4 LEMorpheus software infrastructure

This chapter will detail pertinent aspects of the supporting infrastructure of the software (*LEMorpheus*) I created to experiment with automated and interactive note sequence morphing algorithms. This provides synoptic insights into the note sequence morphing system, rather than explanations of particular algorithms. The infrastructural knowledge contributed here can be applied to note sequence morphing and, in some cases, more widely to algorithmic composition. This description of infrastructure will also assist in understanding the note sequence morphing algorithms discussed later, in chapters five, six and seven.

Firstly, an overview of the software infrastructure (4.1) explains, at a high level, how various system components for morphing relate to each other. A simple diagram is used to summarise the system workings, clarified by a written explanation. This was chosen over a diagram using the detailed modelling language UML (Unified Modelling Language), which appeared overly complex.

Following this, the aspects of the system which are easily controlled by the user are explained, from the high-level Graphical User Interface (GUI) I designed for morphing (4.2), to the layout of the note sequence editor (4.3). The "meta-morphing" parameters which are available to all compositional morphing algorithms are then detailed (4.4), including parameters to control inter-part tonal communication as well as a number of morph index transformations that operate in parallel on different parameters. Pertinent aspects of the note sequence morphing software infrastructure are then explained (4.5), including music representations, and the extensible design of the software. Lastly, the method for producing MIDI output in realtime is described (4.6).

Because compositional morphing is the focus of the research, the software infrastructure was only evaluated in simple "pass/fail" terms of whether it could sufficiently support the morphing algorithms. While it has obviously succeeded, there are numerous directions for improvement of the software infrastructure that have become apparent and these possible extensions to the architecture are expressed in the summary (4.7).

Throughout this chapter, only the system architecture components related to the topic of compositional morphing will be explained. Many other aspects of *LEMorpheus* required substantial development effort to implement and made certain tasks easier, however, they are not discussed anywhere within the thesis because they are not directly relevant. For example, file saving and custom-built interface components. Despite this, curious readers can refer to the source-code in the folder "4. digital appendix" on the accompanying CD.

4.1 LEMorpheus overview

LEMorpheus has been designed to enable investigation into interactive note sequence morphing between two MIDI sequence loops, within the musical context of Mainstream Electronic Music (MEM). *LEMorpheus* allows considerable realtime control over note sequence morphing and the software is flexible and extensible. It is written in the Java programming language using a personalised realtime branch of the jMusic (Sorenson and Brown 2004) open source Java music library and the Midishare (Letz 2004) open source MIDI in/out library.

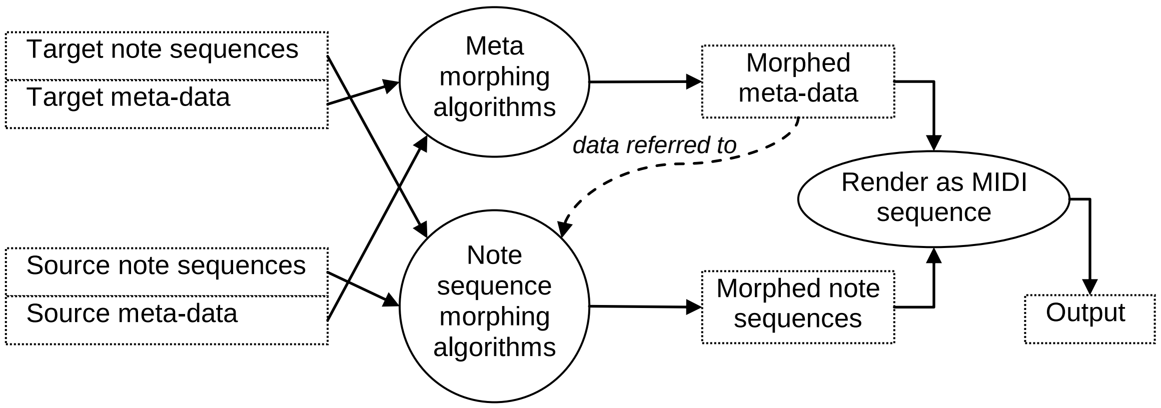

Figure 1 **Overview of the infrastructure for morphing in *LEMorpheus***

Before and during morphing, the user can edit the note sequences of the source and target loops. The meta-data of the loops can also be changed, for example, the tempo (beats per minute) or the key and scale labels for a particular part. The type of algorithm that will be used to generate the notes during the morph (the "Note sequence morphing algorithm" circle in Figure 1) can be selected and various parameters specific to the algorithm can be tweaked. The algorithm for morphing meta-data ("Meta morphing algorithm" in Figure 1) is interpolation, with adjustable parameters that influence the interpolation for particular types of meta-data, as described in further detail below (4.3).

