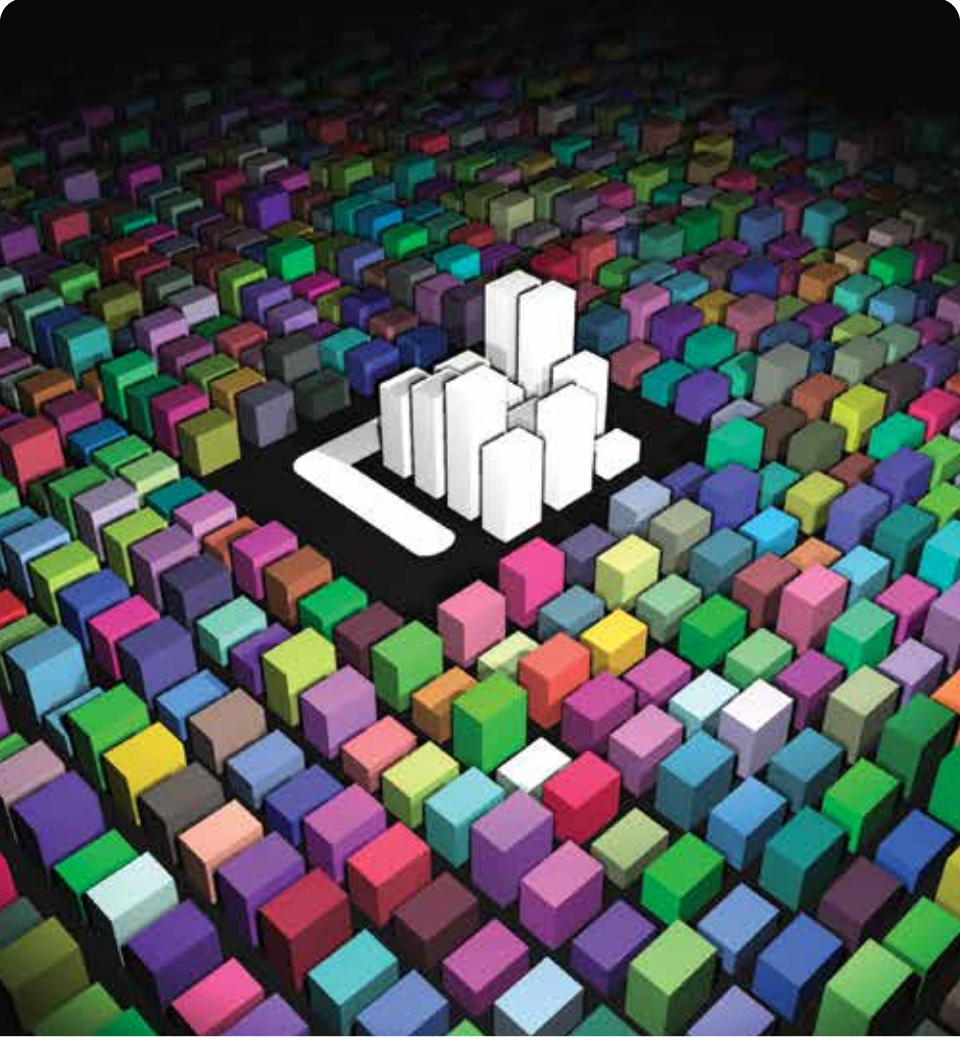


MANHATTAN

BYTES + PIECES

 nash.audio/manhattan

INSTRUCTIONS

BYTES + PIECES presents a series of fun challenges exploring music through code using **Manhattan** – an innovative digital platform for coding and composing.

Challenges involve completing pieces of music using code, looking at specific concepts in music or code.

Each task is presented on a single card describing:

- the **OBJECTIVE** and how the final piece works
- which music and coding concepts you'll **LEARN**
- key concepts illustrated as an **IMAGE** (e.g. screenshot)
- clues on **HOW** to go about completing the piece
- **TIPS** providing pointers and techniques to help

The **difficulty** level (1 to 3) is shown top right of the card, but *all* require you to **THINK** and **FIND** solutions, using information from not only the card but also the provided **REFERENCE** cards. Music and programming are both highly complex, so it is as important to know how to piece together information and work with new concepts, as it is to establish working knowledge.

Start with **OVERTURE** (Level 1), then move onto a Level 2 or 3 challenges as your skills develop!

For questions, support and feedback – or if you're interested in Manhattan, contact the project lead:

Dr Chris Nash (chris@nash.audio)

Don't have Manhattan? Download for Mac / Win:

http://nash.audio/manhattan.

