Dokumentation zu dem Stück

Rrumpff tillff toooo? (2010)

eine elektronische Adaption

der *Sonate in Urlauten (1932)* von Kurt Schwitters

für 4-Kanal-Anlage

Dauer: 11 min

Hendrik Dingler
Eschollbrücker Straße 5
64283 Darmstadt
Tel.: 0170 495 1909
www.hendrikdingler.de
mail: hd@hendrikdingler.de

Vorwort

Innerhalb dieser Dokumentation wird zum Einen das Stück *Rrumpff tillff toooo ?* beschrieben und zum Anderen das hierfür erstellte Framework für Sprachtransformations-Prozesse. Die Prozesse bedingen die Form und den Verlauf und können daher größtenteils nicht getrennt vom Stück betrachtet werden.
Innerhalb des Textes befinden sich Verweise auf den Anhang, welcher separat als digitale pdf-Datei beiliegt und sich aus dem Quelltext der Software zusammenstellt.

Inhaltsverzeichnis

0. Einleitung

Das Stück beschäftigt sich mit kontinuierlicher und sukzessiver Abstraktion von Sprachlauten. Es ist die logische Konsequenz auf meine, während des Studiums der elektronischen Komposition, angefertigte Sample-Studie *Suite No.2 Op.64c & ein Zwerg der linkisch auf missgestalteten Beinen herum hüpft*.
In dieser Studie wurde das Scratching (Scratch-Turntablism) auf die digitale Ebene übertragen und kann hier als eine Form von Granularsynthese betrachtet werden. In *Rrumpff tillff toooo ?* geht es um die Weiterführung und Vertiefung dieser granularen Synthese-Ansätze, wobei weitere Prozesse hinzukommen.
Das Material, welches hier verwendet wurde, stammt aus der *Sonate in Urlauten* von Kurt Schwitters (1887-1948):

Kurt Schwitters, „Sonate in Urlauten“ aus: ders., Das literarische Werk, hrsg. Von Friedhelm Lach

Konzept

Gegeben sind zwei Themen der *Sonate in Urlauten*, deren einzelne Sprachfragmente im Verlauf des Stückes in ihre Bestandteile zerlegt und neu zusammengesetzt werden, um sich weit von ihrem Ursprung zu entfernen. Dadurch sollen neue Klänge generiert werden, welche im eigentlichen Sinne nichts mehr mit Sprache zu tun haben, jedoch immer noch auf diese verweisen.
Die Form des Stücks kann, mit einem Abstand zu klassischen Modulationsverfahren und im Sinne elektoakustischer Mittel, verglichen werden mit dem Aufbau einer Sonatenhauptsatzform. Nach der Exposition der Themen (in diesem Fall zwei dadaistische Phrasen) folgt die „Durchführung“, welche hier die abschnittsweise Zerlegung, Veränderung und somit Entfremdung dieser Themen vorsieht. Anstatt einer „Reprise“ und einer „Coda“ am Ende, folgt eine Rückführung zum Ausgangsmaterial, mit Rückblick auf das Geschehene. Somit schlägt der Verlauf des Stückes einen Bogen zurück zum Anfang.
Als Werkzeuge zur Umsetzung der Abstraktion von Sprachlauten dienen hauptsächlich granulare Syntheseverfahren, sowie die Kombination dieser mit den Funktionen eines Phase Vocoders und der Convolution Synthese.

Hintergrund

Die Verwendung des dadaistischen Sprachmaterial beruht auf der Tatsache, dass bereits die Dadaisten eine frühe Form der heutigen granularen Synthese verwendeten. Sie zerlegten die menschliche Sprache in ihre einzelnen Fragmente (die Buchstaben), um sie anschließend wieder neu zusammen zu setzten.
Auf diese Weise betrachtet, wird mit dem Stück *Rrumpff tillff toooo?,* zusammen mit den heutigen elektronischen Möglichkeiten, dieser Prozess fortgesetzt.

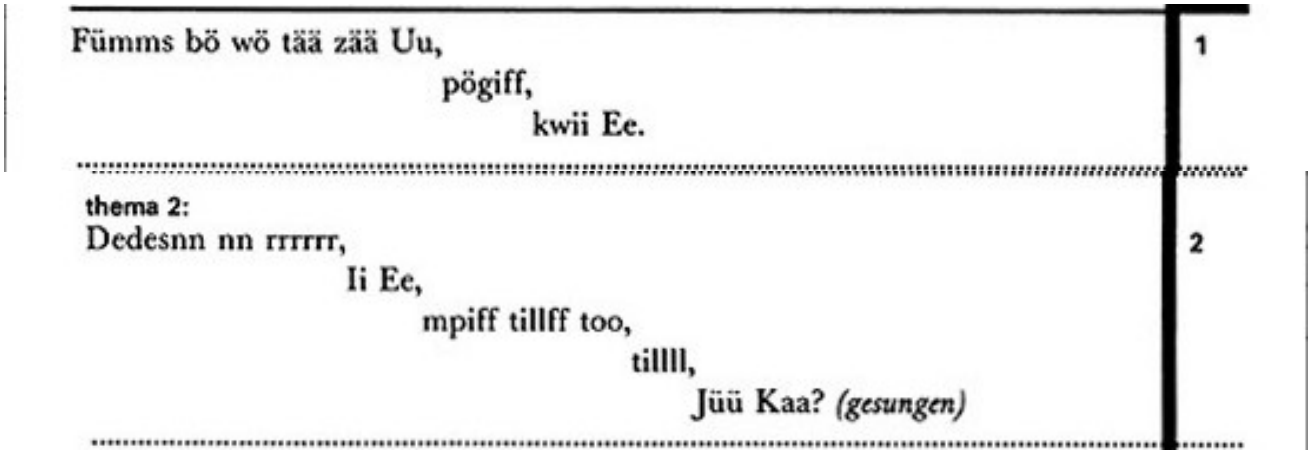
Die strenge Form des Stückes widerspricht jedoch den dadaistischen Prinzipien, welche aus dem Dadaistischen Manifest (Richard Huelsenbeck, 1918) und den Aufzeichnungen von Hugo Ball (1886-1927) hervor gehen. Hier stehen die Loslösung von Normen, Grenzen oder gegebenen Bewegungsradien im Vordergrund.
Kurt Schwitters dagegen widersetzt sich diesen Prinzipien durch die Erstellung eines dadaistischen Lautgedichts in Form einer klassischen Sonate und kombiniert somit zwei unterschiedliche Vorgehensweisen.

Seine Schriften haben mich bei meiner Arbeit an „Rrumpff tillff toooo?“ sehr inspiriert. In seinem Text „Merz“, in „der Sturm“ XVIII, 3 (1927), S. 43 schreibt er:

> "....In jedem Stadium vor der Vollendung ist das Werk für den Künstler nur Material für die nächste Stufe der Gestaltung. Nie ist ein bestimmtes Ziel erstrebt außer der Konsequenz des Gestaltens an sich. Das Material ist bestimmt, hat Gesetze, hat Vorschriften für den Künstler, das Ziel nicht..."

1. Ausgangsmaterial

Für die hauptsächliche Umsetzung des Stücks wurden 2 Themen der Sonate in Urlauten verwendet: (Auszüge aus der originalen Partitur)



Neben diesen Themen welche im Weiteren als Thema 1 (TH1, **Fümms bö wö ..**) und Thema 2 (TH2, **Dedesnn nn rrrrr ...**) bezeichnen werden, wurden weitere Sprachfragmente verwendet um den Material-Pool für bestimmte Prozesse zu erweitern (s. Klangsynthese – Sprachtransformations-framework → Kapitel 2.7.8.1. Funktionen des Fragment-Pools: Alternativen, S. 36)

Das Material wurde auf der rechtlichen Basis einer, vom DuMont Verlag erteilten, Genehmigung zur Vertonung von mir selbst eingesprochen.

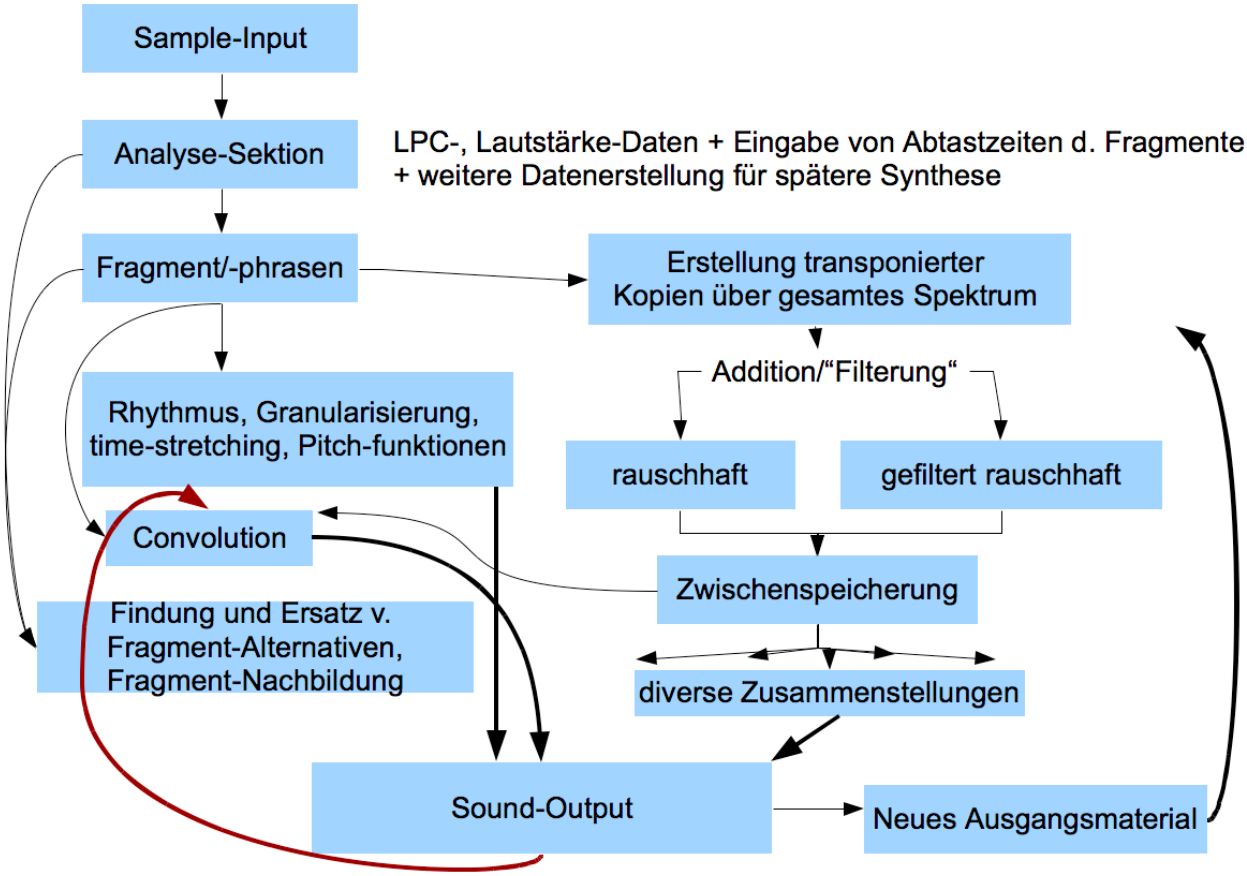
2. Klangsynthese - Sprachtransformationsframework

Für die Bearbeitung der einzelnen Sprachfragmente der Themen wurde eine Framework zur Sprachtransformation auf Basis der Programmiersprache Scheme (eine Variante der Programmiersprache LISP) und der Klangsynthesesoftware Csound entwickelt. Des Weiteren wurden Funktionen der Audioanalyse Software Praat und des Kommandozeilen-Tools SoX (Sound eXchange) verwendet.

Bei den verwendeten Programmen setze ich die Kenntnis von deren Funktionen voraus. Ist dies nicht der Fall, verweise ich an dieser Stelle auf die jeweiligen Manuals.

Das Framework ist in seiner Form als ein Ordner- und Dateisystem aufgebaut.
(s. Sprachtransformation_Fw/ → Anfragen über hd@hendrikdingler.de)

Die grobe Struktur sieht wie folgt aus:



2.1. Fragmente und Events

Das Transformationsframework dient der Bearbeitung von Sprachfragmenten und Soundevents, welche aus Wertefolgen bestehen, die durch exakte Parameter definiert werden.
Ein Sample mit einer Sprachfragment-Phrase, wie beispielsweise die des TH1 („Fümms bö wö tä zä uu pö gif kwi ee“) wird zu Beginn in einzelne Fragmente unterteilt.

