 0.2 The release duration of the sound is called the release phase and by default is a linear transition (i.e. straight line). The following diagram illustrates this transition: The vertical line on the far left side of the diagram shows that the sound starts at 0 amplitude, but immediately goes to the full amplitude (this is the attack phase that we will cover next). Once in full amplitude, then moves in a straight line down to zero taking the amount of time specified by the release:. Longer release times produce longer synth fade outs. Therefore, you can change the sound duration by changing the release time. Have a game that adds release time to your music. Attack phase By default, the attack phase is 0 for all synths, which means they immediately range from 0 amplitude to 1. It gives the synth a starting percussion sound. However, you may want to fade your sound in. This can be achieved by attacking: make up your mind. Try to fade into some sounds: play 60, offense: 2 sleep 3 play 65, offense: 0.5 You can use multiple opts at a time. For example, for a short attack and a long release try: play 60, attack: 0.7, release: 4 This short attack and long release envelope are illustrated by the following diagram: Of course, you can change things. Try long attack and short release: play 60, offense: 4, release: 0.7 Finally, you can also have short attacks and release times for shorter sounds. Game 60, offense: 0.5, release: 0.5 Maintenance phase In addition to determining the time of attack and release, you can also specify the maintenance time to control the maintenance phase. This is the time for which sound is maintained in full amplitude between the attack and release phase. Game 60, offense: 0.3, maintenance: 1, release: 1 Maintenance time is useful for important sounds that you want to give a full presence in the mix before entering the optional release phase. Of course, it is completely valid to set and attack: and release: it is decided by 0 and just use maintenance so that you absolutely do not fade or fade to sound. However, be warned, release 0 can produce clicks in sound and it is often better to use a very small value such as 0.2. Decay Phase For an additional level of control, you can also specify the time of decay. This is the envelope phase that corresponds to phase of attack and maintenance and determines the time at which amplitudes will fall from attack_level: to decay_level: (which, unless explicitly set, will be set to sustain_level:). By default, decay: opt for 0, and the levels of attack and maintenance are 1, so you will need to determine them for the time of decay to have any effect: play 60, attack: 0.1, attack_level: 1, decay: 0.2, sustain_level: 0.4, maintenance: 1, release: 0.5 Level of dissolution One last trick is that although decay_level: decide the default values to be the same values as sustain_level : You can explicitly set them to different values for full control of the envelope. This allows you to create envelopes such as the following: game 60, attack: 0.1, attack_level: 1, decay: 0.2, decay_level: 0.3, maintenance: 1, sustain_level: 0.4, Release: 0.5 It is also possible to set decay_level: be higher than sustain_level:: play 60, attack: 0.1, attack_level: 0.1, decay: 0.2, decay_level:1, maintenance: 0.5, sustain_level: 0.8 , issue: 1.5 ADSR envelopes So concise Sonic Pi's ADSR envelopes have the following stages: attack - time from 0 amplitude to attack_level, decay - it's time to move amplitude from attack_level to decay_level, maintenance - time to move amplitude from decay_level to sustain_level, release - time to move amplitude from sustain_level to 0 It is important to note that the duration of the sound is the sum of the time of each of these stages. Therefore, the following sound will have a duration of 0.5 + 1 + 2 + 0.5 = 4 beats: game 60, attack: 0.5, attack_level: 1, decay: 1, sustain_level: 0.4, maintenance: 2, release: 0.5 Now go and have a show that adds envelopes to your sounds... 3 - Patterns Another great way to develop music is to use prerecorded sounds. In the great hip-hop tradition, we call these prerecorded patterns of sounds. So if you take out the microphone outside, go and record a slight sound of rain hitting the canvas, you just created a pattern. Sonic Pi allows you to do a lot of fun things with patterns. Not only does a ship with 130 samples of public domain ready for you to get stuck with, it allows you to play and manipulate your own. Let's get to it... 3.1 - Starting patterns playing beeps is just the beginning. Something that's a lot of fun triggers prerecorded samples. Try: the pattern :ambi_lunar_land Sonic Pi includes many patterns to play from. You can use them just as you use the game command. To play multiple patterns and notes just write them one after the other: play 36 play 48 sample :ambi_lunar_land pattern :ambi_drone If you want to space them in time, use the sleep command: sample :ambi_lunar_land sleep 1 play 48 sleep 0.5 play 36 sample :ambi_drone sleep 1 play 36 Notice how Sonic Pi does not wait for the sound to end before the next sound begins. Sleep command describes separating sound activation. This allows you to Although layer sounds together creating interesting overlapping effects. Later in this guide we will look at controlling the duration of sounds with envelopes. Sample detection There are two ways to detect the range of samples in Sonic Pi. First, you can use this help system. Click the patterns in the menu at the bottom of this Help screen, select your category, and then you'll see a list of available sounds. Alternatively, you can use the automatic completion system. Simply type the beginning of a group of patterns such as: pattern :ambi_ and you will see a drop in the name of the selection patterns. Try the prefixes of the following category: ambi_ :bass_ :elec_ :p erc_ :guit_ :d rum_ :misc_ :bd_ Now start mixing patterns into your compositions! 3.2 - Sample parameters: Amp and Pan As we have seen with synths, we can easily control our sounds with parameters. The samples support exactly the same parameterization mechanism. Let's go again our friends amp: and pan:. Amplification patterns You can change the amplitudity of samples with exactly the same approach that you used for synthesizers: sample :ambi_lunar_land, amp: 0.5 Scroll patterns We are also able to use the pan: parameter on the samples. For example, here's how we would play an amen break in the left ear, then midway through the game through the right ear: pattern :loop_amen, pan: -1 sleeping 0.877 pattern :loop_amen, pan: 1 Keep hinting that 0.877 is half the sample :loop_amen per second. Finally, note that if you set some synth default settings use_synth_defaults (which we will discuss later), they will be ignored by the pattern. 3.3 - Stretching patterns Now that we can play a variety of synths and patterns to create some music, it's time to learn how to modify both synths and patterns to make music even more unique and interesting. First, let's explore the ability to stretch and crush patterns. Patterns of patterns are prerecorded sounds stored as numbers representing how to move the sound playback cone of the speaker. The cone of speakers can move in and out, so the numbers just need to represent how far the cone and cone should be for every moment in time. To be able to faithfully play recorded audio, the pattern usually needs to store thousands of numbers per second! Sonic Pi takes this list of numbers and feeds them at the right speed to move your computer's speaker in the right way to play audio. However, it's also fun to change the speed at which numbers feed on the speaker to change the sound. Changing Rate Let's play with one of the ambient sounds: :ambi_choir. To play it with the given speed, you can pass the rate: opt for a pattern: pattern :ambi_choir, rate: 1 It plays it at normal speed (1), so nothing special yet. However, we are free to change that number to something else. How about 0.5: sample :ambi_choir, rate: 0.5 What's going on here? Well, two things. First, the pattern takes twice as long to play, secondly, the sound is octate lower. Let's explore these things in a little more detail. Let's stretch a pattern that's fun to stretch and compress with Amen Break. At normal speed, we can imagine throwing in a drum 'n' bass song: sample :loop_amen however, by changing gear we can switch genres. Try half the speed for old school hip-hop: sample :loop_amen, rate: 0.5 If we accelerate it, we enter jungle territory: sample :loop_amen, rate: 1.5 Now for our final party trick - let's see what happens if we use a negative rate: sample :loop_amen, rate: -1 Woah! He's playing backwards! Now try to play with a lot of different patterns at different rates. Try very fast prices. Try insanely slow prices. See what interesting sounds you can make. A simple explanation of the sample rate A useful way to think about patterns is like a spring. The rate of reproduction is like squashing and stretching spring. If you are playing a pattern at a rate of 2, crush the spring to half the normal length. The pattern therefore takes half the time to play because it is shorter. If you play a half-rate pattern, stretch the spring to double its length. The sample therefore takes twice as much time to play as it is longer. The more you squash (the higher the rate), the shorter it is, the more you stretch (the lower the rate), the longer it becomes. The compression of the spring increases its density (the number of coils per cm) - this is similar to a pattern that sounds higher. Stretching the spring reduces its density and is similar to a sound that has a lower height. The math behind the sample rate (This section is provided for those interested in detail. Feel free to skip it...) As we have seen above, the pattern is represented by a large long list of numbers representing where the speaker should be over time. We can take this list of numbers and use it to draw a chart that would look similar to this: You may have seen pictures like this before. It's called the waveform of the pattern. It's just a graph of numbers. Usually a waveform like this will have 44100 data points per second (this is due to Nyquist-Shannon sampling theorem). So, if the sample lasts 2 seconds, the waveform will be represented by 88200 numbers that we would feed the speaker at a rate of 44100 points per second. Of course, we could feed him at twice the speed that would be 88200 points per second. So that would only take one second to get back. We could also play it back to half a rate which would be 22050 points per second, taking 4 seconds to play back. Sample duration is influenced by playback rate: Doubling the playback rate halves playback time, halving the playback rate doubles playback time, using a playback rate of one-quarter quadruples Time, Using a playback rate of 1/10 makes playback the last 10 times longer. We can present this with the new_sample_duration: new_sample_duration = (1 / foot) * sample_duration change in the rate of reproduction also affects the height of the sample. The frequency or height of the waveform is determined by how fast it moves up and down. Somehow, our brains turn the speaker's rapid movement into high notes and the slow movement of the speakers into low notes. This is why sometimes you can even see a large bass speaker moving as it pumps out super low bass - it actually moves much slower than a speaker that produces higher notes. If you take a waveform and squash it, it will move up and down multiple times per second. It'll make it sound more. It turns out that doubling the amount of motion up and down (oscillations) doubles the frequency. So playing the sample at twice the speed will double the frequency you hear. Also, halving the rate will halve the frequency. Other rates will affect frequency accordingly. 3.4 - Envelope samples It is also possible to modify the duration and amplitudes of the sample using the ADSR envelope. However, this works somewhat differently from the ADSR envelope available on synths. Sample envelopes only allow you to reduce amplitudes and sample duration - and never increase it. The pattern will stop when the pattern is finished playing or the envelope is completed - whichever is first. So, if you use a very long edition:, it will not extend the duration of the sample. Amen Envelopes Let's get back to our trusted friend Amen Break: sample :loop_amen No studs, we hear the whole pattern in full amplitude. If we want to fade this in more than 1 second we can use the attack: param: pattern :loop_amen, attack: 1 For shorter fade, choose a shorter attack value: pattern :loop_amen, attack: 0.3 Auto Sustain Where the behavior of the ADSR envelope differs from the standard synth envelope is in permanent value. In a standard synth envelope, maintenance was set to 0 unless you set it manually. With patterns, the permanent value is set to the automagic value - there is time left to reproduce the rest of the sample. That's why we hear the whole pattern when we don't go through any default settings. If the values of attack, decay, maintenance and firing were 0, we'd never hear a voice. Sonic Pi therefore calculates how long the sample is, subtracts any attack, decay and release time and uses the result as your maintenance time. If the values of attack, decay and release are added together more than the duration of the sample, maintenance is simply set to 0. Fade Outs To investigate this, let's consider a more detailed interruption of our Amen. If we ask Sonic Pi how long the pattern is: print sample_duration :loop_amen Will print 1.753310657596372 which is the length of the sample per second. Let's round it off to 1.75 for convenience here. Now, if we set the release to 0.75, Surprisingly it will happen: sample :loop_amen, release: 0.75 Will play the first second of the sample in full amplitude before then fading in a period of 0.75 seconds. This is auto maintenance in action. By default, the release always works from the end of the pattern. If our sample was 10.75 seconds long, it would have played the first 10 seconds in full amplitude before fading over 0.75s. Remember: by default, release: fades at the end of the pattern. Fade In and Out We can use both attack: and release: along with automatic maintenance behavior to fade both in and out of the sample duration: sample :loop_amen, attack: 0.75, release: 0.75 How the full duration of the sample is 1.75s, and our phases of attack and release are added up to 1.5s, maintenance is automatically set to 0.25s. This allows us to easily fade the pattern. Explicit maintenance We can easily return to our usual synth ADSR behavior by manually setting a permanent one: up to a value such as 0: sample :loop_amen, maintenance: 0, release: 0.75 Now, our sample game only 0.75 seconds in total. With default attack: and decay: at 0, the pattern jumps straight into full amplitude, is held there 0s, and then released back to 0 amplitude during the release period - 0.75s. Percussion cimbali We can use this behavior for good effect to turn longer sound patterns into shorter, percussion versions. Consider the pattern :d rum_cymbal_open: sample :d rum_cymbal_open you can hear the sound of cymbals ringing over a period of time. However, we can use our envelope to make it percussionist: a pattern :d rum_cymbal_open, Attack: 0.01, maintenance: 0, release: 0.1 Then you can mimic the hitting of cymbals and then clump it by increasing the maintenance period: pattern :d rum_cymbal_open, attack: 0.01, maintenance: 0.3, release: 0.1 Now go and have fun putting envelopes over samples. Try to change the rate too much for really interesting results. 3.5 - Partial samples This section will conclude our research into sonic pi player samples. Let's make a quick recap. So far we have looked at how we can run patterns: sample :loop_amen Then we looked at how we can change the speed of patterns such as playing at half speed: pattern :loop_amen, speed: 0.5 Then we looked so we could fade the pattern (let's do it at half speed): sample :loop_amen, speed: 0.5, attack: 1 We also looked at how we could use the beginning of the pattern percussion giving maintenance : explicit value and short-valued attack and release: sample :loop_amen, rate: 2, attack: 0.01, maintenance: 0, release: 0.35 However, wouldn't it be nice if we didn't always have to start from the beginning of the sample? Wouldn't it be nice if we didn't always have to finish at the end of the sample? Selecting a starting point It is possible to select an arbitrary starting point in the sample as between 0 and 1 1 0 is the beginning of the pattern, 1 is the end, and 0.5 is halfway through the pattern. Let's try to play only the last half of the amen break: pattern :loop_amen, start: 0.5 How about the last quarter of the sample: pattern :loop_amen, start: 0.75 Choosing a target point Similarly, it is possible to select an arbitrary target point in the sample as a value between 0 and 1. Let's finish the amen break halfway through: sample :loop_amen, finish: 0.5 Determining the beginning and end Of course, we can combine these two to play arbitrary segments of the audio file. How about only a small part in the middle: pattern :loop_amen, start: 0.4, finish: 0.6 What happens if we choose the starting position after the goal? sample :loop_amen, start: 0.6, finish: 0.4 Cool! He's playing backwards! Combining with a rate We can combine this new ability to play arbitrary segments of sound with our friend rate:. For example, we can play a very small part of the middle of the amen break very slowly: pattern :loop_amen, start: 0.5, Finish: 0.7, foot: 0.2 Combining with envelopes Finally, we can combine all this with our ADSR envelopes to produce interesting results: pattern :loop_amen, start: 0.5, finish: 0.8, rate: -0.2, attack: 0.3, release: 1 Now go and play mashing up patterns with all this fun stuff... 3.6 - External patterns While embedded patterns can get you up and get started quickly, you might want to experiment with other recorded sounds in your music. Sonic Pi totally supports that. First, let's have a quick discussion about the portability of your piece. Portability When you compose your piece exclusively with built-in synths and patterns, the code is all you need to play music faithfully. Think about it for a moment - it's amazing! Simple text that you can email or paste into Gist represents everything you need to play sounds. It makes it really easy to share with your friends as you just need to get to the code. However, if you start using your own prerecorded samples, you lose that portability. This is because to play music other people not only need your code, they also need your samples. This limits the ability of others to manipulate, mash-up and experiment with your work. Of course, this should not prevent you from using your own patterns, it is just something to consider. Local samples So how to play any arbitrary WAV, AIFF or FLAC file on your computer? All you have to do is pass the path of that file to the sample: # Raspberry Pi, Mac, Linux sample /Users/sam/Desktop/my-sound.wav # Windows sample C:/Users/sam/Desktop/my-sound.wav Sonic Pi will automatically load and play the sample. You can also go through all the standard parmes you're used to the passing pattern: # Raspberry Pi, Mac, Linux sample /Users/sam/Desktop/my-sound.wav, rate: 0.5, amp: 0.3 # Windows sample Rate: 0.5, amp: 0.3 3.7 - Sample package Note: this part of the tutorial covers the advanced theme of working with large directories of own samples. This will be the case if you have downloaded or purchased your own sample packages and want to use them inside Sonic Pi. Feel free to skip this if you are satisfied with working with the built-in samples. When working with large folders of external patterns, it can be cumbersome each time you need to type the entire path to start an incremental pattern. For example, let's say you have the following folder on your machine: /path/to/my/samples/ When we look inside that folder we find the following patterns: 100_A#_melody1.wav 100_A#_melody2.wav 100_A#_melody3.wav 120_A#_melody4.wav 120_Bb_guit1.wav 120_Bb_piano1.wav Typical In order to play piano samples we can use the whole route: sample /put/to/my/samples/120_Bb_piano1.wav If we want to then play the guitar pattern we can also use its full path: pattern /put/do/my/samples/120_Bb_guit.wav However, both of these calls to the sample require us to know the names of the samples inside our directory. What if we just want to listen to each sample in return quickly? Indexing package samples If we want to play the first sample in the directory we just need to forward the directory to sample and index 0 as follows: sample /path/to/my/samples/, 0 We can even make a shortcut to our directory using a variable: samps = /path/to/my/samples/ sample samps, 0 Now, if we want to play another sample in our directory, we just need to add 1 to our index: samps = /path/to/my/samples/ sample samps, 1 Notice that we no longer need to know the names of the samples in the directory - we just need to know the directory itself (or have a shortcut to it). If we are looking for an index that is greater than the number of samples, it is simply wrapped like rings. Therefore, whatever number we use, we will be guaranteed to receive one of the samples in that directory. Filtering sample packages Usually indexing is enough, but sometimes we need more power to sort and organize our samples. Fortunately, many sample packages add useful information to filenames. Let's take another look at the file sample names in our directories: 100_A#_melody1.wav 100_A#_melody2.wav 100_A#_melody3.wav 120_A#_melody4.wav 120_Bb_guit1.wav 120_Bb_piano1.wav Notice that we have a lot of information in these file names. First, we have a BPM sample (beats per minute) at the beginning. So the piano pattern is at 120 BPM and our first three melodies are at 100 BPM. Also, our sample names contain a key. So the guitar pattern is in Bb and the melodies are in A#. This information is very useful for mixing these samples with our other code. For example, we know that we can only play a piano sample with a code that is in 120 BPM and in key Bb. It turns out that we can use this particular convention of naming our sets up code to help us filter out the ones we want. For example, if we work at 120 BPM, we can filter all samples containing a string of 120 with the following: samps = /path/to/my/samples/ sample samps, 120 This will play our first match. If we want a second match we just need to use the index: samps = /path/to/my/samples/ sample samps, 120, 1 We can even use more filters at the same time. For example, if we want a sample whose filename contains both substrings 120 and A# we can easily find it with the following code: samps = /path/to/my/samples/ sample samps, 120, A# Finally, we can still add our usual decisions to the sample call: samps = /path/to/my/samples/samps samples, 120, Bb, 1, lpf: 70, amp: 2 Sources The sample filter system understands two types of information: sources and filters. Sources are information used to create a list of potential candidates. The source can have two forms: /path/to/samples - a string representing a valid path to the directory /path/to/samples/foo.wav - a string representing a valid trajectory to the Sample fn sample will first collect all sources and use them to create a large list of candidates. This list is designed to first add all valid paths, and then add all valid .flac, .aif, .aiff, .wav, .wave files contained in the directories. For example, see the following code: samps = /path/to/my/samples/ samps2 = /path/to/my/samples2/ path = /path/to/my/samples3/foo.wav sample samps, samps2, path, 0 Here we combine the contents of the samples within two directories and add a specific sample. If /path/to/my/samples/ contained 3 samples and /put/do/my/samples2/ contained 12, we would have 16 potential index and filtering samples (3+12+1). By default, only sample files within the directory are collected into the list of candidates. Sometimes you may have a number of nested sample folders that you want to search and filter within you. Therefore, you can re-search all samples within all subsets of a specific folder by adding ** to the end of the path: samps = /path/to/nested/patterns/** samples of the samps, 0 Take care even though searching through a very large set of folders can take a long time. However, the contents of all folder sources are cached, so the delay will only occur for the first time. Finally, keep in the beginning that sources must go first. If no source is specified, the set of embedded samples will be selected as the default list of candidates for the paper. Filters After you get a list of candidates, you can use the following types of filtering to further reduce the selection: foo Wires will filter to a substacial occurrence within the file name (minus directory mode and extension). /fo[oO]/ Regular expressions will be filtered to match patterns of file names (minus trajectory and extension). :foo - Keywords will filter candidates on whether the keyword is a direct match with fillies (minus directory path and extension). lambda{|a| ... } - Single-argument Procs will be treated as a candidate filter or generator function. It will pass the list of current candidates and must return a new list of candidates (list of valid paths to sample files). 1 - The numbers will select a candidate with this index (wrapping the circle like a ring if necessary). For example, we can filter all samples in a directory that contains a series of foos and reproduce the first corresponding pattern at half speed: pattern/path/to/samples, foo, rate: 0.5 See sample help for many detailed usage examples. Have filters to be easused. Composites Finally, you can use lists wherever you can set a source or filter. The list will be automatically aligned, and the content will be treated as regular sources and filters. Therefore, the following sample calls are semantically equivalent: sample/path/to/dir, 100, C# sample [/way/to/dir, 100, C#] sample /path/to/1dir, [100, C#] sample [/way/to/dir, [100, [C#]]] Finishing This was an advanced section for people who need real power to manipulate and use patterns. If most of this part didn't make too much sense, don't worry. You probably don't need any of this functionality yet. However, you will know when you need it and you can go back and read this again when you start working with large sample directories. 4 - Randomization A great way to add interest to your music is to use some random numbers. Sonic Pi has some great functionalities for adding randomness to your music, but before we start we have to learn a shocking truth: in Sonic Pi random is not truly random. What on the other end of the day does that mean? Let's see. Repetition A really useful random function is a rrand that will give you a random value between two numbers - min and max. (rrand is short for ranged random). Let's try to play a random note: play rrand (50, 95) Ooh, it played a random note. He played a note of 83.7527. Nice random note between 50 and 95. Woah, wait, did I just anticipate the exact random note you got? There's something suspicious going on here. Try running the code again. What? He picked 83.7527 again? That can't be random! The answer is that it's not really random, it's pseudo-random. Sonic Pi will give you random numbers in a repeatable way. This is very useful for ensuring that the music you make on your machine sounds identical on someone else's machine - even if you use some coincidence in your composition. Of course, in a particular piece of music, if she 'randomly' chose 83.7527 each time, then that wouldn't be very interesting. However, it does not. Try the following: loop not play rrand (50, 95) sleep 0.5 end It finally sounds random. Within a specified run, subsequent calls to random functions will return random values. However, the next ride will produce exactly the same sequence of random values and sound exactly the same. It's like all the Sonic Pi code came back in time to exactly the same point every time the Run button was pressed. Groundhog Day of music synthesis! Haunted Bells A nice illustration of randomization in action is an example of haunted bells looping :p erc_bell pattern at random pace and sleep time between ringtones: loop to sample :p erc_bell, rate: (rrand 0.125, 1.5) dream rrand (0.2, 2) end Random interruption Another fun example of randomization is to randomly modify synth interruption. The great synth to try this on is :tb303 emulator: use_synth :tb303 loop not play 50, edition: 0.1, cutoff: rrand(60, 120) sleep 0.125 end random seeds So what if you do not like this particular sequence of random numbers Sonic Pi provides? Well, it's entirely possible to choose another starting point use_random_seed. The default seed is 0, so choose a second seed for a different random experience! Consider the following: 5th time play rrand (50, 100) sleeping 0.5 end Every time you run this code, you will hear the same sequence of 5 notes. To get a different sequence simply change the seeds: use_random_seed 40 5.times play rrand (50, 100) sleep 0.5 end It will produce a different sequence of 5 notes. By changing the semen and listening to the results, you can find something you like - and when you share it with others, they will hear exactly what you have heard. Let's look at some other useful random functions. Select A very common thing to do is to select an item randomly from a list of known items. For example, I might want to play one note from the following: 60, 65, or 72. I can do this by choosing which one will allow me to select an item from the list. First, I have to put my numbers on a list that's done by wrapping them in square brackets and separating them with commas: [60, 65, 72]. Then I just need to pass them to choose: select([60, 65, 72]) Let's hear what it sounds like: loop do play choose([60, 65, 72]) sleep 1 end rrand We've already seen rrand, but we run it over again. Returns a random number exclusively between two values. This means that it will never return either the upper or lower numbers - always something between the two. The number will always be a float - which means that it is not an integer, but a fraction of the number. Examples of floats returned by rrand (20, 110): 87,5054931640625 86,05255126953125 61,77825927734375 rrand_i You will occasionally want an entire random number, not a float. This is rrand_i he comes to the rescue. It works similarly to rrand, except that it can return min and max values as potential random values (meaning it is inclusive rather than range). Examples of numbers returned by rrand_i(20, 110) are: rand This will return a random float between 0 (inclusive) and the maximum value you specify (exclusive). By default, it will return a value between 0 and 1. Therefore, it is useful to select a random amplifier: values: loop to play 60, amp: rand sleep 0.25 end rand_i Similar to the relationship between rrand_i and rrand, rand_i will return a random integer between 0 and the maximum value you specify. Dice Sometimes you want to imitate the roll of dice - this is a special case rrand_i where the lower value is always 1. A call to the dice requires you to specify the number of sides on the cubes. Standard cubes have 6 sides, so the dice (6) will act very similarly - they return values of either 1, 2, 3, 4, 5 or 6. However, just like fantasy role-play games, you may find a value in 4 side cubes, or 12 side cubes, or 20 one-sided cubes - maybe even 120 one-sided cubes! one_in Finally, you might want to emulate the casting of a top dice rating such as 6 in standard dice. one_in returns true with the likeness of one in the number of sides on the dice. Therefore one_in(6) will return true with a likely of 1 to 6 or false in another way. True and false values are very useful if the statements we will cover in the next section of this tutorial. Now, go and mess up your code with some randomness! 5 - Programming structures Now that you have learned the basics of making sounds with play and pattern and creating simple melodies and rhythms by sleeping between sounds, you may be wondering what else the world of code has to offer you... Well, you're in for an exciting treat! It turns out that basic programming structures such as loop, conditional, functions and threads give you incredibly powerful tools to express your musical ideas. We get stuck with the basics... 