Note morphing algorithms, such as those explained in chapters five to seven, can be either realtime or non-realtime. Realtime morphing algorithms are able to respond to adjustments of parameters and source and target note sequences while the morph is progressing. Non-realtime algorithms have too much time complexity for this; instead, a list of note sequence frames is rendered beforehand and different frames are selected for playback depending on the morph index. The use of frames is explained in more detail when describing the non-realtime *TraSe* algorithm in chapter seven.

To generate the final stream of MIDI events, the morphed meta-data is applied to the morphed note sequence data. This process is different depending on which note sequence morphing algorithm is selected and the type of meta and note-level representations that are used by it, as explained below (4.5).

4.2 High level morphing controls

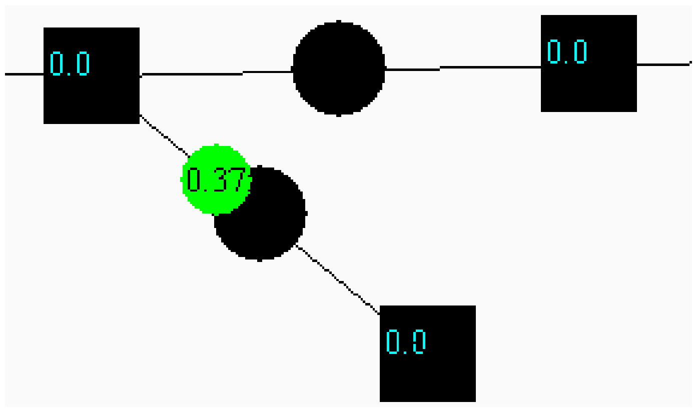

Figure 2 High-level user interface for LEMorpheus.

At the highest level of control the user is presented with a simple graphical interface of multi-part loops (the square boxes in Figure 2) connected to other loops via a morph (the line with a circle in the middle in Figure 2). A loop can be played by clicking a box, and morphing between loops can be initiated by clicking on a circle. While a morph is playing, the morph index is controlled and displayed by a green ball that automatically moves from one side to the other, unless dragged directly by the mouse pointer.

Loops can be arranged in progression, analogous to the way a DJ might prepare a playlist, except that any non-linear path can be created. The morph index can be moved using external input from MIDI or the *reacTIVision* video tracking software (Bencina, Kaltenbrunner and Costanza 2006). The morph index can also be controlled by a variable in a computer game. Both computer game and table top interfaces have been implemented, however, further discussion of these components is reserved for chapter eight, as they are more related to future applications than software infrastructure.

4.3 Loop editor

The loop editor within *LEMorpheus* allows for multi-part loops to be edited and played back and is abstracted by the layout diagram below:

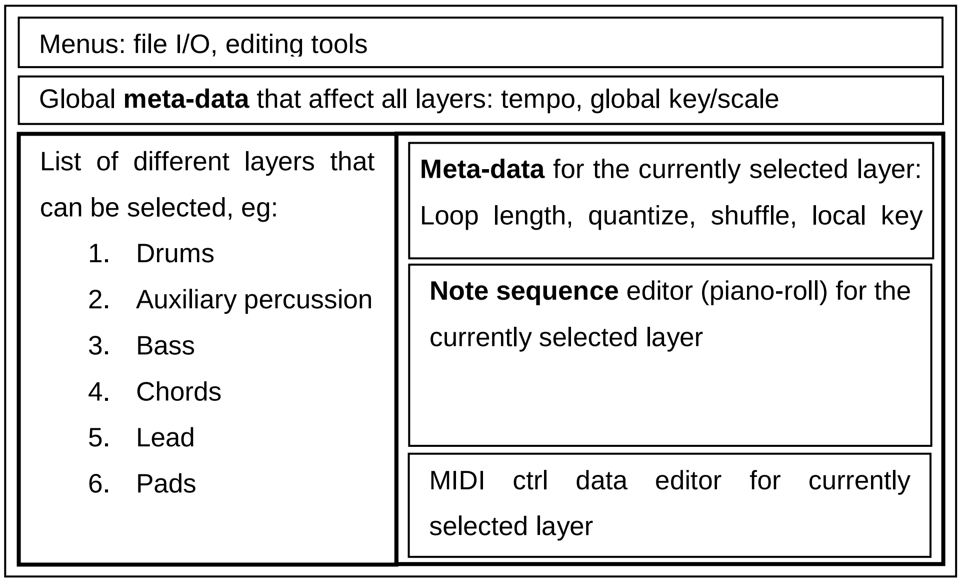
Menus: file I/O, editing tools
Global **meta-data** that affect all layers: tempo, global key/scale
List of different layers that can be selected, eg:
1. Drums
2. Auxiliary percussion
3. Bass
4. Chords
5. Lead
6. Pads
Meta-data for the currently selected layer: Loop length, quantize, shuffle, local key
Note sequence editor (piano-roll) for the currently selected layer
MIDI ctrl data editor for currently selected layer