(Anmerkung: *Kursiv* gedruckte Ausdrücke verweisen auf den Quelltext des Programmcodes, welcher im Anhang zu finden ist)

Ein Fragment hat die Parameter:

- Samplenummer (Ausgangsmaterial; *orig-samplenr*)
- Fragment-Position innerhalb der Fragmentphrase (*orig-pos*)
- Zielsamplenummer (*ziel-samplnr*)
- Zielposition (*ziel-pos*)
- Liste mit Abtastzeitpunkten
  - ZA (Abtast-Start, Start des Fragments innerhalb des Samples)
  - ZP (Abtast-Peak, Peakposition (rhythmische Spitze des Fragments)
  - ZE (Abtast-Ende, Ende des Fragments innerhalb des Samples)

Die Definitionen der *ziel-samplenr* und *-pos* werden innerhalb der Ersatzfunktion innerhalb des Rückführungsteils des Stückes verwendet um alternativen Samples ihre Zielpositionen zu geben. (s. Kapitel 2.7.8. Prozess Fragment-Alternativen-Setzung, S.36). Bis dahin sind diese Parameter ohne Funktion und tragen die selben Werte wie die *orig-samplnr* und *-pos*

Anhand der Parameter eines Fragment lassen sich folgende Daten ermitteln:

- Fragmentdauer (*f-dur*), sowie die zeitlichen Abstände Abtast-Start bis -Peak (*f-abtaststp*) und Abtast-Peak bis -Ende (*f-abtastp-end*)

- Funktionstablle (ftable) des Samples für den Aufruf in Csound (*f-ft*)

→ Siehe Anhang S. 33

Ein Event ist die genaue Definition für die Umsetzung eines Aufrufes in Csound. Es besteht aus den folgenden Parametern:

- Abspielzeitpunkt (*entry*)
- Abspieldauer (*dur*)
- Funktionstabelle (*ft*)
- Abtastpostition innerhalb des *fts* (*pos*)
- Funktionstabelle der Hüllkurvenfunktion (*huell-ft*)
- Lautstärke in dB (*amp*)
- Abspielgeschwindigkeit/Pitchlevel in cpsoct (*pitch*; 8.0 = unverändert)

→ Siehe Anhang S. 43

Im Weiteren wird sowohl mit Fragmenten als auch mit Events umgegangen. Wobei ein Fragment leicht in ein Event gewandelt werden kann durch die Funktion *frag->event*.
So wird zum Beispiel das Fragment 1 des TH1 (Fümms) = '(1 0 1 0 (0 0.065 0.412)) bei einer Startzeit von 0.0 zum Event : '(0.0 0.412 2 0 39 90 8.0)

→ Siehe Anhang S. 47

2.2. Aufruf des Quelltextes

Folgendes geschieht beim Laden der Datei „load.scm“:

- Integration der Zusatzmodule (ice-9 format), (ice-9 popen) und (ice-9 rdelim)
- Definition des Hauptpfades und aller weiteren Pfade, sowie des Namens und Ortes des *huellkurven-ftable-files*, durch laden der Datei „pfade.scm“
- Definition des Befehls *csound-render* und *praat*
- Definition der Zahl „pi“ (mit 14 Stellen hinter dem Komma)
- Laden der Stream-Implementierung und weiterer Basisfunktionen
- Definition des Namens der ftable-Datei, in das die ftables der Samples während der der Analyse geschrieben werden
- Laden des Scheme-Quelltextes und Aufruf der automatischen Analyse-Funktionen
- Definition des„*huellkurven-fts-streams* (dieser wurde bis dahin erzeugt)

Für die Arbeit an dem Stück *Rrumpff tillff toooo?* wurden anschliessend noch die Fragment-Phrasen TH1 und TH2, sowie alle einzelnen Fragmente dieser 2 Themen definiert als *TH1-f1 – TH1-f10* und *TH2-f1 – TH2f14*.

→ Siehe Anhang S. 2/3

2.3. Hüllkurven

Die Csound Hüllkurven-Funktionstabellen wurden vorab manuell erstellt und im Ordner „Zugriffsdaten/csound/ftables/“ abgelegt.

Es gibt 39 Hüllkurven-Funktionen die zur Auswahl stehen:

1. Dreieck (Bartlett Fenster)
2. Trapez
3. expodec (exponnetial decay)
4. rexpodec (reversed Expodec)
5. Gauß-Funktion
6. Blackman-Harris-Funktion
7. Hanning-Funktion
8. Positive Sinushalbwelle

9.-39. Interpolationen zwischen der Sinushalbwelle und der positiven Halbwelle einer Rechteck-Funktion (= keine Hüllkurve)

→ Siehe Anhang S. 116/117

2.4. Csound Orchestras

Es gibt eine zentrale Orchestra-Datei (Orchestra), mit dem letzten Endes fast alle Klänge errechnet werden. Dies ist die Datei *complete.orc*. Dieses Orchestra beinhaltet das Instrument 1, welches ein simpler Sampleplayer ist, der einzelne Abschnitte aus einer Sample-Funktionstabelle ausliest, mit einer gegeben Amplitude und einer Hüllkurven-Funktion belegt und ausgibt. Weiterhin erhält das Instrument einen Transpositionswert (in cpsoct), welcher die Abspielgeschwindigkeit bestimmt.

→ Siehe Anhang S. 115

Für die Erstellung von Fragmentnachbildungen wird das Orchestra *complete-mit-globalhuell.orc* verwendet, welche als Erweiterung zum genannten Instrument 1 eine globale Hüllkurven-Funktion hat (Instrument 10). Diese belegt alle Aufrufe des Instruments 1 in ihrer Gesamtheit mit einem gegebenen Amplitudenverlauf. Hierzu Genaueres in Kapitel 2.7.7.Fragment-Nachbildung, S. 34.

2.5. Sound-input und-output

In den *sound-input* Ordner (Sound-input) kommen vor dem Öffnen und Laden des Quelltextes die benötigten Samples. Wichtig ist, dass die Samples in den Formaten .wav oder .aiff, mit einer Samplerate von 48 kHz und einkanalig, d. h. als Mono-Audiodatei vorliegen.

In den *sound-output* erstellt die Funktion *csound-render* die Ausgabe-Audiodateien.

2.6. Analysen und weitere vorbereitende Funktionen

Sind neue Samples im Sound-Input werden diese wie folgt bearbeitet und analysiert.

Die Funktion *do-analysis* beinhaltet die ersten Bearbeitungs- und Analyse-Funktionen und erstellt die Analysedaten-files, die notwendig sind um die Erstimplementierung der Fragmente zu gewährleisten. Diese sind neben der Umwandlung der Samples in Referenz-Samples die Analyse-Funktionen *make-csoundlpanal-daten* und *make-praatintens-daten*.

→ Siehe Anhang S. 17

new-samples-preparation

Die Prozedur *new-samples-preparation* schaut nach, ob neue samples im Sound-input sind und wandelt diese in Referenz-Samples (Ref-Sample = sampleXXX-'name_endung', XXX = Ref-Samplenummer)

Beispiel: Neues (1.) Sample = thema3.wav → (Ref-Sample=) sample000-thema3.wav

→ Siehe Anhang S. 15 und S. 28

make-csoundlpanal-daten

Diese Analyse-Funktion erzeugt 'samplename'-lpc.daten -Dateien im lpc-daten Ordner. Sollte die Datei bereits existieren, wird die Analyse übersprungen.

Für die Analyse wird die Analysefunktion LPANAL von CSound verwendet. Die Einstellungen hierfür sind:

- SR 48000 kHz
- channel number (c) 1
- lowest frequency (P) 60 Hz

LPANAL wird aufgerufen und erzeugt eine Analysedatei mit dem Namen des Ref-Samples und der Dateiendung .lpc im lpc-daten ordner.
Die Unterfunktion *make-lpc-sco* erstellt eine Score-Datei (Score) im Scores-Ordner. Diese enthält eine Zeile für den Aufruf des Textdatei-Erstellungs-"Instruments" des *lpc-print-data.orc* mit den Angaben der Dauer des Ref-Samples und dem Pfad und der Bezeichnung der Analysedatei (.lpc) als String.
Bei dem zweiten Aufruf von CSound wird die erstellte Score mit dem *lpc-print-data.orc* aufgerufen um eine les- und auswertbare Daten-Textdatei zu erzeugen, ebenfalls im lpc-daten ordner.
Die resultierende Datei hat den Namen des Ref-Samples mit dem Anhang *-lpc* und der Endung *.daten*. Darin befinden sich einzelnen Zeiten mit einem Zeitpunkt und den dazugehörigen Daten. Diese sind die Grundtonhöhe (*pitch*) und der normalisierte Rausch- bzw. Erroranteil (*error*) des Signals. Das zeitliche Interval zwischen den Werten liegt bei 0.2 ms (krate des Orchestra).
Die während des Prozesses erzeugen Nebendateien (*muell.aiff, lpc-file, sco-file*) werden nach Abschluss gelöscht.

→ Siehe Anhang S. 16 und S. 114

make-praatintens-daten

Diese Analyse-Funktion erzeugt 'samplename'-intens.daten -Dateien im intensity-daten Ordner. Sollte die Datei bereits existieren, wird die Analyse übersprungen.
Das Programm PRAAT wird mit einem Intensitäts-script (*intensity.praat*) und dem Ort und Dateinamen des *Ref-Samples* als String aufgerufen. Der Output wird über die Shell (Unix) in eine les- und auswertbaren Textdatei weitergeleitet (>) (Dateiendung: -*intens.daten*, Ort: *intensity-daten-pfad*, Inhalt: (pro zeile) Zeitwert und dB-Wert)

In PRAAT passiert im Hintergrund folgendes:

- Öffnen des Script (intensitäts-script)
- Befehl "Read from file..." Sample
- Befehl "To Intensity..." 75 (min pitch) 0.0002083 (time-step) [wie bei lpc] yes

Im Script wird definiert :

- start = Get start time
- end = Get end time
- totaldur = Get total duration
- frames = Get number of frames

Loopfunktion des scripts:

```
for i to frames
    time = Get time from frame number... i
    val = Get value in frame... i
    print 'time' 'tab$' 'val' 'newline$'
endfor
```

→ Siehe Anhang S. 16/17

2.6.1. Fragment-Zeiten

Nachdem die Analyse-Funktionen stattgefunden haben muss ein manueller Schritt gegangen werden um die Abtastzeiten der Fragmente zu definieren.
Diese werden erstellt durch das öffnen der Ref-Samples in Praat und das manuelle Durchgehen dieser zur Angabe der Anfangs-, Peak- und Endzeiten der einzelnen Fragmente.

Hierzu muss wie folgt vorgegangen werden:

In Praat → Read → Read from file.. → Auswahl des Ref-Samples → Annotate – To Textgrid → All tiers name: „zaze zp“, Which of these are point tiers?: „zp“ (OK) → Auswahl des Samples und des erstellten Textgrids (in Praat) → Edit → Setzung der Anfangs- und Endpunkte für die Abschnitte „ZaZe“ (des Fragments) im *zaze-tier* und Setzung der Amplituden-Peaks „Zp“ (des Fragments) im *zp-tier*, jeweils durch klicken mit der mouse.

Ist dies geschehen kann das Editierfenster wiedre geschlossen werden und das „*textgrid-list.praat* -script von Praat (s. Daten-DVD → /Rrumpff_tillff_toooo/Sprachtransformation_Fw/source_code/01_analyse/textgrid-list.praat) durchlaufen werden (Praat → Open praat script) um die Zeiten als text-Datei zu exportieren (copy/paste in Textfile). Die so erstellten Daten-Textfiles müssen im Ordner *zazpze-daten* mit der jeweiligen Benennung reference-Samplename(ohne Endung)-*zazezp.daten*.

Das Script beinhaltet die folgenden Zeilen:

```
intervals = Get number of intervals... 1
clearinfo
for i to intervals
    starttime = Get start point... 1 i
    starttime = (round (starttime * 1000)) / 1000
    endtime = Get end point... 1 i
    endtime = (round (endtime * 1000)) / 1000
    peakpointindex = Get high index from time... 2 starttime
    peaktime = Get time of point... 2 peakpointindex
    peaktime = (round (peaktime * 1000)) / 1000
    print 'starttime' 'tab$' 'peaktime' 'tab$' 'endtime' 'tab$'  'newline$'
endfor
```

Ausgegeben pro Zeile werden die Abtaststartzeit, -peakzeit und -endzeit, gerundet auf 3 Stellen hinter dem Komma.

2.6.2. Funktionstabellen

Nachdem die Analysen abgeschlossen sind werden die ftables der *Ref-Samples* und die der Hüllkurven für die späteren Aufrufe in Csound generiert. Hierfür ist die Funktion *make-ftables-as-file* zuständig

Die Funktion *make-ft-line* erzeugt hierbei die benötige Zeile und *add-huellkurven* hängt die h*uellkurven-fts* am Ende an.

→ Siehe Anhang S.21 und S. 29

Die Funktion *add-huellkurven* bedient sich des vorher definierten Hüllkurven-Streams und zeigt die Ft-Funktionen mit neuen ft-Nummern an.

→ Siehe Anhang S. 23

2.6.3. Fragmenthüllkurve als Ascii-Textfiles

Später im Text (bei Fragment-Nachbildung) wird die Nachbildung von Fragmenten des TH1 erklärt. Hierzu ist notwendig die Lautstärkeverläufe der einzelnen Fragmente in Form von ftables zu generieren. Es werden Textdateien mit Amplitudenwerten erzeugt.

Diese Textdateien müssen so erstellt werden, dass sie mit der GEN-Routine 23 in Csound als ftable-Funktion geöffnet werden können. Das bedeutet, dass die Anzahl der zu erstellenden Werte im Textfile eine Höhe von 2^n erreichen muss.
Die Funktion *make-amp-ascii-file* erstellt hierbei für das gegebenes Fragment ein Stream aus dessen Intensitätswerten und gibt diesen als Textfile aus. Die Daten werden abgelegt im Ordner „ZugriffsdatenAnalysedaten/frag-huell-asciis/“.
Um die Länge 2^n zu erreichen wird der Intensitäts-Stream eines Fragment, welcher aus den Intensitätsdaten gezogen wird, aufgefüllt mit Nullen und die einzelnen Lautstärken werden dem-entsprechend angepasst, sodass eine klare Hüllkurven Funktion entstehen kann.

