

# Konzeption erster Synthesizer

Stand: 5.4.2016

Idee:

- einfacher Synthesizer mit einem Oszillator
- Oszillator erzeugt Rechteck
- wird geroutet auf einen Filter
- Filter hat Cutoff und Resonanz
- wird geroutet auf einen EG (ADSR)
- wird geroutet auf einen Amplifier
- wird geroutet auf einen Effekt
- erzeugt nur einen Ton mit einer Frequenz
- kann durch Midi ausgelöst werden
- kann durch OSC ausgelöst werden
- noch keine grafische Oberfläche

Implementationsdetails

- Grundkomponenten: STK, oscpack, RTMidi, RTAudio
- Ausgangspunkt ist Beispiel aus STK:

# 52 Dinge, die man in einem Forschungssemester lernt

Jan-Torsten Milde, beginnend am 8.4.2016

1. C++ ist Scheiße.
2. genauer: der C-Teil von C++ ist Scheiße.
3. Eine statische Variable muss ZWEIMAL deklariert (!!!! Unfassbar, ich kenne keine andere Programmiersprache, in der eine doppelte Deklaration erlaubt wäre) werden: in der .h und der .cpp, andernfalls gibt es einen Compilerfehler.
4. Private, verschachtelte (nested) Klassen haben KEINEN Zugriff auf die sie umgebende Klasse
  1. Dieser Zugriff kann auch nicht über eine Referenz erfolgen, da Referenzen initialisiert werden müssen (GUT SO!), aber sie können nicht mit NULL initialisiert werden.
  2. Also bleibt nur der Pointer, und damit das Unterlaufen aller Sicherheitsmechanismen von C++.
5. Für die vernünftige Erzeugung einer Übersetzungsumgebung auf dem Raspberry, ist es notwendig, sich in automake, autoconf etc einzuarbeiten. Andernfalls wird die Compilierung via Makefile schnell zu komplex (also, wenn man das Ziel hat, möglichst systemübergreifenden Code zu schreiben).
  1. Die Syntax dieser Werkzeuge ist mindestens verwirrend ... die Dokumentation zwar da, aber man muss doch erheblich suchen.
  2. Nach ca. 3 Stunden hat man dann etwas gebastelt, was einem die Übersetzung ermöglicht.
  3. Ich habe 2 Shellscripate angelegt, die die erneute Erzeugung von configure durchführen, das Erste räumt auf, das Zweite übernimmt die Änderungen an der Makefile.am (automake Template zur Erzeugung des eigentlichen Makefiles via configure).
  4. Während der anfänglichen Entwicklungsphases muss dieser Schritt immer wieder durchgeführt werden: das ist ok, man kann über die Modularisierung seiner Software nachdenken
6. Der Raspberry Pi 3 ist mehr als schnell genug, um darauf aktiv zu entwickeln
  1. Trotzdem: ich vermisse meine Entwicklungsumgebung (eclipse), das schafft er nicht.
  2. Man muss also arbeite, wie vor 20 Jahren: shell und emacs.

3. Und als persönlicher Vergleich: große Teile meiner Promotionsarbeit wurden ab 1992 auf einer Sparc Station 10 (später dann auch Sparc Station 20) durchgeführt. Themengebiet KI und Sprachverarbeitung. Ein Wahnsinnsrechner, echt irre. Der Prozessor wurde mit 40 Mhz (!) getaktet und man konnte tatsächlich 512 Mbyte Hauptspeicher einbauen, wenn man denn genug Geld dafür hatte. Und er kam in dieser coolen Pizza-Box.
4. Der Raspberry ist also 25 mal höher getaktet, hat 4 Prozessorkerne statt nur einem, Hardwarunterstützung für Floating-Point Operationen, und doppelt soviel Hauptspeicher. Und er läuft mit einem 2W Netzteil; notfalls mit einem Handyakku. Und man kann mit ihm eine LED zum Blinken bringen, echt cool.
7. Ach übrigens: der C-Teil von C++ ist Scheiße: es gibt 4 Varianten von Strings in C++ und keine ist mit der anderen kompatibel. Jedenfalls, soweit, wie ich das bis jetzt rausbekommen habe.
  1. Ich will doch einfach nur einen String an eine Funktion übergeben, trotzdem gibt es Compilerfehler.
  2. Und das liegt daran, dass die unterliegende Implementierung C-Strings verwendet ... also muss ich meine schönen C++-Strings in C-Strings wandeln (mit `.c_str()` )
8. Hat ein bisschen gedauert, bis mir ein passender Name für die Engine eingefallen ist. Jetzt ist mir klar, es kann nur **Neonlicht** sein :)
  1. Warum das ? Einfache mal anhören: <https://www.youtube.com/watch?v=iPDCiHLsFMU>
  2. Das Lied ist aus dem Jahre 1978 (!) Karl Bartos war ein Melodiegenie, die ganze Melancholie der tristen Endsiebziger in Düsseldorf/Dortmund/Detmold wird weggezaubert durch ein bisschen farbiges Kunstlicht.
  3. Ein Popsong von mehr als 8 Minuten, ohne Strophe und zwei Soli (?) von mehr als 6 Minuten, Ostinatobass, maximal reduzierter Elektrobeat mit einem unfassbaren Groove, "künstlichem" Chor und schimmerdem Klangteppich. So etwas wird heute ja gar nicht hergestellt !
9. Ein Song mit Refrain und ohne Strophe, das geht ja noch, in jedem Fall besser als ein Song nur mit Strophen und **OHNE** Refrain. Sowas funktioniert niemals, die Leute wollen mitsingen. Es sei denn, man heisst Kraftwerk.
  1. Der Lied heisst "Das Modell" und ist Kraftwerks größter internationaler Hit (!) <https://www.youtube.com/watch?v=OQIYEPe6DWY&nohtml5=False>
  2. Ich frage mich, wenn Kraftwerk die Kommerzialisierung und Oberflächlichkeit der Gesellschaft bereits 1980 als so extrem wahrgenommen haben, wie würde ihr musikalischer Kommentar auf den Zustand der aktuellen Welt heute ausfallen ?
  3. Aktuelles Konzert: <https://www.youtube.com/watch?v=xM6B5j378T8>