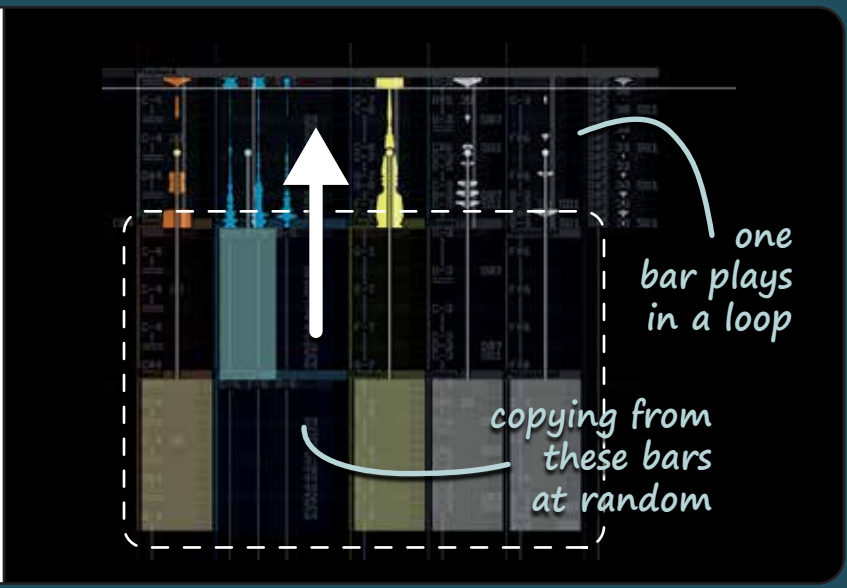
OBJECTIVE

A single bar of music repeats forever, but uses code to randomly select from different fragments.
This first task is about editing musical data - and how it is displayed and entered in Manhattan. Explore the pattern and the program to work out what's going on, and then use the tips overleaf to make the piece your own!

LEARN

- Get used to interacting with the Manhattan UI.
- Learn basic **music editing** (pitch, time, volume).
- Discover the power of **generative music**.
- Start thinking about music as **pattern + process**.

IMAGE



HOW

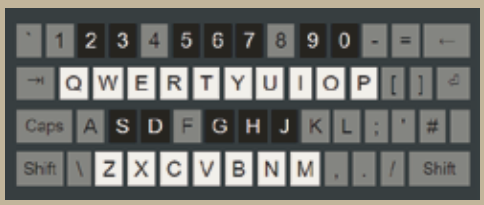
Manhattan represents **music as text** using a grid, like a musical spreadsheet. As such, everything is controlled using the **computer keyboard**. Use the **MUSIC REFERENCE** card to explore the music and UI.

Start by **playing the pattern** - move the cursor to top of the pattern and press **SPACE**. Then try making changes to the bars below row 20 (see **IMAGE**).

Don't be afraid to break things - you can always undo changes (see tips) or even reload the pattern!

TIPS

- **Confused?** Move the cursor to any part of the pattern and press **CTRL-H** / **CMD-H** to show help pages.
- Most values are simply typed, but notes and pitches are entered using the **virtual piano**.



- **New to music?** Just focus on the **white keys**. For drum parts, each key is a different drum.
- Get used to navigating with the **CURSORS** (← ↑ → ↓), as well as the **TAB / SHIFT-TAB** and **PAGE UP/DN** keys.
- Insert a gap by pressing **ENTER**, or clear / delete a cell by pressing **BACKSPACE / DELETE** or entering '**.**'
- Hold **SHIFT** while moving to select multiple cells, and use the clipboard like other programs.
- **Don't panic!** Use **CTRL-Z / CMD-Z** to undo mistakes.

HARMONY
Martian Jazz

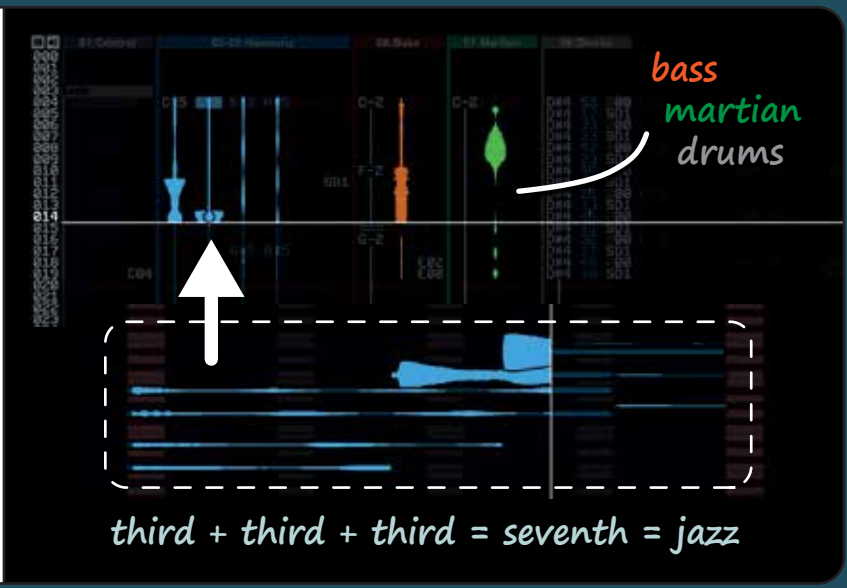
OBJECTIVE

Harmony is the combination of pitches separated by musical intervals, such as thirds and fifths.
In this piece, we will use simple arithmetic to build chords and create deceptively complex jazz harmonies, played on an EP – backed by drums, bass, and 'alien vocals'. As before, the whole piece is based on a single looped bar.

LEARN

- Learn how to write musical **formulas** in Manhattan.
- Begin using code to **procedurally generate** music.
- Learn about musical **harmony** and **consonance**.
- Understand the **mathematical** basis of music.

IMAGE



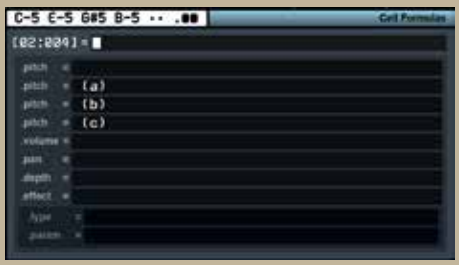
HOW

To create our chord, we will write three simple formulas for the pitches in the first row of the **Harmony** channel.