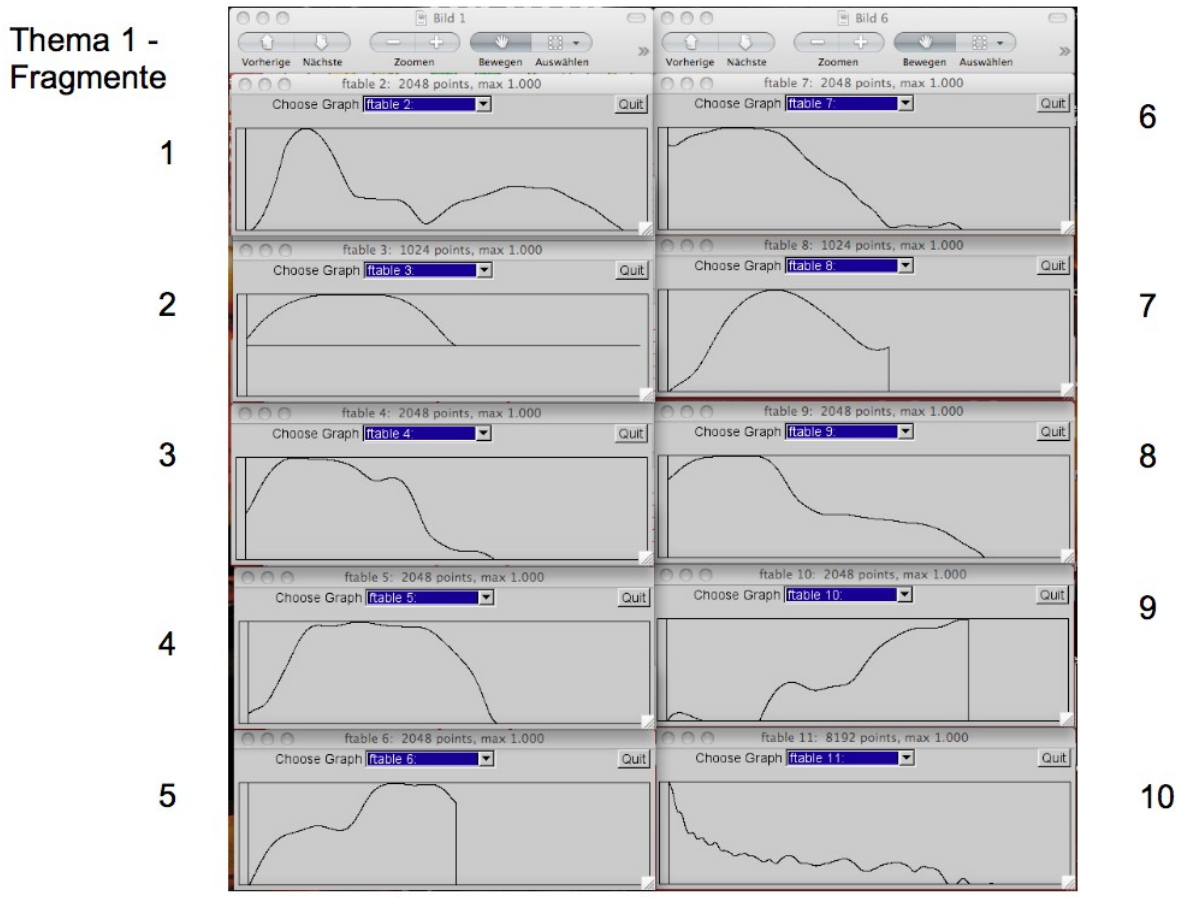
Wichtig hierbei ist, dass im Namen des Textfiles zum einen die 2^n-Länge gespeichert ist und zum andern das Verhältnis der Länge des Fragments zu dieser 2^n-Länge (*ratio*).

Der Name erstellt sich wie folgt:

(Samplenr des Fragments)-(Fragment-Position innerhalb des Ref-Samples)-
(2^n-Länge des Textfiles)-(*ratio* Fragment-Länge/2^n-Länge in „milli“-Darstellung)

Beispiel: Sample1, Fragment1 ‚2^n = 2048, *ratio* = 0.966 → „000-000-02048-966.txt“
die Ref-Samplenummer und die Positionsnummer sind jeweils Indexzahlen.

→ Siehe Anhang S. 88



Das entsprechende ftable-file erstellt die Funktion *make-ftables-with-ascii-huells*. Sie erstellt die ftables der Ref-Samples, eine sinussoide Hüllkurve und die GEN-Routine 23-ftable-Zeilen für die Erstellung der Fragment-Intensitätsverläufe.

→ Siehe Anhang S. 21

2.6.4. Pitches, Errors, Length and Attacks

Nun werden die Analysedaten ausgewertet und es werden weitere Datenfiles erzeugt:

Die Funktion *make-straight-pitches-as-file* erstellt eine nach Pitch-werten (aus der LPC-Analyse) sortierte *.daten*-Textdatei im Pfad „Zugriffsdaten/Analysedaten/Error-Pitch-Daten/“ mit dem Namen *straight-pitches.daten*. Diese beinhaltet Daten für Abschnitte innerhalb der *Ref-Samples* mit einigermaßen gleichbleibender Tonhöhe
Pro Zeile stehen hier der Wert des gleichbleibenden Pitches, die Ref-Samplenummer und die Abtaststart- und -endzeiten.

Das gleiche passiert mit den Error-werten der LPC-Daten durch die Funktion *make-straight-errors-as-file* deren Ausgabe in die Textdatei *straight-errors.daten.*

Beide Funktionen verwenden die abstrahierte Funktion *make-straight-data-as-file* mit unterschiedlichen Prozeduren.

→ Siehe Anhang S. 17/18 und S. 26/27

Im Anschluss wird eine Daten-Textdatei mit den Werten der Attackzeit, der Dauer und den gemittelten Pitch- und Error-Werten für jedes Fragment aller Ref-Samples erzeugt.
Die Fragmente werden hierbei durch die Funktion *make-all-frags-stream* generiert, welche die Stream-Erstellungsfunktion *make-frag-stream* auf jedes Ref-Sample anwendet und die Ergebnisse zusammenhängt.
Im Loop werden die Daten Attack, Length und die gemittelten Pitch- und Error-Werte erzeugt. Letztere durch das auslesen der vorher generierten *straight-pitches.daten*- und *straight-error.daten*-Dateien.

→ Siehe Anhang S. 18 und S. 31

Zum Abschluss der Analyse wird eine Scheme-Datei mit den Definitionen der Fragment-Phrasen (Fragmente eines Ref-Samples in ihrer genauen Reihenfolge) erstellt und eingelesen.

→ Siehe Anhang S. 20

2.7. Transformationskategorien

Die Sprachtranformationskategorien beinhalten die folgenden Syntheseverfahren und Bearbeitungsprozesse, die im Folgenden erklärt werden:

- Rhythmische Bearbeitung der Sprachfragmente
- Granulares Timestretching (g*rob*, *fein* und *sinus*)
- Erstellung von rauschhaften Kopien der Fragmente durch die Verwendung des Phasevocoders und additiver Synthese (*pink, white, notches* (gefiltert), *bandpass* (gefiltert))
- Granulare Synthese mit den rauschhaften Kopien, sowie mit dem Ausgangsmaterial
- Fragment-Nachbildung durch Verwendung des Ausgangsmaterials
- Kombination Pitchshifting und Additive Synthese (so genannte „Pitchhits“)
- Convolution
- Erzeugung und Bearbeitung neu erstellter Klänge als neues Ausgangsmaterial

2.7.1 Rhythmische Bearbeitung der Fragment-Phrasen

Für die rhythmischen Variationen des TH2 in der Exposition des Stückes wurden folgende Funktionen erstellt:

Rhythm-from-orig holt aus einem gegebenen Fragment-Stream die einzelnen Abtast-Peak-Zeiten heraus und liefert sie in einen Stream.

*Rhythm-stretch s*taucht oder dehnt, dynamisch oder statisch, die rhythmischen Zeitwert eines gegebenen Rhythmus-Streams (*rh-s*) je nach *fkta* (Anfang) und *-e* (Ende)

Loop-rhythm erzeugt einen neuen Rhythmus-Stream, bei dem der gegebenen *rh-s* im Loop läuft für eine bestimmte Anzahl an Wiederholungen (*wdhs*). Das Argument (*last-intv*) gibt den zeitlichen Abstand zur nächsten Wiederholung des *rh-s* an und wird bei jeder Wiederholung auf die neue Startzeit aufaddiert.

*Loop-auf-beats e*rhält einen Fragment-Stream und einen Rhythmus-Stream. Die Fragmente werden im Verlauf zu Events mit den rhythmischen Zeitwerten als Startzeiten umgewandelt und als Event-Stream ausgegeben. Ist die Fragmentanzahl hier geringer als die Anzahl an rhythmischen Zeitwerten, werden die Fragmente im Loop als Events ausgegeben.

→ Siehe Anhang S. 53

2.7.2. Granulare Timestretch Funktionen

Beim granularen Streching wird ein Fragment in kurze Abschnitte (Shredds) zerlegt. Diese werden vervielfältigt um das Fragment in die Länge zu ziehen.

Hierbei werden 3 Segmente erzeugt:

- Segment1 (Start-Segment) = Anfang des Fragments unverändert
- Segment2 (dynamischer Stretch) = die Shredds werden durch einen Stretch-Anfangs-Faktor (*fkta*) und einen Stretch-End-Faktor (*fkte*),welche über die Zeit linear interpoliert werden , vervielfältigt.
- Segment3 (Abschluss) = Abschluss des Stretchs. Hier gibt es 4 verschiedene Funktionen:
  1. Der Rest des Fragments wird unverändert abgespielt
  2. Das "Stehenbleiben" auf dem letzten Shredd, was ein recht statisches Brummen erzeugt.
  3. Durch eine Sinus-Funktion, gemappt auf den Abtastzeiger zur Erstellung von Grains, wandert dieser innerhalb des Fragments zwischen zwei Zeitpunkten hin und her. Diese Funktion hat hierbei eine fest eingestellte „Frequenz“.
  4. Zweite Sinus-Funktionsvariante mit einer einstellbaren „Frequenz“.

Die „Frequenz“ (im Aufruf als *freqx* definiert) ist nicht wirklich die Frequenz der Sinusschwingung sondern ein Wert der diese beeinflusst. In der weiter untern stehenden Funktion *make-layer-skips* wird der *pi-inc* bestimmt durch die Teilung (Pi/2) / *freqx*. Der *pi-inc* ist eine aufsteigende Kommazahl für die Bestimmung der Werte der X(Zeit)-Achse der Sinus Funktion.

Folge Kürzel werden hier verwendet:

Kürzel	**ganzer Name**	**Bedeutung**
f	Fragment	
st	Starzeit	
art	*stretch-ab-ratio*	Faktor zwischen 0 und 1 zur Bestimmung ab wann der dynamische Stretch (Segment 2) innerhalb des Fragments einsetzt
brt	*stretch-bis-ratio*	Faktor zwischen 0 und 1 zur Bestimmung wann der dynamische Stretch (Segment 2) innerhalb des Fragments endet
fkta	*stretch-fkt-anf*	Vervielfältigungsfaktor-Anfang Segment2; nur ganze Zahlen ab 1 aufwärts möglich
fkte	*stretch-fkt-end*	Vervielfältigungsfaktor-Ende Segments2; nur ganze Zahlen ab 1 aufwärts möglich
lanz	*layer-anz*	Anzahl der Kopien der erstellten Stretch-Streams mit kurzem zeitlichem Delay um eine höhere Dichte zu gewährleisten
ls	Liste	
evt	Event	

Es gibt 3 Stretch-Funktionen, deren Ausgabe jeweils ein Event-Stream ist, der zur Umsetzung in Csound benötigt wird:

- *f-stretch-fein* : Fragment-shredding mit der Funktion *f-shredd-1*.
  → das Fragment wird in 10 ms große shredds zerlegt

- *f-stretch-grob* : Fragment-shredding mit der Funktion *f-shredd*,
  → das Fragment wird in 30 ms große shredds zerlegt

- *f-stretch-sin* : Sinus-Segment3 mit fest eingestellter oder variabler *freqx*
  Dieser Aufruf ist der gleiche wie für *f-stretch-grob*, hat jedoch einen anderen Namen zur besseren Unterscheidung der Funktionsaufrufs-Übersicht

Die uneingeschränkten Argumente *params* dienen hierbei dazu die Funktion des 3. Segments (Seg3) zu bestimmen:

1. kein Argument	=>	Funktion 1	Der Rest des Fragments wird unverändert abgespielt.
2. Ein Argument (n)	=>	Funktion 2	"Stehenbleiben" für n Events, n = Anzahl der Events des letztes shredds.
3. Zwei Argumente (n x)	=>	Funktion 3	Sinus mit festeingestellter *freqx,* n = Anzahl der Events für die Sinus-Bewegung, x = Faktor zwischen 0 und 1 für die Bestimmung der Range/Auslenkung des Sinus innerhalb des Fragments.
4. Vier Argumente (n x y z)	=>	Funktion 4	Sinus mit variabler *freqx,* n und x wie gehabt, y = '*freqx*-Anfang, z = zusätzlicher Pi-inc Wert um die *freqx* zu beschleunigen (positiv) oder zu verlangsamen (negativ).

Bei der abstrahierten Funktion der Stretches beinhaltet die *stay-ls* die weitergegebenen *params* und *proc* ist die weitergegebene shredd-Funktion.

stretch-start	bildet das 1. Segment
stretch-ende	bildet Seg3 - Funktion 1
stretch-layers	bildet Seg3 - Funktionen 2 - 4
stretch-layer	bildet eine Layer-Kopie, *ls* beinhaltet Werte für die Seg3 - Funktionen 2 – 4 Seg3:
make-layer-skips	liefert die Abtastzeiten der Seg3 - Funktionen 2-4, je nachdem wieviele Elemente in der liste sind:

Des Weiteren gibt es eine sehr spezielle Stretch-Funktion, welche in der Exposition des Stücks einen Klang generiert: *fs-make-events-stretched.*
Diese Funktion erhält neben einem Stream aus Fragmenten einen gleichlangen Rhythmus-Stream, so wie bei der Rhythmischen Bearbeitung. *fs-make-events-stretched* erzeugt quasi das 2. Segment eines Stretches, so wie oben definiert.
Dabei sind alle Parameter fest eingestellt, bis auf den Start. Dieser wird für jedes Fragment der Reihe nach dem Rhythmus-Stream entnommen. Gestretcht wird von dem Faktor 1 auf den Faktor 6. Die Entry-Delays verkürzen sich von 30 ms auf ca. 10 ms im Verlauf. Die Dauern von 40 ms auf 20 ms. Es wird keine Hüllkurve verwendet = Hüllkurven-ft 39.