10. Nur so für die Technikhistoriker: 1978 war das Jahr 1 des Apple II und des Commodore PET 2001, drei Jahre vor dem ersten IBM PC, vier Jahre vor dem Commodore C64 und dem ZX Spektrum.
  1. 1981, mit dem ersten PC, brachte Kraftwerk das Album *Computerliebe* heraus. Sie hatten definitiv ein gutes Gespür für den Geist der Zeit.
11. Ich habe schrittweise die Systemarchitektur des Gesamtsystems festgelegt. Es scheint so zu sein, dass die Architektur der eigentlichen Syntheseengine (also von Neonlicht :)) relativ einfach ist.
  1. Im Kern läuft eine Endlosschleife, die kontinuierlich die Audio-Hardware auf (Daten)-Bedarf abfragt und dann entsprechend die statische Methode *tick()* aufruft.
  2. Die statische Methode *tick()* hat Zugriff auf den Audiostream und kann so Daten aus der Audiohardware lesen und in die Audiohardware schreiben
  3. An dieser Stelle klinkt sich die eigentliche Synthese ein: der zentrale Datenspeicher der Applikation (*CentralStore.cpp*) verwaltet die nutzerdefinierten Syntheseeinheiten und ruft deren *tick()* Funktion auf.
  4. Innerhalb der Syntheseeinheiten werden die einzelnen *UGens* mit ihren *Ports* definiert und das *interne Routing* zwischen Ihnen festgelegt. Dies erfolgt durch den Anwender.
  5. Damit die Syntheseeinheit von außen konfiguriert werden kann, müssen die internen Parameter der Einheit bekannt gemacht werden. Zur Zeit habe ich noch keine definitive Lösung für diese Aufgabe (Aufgabe, kein Problem, ich muss mich einfach entscheiden, welche der möglichen Lösungen ich für die Beste halten).
12. Die Syntheseeinheiten folgen dem bekannten UGen-Ansatz von Max Mathews:
  1. siehe [https://en.wikipedia.org/wiki/Unit\\_generator](https://en.wikipedia.org/wiki/Unit_generator)
  2. Die Klasse *UGen* definiert die *tick()*-Funktionalis die eigentliche Funktion zur Verarbeitung von Audiodaten
    1. Bei jedem *tick()* werden die aktuellen Inputdaten der Ports ausgelesen, verarbeitet und in den Outputports gespeichert.
    2. Jeder *UGen* kennt seine Input/Output Konfiguration, weiss also welche Daten in welchen Ports zur Verfügung stehen (müssen)
  3. Jeder *UGen* trägt eine Identifikation
  4. Jeder *UGen* kann eine unbestimmte Anzahl von Ports enthalten, über die Daten gelesen und geschrieben werden können
    1. Die Ports werden über ein Array verwaltet
  5. Die *Ports* werden unterschieden in Input- und Outputports (und Valueports) und halten jeweils einen Wert. Sie verfügen über eine Identifikation.

6. An einen Inputport kann jeweils nur eine Verbindung geschaltet werden (ein "Patchkabel"). Sollen mehrere Werte kombiniert werden, muss ein Mixer-UGen vorgeschaltet werden.
  1. Ich weiss, das ist sehr klassisch, ähnlich wie Doepfer in seinen ersten Systemen ...  
<http://www.doepfer.de/home.htm>
7. An einen Outputport können beliebig viele Verbindungen geschaltet werden.
8. Es wird nicht unterschieden zwischen Controlports und Datenports. Hier orientiert sich die Implementierung konzeptionell an den realen analogen Systemen (und ich hoffe das klappt auch so ????)
9. Das Routing der Daten wird für die gesamte Einheit (**SoundUnit**) in einer Routingtabelle (**RoutingTable**) festgelegt.
10. Die Einheit legt auch die zeitliche Abfolge, also die Sequenz der **UGens** fest. Dies wird durch die **tick()**-Routine der Einheit geleistet.
13. Eine **.o Datei** darf in einem Target im Makefile.am nur genau einmal auftauchen, sonst gibt es "multiple defined" Fehlermeldungen
14. Und wieder mal eine C++ Überraschung: ich wollte die **UGens** in einer map speichern und dann natürlich Polymorphie ausnutzen. Also abgeleitete Objekte (REFERENZEN!) in der Map speichern. Klappt auch ohne Probleme. Bis man versucht die Objekte auszulesen.
  1. C++ kennt offenbar keine Möglichkeit Objektreferenzen auf den abgeleiteten Typ zurückzuführen, z.B. per cast. Was mal wieder nur bleibt, ist die Speicherung von Pointern ... dann kann man Objekte in der abgeleiteten Form nutzen. Allerdings bekommt man auch wieder Spitzen **Segmentation faults**, wenn der Pointer ins Nirvana zeigt !
  2. Was nützt mir ein objektorientiertes Deckmäntelchen, wenn man die wesentlichen Vorteile des Modellierungsansatzes nicht, bzw. nur unter hohen Schmerzen , nutzen kann ?
  3. Ich sage nur: siehe Punkt 1
15. Ok, ich habe heute (12.4.2016) den folgenden Stand erreicht:
  1. **Neonlicht** ist lauffähig, die Engine synthetisiert Sound (Hurra!)
  2. Eine Reihe von einfachen UGens sind implementiert: **NoiseGen** (weisses Rauschen), **SawGen** (Sägezahngenerator, Interpolation, kein Wavetable), **OnePoleLPFGen** (einfacher Low Pass Filter mit Kontrolle der Cutoff-Frequenz), **TwoInputMixer** (ein Mixer, der 2 Inputs abmischt).
    1. Jede Menge Code-Beispiele für die Implementierung von Audiofiltern findet man z.B. hier: <http://musicdsp.org/archive.php?classid=3>