Move to cell `[02:004]` (Channel 2, Row 4) and press the **equals (=) key** to open the **formula editor**.

This channel can play four notes at once (as a chord), so there are four `.pitch` formulas. The first pitch, or 'root' of the chord, will be `C-5`, so doesn't need code. For the other three, we simply add either a major or minor third (chosen at random) to the previous pitch. When done, the pattern will 'jam' using your pitches!

TIPS



- The three lines of code for `.pitch` should appear at **(a)**, **(b)** and **(c)**,as shown to the right.
- There is more than one correct answer, but it is possible for all three formulas to be the same.
- **Harmony** is a group of **four** individual channels: 02, 03, 04, and 05 – one for each pitch.
- Use the **CODE REFERENCE** card to work out how things work and what to write. You need to find out...

  (1) how to read a pitch from another channel
  (2) how much a minor and major third interval is
  (3) how to randomly choose between the two
  (4) how to add the result to the previous pitch
  (5) how to put it all together in one line of code

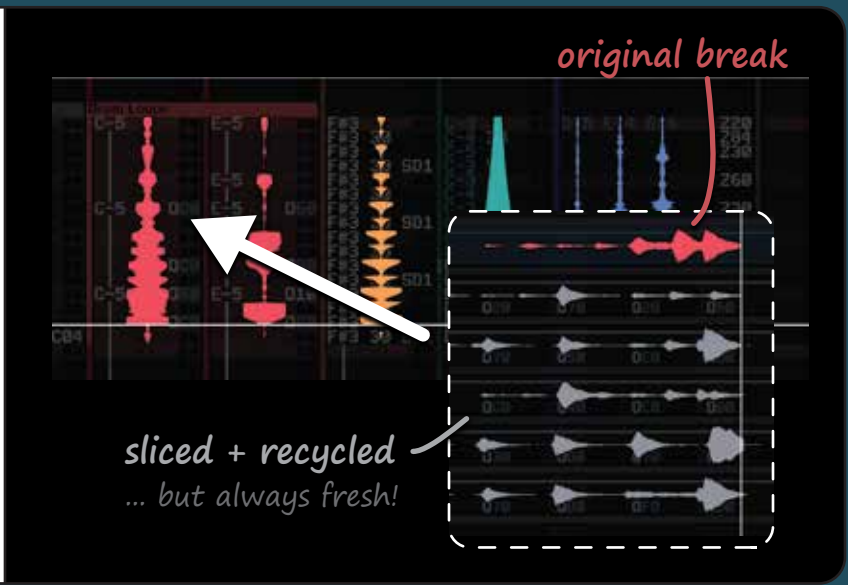
OBJECTIVE

Rhythm is created by controlling patterns in time. This piece takes drum loops (including the famous “amen break”), slices them up, and dynamically rearranges them to create new rhythmic patterns that change over time. By balancing control and randomisation – with both *what* and *when* we sample – an infinite variety of beats are possible.

LEARN

- Learn how to write musical **formulas** in Manhattan.
- Explore patterns in musical **time** and **rhythm**.
- Harness the creative power of **randomisation**.
- Use **arithmetic** to control the random process.

IMAGE



HOW

To slice the beat, we use the **Sample Offset** (`Oxx`) effect, then add code to set the parameter (`xx`) to a random offset, taking advantage of hexadecimal (base 16) to ensure a musical result.

Listen to the pattern, which plays the original loop. Now move to cell `[02:008]` (Channel 2, Row 8), enter '`O20`' as the effect (where the '`.▪▪`' is), and listen again. `Oxx` divides the loop into 256 (16 x 16) segments, so `20` (32 in hex) jumps playback to a new offset, two sixteenths (one eighth) into the full loop. Using the **MUSIC REFERENCE** card, play around with different positions and offsets of `Oxx` to get a feel for hex and beat slicing.

Next, we'll use code to slice the beat automatically (on-the-fly), by randomising the offset parameter. Press the **equals (=) key** to open the **formula editor** for one of your `Oxx` effects, and enter the following for its **.param** formula:

```
.param = rnd(0,15) * 10h
```

Press **ENTER** to accept the code and have a listen. The code generates a random number between 0 and 15, then multiplies it by 16 to ensure the offset aligns well with a beat in the loop.

Use the other channels, supplied parts, and **REFERENCE** cards to continue building your piece using code and beat slicing.

TIPS

- In computing, **hexadecimal** (base 16) is preferred to decimal (base 10) as 16 sub-divides nicely by 2 (i.e. 8, 4, 2, 1), which makes it perfect for music, where notes and times are often expressed as a quarters, eighths, or sixteenths of a bar.
- The **MUSIC REFERENCE** card provides a guide to hex, musical time and rhythm, to help with using the `Oxx` effect.
- In code, to specify a number in hex, add an 'h' (`10h` = 16) otherwise it will be treated as a decimal (`10` = ten).
- The code above ensures a multiple of 16 to align with beat – can you devise a formula to only fall between the beats?