→ Siehe Anhang S. 55 - 58

2.7.3. Materialgenerierung für die weiteren Prozesse

Für die folgenden Klanggenerierungsprozesse (rauschhafte Kopien) wird Klangmaterial benötigt. Dieses Material wird innerhalb der Zugriffsdaten abgelegt.
Für die Ref-Samples des TH1 und des TH2 wird gebildet:

- 380 transponierte Kopien über das gesamte Audiospektrum
- 100 Kopien in verschiedenen weißen Rauschhaftigkeits-Verhältnissen
- 100 Kopien in verschiedenen rosa Rauschhaftigkeits-Verhältnissen
- 1600 Notch-gefilterte Kopien
  → 40 Kopien in verschiedenen Abstufungen der Notch-Filter-Funktion mit jeweils 40 Kopien in verschiedenen weissen Rauschhaftigkeits-Verhältnis (40 x 40 = 1600)
- 1500 Koipen mit verschiedenen Einstellung der Bandpass-Filter-Funktion
  → 100 verschieden Center-Verhältnisse mit jeweils 15 verschiedenen Bandbreiten (100 x 15 = 1500)

Für das Ref-Sample 0 („Rrumpff tillff tooo“) werden ebenfalls 380 transponierte Kopien erstellt.

Aufrufe zur Materialgenerierung:

→ Siehe Anhang S. 95 (Aufruf) und S. 86/87 (Funktionen)

2.7.3.1. Erstellung der transponierten Kopien

Die Umsetzung von rauschhaften Samples wird realisiert durch das Addieren von transponierten Kopien des selben Samples. Um möglichst „weiß“ oder „rosa“ zu werden, werden hier 380 Kopien erstellt, 50 pro Oktave. Diese Kopien werden von der Funktion *pvoc* erstellt, welche, wie der Name schon sagt den Phasevocoder von Csound verwendet. Die transponierten Kopien werden im Ordner „Zugriffsdaten/PVoc/"Samplename"/“ abgelegt.
Hierbei wird eine Pvoc-ex-Datei erstellt, mit dem Namen "Samplename".pvx und abgelegt im Ordner „Zugriffsdaten/analysedaten/pvoc-ex-Daten/“. Um eine transponierte Kopie zu erstellen wird zudem eine score im Scores-pfad erstellt, welche nach ihrem Aufruf wieder gelöscht wird. Für die Umsetzung wird weiterhin ein Stream mit Transpositionsfaktoren benötigt. Dieser wird mit der

Funktion *make-transp-fkts* erstellt, welche im nächsten Absatz erklärt wird. Das entsprechende orchestra wird ebenfalls im Folgenden gezeigt.

→ Siehe Anhang S. 71

Die Funktion *make-transp-fkts* wird ohne Argumente aufgerufen und erstellt einen Stream mit den Transpositionsfaktoren über das gesamte Audiospektrum.
Hierbei wird von männlichem Sprachmaterial ausgegangen, dessen Grundfrequenzen um die 100 Hz liegen.

Es wird begonnen mit dem Transpositionsfaktor 0.25. Bei jedem Loop-Durchlauf werden zwischen dem Bereich *lower-fkt* und *upper-fkt* (*lower-fkt x 2*) 50 Zufallswerte erzeugt. Steigt der *lower-fkt* über 32 wird als *upper-fkt* 51 (die obere Grenze des pvoc) gewählt und weitere 30 Zufallswerte werden aus diesem Bereich gezogen. Insgesamt ergibt dieser Prozess 380 Faktoren, 50 pro Oktave. Der Faktor mit der das Sample nicht transponiert wird (1.0) befindet sich im resultierenden und sortierten Stream an Stelle 100.

→ Siehe Anhang S. 71

2.7.3.2. Erstellung der rauschhaften Kopien

Die Funktionen *make-sample-pnoise* und *-wnoise* erstellen jeweils 100 Kopien eines ref-samples in verschiedenen Abstufungen der Rauschhaftigkeit in dem Ornder „Zugriffsdaten/zwischensounds/sampleXXX-"name des ref-samples"/white/“, beziehungsweise „-/pink/“.

Die Funktionen *make-whites* und *make-pinks* wenden die abstrahierte Funktion *make-mat-abstr* mit der Prozedur *make-white-noise-event*, beziehungsweise *make-white-noise-event* auf das in einen Event gewandelten ref-sample an.

→ Siehe Anhang S. 65

Die Funktion *make-mat-abstr* wendet nun die jeweilige Prozedur mit einem aufsteigenden Rauschhaftigkeits-Verhältniswert (*ratio*) 100 mal im Loop auf das Event an. Der Name wird immer weitergegeben und erweitert durch String-Zahlen, die das Rauschhaftigkeits-Verhältnis angeben.

Die Prozedur *make-white-noise-event* bedient sich der Funktion *make-rausch-event* mit dem Amplituden-Stream für weiße Rauschhaftigkeit mit dem entsprechenden Rauschhaftigkeits-Verhältniswert (*rausch-ratio*). Die *make-pink-noise-event* geht ebenso vor mit einem Amplituden-Atream für die rosa Rauschhaftigkeit.

Die Funktion *make-amps-for-white-noise* beziehungsweise *make-amps-for-pink-noise* ertstellt nun einen Stream mit Amplitudenwerten für die Erstellung der gewünschten Rauschhaftigkeit. Diese Amplitudenwerte werden bei der Addition für die Lautstärke der einzelnen transponierten Kopien verwendet. Die Länge des Streams den *make-amps-for-white-noise* erzeugt richtet sich nach dem Verhältnis (*ratio*).
Mittelpunkt des Rauschhaft werdens ist die 100. Transponierte Kopie, das Original. Von hier ab werden bei *make-amps-for-white-noise* in beide Richtungen, bestimmt durch das die *ratio* die Amplitudenwerte erstellt, wobei auch ein Flankenwert mit einbezogen wird. Dieser wird bei steigender *ratio* geringer und bewegt zwischen 0 und 12 dB.

→ Siehe Anhang S. 64

Bei der Funktion *make-amps-for-pink-noise* hat die *ratio* keine Auswirkung. Hier wird ein in jedem Fall 380 Amplitudenwerte langer Stream erzeugt, wobei über jede Oktave die Ausgangsamplitude um 6 dB abfällt, sowie es beim Rosa Rauschen der Fall ist.

Die Funktion *make-rausch-event* wendet nun die Amplituden auf die entsprechenden transponierten Kopien an und addiert diese zusammen zu einer Audiodatei: Sie reicht hierfür die Werte weiter an *make-rausch-event-abstract*. Diese Funktion ist abstrahiert, da es noch weitere Funktionen gibt, die sie anwenden mit anderen Argumenten.

Die Funktion *make-rausch-event-abstract* erstellt nun die Audiodatei. Hierbei ist die *entry-proc* = (lambda (f)(event-entry f)), d.h. der Startwert des Events wird verwendet, die *render-proc* = *events-make-sco-n-render,* eine simple Funktion, die einen Event-Stream in eine Score wandelt und diese mit Csound aufruft.

Die Funktion ruft zunächst *pvoc* mit dem entsprechenden Ref-Sample auf und wenn es noch keine transponierten Kopien desselben geben sollte, werden sie jetzt erstellt.
Im Weiteren werden die benötigen Events zum Aufruf der transponierten Kopien und ein neues ftable-File erstellt und per *render-proc* das ganze zu einer Audiodatei gerechnet.

→ Siehe Anhang S. 63

Die Funktion *pvoc-ftables-file* erstellt ein ftable-File im ftables Ordner für den Zugriff auf die transponierten Kopien. Sie funktioniert ähnlich wie die argumentlose Funktion *make-ftables-as-file*. Hier wird jedoch ein Soundfile-string-stream und ein Name gegeben.

→ Siehe Anhang S. 64 (s. Unten)

2.7.3.3. Erstellung der rauschhaften Kopien mit Notch-Filter-Funktion

Die Funktion *make-sample-notches* erstellt jeweils 40 Kopien der Ref-Samples in verschiedenen Abstufungen der Notch-Filter-Fuktion mit jeweils 40 Kopien eines ref-samples in verschiedenen weißen Rauschhaftigkeits-Verhältnis. Dies ergibt 1600 Notch-gefilterte Kopien jedes Ref-Samples. Die Kopien werden in den Ordner „Zugriffsdaten/zwischensounds/sampleXXX-"name des samples"/notch/“ abgelegt.

Die Funktion *make-notches* wendet die Funktion *white-random-notches* mit verschiedenen Einstellungen für das Rauschhaftigkeits-Verhältnis und für das Notchfilter-Verhältnis mehrmals auf das in ein Event gewandelte Ref-Sample an. Der Name wird immer weitergegeben und erweitert durch String-Zahlen, die das Rauschhaftigkeits-Verhältnis und das Notchfilter-Verhältnis angeben.

Die Funktion *white-random-notches* erstellt die neue Kopie. Hierbei werden zufällig ausgewählte transponierte Kopien nicht hinzu addiert, was sich durch die Filterung des Amplituden-Streams und im Weiteren des Soundfile-Streams definiert. Die Anzahl richtet sich dabei nach dem Notchfilter-Verhältnis (*filt-rat;* 1.0 = nichts; 0.0 = alles), bezogen auf die verwendeten Kopien, die sich nach dem Rauschhaftigkeits-Verhältnis (*rausch-rat*) für weiße Rauschhaftigkeit (1.0 = alle; 0.0 = keine) richten.

→ Siehe Anhang S. 66

Der Renderbefehl wird ebenfalls mit der Funktion *make-rausch-event-abstract* erstellt, wobei diese hier die Filter-Funktion *notches* erhält zusammen mit der *filt-rat*. Dies ist notwendig um in *make-rausch-event-abstract* den Soundfile-Stream an den gefilterten Amplituden-Stream anzupassen. Dies verhält sich ähnlich wie die Filterung des Amplituden-Streams innerhalb von *white-random-notches*.

→ Siehe Anhang S. 68

2.7.3.4. Erstellung der rauschhaften Kopien mit Bandpass-Filter-Funktion

Die Funktion *make-sample-bandpassed* erstellt jeweils 15 Kopien eines Samples in verschiedenen Abstufungen der Bandbreite von jeweils 100 Kopien eines Samples mit weißer Rauschhaftigkeit (100%) mit verschiedenen Abstufungen der „Center-Frequenz".
Die „Center-Frequenz" wird hier nur so genannt, sie ist in Wirklichkeit auch nur ein Verhältniswert, der auf die Anzahl der transponierten Kopien gemappt wird. Die Bandbreite ist ebenfalls ein Verhältniswert zwischen 0 und 1.
Es ergeben sich 1500 Bandpass-gefilterte Kopien jedes Samples. Die Kopien werden in 100 verschiedene Ordner im Ordner „Zugriffsdaten/zwischensounds/sampleXXX-"name des ref-samples"/bp/" abgelegt.

→ Siehe Anhang S. 86/87 und S. 84

Die Funktion *make-bandpasses* wendet die Funktion *white-bandpass* mit verschiedenen Einstellungen für das Center-Verhältnis und für das Bandbreite-Verhältnis mehrmals auf das, in ein Event gewandelte Ref-Sample an. Der Name wird auch hier immer weitergegeben und erweitert durch String-Zahlen, die das Center-Verhältnis und das Bandbreite-Verhältnis angeben.

Die Funktion *white-bandpass* erstellt die neue Kopie. Hierbei wird ein Center-Index, der auf die entsprechende transponierte Kopie verweist, erstellt und ähnlich wie bei der weißen Rauschhaftigkeit werden von dieser Kopie ab in beide Richtungen, entsprechend des Bandbreite-Verhältnisses, die Auswahl an transponierten Kopien definiert für den Aufruf. Eine zusätzliche Flankenfunktion mit festeingestelltem Wert (Faktor 0.5 über eine Oktave) erstellt die passenden Amplitudenfaktoren. Die resultierende Flanke lässt sich nicht in dB/Okt angeben, da beispielsweise

von einer ausgehenden Amplitude von 90dB über eine Oktave nur noch 45dB übrig sind.

→ Siehe Anhang S. 66/67

Der Renderbefehl wird auch hier mit der Funktion *make-rausch-event-abstract* erstellt. Diese verhält sich ähnlich wie bei *white-notches*, wobei hier die *filter-proc* die Funktion *bandpass* ist. Die Parameter hierfür die Center-ratio und die Bandbreite-ratio.

2.7.4. Prozess „Sinus-Stretched“-Flächen /Verzahnung /rauschhaft werdend

Die Erstellung für diesen sehr speziellen Prozesses findet statt durch die Funktion *make-proz*.

Dies ist die Funktion zur Erstellung des 2. Teils der Durchführung des Stückes *Rrumpff tillff toooo?*. Hierbei werden 10 parallel laufenden "Sinus"-Flächen gebildet, welche aus allen 10 Fragmenten des TH1 stammen. Die Einsatzzeitpunkte der einzelnen Events der Flächen sind exakt gleich, um die Vorraussetzung für den folgenden Prozess zu gewährleisten.

Dieser Prozess ist ein „Verzahnungsprozess“ der einzelnen Event-Streams, aus denen die Flächen bestehen. Verzahnung bedeutet, dass im Verlauf jedes Event eine Position einnimmt in einem resultierenden Stream aus Events. Vorher waren zu jedem Zeitpunkt 10 Events. Nach und nach wird aus den 10 Event-Streams ein einzelner "verzahnter", bei dem die Events in einer Reihe laufen.

Die Funktion *make-proz* erstellt aus den einzelnen Fragmenten des TH1 die so genannten "Sinus"-Flächen. Diese verhalten sich wie die *f-stretch-sin* -gestretchten Fragmente mit einer sehr hohen Eventanzahl und sind dementsprechend lang. Hierfür wurde ein kleines Objekt-orientiertes Framework verwendet, wobei für jedes Fragment zwei Objekte erstellt werden. Eines für die Sinus-bewegten Abtastzeiten (*cycle-obj-fx*) und eines für die Startzeiten (*entry-obj-fx*). Die cycle-Objecte werden Erstellt mit einem *stream-walker-maker*, der einen Stream mit den Abtastzeiten für einen kompletten Wellendurchgang erhält, welchen die Funktion *vz-pos-cycle* erstellt.