Figure 3 Layout of the loop editor interface.

The loop editor (Figure 3) is similar to many other standard MIDI sequencers, with file and edit menus, global parameters such as tempo, a list of parts/layers on the left, a piano-roll note editor for the currently selected layer to the right; various meta parameters such as instrument, channel, length, quantize, shuffle, key and scale above the note editor; and a graph of MIDI controller data below the note editor.

Unlike regular sequencers, the ordering of layers within the list has an important musical effect: layers at the same level (vertical slot) in different loops will be morphed together. For example, if the drums were on the third layer in the source, the user should put the target drums, or the most drum-like part from the target, also on the third layer (see Figure 4).

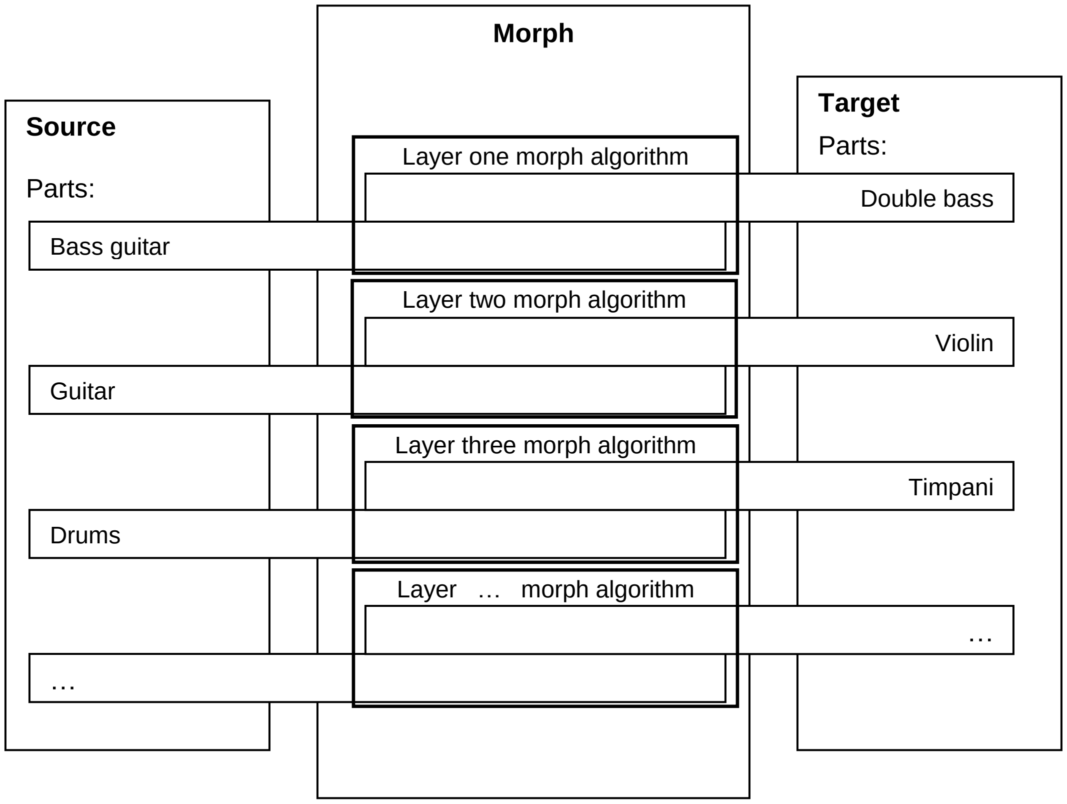

Figure 4 **Parts that appear on the same vertical slot in source and target will be morphed together into the same layer. They may have different MIDI channels and instruments.**

A similar principal follows with other musical functions such as bass, lead and pads. Unlike *The Musifier* (Edlund 2004), the user need not explicitly use these part-types, but any parts that have a similar musical function are able to be correlated.

