Tutorial 4: Morphing

Morphing is a technique that takes two sequences, works out the differences between them and, during playback, subtly changes the first sequence so that it sounds like the second sequence.

Sequence morphing can be used to create long evolving passages that do not repeat though the end result is always predictable and reproducible. If *force-to-scale* is switched on then the notes generated will always be in the currently loaded scale.

How Does Morphing work?

Let's choose a couple of simple examples. To begin, we need to specify a source sequence and a destination sequence. The source sequence is always the currently selected sequence whilst the destination sequence must reside in the battery-backed memory. Let's say that the source sequence is only four notes long and consists of the notes C, D, E and G. We'll restrict the notes in the destination sequence to four notes as well: D, C, A and D.

When you start the *morpher*, *ZEIT* works out the difference between the note pitches and the note velocities in the source sequence and the notes pitches and note velocities in the destination sequence (The *Controller values* are not changed).

Let's consider the first note in each sequence. In the source sequence this is the note C. In the destination sequence, this is the note D. The difference between the two notes is two semitones. The destination note, D, is higher in pitch than the source note, C and so the pitch change is going up the scale. We call this a *positive* change.

Now look at the second note in each sequence. In the source sequence this is the note D. In the destination sequence this is the note C. The difference between the two notes is again two semitones but, in this instance, the destination note is below the source note and so the direction of change is *negative*.

If we repeat this process for each note in the sequence then we can construct a table of values:

Source	C3	D3	E3	G3
Destination	D3	C3	A3	D4
Difference	2	2	5	7
Direction	+	-	+	+

To begin the process, the source sequence plays through once. At the end of this *pass*, *ZEIT* calculates a new value for the note pitch in each step. The modified sequence plays

through once and *ZEIT* again updates the note pitch for each step. This process repeats until the morphed source sequence is identical to the destination sequence.

So, what would the above sequence sound like and how many times would the sequences play through until the source and the destination sequence sound the same?

The time taken before the source sequence sounds like the destination sequence is based on the largest difference in pitch between the source and destination sequence. In the above example, the largest difference is 7 semitones and so the time taken for the source sequence to match the destination sequence would be (seven + one) passes, ie. 8 passes.

Let's assume that *force-to-scale* is not enabled.

Pass	Step No.			
	1	2	3	4
1 – Source Sequence	C	D	E	G
	+1	-1	+1	+1
2	C#	C#	F	G#
	+1	-1	+1	+1
3	D	C	F#	A
	0	0	+1	+1
4	D	C	G	A#
	0	0	+1	+1
5	D	C	G#	B
	0	0	+1	+1
6	D	C	A	C
	0	0	0	+1
7	D	C	A	C#
	0	0	0	+1
8 - Destination Sequence	D	C	A	D

So, what do we hear? On the first pass through, *ZEIT* would play the source sequence, C D E G. On the second pass through, *ZEIT* adds one semitone to the first note taking the C to a C#. The second note moves downwards in pitch from D to C#. The third note moves up in pitch from E to F and the fourth note also moves up in pitch from G to G#. So, on the second pass through we would hear the *morphed* sequence C# C# F G#. And so on, until the source sequence sounds like the destination sequence.

Ok, let's try using the same source and destination sequences except on this occasion, we'll switch on *force-to-scale*. *Force-to-scale* takes the incoming note and matches it against the current scale. If the note is in that scale then the note is not affected by *force-to-scale*. However, if the note is not in the current scale then the *force-to-scale* processor looks for the nearest note in the scale **below** the incoming note. For example, if you passed a C to the *force-to-scale* unit and the current scale was C major then the

note would be unaffected. If instead, you passed a C# to the *force-to-scale* unit then it would output the nearest note below C# which is C.