3. Damit sind die zentralen Elemente eines Synthesizers verfügbar: Oszillatoren, Filter und Verstärker.
4. Es fehlen u.a. noch:
  1. *Envelopegenerator* (EG) als Spezialfall des Oszillators,
  2. *Low Frequency Oscillator* (LFO) als Spezialfall des Oszillators,
  3. *Effekte* (Delay, Reverb), als Spezialfall der Filter und
  4. die *Modulationsmatrix* zur interaktiven Definition des Routings
16. Ich habe die Schnittstelle von *UGen* und *SoundUnit* doch um eine *control()*-Methode erweitert
  1. Eigentlich wollte ich die reine Lehre: alles ist Spannung :)
  2. Aber dann wird der Code zu komplex und undurchschaubar
  3. Mir ist wichtig, dass die Anwender (u.a. Studierende) schnell und einfach zu Ergebnissen kommen können, entsprechend vereinfache ich die Nutzung durch die Hilfsroutine
  4. *control (std::string portName, float value)* erlaubt das Setzen von Werten. Innerhalb der Routine kann/muss der Anwender die Nachrichten dispatchen, also entsprechend in dem *UGen* oder der *SoundUnit* verteilen.
    1. Angestoßen wird das Ganze im Moment noch in der tick()-Routine von Neonlicht
      1. Das kann nicht so bleiben, ansonsten muss der Anwender in die Engine rein, und das ist absolut verboten (!)
      2. Da das Messagehandling im *CentralStore* erfolgt, werde ich das Ganze wohl so regeln, dass Neonlicht den *CentralStore* antickt, dann die Nachricht holt und sie unverändert an die *SoundUnit* weiterleitet
        1. Hier kann dann der Anwender in seinem Code (und natürlich auch die Anwenderin in ihrem Code) die Nachricht abarbeiten
17. Um eine eindeutige Identifikation der *UGens* und *Ports* zu gewährleisten, habe ich einen *ID-Generator* implementiert.
  1. In UGens und Ports wird jetzt unterschieden zwischen
    1. *Name*, frei wählbar vom Anwender, aber nicht unbedingt eindeutig, und
    2. *ID*, bestehend aus Name und dem automatisch generierten eindeutigen ID-String
  2. Für beider Felder sind entsprechende Getter-Methoden erstellt worden
18. Um das Austesten von Neonlicht zu vereinfachen habe ich einen (sehr) einfachen Sequenzer implementiert

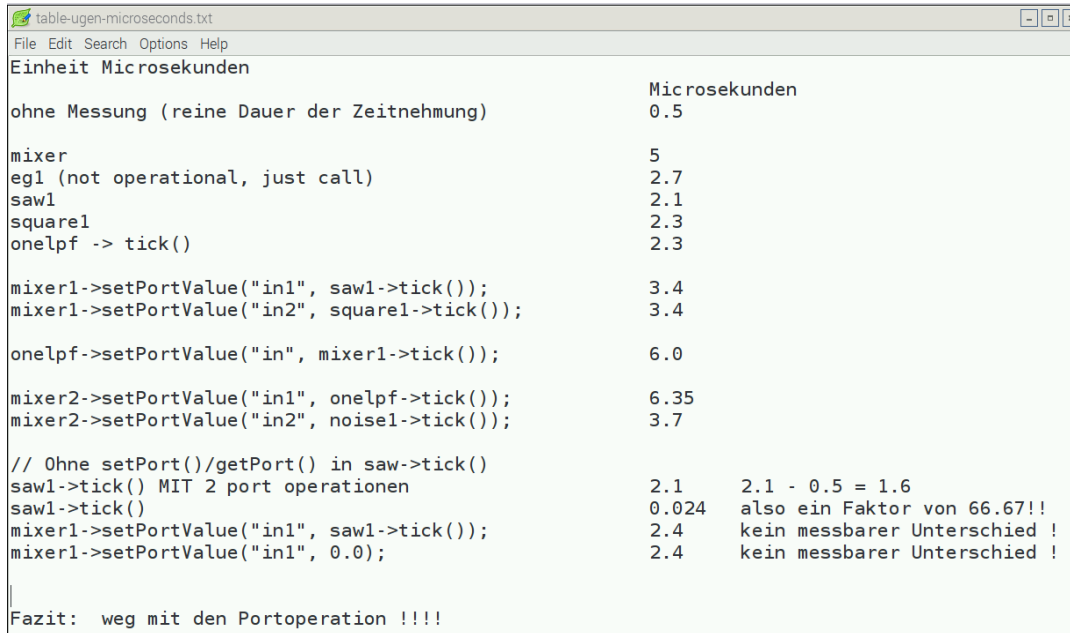
1. Als Name wähle für den Sequenzer wähle ich *Fliehkraft* :)
2. Fliehkraft generiert automatisch Midimessages und versendet diese zeitgesteuert per *OscOutConnector*
  1. Im Prinzip also nur ein “automatisches” Keyboard
3. Die Grundlage des Sequenzers ist ein einfacher Thread, der über die *<chrono>* Bibliothek auf hochgenaue Zeitmessungen zugreifen kann
4. Damit wird es möglich, die Midimessages zu sehr exakt definierten Zeitpunkten zu versenden
5. Im ersten Ansatz bleibe ich dabei rhythmisch im westlichen Popraster: der Sequenzer generiert Ganze, Halbe, Viertel, Achtel, Sechszehntel und Zweiundreissigstel
6. Der Sequenzer ist “ungenau”: die kleinste Zeithinheit wird auf einen *int*-Wert gecastet, alle größeren Einheiten sind ganzzahlige Vielfache hiervon
  1. Damit kann die Geschwindigkeit des Sequenzer nicht auf jede beliebige BPM gestellt werden
  2. Ich weiss, ich weiss, das ist nicht gut, aber im Moment brauche ich das Ding nur zum Rumdudeln, wenn mehr Zeit da ist, überarbeite ich das Konzept
19. Ok, Fliehkraft läuft, und zeigt auch direkt die nächste Schwachstelle auf: aktuell werden alle Nachrichten auf einem Osc-Kanal (Standardwert ist “*localhost*”, *Port 7000*) gesendet und entsprechend durch Neonlicht empfangen.
  1. Die Nachrichten werden ordnungsgemäß in die Warteschlange eingetragen und nacheinander abgearbeitet
  2. Das führt jetzt dazu, dass ein schnelles Drehen am Knopf des Midi-Controllers (Arturia MiniLab, <https://www.arturia.com/minilab/details>) so viele Nachrichten erzeugt, dass die Wiedergabe der Noten, die Fliehkraft generiert, unterbrochen wird.
    1. Schnelles Drehen am Controller unterbricht also die Wiedergabe
  3. Also müssen Control Nachrichten auf einem eigenen Kanal gesendet werden, der unabhängig von den Notendaten ist, welche Priorität besitzen
20. Formel für das lineare Mapping von Wertebereichen:
  1.  $\text{map}(v) = ((v-s1)/(e1-s1) * (e2-s2)) + s2$
  2. Achtung: e1 und s1 dürfen nicht gleich sein, sonst kracht es (das Ausgangsintervall muss eine Länge ungleich 0 haben).
  3. Nachtrag: ich bin prompt reingefallen! Ich habe die Funktion überladen und so für *int* Eingabeintervalle verfügbar gemacht und dabei vergessen, die Division auf *float* umzustellen, was natürlich zu falschen Ergebnissen (sprich immer 0) führt.