4.4 Morphing parameters

Each morph (between source and target loops) has a number of parameters that can be manually configured, separately to the other morphs. Some of these parameters affect the morph as a whole, others relate to each individual layer being morphed and others are specific to the note sequence morphing algorithms selected for each layer. In this section I will discuss parameters that affect the morph as a whole and parameters that relate to each individual layer. Parameters that are specific to each note sequence morphing algorithm will be explained, along with the algorithm, in chapters five, six and seven.

4.4.1 Parameters affecting the morph as a whole

There are only two parameters that affect the morph as a whole – the *morph length* and the *tonal leader/follower*. The *morph length*, in beats, can be set to commonly occurring cycle lengths such as 2, 3, 4, 6, 8, 12, 16, 24, 32 etc up to 1024. Only one *tonal leader* can be specified, and any number of layers can be marked as a *tonal follower*. Each *tonal follower* will take the key and scale data from the *tonal leader* in realtime. The only other feature that affects the morph as a whole is the saving and loading of all parameters simultaneously, including those parameters that affect each layer individually.

4.4.2 Parameters affecting each layer individually

For each layer, there are 18 manual parameters that affect the morphing for that layer. Some of these are structural and one of them influences the note sequence morphing algorithm, however, most parameters are for morph index transformation functions, which are applied to interpolation of meta parameters from the source and target. In addition, the particular note sequence morphing algorithm for each layer can be selected from a list that includes parametric morphing (see chapter five), the *Markov Morph* (chapter six) or the *TraSe* morph (chapter seven).

Structural parameters

Structural parameters allow control over elements of the morph structure for a particular layer through volume, repetition and morph index quantization.

If the two parts from source and target that are being morphed together in the same layer (for example, “bass guitar” from source and “double bass” from target in Figure 4) are set to different MIDI channels, the volume of the two channels will be cross-faded and the MIDI events (notes) for that layer will be simultaneously sent to both MIDI channels. In this way, it is possible to morph between parts with different instruments[1].

[1] Use of the same MIDI channel and different instruments for source and target was also experimented with briefly. A program change was sent immediately before notes where necessary, however, the time taken by synthesisers to load the different instruments was inhibiting.

However, because of the perceived drop in loudness when using a linear cross-fade (blue in Figure 5), the gradient of the fade-in (dashed lines) and fade-out (solid lines) usually needs to be increased. For practical purposes this was sufficient, however the equal power (logarithmic) curves used in two-channel DJ mixing desks are probably optimal. In Figure 5, the red arrows show a change in gradient and the red lines show the resulting volume envelopes for the two MIDI channels. The user may prefer one timbre to enter earlier or later than usual, so that the volume envelope for each MIDI channel of source and target can be offset independently, as shown by the green lines in Figure 5, which are shifted by differing amounts. The lengths of the green arrows show different degrees of offset from the standard cross fade (blue).

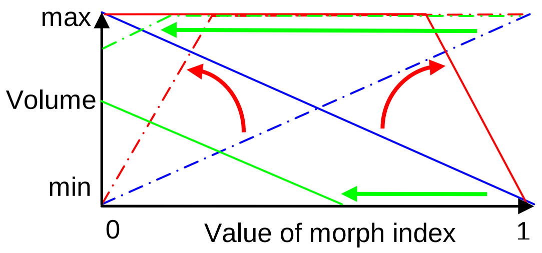

Figure 5 **Volume cross-fade functions. Solid lines for source MIDI channel volume, dashed for target MIDI channel volume. Blue shows the linear cross-fade, red shows a change in gradient, green shows a change in offset.**

Often it does not make musical sense for a morph to be completely smooth. For example, abrupt cuts are often used by composers to effectively segue into a new theme[2]. In response to this, a parameter can be used to quantize the morph index before it is applied to the note sequence morphing algorithm of a particular layer. This is not a complete solution, but it means that the morph can be easily split into similar sounding segments. Effectively, the morph index that is sent to the note sequence morphing algorithm remains the same throughout these similar sounding segments, even when the original morph index is changing, as shown in Figure 6.

[2] As shown in the results of the web questionnaire within chapter seven.