LAYERING
Requiem + Repeat

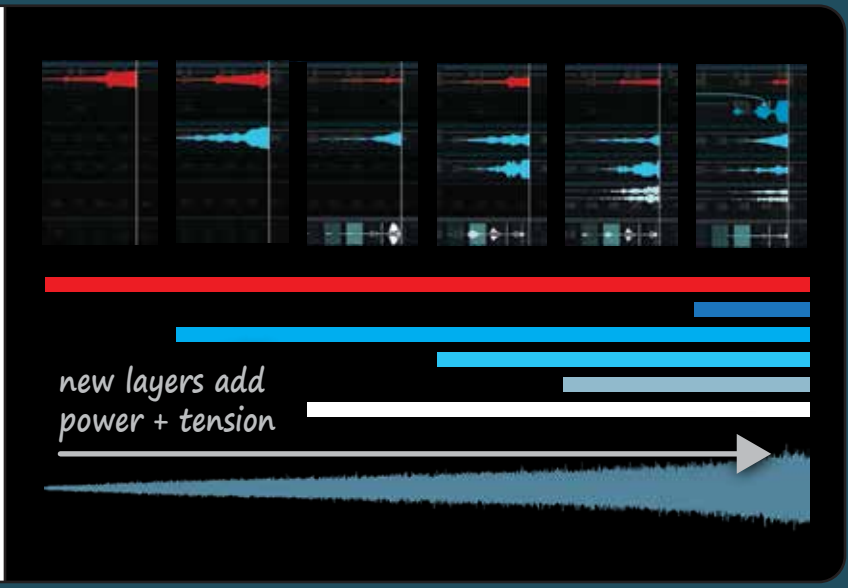
OBJECTIVE

Repeating a musical pattern with subtle variations is a key technique for composing impactful music. In this piece, up to six parts play simple motifs over two repeating bars. Over time, new parts are introduced on top of one another, adding new layers and tension to the music, before the process is reversed, and it fades away.

LEARN

- Learn to write reusable code as **macros**/**functions**.
- Learn to develop longer pieces by **layering parts**.
- Exploit the key role of **repeating patterns** in music, lending emotional impact to even subtle variations.

IMAGE



HOW

The six parts of this piece have already been written and can be toggled by muting / unmuting channels manually (**ALT-3/4/5/6/8**), but our goal is to control this process automatically with code.

For this, we create two **macro functions** that can be used and reused as effects in the pattern: **`1xx`** (to enable channel *xx*) and **`0xx`** (to disable channel *xx*). To edit the code for macros, press **CTRL-SHIFT-=**. To switch between the ten available macros (**`0xx`** to **`9xx`**) press **TAB**. Use the **CODE REFERENCE** to work out a single line of code for each macro: for **`0xx`**, set channel xx's **`.mute`** property equal to 1 (mute on = channel off); for **`1xx`**, set it to 0.

We want the bars to repeat a few times before we add or take away new channels, so we need a way of making changes over a longer period than the two bars visible. To this end, playback has been split between the looped bars (**Playback**) and another parallel thread, **Sequence**, that plays at a tempo 32x slower – so that it plays exactly one row for every repeat of the bars.

In cell **`[1:008]`**, enter **`104`** as the effect (where you see **`.▮▮`**). Now when you play the pattern, it should introduce the **Basses** after two repeats. Use **`1xx`** or **`0xx`** effects in the subsequent rows of **Sequence** to develop the piece as you see fit...

TIPS

- **Functions** are a key tool in programming, allowing code to be written that does a specific job, but which can also take input values (known as **arguments**) to vary their behaviour to adapt it to different situations. In this scenario, our function mutes/unmutes a channel and the argument specifies which.
- Explore the other formulas in the pattern, to see how each part depends on other parts. For example, the first cell of Cellos simply copes the Basses' part, transposing it up.
- **Parallel threads** are common in computing, but not music, highlighting just some of the potential of composing with code, which we'll explore further in other exercises.

COUNTERPOINT
5PM (Till I Count)

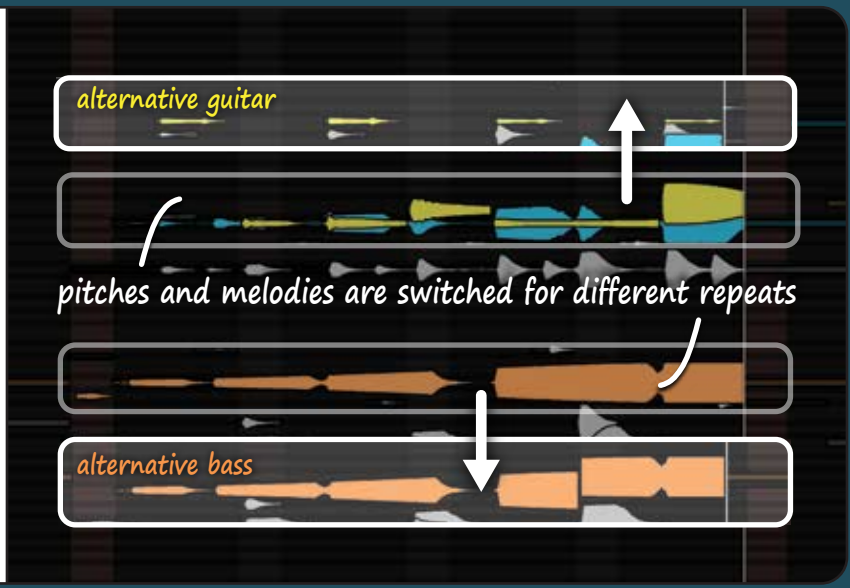
OBJECTIVE

Changing pitch over time (melodically) is the basis for melodies, bass lines, and chord progressions. Using a counter and conditional statements (if-then-else), this piece varies pitch over four bars to create the bass, chords, and melody of a well-known club/dance anthem, and demonstrate two powerful programming concepts.