5.1 - Blocks Structure you'll see a lot in Sonic Pi's block. Blocks allow us to do useful things with large pieces of code. For example, with synth and sample parameters we were able to change something that happened on one line. However, sometimes we want to do something significant to a number of lines of code. For example, we may want to loop it, add reverb to it, run it only 1 time out of 5, etc. Consider the following code: play 50 sleep 0.5 pattern :elec_plip sleep 0.5 play 62 To do something with a piece of code, we need to tell Sonic Pi where the block code begins and where it ends. We use do to start and end for the end. For example: play 50 sleep 0.5 sample :elec_plip sleep 0.5 play 62 end However, it is not yet complete and will not work (try and you will get an error) because we did not tell Sonic Pi what we want to do with this do/ end block. We say Sonic Pi this by writing some special code before we do it. We'll see a few of these special pieces of code later in this. For now, it's important to know that wrapping code inside up to and end tells Sonic Pi that you want to do something special with that piece of code. 5.2 - Iteration and loops So far we have spent a lot of time watching the different sounds you can make with the game and pattern blocks. We also learned how to run these sounds through time using sleep. As you probably found out, there's a lot of fun you can have with these basic building blocks. However, a whole new dimension of entertainment opens up when you start using the power of code to structure music and compositions. In the next few sections, we will explore some of these powerful new tools. The first is iteration and loops. Repeat Have you written any code that you would like to repeat several times? For example, you may have something like this: play 50 sleep 0.5 pattern :elec_blup sleep 0.5 play 62 sleep 0.25 What if we want to repeat this 3 times? Well, we could do something simple and just copy and paste three times: play 50 sleep 0.5 sample :elec_blup sleep 0.5 play 62 sleep 0.25 play 50 sleep 0.5 sample :elec_blup <7> Sleep 0.5 Play 62 Sleep 0.25 Play 50 Sleep 0.5 Sample :elec_blup Sleep 0.5 Play 62 Sleep 0.25 Now There's a Lot of Code! What happens if you want to change the pattern to :elec_plip? You will need to find all the places with the original :elec_blup transfer them. More importantly, what if you want to repeat the original piece of code 50 times or 1000? That would be a lot of code, and a lot of lines of code to change if you want to make a change. Iteration Actually, repeating the code should be as easy as saying do it three times. Well, it's quite. Remember our old friend's block of codes? We can use it to mark the beginning and end of a code that we would like to repeat three times. Then we use a special code 3.times. So instead of writing it three times, we write 3.times do - it's not too difficult. Just don't forget to write the end at the end of the code you want to repeat: 3.times play 50 sleep 0.5 sample :elec_blup sleep 0.5 play 62 sleep 0.25 end Now it's not so much neahier than cutting and mating! We can use this to create a lot of beautiful repetitive structures: 4.times play 50 sleep 0.5 end of 8th time play 55, release: 0.2 sleep 0.25 end 4.times play 50 sleep 0.5 end nesting iteration We can put iterations within other iterations to create interesting patterns. For example: 4.times make a pattern :d rum_heavy_kick 2.times do a pattern :elec_blip2, rate: 2 sleep 0.25 end sample :elec_snare 4.times do a pattern :d rum_tom_mid_soft sleep 0.125 end of the loop If you want to repeat something many times, you may find yourself using really large numbers such as 1000 times do. In this case, you probably better ask Sonic Pi to repeat forever (at least until you press the stop button!). Let's loop amen break forever: loop :loop_amen sleep sample_duration :loop_amen end The important thing to know about loops is to behave like black holes for code. Once the code has entered the loop, it can never leave until you press stop - it will only round up forever and round the loop. This means that if you have a code after a loop you will never hear it. For example, cymbal after this loop will never play: loop play 50 sleep 1 end pattern :d rum_cymbal_open Now, get structuring your code with iterations and loops! 5.3 - Conditions The usual thing you'll probably find wanting is not only to play a random note (see previous section on coincidence), but also to make a random decision and based on the outcome run some code or some other code. For example, you might want to randomly play a drum or cymbal. We can achieve that by declaring. Toss coins So let's turn the coin: if they are heads, play the drum, if they are tails, play the cymbal. Easy. We can mimic a coin toss with one_in function (introduced in the randomness section) stating a probability of 1 in 2: one_in(2). Then we can use the result of this to decide between two parts of the code, code for playing drums and code for playing cymbals: loop do if one_in(2) pattern :d rum_heavy_kick second pattern :d rum_cymbal_zatvoren end of sleep 0.5 end Notice that if statements have three parts: The question to ask the first choice of code to run (if the answer to the question is yes) The second choice of the boot code (if the answer to the question is no) Usually in programming languages, the notion that it is represented by the phrase true, and the term is not represented by the term false. So we need to find a question that will give us a true or false answer which is exactly what one_in does. Notice how the first choice is wrapped between if and second, and the second choice is wrapped between the second and the end. Just like up/end blocks you can put multiple lines of code anywhere. For example: loop do if one_in(2) pattern :d rum_heavy_kick sleep 0.5 second pattern :d rum_cymbal_zatvorena sleep 0.25 end This time we sleep for different amounts of time depending on which choice we make. Simply if sometimes you want to optionally execute only one line of code. This is possible by asking if and then the question at the end. For example: use_synth :d loops do not play 50, amp: 0.3, edition: 2 play 53, amp: 0.3, release: 2 if one_in(2) play 57, amp: 0.3, edition: 2 if one_in(3) play 60, amp: 0.3, edition: 2 if one_in(4) sleep 1.5 end It will play chords of different numbers with a chance of each note playing has a different probability. 5.4 - Neither So did you make your killer bassline and phat rhythm. How do you play them at the same time? One of the solutions is to weave them by hand - play bass, then some drums, and more bass... However, the weather is soon becoming difficult o, especially when you start weaving in multiple elements. What if Sonic Pi can weave things for you automatically? Well, you can, and you do it with a special thing called thread. Infinite Loops To make this example simple, you will need to imagine that this is a phat beat and a killer bass line: loop to sample :d rum_heavy_kick sleep 1 end loop to use_synth :fm play 40, release: 0.2 sleep 0.5 end As we have already discussed, loops are like black holes for the program. Once you have entered the loop, you can never get out of it until you stop. How can we play both loops at the same time? We need to tell Sonic Pi that we want to start something at the same time as the rest of the code. This is where the threads come to the rescue. Threads to the Rescue in_thread to loop do sample :d rum_heavy_kick sleep 1 end end loop to use_synth :fm play 40, release: 0.2 sleep 0.5 end By wrapping the first loop in the in_thread to/end block we tell Sonic Piu to do the content of the do/end block at exactly the same time as the following statement after the block to/end (which happens to be the second loop). Try and you'll hear drums and bass line woven together! Now, what if we wanted to add synth at the top. Something like: in_thread to loop do sample :d rum_heavy_kick sleep 1 end end loop to use_synth :fm play 40, release: 0.2 sleep 0.5 end loop to use_synth :zawa play 52, release: 2.5, phase: 2, amp: 0.5 sleep 2 end Now we have the same problem as before. The first loop is played at the same time as the second loop due to in_thread. However, the third loop was never reached. Therefore, we need another thread: in_thread to loop to sample :d rum_heavy_kick sleep 1 end end in_thread to loop up to use_synth :fm play 40, Release: 0.2 sleep 0.5 end end loops do use_synth :zawa play 52, release: 2.5, phase: 2, amp: 0.5 sleep 2 end Works as threads What can surprise is that when you press the Run button, you actually create a new thread for the boot code. This is why pressing repeatedly will layer sounds over each other. Since the paths themselves are threads, they will automatically weave sounds together for you. Scope As you learn how to master Sonic Pi, you'll learn that threads are the most important building blocks for your music. One of the important jobs they have is to isolate the notion of current settings from other threads. What's that mean? Well, when you change synths using use_synth you're really just shifting synth into the current thread - no other thread will have your synth replaced. Let's look at this in action: play 50 sleep 1 in_thread to use_synth :tb303 play 50 end sleep 1 play 50 Notice how the middle sound was different from the others? The use_synth only affected the thread it was in, not the external main thread. Inheritance When you create a new thread with in_thread, the new thread will inherit all current settings from the current thread. Let's see that: use_synth :tb303 play 50 sleep 1 in_thread play 55 end Notice how the second note plays with :tb303 synth even though it was played from a separate thread? Any of the settings modified by different use_* functions will behave in the same way. When threads are created, they inherit all settings from parents, but do not share any changes back. Naming threads Finally, we can give our threads names: in_thread (name: :bass) to loop to use_synth :p rohet play chord(:e2, :m7).choose, release: 0.6 sleep 0.5 end in_thread(name: :d rums) to loop to sample :elec_snare sleep 1 end End Look in the log pane when you run this code. See how the journal reports the name of the thread with the message? [Run 36, Time 4.0, Thread :bass] |- synth :p rophet, {release: 0.6, notes: 47} Only One Thread per Allowed Name The last thing to know about named threads is that only one thread of a particular name can work at a time. Let's look into this. Consider the following code: in_thread to loop to sample :loop_amen sleep sample_duration :loop_amen end Go ahead and paste that into a buffer and press the Run button. Press him again a few times. Listen to cacophony more amen breaks loops from time with each other. Okay, you can press Stop now. This is a behavior we have seen over and over again - if you press the Run button, the sound layers on top of any existing sound. Therefore, if you have a loop and press the Run button three times, you will have three layers of loops playing simultaneously. However, with named threads is different: in_thread (name: :amen) to loop to sample :loop_amen sleep sample_duration :loop_amen end End Try pressing the Run button multiple times with this code. You'll only ever hear one amen break a loop. You'll also see this in the log: ==> Skip thread creation: thread with name :amen already exists. Sonic Pi tells you that thread with the name :amen is already playing, so as not to create another. This behavior may not seem immediately useful to you now - but it will be very useful when we start living code... 5.5 - Functions Once you start writing a lot of code, you may want to find a way to organize and structure things to make them more neat and easier to understand. Functions are a very powerful way to do that. They give us the ability to name a bunch of codes. Let's. Defining functions defines :foo to play 50 sleep 1 play 55 sleep 2 end Here we have defined a new function called foo. We do this with our old friend a block to/end and a magic word that follows the name we want to give to our function. We didn't have to call it foo, we could call it whatever we wanted like a bar, base or ideal something meaningful for you like main_section or lead_riff. Don't forget Colon : to the name of your function when you define it. Dialing functions After defining our function we can only call it by writing names: define :foo to play 50 sleep 1 play 55 sleep 0.5 end foo sleep 1 2.times to foo end We can even use foo inside iteration blocks or anywhere we may have written a game or pattern. This gives us a great way to express ourselves and create new meaningful words to use in our compositions. Functions are remembered across the track So far, every time you press the Run button, Sonic Pi started from a completely empty slate. He doesn't know anything but what's in the tampon. You cannot reference code on another clipboard or other thread. However, functions change this. When you define a function, Sonic Pi remembers it. Try. Delete all code on the Clipboard and replace it with: foo Press the Run button - and hear your function game. Where did the code go? How did Sonic Pi know what to play? Sonic Pi just remembered your function - and even after you deleted it from the buffer, he remembered what you typed. This behavior works only with functions created by defining (and defonce). Parameterized functions You may be interested to know that just as you can pass min and maximum values on rrand, you can learn your functions to accept arguments. Let's see: let's my_player |n| play n end my_player 80 sleep 0.5 my_player 90 This isn't very exciting, but it illustrates the point. We created our own version of the game called my_player is parameterized. Parameters must go after being up to the defined to / end block, surrounded by vertical gates | and separated by commas,. You can use the words you want for parameter names. Magic happens within the define to/end block. You can use parameter names as if they were actual values. In this example, I play notes n. You can consider parameters as a kind of promise that when the code works, they will be replaced by actual values. You do this by passing the parameter to the function when you call it. I'm doing this my_player 80 to play note 80. Within the definition of function, n is now replaced by 80, so play n is converted to game 80. When I call him again with my_player 90, n is now replaced by 90, so play n turns into game 90. Let's look at a more interesting example: define :chord_player |, repeats| repeats.times do play chord(root, :minor), release: 0.3 sleep 0.5 end chord_player :e3, 2 sleep 0.5 chord_player :a3, 3 chord_player:g3, 4 sleep 0.5 chord_player :e3, 3 Here I used repetitions as if the number in the line repeats.times do. I also used root as if it were a name for a note in my invitation to play. See how we are able to write something very expressive and easy to read by moving a lot of our logic into operation! 5.6 - Variables The useful thing you can do in your code is create names for Sonic Pi makes this very simple: you write the name you want to use, the same character (=), and then what you want to remember: sample_name = :loop_amen Here, we 'remembered' the symbols :loop_amen in variable sample_name. Now we can sample_name wherever we could use loop_amen. For example: sample_name = :loop_amen sample sample_name There are three main reasons to use variables in Sonic Pi: communicating meanings, managing repetition, and capturing the results of things. Communicating meaning When writing code, it's easy to just think that you're telling your computer how to do things - as long as the computer understands it's ok. However, it is important to remember that it is not only the computer that reads the code. Other people can read it too and try to figure out what's going on. Also, you will probably read your own code in the future and try to figure out what is happening. Although it may seem obvious to you now - it may not be so obvious to others or even your future self! One way to help others understand what your code does is to write comments (as we saw in the previous section). The second is the use of meaningful variable names. Check out this code: sleep 1.7533 Why does it use the number 1.7533? Where did that number come from? What's that mean? However, look at this code: loop_amen_duration = 1.7533 sleep loop_amen_duration Now, it is much clearer what 1.7533 means: this is the duration of the sample :loop_amen! Of course, you could say why you simply would not write: sleep sample_duration(:loop_amen) Which, of course, is a very nice way of communicating the intent of the code. Repeat management You often see a lot of repetitions in your code and when you want to change things, you have to change it in many places. Check out this code: sample :loop_amen sleep sample_duration(:loop_amen) pattern :loop_amen, rate: 0.5 sleep sample_duration(:loop_amen, rate: 0.5) pattern :loop_amen sleep sample_duration(:loop_amen) We do a lot of things with loop_amen! What if we wanted to hear how it sounds with another pattern loop like loop_garzul? We would have to find and replace everything :loop_amens with :loop_garzul. That might be fine if you have a lot of time - but what if you're performing on stage? Sometimes you don't have the luxury of time - especially if you want to keep people dancing. What if you had written your code like this: sample_name = :loop_amen sample sample_name sleep sample_duration(sample_name) pattern sample_name, rate: 0.5 sleep sample_duration(sample_name, rate: 0.5) sample sample_name sleep sample_duration(sample_name) Now, it works exactly the same as above (try). It also gives us the ability to just change one line sample_name = :loop_garzul and change it in many places through the magic of variables. Recording results Finally, good motivation using variables is recording the results of things. For example, you may want to do things with the duration of the sample: sd = sample_duration(:loop_amen) Now we can use the SD wherever we need the duration of the sample :loop_amen. Perhaps more importantly, the variable allows us to record the results of a game call or pattern: s = game 50, release: 8 Now we have caught and remembered as a variable, which allows us to control synth while it works: s = game 50, release: 8 sleep 2 control s, keep in mind: 62 In more detail we will look into the control of synths in the later section. Warning: Variables and threads While variables are great for naming things and capturing the results of things, it's important to know that they usually only need to be used locally within a thread. For example, do not do this: a = (ring 6, 5, 4, 3, 2, 1) live_loop :shuffled do a = a.shuffle sleep 0.5 end live_loop :sorted to a = a.sort sleep 0.5 puts sorted: , end In the example above we assign a ring of numbers to a variable and then use it within two separate live_loops. In the first living loop every 0.5s we sort the ring (on (ring 1, 2, 3, 4, 5, 6)) and then print it in the log. If you run the code, you'll find that the printed list isn't always sorted!. This may surprise you - especially since sometimes the list is printed as sorted and sometimes it is not. It is called indefinite behavior and it is the result of a pretty nasty problem called a racial state. The problem is due to the fact that the second living loop also manipulates the list (in this case mixing) and by the time the list is printed, sometimes it is just sorted, and sometimes it is just stirred. Both live loops race to do something different from the same variable and each time they round out the other loop 'wins'. There are two solutions to this. First, do not use the same variable in multiple live loops or threads. For example, the following code will always print a sorted list because each living loop has its own separate variable: live_loop :shuffled do a = (ring 6, 5, 4, 3, 2, 1) a = a.shuffle sleep 0.5 end live_loop :sorted to a = (ring 6, 5, 4, 3, 2, 1) a = a.sort sleep 0.5 puts sorted: , end However, sometimes we want to share things across threads. For example, the current key, BPM, synth etc. In these cases, the solution is to use Sonic Pi's special state system without a thread through the use of fns get and set. This is later discussed in section 10. 5.7 - Thread synchronization After becoming advanced enough live coding with numerous functions and threads simultaneously, you probably noticed that it is quite easy to err in one of the threads that kills it. This is not a big deal, because you can easily restart the thread by hitting the Run. However, when you restart the thread, it is now out of time with the original Legacy time As we discussed earlier, new threads created in_thread inherit all settings from the parent thread. That includes the current time. This means that the threads are always on time with each other when they start simultaneously. However, when you start the thread yourself, it begins with its own time that is unlikely to be aligned with any of the other currently running threads. Cue and Sync Sonic Pi provides a solution to this problem with sign and sync functions. the sign allows us to send heart rate messages to all other topics. By default, other threads are not interested and ignore these heartbeat messages. However, you can easily register an interest with the synchronization function. It is important to be aware that synchronisation is a simple function in that it stops the current thread from doing anything for a while. However, we should think about how long you want to wait and where you are sync you do not know how long you will wait - because sync awaits the next character from another thread that may be soon or a long time away. Let's explore in_thread a little more detail: in_thread to loop to cue :tick sleep 1 end end in_thread to loop to sync :tick sample :d rum_heavy_kick end end Here we have two threads - one that behaves like a metronome, does not play any sounds but sends :tick heartbeat messages every beat. The second thread is synchronized on the messages of ticks, and when it receives one, it inherits the thread time of the cue thread and continues to run. As a result, we will hear a :d rum_heavy_kick pattern exactly when the second thread sends :tick message, even if two strands have not started their performance at the same time: in_thread to loop to cue :tick sleep 1 end end sleep(0.3) in_thread to loop to sync :tick sample :d rum_heavy_kick end end That naughty sleep call would usually take the second thread out of phase from the first. However, while using sign and sync, we automatically synchronize threads by bypassing any random time shifts. Cue Names You can use any name you want for your messages - not just :tick. You just need to ensure that any other threads are synchronized to the correct name - otherwise they will wait forever (or at least until you press the Stop button). Let's play with a few character names: in_thread to loop to cue [:foo, :bar, :baz].choose sleep 0.5 end end in_thread to loop to sync :foo sample :elec_beep end end in_thread to loop to sync :bar sample :elec_flip end end in_thread to loop to sync :baz sample :elec_blup end end Here we have a master loop of characters that randomly sends one of the heart rate names :foo, :bar or :baz. Then we also have three thread loops that are synchronized on each of these names independently, and then play a different pattern. The net effect is that we hear a sound every 0.5 beats because each of the synchronized threads is randomly synchronized with the thread and plays its pattern. Of course, this also works if you organize threads in reverse because the sync threads will simply sit and wait for the next character. 6 - Studio FX One of the most exciting and fun aspects of Sonic Pi is the ability to easily add studio effects to your sounds. For example, you might want to add some echo to parts of your piece, or some echo or maybe even distort or wobble your bass lines. Sonic Pi provides a very simple but powerful way to add FX. It even allows you to chain them (so you can pass your sounds through distortion, then echo and then reverb) and control each individual FX unit with opts (similarly giving parame to synths and patterns). You can even modify THE FX's opts while it's still running. So, for example, you can increase the reverb on bass throughout the course... Guitar pedals If this all sounds a little complicated, don't worry. After playing with it a little, everything will become quite clear. Before you do this, a simple analogy is that of the guitar FX pedal. There are many types of FX pedals that you can buy. Some add reverb, others distort, etc. The guitarist will plug his guitar into one FX pedal - that is, distortion -, then take another cable and connect the (chain) reverb pedal. The reverb pedal output can then be connected to the amplifier: Guitar -> Distortion -> Reverb -> Amplifier It's called FX Chain. Sonic Pi supports just that. In addition, each pedal often has dials and sliders that allow you to control how many distortions, reverbs, echoes, etc. Sonic Pi also supports this - but instead of needing someone else to control things for you, that's where the computer jumps in. Let's explore FX! 6.1 - Adding FX In this section we will look at a pair of FX: echo and echo. We will see how to use them, how to control their opts and how to tie them with chains. Sonic Pi's FX system uses blocks. So if you haven't read section 5.1 you might want to take a quick look, then head back. Reverb If we want to use reverb we write with_fx :reverb as special code to our block like this: with_fx :reverb to play 50 sleep 0.5 sample :elec_plip sleep 0.5 play 62 end Now play this code and you will hear it played with reverb. Sounds good, doesn't it? It all sounds pretty nice with a reverb. Let's see what happens if we have code outside the block to=end: with_fx :reverb to play 50 sleep 0.5 sample :elec_plip sleep 0.5 play 62 end sleep 1 play 55 Notice how the final game 55 is not played with reverb. That's because it is outside the do/end block, so it's not caught by the reverb FX. Similarly, if you make sounds before the block to/end, they will also not be caught: play 55 sleep 1 with_fx :reverb not play 50 sleep 0.5 sample :elec_plip sleep 0.5 play 62 end sleep 1 play 55 Echo There is a lot more FX to choose from. How about an echo? with_fx :echo to play 50 sleep 0.5 sample :elec_plip sleep 0.5 play 62 end One of the powerful aspects of Sonic Pi's FX blocks is that opts similar to the opts we have already seen with the game and pattern can be forwarded. For example, the fun echo you choose to play with is the stage: representing the duration of a particular echo in your beats. Let's make the echo slower: with_fx :echo, stage: 0.5 to play 50 sleep 0.5 sample :elec_plip sleep 0.5 play 62 end Let's make and echo faster: with_fx :echo, stage: 0.125 to play 50 Sleep 0.5 sample :elec_plip sleep 0.5 play 62 End Let's Echo Take Longer Fade By Setting Decay: Time to 8 Beats: with_fx:echo, stage: 0.5, Decay: 8 Play 50 Sleep 0.5 Pattern :elec_plip Sleep 0.5 Play 62 End Nesting FX One of the most powerful aspects of FX blocks is that you can nest them. This allows you to very easily chain FX together. For example, what if you wanted to play some code with an analyte and then with reverb? Simply, just put one FX inside the other: with_fx :reverb to with_fx :echo, stage: 0.5, decay: 8 to play 50 sleep 0.5 sample :elec_blup sleep 0.5 play 62 end End Think of the sound flowing from the inside out. The sound of all the code within the inner to/end block such as play 50 is first sent to echo FX and the echo sound is in turn sent to reverb FX. We can use very deep nesting for crazy results. However, be warned, FX can use a lot of resources and when you nest them you are effectively running multiple FX simultaneously. So be frugal with using FX especially on low-drive platforms like Raspberry Pi. Discover FX Sonic Pi ships with a large number of FX for you to play with. To find out which ones are available, click on FX in the far left of this Help system and you'll see a list of options available. Here's a list of some of my favorites: wobble, echo, distillgerment, cutter Now go crazy and add FX everywhere for some amazing new sounds! 6.2 - FX in practice Although they look deceptively simple on the outside, FX are actually quite complex beasts internally. Their simplicity often lures people to overuse them in their pieces. This may be fine if you have a powerful machine, but if - like me - you use Raspberry Pi for jam, you need to be careful about how much work you're looking for if you want to make sure your beats keep flowing. Consider this code: loop up to with_fx :reverb to play 60, release: 0.1 sleep 0.125 end end In this code we play note 60 with a very short exit time, so it's a short note. We also want an echo, so we've got him in a reverb block. It's all good so far. Except... Let's see what the code does. First we have a loop, which means everything inside it repeats itself forever. Next we have with_fx block. That means we're going to create a new FX reverb every time we loop around. It's like having a separate FX reverb pedal for every time you pull a string on your guitar. It's cool that you can do it, but it's not always what you want. For example, this code will struggle to run nicely at Raspberry Pi. All the work of creating a reverb and then waiting until it needs to be stopped and removed, everything is solved by with_fx, but it takes CPU power that can be precious. How do you make it more like a traditional line-up where our guitarist has only one reverb pedal through which all the sounds go? Simple: with_fx :reverb to loop to play 60, release: 0.1 sleep 0.125 end end We put a loop inside with_fx block. In this way, we create only one echo for all the notes played in our loop. This code is much more efficient and would work well on Raspberry Pi. The compromise is to use with_fx over iteration within the loop: loop up with_fx:reverb to play 60, release: 0.1 sleep 0.125 end end That's it. We raised with_fx from the inner part of the loop and now we create a new reverb every 16 notes. It's such a common pattern that with_fx supports opting for just that, but without writing 16 times: loop up to with_fx:reverb, reps: 16 to play 60, release: 0.1 sleep 0.125 end end Both repetitions: 16 and 16.times up examples will behave identically. Repeat: 16 basically repeats the code in the block to/end 16 times, so you can use them both interchangeably and choose the one that feels best for you. Remember, no mistakes, just possibilities. However, some of these approaches will have a different sound and different performance characteristics. So play around and use the best-sounding approach while working within your platform's performance limits at the same time. 7 - Controlling running sounds So far we've looked at how you can run synths and patterns, as well as how to change their default opts such as amplitude, pan, envelope settings, and more. Each activated sound is intestou its own sound with its own list of options set for the duration of the sound. Wouldn't it be cool if you could change the sound while he's still playing, just like you could bend a guitar string while it's still vibrating? You are lucky - this part will show you how to do just that. 7.1 - Controlling Running Synths So far we have only been engaged in launching new sounds and FX. However, Sonic Pi gives us the ability to manipulate and control currently running sounds. We do this by using a variable to record a reference to synth: s = play 60, release: 5 Here we have a run-local variable s representing synth playing notes 60. Have the competition that it's run-local - you can't access it from other tracks such as functions. Once we have had s, we can begin to control it through the control function: s = game 60, release: 5 sleep 0.5 control s, notes: 65 sleep 0.5 control s, notes: 67 sleep 3 control s, notes: 72 The thing to notice is that here we do not run 4 different synths - we only run one synth and then change the terrain 3 times after that, while the game is running. We can pass any of the standard decides to control, so you can control things like amp:, cutoff: or pan:. Un-controllable options Some of the decision-making cannot be controlled once synth has started. This is the case for all parameters of the ADSR envelope. You can find out which options can be controlled by looking at their documentation in the help system. If the documentation says it can't be changed once it's set up, you know it's not possible to control the opt after the synth has started. 7.2 - Controlling FX It is also possible to control FX, although this is achieved in a slightly different way: with_fx :reverb to |r| play 50 sleep 0.5 control r, mix: 0.7 play 55 sleep 1 control r, mix: 0.9 sleep 1 play 62 end Instead of using variables, we use the parameters of the gate to/end block. Inside | Bars, we must specify a unique name for our running FX which we then reference from containing up to/end block. This behavior is identical to using parameterized functions. Now go and control some synths and FX! 7.3 - Sliding opts While exploring synth and FX opts, you may have noticed that there are a number of opts that end with _slide. You may have even tried to call them and see no effect. This is because these are not normal parameters, they are special opts that work only when you control synthesizers as introduced in the previous section. Consider the following example: s = game 60, release: 5 sleep 0.5 control s, notes: 65 sleep 0.5 control s, notes: 67 sleep 3 control s, notes: 72 Here you can hear how the synth pitch changes immediately on each control call. However, we might want the plot to slide between changes. While we control the note: parameter, to add a slide, we need to set the note_slide parameter synthe: s = game 60, release: 5, note_slide: 1 sleep 0.5 control s, notes: 65 sleep 0.5 control s, notes: 67 sleep 3 control s, notes: 72 Now we hear notes bending between control calls. Sounds nice, doesn't it? You can speed up a slide with a shorter time note_slide: 0.2, or slow it down with a longer scrolling time. Each parameter that can be controlled has an appropriate _slide parameters to play with. Sliding is sticky After you set _slide parameters to a liquid synth, it will be remembered and used each time you slide the appropriate parameter. To stop slipping, you must set the _slide value to 0 before the next control call. Sliding FX Opts It is also possible to glide FX opts: with_fx :wobble, stage: 1.5 to |e| use_synth :d game now use_synth to control e, stage: 0.025 end Now have fun with sliding things for smooth transitions and flow control... 