AUDIO-PROJEKT WS 17/18:

Teilnehmer: Arpad Krause

Einführung:

Mein Projekt war die Programmierung eines Synthesizers, der auch komplexere Töne abspielen kann. Komplexere Töne meint hier, dass nicht nur Sinuskurven verwendet werden können. Um diesen Synthesizer programmieren zu können, war es unerlässlich, sich mit dem Thema der Soundtechnik auseinanderzusetzen. Dieses Wissen war nötig, um zum Beispiel den Kompressor zu bauen. Andere wichtige Themen, um den Synthesizer bauen und visualisieren zu können, waren Pygame, Tkinter und Pyaudio. Der klare Fokus des Projekts lag auf dem Synthesizer.

Ich möchte noch erwähnen, dass ich kein Interesse daran hatte, ein schönes oder praktisches interface aufzubauen; dies ist auch in der jetzigen Version nicht vorhanden.

Gliederung:

1. Arbeitsverlauf:

Zum Start der Projektarbeitsphase gab mir Stefan ein Programm, welches mit Hilfe von Pyaudio eine simple Sinuskurve abspielen kann. Die Struktur, mit der dieses Programm den Ton herstellt, verwende ich immer noch. Meine ersten Schritte bestanden also darin zu verstehen, wie dieses Programm funktioniert und wie ich dafür sorgen kann, dass man den abgespielten Ton verändern konnte. Hierbei ging es darum, eine Möglichkeit zu finden, die Variablen zu verändern, die einen Ton charakterisieren. Hier gab Stefan mir den Tipp, Tkinter zu verwenden, um eine Eingabemöglichkeit zu haben. Tkinter ist ein Modul, welches unter anderem in Python verwendet werden kann, um ein Interface bzw. eine GUI zu bauen.

Als nächstes versuchte ich, dafür zu sorgen, dass mehrere Töne gleichzeitig abgespielt werden konnten. Ein wichtiger Schritt dahingehend war es, die Eingabe über die GUI abzuschaffen. Ich nutzte dann ein MIDI-Keyboard, welches dafür sorgte, dass die Eingabe erleichtert wurde und mehrere Töne gleichzeitig gespielt werden konnten. So eröffneten sich mehr Möglichkeiten, Töne komplexer zu machen, z.B. die Geschwindigkeit vom Tastendruck mit einzuberechnen, was ich jedoch bis heute nicht nutzte.

Nun gab es natürlich immer noch große Probleme, beispielsweise hört man ein Knacken, wenn man einen Ton beendete. Des Weiteren gab es keinen Lautstärkeverlauf, wird zum Beispiel ein Ton bei einer Gitarre angeschlagen und klingt aus, dann kann man diesen Ausklang beeinflussen und den Anschlag beeinflussen. Das war bis dahin bei dem Synthesizer nicht so. Ein Ton war direkt so laut wie es ging und wurde auch nicht leiser, also fehlte die Dynamik. Hier kommt die Hull-Kurve ins Spiel. Diese sorgt dafür, dass das Knacken verschwand und eine Dynamik existierte.

Um den Pegel, also die Lautstärke des Outputs, zu kontrollieren, was für Pyaudio wichtig war, verwendete ich eine einem Mischer ähnliche Struktur, die dafür sorgen sollte, dass der Pegel nicht übersteuerte. Diese Mischer-ähnliche Struktur behinderte allerdings wiederum die Dynamik, was dazu führt, dass Komplexität von Tönen für den Hörer verloren geht. Um diese Problem zu beheben, programmierte ich einen Kompressor.

Nun waren alle grundlegenden Strukturen vorhanden, jetzt konnte man noch dafür sorgen, dass es eine Soundsynthese mit mehr Möglichkeiten gibt. Das, was ich darauf einbaute, war: ein Generator, der mit der FM-Synthese arbeitete, ein Resonanzgenerator und ein Delay, welches der bisher einzige Soundeffekt ist.

2.grundlegende Strukturen:

Ich werde in diesem Teil darauf eingehen, wie einzelne Teile aus dem Kompressor funktionieren.