LEARN

- Move from formulas to writing **code** in Manhattan.
- Create pieces that develop and change **over time**.
- Learn about **iteration** (loops and counters).
- Learn about **conditional statements** (if-then-else).

IMAGE



HOW

To reveal the piece, complete the code in the cells labelled **A**, **B**, and **C** – but instead of editing formulas, press **TAB** to use the more powerful **code mode**, which still runs when the cell is played, but can change or do practically anything.

For example, the code for **Counter** increments (+1) its value, (resetting at 5), to keep track of which iteration of the loop we're on, for use in the conditional statements in our cells.

In the cells you need to complete, code 'comments' tell you what your own lines of code need to do in plain language (or 'pseudocode'). Use the tips below and **CODE REFERENCE** to work out how to translate these into code... good luck!

TIPS

- Several of the cells in the pattern have been **labelled**, so they can be referred to by name, rather than address. To add/edit a label, move to the cell and press **@** or **"**.
- In code, labelled cells are also referenced using **@**:

  ```
  @Bass.pitch = A-3 ' set the pitch of cell "Bass"
  ```

- The **.param** value in a cell is simply a number (00 to FF), which can be used in code as a **named variable**, and for which the ".param" can usually be left out:

  ```
  @Counter = @Counter + 1 ' increment "Counter"
  ```

- Conditional (if-then-else) statements are powerful tools in programming, and can be used in many ways:

  ```
  @Counter == 3 ? .pitch = D-4
  @Counter < 4 ? @Chord = 15 : @Chord = 16
  @Chord = (@Counter < 4 ? 15 : 16)
  .pitch = (0.5 ? D-4 : A-3)
  ```

  Use the **CODE REFERENCE** to work out how these lines of code work – two may be useful; two do the same thing.

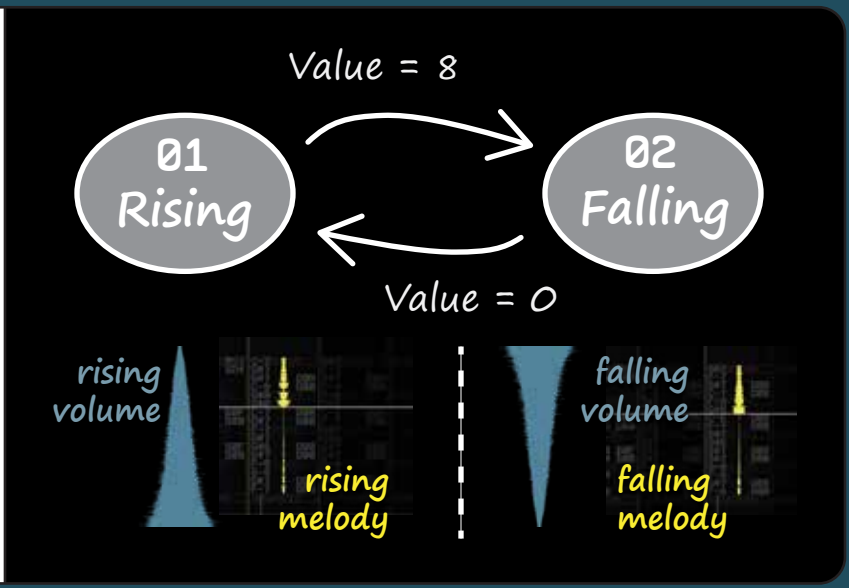
OBJECTIVE

Music is not only about note-to-note relationships; but processes that act over longer periods of time. In this piece, a bar repeats with a wobble bass, beat, and synth parts offering some subtle bar-to-bar variation. To add more variation over time, we switch between two longer phases, using rising or falling pitches and volume.

LEARN

- Move from formulas to writing **code** in Manhattan.
- Create pieces that develop and change **over time**.
- Learn about maintaining **states** using **variables**.
- Learn about **conditional statements** (if-then-else).

IMAGE



HOW

The piece plays in one of two **states**: ❶ **Rising** and ❷ **Falling**. Each is accompanied by a rising or falling melody so that you can hear which state you're in. To audition them, hit play and manually change the value of `State` from ▪▪ to `01` or `02`.

To complete the piece, we'll use code to switch between the states (and melodies), fading in **Synth 2** in ❶, and fading it out in ❷. The process (as illustrated in the **IMAGE**) works by incrementing `Value` while in ❶, switching to ❷ when we hit 8, then decrementing `Value` while in ❷, reverting to ❶ at 0. `Value` thus counts from 0 to 8 to 0, and can thus be used to control the **Synth 2** channel volume, to affect a fade in/out.

Open the formula editor for the **Code** cell (=) and press **TAB** to switch to the more flexible **code** mode, where comments describe the lines of code you need to add. Use the **CODE REFERENCE** to work out how to translate them in code.

TIPS

- Several of the cells in the pattern have been **labelled**, so they can be referred to by name, rather than address. To add/edit a label, move to the cell and press @ or ".
- In code, labelled cells are also referenced using `@`:

  `@State.param = 1` ' set "States" parameter to 1 (i.e. Rising)
- When not used to control an effect, the **.param** value in a cell is simply a number (00 to FF), and can be used in code as a **variable**, for which the ".param" is optional:

  `@Value = @Value + 1` ' increment "Value" by 1
- Conditional (if) statements are powerful tools in coding, executing code only if a condition is true:

  `@State == 1 ? @Value = @Value + 1`

  See the **CODE REFERENCE** for details and examples, or the **COUNTERPOINT** challenge for additional tips.