8 - Data Structures A very useful tool in the developer's tool is the data structure. Sometimes you may want to represent and use more than one thing. For example, it may be useful for you to have a series of notes to play one after the other. Programming languages have data structures that allow you to do just that. Many exciting and exotic data structures are available to developers , and people are always inventing new ones. However, for now we just need to consider a very simple data structure - a list. Let's look at it in more detail. We'll cover your basic shape and then how lists can be used to present scales and chords. 8.1 - Lists In this section we will look at the structure of the data that is very useful - the list. We met him very shortly before in the randomization section when we randomly selected from the list of notes for the game: play choose([55, 62]) In this section, we will explore using lists that also represent chords and scales. Let's do it again to see how we can play the chord. Remember that if we do not use sleep, sounds happen at the same time: play 52 play 55 play 59 Let's look at other ways to present this code. Playing list One option is to put all notes on the list: [52, 55, 59]. Our friendly game function is smart enough to know how to play a list of notes. Try: play [52, 55, 59] Ooh, it's already nicer to read. Playing a list of notes doesn't stop you from using any of the parameters as usual: play [52, 55, 59], amp: 0.3 Of course, you can also use traditional note names instead of MIDI numbers: play [:E3, :G3, :B3] Now those of you lucky enough to have studied some musical theory might recognize that chord as E Minor played at 3. Access to the List Another very useful feature of the list is the ability to get information out of it. It may sound a little weird, but it's no more complicated than being asked to turn a book on page 23. With a list, you'd say, what's the element on index 23? The only strange thing is that in programming indices usually start at 0 not 1. With list indices, we don't count 1, 2, 3... Instead, we count 0, 1, 2... Let's take a closer look at this. Check out this list: [52, 55, 59] There's nothing particularly scary about it. What's the other element on that list? Yes, of course, that's 55. That was easy. Let's see if we can get a computer to respond to it for us too: it puts [52, 55, 59][1] OK, it looks a bit weird if you've never seen anything like it before. Believe me, it's not too hard. There are three parts of the line above: word puts , our list of 52, 55, 59 and our index [1]. First we say puts because we want Sonic Pi to print an answer for us in Then we give him our list, and finally our index looks for another element. We need to surround our index with square brackets and because counting starts at 0, the index for the second element is 1. Watch: # indices: 0 1 2 [52, 55, 59] Try starting code puts [52, 55, 59][1] and you will see 55 pop-ups in the log. Change index 1 to other indexes, try longer lists, and think about how you might use the list in the next jam code. For example, what musical structures can be represented as a series of numbers... 8.2 - Chordi Sonic Pi has built-in support for chord names that will restore lists. Try it yourself: play the chord(:E3, :minor) Now, we're really getting somewhere. It looks a lot more beautiful than raw lists (and it's easier to read for other people). What other chords does Sonic Pi support? Well, a lot. Try some of these: chord(:E3, :m7) chord(:E3, :minor) chord(:E3, :d im7) chord(:E3, :d om7) Arpeggios Easy chords we can turn into arpeggios with function play_pattern: play_pattern chord(:E3, :m7) Ok, it's not so fun - played it really slow. play_pattern will note each note on the list separated by a sleeping call 1 between each play call. We can use another play_pattern_timed function to specify our own time and speed things up: play_pattern_timed chord(:E3, :m7), 0.25 We can even pass a list of times that will be treated as a circle of time: play_pattern_timed chord(:E3, :m13), [0.25, 0.5] This is the equivalent: play 52 sleep 0.25 play 55 sleep 0.5 play 59 sleep 0.25 play 62 sleep 0.5 play 66 sleep 0.25 play 69 sleep 0.5 play 73 What would you rather write? 8.3 - Scales Sonic Pi has support for a wide range of scales. How about playing C3 leaderboard? play_pattern_timed rankings(:c3, :major), 0.125, release: 0.1 We can even request more octave: play_pattern_timed scale(:c3, :major, num_octaves: 3), 0.125, release: 0.1 How about all notes on the pentatonic scale? play_pattern_timed (scale :c3, :major_pentatonic, num_octaves: 3), 0.125, release: 0.1 Random Chords and Scales notes are great ways to limit random choice to something meaningful. Have a game with this example that selects random notes from the chords of E3 minors: use_synth :tb303 loop not play choose(chord(:E3, :less)), release: 0.3, cutoff: rrand(60, 120) sleep 0.25 end Try switching to different chord names and cutoff ranges. Detect chords and scales To find out which scales and chords are supported by Sonic Pi simply click the Lang button on the far left of this tutorial, and then select a chord or scale in the API list. In the information on the main panel, scroll down until you see a long list of chords or scales (depending on which one you're looking at). Have fun and remember: no mistakes, only possibilities. 8.4 - Rings Interesting spin on standard lists are rings. If you know some you may have encountered ring buffers or ring strings. Here, we will just go to the ring - it is short and simple. In the previous census section, we saw how we can retrieve elements from them using the indexing mechanism: puts [52, 55, 59][1] Now, what happens if you want the 100 index? Well, obviously there's no element to the 100 index because the list only has three elements in it. SoNic Pi will give you back zero, which means nothing. However, consider that you have a counter such as the current rhythm that is constantly increasing. Let's create our counter and our list: counter = 0 notes = [52, 55, 59] Now we can use our counter to access the being on our list: puts notes [counter] Super, we got 52. Now, let's step our counter and get another note: counter = (inc counter) puts notes [counter] Super, now we get 55 and if we do it again we get 59. However, if we do it again, we run out of numbers on our list and get zero. What if we just wanted to loop back the circle and start at the beginning of the list again? That's what the rings are for. Creating rings We can create rings in one of two ways. Or we use the ring function with ring elements as parameters: (ring 52, 55, 59) Or we can take a normal list and turn it into a ring by sending it a .ring message: [52, 55, 59].ring Indexing Rings Once we have the ring, you can use it in the same way that you would use a normal list with the exception that you can use indexes that are negative or larger than the ring size and they will wrap to always comply with one of the elements of the ring: (ring 52, 55, 59)[0] #=> 52 (ring 52, 55, 59)[1] #=> 55 (ring 52, 55, 59)[2] #=> 59 (ring 52, 55, 59)[3] #=> 52 (ring 52 , 55, 59)[-1] #=> 59 Using rings Let's say we use a variable that represents the current heart rate. We can use this as an index in our ring to get notes for the game, or the release time or anything useful that we have stored in our ring regardless of the number of beats we are currently at. Scales and chords are rings A useful thing to know is that lists returned by scale and chord are also rings and allow you to access arbitrary indices. Ring constructors Besides the ring there are a number of other functions that will build a ring for us. Range invites you to specify a starting point, endpoint, and step size. bools allows you to use 1s and 0s summarized to represent booleans. knitting allows you to knit a series of repeated values. Expansion creates a ring of bools with Euclidean distribution. See their documentation for more information. 8.5 - Ring chains In addition to constructors such as range and expansion another way to create new rings is to manipulate existing rings. Chain commands To investigate this, take a simple ring: (ring 10, 20, 30, 40, What if we want him backwards? Well, we'd use a chain command.the other way around to get the ring and turn around. Se, se. Eye: (ring 10, 20, 30, 40, 50).vice versa #=> (ring 50, 40, 30, 20, 10) Now, what if we want the first three values from the ring? (ring 10, 20, 30, 40, 50).take(3) #=> (ring 10, 20, 30) Finally, what if we want to stir the ring? (ring 10, 20, 30, 40, 50).shuffle #=> (ring 40, 30, 10, 50, 20) More chains This is already a powerful way to create new rings. However, real power comes when you merge several of these commands. How about mixing the ring, releasing 1 element, and then taking the next 3? Let's take it in stages: (ring 10, 20, 30, 40, 50) - our starting ring (ring 10, 20, 30, 40, 50).shuffle - shuffles - (ring 40, 30, 10, 50, 20) (ring 10, 20, 30, 40, 50).shuffle.drop(1) - drop 1 - (ring 30, 10, 50, 20) (ring 10, 20, 30, 40, 50).shuffle.drop(1).take(3) - take 3 - (ring 30, 10, 50) You can see how we can only create a long chain of these methods by just putting them together. We can combine them in any order we want by creating an extremely rich and powerful way to generate new rings from existing ones. Failure These rings have a powerful and important property. They're immutable, which means they can't change. This means that the chain chain methods described in this section do not change the rings, but create new rings. This means that you are free to divide the rings across the threads and begin to tie them inside the thread knowing that you will not affect any other thread using the same ring. Available chain methods Here's a list of available chain methods from which you can play: .reverse - returns the reverse version of the ring .sort - creates a sorty version of the ring .shuffle - creates a mixed version of the ring .pick(13) - returns ring with call results .select 3 times .pick - similar to .pick(3) only the size given to the same as the original ring .take(5) - returns a new ring that contains only the first 5 elements .drop (3) - Returns a new ring with everything but the first 3 elements .butlast - returns a new ring with the last missing element .drop_last(3) - returns a new ring with the last 3 missing elements .take_last(6)- returns a new ring with only the last 6 elements .stretch(2) - repeats each element in the ring twice .repeat(3) - repeats the entire ring 3 times .mirror - adds the ring to the reverse version of itself .reflects - the same as the mirror , but does not duplicate the mean .scale(2) - returns a new ring with all elements multiplied by 2 (assumes that the ring contains only numbers) Of course, those chain methods that take numbers can take other numbers! So feel free to call .drop(5) instead of .drop(3) if you want to drop the first 5 elements. 9 - Live Coding One of the most exciting aspects of Sonic Pi is that it allows you to write and modify live code to make music, just as you could perform live with a guitar. One of the advantages of this approach is to give you more while composing (get a simple loop and continue to tweak it until it sounds perfect). However, the main advantage is that you can take Sonic Pi on stage and concert with him. In this section we will cover the basics of converting static code compositions into dynamic versions. Hold on to your seats... 9.1 - Live Coding Now we've learned enough to really start having fun. In this section, we'll pull from all the previous sections and show you how you can start creating your music tracks live and turn them into a performance. For this, we will need 3 main ingredients: The ability to write code that makes sounds - CHECK! Ability to write functions - CHECK! Possibility of using (named) threads - CHECK! All right, let's start. Let's live with our first sounds. First we need a function that contains the code we want to play. Let's just start by. We also want to loop calls to this function in a thread: define :my_sound play 50 sleep 1 end in_thread(name: :looper) to loop to my_sound end end If my_sound looks a little too complicated to you, go back and read the sections on functions and threads again. It's not too complicated if you've already wrapped your head around these things. What we have here is a definition of a function that just plays notes 50 and sleeps for rhythm. Next, we define a named thread called :looper that just loops around the my_sound multiple times. If you create this code, you will hear that note 50 is repeated over and over again... Changing it now, this is where the fun begins. While the code is still running, change 50 to another number, say 55, and then press the Run button again. Woah! It's changed! Live! It did not add a new layer because we use named threads that allow only one thread for each name. Also, the sound changed because we redefined the function. We gave :my_sound new definition. When the :looper thread loops around it simply calls a new definition. Try changing it again, change the note, change your sleep time. How about adding a use_synth statement? For example, change it to: define :my_sound to use_synth :tb303 play 50, edition: 0.3 sleep 0.25 end Now sounds quite interesting, but we can spice it up further. Instead of playing the same note over and over again, try playing a chord: define my_sound to use_synth :tb303 play chord(:e3, :minor), release: 0.3 sleep 0.5 end How about playing random notes from chords: define :my_sound to use_synth :tb303 play choose(chord(:e3, :minor)), release: 0.3 sleep 0.25 end Or using random cutoff value: define :my_sound to use_synth :tb303 play choose(chord(:e3, :minor)), release: 0.2, cutoff: rrand(60, 130) sleep 0.25 end Finally, add some drums: define :my_sound to use_synth :tb303 pattern :d rum_bass_hard, rate: rrand (0.5, 2) play choose(chord(:e3, :minor)), release: 0.2, cutoff: rrand(60, 130) sleep 0.25 end things Exciting! Exciting! before you jump up and start living coding with functions and threads, stop what you're doing and read the next section on live_loop that will forever change the way you code in Sonic Pi... 9.2 - Live Loops Ok, so this part of the tutorial is a real gem. If you've read just one section, it should be this one. If you read the previous section on Live Coding Fundamentals, live_loop is an easy way to do just that, but without having to write so much. If you haven't read the previous section, live_loop's the best way to get stuck with Sonic Pi. Let's play. Write the following on the new clipboard: live_loop :foo to play 60 sleep 1 end Now press the Run button. You hear the basic beep every beat. There's nothing fun there. However, don't press Stop just yet. Change 60 to 65 and press Run again. Woah! It automatically changed without missing a beat. This is live coding. Why don't you change it to be more of a bass? Just update your code as it plays: live_loop :foo to use_synth :p rohet play :e1, release: 8 sleep 8 end Then hit Run. Let the cutoff move: live_loop:foo to use_synth :p rophet play :e1, release: 8, cutoff: rrand(70, 130) sleep 8 end hit run again. Add some drums: live_loop :foo to pattern :loop_garzul use_synth :p rohet play :e1, Release: 8, cutoff: rrand(70, 130) sleep 8 end Change the note from e1 to c1: live_loop :foo to pattern :loop_garzul use_synth :p rofet play :c1, release: 8, cutoff: rrand(70, 130) sleep 8 end Now stop listening to me and play around! have fun! 9.3 - Multiple Live Loops Consider the following live loop: live_loop:foo to play 50 sleep 1 end You may have wondered why it needs the name :foo. This name is important because it indicates that this living loop differs from all other living loops. There can never be two live loops working with the same name. This means that if we want more simultaneously running live loops, we just need to give them different names: live_loop :foo to use_synth :p rofet play :c1, release: 8, cutoff: rrand (70, 130) sleep 8 end live_loop :bar to sample :bd_haus sleep 0.5 end Now you can update and change every living loop independently and all this just works. Live Loops synchronization One thing you may have already noticed is that live loops work automatically with the thread cue mechanism we've previously explored. Each time it loops live, it generates a new event with the name of a live loop. Therefore, we can synchronise on these signs to ensure that our loops are synchronized without stopping anything. Consider this poorly synced code: live_loop:foo to play :e4, release: 0.5 sleep 0.4 end live_loop :bar to sample :bd_haus sleep 1 end Let's see if we can fix the time and sync it without stopping. First, let's fix the :foo loop to make sleep a factor of 1 - something like 0.5 will do: live_loop :foo to play Release: 0.5 Sleep 0.5 end live_loop :bar up sample pattern Sleep 1 end Yet we are not quite finished - you will notice that the beats do not exactly line up properly. That's because the loops aren't in the phase. Let's fix this by syncing to each other: live_loop :foo to play :e4, release: 0.5 sleep 0.5 end live_loop :bar to sync :foo sample :bd_haus sleep 1 end Wow, everything is perfect now on time - all without stopping. Now, go ahead and live code with live loops! 9.4 - Ticking off something you'll probably find to do a lot when live coding is spinning through the rings. You'll put notes in ringtones, sleep for rhythms, chord progressions, timbral variations, etc. Ticking Rings Sonic Pi provides a very handy tool for working with rings within live_loops. It's called a tick system. In the ring section, we talked about a constantly increasing counter, like the current heart rate. Tick is just implementing this idea. It gives you the ability to mark rings. Let's look at the example: counter = live_loop :arp to play (scale :e3, :minor_pentatonic)[counter], release: 0.1 counter += 1 sleep 0.125 end This is the equivalent: live_loop :arp do play (scale :e3, :minor_pentatonic).tick, release: 0.1 sleep 0.125 end Here, we're just grabbing scale E3 smaller pentatonic and ticking through each element. This is done by adding a .tick to the end of the scale declaration. This checkmark is local to a live loop, so each live loop can have its own independent label: live_loop:arp do play (scale :e3, :minor_pentatonic).tick, release: 0.1 sleep 0.125 end live_loop :arp2 to use_synth :d saw play (scale :e2, :minor_pentatonic, num_octaves: 3).tick, release: 0.25 sleep 0.25 end Tick You can also call the tick as a standard fn and use the value as an index : live_loop :arp do idx = tick play (scale :e3, :minor_pentatonic)[idx], release: 0.1 sleep 0.125 end However, it is much nicer to call .tick at the end. The fn tick is for when you want to do fancy things with tick value and for when you want to use ticks for things other than indexing into rings. Look At the magic thing about the tick is that not only returns a new index (or ring value on that index) also makes sure that the next time you call a tick, it's the next value. See examples in documents for a tick for many ways to work with it. However, for now it is important to point out that sometimes you will just want to look at the current value of the tick and not increase it. This is available via the fn layout. You can call it look like a standard fn or by adding a .look at the end of the ring. Naming a tick Finally, sometimes you will need more than one tick per live loop. This is achieved by giving the name of the check mark: live_loop :arp do play (scale :e3, :minor_pentatonic).tick(:foo), edition: 0.1 sleep (ring 0.125, 0.25).tick(:bar) end Here we use two ticks one for one play and another one while sleeping. Since they are both in the same living loop, in order to separate them we must give them unique names. This is exactly the same thing as naming live_loops - we just pass the symbol prefix with :. In the example above, we called one tick :foo, and another :bar. If we want to look at them, we also need to pass the name of the tick to look. Do not make it too complicated Most of the power in the tick system is not useful when you start. Do not try to learn everything in this section. Just focus on knocking through one ring. This will give you most of the joy and simplicity of knocking through the rings in your live_loops. Look at the documentation for the tick where there are many useful examples and happy ticking! 10 - Weather condition It is often useful to have information that is divided into multiple threads or live loops. For example, you might want to share the concept of the current key, BPM, or even more abstract concepts such as current complexity (which you would potentially interpret in different ways on different topics). We also don't want to lose any of our existing deterministic guarantees when we do. In other words, we would still like to be able to share code with others and know exactly what they will hear when they run it. At the end of section 5.6 of this guide, we briefly discussed why we should not use variables to share information on all topics due to loss of determinism (in turn due to race conditions). Sonic Pi's solution to the problem of simply working with global variables in a deterministic way is through a new system he calls Time State. This can sound complex and difficult (in fact, in the UK, multi-thread programming and shared memory is usually a subject at university level). However, as you'll see, just like playing the first note, Sonic Pi makes it incredibly easy to share the situation on all topics while keeping your programs safe and deterministic.. Meet the get and set ... 10.1 - Set and Get Sonic Pi has a global memory store called Time State. The two main things you do with it are set up information and get information. Let's dive deeper... Set To store information in a time state we need two things: the information we want to store, the unique name (key) for the information. For example, we might want to store the number 3000 with the key :intensity. This is possible by using the set function: set :intensity, 3000 We can use any name for our key. If the information is already stored with this key, our new set will override it: set :intensity, 1000 set :intensity, 3000 In the example above, how we stored both numbers under the same key, the last call to set a 'win', so the number associated with :intensity will be 3000 because the first setup call is effectively overridden. for the ation of the information from the weather country we just need the key we used to set it up, which in our case is :intensity. Then we just need to call the get[:intensity] that we can see by printing the results on the log: printing profits[:intensity] #=> prints 3000 Notification that profit calls can return information that was set in the previous run. Once some of the information is set up, it is available until the information is overridden (just as we closed the value :intensity from 1000 to 3000 above) or Sonic Pi. Multiple threads The main advantage of the Time State system is that it can be used safely over threads or live loops. For example, you can have one piece of information about setting up live loops and another that gets them: live_loop:setter do set :foo, rrand(70, 130) sleep 1 end live_loop :getter to put get[:foo] sleep 0.5 end Nice thing about using get and set over threads like this is that it will always produce the same result every time you hit a run. Go ahead, try it. See if you get the following in the log: {run: 0, time: 0.0} └─ 125.72265625 {run: 0, time: 0.5} └─ 125.7226625 {run: 0, time: 1.0} └─ 76.26220703125 {run: 0, time: 1.5} └─ 76.26220703125 {run: 0, time: 2.0} └─ 114.93408203125 {run: 0, 0 time: 2.5} └─ 114.93408203125 {run: 0, time: 3.0} └─ 75.6048583984375 {run: 0, time: 3.5} └─ 75.6048583984375 Try running it several times - see, it's the same every time. This is what we call deterministic behavior and it really matters very much when we want to share our music as code and know that the person who plays the code hears exactly what we wanted them to hear (just as playing mp3 or internet stream sounds the same for all listeners). A simple deterministic state system in section 5.6 discussed why using variables over threads can lead to random behavior. This prevents us from reliably playing code like this: ## Example of indecisive behavior ## (due to race conditions caused by multiple ## live loops manipulating the same variable ## at the same time). # # # If you run this code you'll notice that the list that's printed ## isn't always sorted! = (ring 6, 5, 4, 3, 2, 1) live_loop :shuffled do a = a.shuffle sleep 0.5 end live_loop :sorted to a = a.sort sleep 0.5 puts sorted: , and end Let's look at what it might look like using get and set: ## Example of deterministic behavior ## (despite simultaneous access to shared state) ## using Sonic Pi's new Time State system. ## # When this code is executed, the list that is ## printed is always sorted! set :a, (ring 6, 5, 4, 3, 2, 1) live_loop :shuffled to set :a, get[:a].shuffle sleep 0.5 end live_loop :sorted to set :a, get[:a].sort sleep 0.5 puts sorted: , get[:a] end Notice how this code is quite identical to the version using the variable before it. However, when you run a code, it behaves as you expected with any typical Sonic Sonic Code - it does the same thing every time in this case thanks to the time state system. Therefore, when sharing information over live loops and threads, use the get and set instead of variables for deterministic, repeatable behavior. 10.2 - Sync Section 5.7 has introduced functions and synced when it comes to thread synchronization issues. What it didn't explain was that it was the State system that provided this functionality. Coincidentally, this set is actually a variation of the character and is built on top of the same basic functionality that is the insertion of information into the time state system. In addition, synchronisation is also designed in such a way that it works seamlessly with Time State - any information we plan to store in Time mode where we can sync. In other words - we synchronize at events that have yet to be inserted into the Time State. Waiting for Events Let's take a quick look at how to use sync to wait for new events to be added to Time State: in_thread to sync :foo sample :ambi_lunar_land end sleep 2 set :foo, 1 In this example, we first create a thread waiting for the :foo event to be added to the Time State. After this declaration, we do not sleep for 2 beats, and then we set :foo to be 1. This then releases the synchronization, which then moves to the next line, which is to run the pattern :ambi_lunar_land pattern. For example, sync always waits for future events and will block the current thread waiting for a new event. It will also inherit the logical thread time that triggered it through a set or character so that it can also be used to synchronize time. Transfer value to the future In the example above, we set :foo to 1 with which we did nothing. We can actually get this value from thread calling synchronization: in_thread to amp = sync :foo sample :ambi_lunar_land, amp: amp end sleep 2 set :foo, 0.5 Keep hinting that values transmitted through the set and stick must be safe from threads - i.e. fixed rings, numbers, symbols or frozen wires. Sonic Pi will throw an error if the value you are trying to store in the time state is invalid. 10.3 - Match patterns When obtaining and placing information in a time state, it is possible to use more complex keys than basic symbols such as :foo and :bar. You can also use URL-style wires called paths such as /foo/bar/baz. Once we start working with the trails, then we can start taking advantage of the sophisticated Sonic Pi pattern matching system to get and synchronise with 'similar' rather than 'identical' paths. Let's. Align any segment of the path Suppose we want to wait for the following event that has three segments of the trip: synchronization /*/*/* This will correspond to any Time State event with exactly three segments of the trip, regardless of their names. For example: a character /foo/bar/baz sign /foo/baz/quux sign /eggs/beans/toast sign /moog/synths/rule However, it will not correspond to paths with more or less segments of the track. The next one won't cue /foo/bar sign /foo/baz/quux/quaax character /eggs Each * means any content. Thus, we could match the paths with only one segment with /* or paths with five segments with /*/*/*/*/* Matching partial segments If we know what the segment will start or end with, we can use * with a partial segment name. For example: /foo/b*/baz will correspond to any path that has three segments, the first of which is foo, the last base and the middle segment can be anything that starts with b. Thus, this would correspond to: character /foo/bar/baz character /foo/baz/baz character /foo/beans/baz However, would not match the following: /foo/flibble/baz character /foo/abaz/baz character /foo/beans/baz/eggs You can also set * at the beginning of the segment specify the last segment characters: /foo/*/*zz/baz that will correspond to any 3 segment of the character or set where the first segment is foo, the last is the base and the middle segment ends with a zz such as cue/foo/whizz/baz. Corresponding segments of a nested path Sometimes you don't know how many segments of the path you want to match. In these cases, you can use a powerful double star: ** such as /foo/**/base to match: cue/foo/bar/baz character /foo/bar/beans/baz character /foo/baz character /foo/a/b/c/d/e/f/baz Matching Single Letters You can use the ? a single char character such as /?oo/bar/base to match: character /foo/bar/baz character /goo/bar/baz character /too/bar/base character /woo/bar/baz Corresponding multiple words If you know that the segment can be one of the selected number of words, you can use { and } matchers to specify a list of choices such as /foo/{bar,beans,eggs}/quux that will match only the following: character /foo/bar/quux character /foo/beans/quux character /foo/eggs/quux Multi-letter match Finally, you can match the choice of letters if you use [i] matchers to determine the list of choices such as /foo/[abc]ux/baz that will match only: character /foo/aux/baz character /foo/bux/baz character /foo/cux/baz You can also use - a character to determine the range of letters. For example/foo/[a-e]ux/baz that will match only: character /foo/aux/baz character /foo/bux/baz character /foo/cux/baz /foo/dux/base character /foo/eux/baz Combine matchers When syncing a call or get you can combine matchers in any order you see suitable for a powerful match of any time state event created by a character or set. Let's look at a crazy example: in_thread sync /?oo/[a-z]*/**/ba*/{quux,quaax}/ sample :loop_amen end sleep 1 cue /foo/beans/a/b/c/d/e/bark/quux/ OSC Pattern Matching For those curious, These matching rules are based on the Open Sound Control sample match specification explained in detail here: 11 - MIDI Once you've mastered converting code to music, you might ask yourself - what's next? Sometimes work restrictions exclusively within the Sonic Pi syntax and sound system can be exciting put you in a new creative position. However, sometimes it is essential to get out of code into the real world. We want two additional things: Being able to turn real-world actions into Sonic Pi coding events so they can use Sonic Pi's powerful timing model and semantics to control and manipulate objects in the real world Fortunately there is a protocol that has existed since the '80s that allows just such interaction - MIDI. There are an incredible number of external devices, including keyboards, controllers, sequencers and professional audio software that all support MIDI. We may use MIDI to receive data and use it to send data. Sonic Pi provides full support for the MIDI protocol, which allows you to connect your live code to the real world. Let's explore it further... 