Pass	Step No.							
	1		2		3		4	
1 – Source Sequence	**C**		**D**		**E**		**G**	
	+1		-1		+1		+1	
2	C#	**C**	C#	**C**	F	**F**	G#	**G**
	+1		-1		+1		+1	
3	D	**D**	C	**C**	F#	**F**	A	**A**
	0		0		+1		+1	
4	D	**D**	C	**C**	G	**G**	A#	**A**
	0		0		+1		+1	
5	D	**D**	C	**C**	G#	**G**	B	**B**
	0		0		+1		+1	
6	D	**D**	C	**C**	A	**A**	C	**C**
	0		0		0		+1	
7	D	**D**	C	**C**	A	**A**	C#	**C**
	0		0		0		+1	
8 - Destination Sequence	D	**D**	C	**C**	A	**A**	D	**D**

What's happening here? As before, on the first pass through, the source sequence plays C D E and G. On the second pass, the *morpher* adds one semitone to the first note, taking the C to a C#. However, the note C# isn't in the scale of C major and so this is *forced* to the new pitch of C, so there's no apparent change in pitch. However, with the second note in the sequence, the D is pitched downwards to C# and, as we've already seen, this isn't in the key of C major. Consequently, the D note is also *forced* to the note C. In the third note, the E is pitched upwards to F and, since F is in the scale of C major, then the *force-to-scale* function has no effect. Finally, the G is pitched upwards to G# and, as before, the *force-to-scale* function will re-pitch this note at G.

So, on the first pass we'll hear the source sequence C D E and G. On the second pass, we'll hear C C F and G (as shown in bold font in the table above).

Let's look at what happens on the third pass through. The first note, which is C#, is pitched upwards by a semitone to the note D. Here, the *force-to-scale* function has no effect. The second note, which was C#, is pitched downwards by one semitone to C and, again, the *force-to-scale* function has no effect. The third note, F, is pitched upwards to F# but modified by the *force-to-scale* function back to the original F. The fourth note, which was G#, is pitched upwards and becomes A. Hence, on the third pass, our *morphed* sequence plays D C F A.

Try setting this up for yourself using the *note editor* to set the various parameters.

Note Velocities

Are the *note velocities* affected by the *morphing* function? Yes, they are.

Let's stick with the current example where the source sequence is C D E G and the destination sequence is D C A D but, here, we'll give the notes a velocity component. The velocity components are shown in parentheses.

Source	C3 (50)	D3 (60)	E3 (70)	G3 (80)
Destination	D3 (50	C3 (74)	A3 (98)	D4 (38)
Velocity difference	0	+14	+28	-42
Direction	0	+	+	-

In the first note, there's no difference between the velocity values. In the second note, the source note D is played with a velocity of 60 and the destination is played with a velocity of 74. So, the difference between the two notes is +14. In the third note, the source velocity is 70 and the destination velocity is 98 and so there is a difference of +28. Finally, in the fourth note, the source note is played with a velocity of 80 and the destination with a velocity of 38 and so there is a difference of –42.

How are the note velocities *morphed*? Well, we saw in the previous example where *force-to-scale* was enabled that the total number of passes required to *morph* the source sequence into the destination sequence was 7. However, to *morph* the velocity components, we require one less pass and so we divide the velocity difference by the number of passes minus 1. So, in this example, the number of passes required for the velocity is 6.

And so the table above now looks like this:

Source	C3 (50)	D3 (60)	E3 (70)	G3 (80)
Destination	D3 (50)	C3 (74)	A3 (98)	D4 (38)
Velocity difference	0	+14	+28	-42
Direction	0	+	+	-
Velocity Increment	**0**	12/6 = **2**	24/6 = **4**	42/6=**7**

Now, what would this sound like?

	1	2	3	4
1	C(50)	D(60)	E(70)	G (80)
2	D(50)	C(62)	F(74)	A(73)
3	D(50)	C(64)	F(78)	A(66)
4	D(50)	C(66)	G(82)	B(59)
5	D(50)	C(68)	G(86)	C(52)
6	D(50)	C(70)	A(90)	C(45)
7	D(50)	C(72)	A(94)	D(38)

As the *morph* progresses then we'll hear the loudness of first step remain exactly the same whereas the second and third steps will gradually increase in volume and the fourth step will slowly decrease in volume.

Flipping

Suppose now that the source sequence and the destination sequence have different clock rates, different directions and different lengths. What effect does the *morpher* have on these parameters?

Let's look at a longer sequence.

Step Number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Source	C	C	C	D	E	F	C	D
Destination	E	E	D	B	A	A	F	F
Difference	+4	+4	+2	+9	+5	+4	+5	+1

The maximum difference in pitch occurs at step four, where the source pitch is D and the destination pitch is B. The difference is 9 semitones, which means that the *morph* function will take (9 + 1) passes to complete.