→ Siehe Anhang S. 14 (objects-fw) und S. 59 (vz-pos-cycle)
→ Für alle folgenden Funktionen, Siehe Anhang S. 59 – S. 63

Das cycle-Object liefert bei jedem Aufruf den aktuellen Abtastzeitpunkt und wandert nachdem 'update, nachdem jeder Layer der Fläche mit dem selben Abtaststart versehen wurde, einen Schritt weiter im Stream. Ist der Stream zu Ende fängt das Objekt von Vorne an, sodass die Sinusbewegung gewährleistet ist.
Das entry-Object ist ein simpler *counter-maker* der ab einem gegebenen Wert um ein gegebenes Intervall hoch zählt und somit den Events ihre Startzeit gibt.
Das objektorientierte Framework wurde hierbei gewählt, da die Streams für diesen Prozess zu lange wären.

Make-proz erhält 3 Zeiten (*t1 t2 t3*). *t1* ist die Zeit, ab der der Verzahnungsprozess beginnt. Dieser dauert bis *t2*. Ab *t2* und bis *t3* wird der verzahnte Stream in seinen Dauern und seiner Rauschhaftigkeit manipultiert. Beides steigt an. Die Rauschhaftigkeit bis zu einem weißen Rauschen -ähnlichen Audiospektrum.
Die beginnenden „unverzahnten“ Flächen erhalten innerhalb der Prozedur Ihre Startpunkte:

TH1-f1 beginnt bei 0.0,	TH1-f2 bei 83.0,	TH1-f3 bei 76.0,	TH1-f4 bei 47.0,
TH1-f5 bei 63.0,	TH1-f6 bei 46.0,	TH1-f7 bei 88.0,	TH1-f8 bei 84.0,
TH1-f9 bei 30.0 und	TH1-f10 bei 93.0		

Diese Zeiten werden zu Beginn der "Sinus"-Flächen-Starts so angepasst, dass gewährleistet ist, dass die folgenden Events synchron laufen. Hierfür wird vor dem Aufruf eine globale Variable definiert (*g-synch*), die anhand des ersten Fragment-Stretches einen Synchronisations-Zeitpunkt beinhaltet. Hierzu wird die Funktion *sync-start* verwendet. Diese bedient sich an den Zeitpunkten der nächsten Events, die die Funktionen *stretch-start* und *stretch-layers* (s.o.) als erstes Element ausgeben und vergleicht den Start des 3. Segments mit dem globalen *g-synch* durch die Funktion *check-synch*.

Die Funktion *check-synch* gibt die Verschiebung des Starts der *stretch-prod* an. Dabei wird immer zum letzten Zeitpunkt synchronisiert, auch wenn der nächste Zeitpunkt näher ist. Die *time* muss dabei immer größer sein als der *g-synch.*

Den Beginn der Stretches (vor den „Sinus“-Flächen) erstellt die Funktion *vz-stretch*, welche wie die *f-stretch-*.. Funktionen aufgebaut ist, jedoch ohne das 3. Segment. (siehe oben → granulares Stretching)

Mit der Funktion *out* werden die Event-Streams in Textfiles ausgegeben und abgelegt in den Ordner „zugriffsdaten/analysedaten/str_vz_rausch/"

Das 3. Segment der Stretches und somit die "Sinus"-Fläche wird von der Funktion *vz-bis-vzng* erstellt. Diese durchläuft die beiden Objekte (cycle und entry) bis zu einem gegebenen Zeitpunkt (*t1*) und erstellt und schreibt Events, ebenso wie oben, in ein Textfile mit dem Namen *filename.*

Ab dem Zeitpunkt *t1* beginnt der Verzahnungsprozess mit der Funktion *verzahnung*. Diese erhält ein Entry-object (mittlerweile sind alle an der selben Positon angelangt = *t1*), den Zeitpunkt bis wann die Verzahnung beendet sein soll (*bis-t = t2*) und alle cycle-Objekte als uneingeschränkte Argumente.

Die *switcher*-Unterfunktion erledigt den Prozess. Hierbei steigt die Wahrscheinlichkeit einer "Einreihung" eines Events in den verzahnten Event-Stream im Verlauf. Ausgabe ist ein Textfile mit Listen. Jede der Listen beinhaltet den Anstieg des Einsatzzeitpunktes (hier immer null), den Anstieg der Dauer (hier sehr geringfügig), den Startzeitpunkt eines Events oder aller Events (je nachdem ob "eingereiht" wurde oder nicht) und dementsprechend eine Abtastposition oder alle.

Weiter geht es mit der Funktion *verzahnt,* welche die Daten in Reihe schaltet. Sie verhält sich wie die Funktion *verzahnung* ohne eine Wahrscheinlichkeitserhöhung. Hier wird immer "eingereiht". Neben dem Anstieg der Dauern werden hier auch Werte größer als Null für die Vergrößerung der Einsatzabstände erstellt. Die Ausgabe ist auch hier ein Textfile.

Um nun wieder die 10 Event-Streams zu erhalten, um sie für den Prozess der Rauschhaftwerdung einzeln zu verwenden, holt die Funktion *get-out-of-vzng* die einzelnen Daten aus den erstellten Textfiles und reiht sie zusammen. Um die richtigen Daten zu finden erhält die Funktion die Center-cycle-postion (*cycle-time*; = Nulldurchgang der Sinus-Abtastung) und die Range der Sinus-Abtastung. Das Ergebnis sind hierbei auch wieder Textfiles.

→ Siehe Anhang S. 96-98 (*make-proz*)

Im Anschluss an die Textfile-Erstellung werden die Events aus diesen zusammengefügt und ausgegeben mit der Funktion *vz-render.* Diese Funktion erhält als erstes Argument der uneingeschränkten Argumente s*trings*, den Namen des Output-files. Alle folgenden strings stellen

die Namen der Daten-Textfiles dar. *Vz-render* schreibt alle Events in csound-i-Zeilen um und gibt diese zusammen mit allen ftables in einer Score aus. Im Anschluss wird der Befehl *csound-render* aufgerufen.

Nun ist der größte Teil des Prozesses in Soundfiles geschrieben („Reihe-f1.. - f10“) und es kann der Prozess der Rauschhaftwerdung erfolgen. Die Textfiles werden zunächst in Event-Streams gewandelt:

(define f01-to-noise (file-to-stream-each-obj (string-append analysedaten "str_vz_rausch/" "4-f01.daten") und folgende.

Diese Streams erhält die Funktion *events->rauschen* zusammen mit einem Namen für das zu erstellende Soundfile.
Diese Funktion ist auch speziell auf diesen Prozess ausgelegt und bedient sich in aufsteigender Reihenfolge der rauschhaften Kopien des TH1 im Ordner „Zugriffsdaten/zwischensounds/ sample001-thema2/white/“. Hierbei erhalten die Events neue ftable-Nummern und es wird eine ftable-Datei erstellt mit den ftables für die transponierten Kopien. Dies erledigt die Funktion *make-zwischensounds-ft*.

→ Siehe Anhang S. 69 und S. 85

Nun wurden bei der Realisierung des Stücks die Audiodateien "Reihe-fxx-2" (xx = 01 – 10) zeitlich an die Audiodateien „Reihe-f01“ bis „Reihe-f10“ angepasst und eine kurze Überblendung erstellt, sodass der Prozess komplettiert wurde.

2.7.5. Granuliertes Rauschen

Die Klänge der Durchführung, welche in ihrem Klangbild am weitesten entfernt liegen vom Ausgangsmaterial, wurden erstellt mit der Funktion *make-multi*.
Diese erhält ein Fragment, ein Argument (*arg*), eine Anzahl für die zu erstellenden Events (*evts-anz*), einen Stream für die Audiodatei-ftable-nummer (*ft-s*) , einen Stream für die Hüllkurven-ft-nummer (*huell-s*), einen Namen und ein mögliches weiteres Bandpass-argument (*bp-arg*).

Make-multi erstellt einen Klang, der durch das Durchfahren einer Kategorie von verrauschten Kopien erstellt wird. Hierbei werden Events erzeugt mit feststehender kurzer Dauer (20 ms) und Abtastzeiten, welche linear vom Fragment-Abtaststart bis zu seinem Ende über die *evts-anz* ansteigt. Das Fragment bestimmt welche Kopien welches Ref-Samples durchfahren werden.
Das *arg* bestimmt welches rauschhaftes Material verwendet wird ('white 'pink oder 'bp). Durch den *ft-s* wird bestimmt von welcher Kopien zu welcher anderen Kopie die Dateien durchfahren werden sollen und reguliert somit das Klangbild. Das mögliche *bp-arg* bestimmt bei der Nutzung der Bandpass-Kopien welche Bandbreite verwendet werden soll. *bp-arg* muss eine ganz Zahli sein und sollte den Wert 15 nicht überschreiten, andernfalls wird es ignoriert und eine 3 stattdessen verwendet. Dies geschieht in der Funktion *make-zwischensounds-ft-bps*, an die das *bp-arg* weiter gegeben wird.
Hauptfunktion von *make-multi* ist die Erstellung von ftable-files und die Ausführung des *csound-render* -Befehls.

→ Siehe Anhang S. 69

Die Funktion *make-zwischensounds-ft-bps* ist speziell auf die Verwendung von Bandpass-gefilterten Kopien ausgelegt. Diese Kopien stellen eine Ausnahme dar, da sie in 100 unterschiedlichen Ordnern untergebracht wurden. Innerhalb der Funktion wird nun ein Stream mit Pfaden und Audiodateien erstellt, aus dem dann die ftable-Zeilen generiert werden. Es wird eine ftable-Datei erstellt.

→ Siehe Anhang S. 85

2.7.6. Die Funktion *rand-rotate-notch-ersatz*

Die Funktion *rand-rotate-notch-ersatz* dient dazu Fragmente mit zufällig gewählten Notch-gefilterten Fragmenten zu ersetzten. Zudem wird ein Rhythmus eines Fragment-Streams, idealerweise einer Fragment-Phrase, rotiert und die so gewandelten Zeiten werden als neuer Rhythmus verwendet. Die Fragmente-Ersatz-Kopien werden ebenfalls rotiert auf den Rhythmus gemappt.
f-s1 = die Fragmentphrase, dessen Rhythmus, rotiert durch *rot1,* verwendet wird. *f-s2* beinhaltet die Fragmente, die ersetzt und ebenfalls rotiert in ihrer Reihenfolge (*rot2*) auf den Rhythmus gemappt werden.

→ Siehe Anhang S. 69 (s. Unten) und S. 53/54 (s. Unten *make-rhythm-rotation*)

2.7.7. Fragment-Nachbildung

Zur Nachbildung eines Fragments des TH1 werden alternative Abschnitte aus stimmlosen und stimmhaften Sprachmaterial verwendet um die entsprechenden Abschnitte innerhalb eines Fragments nachzubilden. Diesen Prozess erledigt die Funktion *build-frag* und gibt einen Event-Stream aus. Diese Events werden mit einer sinusoiden Hüllkurve versehen. Ausgegeben wird hierbei mit einem Orchestra *komplete-mit-globalhuell.orc*, welches ein globales Instrument für einen Lautstärkeverlauf über alle Events erstellt. Hier kommen die vorab gefertigten Fragment-Hüllkurven in Form von Ascii-Textfiles ins Spiel (s. Kapitel 2.6.3. Fragmenthüllkurve als Ascii-Textfiles, S. 16).
Dessen ftable-Datei wird mit in die Score geschrieben um darauf zuzugreifen. Zudem wird aus dem Fragment entsprechenden Textfile der benötigte Verhältniswert gezogen, der das Verhältnis der Fragmentdauer zu der 2^n-Länge des Textfiles angibt. Dieses wird wiederum auf die Dauer des Aufrufs des globalen Lautstärkeverlaufs angewendet, um zu gewährleisten, dass die komplette Hüllkurve richtig eingesetzt wird.

→ Siehe Anhang S. 89 - 91

Build-frag ist die Nachbildungsfunktion. Sie erstellt die Nachbildung eines Fragments anhand seiner stimmlosen und stimmhaften Abschnitte (erzeugt durch die Funktion *error-pitch-abschnitte*)

durch passende Abschnitte aus den vorab erstellten „straight-Error-“ und „-pitch-Daten“ der LPC-Analysen. Ausgewählt werden die Abschnitte durch die Funktion *def-abschnitt->auswahl*. Die auf diesem Wege erstellten "Ersatz"-Events erhalten im Loop noch angepasste Startzeiten und Dauern.

Die Funktion *error-pitch-abschnitte* erstellt eine Liste aus Wertelisten. Die Wertelisten geben stimmlose oder stimmhafte Abschnitte innerhalb eines gegebenen lpc-Daten-Streams an. Stimmlose Abschnitte sind definiert mit dem beginnenden Argument 'e, da in diesem Abschnitt der Error-Wert dominiert. Stimmhafte Abschnitte sind definiert mit dem beginnenden Argument 'p, da in diesem Abschnitt der Error-Wert unerheblich ist. Auf diesem Wege (Error dominiert oder ist unerheblich) untersucht die Funktion den Daten-Stream. Dabei wird das Argument, ausgehend vom ersten Datenpaar, immer wieder gewechselt und bei einem Wechsel eine neue Abschnittslist erstellt, wobei die bis dahin gesammelten Daten nun gemittelt der Abschnittsliste hinzugefügt werden.

Die Funktion *def-abschnitt->auswahl* untersucht die *straight-error*- oder die *straight-pitch*-Daten, entsprechend dem Argument innerhalb der *def-ls* (= '(arg start end avereged-data), arg = 'e (Error) od. 'p (Pitch)) nach passenden Abschnitten.