11.1 - MIDI In this section, we will learn how to connect a MIDI event-sending controller to Sonic Pi to control our synths and sounds. Go and grab a MIDI controller like a keyboard or control surface and let's go physical! Connecting the MIDI controller In order to get information from the external MIDI device in Sonic Pi, we must first connect it to our computer. Usually it will be via USB connection, although older equipment will have a DIN connector with 5 pins for which you will need hardware support for the computer (for example, some sound cards have MIDI DIN connectors). After you connect your device, launch Sonic Pi and see the IO part of the Preferences panel. You should see your device listed there. If not, try pressing the Reset MIDI button and see if it appears. If you still don't see anything, the next thing you need to try is to consult the MIDI configuration of your operating system to see if it sees your device. If all this fails, feel free to ask questions in the public chat room: Receiving MIDI events Once your device connects, Sonic Pi will automatically receive events. You can see for yourself by manipulating your MIDI device and looking at the character lumberjack in the lower-right corner of the app window below the log (if this is not visible go to Preferences->Editor->Show & Hide and enable the 'Show Character Log' tab). You will see a series of events such as: /midi:nanokey2_keyboard:0:1/note_off [55, 64] /midi:nanokey2_keyboard:0:1/note_on [53, 102] /midi:nanokey2_keyboard:0:1/note_off [57, 64] /midi:nanokey2_keyboard:0:1/note_off [53,64] /midi:nanokey2_keyboard:0:1/note_on [57, 87] /midi:nanokey2_keyboard:0:1/note_on [55, 81] /midi:nanokey2_keyboard:0:1/note_on [53,96] /midi:nanokey2_keyboard:0:1/note_off [55, 64] Once you can see a stream of messages like this, you have successfully connected your MIDI device. Congratulations, let's see what we can do with it! MIDI Time State These events are divided into two parts. First, there is the name of an event such as and secondly, there are event values such as [18,62]. Interestingly, these are two things we need to store information in Time State. Sonic Pi automatically installs incoming MIDI events at Time State. This means that you can get the latest MIDI value and synchronize the wait for the next MIDI value using everything we learned in section 10 of this tutorial. Controlling the code We have now connected the MIDI device, seen its events in the character log and found that our knowledge of the state of time is all we need to work with events, now we can start having fun. Let's build a simple MIDI piano: live_loop :midi_piano notes, speed = synchronisation /midi:nanokey2_keyboard:0:1/note_on synth :p iano, notes: notes end There are a few things that happen in the code above, including some issues. First, we have a simple live_loop that will be repeated forever by running the code between the block to / end. This was introduced in section 9.2. Second, we call sync wait for the next matching Time State event. We use a string that represents the MIDI message we are looking for (which is the same as shown in character loggers). Notice that this long string provides you with Sonic Pi's automatic complementary system, so you don't have to type everything manually. In the log, we saw that there are two values for each MIDI note at the event, so we assign two separate note and speed variables to the result. We've finally launched :p iano synth passes our note. Now, try. Type the code above, replace the sync key with a string that matches your specific MIDI device, and guess Run. Hey presto, you've got a working piano! However, you'll probably notice a few problems: first all notes are the same volume no matter how hard you hit the keyboard. This can be easily repaired using midi speed and converting to amplitude. Since MIDI has a range of 0->127, convert this number to a value between 0->1 we just need to divide it by 127: live_loop :midi_piano no, speed = synchronization /midi:nanokey2_keyboard:0:1/note_on synth :p iano, notes: notes, amp: speed / 127.0 end Update code and strike again. Now the speed of the keyboard is honored. Then let's get rid of that boring pause. Removing latency Before we remove the break, we need to know why it's there. In order for all synths and FX to be well timed on different capable CPUs, Sonic Pi pre-schedules sound for 0.5s by default. (Note that this added latency can be configured via fns set_sched_ahead_time! and use_sched_ahead_time). This 0.5s latency is added to our :p iano synth triggers as it is added to all synths powered by Sonic Pi. We usually really want this extra latency because it means all synths will be good in time. However, this only makes sense for code-driven synths using the game and In this case, we're. We. it :p iano synth with our external MIDI device and therefore do not want Sonic Pi to control the time for us. We can rule out this latency with use_real_time which disables latency for the current thread. This means you can use real-time mode for live loops that control time by syncing with external devices and keep the default latency for all other live loops. Let's see: live_loop :midi_piano to use_real_time notes, speed = sync /midi/nanokey2_keyboard/0/1/note_on synth :p iano, notes: notes, amp: speed / 127.0 Update your code to match the above code and guess Run again. Now we have a low latency piano with variable speed encoded in just 5 lines. It wasn't that easy! Getting value Finally, since our MIDI events go straight to the Time State, we can also use the get fn to download the last seen value. This does not block the current thread and returns zero if there is no value that you can override by adding a default value - see documents for profit). Remember that you can call to get in any thread at any time to see the latest matching Time State value. You can even time_warp to jump back in time and call to see past events... Now you're in control The exciting thing now is that you can now use the same code structures to sync and get MIDI information from any MIDI device and do whatever you want with the values. Now you can choose what your MIDI device will do! 11.2 - MIDI Out In addition to receiving MIDI events, we can also send MIDI events to run and manage external hardware synths, keyboards, and other devices. Sonic Pi provides a full set of fns to send a variety of MIDI messages such as: Note on - midi_note_on Note off - midi_note_off Control change - midi_cc Pitch band - midi_pitch_bend Clock ticks - midi_clock_tick There are many other supported MIDI messages - see the API documentation for all other fns that start with midi_. Connect to a MIDI device To send a MIDI message to an external device, we first connected it. See the Connect MIDI Controller subsect in section 11.1 for more details. Keep on what to do if you use USB, connecting to the device you're sending to (instead of receiving) is the same process. However, if you use classic DIN connectors, be sure to connect to the COMPUTER's MIDI port. You should see your MIDI device listed in the settings pane. Sending MIDI events Many midi_* fns work just like game, pattern and synth in that they send a message at the current (logical) time. For example, to spread the call midi_* fns you need to use sleep just like you did with the game. Let's midi_note_on: midi_note_on:e3, 50 This will send a MIDI note to the event on a connected MIDI device with speed 50. (Keep in touch that Sonic Pi will convert notes as :e3 to their corresponding MIDI number such as 52 in this case.) If your connected MIDI device is a synthesizer, you should be able to hear it play a note. To disable the use of midi_note_off: midi_note_off :e3 Select a MIDI device By default, Sonic Pi will send every MIDI message to all connected devices across all MIDI channels. This is to make it easier to work with one connected device without having to configure anything. However, sometimes a MIDI device will treat MIDI channels in a special way (perhaps each note has a separate channel) and you can also connect more than one MIDI device at a time. In more complex settings, you may want to be more selective about which MIDI device receives which message and on which channel. We can determine which device to send using the port: decide, using the device name as shown in preferences: midi_note_on :e3, port: moog_minitaur We can also determine which channel to send using the channel: opt (using a value in the range 1-16): midi_note_on:e3, channel: 3 Of course, we can also determine both at the same time to send to a particular device on a particular channel. : midi_note_on :e3, port: moog_minitaur, channel: 5 MIDI Studio Finally, a really fun thing to do is connect the audio output of your MIDI synthesizer to one of the audio inputs of your soundcard. You can then control your synth with the code using midi_* fns and manipulate the live_audio using live_audio and FX: with_fx :reverb, Room: 1 to live_audio :moog end live_loop :moog_trigger to use_real_time midi (octs :e1, 3).tick, sustain: 0.1 sleep 0.125 end (Fn midi is available as a handy shortcut to send and notes on and note events with one command. For more information, see its documentation). 12 - OSC In addition to MIDI, another way to learn information in and out of Sonic Pi is through a network using a simple protocol called OSC - Open Sound Control. This will allow you to send messages to and from external programs (both on your computer and on external computers) which opens up the potential for control far beyond MIDI which has limitations due to its 1980s design. For example, you can write a program in another programming language that sends and receives OSC (there are OSC libraries for almost every common language) and work directly with Sonic Pi. What you can use this for is limited only by your imagination. 12.1 - Receiving OSC By default when starting Sonic Pi listens to port 4560 for incoming OSC messages from programs on the same computer. This means that without any configuration you can send a Sonic Pi OSC message and it will be displayed in the character log just like incoming MIDI messages. This also means that each incoming OSC message is also automatically added to the Time Country, which means you can also use get and sync to work with Data - just like midi and live_loops - see sections 5.7 and 10.2 to recap how it works. Basic OSC listener Let's build a basic OSC listener: live_loop :foo to use_real_time a, b, c = sync /osc*/trigger/prophet synth :p rofeta, note: a, cutoff: b, sustain: c end In this example we described the OSC path /osc*/trigger/prophet that we synchronize. This can be any valid OSC path (all letters and numbers are supported and/used as in the URL to break the path to multiple words). The prefix/osc adds Sonic Pi to all incoming OSC messages, so we must send an OSC message with a path/trigger/prophet so that our synchronization will stop blocking and the prophet synth to be activated. Sending OSC to Sonic Pi We can send OSC sonic pi from any programming language that has an OSC library. For example, if we send OSC from Python we could do something like this: from pythonosc imports osc_message_builder from pythonosc imports udp_client sender = udp_client. SimpleUDPClient('127.0.0.1', 4560) sender.send_message('/trigger/prophet', [70, 100, 8]) Or, if we send an OSC from Clojure we could do something like this from REPL: (use 'overtone.core) (def c (osc-client

127.0.0.1 (osc-send c /trigger/prophet 70 100 8) Receiving from external machines for safety reasons, by default, Sonic Pi does not allow remote machines to send him OSC messages. However, you can enable remote machine support in preferences->IO->Network->Receive Remote OSC Messages. After you enable this, you can receive OSC messages from any computer on your network. Normally, the send machine will need to know your IP address (a unique identifier for your computer on the network - something like a phone number or email address). You can detect your computer's IP address by looking at the IO part of the settings. (If your machine accidentally has more than one IP address, hovering the mouse above the specified address will appear with a list of all known addresses). For example, some programs such as TouchOSC for iPhone and Android support sending OSC as standard features. So, after listening to remote machines and knowing your IP address, you can immediately start sending messages from apps like TouchOSC that allow you to build your own custom touch handlebars with sliders, buttons, dials, etc. This can give you a huge range of input options. 12.2 - Sending the OSC In addition to receiving and working with the OSC using Time State, we can also send OSC messages in time with our music (just as we can send MIDI messages on time with our music). We just need to know what IP address and port we're sending. Let's try: use_osc: use_osc localhost, 4560 osc /hello/world If you run the code above, you will notice that Sonic Pi sends an OSC message! That's because we set up the current machine and port to the default OSC in the port. This is essentially the same as publishing a letter for yourself - the OSC package was created, leaves Sonic Pi, gets to the network stack of the operating system which then routes packed back into Sonic Pi and then is received as a standard OSC message and is visible in the signer of the logger as the incoming message/ osc:127.0.0.1:4560 / hello / world. (Note that Sonic Pi automatically prefixes all incoming OSC messages with /osc, followed by hostnames and sender port.) Sending OSC to other programs Of course, sending OSC messages to yourself can be fun, but it's not that useful. The real benefit begins when we send messages to other programs: use_osc localhost, 123456 osc /hello/world In this case we assume that there is another program on the same machine that listens to port 123456. If there is, then he will receive a /hello/world OSC message with which he can do whatever he wants. If our program works on another machine, we need to know its IP address that we use instead of localhost: use_osc 192.168.10.23, 123456 osc /hello/world Now we can send OSC messages to any device that can be reached to us through our local networks and even the Internet! 13 - Multichannel Audio So far, in terms of sound production, we have investigated the launch of synths and recorded sounds via fns play, synth and pattern. Then they generated the sound played through our stereo speaker system. However, many computers also have the ability to enter audio, perhaps through a microphone, with the ability to send audio to more than two speakers. Often this option is possible by using an external sound card - they are available for all platforms. In this part of the tutorial we will look at how we can take advantage of these external sound cards and effortlessly work with multiple sound channels in and out of Sonic Pi. 13.1 - Sound In One a simple (and perhaps familiar) way to access sound inputs is used by our synth friend stating :sound_in synth: synth :sound_in This will work just like any synth such as synth :d saw with the exception that the generated sound will be read directly from the first sound card entry of your system. On laptops it is usually a built-in microphone, but if you have an external sound card, you can connect any audio input to the first input. Increasing duration One thing you might notice is that just like synth :d saw :sound_in synth takes only 1 rhythm because it has a standard envelope. If you want to keep open a little longer, change the ADSR envelope settings. For example, the following will keep the synth open by 8 beats before closing the link: synth :sound_in, sustain: 8 Adding FX Of course, just like any normal synth, you can easily layer on effects with fx block: with_fx :reverb do with_fx :distortion do synth :sound_in, sustain: 8 end end If you have connected the guitar to your first entrance, you should be able to hear it with distortion and echo until the synth ends as expected. You're free to use your synth as many times as you like simultaneously (just like you would with any normal synth). For example, the following will play two :sound_in synths at a time - one through distortion and one through reverb: with_fx :distortion do synth :sound_in, sustain: 8 end with_fx :reverb do synth :sound_in, sustain: 8 end Multiple Inputs You can choose which audio input you want to play with the input: decide. You can also specify stereo input (two consecutive inputs) using :sound_in_stereo synth. For example, if you have a sound card with at least three inputs, you can treat the first two as a stereo stream and add distortion, and the third as a stream and add a reverb with the following code: with_fx :distortion do synth :sound_in_stereo, sustain: 8, input: 1 end with_fx :reverb to synth :sound_in, sustain : 8, input: 3 end Potential questions However, although this is a useful technique , there are several limitations to this approach. First, it works only for a certain duration (because it has an ADSR envelope) and secondly, there is no way to switch FX after the launch of synth. Both of these things are typical requirements when working with external audio feeds such as microphones, guitars, and external synthesizers. So we'll take a look at Sonic Pi's solution to the problem of manipulating a (potentially) endless stream of live audio inputs: live audio. 13.2 - Live Audio The :sound_in synth as described in the previous section provides a very flexible and familiar method for working with input sound. However, as it has also been discussed, it has several problems when working with a single sound input as a single instrument (such as a voice or guitar). By far the best approach to working with a single continuous stream of sound is to use live_audio. The named audio input live_audio shares several limitations of the underlying design live_loop (hence the similar name). First, it must have a unique name and second, only one live_audio with that name can exist at any time. Let's live: live_audio :foo This code will act in a similar way to synth :sound_in with some key differences: it works forever (until you explicitly stop it) and you can dynamically move it to new FX contexts. Working with FX At the start, live_audio works exactly as you might expect to work with FX. For example, to run a live audio stream with added reverb, simply use the :reverb FX block: with_fx:reverb to live_audio:foo end However, given that live_audio works forever (at least until you stop it) it would be quite limiting if, like typical synths, live audio was tied inside :reverb FX for its entire existence. Fortunately this is not the case and is designed to move easily between different FX. Try the code above to hear live audio comes directly from the first entry of your sound card. For example, if you use a laptop, it will usually be outside the built-in microphone, so it is recommended to use headphones to stop feedback. Now, while still listening to live audio from the reverb sound card, change the code to the following: with_fx:echo to live_audio :foo end Hit Run, and you will immediately hear a sound playing through echo FX and no longer through reverb. If you wanted them both, just re-edit the code and press Run: with_fx :reverb to with_fx :echo to live_audio :foo end End It is important to point out that you can call live_audio :foo from any thread or live loop and it will move the synth to the current FX context of that thread. Therefore, you could easily have multiple live loops calling live_audio :foo at different times, resulting in the FX context being automatically replaced for some interesting results. Stopping live audio Unlike standard synths, because live_audio doesn't have an envelope, it will continue to work forever (even if you delete the code, just as the function is still defined in memory if you delete the code in the editor). To stop it, you need to use :stop arg: live_audio :foo, stop It can easily be restarted by calling without :stop arg again: live_audio :foo Additionally everyone starts live audio synths stop when you press the global Stop button (as with all other running synths and FX). Stereo input With respect to audio channels, by default live_audio acts similarly to the sound_in synth in that it requires a single mono input stream of audio and converts it into a stereo stream using the specified panning. However, just like :sound_in_stereo, it is also possible to tell live_audio to read two consecutive audio inputs and treat them like a left and right channel directly. This is achieved through :stereo opt. For example, to treat input 2 as a left signal and input 3 as a right signal, you need to configure the input: opt for 2 and enable stereo as follows: live_audio :foo, stereo: truth, input: 2 Keep hinting that once you start a live audio stream in stereo mode, you can't change it in without stopping and starting. Similarly, if you start it in default mode, you can't switch to stereo without starting and stopping the stream. 13.3 - Sound Out So far in this section we have looked at how to get more streams of sound in Sonic Pi – either through the use of :sound_in synth or via a powerful live_audio system. In addition to working with multiple inbound audio streams, Sonic Pi can also create multiple streams of audio. This is achieved through :sound_out FX. Output Contexts Let's Go Recapitulate about how Sonic Pi synths and FX output their sound to their current FX context. For example, consider the following: with_fx:reverb do #C with_fx:echo do #B sample :bd_haus # End End The easiest way to figure out what's going on with an audio stream is to start with the deepest audio context and work your way out. In this case, the deepest context is marked as A and is :bd_haus triggered. The sound for this goes directly into its context which is B - :echo FX. This then adds an echo to the incoming sound and excedites it into its context which is C - :reverb FX. This then adds reverb to the upcoming sound and exits into its top-level context - left and right speakers (outputs 1 and 2 in your audio card). The sound flows outward with a stereo signal to the end. Sound Out FX The above behavior applies to all synths (including live_audio) and most FX with the exception of :sound_out. FX :sound_out FX does two things. First, it transfers its sound to the external context as described above. Secondly, it also goes out its sound directly to the output on your sound card. Let's look: with_fx:reverb to with_fx:sound_out, output: 3 to # B pattern :bd_haus # End End In this example, our :bd_haus pattern outputs its sound into its external context which is :sound_out FX. This in turn transfers its sound to its external context :reverb FX (expected). However, a mixture also comes on the third output of the sound card system. Audio generated within :sound_out therefore has two destinations - :reverb FX and audio card output 3. Stereo out As we have seen, by default, :sound_out FX outputs a mixture of stereo inputs to a particular channel while transmitting the stereo feed in an external context (as expected). If producing the mixture is not exactly what you want to do, there are a number of alternative options. First, using mode: Decide that you can select the output of only the left or only the right input signal to the audio card. Or you can use :sound_out_stereo FX to output to two consecutive sound card outputs. See functional documentation for more information and examples. Direct Out As we also saw, the default behavior for :sound_out and :sound_out_stereo is to send audio to both their external context (as is typical for the entire FX) and to the specified output on your sound card. However, occasionally you can only send to your sound card output and not to the external context (and therefore you have no chance of mixing the sound and sending it to standard output channels 1 and 2). This is possible using a standard FX opt amplifier: which works on sound after FX has managed to manipulate sound: with_fx :sound_out, output: 3, amp: 0 to # B pattern :loop_amen # End In the above sample :loop_amen is sent to its external context, :sound_out FX. This then sends the mix to audio card output 3, then multiplies the sound by 0 which essentially mutes it. It is this silenced signal that is then sent to :sound_out external context that is the standard output. Therefore, with this code, the default output channels will not receive any sound, and channel 3 will receive a drum amen mix. 14 - Conclusions This concludes the introductory guide of Sonic Pi. I hope you learned something along the way. Don't worry if you feel you haven't understood everything - just play and have fun and you'll pick things up in your own time. Feel free to dive back in when you have a question that might be covered in one of the sections. If you have any questions that aren't covered by the guide, then jump on the Sonic Pi community forums and ask your question there. You'll find someone friendly and ready to lend a hand. Finally, I invite you to take a deeper look at the rest of the documentation in this help system. There are numerous features that are not covered in this guide that await your discovery. So play, have fun, share your code, perform for your friends, show screens and remember: No mistakes, just opportunities. Sam Aaron A - MagPi Articles Appendix A collects all Sonic Pi articles written for MagPi magazine. Dive into the topics These articles are not intended to be read in any strict order and contain a lot of material to bridge from the tutorial itself. Instead of trying to teach you all Sonic Pi, they instead focus on a certain aspect of Sonic Pi and cover it in a fun and affordable way. Read the Magpies You can see them in their glorious professional typed form in the free PDF downloads of The MagPi here: Suggest a topic If you don't see the topic you're interested in in these articles - why not suggest it? The easiest way to do this is to tweet your suggestion @Sonic_Pi. You never know - your suggestion may be the subject of the following article! A.1 - Top five tips The most important lesson you can learn with Sonic Pi is that there are really no mistakes. The best way to learn is to try. Try a lot of different things, stop worrying about whether your code sounds good or not, and start experimenting with as many different synths, notes, FX and opts as you can. You're going to discover a lot of things that make you laugh because they sound really awful and some real gems that sound really amazing. Just throw away things you don't like and keep the things you do. The more 'errors' you make, the faster you learn and discover your personal sound coding. 2. Use FX Let's say you have already mastered the Sonic Pi basics of making With a pattern, play? What's next? Did you know that Sonic Pi supports over 27 studio FX to change the sound of your code? FX are like fancy image filters in drawing programs, except that instead of blurring or making something black and white, you can add things like reverb, distortion and echo to your sound. Think of it as sticking cables from your guitar to the pedal effects of your choice and then into the amplifier. Luckily, Sonic Pi makes using FX really easy and doesn't require cables! All you have to do is choose which part of your code you want to add FX and wrap it with FX code. Let's look at the example. Let's say you had the following code: sample :loop_garzul 16.times make a sample :bd_haus sleep 0.5 end If you wanted to add FX to :loop_garzul pattern, just make it tuck inside a with_fx block like this: with_fx :flanger to pattern :loop_garzul end 16.times do not pattern :bd_haus sleep 0.5 end Now if you wanted to add FX to bass drum , go and wrap and do it with with_fx also: with_fx :flanger to sample :loop_garzul end with_fx :echo to 16.times to sample :bd_haus sleep 0.5 end end Recall, any code you can wrap inside with_fx and all the sounds created will pass through that FX. 3. Parameterize your synths To truly discover your sound coding, you'll soon want to know how to modify and control synths and FX. For example, you might want to change the duration of your note, add more echoes, or change the time between echoes. Fortunately, Sonic Pi gives you an incredible level of control to do just that with special things called optional parameters or opt for short. Let's take a quick look. Copy this code into the workspace and hit run: sample :guit_em9 Ooh, nice guitar sound! Let's start playing with that. How about a gear change? sample :guit_em9, rate: 0.5 Hey, what's that rate: 0.5 bit I just added at the end? It's called opt. All Sonic Pi synths and FX support them and there is a lot to play for. They are also available for FX. Try this: with_fx :flanger, feedback: 0.6 do sample :guit_em9 end Now, try increasing that feedback to 1 hear some crazy sounds! Read the documents for full details on all the many opties available to you. 4. Live Code The best way to quickly experiment and explore Sonic Pi is to live code. This allows you to start some code and constantly change and tweak it while it's still playing. For example, if you don't know what the break parameter is doing to the sample, just play around. Let's try! Copy this code into one of your Sonic Pi workspaces: live_loop:experiment to sample :loop_amen, cutoff: 70 sleep 1.75 end Now, hit run and you'll hear a slightly muffled drum break. Now change the cutoff: value to 80 and hit run again. Do you hear the difference? Try 90, 100, 110... Once you get to hang the use live_loops you won't come back. Whenever I make a live coding gig I rely on live_loop how much the drummer relies on his chopsticks. For more information about live broadcasting code, see Section 9 of the built-in textbook. 5. Surf random streams Finally, one thing I like to do is cheat by getting Sonic Pi to compose things for me. A really great way to do that is to use randomization. It may sound complicated, but it really isn't. Let's see. Copy this to the spare workspace: live_loop :rand_surfer make use_synth :d saw = (scale :e2, :minor_pentatonic, num_octaves: 2) 16.times to play notes.choose, release: 0.1, cutoff: rrand(70, 120) sleep 0.125 end end Now, when you play this, you will hear a steady stream of random notes from the scale :e2:minor_pentatonic toyed :d with the sad chicken. Wait, wait! It's not a melody, I can hear you yelling! Well, here's the first part of the magic trick. Every time we go around live_loop we can tell Sonic Pi to reset a random stream to a familiar point. It's a bit like going back to time in the TARDIS with the Doctor up to a point in time and space. Let's try - let's use_random_seed line 1 live_loop: live_loop :rand_surfer do use_random_seed 1 use_synth :d think notes = (scale :e2, :minor_pentatonic, num_octaves: 2) 16.times to play notes.choose, release: 0.1, cutoff: rrand(70, 120) sleep 0.125 end end Now, every live_loop loops around, random stream resets. This means that he selects the same 16 notes each time. Hey presto! Instant melody. Here's a really exciting bit. Change the seed value from 1 to another number. Let's say 4923. Wow! Another tune! So, just by changing one number (random seeds), you can explore as many melodic combinations as you can imagine! It's code magic. A.2 - Live Coding Laser beams sliced through waves of smoke as the subwoofer pumped bass deep into the bodies of the crowd. The atmosphere was ripe with a great mix of synths and dance. However, there was something wrong at this nightclub. Projected in bright colors above the DJ booth was futuristic text, moving, dancing, flickering. These weren't fancy visuals, it was just a projection of Sonic Pi running on Raspberry Pi. The occupant of the DJ booth did not spin the discs, but wrote, edited and evaluated the code. Live. This is Live Coding. This may sound like a spin-out story from a futuristic nightclub, but coding music like this is a growing trend and is often described as Live Coding ( ). One of the recent directions this approach to making music has taken are Algorave ( ) - events where artists like me code music for people to dance to. However, you don't have to be in a nightclub to Live Code – with Sonic Pi v2.6+ you can do it anywhere you can take a raspberry Pi and a pair of headphones or some speakers. Once you have reached the end of this article, you will program your own beats and modify them live. Where you go after that will be limited only by your imagination. Live Loop Key to Live Coding with Sonic Pi is mastering live_loop. Let's look at one: live_loop :beats do sample :bd_haus sleep 0.5 end There are 4 basic ingredients for a live_loop. The first is his name. Our live_loop up is called :beats. You're free to call live_loop whatever you want. Crazy. Get creative. I often use names that communicate something to the audience about the music they do. The second ingredient is up to a word that indicates where live_loop begins. The third is the closing statement indicating where live_loop ends, and in the end there is the body live_loop which describes what loop will repeat itself - this is the part between the to and the end. In this case, we repeatedly play a sample bass drum and wait half a stroke. It produces a nice regular bass rhythm. Go ahead, copy it into an empty Sonic Pi buffer and hit run. Boom, Boom, Boom! Redefining On-the-fly Ok, so what's so special about live_loop? So far it seems like a glorified loop! Well, the live_loops is that you can redefine them on the fly. That means while they're still running, you can change what they do. This is a secret code with Sonic Pi. Let's have a live_loop: choral_drone :choral_drone make a pattern :ambi_choir, rate: 0.4 sleep 1 end Now press the Run button or press Meta-r. Now you're listening to some beautiful choir sounds. Now, while it's still playing, change the rate from 0.4 to 0.38. Hit the run again. Woah! Did you hear the choir changed the note? Change it back to 0.4 to go back to the way it was. Now, lower it to 0.2, down to 0.19 and then back to 0.4. See how changing just one parameter on the fly can give you real control over music? Now play yourself with the rate - choose your own values. Try negative numbers, really small numbers and big numbers. have fun! Sleep is important One of the most important lessons live_loops is that they need rest. Consider the following live_loop: live_loop :infinite_impossibilities make a sample :ambi_choir end If you try this code, you will immediately see Sonic Pi complaining that live_loop not sleep. This is a security system that's made itself in! Take a moment to think about what this code requires of your computer. That's right, he's asking the computer to play an infinite amount of choir samples at zero time. Without a security system bad computer will try to do this and crash and burn in the process. Remember, your live_loops must contain sleep. Combining Sounds Music is a lot of things that happen at the same time. Drums at the same time as bass at the same time as the vocals at the same time as the guitars... In computing, we call it consonation and Pi provides us with an incredibly simple way to play things at the same time. Simply use more than one live_loop! live_loop :beats do sample :bd_tek with_fx :echo, phase: 0.125, mix: 0.4 sample :d rum_cymbal_soft, sustain: 0, release: 0.1 sleep 0.5 end live_loop :bass to use_synth :tb303 synth :tb303, notes: :e1, release: 4, cutoff: 120, cutoff_attack: 1 sleep 4 end Here, we have two live_loops, one looping quickly making crazy bass sound. One of the interesting