21. Ich war heute (14.4.2016) auf der Verteidigung einer *halben* Dissertation

1. Die Schweden haben eine mögliche Zwischenprüfung auf der halben Strecke zum PhD eingeführt.
2. Davon dann später mehr (siehe Punkt ?????)

22. Das System ist zu langsam, schon jetzt kommen hörbare Glitches auf ... MIST

1. Das kann doch echt nicht wahr sein. Habe ich mich so vertan ? Woran liegt es ? Zeit mal ein paar Messungen zu machen



```
table-ugen-microseconds.txt
File Edit Search Options Help
Einheit Microsekunden

ohne Messung (reine Dauer der Zeitnehmung)      Microsekunden
                                                0.5

mixer                                             5
egl (not operational, just call)                 2.7
saw1                                              2.1
square1                                          2.3
onelpf -> tick()                                2.3

mixer1->setPortValue("in1", saw1->tick());        3.4
mixer1->setPortValue("in2", square1->tick());      3.4

onelpf->setPortValue("in", mixer1->tick());        6.0

mixer2->setPortValue("in1", onelpf->tick());        6.35
mixer2->setPortValue("in2", noise1->tick());      3.7

// Ohne setPort()/getPort() in saw->tick()
saw1->tick() MIT 2 port operationen             2.1      2.1 - 0.5 = 1.6
saw1->tick()                                     0.024    also ein Faktor von 66.67!!
mixer1->setPortValue("in1", saw1->tick());         2.4      kein messbarer Unterschied !
mixer1->setPortValue("in1", 0.0);                 2.4      kein messbarer Unterschied !

Fazit: weg mit den Portoperation !!!!
```

2. Wie man sieht, funktioniert das Konzept der dynamischen Wertzuweisung auf den Ports nicht.

1. Sie sind mindestens um den Faktor 25 zu langsam !
2. Und wieder mal trifft die Realität auf Wunschdenken: die Modellierung war schön, aber das funktioniert hier leider nicht :)

23. Ok, jetzt (15.4.2016, 0:19 Uhr :)) ist die Umstellung erfolgt

1. Im aktuellen Prototypen werden die Portoperation nicht mehr verwendet
2. Im Prinzip hat sich die Schnittstelle für den Anwender nur unwesentlich verschlechtert
3. Insgesamt kann sogar WENIGER Code geschrieben werden, allerdings ist der Code etwas umständlicher und anfälliger für Programmierfehler
  1. Vorher wurden Ports benannt, jetzt muss der Entwickler entscheiden, welchem Port er welchen Parameter zuordnet
4. Resultat der Umstellung: die Engine ist um einen Faktor > 30 schneller !



5. Na also, geht doch :)
24. Habe mit *std::strings* in den Gettern/Settern von Amnt1 und Amnt2 experimentiert
1. Effekt: die Machine wird durch den Stringvergleich **17(!) mal** langsamer
  2. Also: keine Strings in den zeitrelevanten Bereichen !
25. Die Oszillatoren *knacksen* beim Umschalten der Frequenz
1. Ist auch kein Wunder: im Moment schalte ich irgendwo in der Schwingung einfach die Frequenz um, berechne den Index neu und entsprechend daraus den neuen Samplewert
  2. Da sind Sprünge in der Werten natürlich unvermeidbar und das gibt so ein schönes Knacksen :)
    1. ich brauche also einen sanften, aber schnellem Übergang auf die neue Frequenz
    2. Vielleicht einen Kurzzeitinterpolator ? So über 5 Werte vielleicht ? Hmm, gar nicht so ganz einfach ... da muss ich überlegen
      1. Ne, die Lösung ist: Umschalten im Nulldurchgang, also Verzögerung um ein "paar" Samples, bis die Schwingung im Nullschurchgang ist und dann die Frequenz wechseln (denn  $3 * 0$  is 0, is 0 [https://www.youtube.com/watch?v=4d\\_iiOcNCyU](https://www.youtube.com/watch?v=4d_iiOcNCyU), oder auch [https://www.youtube.com/watch?v=BZQA\\_mEzVxs](https://www.youtube.com/watch?v=BZQA_mEzVxs) (ein bischen besser verständlich für Ostwestfalen))
    3. Es ist 2.32 Uhr ... es reicht, ich muss ins Bett, definitiv :)
26. Wieviel der unterliegenden Implementation von STK soll ich übernehmen ?
1. Ich versuche zu entscheiden, welche Basis-Unitgeneratoren als Teil von Neonlicht implementiert werden sollen.
  2. Wenn man sich beispielsweise *Chuck* anschaut (<http://chuck.cs.princeton.edu/doc/program/ugen.html>), das ebenfalls auf dem STK aufsetzt, so erkennt man, dass hier praktisch alles übernommen wurde und lediglich eine schmale Schicht drüber gelegt worden ist. Und natürlich wurde ein Interpreter mit einfacher VM implementiert, der die Bedienung "erleichtert"
    1. Das gefällt mir nicht so richtig, ich will SELBER verstehen, wie die UGens funktionieren und wie sie zu implementieren sind
    2. Andererseits: wenn die Implementierung schon da ist, ist das im Grunde genommen schwachsinnig
  3. Ich werde Folgendes machen: für einen Teil der STK UGens baue ich eine Abstraktion in meinen UGen-Mechanismus ein, sodass diese dann transparent nutzbar sind
    1. Allerdings nur dann, wenn die Performanz stimmt (wass allerdings zu erwarten ist, da die Implementierung in STK sehr effizient ist)

4. Komplexere und höherwertigere UGens implementiere ich unmittelbar

27. Unit Generatoren, die Neonlicht in jedem Fall umsetzen muss:

Name	Funktion		
SinOsc			
PulseOsc			
SqrOsc		ok	
TriOsc			
SawOsc		ok	
Phasor		ok	
Noise		ok	
Impulse			
Step			
Gain			
SndBuf			
HalfRect			
FullRect			
ZeroX			
Mix2			
Pan2			
GenX			
CurveTable			
WarpTable			