We'll assume, for simplicity, that the source and destination sequences have the same directional settings. Here, both sequences are moving in the *forwards* direction.

	1	2	3	4	5	6	7	8
1 Source	C	C	C	D	E	F	C	D
2	C#	C#	C#	D#	F	F#	D#	D#
3	D	D	D	E	F#	G	D	E
4	D#	D#	D#	F	G	G#	D#	F
5	E	E	E	F#	G#	A	E	F
6	E	E	E	G	A	A	F	F
7	E	E	E	G#	A	A	F	F
8	E	E	E	A	A	A	F	F
9	E	E	E	A#	A	A	F	F
10 Destination	E	E	E	B	A	A	F	F

Now, let's change the direction of the *destination* sequence so that it's now moving backwards.

The most logical place to *flip* the direction and/or clock rate is at the mid-point in the *morph* and the midpoint is calculated by dividing the total number of passes by 2. If the total number of passes was 10 then the mid-point would fall at the end of pass 5, which is where the *flip* would take place.

When the *morph* begins, playback will be in the *forwards* direction. After pass number 5, the playback direction will change to that of the destination sequence i.e. backwards. What effect will this *flip* have on the above table? Let's see.

	1	2	3	4	5	6	7	8
1 Source	C	C	C	D	E	F	C	D
2	C#	C#	C#	D#	F	F#	D#	D#
3	D	D	D	E	F#	G	D	E
4	D#	D#	D#	F	G	G#	D#	F
5 (*flip!*)	E	E	E	F#	G#	A	E	F
6	E	E	E	G	F	F	A	A
7	E	E	E	G#	F	F	A	A
8	E	E	E	A	F	F	A	A
9	E	E	E	A#	F	F	A	A
10 Destination	E	E	E	B	F	F	A	A

Suppose the source sequence is clocked at a rate of 2x and the destination sequence is clocked at a rate of 4x. The *morph* would begin at the source clock rate of 2x. It would remain at this clock rate until the end of pass 5, upon which it would change to the destination clock rate of 4x.

Active and Muted Steps

Let's use the previous example, this time, with active and muted steps:

Step Number	1	2	3	4	5	6	7	8
Source	C	C	C	D	E	F	C	D
Active	Y	Y	Y	Y	Y	Y	Y	N
Destination	E	E	D	B	A	A	F	F
Active	Y	Y	N	N	Y	Y	N	N
Difference	+4	+4	+2	+9	+5	+4	+5	+1
Flip pass	3	3	2	5	3	3	3	2

First, let's see what the sequence would sound like if all of the steps were active. For simplicity, *force-to-scale* is not enabled and the shaded cells indicate that the step has reached its target pitch.

	1	2	3	4	5	6	7	8
1 (Source)	C	C	C	D	E	F	C	D
2	C#	C#	C#	D#	F	F#	D#	D#
3	D	D	D	E	F#	G	D	E
4	D#	D#	D#	F	G	G#	D#	F
5	E	E	E	F#	G#	A	E	F
6	E	E	E	G	A	A	F	F
7	E	E	E	G#	A	A	F	F
8	E	E	E	A	A	A	F	F
9	E	E	E	A#	A	A	F	F
10	E	E	E	B	A	A	F	F

As before, the maximum difference in pitch occurs at step four, where the source pitch is D and the destination pitch is B and the difference, in semitones, is 9.

However, step four in the destination sequence is muted and so *ZEIT* must decide when to mute step four. *ZEIT* works out the difference in pitch between the source step and the destination step and divides this value by two, adding one if the total number of passes is an odd number. So, step four will play in pass one, two, three and four but will be muted on the fifth pass.

What would playback sound like? In the following table, the step number runs horizontally and the pass number runs vertically. As before, the greyed-out boxes indicate steps where the morphed pitch matches the destination pitch. The blacked-out boxes indicate muted steps.

	1	2	3	4	5	6	7	8
1	**C**	**C**	**C**	**D**	**E**	**F**	**C**	
2	C#	C#	C#	D#	F	F#	D#	
3	D	D	D	E	F#	G	D	
4	D#	D#		F	G	G#		
5	E	E		F#	G#	A		
6	E	E			A	A		
7	E	E			A	A		
8	E	E			A	A		
9	E	E			A	A		
10	**E**	**E**			**A**	**A**		

So, the *morphed* source sequence will be the same as the destination sequence after only 6 passes and so the entire *morph* operation will be considerably shorter.