things about using multiple live_loops is that each of them manages its own time. This means that it is really easy to create interesting polyrhythmic structures and even play with the phase style Steve Reich's Piano Phase notes = (ring :E4, :Fs4, :B4, :Cs5, :D5, :Fs4, :E4, :Cs5, :B4, :Fs4, :D5, :Cs5) live_loop :slow do play notes.tick, release: 0.1 sleep 0.3 end live_loop :faster play notes.tick, release: 0.1 sleep 0.295 end Bringing it all together In each of these tutorials we will finish with a final example in the form of a new piece of music that draws from all the presented ideas. Read this code and see if you can imagine what it does. Then copy it to the fresh Sonic Pi clipboard and hit Run and actually hear what it sounds like. Finally, change one of the numbers or comments and uncomment things out. See if you can use this as a starting point for a new performance, and have the most fun! See you next time... with_fx :reverb, room: 1 to live_loop :time to synth :p rophet, release: 8, notes: :e1, cutoff: 90, amp: 3 sleep 8 end live_loop :machine to sample :loop_garzul, rate: 0.5, ending: 0.25 sample :loop_industrial, beat_stretch: 4, amp: 1 sleep 4 end live_loop :kik do sample :bd_haus, amp: 2 sleep 0.5 end with_fx :echo to live_loop :vortex do # use_random_seed 800 notes = (scale :e3, :minor_pentatonic, num_octaves: 3) 16.times do not play notes.choose, release: 0.1, amp: 1.5 sleep 0.125 end end end A.3 - Coded beats One of the most exciting and disruptive technical achievements in modern music was the invention of samplers. These were boxes that allowed you to record any sound into them, and then manipulate and play those sounds in many interesting ways. For example, you can take an old record, find a solo drum (or break it), record it in your sampler, and then play it again at half speed to provide the basis for your latest beats. This is how early hip-hop music was born and today it is almost impossible to find electronic music that does not include some patterns. Using patterns is a really great way to easily introduce new and interesting elements into your live coded performances. So, where can you get a sampler? Well you already have one - it's your Raspberry Pi! The built-in live coding app Sonic Pi has an extremely powerful sampler built into its Come on, come on, with him! Amen Break One of the most consistent and recognizable drum patterns is called Amen Break. It was first performed in 1969 in the song Amen Brother by Winstons as part of a drum break. However, when it was discovered by early hip-hop musicians in the 80s and used in samplers, it began to be heavily used in a wide range of other styles such as drum and bass, breakbeat, hardcore techno and breakcore. I'm sure you're excited to hear that it was also built right in Sonic Pi. Clean the buffer and insert the following code: sample :loop_amen Hit Run and boom! You're listening to one of the most influential drums in dance music history. However, this pattern was not known for playing as a single shot, it was built for a loop. Beat Stretching Let's loop the Amen Break using our old friend live_loop featured in this tutorial last month: live_loop :amen_break to sample :loop_amen sleep 2 end OK, so it's a loop, but every time there is a boring pause. This is because we asked him to sleep for 2 beats and with a default BPM of 60:loop_amen the sample only lasts 1,753 beats. Therefore, we have a silence of 2 - 1,753 = 0.247 beats. Even though it's short, it's still noticeable. To solve this problem, we can use beat_stretch: decide to ask Sonic Pi to stretch (or reduce) the pattern to match the specified number of beats. Sonic Pi pattern and synth fns give you a lot of control through optional parameters such as amp:, cutoff: and release:. However, the term optional parameter is a real snack, so we just call them decide to keep things nice and simple. live_loop :amen_break make a pattern :loop_amen, beat_stretch: 2 sleep 2 end Now we dance! Although, we may want to speed it up or slow it down to suit the mood. Playing with Time OK, so what if we want to change styles in old school hip hop or breakcore? One simple way to do this is to play with the time - or in other words mess with the pace. It's super easy in Sonic Pi - just insert use_bpm into the live loop: live_loop:amen_break make use_bpm 30 sample :loop_amen, beat_stretch: 2 sleep 2 end While you're rapping over these slow beats, notice that we're still sleeping for 2, and our BPM is 30, but everything is on time. The beat_stretch opt works with the current BPM to make sure everything just works. Here's the fun part. While the loop is still alive, change 30 to use_bpm line to 50. Woah, everything's gotten faster, but held back in time! Try to go faster - up to 80, to 120, now go crazy and hit 200! Filtering Now we can live patterns of loops, let's look at some of the most fun opts provided by sample synth. The first is cut off: which controls the sampler boundary filter. By default, this is disabled, but you can easily turn it on: live_loop :amen_break do use_bpm 50 sample :loop_amen, beat_stretch: 2, cutoff: 70 sleep 2 end Go ahead and change cutoff: decide. For example, increase it to 100, hit the Run and wait for the loop to spin in a circle to hear the sound change. Notice that low values like 50 sound mild and bassy and high values such as 100 and 120 are more full-sounding and raspy. That's because it's cutoff: opt will shred high-frequency parts of the sound just like a lawn mower cuts off the tip of the grass. Cutting off: Opt as a length setting - determining how much grass is left. Cutting Another great tool for the game is the FX cutter. This will chop (cut) the sound upwards. Wrap the sample line with the FX code like this: live_loop :amen_break do use_bpm 50 with_fx :slicer, phase: 0.25, wave: 0, mix: 1 to pattern :loop_amen, beat_stretch: 2, cutoff: 100 end of sleep 2 end Notice how the sound bounces up and down a little bit more. (You can hear the original sound without FX by changing the mix: decide on 0.) Now try to play with the phase: make up your mind. This is the rate (in beats) of the cutting effect. A smaller value such as 0.125 will cut faster, and higher values like 0.5 will cut more slowly. Notice that successive halving or doubling of the phase: the opts wave tends to always sound good. Finally, change the wave: opt for one of 0, 1 or 2 and hear it change the sound. These are different forms of waves. 0 wave saw, (hard, fade) 1 is a square wave (hard, hard) and 2 is a triangular wave (fade, fade). Putting it all together Finally, let's go back in time and revisit the early Bristol drum and bass scene with this month's example. Don't worry too much about what it all means, just type it, hit Run, then start living by coding by changing the opt numbers and see where you can take it. Please share what you create! See you next time... use_bpm 100 live_loop :amen_break to p = [0.125, 0.25, 0.5].choose with_fx :slicer, phase: p, wave: 0, mix: rrand(0.7, 1) to r = [1, 1, 1, -1].choose pattern :loop_amen, beat_stretch: 2, foot: r, amp: 2 end of sleep 2 end live_loop :bass_drum make a pattern of :bd_haus, cutoff: 70, amp: 1.5 sleep 0.5 end live_loop :landing by bass_line = (knitting :e1, 3, [:c1, :c2].choose, 1) with_fx :cutter, stage: [0.25, 0.5].choose, invert_wave: 1, wave: 0 to s = synth :square, notes: bass_line.tick, sustain: 4, cutoff: 60 control s, cutoff_slide: 4, cutoff: 120 end sleep 4 end A.4 - Synth Riffs Whether it's haunting drift of tudling oscillators or detuned punch of saw piercing waves through the mix , the main synthete plays an essential role on any electronic track. In last month's edition of this series of textbooks, we covered how to code our beats. In this guide we will cover how to encode the three basic components of synth riff - color, melody and rhythm. OK, so power your Raspberry Pi, open the Sonic Pi v2.6+ let's do a little bit of Timbral capabilities An essential part of any synth riff is changing and playing with timbre sounds. Timbre in Sonic Pi can be controlled in two ways - by selecting different synths for dramatic change and setting up different synths, it opts for more subtle modifications. We can also use FX, but it's for another tutorial ... Let's create a simple live loop in which we continuously change the current synth: live_loop :timbre to use_synth (ring :tb303, :blade, :p rophet, :saw, :beep, :three).tick play :e2, attack: 0, release: 0.5, cutoff: 100 sleep 0.5 end Look at the code. We simply tick through the ring synth names (it will circulate through each of them in turn repeating the list over and over again). We forward this synth name to use_synth fn (function) that will live_loop current synth company. We also play notes :e2 (E in the second octave), with a release time of 0.5 beats (half a second on the given BPM of 60) and with a break: decide on the 100th hear how different synths have very different sounds even though they all play the same note. Now experiment and play. Change the release time to higher and smaller values. For example, change the attack: and release: decide to see how different fade in/out times have a huge impact on sound. Finally change the break: decide to see how different break values also massively affect color (values between 60 and 130 are good). See how many different sounds you can create just by changing a few values. Once you've mastered this, just head to the Synths tab in Help for a complete list of all synths and all available opt for each individual synth support to see how much power you have under your fingertips of coding. Timbre Timbre is just a fancy word that describes the sound of sound. If you play the same note with different instruments such as violin, guitar or piano, the height (how high or low it sounds) would be the same, but the sound quality would be different. This sound quality - the thing that allows you to know the difference between piano and guitar is color. Melodic composition Another important aspect of our main synth is the selection of notes we want to play. If you already have a good idea, then you can easily create a ring with notes and mark them: live_loop :riff to use_synth :p rohet riff = (ring :e3, :e3, :r, :g3, :r, :r, :r, :a3) play riff.tick, release: 0.5, cutoff: 80 sleep 0.25 end Here, we defined our melody with a ring that includes both notes such as :e3 and rests presented :r. Then we use a .tick to cycle through each note to get a repetitive riff. Melody's car isn't always easy to come up with a nice riff at first. Instead, it's often easier to ask Sonic Pi for a selection of random riffs and choose the one you like best. To do this, we need to combine rings, randomisation and random seeds. Let's look at the example: live_loop :random_riff make use_synth :d use_random_seed 3 notes = (scale :e3, :minor_pentatonic).shuffle play notes.tick, release: 0.25, cutoff: 80 sleep 0.25 end There are a few things going on - let's look at them in turn. First, we state that we use random seeds 3. Well, the useful thing is that when we set the seeds, we can predict what the next random value will be - it's the same as the last time we set the seeds at 3! Another useful thing to know is that mixing a note ring works the same way. In the example above, we are basically looking for a third shuffle on the standard mixing list - which is also the same every time we always set random seeds to the same value just before mixing. We're finally just going through our mixed notes to play the riff. Here's where the party starts. If we change the random seed value to another number, say 3,000, we get a completely different mixing of notes. Now it is very easy to explore new melodies. Simply select the list of notes we want to mix (scales are a great slowing point), then select the seeds we want to mix from. If we don't like the melody, just change one of those two things and try again. Keep repeating until you like what you hear! The randomization of Sonic Pi's pseudo randomization is not really random, but what is called pseudo random. Imagine rolling the dice 100 times and writeing the result of each roll on a piece of paper. Sonic Pi has the equivalent of this list of results that it uses when searching for a random value. Instead of rolling actual dice, it just selects the next value from the list. Placing a random seed is just jumping to a certain point on that list. Finding your rhythm Another important aspect of our riff is the rhythm - when to play a note and when not to. As we saw above we can use :r in our rings to insert rest. Another very powerful way is to use spreads that we will cover in a future tutorial. Today we will use randomization to help us find a rhythm. Instead of playing each note we can use tentatively to play a note with a certain likeness. Let's look: let's live_loop: random_riff use_synth :d :random_riff use_synth :d 30 use_random_seed = (scale :e3, :minor_pentatonic).shuffle 16.times to play notes.tick, release: 0.2, cutoff: 90 if one_in(2) sleep 0.125 end A really useful fn to know is one_in that will give us a real or false value with the stated likely. Here we use the value 2 so that on average every time every two calls for one_in this will return to the truth. In other words, 50% of the time the truth will come back. Using higher values will make it more common to return the false introduction of more space into the riff. Notice that we have added a little iteration here with the 16th time. That's because we want to reset our random seed value every 16 notes so that our rhythm repeats every 16 times. This does not affect mixing as this is still done immediately after the seeds have been placed. We can use the size of the iteration to change the length of the riff. Try to change 16 to 8 or even 4 or 3 and see how it affects the rhythm of the riff. Putting everything together OK, so let's combine everything we learned together into one final example. See you next time! live_loop :random_riff to # uncomment to bring in: #synth:blade, notes: :e4, release: 4, cutoff: 100, amp: 1.5 use_synth :d saw use_random_seed 43 notes = (scale :e3, :minor_pentatonic, num_octaves: 2).shuffle.take(8) 8.times to play notes.tick, release: rand(0.5), cutoff: rrand (60, 130) if one_in(2) sleep 0.125 end live_loop :d rums do use_random_seed 500 16.times do sample :bd_haus, rate: 2, cutoff: 110 if rand < 0.35 sleep 0.35 sleep 0.125 end live_loop :bd to pattern :bd_haus, cutoff: 100, amp: 3 sleep 0.5 end A.5 - Acid Bass It is impossible to look through the history of electronic dance music without watching the huge impact of the tiny Roland TB-303 synthesizer. It's the secret sauce behind the original acid bass sound. Those classic squealing and squelching TB-303 bass riffs can be heard from the early Chicago House scene to newer electronic artists such as Plastikman, Squarepusher and Aphex Twin. Interestingly, Roland never intended for TB-303 to be used in dance music. It was originally created as a practice aid for guitarists. They imagined that people would program them to play bass lines to jam along. Unfortunately, there were a number of problems: they were a little fiddly for the program, they didn't sound particularly good as bass guitar replacements and were quite expensive to buy. Deciding to cut the losses, Roland stopped making them after 10,000 units were sold and after many years of sitting on the guitar players' shelves, they could soon be found in second hand shop windows. These lonely discarded TB-303s were waiting to be discovered by a new generation of experimenters who began to use them in ways Roland never imagined making new crazy noises. Acid House was born. Although getting your hands on the original TB-303 isn't so easy you'll be pleased to know that you can turn your Raspberry Pi into one using the power of Sonic Pi. Look, light up Sonic Pi and throw this code into an empty tampon and hit Run: use_synth:tb303 play :e1 Instant acid bass! Let's play... Squelch da Bass First, let's build a living arpeggiator to make things fun. In the last guide, we looked at how riffs can only be a ring of notes that we pass one after the other, repeating when we get to the end. Let's create a live loop that does just use_synth: use_synth :tb303 live_loop :squelch do n = (ring :e1, :e3).tick play n, release: 0.125, cutoff: 100, res: 0.8, wave: 0 sleep 0.125 end See each line. On the front line we set the default synth to be tb303 with use_synth fn. On line two we create a live loop called :squelch that will only loop in a circle and circle. Line three is where we create our riff - a ring of notes (E in octaes 1, 2 and 3) that we simply tick off with .tick. We define n to represent the current note in the riff. A character equal to just means assigning a value to the right to the name on the left. It will be different every time round loop. For the first time round, n will be set to :e1. The second time it will be :e2, followed by :e3, and then back to :e1, cycling round forever. Line 4 is where we actually trigger our :tb303 synth. Here are some interesting opts here: release:, cutoff:, res: and wave: what we will discuss below. Line five is our dream - we're looking for a live loop to spin every 0.125 or 8 times per second on a default BPM of 60. Line six is the end of a live loop. This just says Sonic Pi where the end of the living loop is. While you're still figuring out what's going on, type the code above and press the Run button. You should hear a :tb303 kick into action. Now, this is where the action is: let's start living coding. While the loop is still alive, change the break: decide on 110. Now press the Run button again. You should hear the sound become a little sharper and more squelchy. Dial in 120 and hit run. Now 130. Listen to the more severed values make it sound more penetrating and intense. Finally, lower it to 80 when you feel like a vacation. Then repeat as many times as you like. Don't worry, I'll still be here... Another opt worth playing is res:. This controls the level of filter resonance. High resonance is characteristic of the sounds of sour bass. Currently we have res: set to 0.8. Try turning it up to .85, then 0.9, and finally 0.95. You may find that a breakup such as 110 or more will make it easier to hear differences. Finally go crazy and call at 0.999 for some crazy noises. At such a high level, you hear the cutoff filter resonate so much that it starts making its own sounds! Finally, for a big impact on the color, try to change the wave: decide on 1. This is a selection of source oscillators. The default setting is 0 which is a wave of pilatoote. 1 is a pulse wave and 2 is a triangular wave. Of course, try different riffs by changing notes in the ring or even picking notes from scales or chords. Have fun with your first acid bass synth. Deconstruction of the TB-303 The design of the original TB-303 is actually quite simple. As you can see from the following diagram, there are only 4 basic parts. The first is the oscillator wave - the raw ingredients of sound. In this case, we have a square wave. Then there is the oscillator amplitude envelope controls the amplifier of the square wave over time. They approached the Sonic Pi attack:, decay:, maintenance: and release: decided together with their colleagues on the level. For more information, see section 2.4 Duration with envelopes in the built-in guide. Then we pass our shrouded square wave through a resonant low pass filter. It cuts off higher frequencies, as well as having that nice resonance effect. Now the fun begins. The severed value of this filter is also controlled by its own envelope! That means we have incredible control over timbre sound playing with both of these envelopes. Let's look: use_synth:tb303 with_fx:reverb, room: 1 to live_loop :space_scanner to play :e1, cutoff: 100, release: 7, attack: 1, cutoff_attack: 4, cutoff_release: 4 sleep 8 end End For each standard envelope decide, there is an cutoff_ equivalent opt in :tb303 synth. So, to change the time when the attack was interrupted, we cutoff_attack: make up your mind. Copy the above code to the empty Clipboard and hit Run. You'll hear a crazy sound seeing and coming out. Now start playing. Try changing cutoff_attack: time up to 1, then 0.5. Now try 8. Notice that I went through everything :reverb FX for an extra atmosphere - try another FX to see what works! Bringing it all together Finally, here's the part I composed using the ideas in this textbook. Copy it to an empty Clipboard, listen for a while, and then start living coding your own changes. Look at the crazy noises you can make with him! See you next time... use_synth :tb303 use_debug false with_fx :reverb, room: 0.8 to live_loop :space_scanner to with_fx :slicer, stage: 0.25, amp: 1.5 to co = (line 70, 130, steps: 8).tick play :e1, cutoff: co, release: 7, attack: 1, cutoff_attack: 4, cutoff_release: 4 sleep 8 end live_loop :squelch to use_random_seed 3000 16.times to n = :e1, :e2, :e3).tick play n, release: 0.125, cutoff: rrand(70, 130), res: 0.9, wave: 1, amp: 0.8 sleep 0.125 end end end A.6 - Musical Minecraft Hello and welcome back! In previous tutorials, we focused solely on Sonic Pi's musical capabilities - (turning your Raspberry Pi into a performance-ready musical instrument). So far we have learned how: Live Code - changing sounds in flight, At some huge beats, generate powerful synth leads, recreate the famous TB-303 acid-bass sound. There is so much more to show for it (which we will explore in future editions). This month, however, let's look at something Sonic Pi can do that you probably didn't realize: control Minecraft. Hello Minecraft World OK, let's start. Raise raspberry pi, set Minecraft Pi on fire and create a new world. Now run Sonic Pi and resize and move your windows so you can see both Sonic Pi and Minecraft Pi at the same time. On the fresh Clipboard, type mc_message Hello Minecraft from Sonic Pi! Nwo Nwo Run. Boom! Your message appeared in Minecraft! How easy was that? Stop reading this for a moment and play with your own messages. have fun! Sonic Transporter Now let's do some research. The standard option is to reach for the mouse and keyboard and start walking around. It works, but it's pretty slow and boring. It would be much better if we had some kind of teleportative machine. Well, thanks to Sonic Pi, we have one. Try this: mc_teleport 80, 40, 100 Crikey! It's been a long road up. If you hadn't been in flying mode, you'd have fallen all the way back to earth. If you double-tap the space to enter flying mode and re-beam, you'll be left hovering where you are. What do these numbers mean? We have three numbers describing the coordinates of where in the world we want to go. We give each number a name - x, y and z: x - as much as left and right (80 in our example) y - how high we want to be (40 in our example) z - how far back and forth (100 in our example) By choosing different values for x, y and z we can teleport anywhere in our world. Try! Select different numbers and see where you can end up. If the screen turns black, it's because you've beamed underground or into a mountain. Just select a higher y value to get back above ground. Keep exploring until you find a place you like... Using ideas so far, let's build a Sonic Teleporter that produces a fun teleport sound while whizzing us all over the Minecraft world: mc_message Preparing for a teleport.... sample :ambi_lunar_land, rate: -1 sleep 1 mc_message 3 sleep 1 mc_message 2 sleep 1 mc_message 1 sleep 1 mc_teleport 90, 20, 10 mc_message Whoosh! Magic Blocks Now you find a nice place, let's start building. You can do what you're used to and start clicking furiously with your mouse to place blocks under the pointer. Or you could use the magic of Sonic Pi. Try this: x, y, z = mc_location mc_set_block :d inja, x, y +5, z Now look up! There's melon in the sky! Take a moment to look at the code. What did we do? On the front line, we grabbed Steve's current location as variables x, y and z. This corresponds to our coordinates described above. We use these coordinates in the fn mc_set_block which will place the block of your choice on the specified coordinates. In order for something to be higher in the sky, we just need to increase the value of y which is why we add 5. Let's make a long mark from them: live_loop :melon_trail to x, y, z = mc_location mc_set_block :d inja, x, y-1, z sleep 0.125 end Now, jump into Minecraft, make sure you're in flying-mode (double tap space if not) and fly all over the world. Look behind you to see a nice trail of melon blocks! See what twisted patterns you can make in the sky. Live Coding Minecraft Those of you who followed this over the last few months it is likely that your mind will be blown at this time. The trail of melons is pretty cool, but the most exciting part of the previous example is that you can live_loop with Minecraft! For those who don't know, live_loop special magical ability of Sonic Pi that no other programming language has. It allows you to run multiple loops at the same time and allows you to change them while they work. They are incredibly powerful and amazing fun. I live_loops to play music in nightclubs with Sonic Pi - DJs use discs and I use live_loops :-) However, today we will live with both music and Minecraft. Start. Run the code above and start making a melon trail again. Now, without stopping the code, just simply change :d inja to :brick and hit run. Hey, presto, you're making a brick trail now. How simple it was! Fancy music? Easy. Try this: live_loop :bass_trail do tick x, y, z = mc_location b = (ring :d inja, :brick, :glass).look mc_set_block b, x, y-1, z note = (ring :e1, :e2, :e3).look use_synth :tb303 play notes, release: 0.1, cutoff: 70 sleep 0.125 end Now, while's playing start changing the code. Change the types of blocks - try :water, :grass or your favorite block type. Also, try changing the break value from 70 to 80, and then up to 100. Isn't this fun? Let's put it all together Let's combine everything we've seen so far with a little extra magic. Let's combine our teleportation ability with block placement and music to make a Minecraft music video. Don't worry if you don't understand everything, just type it and play by changing some values while it's live. Have fun and see you next time... live_loop :note_blocks mc_message This is Sonic Minecraft with_fx :reverb to with_fx :echo, stage: 0.125, repetitions: 32 to tick x = (range 30, 90, step: 0.1.look y = 20 z = -10 mc_teleport x, y, z ns = (scale :e3, :minor_pentatonic) n = ns.shuffle.choose bs = (knit :glass, 3, :p axe, 1) b = bs.look synth :beep, notes: n, issue: 0.1 mc_set_block b , x+20, n-60+y, z+10 mc_set_block b, x+20, n-60+y, z-10 sleep 0.25 end end live_loop :beats to sample :bd_haus, cutoff: 100 sleep 0.5 end A.7 - Bizet Beats After our short trip to the fantastic world of coding Minecraft with Sonic Pi last month, let's go back to music. Today we're going to bring a classic opera dance piece straight into the 21st century. Outrageous and distracting Let's jump into the time machine back in 1875. A composer named Bizet has just finished his latest opera Carmen. Unfortunately, like many exciting and distracting new musical acts people didn't like it at first because it was too outrageous and different. Unfortunately Bizet died ten years before the opera gained huge international success and became one of the most famous often performed operas of all time. In sympathy with this tragedy let's take one of the main themes from Carmen and turn it into a modern format of music that is also too outrageous and different for most people in our time - live coded music! Decoding Habanera Trying to live with the whole opera would be a bit of a challenge for this tutorial, so let's focus on one of the most famous parts - bass line to Habanera: This can seem extremely unhearing to you if you haven't studied music notation yet. However, as developers we see musical notation as just another form of code - only it represents instructions to the musician instead of the computer. So we need to figure out a way to decode it. A.7 - Notes Are arranged from left to right like words in this magazine, but also have different heights. The height on the result represents the height of the note. The higher the note on the result, the higher the note flow. At Sonic Pi, we already know how to change the pitch height of the note - we either use high or low numbers such as play 75 and play 80 or we use note names: play :E and play :F. Luckily each of the vertical positions of the musical partide represents a certain name of the note. Check out this handy look up table: Rests Music scores are an extremely rich and expressive kind of code capable of communicating many things. So it shouldn't be so much of a surprise that music ratings can not only tell you which notes to play, but also when not to play the notes. In programming, this is quite equivalent to the idea of zero or null - the absence of value. In other words, not reproducing a note is like the absence of a note. If you look closely at the result, you will see that it is actually a combination of black dots with lines representing notes for the game and squid things representing the rest. Luckily Sonic Pi has a very handy holiday display: :r, so if we run: play :r it actually plays silence! We could also write a game of :rest, play zero or play falsely which are all equivalent ways of presenting a holiday. Rhythm Finally, there is another thing to learn how to decode in a being - the timing of notes. In the original battle you will see that the notes are connected to thick lines called beams. The second note has two of these beams which means it lasts 16. The other notes have one beam, which means they last 8. The rest has two squiggly rays which means it also represents the 16th century. When we try to decode and explore new things a very handy trick is to make everything as similar as possible to try to see any relationships or patterns. For example, when we rewrite a note purely at 16, you can see that our notation simply turns into a nice sequence of notes and rests. Re-encoding Habanera We are now in start translating this bass line at Sonic Pi. Let's encod these notes and rest in the ring: (ring :d, :r, :r, :a, :f5, :r, :a, :r) Let's see how it sounds. Throw it into a live loop and mark it through it: live_loop:habanera do play (ring :d, :r, :r, :a, :f5, :r, :a, :r).tick sleep 0.25 end Fabulous, that instantly recognizable riff springs to life through your speakers. It took a lot of effort to get here, but it was worth it - a high five! Moody Synths Now we have a bass line, let's recreate some of the ambience of the opera scene. One synth to try is :a blade that is moody 80s style synth lead. Let's try the initial note :d went through the cutter and reverb: live_loop :habanera to use_synth :fm use_transpose -12 play (ring :d, :r, :r, :a, :f5, :r, :a, :r.tick sleep 0.25 end with_fx :reverb to live_loop :space_light to with_fx :slicer, stage: 0.25 to synth :blade, notes: :d, release: 8, cutoff: 100, amp: 2 end sleep 8 end Now, try the other notes in the bass line: :a and :f5. Remember, you don't have to hit stop, just modify the code while the music plays and hit the run again. Also, try different values for the cutter phase: opt for 0.5, 0.75, and 1. Putting it all together Finally, let's combine all the ideas so far into a new remix of Habanera. You may notice that I included another section of the bass line as a comment. Once you've typed it all in a fresh buffer hit Run to hear the composition. Now, without stopping, unwrap the other line by removing it # and hit the run again - how amazing it is! Now, start crushing it around and have fun. use_debug false bizet_bass = (ring :d, :r, :r, :a, :f5, :r, :a, :r) #bizet_bass = (ring :d, :r, :r, :Bb, :g5, :r, :Bb, :r) with_fx :reverb, room: 1, mix: 0.3 do live_loop :bizet do with_fx :slicer, phase: 0.125 do synth :blade, note: :d4, release: 8, cutoff: 100, amp: 1.5 end 16.times do tick play bizet_bass.look, release: 0.1 play bizet_bass.look - 12, release: 0.3 sleep 0.125 end end end live_loop :ind do sample :loop_industrial, beat_stretch: 1, cutoff: 100, rate: 1 sleep 1 end live_loop :drums do sample :bd_haus, cutoff: 110 synth :beep, note: 49, attack: 0, release: 0.1 sleep 0.5 end A.8 - Become a Minecraft VJ Everyone has played Minecraft. You will all have built amazing structures, designed tricky traps and even created elaborate trolley lines controlled by redstone switches. How many of you performed with Minecraft? We bet you didn't know you could use Minecraft to create amazing visuals just like a professional VJ. If your only way to modify Minecraft was with a mouse, you'd have a hard time changing things fast enough. Luckily for you, your Raspberry Pi comes with a code-controlled version of Minecraft. It also comes with an app called Sonic Pi that makes minecraft encoding a no Easy, but also incredibly fun. In today's article we will show you some of the tips and tricks we used to create performances in nightclubs and music venues around the world. Start... Start with a simple warm-up exercise to refresh with the basics. First open raspberry pi, then light Minecraft and Sonic Pi. In Minecraft, create a new world, and in Sonic Pi choose a fresh clipboard and write in this code: mc_message Let's start... Press the Run button and you'll see a message in the Minecraft window. Okay, we're ready to start, let's have fun...... Sandstorms When we use Minecraft to create visuals we try to think about what both will look interesting and also be easy to generate from the code. One nice trick is to create a sandstorm by releasing sand blocks from the sky. For this, we only need a few