28. Und diese Filter sollten da sein:

Name	Beschreibung	Parameter	Anm.	Mapping
OneZero	STKOneZeroGen	zero := amnt1	LPF HPF	[0,1] → [-1,1]
TwoZero	STKTwoZeroGen	frequency := amnt1 radius := amnt2	Notch	[0,1] → [0.0, SF/4] [0.1] → [0.7, 0.9999]
OnePole	STKOnePoleGen	pole := amnt1	HPF LPF	[0,1] → [-0.9999, 0.9999]
TwoPole	STKTwoPoleGen	frequency := amnt1 radius := amnt2	Resonance	[0,1] → [0.0, SF/4] [0.1] → [0.7, 0.9999]
PoleZero				
BiQuad	STKBiQuadGen	frequency := amnt1 radius := amnt2 type := amnt3	Resonance (0) Notch (1)	[0,1] → [0.0, SF/4] [0.1] → [0.7, 0.9999] [0,1] → v > 0.5 ? 1:0;
LPF				
HPF				
BPF				
BRF				
ResonZ				

29. Mir fehlen Analysewerkzeuge zur Bewertung der Qualität der Signalverarbeitung in Neonlicht

1. Ich benötige eine Möglichkeit, den generierten Sound als Wave zu speichern
2. Das sollte mit STK problemlos machbar sein; hierzu existieren entsprechende Klassen (muss ich nachschauen, welche das sind)
3. Cool wäre es, das Ganze als (virtuelles) Speicheroszilloskop zu realisieren, das die Daten direkt aus der Engine empfängt
  1. Zur Visualisierung könnte man beispielsweise openFrameworks (<http://openframeworks.cc/>) verwenden.
  2. Ich habe **oF** auf der virtuellen Maschine installiert, compiliert auch, aber irgendwie habe ich noch Probleme mit dem Projectgenerator :( (siehe auch Punkt 32)
  3. Mal sehen, ist im Moment ein Nebenschauplatz, aber vielleicht wäre das eine Bachelorthesis für einen guten Programmierer ?
4. Dazu müsste ein UGen implementiert werden, der die Audio- und Kontrolldaten (wenn

es geht in Echtzeit) über das Netzwerk verschickt.

5. Ich habe ein Video auf Youtube gesehen, das ein Modul für modulare analoge Synthesizer zeigt, welches genau diese Funktion bereitstellt, finde aber das Video nicht mehr ?
30. Sehr cooler Prototyp eines Synthesizers, der C15 von Nonlinear Labs aus Berlin (ehemaliger Gründer von Native Instruments !, <http://nonlinear-labs.de/>), präsentiert auf der Superbooth 2016, (<https://www.youtube.com/watch?v=konMG2KftVM>)
  1. Skalierbares webbasiertes Interface! Der Synthesizer funktioniert als Access Point.
  2. Synthesizer extrem konfigurierbar
  3. Vollständig digitales Design, Engine entwickelt und getestet in Reaktor
  4. Hardware Interface wird über magnetische Aufsätze konfiguriert
  5. Ribboncontroller, zentraler Controlknopf mit kleinem digitalen Display
  6. Da soll nochmal jemand sagen, digitale Synthesizer klingen nicht!
31. Nur so nebenbei: ich merke gerade, dass ich die Songs von Lunch Money Lewis mag: sehr gut produzierter eingängiger Pop zum Mitsingen, zum Beispiel *Love Me Back* (<https://www.youtube.com/watch?v=UvWOkm8QKBE>)
32. Um *openFrameworks* (oF) auszuprobieren habe ich eine Ubuntu Installation auf dem Laptop unter VirtualBox aufgesetzt
  1. Klappt alles gut, zunächst habe ich Ubuntu 12.XX installiert, das war allerdings eine 32 Bit version
  2. Also nochmal neu und jetzt Ubuntu 14.XX, diesmal in der 64 Bit Fassung ... mit dem Effekt, dass das System sich deutlich langsamer anfühlt ... keine Ahnung woran es liegt, vielleicht fehlt die Unterstützung der Grafikkarte
  3. Egal: openFramework in der 64 Bit Version lässt sich ohne Probleme übersetzen
  4. Das Problem mit dem Projectgenerator (siehe Punkt 29.2) ist auch behoben ... man muss mindestens ein Projekt übersetzen, was dann eine Kommandozeilen-Variante des Projectgenerators erzeugt, diese im Basisverzeichnis von oF installiert und welche dann über die grafische Oberfläche gesteuert wird.
33. Und noch was Schönes aus dem weiten Feld der Compilierung unter Unix (siehe dazu auch Punkt 5, unser Anspruch auf systemübergreifende Nutzung!).
  1. Ich habe *Neonlicht* auf der virtuellen Maschine (Ubuntu) auf dem Laptop installiert
  2. Die Bibliotheken (libasound2-dev, oscpack und STK) compilieren ohne Probleme, bzw. sind entsprechend für Ubuntu vorkonfiguriert. Soweit alles gut.
  3. Dann habe ich über automake + configure die Makefiles von Neonlicht neu erzeugt und

den Übersetzungsvorgang angestossen ... und Bumm, erhalte ich einen Linkerfehler

1. Das System kann die Bibliotheken nicht finden
2. Sie sind aber installiert, die Pfade stimmen auch.
3. Bei kleinen Testprogrammen funktioniert auch alles, der Linker findet die Bibliotheken. Aber warum in den generierten Makefiles nicht ?
4. Die Lösung: unter Ubuntu (vielleicht auch unter anderen Unix/Linux Systemen) müssen die gelinkten Bibliotheken im Übersetzungsbefehl **NACH** den Quelldateien stehen

5. Wie das jetzt wieder hinbekommen? Im **Makefile.am** statt dem Parameter AM\_LDFLAGS, den Parameter LDADD verwenden:

~~1. AM\_LDFLAGS = -lm -lpthread -lstk -lasound -losepack,~~

2. LDADD = -lm -lpthread -lstk -lasound -lospack

6. Damit werden die Bibliotheken hinter die .o Dateien gestellt
1. Jetzt klappt's auch mit dem Ubuntu.

34. Für die einfachere Zuordnung der Belegung des Midi-Controllers sollte ein Teach-In Tool entwickelt werden

1. Etwas Ähnliches gibt es in den meisten Audioanwendungen und sollte deshalb auch für den Midiconnector von Neonlicht verfügbar sein
2. Die entsprechende Konfiguration kann dann als Zuordnungsdatei unter dem Namen des jeweiligen Controllers abgespeichert werden
3. Die Auswahl der Konfigurationsdatei erfolgt dann unixtypisch über einen Kommandozeilenparameter