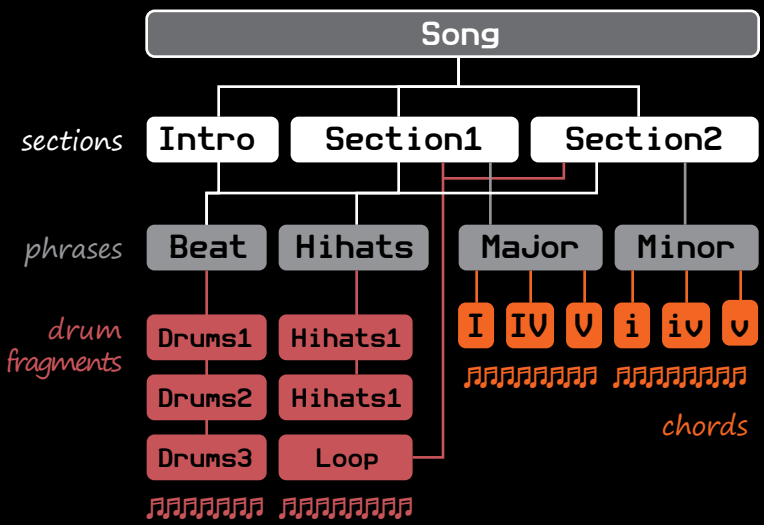
OBJECTIVE

Abstracting structure and relationships between elements is key in both music and programming. This piece explores layers of abstraction in pitch and time. Instead of individual pitches, we create harmonies using named chords; while the song itself is broken down into a hierarchy of three playback layers: song, phrase, and note.

LEARN

- Explore **parallel threads** in the context of music.
- Experiment with **harmony** and **chord progressions**.
- Learn how **abstraction** in both music and code can hide low-level detail to give us the “bigger picture”.

IMAGE



HOW

Any piece of music can be considered at various levels of abstraction: as notes, bars, phrases, sections, themes, etc. Higher levels allow us to more easily view the "big picture" and broader structure, which can be helpful in understanding and creating complex pieces.

All software in some way relies on abstractions to make it more powerful or accessible, but programming is a tool for *creating* abstractions: modelling data structures, processes, etc., which can follow traditional thinking, but also allows the coder to define new ways of interacting with a given field.

The piece provided here allows you to interact with music at a higher level than individual notes (as shown in the **IMAGE**):

1. the **Song** is broken into sections (**Intro**, **Section1**, **Section2**);
2. sections trigger phrases defining drum patterns (**Beats**, **Hihats**) or chord sequences (**Major**, **Minor**);
3. phrases trigger rhythm fragments (**Drums1**, **Drums2**, **Drums3**, **Hihats1**, **Hihats2**, **Loop**) or predefined chords (**I**, **IV**, **V**, **i**, **iv**, **v**);
4. the fragments and chords contain the actual notes (♫).

Explore the various levels of abstraction by making changes to each layer, to make the piece your own, using the **MUSIC** or **CODE REFERENCE** cards, if you get stuck.

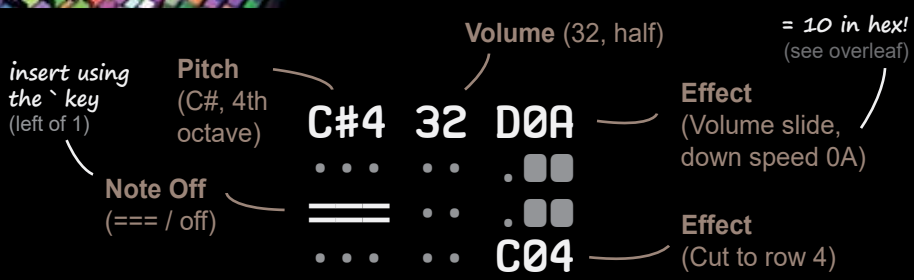
TIPS

- Start by looking at the code for **Major** and **Minor** (using cell **code** mode; press **TAB** to switch) and switching the chords that are triggered, and their order. (Note: to accept edits in cell code mode, you need to press **CTRL-ENTER**).
- You might think about tinkering with the drum parts to vary the rhythmic patterns or editing/extending the chord notes.
- Two blank segments (**Phrase1**, **Phrase2**) have been provided, should you wish to add new phrases.

MANHATTAN BYTES + PIECES

MUSIC REFERENCE

Notation, Examples, Shortcuts



Bass	Drums
C-3	Bass
C#3	Side Stick
D-3	Snare 1
D#3	Clap
E-3	Snare 2
F-3	Tom 1
F#3	Cl. Hihat
G-3	Tom 2
G#3	Pd. Hihat
A-3	Tom 3
A#3	Op. Hihat
B-3	Tom 4
C-4	Tom 5
C#4	Crash
D-4	Tom 6
D#4	Ride

	00 – 16 – 32 – 48 – 64
Volume	Silent···Half·····Full
Pan	Left····Centre···Right
Depth	Front···Middle····Back

Effects

- `@xx` Set tempo (to xx beats per minute)
- `Cxx` Cut to row xx (applied after row)
- `Dxy` Volume slide (at speed up/dn x/y)
- `Exx` Pitch slide down (at speed xx)
- `Fxx` Pitch slide up (at speed xx)
- `Gxx` Slide to pitch (at speed xx)
- `Mxx` Set channel volume (00-FF)
- `Oxx` Sample offset (see overleaf)
- `SBx` Repeat x times (SB0 = start)
- `SDx` Note delay (by x ticks)
- `Zxx` Filter cutoff (00-7F)
- `0xx` Macro 0 (using custom code)