basic fns: sleeping - to insert delays between actions mc_location - find our current location mc_set_block- set sand blocks at a specific rrand location - to allow us to generate random values within the live_loop range - to allow us to constantly make it rain sand If you are unfamiliar with any of the built-in fns such as rrand , just type a word on your Clipboard, click on it, and then click the Control combo keyboard to bring the built-in documentation. Alternatively, you can navigate to the lang tab in the help system and then search for fns directly along with all the other exciting things you can do. Let it rain a little before we unleash the full force of the storm. Grab your current location and use it to create several sand blocks in the sky nearby: x, y, z = mc_location mc_set_block :p ax, x, y+20, z +5 sleep 2 mc_set_block :p ax, x, y+20, z +6 sleep 2 mc_set_block :p ax, x, y+20, z +7 sleep 2 mc_set_block :p ax, x , y +20 , z +8 When you hit the Run, you may need to look around a little as the blocks may start falling behind you depending on which direction you are currently facing. Don't worry, if you missed them just hit Run Again for another batch of sand rain - just make sure you're looking the right way! Let's go over what's going on here. On the front line we grabbed Steve's location as coordinates with fn mc_location and put them in vars x, y and z. Then on the following lines we mc_set_block fn to place some sand on the same coordinates as Steve, but with some modifications. We chose the same x coordinate, y coordinate 20 blocks higher and then successively larger z coordinates so that the sand fell in a line away from Steve. Why don't you take that code and start playing with it? Try adding more lines, changing your sleep time, try mixing :p searing with :gravel and choose different coordinates. Just and have fun! Live Live Freed OK, it's time for the storm to rage with the release of full-strength live_loop - Sonic Pi's magical ability that unleashes the full force of live coding - changing code on the fly while it works! live_loop :sand_storm to x, y, z = mc_location xd = rrand(-10, 10) zd = rrand (-10, 10) co = rrand (70, 130) synth :cnoise, attack: 0, release: 0.125, cutoff: co mc_set_block :p ax, x +xd, y+20, z+zd sleep 0.125 end What fun! We spin pretty fast (8 times a second) and during each loop we find Steve's location like before, but then we generate 3 random values: xd - the difference for x which will be between -10 and 10 zd - the difference for z also between -10 and 10 co - cut off the value for the low pass filter between 70 and 130 Then we use those random values in fns synth and mc_set_block giving us sand falling randomly locations around Steve along with percussion rain sound from :cnoise syntha. For those of you new to live loops - this is where the fun really starts with Sonic Pi. While the code is working and the sand is pouring in, try changing one of the values, perhaps the sleep time to 0.25 or :type of sand block to :gravel. Now hit the run again. Hey Presto! Things changed without stopping the code. This is your way through performing like a real VJ. Keep practicing and changing things. How different can you make visuals without stopping the code? Epic Block Patterns Finally, another great way to generate interesting visuals is to generate huge patterned walls flying in and out. For this effect, we will need to go from randomly placing the blocks to placing them in the ordered way. We can do this by nesting two sets of iterations (press the Help button and navigate to section 5.2 of the Iterations guide and loops for more iteration backgrounds). Funny |xd| after it does this means that xd will be set for each iteration value. So the first time will be 0, then 1, then 2... Etc. By nesting two iterations together like this we can generate all the coordinates for the square. Then we can randomly select a block of type from the ring blocks for interesting effect: x, y, z = mc_location bs = (ring :gold, :d iamond, :glass) 10.times do |xd| 10.times |yd| mc_set_block bs.choose, x + xd, y +yd, z end Pretty neat. While we're having fun here, try changing bs.choose to bs.tick to move from a random pattern to a more ordinary one. Try to change the types of blocks and more adventurous than you might want to try to put it within live_loop so that the forms are constantly changing automatically. Now, for the VJ final - change two 10th times to 100th time and hit run. Kaboom! A huge giant wall of randomly placed bricks. Imagine how long it would take you to build it by hand with a mouse! Double-tap space to enter fly-mode and launch by for some great visual effects. Don't stop here though - use your imagination to conjure up some cool ideas, then use the sonic Pi encoding power to make it real. When you've been practicing enough to cluff the lights and put on a VJ show for your friends! A.9 - Surfing Random Streams Back in Episode 4 of this tutorial series we briefly looked at randomization while coding some sizzling synth riffs. Given that randomization is such an important part of my live coding DJ kits, I thought it would be helpful to cover the basics in more detail. So get your lucky hat on and let's surf some random streams! There's no random first thing to learn that might really surprise you when playing with Sonic Pi's randomization functions is that they're not really random. What does that really mean? Well, let's try a few tests. First, imagine a number in your head between 0 and 1. Keep him there and don't tell me. Let me guess... Was it 0.321567? No? Bah, I'm obviously not good at this. Let me go one more time, but let's ask Sonic Pi to pick a number this time. Light a Sonic Pi v2.7+ and ask him for a random number, but again don't tell me: print rand Now to discover... Was it 0.75006103515625? I do! Ha, I see you're a little skeptical. Maybe it was just a happy assumption. Let's try again. Press the Run button again and see what we get... What? 0.75006103515625 again? This obviously can't be random! You're right, it's not. What's going on here? The fancy word in computer science here is determinism. It just means that nothing is random and that everything is destined to be. Your version of Sonic Pi is destined to always return 0.75006103515625 in the above program. This might sound pretty useless, but I can assure you it's one of the most powerful parts of Sonic Pi. If you stick to it, you'll learn how to rely on the deterministic nature of Randomization of Sonic Pi as a foundation building block for your live compositions and DJ sets. Random Melody When Sonic Pi boots actually load into memory a sequence of 441,000 pre-generated random values. When you call a random function such as rand or rrand, this random stream is used to generate your result. Each random function call consumes value from this stream. Therefore, the 10th random call will use the 10th value from the stream. Also, every time you press the Run button, the stream resets for that ride. Because of this, I was able to predict the result for Rand and why the 'random' melody was the same every time. Everyone's version of Sonic Pi uses exactly the same random flow that is very important when we start sharing our pieces with each other. Let's use this knowledge to generate a repeatable random melody: 8.times let's play rrand_i(50, 95) sleep 0.125 end Type this into the backup clipboard and hit Run. Hear a melody consisting of random notes between 50 and 95. When it's over, hit Run again to hear the exact same melody again. The practical randomization functions Sonic Pi come with a number of useful functions for working with a random stream. Here's a list of some of the most useful: rand - Simply returns the following value in a random stream rrand - Returns a random value within the range rrand_i - Returns a random integer within the range one_in - Returns true or false with a given rule dice - Imitates rolling the cube and returns a value between 1 and 6 choose - Selects a random value from the list Check Your Documentation in The Help System for Detailed Information and Examples. Reset your stream Although being able to repeat a series of selected notes is key so you can play the riff on the dance floor, it may not be exactly the riff you want. Wouldn't it be great to try a series of different riffs and choose the one we like best? This is where the real magic begins. We can manually set up a stream with fn use_random_seed. In computer science, random seeds are the starting point from which a new stream of random values can rise and flourish. Let's use_random_seed 0 3.times to play rrand_i(50, 95) sleep 0.125 end Great, we get the first three notes of our random melody above: 84, 83 and 71. However, we can now change the seeds to something else. How about this: use_random_seed 1 3.times to play rrand_i(50, 95) sleep 0.125 end Interesting, we get 83, 71 and 61 . You may notice that the first two numbers here are the same as the last two numbers before - this is no coincidence. Remember that a random stream is just a giant list of pre-valid values. Using random seeds simply brings us to a point on that list. Another way of thinking about it is to imagine a large deck of pre-shuffled cards. Using random seeds is cutting the deck at a certain point. The amazing part of this is that it is this ability to jump around a random stream that gives us tremendous power when making music. Let's repeat our random 8-note melody with this new flow reset power, but let's also throw in a live loop so we can experiment while it plays: live_loop :random_riff to use_random_seed 0 8.times to play rrand_i(50, 95), release: 0.1 sleep 0.125 end end Now, while it's still playing, change the seed value of 0 to something else. Try 100, what about 999. Try your own values, experiment and play around - see which seeds generate the riff you like best. Bringing this month's textbook closer together is a pretty technical dive into the work of Sonic Pi's randomization functionality. I hope it has given you an insight into how it works and how you can start using randomization in a reliable way to create repeatable patterns within your it is important to emphasise that you can use repeatable randomization wherever you want. For example, you can randomize the amplitude of notes, rhythm time, reverb volume, current synth, FX mix, etc. In the future, we will carefully consider some of these applications, but for now let me leave you with a short example. Type the following on the backup clipboard, hit run, then start changing seeds, hit Run again (while still playing) and explore the different sounds, rhythms, and melodies you can make. When you find a beautiful one, remember the number of seeds so you can go back to it. Finally, when you've found a few seeds you like, put on a live coded performance for your friends by simply switching between your favorite seeds to create a whole piece. live_loop :random_riff do use_random_seed 10300 use_synth :p rophet s = [0.125, 0.25, 0.5].choose 8.times to r = [0.125, 0.25, 1, 2].choose n = (scale :e3, :minor).choose co = rrand(30, 100) play n, release: r, cutoff: co sleep with end live_loop :d rums to use_random_seed 2001 16.times to r = rand(0.5 , 10) sample :drum_bass_hard, rate: r, amp: rand sleep 0.125 end end A.10 - Controlling Your Sound So far during this series we have focused on starting sounds. We found that we can run many synths embedded in Sonic Pi with a game or synth and how to run prerecorded patterns with a pattern. We also looked at how we can wrap these stimulated sounds within the FX studio, such as echo and distortion using with_fx commands. Combine this with Sonic Pi's incredibly accurate timing system and you can produce a wide array of sounds, beats and riffs. However, after carefully selecting the options of a particular sound and running it, there is no possibility to mess with it while it is playing off? Wrong! Today you will learn something very powerful - how to control running synths. Basic sound Let's make a nice simple sound. Light up Sonic Pi and in a fresh buffer type the following: synth :p rophet, note: :e1, release: 8, cutoff: 100 Now press the Run button on the top left to hear the beautiful sound of the tudling synth. Go ahead, press it a few times to get a feel for it. All right, done? Let's start controlling it! Synth Nodes A little-known feature in Sonic Pi is that fns play, synth and pattern, restore something called SynthNode that represents running sound. You can record one of these SynthNodes using a standard variable and then control it at a later point in time. For example, let's change the value of cutoff: opt after 1 beat: sn = synth :p rofed, notes: :e1, release: 8, cutoff: 100 sleep 1 control sn, cutoff: 130 Let's look at each line in return: First we activate :p soft with using synth fn as usual. However, we also record a result in a variable called sn. We could have called this something else entirely synth_node such as jane or jane - the name does not matter. However, it's important to choose a name that's meaningful to you for your performances and for people who read your code. I chose the sn because it's a nice short mnemonics for synth nodes. On line 2, we have a standard sleep command. This does nothing special - it just asks the computer to wait 1 beat before moving on to the next line. Line 3 is where the control party begins. Here we use the control fn to tell our synthNode to change the break value to 130. If you press the Run button, you will hear that :p rophet synth starts playing as before, but after 1 beat it will switch to a sound much brighter. Modulable options Most Sonic Pi synths and FX opts can be changed after launch. However, this is not the case for all of them. For example, the envelope decides the attack:, decay:, maintenance: and release: it can only be set when starting synth. It's easy to figure out which one is selected and what can't be changed - just head to the documentation for a specific synthesizer network or FX, then scroll down to the individual option documentation and look for expressions It can be changed after the start. For example, attack documentation :beep synth: decide to make it clear that it cannot be changed: Default: 0 Must be zero or higher You cannot change after setting yourself Scaled with the current BPM value Multiple changes While synth is done you are not limited to change only once - you are free to change it as many times as you want. For example, we can turn our :p into a mini arpeggiator with the following: notes = (scale :e3, :minor_pentatonic) sn = synth :p rophet, note: :e1, release: 8, cutoff: 100 sleep 1 16.times to control sn, note: notes.tick sleep 0.125 end In this code snippet we just added a few extra things. First, we defined a new variable called a note that contains notes that we would like to cycle through (arpeggiator is just a fancy name for something that circulates through a list of notes in order). Second, we replaced our only call for control with an iteration that called him 16 times. In each call to control me .tick through our ring of notes that will automatically repeat after we reach the end (thanks to the incredible power of Sonic Pi rings). For a little variety try to replace .tick with .choose and see if you can hear the difference. Keep in mind that we can change multiple charges at the same time. Try to change the control line to the following and listen to the difference: control muzzle, note: notes.tick, cutoff: rrand(70, 130) Slide When we control SynthNode, reacts exactly in time and immediately changes the value of the opta to a new one as if you pressed a button or pressed a switch asking for a change. This may sound rhythmic and percussive - especially if the opt controls aspect of colors such as interruption:. However, sometimes you don't want the change to happen right now. Instead, you might want to smoothly move from the current value to the new one as if you had moved the slider or dial. Of course, Sonic Pi can also do this using _slide: decides. Each opt that can be modified also has a special _slide: decide which allows you to specify the slide time. For example, amp: has amp_slide: and cutoff: there are cutoff_slide:. These slides choose to work a little differently than anyone else who decides to speak synth notes on how to behave the next time they're controlled. Let's look: sn = synth :p rophet, note: :e1, release: 8, cutoff: 70, cutoff_slide: 2 sleep 1 control sn, cutoff: 130 Notice how this example is exactly the same as before, except with the addition of cutoff_slide:. It says that the next time this synth has its cutoff: opt controlled, it will take 2 beats to slide from current value to new. Therefore, when we use the control, you can hear the slide cut off from 70 to 130. It creates an interesting dynamic feeling to sound. Now try changing cutoff_slide: Time to a shorter value such as 0.5 or longer, such as 4 to see how it changes sound. Remember, you can glide any of the variable opts on exactly this and every _slide: the value can be completely different so you can have cutoff sliding slowly, amp skating fast and pan slipping somewhere in between if that's what you're looking to create... Bonding Let's look at a brief example that shows the power of synth control after they are activated. Notice that you can also glide FX just like synths though with slightly different syntax. See section 7.2 of the built-in textbook for more information about FX control. Copy the code to the backup clipboard and listen. Don't stop there - play with the code. Change slide time, change notes, synth, FX, and sleep time, and see if you can turn it into something else entirely! live_loop :moon_rise do with_fx :echo, mix: 0, mix_slide: 8 do |fx| control fx, mix: 1 notes = (scale :e3, :minor_pentatonic, num_octaves: 2).shuffle sn = synth :p rophet , sustain: 8, note: :e1, cutoff: 70, cutoff_slide: 8 control sn, cutoff: 130 sleep 2 32.times to control sn, notes: notes.tick, pan: rrand(-1, 1) sleep 0.125 end end end A.11 - Tracking the Beat Last month in this series we took a deep technical dive into the randomization system on which Sonic Pi is based. We explored how we can use it to determine the addition of new levels of dynamic control over our code. This month we'll continue our technical lead and draw attention to the Sonic Pi unique tick system. By the end of this article you will be ticking your way through rhythms and riffs on your way to being a live coding DJ. Counting When making music we often want to do a different thing depending on what rhythm it is. Sonic Pi has a special beat counting system called tick to give you precise control over when a rhythm actually happens, and even supports multiple beats at its own pace. Let's have a game - the rhythm we just need to call a tick. Open the fresh clipboard, type the following, and press Run: puts tick #=> 0 This will return the current rhythm: 0. Note that even if you press the Run button a few times it will always return 0. That's because every ride starts a fresh rhythm counting from 0. However, while running is still active, we can improve the rhythm as many times as we want: puts tick #=> 0 puts tick #=> 1 puts tick #=> 2 Whenever you see the #=> symbol at the end of the code line it means that this line will record the text on the right. For example, putting foo #=> 0 means that the code puts foo prints 0 on the log at that point in the program. Checking beat we saw the tick doing two things. Increases (adds one) and restores the current rhythm. Sometimes we just want to look at the current rhythm without the magnification that we can do via appearance: puts tick #=> 0 puts tick #=> 1 puts look #=> 1 puts the look #=> 1 In this code we double-mark the rhythm, and then we call look twice. We will see the following values in the log: 0, 1, 1, 1. The first two ticks returned 0, then 1 as expected, and then the two layouts just returned the last beat value twice which was 1. Rings So now we can improve the rhythm with a tick and check the rhythm with appearance. What next? We need something to tick off. Sonic Pi uses rings to represent riffs, melodies and rhythms, and the tick system is specifically designed to work closely with them. In fact, the rings have their own version of the tick that does two things. First, it acts as a regular tick and increases the rhythm. Second, it looks up the value of the ring using the beat as an index. Let's look: puts (ring :a, :b, :c).tick #=> :a .tick is a special dotted version of the tick that will return the first value of the ring :a. We can grab each of the values in the ring by calling on .tick multiple times: puts (ring :a, :b, :c).tick #=> :a puts (ring :a, :b, :c).tick #=> :b puts (ring :a, :b, :c).tick #=> :c puts (ring :a, :b, :c).tick #=> :a puts look #=> 3 See the log and you will see :a, :b, :c and then :a again. Notice the layout returns 3. Calls to .tick act just as regular calls for tagging - they increment the local rhythm. A Live Loop Arpeggiator The real power comes when you mix a tick with rings and live_loops. When combined we have all the tools we need to build and understand a simple arpegiator. We only need four things: a ring containing the notes that we want to go through. A means of incrementing and getting beats. The ability to play a note based on the current beat. Loop Loop arpegiator repeats itself. These concepts can be found in the following code: notes = (ring 57, 62, 55, 59, 64) live_loop :arp to use_synth :d saw play notes.tick, release: 0.2 sleep 0.125 end Let's look at each of these lines. First we define our ring of notes that we will play all the time. Then we live_loop called :arp that goes round in circles for us. Every time we round live_loop we set our synth to :d, and then we play the next note in our ring using .tick. Remember that this will gradually enhance our beat counter and use the latest beat value as an index in our note ring. Finally, we wait for an eighth of a beat before going around in a loop again. Multiple simultaneous beats A really important thing to know is that ticks are a local live_loop. This means that each live_loop has its own independent beat counter. This is much more powerful than a global metronome and a beat. Let's look at this in action: notes = (ring 57, 62, 55, 59, 64) with_fx :reverb to live_loop :arp to use_synth :d pulse play notes.tick + 12, Release: 0.1 sleep 0.125 End End live_loop :arp2 to use_synth :d saw play notes.tick - 12, release: 0.2 sleep 0.75 end Clashing Beats The big cause of confusion with the Sonic Pi tick system is when people want to mark multiple rings in the same live_loop : use_bpm 300 use_synth :blade live_loop :foo do play (ring :e1, :e2, :e3).tick play (scale :e3, :minor_pentatonic).tick sleep 1 end Although each live_loop has its own independent beat counter, we call .tick twice within the same live_loop. This means that the rhythm will be incremented twice each time the circuit. It can produce some interesting polyrhythms, but it's often not what you want. There are two solutions to this problem. One option is to manually call the tick at the beginning live_loop and then use .look to use the current rhythm in each live_loop. Another solution is to transfer a unique name to each call of a .tick such as .tick(:foo). Sonic Pi will then create and track a separate beat counter for each named tag you use. That way you can work with as many beats as you need! See the named ticks section in the 9.4 built-in guide to more information. Bonding Let's gather all this knowledge about ticks, rings and live_loops for the final entertainment example. As usual, don't treat this like a ready-made piece. Start changing things and play with it and see what you can turn it into. See you next time... use_bpm 240 notes = (scale :e3, :minor_pentatonic).shuffle live_loop :foo to use_synth :blade with_fx :reverb, Repetitions: 8, room: 1 to tick co = (line 70, 130, steps: 32).tick(:cutoff) play (Octobers :e3, 3).look, cutoff: co, amp: 2 play notes.look, amp: 4 sleep 1 end end live_loop :bar to tick sample :bd_ada if (spread 1 , 4).look use_synth :tb303 co = (line 70, 130, steps: 16).look r = (line 0.1, 0.5, steps: 64).mirror.look play notes.look, release: r, cutoff: co sleep 0.5 end A.12 - Sample Cutting Way back in episode 3 of this Sonic Pi series we watched how to loop, stretch and filter one of the most famous drum breaks of all time - Amen Break. In this tutorial we'll take this step further and learn how to slice it, shuffle the slices and glue it back together in a completely new order. If this sounds a bit crazy to you, don't worry, everything will become clear and you'll soon be mastering a powerful new tool for your live coded sets. Sound as data Before we begin, let's take a brief moment to figure out how to work with patterns. By now you've all hoped to be playing with sonic pi's powerful sampler. If not, there's no time like the present! Lift the Raspberry Pi, launch Sonic Pi from the programming menu, type the following into the fresh clipboard, and then press the Run button to hear the prerecorded drum beat: pattern :loop_amen The sound recording is simply presented as data - lots of numbers between -1 and 1 representing the tops and troughs of the sound wave. If we play those numbers okay, we'll get the original sound. However, what prevents us from playing them again in a different order and making a new sound? How are patterns actually recorded? It's actually pretty simple once you understand the basic physics of sound. When you make a sound - for example by hitting the drum, the noise travels through the air in a similar way to how the surface of the lake ripples when you throw a pebble into it. When those waves get to your ears, your eardrum moves sympathetically and turns those moves into the sound you hear. If we want to record and play sound, we therefore need a way to capture, store and play these waves. One way is to use a microphone that acts as an eardrum and moves back and forth as the sound ripples it. The microphone then turns its position into a tiny electrical signal which is then measured many times per second. These measurements are then presented as a series of numbers between -1 and 1. If we were to plot a sound visualization it would be a simple graph of data over time on the x axis and the position of the microphone/speaker as a value between -1 and 1 on the y axis. You can see an example of such a chart at the top of the diagram. Playing part of the pattern So, how about Sonic Pi play the pattern back in the second order? To answer this question, we need to look at the beginning: and finish: it opts for a pattern. This allows us to control the initial and final positions of our reproduction of numbers representing sound. The values for both of these optas are represented as a number between 0 and 1 where 0 represents the beginning of the sample and 1 is the end. So, to play the first half of Amen Break, we just need to determine the finish: from 0.5: sample Finish: 0.5 We can add at the start: value play an even smaller part of the pattern: sample :loop_amen, start: 0.25, finish: 0.5 For fun, You can even have a finish: the value of the opta is before the start: and will play the section backwards: pattern :loop_amen, start: 0.5, finish: 0.25 Reordering the playback of the sample Now that we know that the pattern is simply a list of numbers that can be reproduced in any order and how to play a certain part of the pattern we can now start having fun playing the pattern back in the 'wrong' order. Let's take our Amen Break and chop it into 8 slices of equal size, then stir in the pieces. See diagram: top A) presents a chart of our original sample data. Chopping on 8 slices gives us b) - notice that we gave each slice a different color to distinguish them. At the top, you can see the initial and ending values of each part. Finally C) is one of the possible re-orders of slices. Then we can play this back to create a new rhythm. See code to do this: live_loop :beat_slicer to slice_idx = rand_i(8) slice_size = 0.125 s = slice_idx * slice_size f = s + slice_size sample :loop_amen, start: s, finish: f sleep sample_duration :loop_amen, start: s, finish: f end we choose a random piece for the game that should be a random number between 0 and 7 (remember that we start counting to 0). Sonic Pi has a practical function for this: rand_i(8). Next, we store this random slice index in a variable slice_idx. We define our slice_size which is 1/8 or 0.125. The slice_size is to convert slice_idx value between 0 and 1 so we can use it as our beginning: make up your mind. We calculate the starting position by multiplying slice_idx with slice_size. We calculate the target position f by adding slice_size to the starting position with. Now we can reproduce the slice of the pattern by attaching the value with and f to the start: and finish: decides on the pattern. Before we play the next slice we need to know how long to sleep which should be the duration of the slice of the pattern. Fortunately, Sonic Pi has been sample_duration that accepts all the same opts as a pattern and simply restores duration. Therefore, passing sample_duration beginning: and finish: decides, we can find out the duration of one slice. We wrap all these codes in live_loop so that we continue to choose new random slices for the game. Bonding Let's merge everything we've seen so far into a final example that shows how we can take a similar approach to combine randomly sliced beats with some bass to create the beginning of an interesting song. Now it's your turn - take the code below as a starting point and see if you can take it in your direction and create something new... live_loop :sliced_amen to n = 8 s = line(0, 1, steps: n.choose f = s + (1.0 / n) sample :loop_amen, 2, start: s, finish: f sleep 2.0 / n end live_loop :acid_bass do with_fx :reverb, room: 1, repetitions: 32, amp: 0.6 to tick n = (octs :e0, 3).look - (knitting 0, 3 * 8, -4, 3 * 8).look co = rrand(70, 110) synth :beep, note: n+ 36, release: 0.1, wave: 0, cutoff: co synth :tb303, notes: n, release: 0.2, wave: 0, cutoff: co sleep (ring 0.125, 0.25).look end end A.13 - Code a Probabilistic Sequencer In the previous episode of this Sonic Pi series, we explored the power of randomization to introduce diversity, surprise and change to our live coded songs and performances. For example, we randomly dialed chart notes to create endless melodies. Today we will learn a new technique that uses randomization for rhythm - probabilistic beats! Probability Before we can start doing new beats and synth rhythms we need to quickly dive into the basics of probability. This may sound scary and complicated, but it's actually as simple as rolling dice - honestly! When you take a regular 6 side board game dice and roll this what's actually going on? Well, first you'll roll or 1, 2, 3, 4, 5 or 6 with exactly the same chance of getting any of the numbers. In fact, given that it's 6 side dice, on average (if you roll lots and lots of times) you'll throw 1 every 6 pitches. That means you have a 1 in 6 chance of throwing 1. We can mimic dice rolls in Sonic Pi with fn cubes. Let's roll one 8 times: 8.times does not put dice sleep 1 end Notice how the log prints values between 1 and 6 just as if we're rolling real cubes ourselves. A.13 - Random Beats Now imagine you had a drum and every time you needed to hit it you rolled the dice. If you rolled 1, you hit the drum, and if you rolled any other number you didn't. Now you have a probabilistic drum machine that works with a 1/6 likely likely! Let's hear how it sounds: live_loop :random_beat make a sample :d rum_snare_hard if dice == 1 sleep 0.125 end Let's go quickly over each line to make sure everything is very clear. First, we create a live_loop a song called :random_beat that will continuously repeat the two lines between to and the end. The first of these lines is a call to a pattern that will play a prerecorded sound (:d rum_snare_hard sound in this case). However, this line has a special condition if it ends. This means that the line will only be executed if the statement is on the right if it is true. The statement in this case is dice == 1. This calls our function a cube that, as we have seen, returns a value between 1 and 6. We then use the equality operator == to verify that this value is 1. If it's 1, then the statement decides to be true and our drum sounds, if it's not 1 then the statement gets rid of the fake and the trap is skipped. The second line simply waits 0.125 seconds before rolling the dice again. Change the probability Of One You who have played role play games will be familiar with a lot of strangely shaped dice with different ranges. For example, there are tetrahedron-shaped cubes that have 4 sides and even 20 side cubes in the form of icosahedron. The number of sides on the dice changes the chance or likelihood of rolling the number 1. The fewer sides, the more likely you are to roll 1, and the more the side is less likely. For example, with 4 side dice, there is one of 4 chances of rolling 1 and with 20 side dice there is one in 20 chances. Fortunately, Sonic Pi has a one_in fn on hand to describe just that. Let's live_loop :d ifferent_vjerojatnosti: You can :d rum_snare_hard a pattern one_in(6) sleep 0.125 end Start a live loop above and you will hear a familiar random rhythm. However, do not stop the code from starting. Instead, change 6 to another value such as 2 or 20, and press the Run button again. Notice that lower numbers mean that the drum sounds more often, and higher numbers mean that the trap moves fewer times. You make music with probabilities! Combining probabilities Things