35. Ok, ich habe es doch gemacht: ich habe **openFrameworks (oF)** auf dem raspi 3 installiert

1. Ich folge dieser einfachen Beschreibung  
<http://openframeworks.cc/setup/raspberrypi/raspberry-pi-getting-started/>
2. Ich bin mir nicht sicher, ob die Compilierung ohne den beschriebenen Speicherkonfigurationsschritt funktioniert. Unter Ubuntu habe ich das natürlich nicht machen müssen ... mal sehen was passiert :)
  1. ok, er compiliert ... und compiliert ... und compiliert ... RASPI. DER RECHNER.
  2. Er nervt mit immer der gleichen Warnung: es scheint da eine unbenutzte Variable **ok** zu geben :)
  3. Jabadabadoo ... er sagt Done! Scheint geklappt zu haben
3. **openFrameworks** hat so ziemlich alle **Multimedialbibliotheken** auf den Raspberry

aufgespielt, die man so braucht. Alleine aus diesem Grunde lohnt es sich schon, das Framework zu benutzen.

4. Ich bin gespannt, ob die Installation meinen Rechner “intakt” gelassen hat, ob die Compilierung und der Start von Neonlight noch klappt ?

1. Ja, läuft alles noch.

36. Ok, **oF** will Jack (<http://jackaudio.org/>) als Soundserver ... muss man starten, klappt auch

1. Allerdings habe ich keinen Input ... also ich habe schlichtweg kein Mikrofon da ;(
2. Audio output funktioniert. Klasse !
3. Und das Framework ist schnell :) Er zeichnet die Wellenformen wie nix. Das klappt mit dem Oszilloskop.
4. Er greift direkt auf den Bildschirmspeicher zu, benötigt also kein X11. Das erzeugt ein bisschen komische Ergebnisse unter X11, heisst aber auch, dass man das abschliessende System ohne rechnerzeitaufwändige grafische Oberfläche nutzen kann.
5. Bei der Wellenformanzeige geht die Rechnerauslastung nicht höher als 10%. Und da ist dann noch die komplette Soundgenerierung mit drin !

37. Ich probiere eine Reihe von **oF** Beispielen durch

1. Die meisten funktionieren tadellos ...
  1. AudioOut, Fonts, GUI, OSC ...
2. 3D Primitive werden angezeigt und drehen sich schön, aber dann schmiert der Rechner ab ??
3. Na gut, man kann nicht alles haben:) Der Raspi hat nun wirklich keine besondere Hardware zur Darstellung von 3D-Objekten ... 2D geht dafür wirklich gut, und das reicht mir im Moment
4. Video zeigt er an, die Interaktion bringt ihn zum Einfrieren
  1. ok, also komplexere Interaktionen mit dem FrameBuffer scheinen ihn aus dem Tritt zu bringen
  2. Immerhin: das System bleibt heile, kein reboot notwendig!

38. Die Leistungsstärke von **oF** ist beeindruckend.

1. Insgesamt könnte es tatsächlich ein Werkzeug für den Unterricht im Bereich interaktive Medienproduktion sein.
  1. Als schrittweise Ergänzung zu Processing
  2. Der zu schreibende Code ist nicht komplizierter und durch den Projektgenerator erhält man zumindestens einen guten Ausgangspunkt für die Eigenentwicklung

3. Es ist portabel, läuft unter Linux, Windows und auch OSX und sogar iOS, damit können dann auch die Apple Jünger was anfangen.
2. Man müsste das Ganze mal auf unseren Monstermaschinen im Unixlabor installieren ...
  1. Dann wäre Linux möglicherweise auch wieder für die Medienleute ein interessanter Spielplatz
39. Ich habe ein Addon in *oF* installiert: *ofxBox2D* (<https://github.com/vanderlin/ofxBox2d>), ein openFrameworks Wrapper um die bekannte 2D Physik- und Kollisionsengine
  1. Die "Installation" eines Addons reduziert sich auf das Kopieren in ein anzulegendes Unterverzeichnis im Verzeichnis *addons* von oF.
  2. Danach dann das übliche *make* und schon kann man die Bibliothek verwenden
  3. Es sind einige schöne Beispiele mit Partikelsystemen dabei.
  4. Leider kann ich keine Screenshots zeigen ... die Zeichnung erfolgt ja direkt in den Framebuffer, deshalb kann *scrot* keine Daten auslesen :)
40. Wieviele und welche Unit-Generatoren müssen in Neonlicht zur Verfügung stehen ? Das System sollte leistungsstark genug sein, um *alle* aktuellen Syntheseansätze nutzen zu können, zumindestens experimentell. Das heisst wiederum, dass man sich an dem Stand der Entwicklung der aktuellen System orientieren muss.
  1. Ein Referenzsystem stellt dabei Pure Data dar (<https://puredata.info/>).
  2. Eine Liste der Objekte in Pure Data findet man z.B. in den Floss Manuals (<http://en.flossmanuals.net/pure-data/>)
  3. Sie ist ziemlich lang. Ich werde eine Auswahl treffen, und diese für Neonlicht umsetzen.
41. Um die Schnittstelle der komplexeren Soundunits besser kapseln zu können, muss es möglich sein Eingabe/Ausgabe Ports festzulegen zu können
  1. Am einfachsten geht das wahrscheinlich über einen *NumberGen*, also einen Generator, der einfach einen Zahlenwert repräsentiert
  2. Dieser kann dann extern gesetzt und gelesen werden
  3. In der Soundunit selbst wird dann im ersten Schritt der Wert ausgelesen.
42. Um die Unit Generatoren und die Soundunit später besser/einfacher kombinieren zu können, sollte eine Konvention, ein Mechanismus zur Beschreibung der Parameter entwickelt werden
  1. In etwa so ähnlich, wie die Anfrage an Objekte in Python, bei der man einen deskriptiven Text für die jeweilige Funktion, den jeweiligen Parameter erhält
  2. Das verhindert zwar keine Fehlbenutzung, macht aber die richtige Benutzung deutlich leichter