* Effect parameters (x / xx) use hex (see overleaf)

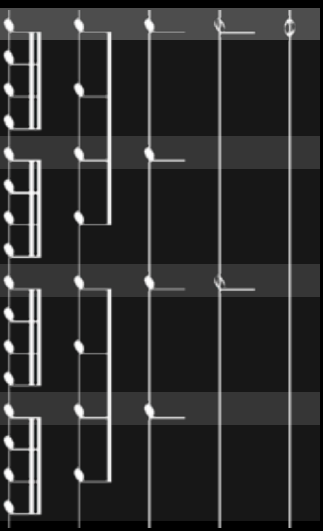
Keyboard Shortcuts

Ins/Enter	Insert
Ctrl/⌘A	Select All
Del/⌫	Delete
Alt-X	Clear
Ctrl/⌘X	Cut
Ctrl/⌘C	Copy
Ctrl/⌘V	Paste
Alt-Ctrl/⌘V	Overwrite
Ctrl/⌘M	Mix
Ctrl/⌘S	Save
Ctrl/⌘Z	Undo
Ctrl/⌘H	Show help description
Ctrl/⌘K	Show keyboard shortcuts
Space	Play from Cursor / Stop
Alt-Space	Play from Start
Alt-M	Mute/unmute channel
F8 x2	Reset Playback / Synth
Alt-↑/↓	+1 / -1 to selected value
Alt-I	Interpolate between values
(Shift-)Tab	Move one cell left (or right)
PgUp/PgDn	Jump up / down
Home/End	Jump left / right

Hex	Dec	Intervals	Chords						
			M	m	7	M7	m7	o7	mM
00	00	C Unison							
01	01	C# Minor Second							
02	02	D Major Second							
03	03	D# Minor Third		3			3	3	3
04	04	E Major Third	4		4	4			
05	05	F Perfect Fourth							
06	06	F# Augmented Fourth						6	
07	07	G Perfect Fifth	7	7	7	7	7		7
08	08	G# Minor Sixth							
09	09	A Major Sixth						9	
0A	10	A# Minor Seventh			10		10		
0B	11	B Major Seventh				11			11
0C	12	C Octave							
0D	13								
0E	14	7Fh = 127 FFh = 255							
0F	15								

Rhythm / Loop Offset (0xx)

10	16	00	0	Beat	Bass
11	17	10	16	1/16	
12	18	20	32	1/8	Cl. Hihat
13	19	30	48	3/16	
14	20	40	64	1/4	Beat Snare
15	21	50	80	5/16	
16	22	60	96	3/8	Cl. Hihat
17	23	70	112	7/16	
18	24	80	128	1/2	Beat Bass
19	25	90	144	9/16	
1A	26	A0	160	5/8	Op. Hihat
1B	27	B0	176	11/16	
1C	28	C0	192	3/4	Beat Snare
1D	29	D0	208	13/16	
1E	30	E0	224	7/8	Cl. Hihat
1F	31	F0	240	15/16	



User Interface

▶ Play from cursor ■ Stop / Reset ⚙ Pattern Options Channel Mixer

Switch vertical / horizontal layout

Toggle fullscreen

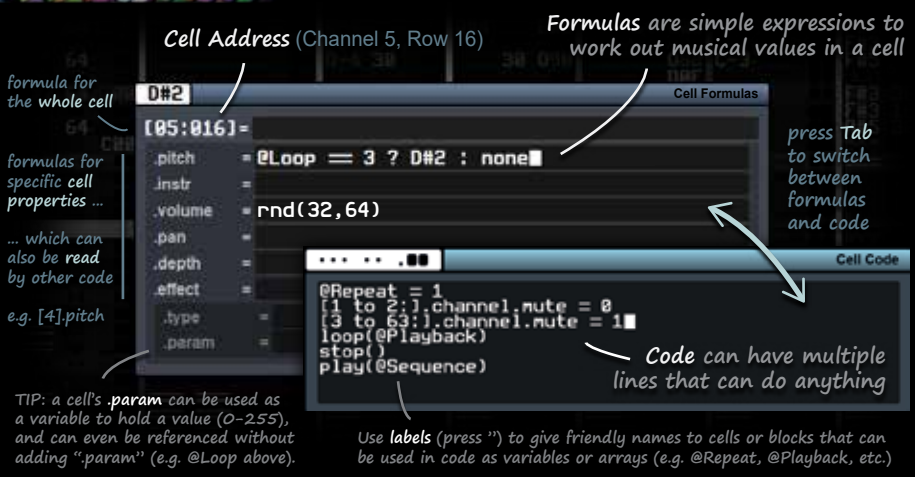
Cycle volume / pan / depth edit mode

? Show/hide info bar

$ / # Cycle editing / waveforms / pianoroll views

MANHATTAN
BYTES + PIECES

CODE REFERENCE
Syntax, Examples, Shortcuts



Keyboard Shortcuts

=	Show/edit cell formulas
Tab	Toggle formulas / code
Esc	Cancel edit
Enter	Apply formula edits
Ctrl-Enter	Apply cell code edit
Ctrl/⌘H	Show code help
" / @	Add/edit cell label
Ctrl-Shift-=	Edit macros / functions
Ctrl-Alt-=	Edit MIDI input code
Shift-=	Apply code to selection
Shift-Alt-=	Show all code (code listing)
Ctrl-L	Show error log