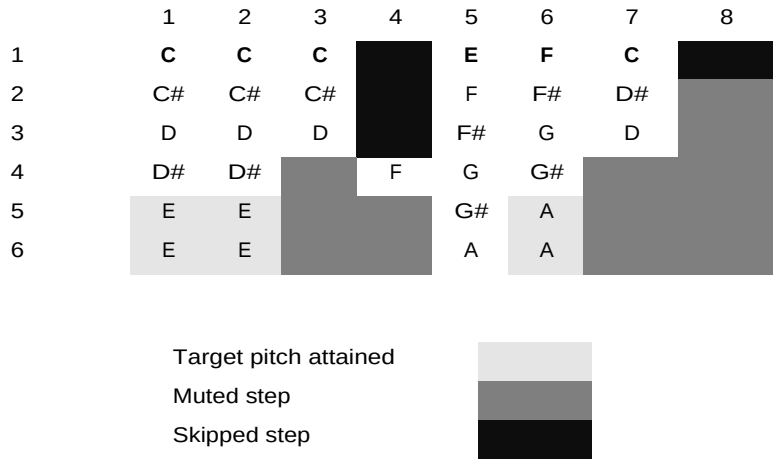
Skipped Steps

Skipped steps require careful consideration. Let's look at the previous example again, this time replacing some of the muted steps with *skipped steps*.

Step Number	1	2	3	4	5	6	7	8
Source	C	C	C	D	E	F	C	D
Active	Y	Y	Y	Y	Y	Y	Y	N
Skipped	N	N	N	**Y**	N	N	N	**Y**
Destination	E	E	D	B	A	A	F	F
Active	Y	Y	N	N	Y	Y	N	N

Skipped	N	N	**Y**	N	N	N	**Y**	N
Difference	+4	+4	+2	+9	+5	+4	+5	+1
Flip pass	3	3	2	5	3	3	3	2

And the sequence would play as:



Looping and Morph Modes

In normal circumstances, a *morph* would simply begin with the source sequence and proceed until the source sequences sounds the same as the destination sequence. Once the *morph* has completed, the destination sequence will play until the sequencer itself is stopped. However, you can make a *morph* restart simply by enabling *loop mode*.

When looping is on, the source sequence is morphed until it matches the destination sequence. When it matches the destination sequence, the process is reversed so that the destination sequence becomes the source sequence. The morphing process then begins to work back towards what was the original source sequence. We've found that this method creates some very smooth, evolving passages.

Morph Dwell is the number of times the destination sequence plays once the morph process has completed. If this parameter is set to, say, three then the source sequence will morph until it matches the destination sequence. The destination sequence will then play three times before restarting the morph back towards the original source sequence.

Hints and Tips

Morphing is a great way of finding new sequences that are closely related to the sequences you're currently working on. We've used it many, many times to create a large number of subtle variations based around an original theme.

To do this, pick your source and destination sequences, start the ***Morpher*** with the ***data wheel*** and hit the ***Play*** button. When you hear a sequence variation that you like, simply turn the ***data wheel*** one step anti-clockwise to stop the ***morphing process*** at that point. The sequence will continue to play as normal and will not be over-written by the next progression until you start the ***morpher*** running again. This leaves you with time to manipulate and save the sequence as required.

Restarting the ***Morpher*** will start the process off again with your original sequences.

Some Morphing questions

What happens to the Controller Data?

The governing factor in the morphing process is always the note values. The controller values are generated and processed without modification. However, you can still change them from the front panel controls, the low frequency oscillator, the sweep generator etc.

Can I have more than one morph running at one time?

Unfortunately, this is not currently possible. Morphing is very hungry on processor resources and, at present, *ZEIT* is not able to generate more than one morph. This may change in future software revisions.

Morphing only seems to work with the Forward, Reverse and Pendulum directions?

Yes, this is true. As stated, morphing is a complicated process. Adding randomised and algorithmic directions to the process takes up too many CPU resources. Again, this may be addressed in future software revisions.