Für das Stück wurden nur die Fragmente des TH1 nachgebildet. Sie sind zu hören beim Höhepunkt der Durchführung.

2.7.8. Prozess Fragment-Alternativen-Setzung, shredding und pitchhits

Am Anfang des letzten Drittels des Stücks *Rrumpff tillff toooo?* (Teil RF-1; RF = Rückführung) kommen die Funktionen des Fragmentpools zum Tragen. Hier wurden aus allen Fragmenten des Ausgangsmaterials Alternativen zum Ersatz der Fragmente des TH1 und des TH2 gezogen. Es wurden hierbei nach der Dauer, der Error-Werte und der Pitch-Werte der LPC-Analysedaten und nach den Attack- und Release-Zeiten gesucht und verglichen.
Die alternativen Fragmente wurden auf ihre neuen zeitlichen Positionen gesetzt und somit wurden die Haupthemen des Stücks (TH1 und TH2) neu gebildet.

Nun wurden Loops gebildet, während deren Verlauf der Ersatz stattfindet. Hierbei erscheinen TH1 und TH2 im Wechsel. Beim TH1 geht es von den originalen Fragmenten in 10 Wiederholungen (wegen den 10 Fragmenten des Satzes) zu den Alternativen und beim TH2 verläuft der Prozess

rückwärts auch über 10 Wiederholungen.
Eine weitere Funktion wurde auf die ersetzen Fragmente angewendet, das Shredding.

Das Shredding zerlegt die Fragmente in gleichlange Shredds, ähnlich wie es die *f-shredd* Funktionen machen. Im Verlauf der Loops werden die ersetzen Fragmente in immer kleinere Schnipsel zerlegt und zum Ende hin zeitlich durchmischt.
Zum Ende des ganzen Prozesses bildet sich aus den zu Shredds zerlegten Fragmenten eine lauter werdende Grain-Wolke. Die Grains erfahren zudem zum Ende des ersten Drittels des RF Teil1 diverse Pitchshifts um noch konfuser zu wirken.

2.7.8.1. Funktionen des Fragment-Pools: Alternativen

Um alternative Fragmente zum Ersatz eines Fragments zu finden wurde die Funktion *f-find-alternate* entwickelt. Diese untersucht einen gegebenen *global-data-pool* nach den nächstgelegensten Werten für den Error-Wert, den Pitch-Wert, die Releasezeit und die Attackzeit.
Der *global-data-pool* wird in der Regel erstellt durch die Funktion *attack-length-averaged-data-stream-maker*, welche die in der vorangegangenen Analysephase erstellten Daten (Attack, Length und die gemittelten pitch- und error-Werte) in einen Stream packt.

→ Siehe Anhang S. 31 (S. Unten) und S. 18

F-find-alternate holt zunächst 40 Datenlisten aus dem Daten-Pool, die in ihrem Error-Wert nahe an den des Fragments heran kommen. Aus diesem neuen Pool werden im Anschluss 20 Datenlisten gezogen die in ihrem Pitch-Wert nahe an den des Fragments reichen. Daraus werden weitere 10 gezogen, die in Ihrer Releasezeit am nächsten zu der des Fragments liegen. Zuletzt wird aus diesen 10 die Datenliste ausgewählt, die am nächsten in ihrer Attackzeit zu der des Fragments liegt.

→ Siehe Anhang S.75 – S 77

Alle *find-(n-)nearest-X* Funktionen verwenden die Funktion *find-n-nearest-abstr*, welche die Auswahl trifft. Hierbei werden die Listen des Pool an der entsprechenden Stelle nach einem gegebenen Wert (*val*) untersucht und „*n*“-viele nahe liegende Daten werden in ihre Datenliste ausgegeben.

Durch die Funkiton *fs-find-alternates* ist es möglich *f-find-alternate* auf einen ganzen Stream von Fragmenten anzuwenden, ohne ein alternatives Fragment mehrmals zu ziehen oder ein Fragment desselben Stream zu ziehen.
Der *global-pool* auf den zugegriffen wird, wird zunächst von allen Fragmenten des *frag-streams* gesäubert und nach jedem Fund wird das alternierende Fragment ebenfalls aus dem Pool abgezogen. Weitere Fragmente, die aus dem *global-pool* verbannt werden sollen, können als *Ausnahmen*-Stream angehängt werden.
Die gezogenen Fragmente werden in ihren Zielsamplenummern und den Zielpositionen auf die Originalen des Fragments gesetzt.

Alternative Fragment-Streams beinhalten Fragmente deren Zielsamplenummern und -posi-tionen nicht denen der Originalen entsprechen.

Für den Prozess der Fragment-Alternativensetztung des Stückes wurde die Funktion *p-durchlauf+ersatz* entwickelt die eine gegebene originale Fragmentphrase (=stream) und eine mit dessen alternativen Fragmenten in der richtigen Reihenfolge erhält. Es wird ein Loop der Fragmentphrase erstellt, in dem über eine fragment-anzahl-fache Wiederholung die Frag-mente des Originals Schritt für Schritt ersetzt werden durch die Alternativen. Die Position des nächsten ersetzten Fragments wird hierbei zufällig gewählt. So geschieht es für das TH1.
Für das TH2 läuft dieser Prozess anders herum, sodass die Alternativen Fragmente den Loop beginnen und im Verlauf ersetzt werden durch ihre originalen Pendants. Das Ergebnis der Funktion *p-durchlauf+ersatz* wurde hierbei gekürzt um 3 Wiederholungen, sodass die 13 Fragmente des TH2 auch über 10 Wiederholungen der Phrase ersetzt werden.

→ Anhang S. 108 (Aufrufe für das Stück)

2.7.8.2. Loop-Bildung des Prozesses der Ersetzung

Die Funktion *p-durchlauf+ersatz* wandelt zunächst die beiden Fragment-Streams um in Listen, um besser damit umzugehen. Es wird mit einer Positionsliste aus Nullen und Einsen gearbeitet, die im Verlauf der Loop-Erstellung die Positionen innerhalb der Fragmentphrase angibt, die bereits gewechselt wurden (1) oder die noch nicht gewechselt wurden (0). Bei jedem Durchgang der

Haupt-Funktion *build-stream* wird eine Fragmentphrase (wieder als Stream) erzeugt, bei der eine Position gewechselt wurde. Für eine bessere Übersicht bei der Arbeit mit *p-durchlauf+ersatz* wird ein 'x als Element zwischen die Phrasen gesetzt um bei Ansicht der Daten leichter unterscheiden zu können

→ Siehe S. 78/79

Um den so entstandenen neuen Fragment-Stream mit seinem ursprünglichen Rhythmus fortlaufend zu versehen, wird die Funktion *rhythm-from-orig* zunächst auf die originale Fragmentphrase angewendet um die Abtastpeakzeiten zu erhalten. (Siehe Abschnitt „Rhythmus Bearbeitungen“ oben).

Die Funktion *p-orig-altern-loop* wandelt nun die originalen und alternativen Fragmente in Events um und reiht sie in einen Event-Stream.
Dabei wird der von *rhythm-from-orig* erstellte Rhythmus-Stream (*rhythm-s*), eine Startzeit (*start*) und nach Durchlauf einer Phrase eine Loop-Zeit (*wdh-t*) eingehalten. Die alternativen Fragmente werden in Ihren Abgriffzeiten durch die Funktion *f-set-abgriffzeiten* an die Zeiten der originalen angepasst (Hierbei wird auf die Zielsamplenummern und -positionen geschaut, die entsprechenden originalen Fragmente werden als Referenz erstellt und anhand der Abgriffzeiten des Originals wird das alternative Fragment angepasst).
Die Events, die anhand der alternativen Fragmente erzeugt werden, werden mit einem 'x versehen , damit sie leichter erkannt werden für den weiteren Granularisierungsprozess, das Shredding.
Die 'x-Elemente der *p-durchlauf+ersatz* -Funktion werden durch ein 'Z ersetzt.
Neben den gegebenen Argumenten *f-s* (die Fragment-Phrase erstellt von *p-durchlauf+ersatz*), *wdh-t*, *rhythm-s* und *start,* erhält die Funktion *p-orig-altern-loop* eine Präposition (*prep*) und einen Stauchungs-Divisor (*stauch-div)*, der die *wdh-t* teilt, sodass diese im Verlauf immer kürzer wird und somit die Wiederholungen der Phrasen zeitlich immer enger zusammenrücken. Die *prep* reguliert, welche Fragmente (orig oder altern) mit dem 'x für den weiteren Granularisierungsprozess versehen werden. Beim TH1 sind dies die Alternativen, bei TH2 die Originalen. Das heisst die Fragmente, die im Verlauf die startenden Fragmente ersetzen.

Nun müssen für den weiteren Prozess der Granularisierung die Events mit dem 'x aus dem Event-Stream gezogen werden. Die restlichen Events werden ebenfalls separiert für die Ausrechnung der Phrase durch Csound. Die Funktion *p-get-out-w-no-x* holt die events ohne 'x aus dem Event-stream und die Funktion *p-get-out-w-x* jene mit 'x.

2.7.8.3. Schrittweise Granularisierung und pitchhits

Den Zerlegungsprozess der 'x-Events erstellt die Funktion *shreddnzoom-events-pro-peak*.
Diese zerlegt die Events im Verlauf linear in gleich große Abschnitte, wobei die Teile-Anzahlen zwischen *teile-anf* und *-end* aufsteigt bzw. absteigt. Im Falle von *Rrumpff tillff toooo? s*teigt sie an. Die so erstellten, kürzer werdenden Abschnitte werden im Verlauf vervielfacht, in Abhängigkeit zu den Faktoren *zoom-fkta* und *-e*. Zudem steigen im Verlauf die Abtastbereiche der einzelnen Events an (*wachs-a* und *-e*). Den Prozess der Zerlegung erstellt bzw. kontrolliert die Funktion *shredd-kontrolle*. Den Prozess der Vervielfältigung erstellt die Funktion *shredd-events-zoom*.
Am Ende erhalten die neuen Events Hüllkurven-fts, deren Funktionen von dem 1. Segments einer Rechteck-Schwingung (keine Kurve) hin zu dem 1. Segment einer Sinus-Schwingung über 31 Interpolierungen der beiden Kurven laufen, sodass aus den Shredds im Verlauf Grains werden.

→ Siehe Anhang S. 48/49

Die Funktion *shredd-kontrolle* wendet die Funktion *event-shredd-anz* auf einen Event an, kontrolliert jedoch vorher, ob die resultierende Shredd-Dauer nicht unter 10 ms fällt. Ist dies der Fall wird das Argument *teile* so angepasst dass eine Dauer von 10 ms eingehalten wird.
Die Funktion *event-shredd-anz* zerlegt nun ein Event in die gewünschte Anzahl an gleich großen Teilen.

Die Funktion *shredd-events-zoom* erhält den Stream aus geshreddeten Events und den zoom-Faktor, vervielfältigt nun die Shredds und setzt sie auf neue Startzeiten, damit der Zoom zeitlich in beide Richtungen stattfinden kann.

Die geshreddeten Events aus dem TH1 und dem TH2 werden nun durch die Funktion *zerwuerfel-events* mehr und mehr im Verlauf durchmischt. D.h. die Zeiten werden getauscht, sodass innerhalb eines geshreddeten und aufgezoomten Events ein Shredd eines anderen auftaucht. Dieser Prozess läuft bis zur vollständigen Durchmischung. Beide ursprünglichen Event-Streams werden bei Abschluss des Prozesses getrennt voneinander in einer Liste ausgegeben um sie anschliessen zu separieren.

→ Siehe Anhang S. 45

Für den Abschnitt innerhalb des Stückes wurden auf dem oben beschriebenen Weg 4 Audiodateien erzeugt:

1. Loop der Originalen Fragmente des TH1 (*Loop-TH1-flow-orig*)
2. Loop der alternativen Fragmente des TH2 (*Loop-TH2-flow-altern*)
3. Loop der alternativen Fragmente des TH1, welche im Verlauf geshreddet und zerwürfelt wurden (*Loop-TH1-flow-altern*) und
4. Loop der ebenso bearbeiteten originalen Fragmente des TH2 (*Loop-TH2-flow-orig*).

→ Siehe Anhang S. 105 (Aufrufe)

Um den Prozess der Granularisierung weiterzuführen wurden 2 granulare Phrasen erzeugt (*Loop-TH1-TH2-wolke-A* und *-B*) , welche die erstellten und geshreddeten, alternativen Frag-mente des TH1 (*TH1-events-shredded*) und die geshreddeten, originalen Fragmente des TH2 (*TH2-events-shredded*) als Ausgangsmaterial verwenden. (Die Betitelung „Loop“ trifft hier nicht mehr zu, wurde aber gewählt, um die Zugehörigkeit zu den vorhergegangen Loops zu verdeutlichen)

Die Granularisierungs-Funktionen *p-make-wolke* , *p-make-pitchhits-wolke-A* und *-B* erzeugen hierbei die granularen Phrasen. Der Unterschied zwischen beiden ist die Verwendung von unterschiedlichen Pitch-Funktionen. Während die Funktion *p-make-wolke* bei einem gegebenen Pitch-Wert (für Csounds's cpsoct) diesen auf ein Event mappt, erstellen die Funktion *p-make-pitchhits-wolke-A* und *-B,* nach einer bestimmten Wahrscheinlichkeit die so genannten „Pitchhits“, anstatt nur einen normalen Pitchshift zu machen.
Ansonsten sind die Funktionen gleich. Sie erhalten alle zwei Streams mit Events als Material und weitere 4 Streams für die Einsatzabstände, die Dauern, die Lautstärken und die Pitch-Werte. Diese Werte werden bei Aufruf auf die zufällig gewählten Events aus dem Material gemappt und als neuer Event-stream ausgegeben.