get really exciting when you combine multiple patterns that run with different probabilities. For example: live_loop :multi_beat do a pattern :elec_hi_snare if one_in(6) sample :d rum_cymbal_closed if one_in(2) sample :d rum_cymbal_pedal if one_in(3) sample :bd_haus if one_in(4) sleep 0.125 end Again, run the code above, then begin to change the probabilities to change the rhythm. Also, try changing patterns to create a whole new feel. For example, try :d rum_cymbal_closed at :bass_hit_c for extra bass! Repeatable rhythms Next, we can use our old friend use_random_seed to reset a random stream after 8 iterations to create a regular beat. Type the following code to hear a much more regular and repetitive rhythm. After hearing the rhythm, try to change the seed value from 1000 to another number. Notice how different numbers generate different beats. live_loop :multi_beat not use_random_seed 1000 8.times do a sample :elec_hi_snare if one_in(6) sample :d rum_cymbal_zatvoren if one_in(2) sample :d rum_6) cymbal_pedal if one_in(3) sample :bd_haus if one_in(4) sleep 0.125 end One thing I tend to do with this type of structure is remember which seed sounds good and make a note of them. In this way, they can easily recreate their rhythms in future training sessions or performances. By putting it all together finally, we can insert some random bass to give it some nice melodic content. Notice that we can also use the newly discovered method of probabilistic sequencing on synths as well as patterns. Don't leave it at that - tweak the numbers and make your own mark with the power of probability! live_loop :multi_beat use_random_seed 2000 8.times to c = rrand(70, 130) n = (scale :e1, synth :tb303, be noted: n, edition: edition: cutoff: c if rand < 0.9 sample :elec_hi_snare if one_in(6) sample :d rum_cymbal_closed if one_in(2) sample :d rum_cymbal_pedal if one_in(3) sample :bd_haus, amp: 1.5 if one_in(4) sleep 0.125 end of A.14 - Ampplitude Modulation This month we will take a deep dive into one of Sonic Pi's most powerful and flexible audio FX - :cutter. By the end of this article, you will learn how to manipulate the total volume of parts of our live-coded sound in powerful new ways. This will allow you to create new rhythmic and timbral structures and expand sound capabilities. Slice that Amp So, what is :cutter FX actually do? One way to think about it is that it's like someone playing with volume control on a TV or home hi-fi. Let's look, but first, let's listen to the deep growling of the following code powered by a :p conscientious synth: synth :p rophet, notes: :e1, release: 8, cutoff: 70 synth :p rophet, notes: :e1 +4, release: 8, cutoff: 80 Now, let's tube it through :slicer FX: with_fx:slicer to synth :p rophet, notes: :e1, release: 8, cutoff: 70 synth :p rophet, notes: :e1 +4, release: 8, cutoff: 80 end Listen to how the cutter behaves as if it were muting and unmuting sound with regular beat. Also, notice how the :cutter affects all sound generated between blocks up to/end. You can control the speed at which it turns the sound on and off with the phase: decide which one is short for the duration of the phase. Its default value is 0.25, which means 4 times per second at a default BPM of 60. Let's make it faster: with_fx :slicer, stage: 0.125 to synth :p rophet, notes: :e1, release: 8, cutoff: 70 synth :p rophet, notes: :e1 +4, release: 8, cutoff: 80 end Now, play with different stages: duration yourself. Try longer and shorter values. See what happens when you select a really short value. Also, try different synths such as :beep or :d and different notes. See the following diagram to see how different phases: Values change the number of amplitude changes per beat. The duration of the phase is the length of time for one on/off cycle. Therefore, lower values will make FX turn on and off much faster than higher values. Good values to start playing are 0.125, 0.25, 0.5 and 1. Control Waves By Default, :slicer FX uses a square wave to manipulate amplitude through time. That's why we hear amplitudes on for a period and then immediately off for a period and then back on again. It turns out that the square wave is just one of 4 different control waves that support :cutter. The rest were saws, triangles and co's. See the diagram below to see what they look like. We can also hear what they sound like. For example, the following code uses (co)sine as a control wave. Hear that the sound does not turn on and does not turn on abruptly, but fades smoothly and exits: with_fx :cutter, stage: 0.5, 3 to synth :d saw, notes: :e3, release: 8, cutoff: 120 synth :d saw, notes: :e2, release: 8, cutoff: 100 end Have a game with different wave shapes by changing the wave: opt for 0 for saw, 1 for square, 2 for triangle and 3 for sons. See how different waves with different stages sound: decide too much. Each of these waves can be twisted invert_wave: decide which one turns it on the y axis. For example, at one stage the saw wave usually starts high and slowly descends before jumping back to the top. With invert_wave: 1 will start low and slowly go up before jumping down again. In addition, the control wave can be started at different points with phase_offset: decide what the value between 0 and 1 should be. Toying with the phase:, wave:, invert_wave: and phase_offset decide you can dramatically change the way amplitude changes over time. Set level By default, :slicer switches between value amplitude 1 (completely loud) and 0 (silent). This can change with the amp_min: and amp_max: decides. You can use this with the sinus wave setting to create a simple tremolo effect: with_fx :slicer, amp_min: 0.25, amp_max: 0.75, wave: 3, stage: 0.25 to synth :saw, release: 8 end This is like grabbing a volume button on a hi-fi and moving it up and down just a little so that the sound of 'wobbles' and comes out. Probabilities One of the powerful features :slicer is its ability to use the probability of choosing whether or not to turn the cutter on or off. Before :slicer FX starts a new phase, it rolles the dice and uses the selected control wave or keeps the amplitudes off based on the results. Let's with_fx: with_fx:slicer, phase: 0.125, probability: 0.6 to synth :tb303, notes: :e1, cutoff_attack: 8, release: 8 synth :tb303, notes: :e2, cutoff_attack: 4, release: 8 synth :tb303, note: :e3, cutoff_attack: 2, release: 8 end Hear how we now have an interesting rhythm of impulses. Try to change the probability: Opt for another value between 0 and 1. Values closer to 0 will have more space between each sound due to the likelihood that the sound will be much lower. Another thing to note is that the probability system in FX is just like the randomization system available through fns such as rand and shuffle. They're both completely deterministic. This means that every time you hit Run you will hear exactly the same pulse rhythm for a certain probability. If you want to change things around you, you can use seeds: decide to choose a second starting seed. It works exactly the same as use_random_seed but only affects that particular FX. Finally, you can change the position of the 'resting' control wave when the probability test fails from 0 to any other position with prob_pos: decide: with_fx :slicer, stage: 0.125, probability: 0.6, prob_pos: 1 to synth :tb303, note: :e1, cutoff_attack: 8, release: 8 :tb303, note: :e2, cutoff_attack: 4, issue: 8 synth :tb303, notes: :e3, cutoff_attack: 2, release: 8 end Cutting beats One really fun thing to do is use :cutter to chop drum beat in and out: with_fx :cutter, phase: 0.125 to sample :loop_mika end It allows us to take any pattern and create new rhythmic options which is a lot of fun. However, one thing to look out for is to ensure that the sample pace corresponds to the current BPM in Sonic Pi otherwise the cutting will sound completely off. For example, try replacing :loop_mika with loop_amen sample to hear how bad it can sound when tempos don't align. Change of pace As we have already seen, changing the default BPM from use_bpm will make all sleep times and durations of synth envelopes grow or decrease to fit the rhythm. :slicer FX honors that, as a phase: opt is actually measured in beats, not seconds. Therefore, we can solve the problem with loop_amen above by changing bpm to fit the pattern: use_sample_bpm :loop_amen with_fx :slicer, phase: 0.125 to sample :loop_amen end Bringing it all together Let's apply all these ideas into the final example that just uses :slicer FX to create an interesting combination. Go ahead, start changing it and make it into your piece! live_loop :d ark_mist to co = (line 70, 130, steps: 8).mark with_fx :cutter, probability: 0.7, prob_pos: 1 to synth :p rophet, notes: :e1, release: 8, cutoff: co end with_fx :slicer, stage: [0.125, 0.25].choose to sample :guit_em9, installments: 0.5 end sleep 8 end live_loop :crashing_waves to with_fx :slicer, wave: 0, stage: 0.25 to sample :loop_mika, rate: 0.5 end of sleep 16 end A.15 - Five live coding techniques in this month's Sonic Pi tutorial we'll look at how you can start treating Sonic Pi as a real instrument. Therefore, we need to start thinking about the code in a completely different way. Live coders think of codec in a similar way to how violinists think of their bow. In fact, just as a violinist can apply a variety of bowing techniques to create different sounds (long slow motion versus short quick hits) we will explore the five basic live coding techniques that Sonic Pi enables. By the end of this article, you will be able to start practicing for your own live-coded performances. 1. Memorise shortcuts The first tip for life coding with Sonic Pi is to start using shortcuts. For example, instead of wasting valuable time reaching for the mouse, moving it to the Run button and clicking, you can simply press alt and r at the same time which is much faster and keep your fingers at the keyboard ready for the next edit. You can find out shortcuts for the main keys at the top by hovering the mouse over them. See section 10.2 of the built-in textbook for a complete list of shortcuts. When performing, one fun thing to do is add a little flair to hand gestures when Shortcuts. For example, it's often good to interact with the audience when you're going to make a change - so brighten up your movement when hitting the alt-r just like a guitar player would when hitting a great chord of power. 2. Manually layer your sounds Now you can run the code immediately with the keyboard, you can immediately apply this skill for our other technique which is to layer your sounds manually. Instead of 'composing' using lots of play calls and patterns separated by sleep calls, we'll have one game call that we'll manually launch using alt-r. Try. Type the following code in the fresh clipboard: synth :tb303, note: :e2 - 0, release: 12, cutoff: 90 Now, hit Run and while the sound is playing, modify the code to drop four notes changing it to the following: synth :tb303, note: :e2 - 4, release: 12, cutoff: 90 Now, hit Run again, hear both sounds playing at the same time. This is because the Sonic Pi's Run button does not wait for any previous code to end, but runs the code at the same time. This means that you can easily manually line up a lot of sounds with smaller or large changes between each trigger. For example, try to change the note: and break: it is decided, and then reactivates. You can also try this technique with long abstract patterns. For example: sample :ambi_lunar_land, rate: 1 Try to start the pattern and then progressively halve the rate: decide between hitting Run 1 to 0.5 to 0.25 to 0.125, then even try some negative values such as -0.5. Place the sounds and see where you can take it. Finally, try adding a little FX. When performing, working with simple lines of code in this way means that the audience new to Sonic Pi has a good opportunity to follow what you are doing and connect code that they can read with the sounds they hear. 3. Master Live Loops When working with more rhythmic music, it can often be difficult to manually start everything and keep a good time. Instead, it is often better to use live_loop. This provides repetition for your code while giving you the ability to edit the code for the next round of the loop. They will also work at the same time as other live_loops which means you can layer them with each other and manually encode the triggers. See section 9.2 of the built-in guide for more information on working with live loops. When running, remember to use live_loop: decide to allow you to recover from accidental errors in the running time that stop the live loop from starting due to an error. If you already have sync: Decide to point to another valid live_loop, you can quickly fix the error and restart the code to restart things without missing a beat. 4. Use Master Mixer One of Sonic Pi's best kept secrets is that it has a master mixer through which all the sound flows. This mixer also has pass filter and built-in high pass filter, so you can easily perform global sound modifications. The functionality of the master mixer can be accessed via fn set_mixer_control!. For example, while some code is working and making a sound, enter this on the backup buffer and hit Run: set_mixer_control! lpf: 50 Once you run this code, all existing and new sounds will have a low pass filter applied to them and therefore will sound more muffled. Have the guess that this means that the new mixer values stick until they change again. However, if you wish, you can always reset the mixer back to its default state with reset_mixer!. Some of the currently supported opts are: pre_amp:, lpf: hpf:, and amp:. For a complete list, see the embedded documents set_mixer_control!. Use *190-_slide mixer and choose to move one or more values over time. For example, to slowly move the filter to add mixers low from the current value to 30, use the following: set_mixer_control! lpf_slide: 16, lpf: 30 Then you can quickly slip back to high value with: set_mixer_control! lpf_slide: 1, lpf: 130 When running, it is often useful to keep the clipboard free to work with a mixer like this. 5. Practice The most important technique for live coding is practice. The most common attribute in professional musicians of all kinds is that they practice playing with their instruments - often many hours a day. The practice is just as important for a live coder as it is for a guitar player. Practice lets your fingers remember certain patterns and common changes so you can type and work with them more accurately. Practice also gives you the opportunity to explore new sounds and code constructs. When you perform, you will find more practice that you do, it will be easier for you to relax in the gig. Practice will also give you a rich experience from which to draw. This can help you understand what types of modifications will be interesting and work well with current sounds. Bringing this month together, instead of setting a definitive example that combines everything under discussion, let's part by setting a challenge. See if you can spend a week practicing one of these ideas every day. For example, one day practice manual triggers, the other do some basic live_loop and the next day play with a master mixer. Then repeat. Don't worry if things feel slow and clunky at first - just keep working out and before you know it you'll be coding live for the right audience. A.16 - 8 Tips for live coding practice Last month we looked at five important techniques for mastering live coding - in other words, we explored how we could use Sonic Pi to access the code in the same way we would approach a musical instrument. One of the important concepts we discussed was practice. This month we will delve deeper into understanding why practice is also important so that you can get started. Exercise regularly The most important advice is to make sure you exercise regularly. As a rule, I usually exercise 1-2 hours a day, but 20 minutes is just fine when you start. A little bit, but often it's what you're aiming for - so if you can only manage 10 minutes, it's a great start. Practice advice #1 - start developing an exercise routine. Find good weather in the day that works for you and try to exercise at that time as many days of the week as you can. Soon you will be looking forward to your regular session. Learn to touch the species If you watch a professional musician perform on stage you'll probably notice a few things. First, when they play, they don't stare at their instrument. Their fingers, hands and bodies know which keys to press, plucking wires or drums to hit without having to think about it too much. This is known as muscle memory and although it may sound like something only professionals can do - it's the same as when you first learned to walk or ride a bike - exercising through repetition. Live coders use muscle memory to free their minds from having to think about where to move their fingers so they can focus on music. This is called touch typing - typing without the need to look at the keyboard. Practice the advice #2 - learn how to touch the species. There are many apps, websites and even games that can help you achieve this. Find one you like and stick to it until you can code without looking down. The code while standing The body of the musician is conditioned by the playing of their instrument. For example, the tamash must be able to blow hard, the guitarist must be able to firmly catch the fretboard and the drummer must be able to constantly beat the drums over a long period of time. So, what's physical about live coding? Just like DJs, live coders usually perform while standing, and some even dance while codifying! If you practice coding live while sitting at a table and then have to get up and stand at a concert, it's likely that the difference will be very difficult and frustrating for you. Practice advice #3 - stop while exercising. The easiest way to do this is to use a standing height table. However, if you don't have one at home like me, there are a few low-fi options. The approach I use is to use an ironing board that works pretty well. The second is stacking some boxes or large books on a normal table and putting a keyboard on top of it. Also, make sure you stretch before you start exercising and try to dance a little during the session. Remember, no one's looking at you, so have fun and you'll feel a lot more natural on stage. Practice setting Most instruments require some assembly and tweaks before they can be played. Unless you're a rock star with a bus full of roadies, you'll need to to your own instrument before the concert. This is often a stressful time and it is easy for problems to arise. One way to help with this is to include the setup process in your practice sessions. Practice advice #4 - treat setup as an important part of your practice. For example, have a box or bag where you can hold a Raspberry Pi and keyboard, etc. Before each training session, remove all parts, connect everything, and work through the startup process until you start Sonic Pi and you can make sounds. Once you've finished exercising, take the time to pack everything carefully afterwards. It may take some time at first, but soon you will be able to set up and pack everything incredibly quickly without thinking about it. Experiment musically Once you've set up and you're ready to start making music, you may struggle to know where to start. One of the problems many people face is that they may have a good idea of the types of sounds they want to make, but they are frustrated that they can't produce them. Some people don't even know what sounds they want to make! The first thing to do is not worry - it is very common and happens to any musician - even if they have been practicing for a long time. It is much more important to make sounds that you do not like than not to make sounds at all. Practice advice #5 - spend time making sounds and music you don't like. Try to find time to explore new sounds and ideas. Don't worry it might sound scary if it's not the style you're looking for. When you experiment like this, you increase your chances of getting to the sound or combination of sounds you love! Even if 99% of the sounds you make are bad, that 1% can be a riff or a prelude to your new song. Forget things you don't like and remember the parts you do. It's even easier when you're making music with code - just hit save! Listen to Code Many musicians can watch a music game and hear music in their head without having to play it. This is a very useful skill and worth incorporating into your live coding sessions. The imisrant point is to be able to have some understanding of what the code is going to sound like. You do not need to be able to hear it exactly in your head, but instead it is useful to know if the code will be fast, slow, loud, rhythmic, melodic, random, etc. The ultimate goal then is to be able to reverse this process - to be able to hear the music in your head and know which code to write to make it. It may take you a long time to master it, but once you do, you'll be able to improvise on stage and fluently express your ideas. Practice tip #6 - write some code in Sonic Pi, but don't press the Run button. Instead, try to imagine what sound it will make. Then, hit Run, listen, and think about You were right and you weren't. Keep repeating this until it becomes a natural part of your coding process. When I work out, I usually have a good idea of what the code is going to sound like. However, I am still occasionally surprised, and then I will stop and spend some time thinking about why I made a mistake. Every time that happens, I learn new tricks that allow me to express myself in new ways. A common problem when exercising is becoming distracted by other things. Exercise is difficult and requires real discipline regardless of the type of music you make - from jazz to classics to EDM. If you're struggling to get started or progressing, it's often too easy to jump on social media or look up something online, etc. If you've set yourself a 20-minute practice goal, it's important to try to spend all that time to be as productive as possible. Practice advice #7 - before you start exercising remove as many distractions as possible. For example, disconnect from the internet, put your phone in another room and try to exercise in a quiet place where you are unlikely to be upset. Try to focus on coding music and you can get back to your distractions when you're done. Keep an exercise diary When you exercise, you will often find that your mind is full of new exciting ideas - new music directions, new sounds to try, new writing functions, etc. These ideas are often so interesting that you could stop what you're doing and start working on an idea. This is another form of distraction! Practice advice #8 - keep a practice log with your keyboard. When you get an exciting new idea, temporarily pause your workout, quickly record the idea, then forget about it and keep practicing. Then you can spend some quality time thinking and working on your ideas after you're done exercising. Bonding Try to establish a practice routine that involves as many of these ideas as possible. Try to keep the sessions as fun as possible, but be aware that some training will be difficult and feel a bit like work. However, it will all pay off once you've created your first piece or got your first performance. Remember, practice is the key to success! A.17 - Stretching patterns When people discover Sonic Pi, one of the first things they learn is how easy it is to play prerecorded sounds using the pattern function. For example, you can play an industrial drum loop, hear the sound of a chorus, or even listen to a vinyl scratch all through a single line of code. However, many people do not realize that you can actually change the speed at which the pattern is re-played for some powerful effects and a whole new level of control over your recorded sounds. So set fire to a copy of Sonic Pi and let's start stretching some samples! Slowing down patterns to alter the rate of reproduction we must use the rate: decide: pattern :guit_em9, rate: 1 If you specify the rate: from 1 then the sample is reproduced at normal speed. If we want to play it at half speed, we simply use speed: from 0.5: sample :guit_em9, speed: 0.5 Notice that it has two effects on sound. First, the pattern sounds lower in height, and secondly, it takes twice as much time to play back (see sidebar to explain why this is the case). We can even choose lower and lower rates that move towards 0, so the speed: from 0.25 is a quarter of the speed, 0.1 is a tenth of the speed, etc. Try to play with some low rates and see if you can turn the sound into a low turd. Speeding Samples Up In addition to making the sound longer and lower using low speed, we can use higher speeds to make the sound shorter and higher. Let's play with the drum loops this time. First, listen to what it sounds like at the default rate of 1: sample :loop_amen, rate: 1 Now, let's accelerate a little: pattern :loop_amen, rate: 1.5 Ha! We just moved musical genres from old-skool techno to the jungle. Notice how the height of each drum strike is higher, as well as how the whole rhythm accelerates. Now, try even higher rates and see how high and short you can make drum loops. For example, if you use a rate of 100, the drum loop turns into a click! Reverse Gear Now, I'm sure many of you are thinking the same thing right now... what if you use a negative number for a rate?. That's a great question. Let's think about this. If our speed: decide indicates the speed at which the sample is playing, 1 is the normal speed, 2 is double speed, 0.5 is half speed, -1 must mean backwards! Let's try a trap. First, play it at normal speed: pattern :elec_filt_snare, rate: 1 Now, play it backwards: pattern :elec_filt_snare, rate: -1 Of course, you can play it backwards twice as fast with speeds of -2 or back at half speed with a speed of -0.5. Now play with different negative prices and have fun. It's especially fun with a pattern of :misc_burp! One of the effects of speed modification on samples is that faster rates result in a pattern sound higher in height, and slower speeds result in the sample sounding lower in height. Another place you may have heard this effect in everyday life is when cycling or riding past a sound pedestrian crossing - as you head for the sound source the height is higher than when moving away from sound - the so-called Doppler effect. Why is this? Let's consider a simple beep that is represented by a sinus wave. If we use an oskilloscope to draw a beep, we will see something like Image A. If we draw a beep for an octave more, we will see that image B and octave lower will look like Figure C. Notice that the waves of higher notes are more compact and the waves of lower notes are more widespread. Pattern is nothing more than a lot of numbers (x, y, coordinates) that, when plotted on a chart, will redraw the original curves. Look at Figure D where each circle represents the coordinate. To convert the coordinates back to sound, the computer works through each x value and sends the corresponding y value to the speakers. The trick is that the computer's work rate through x numbers doesn't have to be the same as the rate at which they were recorded. In other words, the space (representing a certain time) between each circle can be stretched or compressed. Thus, if the computer walks through x values faster than the original speed, it will have the effect of crushing the circles closer together resulting in a larger beep. It will also make the beep shorter as we will work faster through all circuits. It's shown on figure E. Finally, another thing to know is that a mathematician named Fourier proved that any sound is actually full and full of sinus waves all along. Therefore, when we compress and stretch any recorded sound, we actually stretch and compress many sinus waves at the same time precisely in this way. Pitch Bending As we have seen, using a faster speed will make the sound higher in height and slower speed will make the sound lower in height. A very simple and useful trick is to know that doubling the speed actually results in the pitch being octave higher and vice versa halving the speed results at octave height lower. This means that for melodic patterns, playing alongside yourself at double/half feet actually sounds pretty nice: sample :bass_trance_c, rate: 1 pattern :bass_trance_c, rate: 2 sample :bass_trance_c, rate: 0.5 However, what if we just want to change the rate so that the plot goes up one halftone (one note up on the piano)? Sonic Pi makes this a very simple way of rpitcha: decide: sample :bass_trance_c pattern :bass_trance_c, rpitch: 3 sample :bass_trance_c, rpitch: 7 If you look at the diary on the right, you will notice that rpitch: of the 3 actually corresponds to the rate of 1.1892 and rpitch: of 7 corresponds to the rate of 1.4983. Finally, we can even combine the rate: and rpitch: opts: sample :ambi_choir, foot: 0.25, rpitch: 3 sleep 3 sample :ambi_choir, foot: 0.25, rpitch: 5 sleep 2 pattern :ambi_choir, foot: 0.25, rpitch: 6 sleep 1 pattern :ambi_choir, foot: 0.25, rpitch: 1 Bringing it all together Let's look at a simple piece that combines these ideas. Copy it to the empty Sonic Pi clipboard, hit play, listen to it for a while, then use it as a starting point for your own piece. See how much fun it is to manipulate the rate of reproduction of patterns. As an extra exercise, try recording your own sounds and playing at speed to see what crazy sounds you can make. live_loop :beats up pattern :guit_em9, rate: [0.25, 0.5, -1].choose, amp: 2 :loop_garzul, rate: [0.5, 1].choose sleep 8 end live_loop :melody to vinegar = [-1, 1, 2].choose * 12 with_fx :reverb, amp: 2 to 16.times to n = (scale 0, :minor_pentatonic).choose sample :bass_voxy_hit_c, rpitch: n+4 + oct sleep 0.125 end A.18 - Additive Synthesis This is the first in a series of articles on how to use Sonic Pi for sound design. We will quickly visit a number of different techniques available to you to create your own unique sound. The first technique we're going to look at is called additive synthesis. This may sound complicated - but if we expand each word a little the meaning pops up. First, the additive means a combination of things and the second synthesis means the formation of sound. Additive synthesis therefore means nothing more complicated than combining existing sounds to create new ones. This synthesis technique dates back a very long time - for example, pipe organs in the Middle Ages had many few different sound tubes that you could enable or disable by stopping. Pulling out a stand for a particular tube 'added it to the mix' making the sound richer and more complex. Now, let's see how we can get all the stops out of Sonic Pi. Simple combinations Let's start with the most basic sound there is - a modest purely-toned sinus wave: synth :sine, notes: :d 3 Now, let's look at how this sounds in combination with a square wave: synth :sine, notes: :d 3 synth :square, note: :d 3 Notice how the two sounds combine to form a new, richer sound. Of course, we don't have to stop there, we can add as many sounds as we need. However, we must be careful how many sounds we add together. Just as when we mix colors, adding too many colors will result in a messy brown, similarly - adding too many sounds together will result in a muddy sound. Mixing Add something to make it sound a little scarier. We could use a triangular wave on an octave higher (for that high bright sound), and yet we only play it at amp 0.4 so it adds something extra to the sound instead of downloading it: synth :sin, notes: :d 3 synth :square, notes: :d 3 synth :three, notes: :d 4, amp: 0.4 Now, try to create your own sounds by combining 2 or more synths on different octaves and amplitudes. Also, for example, you can toy with each synth's selections to modify each original sound before it's moved for even more combinations of sounds. Detuning so far, when we combine our different synths we have used either the same pitch or switched octave. How would it sound if we didn't stick to octaves, but to choose a slightly higher or lower note? Let's try: detune = 0.7 synth :square, note: :e3 synth :square, note: :e3 + detune If we deuteron our square waves by 0.7 notes we hear something that may not sound aligned or correct - a 'bad' note. However, as we approach 0 this will sound less and less as the heights of the two waves approach and Change detune: Choose a value from 0.7 to 0.5 and listen to the new sound. Try 0.2, 0.1, 0.05, 0. Each time you change a value, listen and see if you can hear the sound change. Notice that low detune values such as 0.1 produce a really nice 'thick' sound, with both slightly different pitches interacting with each other in interesting, often surprising ways. Some of the built-in synths already include a detune option that does just that in one synth. Try to play with detune: opt for :d, :d pulse and :d three. Amplitude formatting Another way we can fine-tun our sound is by using different envelopes and options for each synth trigger. For example, this will allow you to make some aspects of the sound of percussion and other aspects residual for some time. detune = 0.1 synth :square, notes: :e1, edition: 2 synth :square, notes: :e1 + detune, amp: 2, issue: 2 synth :gnoise, Release: 2, amp: 1, cutoff: 60 synth :gnoise, release: 0.5, amp: 1, cutoff: 100 synth :noise, release: 0.2, amp: 1, cutoff: 90 In the example above I mixed in a nuisous percussionist to sound along with some persistent background turd. This was achieved first by using two medium-valued noise synthesizers (90 