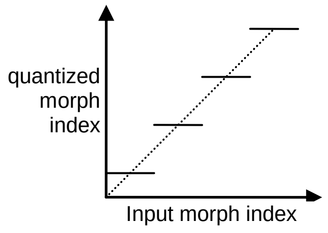

Figure 6 **Example of the morph index being quantized into four discrete values. The dashed line is the original morph index, while the solid line is the quantized morph index.**

The quantization does not always have a perceivable effect on algorithms that are non-deterministic and somewhat unstable. For these situations, an option to repeat the recently generated output over a section of the quantized morph index is also available. When *repeat* is on, the most recent output is looped, the length of the loop being specified by the length of the quantized segment. This effectively negates the instability of the non-deterministic algorithms, and contributes to a sense of structural regularity.

Morph index transformations

The morph index is split into multiple parallel morph indices which are each transformed separately before being applied to the interpolation algorithm for each meta-data parameter, as well as the note sequence morphing algorithm for that layer. These morph indices are separate so that transformation functions that are applied to them can be “tuned” independently, according to the required musical effect. The meta-data parameters are: loop length, rhythmic quantization value and rhythmic shuffle value. The morph indices for the interpolation of each of these is transformed specifically for that meta-data parameter using the simple mathematical functions like the orange, green and blue lines shown in Figure 7.

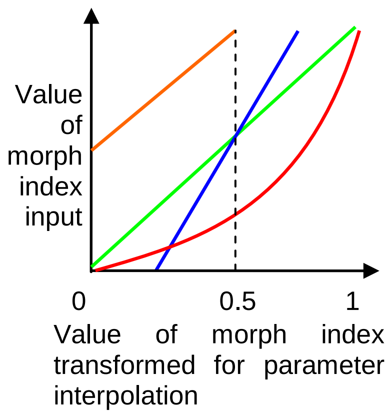

Figure 7 **Example interpolation curves applied in parallel for different values of meta-data parameters: cross-over point of 1 and gradient 1 (orange), crossover point 0.5 and gradient 2 (blue), crossover point 0.5 and gradient 1 (green), exponential curve for note morph index (red).**

The user specifies two values, the gradient and crossover point, for each of these morph index transformation functions, with a total of six controls over the interpolation of the three parameters: loop length, rhythmic quantisation, and rhythmic shuffle. The crossover point means the point at which the output morph index equals 0.5. That is, if the crossover point is set to 0, when the input morph index equals 0, the transformed morph index will equal 0.5. For example, the orange curve in Figure 7 has a crossover of 1, meaning that when the input morph index is 1, the transformed morph index is 0.5. The crossover point must be between -0.5 and 1.5. The term ‘crossover’ was preferred rather than ‘offset’, as I perceived the values in relation to the centre of the morph rather than the start. The gradient, or slope of the curve, must be greater than 0 (flat), for example, the steeper slope on the blue line is due to it having a gradient of 2.

For the morph index that is sent to the actual note sequence morphing algorithm, the transformation includes an exponential (or logarithmic, depending on the setting) curve, in addition to the crossover and gradient of the other parameters. A possible setting for this function is shown by the red curve in Figure 7. This nonlinear curve is added to provide greater control over the interpolation of the morph index for the note sequence morphing algorithm, as it is a more important morph index than the others.

4.5 Note sequence morphing algorithm infrastructure

Important elements of the supporting infrastructure used by the note sequence morphing algorithms is detailed below, including tonal representations, rhythmic representations and extensible software designs. The tonal representation includes key, octave, scale, scale degree and passing note pitches. The rhythmic representation includes quantization, shuffle and loop length. The extensible design includes classes that can be extended for new morphing algorithms and non-standard tunings. These aspects constitute the supporting infrastructure for note sequence morphing algorithms, whereas the note sequence morphing algorithms themselves are discussed later in chapters five to seven.

4.5.1 Tonal Representation

As mentioned earlier, each layer within a state can be labelled as belonging to a particular key and scale; and a particular layer within the morph can be tagged as being the tonal leader. The programmer of the note morphing algorithm has the option of utilising this tonal meta-data and morphing it through a separate process to the note sequence data[3]. When operating in this way, the note pitch needs to be converted into scale degree. Creating the final MIDI pitch output will require the scale degree, octave[4], scale (set of tonal pitch-classes) and key. This fairly common approach is made clear in Figure 8 below.