Operators

a + b	add
a - b	subtract
a * b	multiply
a / b	divide
a = b	assign (copy)
a mod b	modulo
(a)	brackets

Conditions

a == b	equal to
a != b	not equal
a > b	greater than
a >= b	... or equal
a < b	less than
a <= b	... or equal
a && b	logical AND
a || b	logical OR

Functions

copy(a)	copy a
copy(a,b)	copy a to b
copy(a,b)	copy a to b
move(a,b)	move a to b
clear(a)	clears a
mix(a,b)	mix a into b
rnd(a,b)	random value
goto(a)	jump to a
play(a)	play a once
loop(a)	play a looped
stop(a)	stop a
condition ? a	if condition is true, do a, else do nothing
condition ? a : b	if condition is true, do a, else do b
transpose(range,p)	transpose range to starts using pitch, p
arp(pattern,pitches)	arpegiate pattern using specified pitches
rotate(range,x)	cycle the contents of range by x rows

Cell Addresses

[x:y]	channel x, row y
[y]	current channel, row y
[x:]	channel a, current row
[+x]	a rows after current
[-y:]	a channels before current
[a to b]	range of cells from a to b
@Foo	cell / range labelled "Foo"
@Foo[0]	first cell in range "Foo"
@Foo[n]	cell at row n in range "Foo"
@Foo[n]	cell at row n in range "Foo"

Cell Properties

		e.g.
`.pitch`	musical pitch (0 to 120; C-0 to B-9)	`[4].pitch = C-5`
`.instr`	index of instrument (when channel instrument not set)	`[1:4].instr = 1`
`.volume`	volume of note (0 to 64)	`.volume = 32`
`.pan`	stereo pan position (0 to 64; left to right)	`.pan = .pan + 1`
`.depth`	surround depth position (0 to 64; front to back)	`.depth = 32 + 16`
`.effect`	aggregated effect (type + param)	`.effect = none`
`.type`	effect type (A-Z, 0-9, @; see music reference)	`@Foo.type = D`
`.param`	effect parameter (0-255, 00h-FFh; depends on type)	`@Var = .param`
`.row`	row index of cell (read only)	`@Position = .row`
`.track`	channel/track index of cell (read only)	`.volume = .track * 2`
`.repeat`	get current iteration of channel's SB0/SBx repeat	`.repeat == 3 ? 64 : 0`
`last`	get last note / property entry	`.pitch = last.pitch + 1`
`next`	channel/track index of cell (read only)	`next.pitch = .pitch - 1`

Other Objects

		e.g.
`.channel`	get current channel	`.channel`
`[x:].channel`	get channel x	`[1 to 4:].channel.mute`
`[-x:].channel`	get channel relative to current	`[-1:].channel.volume`
`.channel.mute`	enable (0) / disable (1) channel	`.channel.mute = 1`
`.channel.volume`	channel volume (0-255)	`[3:].channel.volume = 128`
`.channel.pan`	channel pan (0-64)	`[+1:].channel.pan = 64`
`.channel.eq`	3-band channel EQ (low/mid/high)	`.channel.eq.low = 76`
`.channel.compressor`	channel compressor (see help)	`.channel.compressor.ratio`
`.channel.reverb`	channel reverb send (0-255)	`.channel.reverb = 32`
`synth / synth[c]`	get synth for current/specified channel*	`synth = saw`
`synth.init()`	initialise synth preset (to basic saw)	`synth[2].init()`
`synth.load(p)`	load factory preset p (0 to 99)	`synth.load(14)`
`synth.osc1`	synth components (see help for details)	`synth.osc1 = triangle`
`synth.osc1.width`	synth properties (see help for details)	`synth.osc1.width = 64`
`guitar`	get virtual guitar* (one string per channel)	`guitar = flamenco1`
`guitar.string`	guitar components (see help for details)	`guitar.string.resonance`

* Channel should be set to a synth (00-31) / virtual guitar (GX/GY) using channel / mixer view (toolbar or F3).

Code Examples

```
[x:y].volume = 32                  set the volume in channel x, row y to half (32)
[+1].pitch = @Foo.pitch + 12       set the next cell's pitch an octave above Foo's
.volume = rnd(32, 64)              set a random volume between 32 and 64
@Scale[rnd(0,7)].pitch             pick a random pitch from the range "Scale"
.pitch = @High ? C-4 : C-6         pitch is C-6 if "High" is true (not zero), else C-4
@Bar == 6 ? play(@Melody)          if "Bar" is 6, play section labelled "Melody"
.pan = 0.5 ? 0 : 64                half (0.5) a chance of panning left (0) or right (64)
copy(1/3 ? @A : (1/2 ? @B : @C))   equal chance of copying "A", "B", or "C" (1/3 each)
```

```
@X = @X + 1                        use cell "X" as a counter, incrementing (+1) each time...
@X == 5 ? @X = 1                   ... reset "X" at 5 to create a loop (1,2,3,4, 1,2,3,4, etc.)
[2:].channel.mute = @X < 3         enable Channel 2 only when counter is 3 or 4
@X == 2 ? .channel.mute = 1        disable current channel when counter is 2
[3 to 5:].channel.mute = @X mod 2  enable channel 3-5 when "X" is even
```