→ Siehe Anhang S. 80/81

Die so genannten „**Pitchhits**“ sind über einen Bereich des Audiospektrums gepitchte und zu-sammen addierte Kopien eines Sounds, wobei das relative Interval einstellbar ist. Hierbei entstehen „Blitz“ -artige Klänge, die in Ihrer Tonhöhe schlagartig in die Tiefe gehen, um dort zu versinken. Dies aufgrund der durch das Pitchshifting veränderten Dauer des Klangs.

Erzeugt werden die „Pitchhits“ durch die Funktion *make-transp-up-n-down*.

Make-transp-up-n-down wandelt einen gegebenen Event in mehrere Events mit unterschiedlichen Pitch-Werten und dementsprechenden Dauern. Hierbei wird das Interval (in cspokt-schritten) und die Minimale und maximale Oktave (auch in cpsoct-Werten) gegeben. Das absolute Minimum ist jedoch festgesetzt bei 6.0 cpsoct, was in einer 4-fachen Dauer resultiert. Alle Werte für die gegebene minimale Oktave, welche unterhalb von 6.0 cpsokt liegen, werden ignoriert.

→ Siehe Anhang S. 82

Es gibt zwei Funktionen um granulare Phrasen mit Pitchhits zu machen, *p-make-pitchhits-wolke-A* und *-B*. Sie beide bedienen sich der Funktion *p-make-pitchhits-wolke-abstr*. Die Funkiton *p-make-pitchhits-wolke-A* erstellt anhand dieser abstrahierten Funktion granulare Phrasen, bei denen die Wahrscheinlichkeit eines Pitchhits generell sehr gering ist und dies auch bleibt im Verlauf (Die Wahrscheinlichkeit wächst in Ihrem Verhältnis nur auf einen maximalen Wert von 0.1).
Bei der Funkiton *p-make-pitchhits-wolke-B* ist die beginnende Wahrscheinlichkeit sehr hoch und sinkt im Verlauf des Prozesses ab.

2.7.8.4. Convolution

Für den im Stück verwendeten Abschnitt der Convolution-Phrase wurden zunächst alle stimmlosen Abschnitte aller Ref-Samples in Events gewandelt. Zum einen wurde daraus eine Audiodatei erstellt (*DF_y_all_straigth_errs.aiff*). Zum anderen wurden aus diesen stimm-losen Events (*error-events*) und den im vorherigen Abschnitt definierten *TH1-events-shredded* und *TH2-events-shredded* das Ausgangsmaterial für eine sehr lange granulare Phrase, bestehend aus 6000 Events über 84 Sekunden. Diese wurde mit der Funkiton *p-make-pitchhits-wolke-B* erzeugt. Hierbei ist die Wahrscheinlichkeit für das Auftreten von „Pitchhits“, wie oben beschrieben, am Anfang sehr hoch und abfallend bis zum Ende.
Weiterhin, in der Erstellung des Materials für die folgende Convolution, wurden die einzelnen Fragmente des TH1 und des TH2 als separate Audiodateien ausgerechnet.

→ Siehe Anhang S. 107 (Aufrufe)

Die Audiodatei *DF_y_all_straigth_errs.aiff* wurde mit dem ersten Fragment des TH1 con-volviert und aus dem Ergebnis wurden die „Einatmungs“-Klänge entnommen und im Stück platziert.
Die Audiodatei *DF-END-to-conv-TH2frags.aiff* wurde convolviert mit jedem Fragment des TH2, daher auch der Name. Das Spektrum dieser Audiodatei eignet sich hierbei sehr gut, da es durch die „Pitchhits“ und die stimmlosen Fragmentabschnitte ein große Fülle enthält:

Die Convolution wird hier ausgeführt durch die Funktion *convolve-output-w-f,* welche die benötigten Pfade der Dateien definiert und diese, zusammen mit einem Faktor für den Pegel des Csound-outputs (*fkt*) und einem Namen, weiterreicht an die Funktion *convolve*.

Convolve erzeugt zunächst eine Score, welche dann mit dem Orchestra *convolution.orc* aufgerufen wird. Hierbei besteht die Möglichkeit die erstellte Audiodatei nicht in den Sound-output zu legen sondern an einen gewünschten anderen Ort (*out-dir*).

→ Siehe Anhang S. 83 (Funktionen), S. 113 (orchestra)

2.7.8.5. New-Events als neues Ausgangsmaterial

Für den letzten Teil des Stücks, welcher bis auf einen Klang aus dem fast kompletten Anfang des Stückes besteht, wurden die Funktionen *new-nevents-preparation* und *make-ne-ftables-as-file* erstellt, um diesen einen abweichenden Klang zu erzeugen.

Zunächst wurden mehrfach Klänge erstellt, die zusammengefasst und gestretcht den Klang *reprise-stretched* ergeben. Dieser verhält sich wie der lang gestretchte Klang der Exposition. Die Vorgehensweise, um den *reprise-stretched* zu erstellen, möchte hier beschreiben und dadurch die Funktionen der „New-Events“ erklären.

Erzeugung der Basisklänge

Mit der Funktion *convolve-f-w-f* wurden zunächst folgende Fragmente convulviert:
TH1-f3 mit TH2-f3, TH1-f3 mit TH2-f14, TH1-f5 mit TH2-f10 und TH1-f10 mit TH2-f14.

Mit der Funkiton *convolve-output-w-output* wurden die „multi-Klänge“ "TH1-f09-multi-to-rhythm-02.aiff" mit "TH2-f03-multi-to-rhythm-01.aiff" und "TH1-f09-multi-to-rhythm-02.aiff" mit "TH2-f09-multi-to-rhythm-01.aiff"convolviert. (Abschnitt granularisiertes Rauschen)

Mit der Funktion *convolve-f-w-output* wurden die einzelnen Fragmente mit folgenden „multi-Klängen“ convolviert:

Fragment:	Convolution mit:
TH2-f1	TH1-f01-multi-to-rhythm-01.aiff TH1-f01-multi-to-rhythm-03.aiff TH1-f02-multi-to-rhythm-03.aiff TH1-f06-multi-to-rhythm-03.aiff TH1-f08-multi-to-rhythm-03.aiff TH1-f09-multi-to-rhythm-03.aiff
TH2-f3	TH1-f06-multi-to-rhythm-03.aiff TH1-f07-multi-to-rhythm-03.aiff
TH2-f5	TH1-f04-multi-to-rhythm-03.aiff TH1-f05-multi-to-rhythm-03.aiff
TH2-f4	TH1-f09-multi-to-rhythm-03.aiff"
TH2-f7	"TH1-f03-multi-to-rhythm-03.aiff"
TH2-f8	"TH1-f08-multi-to-rhythm-03.aiff"
TH2-f9	"TH1-f06-multi-to-rhythm-03.aiff"
TH2-f13	"TH1-f05-multi-to-rhythm-01.aiff"
TH2-f14	"TH1-f09-multi-to-rhythm-03.aiff"

→ Siehe Anhang S. 110 (Aufrufe)

Die so erstellten Klänge wurden anschließend in den Ordner *New-events* verschoben und dort mit der Funktion *samples-norm* normalisiert auf 0 dB. *Samples-norm* verwendet das Kommandozeilen-Tool SoX mit dessen Normalisierungsfunktion.

→ Siehe Anhang S. 29/30 (s. Unten)

Nun wurde die argumentlose Funktion *new-nevents-preparation* aufgerufen, welche wie die Funktion *new-samples-preparation* funktioniert (s. Kapitel 2.6. Analysen und weitere vorbereitende Funktionen). Der Unterschied ist der Ordner und die Namensgebung. *New-nevents-preparation* definiert ein Samples aus dem Ordner *New-events* als *nevent-sample* = neventXXX-'name_endung', wobei XXX eine Stringnummer ist. Beispielsweise wird das File df_36_pe-stretch.aiff neudefiniert als nevent001-df_36_pe-stretch.aiff.
Alle Funktionen, die hier benötigt werden, verhalten sich wie Ihre Pendants für den Umgang mit den Ausgangssamples.

→ Siehe Anhang S. 51/52

new-nevents-preparation	entspricht	*new-samples-preparation*
make-ne-ftables-as-file	entspricht	*make-ftables-as-file*
new-events-samples-stream	entspricht	*soundfile-stream*
make-reference-event	entspricht	*make-reference-sample*
reference-event?	entspricht	*reference-sample?*
get-next-neventnr	entspricht	*get-next-samplnr*
ne-sample->event	entspricht	*refsample->event*
new-events-stream	entspricht	*soundfile-stream*

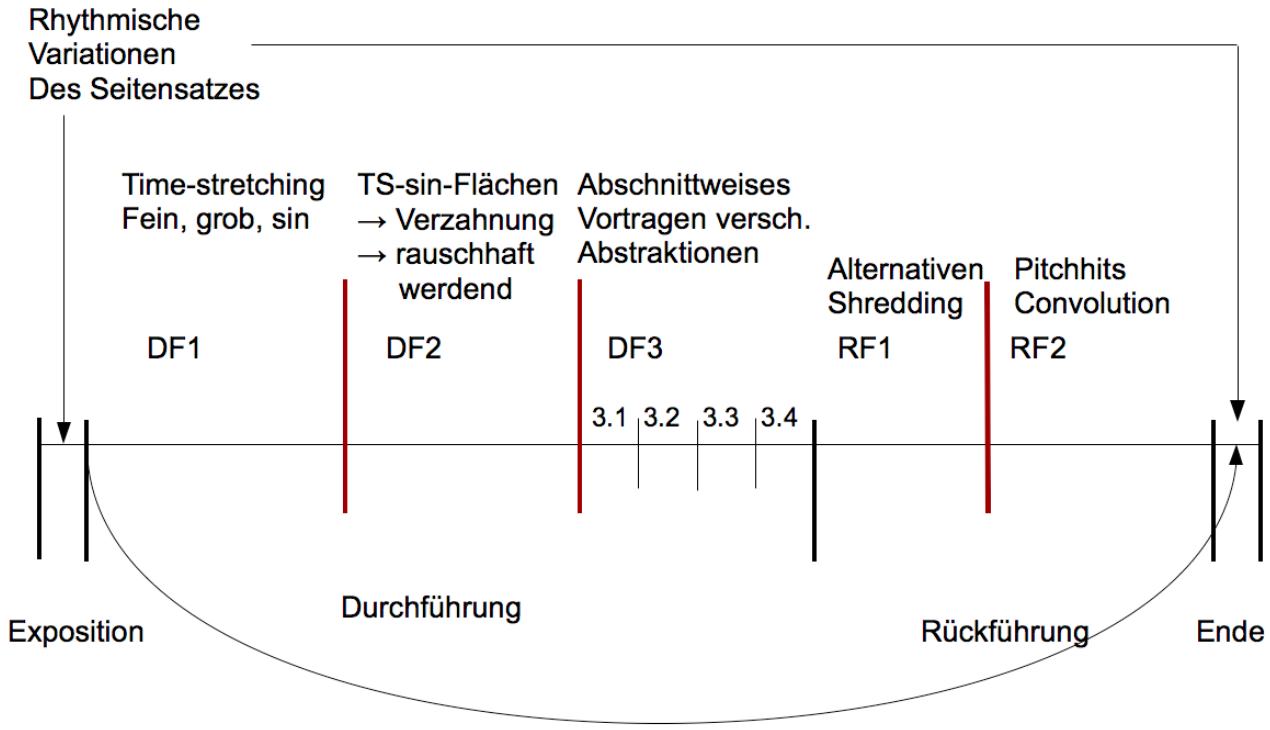
Die Funktion, die den Stretch erstellt, ist *new-events-stretch* und entspricht in etwa der Funktion, die den Stretchsound für die Exposition erstellt *fs-make-events-stretched* (s. Kapitel 2.7.2. Granulare Timestretch-Funktionen, S.20).
Im Gegensatz zu *fs-make-events-stretched* verwendet *new-events-stretch* keine Fragmente, sondern Events und erhält keinen Rhythmus-Stream. Gestrecht werden alle Events gleichzeitig bei einer Startzeit von 0.0s.

So werden also die Funktionen *new-nevents-preparation* und *make-ne-ftables-as-file* auf-gerufen, der *ne-stream* wird durch *new-events-stream* definert und die *stretched-events* werden erzeugt durch den Aufruf (*new-events-stretch ne-strm*)“ und gerendert mit Csound.

→ Siehe Anhang S. 110 (s. Unten)

3. Formverlauf und Struktur

Die Form des Stückes ähnelt, wie bereits innerhalb des Konzepts angegeben, einer Mischung aus „Sonatenhauptsatzform“ (im elektronischen Sinne) und Bogenform, wobei diese Mischung nur in den letzten Teilen stattfindet.



Das Stück ist grob unterteilt in 4 Abschnitte:

Teil 1 Exposition

Teil 2 Durchführung (DF)

Teil 3 Rückführung (RF) mit Rückblick,

Teil 4 Ende

Teil 2 und Teil 3 sind weiterhin unterteilt:

Teil 2, die Durchführung ist in 3 Teile Unterteilt, wobei der 3. Teil wiederum in 4 Teile unterteilt ist:

DF Teil 1 – Stretch-Variationen
DF Teil 2 – Stretch-Flächen und Überführung zur Rauschhaftigkeit
DF Teil 3 – Rausch-Variationen

- DF Teil 3_1 – Weißes und Rosa Rauschen - Variationen
- DF Teil 3_3 – Notches-Filterung – Variationen
- DF Teil 3_3 – Granulierte Rausch-Fragmente (Am weitesten entfernt vom Original)
- DF Teil 3_4 – Bandpass-Filterung – Variationen und ein Einschub der Fragment-Nachbildungen

Teil 3, die Rückführung mit Rückblick, ist in 2 Teile unterteilt:

RF Teil 1 Fragment-Ersatz und Granularisierung
RF Teil 2 Pitchhits und Convolution mit Rückblicken

Die Definitionen der einzelnen Teile zeigen auf, was hauptsächlich in den Teilen geschieht. Hinzu kommen immer wieder Vorausblicke oder Rückblicke, die hier noch nicht genannt sind.