3. Anhand dieses Textes kann dann beispielsweise die grafische Schnittstelle besser/leichter an eine neue Soundunit angepasst werden
43. Für die Koppelung von grafischer Schnittstelle und Neonlicht würde ich gerne die übliche **Model/View/Controller** Architektur ([https://de.wikipedia.org/wiki/Model\\_View\\_Controller](https://de.wikipedia.org/wiki/Model_View_Controller)) umsetzen
  1. Allerdings distribuiert über drei Komponenten, die über **Osc** miteinander kommunizieren und eben nicht als drei Komponenten in einer Anwendung
  2. Damit wird es möglich Steuerungskomponenten auf jeder beliebigen Hardware laufen zu lassen
    1. z.B. als Client auf einem Handy ... *“Ich bin der Musikant mit einem Handy in der Hand”*
44. Der erste Schritt in Richtung bessere Analyse der Synthese ist getan
  1. Ich habe einen einfachen **WaveOutGen** implementiert, der den generierten Stream abgreifen kann und in eine Wavedatei speichern kann.
  2. Damit wird es jetzt möglich, die Ausgabe von Neonlicht in **Audacity** (<http://www.audacityteam.org/>) abzuspielen und natürlich auch
    1. die Wellenform genauer zu betrachten und
    2. über das Spektrum die (statische) Verteilung der Obertöne zu betrachten
45. Der nächste Schritt wäre jetzt das Versenden der Audiodaten über eine UDP Verbindung zu einer Oszilloskop Anwendung, die das Signal live zeigt
46. Habe eine Reihe von Glue Unit Generatoren implementiert
  1. **AddTwoGen** → berechnet den Mittelwert der Eingabeports In1 und In2
  2. **MultiplyTwoGen** → berechnet das Produkt der Eingabeports In1 und In2
  3. **MultiplyGen** → berechnet das Produkt aus Amnt1 und In1
  4. **NumberGen** → definiert einen konstanten Wert, gesetzt in Port Amnt1
  5. **PhasorGen** → Rechnet den SawGen in eine Rampe von 0 nach 1 um
47. Ok, Schrittchen für Schrittchen geht es voran: 18 Unit Generatoren sind implementiert, fehlen noch mindestens 32 weitere :)
  1. Ganz wesentlich natürlich noch der mehrschrittige Hüllkurvengenerator
    1. In der Standardform als ADSR (<https://de.wikipedia.org/wiki/ADSR>)
    2. In der allgemeinen Form als eine beliebige Abfolge von linearen Einzelschritten, bestehend aus Zeit/Wert Paaren
  2. Die **LFOs (Low Frequency Oscillator)** als Sonderform der Oszillatoren, sind prinzipiell

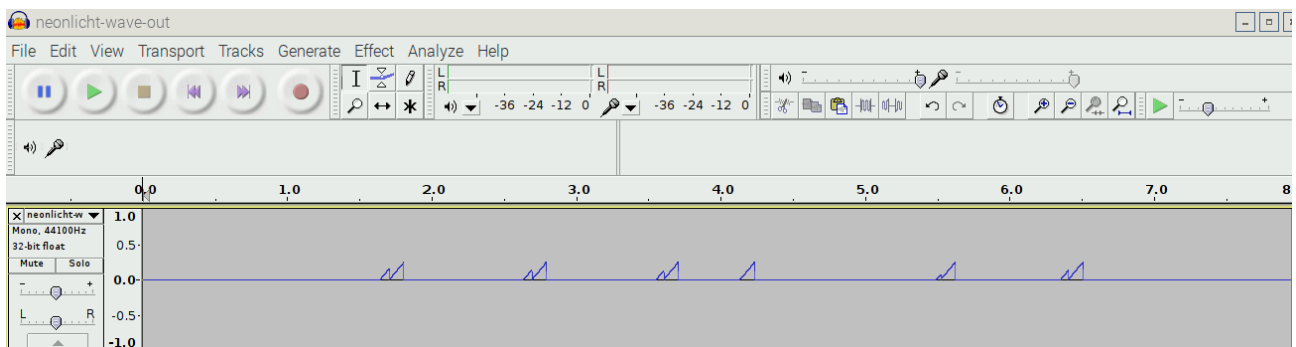


einfach umzusetzen, allerdings:

1. Wenn der Oszillator sehr langsam schwingt, sollten mehr Zwischenwerte berechnet werden, da es sonst zu hörbaren Sprüngen kommt
3. Ein **Sample/Hold Generator** (<https://de.wikipedia.org/wiki/Sample-and-Hold-Schaltung>), weil man damit in Kombination mit dem Rauschgenerator so schöne glucksende Töne erzeugen kann
4. Und natürlich, auch sehr wichtig, ein **Wavetable Generator**
  1. Einerseits, um die Variationsbreite der Grundschrwingungen noch zu erweitern
  2. Andererseits sind die Wavetables natürlich die Grundlage für die nicht-lineare Wavetablesynthese ([https://en.wikipedia.org/wiki/Wavetable\\_synthesis](https://en.wikipedia.org/wiki/Wavetable_synthesis)), und die erzeugt wirklich sehr interessante Grundklänge, vor allen Dingen für Pads (hier sind “Klangteppiche” gemeint, keine Apple-Produkte).
    1. Man höre sich den PPG 2.3 von Wolfgang Palm an (<https://www.youtube.com/watch?v=izVH1CENIDs>)
48. Ich habe ein sehr schönes GUI Addon für **oF** gefunden, damit lassen sich, glaube ich, genau die Interfaces für die aktuellen Synthesizer sehr schön umsetzen
  1. Es heisst **ofxUI** (<https://github.com/rezaali/ofxUI>).
  2. Mehr Information und ein Video finden sich unter (<http://www.creativeapplications.net/openframeworks/ofxui-new-gui-addon-for-your-openframeworks-projects-openframeworks/>)
  3. Allerdings scheint es **deprecated** zu sein ... und wird seit 2012 nicht mehr unterstützt. Schade. Die Demos sind sehr gut, compilieren und laufen auch noch.
  4. Hmm, ich glaube ich probier's trotzdem mal aus ... auch wenn es letztlich keine Zukunft hat.
49. Der erste Schritt zum allgemeinen Hüllkurvengenerator ist getan.
  1. Mit **EGOneStepGen** steht eine lineare Interpolation zur Verfügung, aus der die mehrschrittigen Hüllkurven aufgebaut werden können
  2. Der Generator ist für die interne Verwendung in anderen UGens konzipiert, stellt also im Gegensatz zu den meisten der implementierten UGens nur eine rudimentäre Schnittstelle über die **control** Methode zur Verfügung
    1. Konkret lässt sich hier nur mit der Nachricht **“trigger”** die Interpolation auslösen
  3. Über die Methoden **setDuration**(float seconds); **setStartLevel**(float level); und **setEndLevel**(float level); lassen sich die drei Parameter des UGen direkt setzen.
  4. Mit der Methode **bool finished()** lässt sich überprüfen, ob die Interpolation

abgeschlossen ist.