Key = C#	12 keys: 0, ***1***, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11		1
Octave = 6th	12 steps/octave: 0, 12, 24, 36, 48, ***60***, 72, …	+	60
Degree = 3rd	Major scale: 0, 2, ***4***, 5, 7, 9, 11	+	4
	MIDI pitch	=	65

Figure 8 Representing MIDI pitch 65 (E#) as the 3rd of C# Major.

In this diagram, we can see how the MIDI pitch 65, or E#, can be represented by a combination of four musical components: the major scale, the third degree, the key of C# and the 5th octave.

This representation works well in theory; however, a surprising amount of tonal music utilises passing note pitches to great effect, either as part of a melodic contour or as discord. One method to overcome this could be to include information on whether the scale degree is "raised"

[3] The *TraSe* algorithm of chapter seven is currently the only algorithm that morphs the key/scale meta-data.

[4] The current implementation includes octave information within the scale degree, however they are kept separate here for simplicity.

or “lowered”. This would require the algorithm that converts MIDI pitch to the tonal representation to judge whether or not the passing note is a raised version of the lower scale degree, or a lowered version of the higher scale degree. However, a passing note is often perceived as neither – either as part of a run, or an “in-between” note.

It therefore would be more appropriate for the passing note to be represented as an “in-between” note, rather than a raised or lowered version of a scale degree. This way, if a judgement needs to be made concerning which pitch the passing note should resolve to in the case of some transformation, it can be made by the algorithm that renders the MIDI pitch from the tonal representation in realtime – enabling a greater, more up-to-date contextual breadth to inform the decision.

In order to achieve this, the *Degree-Passing note (DePa)* representation was created. Within this scheme, additional “passing note” scale degrees are added between each of the original scale degrees, which means that, in the typical case, every second degree is a passing note. Whether or not the passing note exists as a MIDI pitch will depend on the scale used.

To ensure that every possible passing note can be represented, the total number of *DePa* degrees, including the passing notes, is equal to the largest semitone interval between consecutive scale degree pitches multiplied by the length of the scales being used (all scales must be the same length). In the standard case of major and minor, the scale lengths are all 7, and the largest interval between consecutive tonal notes is 2 semi-tones. This means that in order to represent the passing notes, 14 degrees are needed.

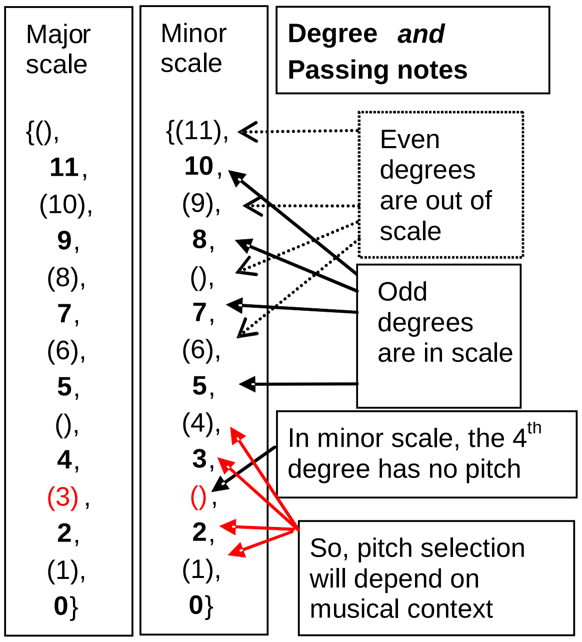

Figure 9 ***DePa* enables accurate representation of passing notes. Counting from one, scale degrees that are odd are in the scale, while those that are even are passing note pitches. Starting with a passing note in the major scale, between the Major 2nd and the Major 3rd, it will be represented as the 4th degree of a diatonic DePa scale. In a minor scale, there is no passing note between the Major 2nd and Minor 3rd, leaving four options available to interpretation: keep the note as a passing note and raise or lower it to the closest existing passing note; or turn the passing note into a scale degree which is either higher or lower (in this case, "higher" has the equivalent MIDI pitch).**

To further illustrate *DePa*, consider a major scale where the Minor 3^{rd} is used in passing (Figure 9) and thus recorded as the 4^{th} *DePa* degree. If the scale is changed from major to minor, the minor third will no longer exist as a passing note, leaving the algorithm that renders the final MIDI pitch with a number of options. It could preserve the accidental nature of the note, an attribute especially significant in chords, by playing either of the nearest passing note pitch classes 1 and 4. Or, if the note is part of a melodic contour, it may be best to choose from pitch classes 2 or 3 in order to match the contour as close as possible. These options are able to be manually specified in a *DePa* to MIDI pitch conversion function that is part of *LEMorpheus*.