and 100) using short release times along with noise with longer release times, but with a low value rate (which makes the noise less crunchy and rumbly.) Let's combine all these techniques to see if we can use additive synthesis to recreate the basic ringtone. I broke this example into four parts. First we have the 'hit' part that is the initial beginning of the ringtone - so use a short envelope (e.g. release: from about 0.1). Then we have a long ringing section where I use the pure sound of a sinus wave. Note that I often increase my note by approximately 12 and 24, which is the number of notes in one and two octaves. I also inserted a few low sinuses to give the sound a little bass and depth. Finally, I used definition to wrap code in a function that I can then use to play a melody. Try playing your own tune and also mess around with the content of the :bell function until you create your own crazy game sound! define :bell |n| # Triangle waves for 'hit' synth:three, notes: n - 12, release: 0.1 synth :three, notes: n+0.1, release: 0.1 synth :three, notes: n - 0.1, issue: 0.1 synth :three, notes: n, release: 0.2 # Sine waves for 'ringing' synth :sine, notes: n+24, edition: 2 synth :sine, notes: n + 24.1, Release: 2 synth :sine, notes: n + 24.2, issue: 0.5 synth :sine, notes: n + 11.8, release: 2 synth :sine, notes n, release: 2 # Low sinus waves for bass synth :sine, notes: n - 11.8, release: 2 synth :sine, notes: n - 12, release: 2 end Play the melody with our new bell! Bell :e3 sleep 1 bell :c2 sleep 1 bell :d 3 sleep 1 bell :g2 A.19 - Subtractive Synthesis This is the second in a series of articles on how to use Sonic Pi for sound design. Last month we looked at an additive synthesis that we found was a simple act of playing multiple sounds at the same time to make a new combined sound. For example, we could combine different sonic synthesizers or even the same synth on different plots to build a new complex sound from simple ingredients. This month we will look at a new technique commonly referred to as subtractive synthesis, which is simply the act of taking the existing complex sound and removing its parts to create something new. It is a technique usually associated with the sound of analog synthesizers in the 1960s and 1970s, but also with the recent renaissance of modular analog synths through popular standards such as Eurorack. Despite sounding like a particularly complicated and advanced technique, Sonic Pi makes it surprisingly simple and simple - so let's dive right in. Complex original signal For sound to work well with subtractive synthesis, it usually has to be quite rich and interesting. This does not mean that we need something extremely complex - in fact, only the standard :square or :saw wave will do: synth :saw, notes: :e2, release: 4 Notice that this sound is already quite interesting and contains many different frequencies above :e2 (the second E on the piano) that add color creation. If this didn't make much sense to you, try comparing it to :beep: synth :beep, notes: :e2, release: 4 How is :beep synth just a sinus wave, you'll hear a much cleaner tone and only at :e2 and none of the high crunchy/buzzing sounds you've heard in :p or. It's this buzz and variation from a pure sinus wave that we can play with when using subtractive synthesis. Filters After we have our raw source signal, the next step is to go through some kind of filter that will modify the sound by removing or reducing its parts. One of the most common filters used for subtractive synthesis is something called a low pass filter. This will allow all low parts of the sound through, but will reduce or remove higher parts. Sonic Pi has a powerful but easy-to-use FX system that includes a low pass filter, called :lpf. Let's play with it: with_fx:lpf, cutoff: 100 to synth :saw, notes: :e2, release: 4 end If you listen carefully you will hear how some of this buzzing and crunchiness has been removed. In fact, all frequencies in the sound above the note 100 are reduced or removed, and only those below are still present in the sound. Try changing this break: point to lower notes, say 70, then 50 and compare sounds. Of course, :lpf is not the only filter that you can use to manipulate the original signal. Another important FX is high filter called :HPF in Sonic Pi. This does the opposite of :lpf in that it allows high volumes of sound and cut off low parts. with_fx :HPF, cutoff: 90 to synth:saw, notes: :e2, release: 4 end Notice how it sounds a lot buzzy and raspy now that all the low-frequency sounds have been removed. Play around with the cut-off value - notice how the lower values release more original bass parts of the original signal through and higher values sound all tinny and silent. [Low Pass Filter] - The low pass filter break box is such an important part of any subtractive synthesis tool that it's worth taking a deeper look at how it works. This diagram shows the same sound wave (:p sapija) with different filtering quantities. At the top, Section A displays an audio wave without filtering. Notice that the shape of the wave is very sharp and contains a lot of sharp edges. It is these hard, sharp angles that produce high crunchy/buzzing parts of the sound. Section B displays a low pass filter in action - notice how it is less heather and rounder than the wave shape above. This means that the sound will have less high frequencies giving it a softer rounded feel. Section C displays a low pass filter with a fairly low cut-off value - this means that even more high frequencies are removed from the signal resulting in an even softer, rounder waveform. Finally, notice how the size of the wave shape, which represents amplitudes, decreases as we move from A to C. Subtractive synthesis works by removing parts of the signal which means that the total amplitude decreases as the amount of filtering that takes place increases. Filter modulation So far we have made quite static sounds. In other words, the sound in no way changes completely during its duration. Often you may want some movement in sound to give timbre some life. One way to achieve this is through filter modulation - changing filter options over time. Luckily Sonic Pi gives you powerful tools to manipulate time-making FX. For example, you can set slide time on each modulable decide how long it should take for the current value to slide linearly to the target value: with_fx :lpf, cutoff: 50 to |fx| control fx, cutoff_slide: 3, cutoff: 130 synth :p rophet, notes: :e2, sustain: 3.5 end Let's look at the speed of what's going on here. First we start :lpf FX block as usual with the initial break: from the very low 20. However, the first line also ends with the odd |fx| in the end. This is an optional part of with_fx syntax that allows you to directly name and control the current FX synth. Line 2 does just that and controls FX to set cutoff_slide: opt for 4 and new target break: be 130. FX will now start sliding cutoff: opt value from 50 to 130 period of 3 beats. Finally, we also initiate the synthete of the original signal so that we can hear the effect of the modulated low pass filter. Bonding This is just a very basic taster of what is possible when using filters to modify and change the original sound. Try to play around with Sonic Pi's many built-in FX to see what crazy sounds you can design. If your sound seems too static, remember that you can start modulating motion-creating options. Let's finish designing a function that will play a new sound created by subtractive synthesis. See if you can figure out what's going on here - and for the advanced Sonic Pi readers out there - see if you can figure out why I wrapped everything inside the call on (please send answers to @samaaron on Twitter). define :subt_synth not |notes, sus| at to with_fx :lpf, cutoff: 40, amp: 2 to |fx| control fx, cutoff_slide: 6, cutoff: 100 synth :p rophet, notes: notes, sustain: sus end with_fx :HPF, cutoff_slide: 0.01 to |fx| synth :d saw, notes: notes + 12, sustain: sus (sus * 8).times do control fx, cutoff: rrand (70, 110) sleeping 0.125 end subt_synth :e1, 8 sleep 8 subt_synth :e1 - 4, 8 Creative coding in the classroom with Sonic Pi (This article was published in the issue of 9 Hello World Magazine) Code is one of the most creative media that people have created. Initially the opuscous symbols of brackets and lambda are not only deeply rooted in science and mathematics, they are as close as we have been able to get to casting the same kind of magical spells as Gandalf and Harry Potter. I believe this provides a powerful means of engagement in our learning spaces. Through the magic of the code we are able to evoke individually meaningful stories and learning experiences. We're surrounded by magical experiences. From a sleuth of a stage magician who makes a ball disappear in the air, to wonders when you see your favorite band perform on the big stage. It is these wow moments that inspire us to pick up a magic book and learn the French Drop or to start interfering with the chords of power on an old guitar. How could we create similarly deep and lasting senses of wonder that will motivate people to practice and learn the basics of programming? Musical engines and notation The history of music and computers has been intricately woven together since the founding of computer machines, i.e. engines as Charles Babbage's powerful analytical engine was called. Back in 1842, mathematician Ada Lovelace, who worked very closely with Babbage, saw the creative potential of these engines. While these first engines were originally designed to accurately solve difficult math problems, Ada dreamed of making music with them: .. the engine may assemble elaborated and scientific musical parts of any degree of complexity or scope. Ada Lovelace. Of course, today in 2019 a lot Our music, regardless of genre, is either composed, produced or mastered by a digital computer. Eda's dream came true. It's even possible to trace history even further. If you see coding as the art of writing sequences of special symbols that instruct the computer to do certain things, then musical composition is a very similar practice. In Western music, symbols are black dots placed on the length of lines that tell the musician which notes to play and when. Intriguingly, if we return the roots of Western musical notation to the Italian Benedictine monk Guido d'Arezzo, we find that the system of dots and lines used by modern orchestras is just one of a series of notation systems he has worked on. Some of the others were much closer to what we might now see as code. Since the late 1960s, magical meaningful experiences with computers and programming languages have been explored in education. Computer education pioneers Seymour Papert, Marvin Minsky and Cynthia Solomon explored simple Lisp-based languages that moved pencils over large pieces of paper. With just a few simple commands, it was possible to program your computer to draw any picture. They even experimented by expanding their logo language from drawing to music. Papert wrote about learning through experiencing a reconstruction of knowledge, not its transmission. Getting people to play with things directly was an important part of his group's work. Sonic Pi Performances Jylda and Sam Aaron perform at the Thinking Digital Conference in Sage Gateshead. Photo credit: TyneSight Photos. Sonic Pi has been used for performances in a wide range of venues such as school halls, nightclubs, outdoor stages at music festivals, college chapels and prestigious music venues. For example, the incredible Convo project that brought together 1,000 children at the Royal Albert Hall to perform an ambitious new composition by composer Charlotte Harding. The work is written for traditional instruments, choirs, percussion and Sonic Pi code. Pop artist Jylda also performed with Sonic Pi at Sage Gateshead for the Thinking Digital Conference, where she created a unique live-coded impromptu remix of her song Reeled. Sonic Pi was used as one of the instruments as part of conv at the Royal Albert Hall. Photo credit: Pete Jones. Live coding in the Sonic Pi classroom is a tool for creating music and performance based on code that builds on all these ideas. Unlike most computer education software, it is both easy enough to use for education and powerful enough for professionals. It was used to perform at international music festivals, used to compose in a range of styles from classics, EDM and heavy metal, and was even peer-reviewed by Rolling Stone magazine. It has a diverse community of over 1.5 million living coders with different backgrounds that teach and share everything ideas and thoughts through the medium of code. It's free to download for Mac, PC and Raspberry Pi and includes a friendly guide that assumes you don't know anything about code or music. Sonic Pi was initially conceived in response to the newly published computer curriculum in the UK in 2014. The goal was to find a motivating and fun way to teach the basics of programming. It turns out that there is a lot in common and that it is huge fun to explain sequencing as melody, iteration as rhythm, conditioning as musical diversity. I developed initial designs and the first iterations of the platform with Carrie Anne Philbin, which brought teacher perspective to the project. Since then, Sonic Pi has undergone iterative improvements thanks to the feedback it received from student observation and direct collaboration with classroom teachers. The philosophy of basic design was never to add a feature that could not be easily taught to a 10-year-old child. This meant that most ideas had to be heavily refined and reworked until they were simple enough. Making things simple while keeping them powerful remains the hardest part of the project. To provide magical motivation, Sonic Pi's design has never been limited to a pure focus on education. Ideally, there would be famous musicians and performers who use Sonic Pi as a standard instrument with guitars, drums, vocals, synths, violins, etc. These performers would then act as motivational role models that demonstrate the creative potential of the code. For this to be possible, enough focus and effort had to be put in to make it a powerful instrument, and to keep it simple enough for 10-year-olds to pick it up. In addition to teachers, I worked directly with various artists in classrooms, art galleries, studios and spaces in the early stages of Sonic Pi's development. This provided essential feedback that allowed Sonic Pi to grow and ultimately flourish as a tool for creative expression. There have been a few exciting and unexpected side effects of this dual focus on education and professional musicians. Many features are useful for both groups. For example, a lot of effort has been made to make error messages friendlier and more useful (rather than being a big complicated mess of jargon). This turns out to be very useful when you write a bug while performing in front of thousands of people. In addition, functionality such as playing studio quality audio patterns, adding audio effects, providing live audio access from microphones, all turned out to make the learning experience more fun, rewarding and ultimately meaningful. The Sonic Pi community continues to grow and share incredible code compositions, lesson plans, music algorithms and more. There is a lot going on on our friendly forum in_thread (in-thread.sonic-pi.net) which is home a very diverse group of people that includes educators, musicians, programmers, artists and creators. It is a real joy to see people learning to use code to express themselves in new ways and to do so in turn inspire others to do the same. Some fun options From a computer science perspective, Sonic Pi provides you with building blocks that will teach you the basics found in the UK curriculum, such as sequencing, iteration, conditional, functions, data structures, algorithms, etc. However, it also builds on a number of important and relevant concepts adopted in mainstream industry, such as consent, events, pattern matching, distributed computing and determinism - all while keeping things simple enough to explain to a 10-year-old child. The beginning is as simple as: play 70 Melody can be constructed with another command, sleep: play 70 sleep 1 play 72 sleep 0.5 play 75 In this example we play the note 70 (approximately the 70th note on the piano), wait 1 second, play note 72, wait half a second and then play the note 75. What's interesting here is that with just two commands we have access to almost all Western notes (what notes to play and when) and students can code any tune they've ever heard. This leads to a great variety in expressive outcomes, while focusing on the same computer concept: sequencing in this case. Taking ideas from the professional music world, we can also restore any recorded sound. Sonic Pi can play any audio file on your computer, but it also has a number of built-in sounds to make things easy to get started: sample :loop_amen This code will play a drum break that was a pillar stone of early hip-hop, Drum and Bass and Jungle. For example, a number of early hip-hop artists played this drum at half speed to give it a more relaxed feel: sample :loop_amen, rate: 0.5 In the 90s numerous music scenes burst from new technology that allowed artists to separate like this and re-form in a different order. For example: live_loop :jungle to sample :loop_amen, onset: pick sleep 0.125 end In this example we introduce a basic loop called :jungle that selects a random drum hit from our audio pattern, waits an eighth of a second, and then chooses another drum hit. This results in an endless series of random drums to dance to while you experience what a loop is. B - Essential Knowledge This section will cover some very useful - actually essential - knowledge to get the most out of your Sonic Pi experience. We will cover how to take advantage of the many keyboard shortcuts available for you, how to share your work and some performance tips with Sonic Pi. B.1 - Using the Sonic Pi shortcut is as much an instrument as an encoding environment. Shortcuts can therefore make playing Sonic Pi much more efficient and natural – especially live in front of an audience. Most Sonic Pi can be controlled through the keyboard. As you gain more knowledge of working and performing with Sonic Pi, you'll probably start using shortcuts more and more. I personally touch the guy (I recommend you also think about learning) and I find myself frustrated whenever I need to reach for my mouse because it slows me down. Therefore, I use all these shortcuts on a very regular basis! Therefore, if you learn shortcuts, you will learn to use the keyboard effectively and in no time you will be coding live as a professional. However, do not try to teach them at once, just try to remember the ones you use the most, and then proceed to add more to your practice. Consistency on platforms Imagine learning a clarinet. You'd expect all clarinets to have similar controls and fingers. If they hadn't, it would have been difficult to switch between different clarinets and stick to using just one device. Unfortunately, the three main operating systems (Linux, Mac OS X and Windows) come with their own standard default settings for actions such as cut and paste, etc. Sonic Pi will try to meet these standards. However, the priority is placed on consistency across all platforms within Sonic Pi instead of trying to adapt to the standards of a particular platform. This means that when you learn shortcuts while playing with Sonic Pi on Raspberry Pi, you can move to a Mac or PC and feel at home. Part of the concept of consistency is the naming of shortcuts. At Sonic Pi, we use the names Control and Meta to refer to the two main combined keys. On all platforms, control is the same. However, on Linux and Windows, Meta is actually the Alt key while the Command key is on Mac Meta. For consistency, we'll use the term Meta - just don't forget to map it to the appropriate key on your operating system. Abbreviations To help keep things simple and readable, we'll use the abbreviations C- to control plus another key and an M- for Meta plus another key. For example, if the shortcut requires you to hold both Meta and r we will write it as M-r. This means only at the same time as. Here are some of the shortcuts I find most useful. Stop and run Instead of always reaching for the mouse to run the code, you can simply press M-r. Similarly, to stop using the code, you can press M-s. Therefore, I recommend you spend time to learn them. These shortcuts also work extremely well when you've learned to touch a type because they use standard letters instead of requiring you to move your hand to a mouse or arrow keys on your keyboard. You can move to the beginning of the line with C's, end of line with C-e, to the line with C-p, down the line with C-n, forward figure with C-f, and back figure with C-b. You're a C-b. You're a C-b character. Even delete all characters from the cursor to the end of the line with C-k. Neat code To automatically align the code, simply press M-m. Help System To overlap the Help system, you can press M's. However, a much more useful shortcut to know is C's who will search for a word under the cursor and display documents if it finds something. Immediate help! For the full list, see section 10.2 of the Cheatsheet shortcut. B.2 - Cheatsheet Shortcut Follows a summary of the main shortcuts available within Sonic Pi. See Section 10.1 for motivation and background. Conventions We use the following conventions on this list (where Meta is one of alt on Windows/Linux or Cmd on Mac): C's funds hold the control key, and then press the key while holding them both at the same time, and then publishing. M-r means hold the Meta key, then press r key while holding them both at the same time, then releasing them. S-M-z means hold the Shift key, then the Meta key, then finally z key all at the same time, and then release. C-M-f means hold the control key, then press meta key, finally f key all at the same time and then release. Main Application Manipulation M-r - Run code M-s - Stop code M-i - Toggle Help System M-p - Toggle Preferences M-{ - Switch buffer to the left M-} - Switch buffer to the right M-+ - Increase the text size of the current M-- - Reduce the text size of the current Buffer Selection/Copy/Paste M-a - Select all M-A c - Copy Selection for paste tampon M-] - Copy the selection to the M-x paste stand - Cut the choice for the tampon paste C-] - Cut the choice for the paste buffer C-k - Cut to the end of the line M-v - Paste buffer pastes to editor C-y - Pastes from paste buffers to editors C-SPACE - Place the label. Navigation will now manipulate a prominent region. Use the C-g to get away. Text Manipulation M-m - Align all text Tab - Align current line or selection C-l - Centre editor M-/ - Comment/Uncomment current line or selection C-t - Transpose/swap characters M-u - Convert the following word (or selection) to the case above. M-l - Convert the following word (or selection) to the case below. C-a - Move to the beginning of the C-line - Scroll to the end of the C-P line - Move to the previous line C-n - Move to the next line C-f - Move forward one C-b sign - Scroll back one sign M-f - Scroll forward one word M-b - Scroll back one word C-M-n - Move line or selection down C-M-p - Move line or selection up S-M - Move up 10 lines S-M-d - Move down 10 lines M-< - Move to beginning of buffer M-> - Move to End buffer Delete C-h - Delete previous C-d character - Delete the following Advanced Editor Features character C-i - Display documents for word under M-z - Undo S-M-z - Redo C-g - Escape S-M-f - Switch full screen mode S-M-b - Key Visibility - Visibility of the S-M-m okljun log - Switch between light/dark S-M-s modes - Save the contents of the Clipboard to the S-M-o file - Load the contents of buffer buffers File B.3 - Sharing Sonic Pi is all about sharing and learning with each other. Once you've learned to code music, sharing compositions is as easy as sending emails containing your code. Please share your code with others so they can learn from your work and even use the parts in the new mash-up. If you're not sure what's the best way to share your work with others, I recommend putting your code on GitHub and music in SoundCloud. This will help you easily reach a large audience. Code -> GitHub GitHub is a code sharing and working site. It is used by professional developers as well as artists to share and collaborate with the code. The simplest way to share a new piece of code (or even an unfinished piece) is to create a Gist. Gist is an easy way to transfer code in a simple way that others can see, copy and share. Audio -> SoundCloud Another important way to share your work is to record audio and upload it to SoundCloud. After you upload your article, other users can comment and discuss your work. I also recommend putting a link to the Gist of your code in the track description. To record your work, press the Rec button in the tool and recording starts immediately. Hit Run to run the code if it's not already in progress. When you're done recording, press the Flashing Rec button again and you'll be prompted to enter a file name. The recording will be saved as a WAV file, which can be edited and converted to MP3 by any number of free programs (try Audacity for example). I hope to encourage you to share your work and really hope that we will all teach each other new tricks and moves with Sonic Pi. I'm really excited about what you're going to have to show me. B.4 - Performing one of the most exciting aspects of Sonic Pi is that it allows you to use code as a musical instrument. This means that the live writing code can now be viewed as a new way to perform music. We call this Live Coding. Display the screen when you live code I recommend showing the screen to your audience. It's usually like playing guitar, but you hide your fingers and strings. When I work out at home I use Raspberry Pi and a little mini projector on the living room wall. You can use your TV or give one of your school/work projectors a show. Try it, it's fun. Form a band Don't just play alone - form a live coding band! It's a lot of fun jamming with others. One person can do beats, another ambient background, etc. Use live_audio to combine code with traditional instruments such as a guitar or microphone. See what interesting combinations of sounds you can create with code. TOPLAP Live coding is not entirely new - a small number of people have been doing this for several years, usually using custom systems they have built for themselves. A great place to go and find out about other coder and live systems is TOPLAP. Algorave Another great resource for exploring the world of live coding is Algorave. Here you can find everything about a certain strand of live coding to create music in nightclubs. C - Minecraft Pi Sonic Pi now supports a simple API to interact with Minecraft Pi - a special edition minecraft that is installed by default on raspbian linux operating system Raspberry Pi. No need to import library Minecraft Pi integration is designed to be insanely easy to use. All you have to do is launch Minecraft Pi and create the world. Then you are free to mc_* fns just as you could use the game and synth. There is no need to import anything or install any libraries - everything is ready to go and work out of the box. The Minecraft Pi API automatic connection takes care of managing your connection to the Minecraft Pi app. That means you don't have to worry about anything. If you try to use a Minecraft Pi API when Minecraft Pi is not open, Sonic Pi will politely tell you. Similarly, if you close Minecraft Pi while still running a live_loop that uses an API, the live loop will stop and politely tell you it can't connect. To reconnect, restart Minecraft Pi and Sonic Pi will automatically detect and recreate the connection for you. Designed to be live encoded Minecraft Pi API is designed to work seamlessly within live_loops. This means that it is possible to synchronize modifications in your Minecraft Pi worlds with modifications in sonic pi sounds. Instant Minecraft music videos! Keep an eye out, however, minecraft Pi is alpha software and is known to be a bit buggy. If you encounter any problems simply restart Minecraft Pi and continue as before. Sonic Pi's automatic connection functionality will take care of things for you. Requires Raspberry Pi 2.0 It is highly recommended to use Raspberry Pi 2 if you want to run both Sonic Pi and Minecraft at the same time - especially if you want to use sonic pi's sound capabilities. API SUPPORT At this stage, Sonic Pi supports basic block and player manipulations detailed in Section 11.1. Support for event callbacks driven by player interactions in the world is planned for future release. 11.1 - Basic Minecraft Pi API Sonic Pi currently supports the following basic interactions with Minecraft Pi: Display chat messages Set user position Setting user position Set block type at specific coordinate Getting block type on a specific coordinate Let's look at each of them in return. Display chat messages Let's see how easy it is to control Minecraft Pi from Sonic Pi. First, make sure you open Minecraft Pi and Sonic Pi at the same time and also make sure you've entered the Minecraft world and can walk around. in the Sonic Pi Clipboard simply enter the following code: mc_message Hello from Sonic Pi When you press the Run button, you will see your message flashing on the Minecraft window. Congratulations, you wrote your first Minecraft code! That was easy, wasn't it? Setting the user's position now, let's try a little magic. Let's beam somewhere! Try the following: mc_teleport 50, 50, 50 When you hit Run - boom! You were immediately transported to a new place. Most likely it was somewhere in the sky and you fell either on land or in the water. What are the numbers: 50, 50, 50? These are the coordinates of the location you're trying to beam to. Let's take a brief moment to explore what the coordinates are and how they work because they're really, really important for Minecraft programming. Coordinates Imagine a pirate map with a large X indicating the location of a treasure. The exact location of the X can be described with two numbers - how far along the map from left to right and how far along the map from bottom to top. For example 10cm across and 8cm up. These two numbers 10 and 8 are coordinates. You can easily imagine describing the locations of other treasure stocks with other pairs of numbers. Maybe there's a big chest of gold on two over and nine up... Now, in Minecraft, two numbers isn't quite enough. We also need to know how tall we are. So we need three numbers: How far from right to left in the world - x How far back and forth in the world - z How high we are in the world - y One more thing - we usually describe these coordinates in this order x, y, z. Find your current coordinates Let's play with coordinates. It navigates to a nice spot on the Minecraft map, then switch to Sonic Pi. Now enter the following: puts the mc_location when you press the Run button you will see the coordinates of your current position displayed in the log window. Take in their note, then move forward in the world and try again. Notice how the coordinates have changed! Now, I recommend you spend some time repeating just that - move around a little bit in the world, look at the coordinates and repeat. Do this until you start to get a sense of how the coordinates change when you move. Once you understand how the coordinates work, programming with the MINECRAFT API will be a complete breeze. Let's build! Now that you know how to find your current position and teleport using coordinates, you have all the tools you need to start building things in Minecraft with code. Let's say you want to make a block with coordinates 40, 50, 60 to be glass. It's super easy: mc_set_block:glass, 40, 50, 60 Haha, it really was that easy. To see your handiwork just beam nearby and look: mc_teleport 35, 50, 60 Now turn around and you should see your glass block! Try changing it to diamond: :d iamond, 40, 50, 60 If you were looking in the right direction you might even have seen it change before your eyes! This is the beginning of something exciting... Looking at the blocks Let's look at one more thing before we move on to something a little more involved. Given the set of coordinates we can ask Minecraft what kind of a particular block is. Let's try the diamond block you just created: it mc_get_block 40, 50, 60 yes! It's :d iamond. Try to change it back to the glass and ask again - does it say now :glass? I'm sure not :-) Available types of blocks before you go on a Minecraft Pi coding rampage, perhaps this list of available types of blocks will be useful to you: :air :stone :grass :d irt :cobblestone :wood_plank :sopling :bedrock :water_flowing :water :water_stationary :lava_flowing :lava :lava_stationary :p sour :gravel :gold_ore :iron_ore :coal_ore :d rvo :leaves :glass :lapis :lapis_lazuli_block :p shood :bed :p aucine :grass_tall :flower_yellow :flower_cyan :mushroom_brown :mushroom_red :gold_block :gold :iron_block :iron :stone_slab_double :stone_slab :brick :brick_block :tnt :bookshelf :moss_stone :obsidian :torch :fire :stairs_wood :p rsa :d iamond_ore :d iamond_block :d iamond :crafting_table :p aded plot :furnace_inactive :furnace_active :d oor_wood :ladder :stairs_cobblestone :d oor_željezo :redstone_ore :snow :ice :snow_block :cactus :clay :sugar_cane :fence :glowstone_block :bedrock_invisible :stone_brick :glass_pane :d inja :fence_gate :glowing_obsidian :nether_reactor_core :nether_reactor_core