Bei den folgenden Erklärungen verwende ich gerundete Zeiten, auf Sekunden gerundet.

3.1 Teil 1 Exposition

Die Exposition hat eine Dauer von 42". Hier werden das TH1 und das TH2 vorgestellt und das TH2 erfährt rhythmische Variationen.
Den Beginn macht der TH1 in ganzer Länge. Darauf folgt das TH2. Nach dieser kurzen Vorstellung folgen zwei geloopte Phrasen: 11 Wiederholungen der Fragmente „e“ „n“ „pif“ „tiff“ und parallel mehrfache Wiederholungen der ersten 3 Fragmente des TH2 („de de sn“), welche hier über knapp 9 Sekunden ineinander geschoben werden, d.h. die Einsatzabstände laufen gegen Null.
Im Anschluss folgen vier Phrasen des TH2 in Reihe, bei denen der Rhythmus durch die folgenden statischen Faktoren beeinflusst wurden:

1. Faktor 0.0 (Alle Fragmente gleichzeitig)
2. Faktor 0.25 (ohne die letzten beiden Fragmente „jüüü“ und „kaaaa“)
3. Faktor 0.5 (auch ohne die beiden letzten Fragmente)
4. Faktor 0,5 (diesmal mit allen Fragmenten)

Weiter geht es mit einem Doppelschlag Aller Fragmente gleichzeitig (Rhythmischer Faktor 0.0) und zudem dynamisch granular gestreckt (Streckung 1 → 6). Der erste Schlag des Doppelschlags ist kurz und mit einem Fade-out versehen, der 2. in ganzer Länge.
Die Exposition endet mit einer Wiederholung des TH1.

3.2. Teil 2 Durchführung

Teil 2 hat eine Gesamtdauer von 6'33" und beginnt bei 42". Der erste Teil (DF Teil 1) dauert hiervon 2'10" und überlappt sich mit dem zweiten (DF Teil 2), welcher bereits bei 1'44" beginnt, eine Dauer von insgesamt 2' 33' hat ' und bei 4' 19" endet. Teil drei (DF Teil 3) beginnt bei 4'19" und endet nach einer Dauer von 2'58" bei 7'17".

3.2.1. DF Teil 1 Stretch-Variationen

In diesem Teil werden ausschließlich Fragmente des TH1 variiert. Nach einem kurzen „Gebrutzel“ von stimmlosen Fragmentteilen beginnt die Durchführung mit einem Auftakt eines umgekehrten Sinus-Stretches des 1. Fragments (Fümms, bzw. „ssmssmmsms.. mümüüFF“). Darauf folgen Phrasen aus unterschiedlich zusammengesetzten gestretchten Fragmenten, jeweils mit einer kurzen Pause unterbrochen.
Ab 1'44" setzt bereit DF Teil 2 ein, mit der Sinus-gestretchen Fläche des ersten Fragments „fümms“. Diese „atmende“ Fläche bleibt nun stehen bis zum Ende von DF Teil2.
Parallel dazu folgen noch weitere Phrasen, in denen die Fragmente allesamt übergehen in lange gestretchte Sinus-Stretches welche die Einleitungen zu den „atmenden“ Sinus-Flächen bilden, die in DF Teil 2 weitergeführt werden.

3.2.2. DF Teil 2 Stretch-Flächen und Überführung zu Rauschhaftigkeit

Dieser Teil besteht aus den genannten Sinus-Stretch-Flächen. Der gesamte folgende Prozess wurde bereits in Kapitel 2.7.4. Prozess „Sinus-Stretched“-Flächen/Verzahnung/Rauschhaft werdend, S. 29, beschrieben.
Die einzelnen Flächen variieren in ihrer Lautstärke, sodass mal die eine und mal die andere in den Vordergrund tritt.
Der Prozess der „Verzahnung“ der einzelnen Grain-Streams, aus denen die Flächen bestehen, beginnt im Stück ab 3'49" und endet bei 4'7". Von hier ab läuft nur noch der resultierende, verzahnte Grain-Stream mit allen Beteiligten, welche nun die weiteren Modulationen erfährt.
Ab 4'7"" werden die Einsatzabstände und Dauern der Grains größer und beginnen rauschhaft zu werden. Die volle Rauschhaftigkeit, vergleichbar mit einem weißen Rauschen, wird erreicht bei 4'18", wenn DF Teil 2 endet.

3.2.3. DF Teil 3 Abschnittweises Vortragen verschiedener Rausch-Variationen

Teil 3 der Durchführung ist wie oben beschrieben in 4 Abschnitte unterteilt, die zusammen die Dauer von 2'58" ergeben. DF Teil3 beginnt nach einer kurzen Pause bei 4'19".
Hier taucht immer wieder der Klang „TH1-f01-multi-02.aiff“ auf, welcher die einzelnen Abschnitte von einander trennt. Waren es in DF Teil 1 die Pausen, so ist es hier dieser Klang, welcher sich extrem vom Ausgangsmaterial entfernt hat.

DF Teil 3_1 dauert 31', DF Teil 3_2 dauert 43", DF Teil 3_3 dauert 48" und DF Teil 3_4 weitere 53".

3.2.3.1. DF Teil 3_1 weiße und rosa Variationen

Dieser Teil dient als Einführung in die weiteren Abstraktion der Fragmente durch Rauschhaftigkeit. Den Beginn macht erneut das komplette TH1 in weiss, über das volle Spektrum mit einer Verstärkung im Bassbereich. An ihn dran gehängt wurden die letzten beiden Fragmente auch in weiß, jedoch rückwärts, was einer aus dem Turntablism stammenden Scratch-Ähnlichkeit entspricht. (Wenn ich hier von weiss oder rosa spreche, betrifft dies das Spektrum an transponierten und addierten Kopien und die Amplitudenwerte. Der Faktor betrifft das Maß an Rauschhaftigkeit.)

Nach dieser kurzen Vorstellung des weißen TH1 folgen weitere Vor- und Zurückbewegungen. Beginnend mit dem TH1 in rosa (Fkt 75), gefolgt vom TH2 (rosa 0.75) und wieder dem kompletten TH1. Weiter geht es mit Vor- und Zurückbewegungen der Fragmente 4 und 5 (tä und ze) in rosa, 0.75. Danach folgen einzelne Fragmente des TH1 in rosa, 1.0: „Fümms, bö, wö, Füm, Füm, bö, wö, tä, ze, uu“ und das TH1 von Fümms bis Gif.
Mitten im „f“ des Gif's Beginnt eine Reihe von 100 „bö“s in pink die in ihrem Rauschfaktor ansteigen von 0.02 bis auf 1.0 und dabei leiser werden. Parallel startet eine zweite Reihe aus 40 Fragmenten („nn“ des TH2), auch aufsteigend in rosa von 0.02 bis 1.0. Diese Reihe wird langsam über 20 Fragmente eingeblendet, sodass eine Überblendung der beiden Reihen stattfindet.
Ist die letzte Reihe vollendet ,folgt der Titel des Stücks „Rrumpff tilllff toooo?“ in verschieden zusammenaddierten transponierten Kopien, vorwiegend bass lastig.

DF Teil 3_1 endet mit dem bereits erwähnten Klang „TH1-f01-multi-02.aiff“, welcher durch das Durchfahren der verschiedenen weißen Kopien des Fragments „fümms“ des TH1 besteht.

3.2.3.2. DF Teil 3_2 Notches-Filterung – Variationen

Dieser Teil besteht fast komplett aus den „DF-X-f..-notcheratz“ - Klangphrasen, welche den Rhythmus des TH1 verwenden, aber die Fragmente des TH2 als Ersatz.
Markant sind hierbei die Ersetzungen des letzten Fragments „fümms“ durch das notch-filtrierte weisse Fragment „Kaaa“ aus dem TH2. Mit diesem Klang in verschiedenen notch-Filtrierungen geht es nach den kompletten Phrasen weiter. Hierbei nehmen die Einsatzabstände ab, sodass sich eine Spitze der weißen Notch-“Kaaas“ hochschaukelt um anschließend wieder abzubauen. Diese Stauchungs- und Dehnungs-Funktion findet über 14 verschiedene Kopien statt. Abschließend folgt hier eine umgekehrte Kopie, so wie das Zurückziehen einer Schallplatte.

Der Abschnitt DF Teil3_2 wird letztendlich abgeschlossen durch eine erneute Grain-Stream-Modulation. Ähnlich der des Abschlusses von Teil 2 der Durchführung.

3.2.3.3. DF Teil 3_3 weit entfernt vom Ausgangsmaterial

Der dritte Teil der Rauschvariationen stellt den äußersten Rand der Bogenform dar. Hier erscheinen Klänge, die kaum noch an das Ausgangsmaterial erinnern.
Zu hören sind hier eher stark maschinenhafte und elektronische Klänge, die ineinander überfließen. Aus einem der „TH1-f09-multi-xvz-01.aiff“-Klänge, welcher DF Teil 3_3 be-ginnt, wurde eine markante Passage entnommen, die innerhalb dieses Teils noch 4 mal zu hören ist und somit einen Faden durch die abstrakte Elektronik spinnt.

3.2.3.4. DF Teil 3_4 weitere Variationen: Bandpasses und Nachbildungen

Es beginnt allmählich die Rückführung. In diesem Teil der Durchführung erklingen auch sehr entfremdete Fragmente, welche aber im Gegensatz zu den Klängen des letzten Teils stärker, wieder auf das Ausgangsmaterial verweisen. Diese Klänge wurden erstellt durch das Durchfahren von

durch Bandpass-Filter-Funktionen generierten Kopien des Sprachmaterials.
Der Klang erinnert auch hier stark an das Scratching mit Schallplatten, wobei hier keine Umkehrungen stattfinden. In diesem Teil folgen Fragmente aufeinander, die, innerhalb des Spektrums, zum Einen von Hoch nach Tief durchfahren wurden und zum Anderen von Tief nach Hoch, sodass verschiedene Arten von Sweep-artigen Klängen erklingen.

Am Ende des ersten Drittel dieses Teils ist das komplette TH1 in nachgebildeten Fragmenten zu hören.

Weitere Klangphrasen, die hier Einzug fanden, sind stimmlose und zum Ende des Teils auch stimmhafte Sprachschnipsel aus dem Ausgangsmaterial. Der Teil endet und somit endet die komplette Durchführung mit einer Anhäufung dieser zuletzt erwähnten Schnipsel, stimmhaft und stimmlos.

3.3. Teil 3 Rückführung (RF) mit Rückblick

Teil 3 des Stücks ist unterteilt in 2 Abschnitte, RF Teil 1 und RF Teil 2. Die Gesamtdauer beträgt 2' 56". RF Teil 1 dauert hiervon 1'9"' und RF Teil 2 dauert 2'13". Teil 3 beginnt bei 8'12".

3.3.1 RF Teil 1 Fragment-Alternativen und -Zerlegung

In diesem Teil kommen die Funktion des Fragment-Pools zum Tragen (s. Kapitel 2.7.8. Prozess Fragment-Alternativen-Setzung, S. 35). Hier erklingen nun die originalen und alternativen Loops, wobei TH1 und TH2 im Wechsel erscheinen und die ersetzten Fragmente recht schnell ihre Granularisierung erfahren.
Bei dem Finalisieren des Stückes wurden zudem einzelne Rauschkopien des TH1 und transponierte Kopien der Fragmente zwischen die ersten 3 Wiederholungen der alternativ, beziehungsweise original werdenden Sätze gesetzt.
Zum Ende des ganzen Prozesses bildet sich aus den zu kleinen Soundschnipseln zerlegten Fragmenten die lauter werdende Grain-Wolke. Diese erfährt zum Ende des ersten Drittels des RF Teil1 hin, diverse Pitchshifts. Die Wolke endet abrupt am Anfang des letzten Drittel des Teils, wird quasi unterbrochen von der wiederkehrenden Frage „Rrumpff tillff toooo ? “ und geht mit einem

Schlag weiter. Nun aber nicht mehr mit pitchshiftings sondern mit Pitchhits der Grains.
Die nun durch Pitchhits erweiterte Wolke geht weiter und klingt bis zum Ende des Teils aus.
Parallel zum Ausklang beginnt bereits der Auftakt des nächstes Teils mit einem umgekehrten Pitchhit-Fümms.

3.3.2. RF Teil 2 Pitchhits und Convolution-Phrase mit Rückblick

„Aufgetaktet“ durch den umgekehrten Pitchhit erklingt das komplette TH1 in Pitchhits und sorgt durch die klangcharakteristische Versinkung in der Tiefe für ein Einkehren von Ruhe, nach der konfusen granularen Phrase des letzten Teils.
Anschließend wird fünfmalig „Luftgeholt“ um die letzte Hürde zu meistern.

Nach dem ersten, dem dritten und dem vierten Luftholen wird bereits die Lange granulierte Pitchhits-Phrase kurz angespielt. Nach dem fünften Luftholen setzt sie dann ganz ein und wird direkt zu ihrer convolvierten Kopie (mit dem Fragment 1 des TH2, „de“) übergeblendet.

Im Folgenden verlaufen die convolvierten Pitchhit-Granular-Phrasen im Wechsel und in langen Überblendungen. Zwischendrinn taucht immer wieder die ausgehende Pitchhit-Granular-Phrase auf. Die auf diese Weise kombinierte komplette Fläche verläuft bis zum Ende der RF und somit parallel zu dem Auftauchen bereits vertrauter Klänge aus der Vergangenheit des Stücks.

3.4. Teil 4 Ende - Anfang

Als Reprise-artigen Schlag erklingt ein aus vielen Klängen des vergangenen Stücks addierter und gestretchter Doppelschlag (s. Kapitel 2.7.8.5. New-Events als neues Ausgangsmaterial, S. 42), so wie er am Anfang in der Exposition zu hören ist. Dort mit Fragmenten des TH2. Es folgt die fast komplette Exposition (21" – 40"), sodass das Stück endet wie es begonnen hat: mit dem originalen TH1.