Having the ability to represent passing note pitches within the tonal representation has worked well, however, *DePa* is somewhat inflexible when shifting to scales of different length and with larger tonal steps, such as the augmented tones of harmonic minor. Future research will involve a floating point scale degree representation. This will allow, for example, the harmonic minor passing note pitch-classes 2 and 3 to be represented as degree 1.33 and 1.66 respectively. This is appealing due to it being in the same numerical range as the original scale degree and presents advantages to flexibility, yet to be investigated.

4.5.2 Rhythmic representation

For representation of rhythm, each note contains an onset and duration, as with most other note representations. In addition, some rhythmic transformations are included that control looping, quantization and shuffle. These are part of a chain of transformational functions that are activated when the note information of the current time slice is requested by the MIDI player thread (see 4.6). *Loop* crops the data to within the bar-length and the loop length parameter is also referred to outside the chain by the note sequence morphing algorithm. *Quantize* snaps the onset of notes into a grid as well as converting the note sequence into streamed segments. S*huffle* delays notes that are on the offbeat to produce a swing feel.

Using a transformation chain for representation of rhythm affords realtime functions that can act on streamed note events. Despite this, transformation functions that deal with the entire note data of the layer can be included, but must be ordered prior to other transformations that reduce the length. For example, *loop* reduces the length of the sequence to the loop length, before *quantize* reduces it to the resolution of the play cycle (0.25 of a beat) as part of streaming. Rhythmic representations that incorporate more descriptive metres will need to be developed in the future to enable more sophisticated morphs.

4.5.3 Extensible design

The libraries used within *LEMorpheus* were designed for easy customisation to unique styles of music, music transformations and morphing techniques. This includes classes that deal with non-standard tuning systems and scales, as well as classes that support the creation of note sequence morphing algorithms.

The key and scale of each layer is stored within a *Tonal Manager* object, which also has a number of different functions for converting between MIDI pitch and the scale degree representations mentioned above. This includes a *Tonal Composite* object which generates scales and pitch probabilities from note sequences as well as a *Scales* object which stores the names of scales, arrays of pitch-classes representing them and tracks the current scale being used. To enable a deeper level of musical flexibility, a *Tuning System* object is used to store the frequency ratios of each pitch-class in an array. Because synthesis occurs externally through

MIDI, the only data currently utilized from *Tuning System* is the number of steps per octave (12 by default).

New note sequence morphing algorithms can be extended from the *Morpher* class. The logic for note sequence morphing is implemented in a function within the extended class that is called continuously in realtime. Each cycle, it is passed: the two note sequences from source and target that are being morphed, the morph index, the current beat position and a FIFO (First In First Out) queue of notes that have been sent as output from the layer since the morph started. This information can be used in any way by the note sequence morphing algorithm to produce the morphed note sequence of the current time-slice, which is sent as an array of two. The array of two enables notes derived from the source to be kept separate from notes derived from the target, if required. The queue was included for morphing algorithms that are influenced by the recent history and is also used for the "repeat" function (4.4.2).

4.6 Rendering MIDI Output

A call-back routine has been implemented to generate the stream of MIDI events, as shown in Figure 10.

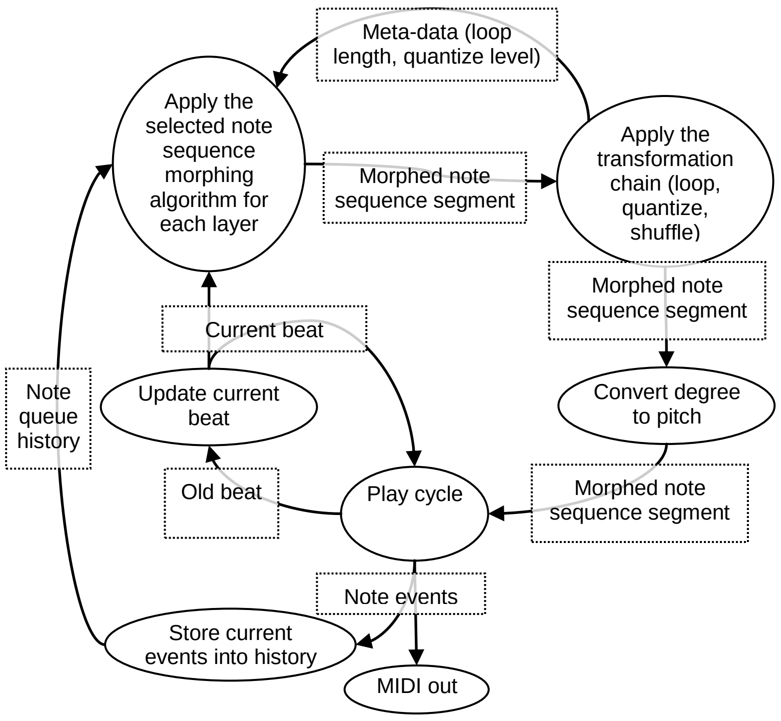
Meta-data (loop length, quantize level)
Apply the selected note sequence morphing algorithm for each layer
Morphed note sequence segment
Apply the transformation chain (loop, quantize, shuffle)
Morphed note sequence segment
Current beat
Note queue history
Update current beat
Convert degree to pitch
Old beat
Play cycle
Morphed note sequence segment
Note events
Store current events into history
MIDI out