5. Aus der linearen Interpolation lassen sich natürlich relativ einfach nichtlineare Kurvenverläufe berechnen, sollten diese zu einem späteren Zeitpunkt mal benötigt werden
  1. Hier käme dann beispielsweise wieder der Wavetable-Ansatz zum Tragen
6. Getestet habe ich den Hüllkurventeilschritt mit dem gestern implementierten **WaveOutGen**. Damit kann man dann schön sehen, wie die lineare Interpolation berechnet wird und wie das System auf schnelle Triggerfolgen reagiert (nämlich in der aktuellen Implementierung mit einem Retrigger)



50. Ich habe eine Reihe von Büchern aus Bibliothek zum Thema “Electronic Music” ausgeliehen, unter anderem **“The Oxford Handbook of Computer Music”** von Roger T. Dean  
(<http://www.oxfordhandbooks.com/view/10.1093/oxfordhb/9780199792030.001.0001/oxfordhb-9780199792030>).

1. In sein Aufsatz **“A Historical View of Computer Music Technology”** beschreibt Douglas Keislar die fortschreitende Abstraktion, welche durch die Entwicklung von Musikinstrumenten und Notationssystemen geleistet wird. In diesem Zusammenhang bezeichnet er Synthesizer als **Meta-Instrumente**, was ich als sehr schlüssig empfinde.
  1. Ausgangspunkt der Abstraktion ist die Idee, dass die menschliche Stimme die ursprüngliche Basis aller Musikinstrumente darstellt, und durch die schrittweise Entwicklung der Musik eine zunehmende Entkoppelung (und damit Befreiung und Verbreiterung der Handlungsmöglichkeiten) bewirkt wird.
  2. Folgt man diese Argumentation ist Neonlicht also ein **Meta-Meta-Instrument!**

51. Ich habe den Ugen **GatedConstantGen** implementiert.

1. Diese liefert in Abhängigkeit von **amnt2** (gate) entweder
  1. **0**, falls gate < 1 ist
  2. **amnt1**, falls gate >= 1 ist;

2. Der UGen wird unter anderem im ADSR benötigt.
  1. Hier setzt er die **Sustain-Phase** um, die solange auf einem konstanten Wert bleibt, wie die Taste auf dem Midi Controller gedrückt ist
    1. Was wiederum mit dem **Gate**-Wert im System repräsentiert wird.
    2. Bei einem monophonen Synthesizer sollte also der Gate-Wert als global zugreifbarer Wert umgesetzt werden, womit dann alle UGens den Zustand auslesen können

52. Um Gate und Trigger Events ordnungsgemäß generieren zu können benötige ich einen **MidiInGen**.

1. Die Hauptaufgabe dieses UGen ist Zwischenspeicherung des Midizustandes, konkret das Speichern des aktuellen **NoteOn/NoteOff** Event Paares.
2. Wenn eine Midi **NoteOn** Event eintrudelt, dann muss der UGen den **Notenwert** und die **Anschlagsstärke** (Velocity) speichern
  1. Im Anschluss muss ein **NoteOff** Event für den vorherigen Notenwert erzeugt werden, damit die Wiedergabe dieses Tons beendet wird
  2. Im nächsten Schritt wird dann der **NoteOn** Event für den aktuellen Notenwert erzeugt und die Wiedergabe der aktuellen Note beginnt
  3. Das Erzeugen eines **NoteOff** Event bedeutet, dass der **Gatewert** auf 0 gesetzt wird. Damit stoppt die Wiedergabe im gesamten System (wenn die Soundunit korrekt implementiert ist!!!)
  4. Das Erzeugen eines NoteOn Event bedeutet, dass der **Gatewert** auf 1 gesetzt wird und gleichzeitig ein **Trigger** gesetzt wird. Der Gatewert bleibt solange auf 1, bis ein **NoteOff** Event ihn zurücksetzt, oder ein weiterer **NoteOn** Event die Wiedergabe eines (möglicherweise anderen) Notenwertes anfordert
3. Wenn ein Midi **NoteOff** Event eintrudelt, dann sind prinzipiell zwei Möglichkeiten denkbar
  1. Der Gatewert wird **unabhängig** vom Notenwert (also von der gedrückten Taste) zurückgesetzt und die Wiedergabe stoppt.
    1. Das hätte den Effekt, dass wenn 2 Tasten “gleichzeitig” gedrückt werden und ein davon losgelassen wird, dass die Wiedergabe stoppt, obwohl möglicherweise die andere Taste noch weiterklingen soll
    2. Gleichzeitig stellt der Ansatz sicher, dass es eigentlch zu keinen “Hängern” kommen sollte, es sei denn die NoteOff Events gehen verloren. Dann kann man mit dem Drücken/Loslassen einer beliebigen Tasten den “Hänger” stoppen.
  2. Der Gatewert wird **abhängig** vom Notenwert (also von der gedrückten Taste)

zurückgesetzt und die Wiedergabe stoppt.

1. Damit wäre eine polyphone Spielweise des System zumindestens vorbereitet. Tastenanschläge werden als Paare verstanden und könnten so prinzipiell jeweils auf unterschiedliche Instanzen der Soundunit geleitet werden, welche parallel arbeiten
  2. Allerdings steigt die Wahrscheinlichkeit von Hängern, da ein **“verlorener”** NoteOff Event nicht durch einen **“notenfremden”** NoteOff Event kompensiert werden kann.
4. Damit das ganze reibungslos und verlustfrei funktioniert, benötigt der UGen eine Ausgangsqueue
1. Die Events müssen nacheinander in mehreren aufeinanderfolgenden Ticks verarbeitet werden.
  2. Entsprechend muss der UGen in der Lage sein seinen Status über einen begrenzten Zeitraum zwischenzuspeichern
  3. Man muss ausprobieren, welche Größe die Queue haben muss. Im Moment denke ich, dass eine Länge von 32 reichen müsste
1.  $32 = 8 \text{ Tasten/Stimmen gleichzeitig mit jeweils } 2 \text{ NoteOn/NoteOff Paaren}$