Figure 10 Overview of system for rendering MIDI data.

The interval between **play cycles** is generally a quarter beat but is automatically changed to an eighth or half beat depending on the tempo. During a morph, the note sequence morphing algorithms that have been configured for each layer are called every cycle and sent the current time. The meta-data parameters from the transformation chain (loop length, quantize resolution and shuffle delay amount) for each layer from source and target that are being interpolated will be updated with the transformed morph indices (see 4.4.2).

Realtime note sequence morphing algorithms will refer to the morphed meta-data parameters. Non-realtime note sequence morphing algorithms will apply the transformation chain with the morphed meta-data parameters to the morphed note sequence. In both cases, a segment that is the length of the current timing resolution (usually 0.25 beats) results. Despite this, notes are allowed to have their onsets at any point (float value) within the segment and are allowed durations of any length.

Following this, the key and scale changes required by tonal leading/following between parts are applied (if used) and the scale degree, key, scale and octave information is combined to determine the MIDI pitch (see 4.5.1). The notes events that span each **play cycle** interval are converted to a sequence of Midishare events and sent to a synthesizer.

4.7 Summary or morphing infrastructure

The *LEMorpheus* software system takes source and target note sequences and applies a user-selected note sequence morphing algorithm to create the morphed material. Meta parameters such as quantize and swing are interpolated and these influence the morphed note sequence data. *LEMorpheus* includes the capacity to create and arrange note sequence loops and morphs, as well as control over the morph index. There are also parameters that influence the morphing process, such as the morph length and a number of transformations that affect the morph index before it is passed to particular components.

The software infrastructure for note sequence morphing includes a novel tonal representation that retains passing note information while using scales and scale degrees, as well as a basic representation of rhythm. The design is extensible, suiting a variety of approaches to morphing and could be easily adapted to unusual tuning systems and scales. Finally, *LEMorpheus* renders MIDI output by calling a play cycle every quarter of a beat. During the play cycle the current beat is updated, morphed note events are generated, transformations are applied, scale degree representation is converted to MIDI pitch representation and note events are sent out to the synthesiser.

Having been developed in response to mainstream electronic music requirements, the system has been limited to music based on loops and layers and deals only with source-to-target morphing. The primary strengths of the *LEMorpheus* infrastructure appears to be its ability to easily modify or create transformations and morphing algorithms due to extensible design and supporting functions. Many parameters are configurable at the user-level and the programming specifications of morphing algorithms encourage a similar style of user configurability. The weaknesses are poor usability and learnability; however, as a piece of research software, Human Computer Interaction (HCI) is not the primary consideration.

Future experimentation for the *LEMorpheus* system architecture will involve floating point scale degrees for the *DePa* representation, more sophisticated representations of metre and rhythm, the capacity for n-source morphing and a two dimensional Cartesian interface for n-sources. The floating point scale degrees will be able to represent passing notes more accurately across a variety of different scales. Metrical representations will enable more rhythmically coherent morphs, in the same way that key/scale representations have increased tonal coherence. Just as *DePa* attaches tonal significance to passing notes, the rhythmic representation should flexibly deal with unusual and offbeat rhythmic patterns. Finally, it is inevitable that n-source morphing algorithms be implemented, as this will clearly add to the capabilities of the system. The two dimensional interface could be modified for more than two sources, as per *The Meta-surface* (Bencina 2005) or Oppenheim's four-source *Mutant Ninja Tennis Court* (Oppenheim 1995).