Pyaudio:

Pyaudio ist ein Modul von Python, welches Informationen an die Soundkarte weitergeben kann, die dann abgespielt werden. Dies funktioniert über ein callback.

```
p = pyaudio.PyAudio()

def play(gen):
        '''Startet den Tongenerator und die Audioausgabe'''

        def callback(in_data, frame_count, time_info, status):

            data = (32767*gen.generate(frame_count/24000)).astype(np.int16)
            return (data.copy(), pyaudio.paContinue)


        global outStream

        outStream = p.open(format=pyaudio.paInt16,channels=1,\
                rate=24000,output=True,frames_per_buffer=2**6,stream_callback=callback)
```

Mit Hilfe einer callback-Funktion wird dann das callback aufrecht gehalten.

Im pyaudio.Pyaudio() objekt werden bestimmte Variablen eingespeichert: *rate, frames_per_buffer.*

Rate: Im Code wird man diese Variable öfter vorfinden; allerdings unter dem Namen RATE. Diese Variable bestimmt die Samplingrate, mit der pyaudio arbeitet. Diese bestimmt, wie detailliert das Tonsignal ist. Die Samplingrate einer wave-Datei beträgt 44100 Hz. Wie man sehen kann, beträgt die Samplingrate hier 24000.

Frames_per_buffer: Im Code ist diese Variable unter dem Namen CHUNK eingespeichert. Diese Variable gibt an, wie groß der Teil eines sample packets ist, welches durch das callback angefordert wird.

Das Programm muss also Informationen an die callback-Funktion von pyaudio weiterleiten, die von pyaudio lesbar sind. Pyaudio benötigt also eine bestimmte Datenstruktur, diese wird durch *format* bestimmt. In diesem Programm sind es arrays mit der Größe CHUNK(*frames_per_buffer*), die floats enhalten. Die in den arrays enthaltenen floats müssen in einem bestimmten Intervall liegen [-1,1]. Wenn diese Bedingung nicht zutrifft, wird man ein „Knacken“ hören.

Das callback fragt nach einer Funktion mit dem Namen: generate, und gibt die Variable (frame_count/24000) an diese weiter.

```
if __name__ == "__main__":
    play(ver)
```

In dieser Struktur wird dann das callback abgefragt, play ist dabei die callback funktion, „ver“ ist eine class, die das callback erstellt, „ver“ muss eine Funktion mit dem Namen: generate enthalten, die generate-funktion in „ver“ muss die Bedingungen einhalten, die Pyaudio stellt.

Pygame-midi-eingabe:

In Pygame gibt es ein Modul „pygame-midi“, welches Daten von einem Midi keyboard abfangen kann. Hier sieht man ein Beispiel von den Daten, die man abfängt.

```
C:\WINDOWS\system32\cmd.exe - python  -i whateverpygamedoes.py
(C:\Users\Arpad\Anaconda2) C:\Users\Arpad\Desktop\Mit Keyboard>python -i whateverpygamedoes.py
 0: interface :MMSystem:, name :Microsoft MIDI Mapper:, opened :0:  (output)
 1: interface :MMSystem:, name :LPK25:, opened :0:  (input)
 2: interface :MMSystem:, name :Microsoft GS Wavetable Synth:, opened :0:  (output)
 3: interface :MMSystem:, name :LPK25:, opened :0:  (output)
using input_id :1:
<Event(34-Unknown {'status': 144, 'vice_id': 1, 'timestamp': 4785, 'data1': 55, 'data3': 0, 'data2': 1})>
<Event(34-Unknown {'status': 128, 'vice_id': 1, 'timestamp': 5230, 'data1': 55, 'data3': 0, 'data2': 127})>
<Event(34-Unknown {'status': 144, 'vice_id': 1, 'timestamp': 5919, 'data1': 52, 'data3': 0, 'data2': 77})>
<Event(34-Unknown {'status': 144, 'vice_id': 1, 'timestamp': 6061, 'data1': 64, 'data3': 0, 'data2': 91})>
<Event(34-Unknown {'status': 144, 'vice_id': 1, 'timestamp': 6184, 'data1': 59, 'data3': 0, 'data2': 115})>
<Event(34-Unknown {'status': 128, 'vice_id': 1, 'timestamp': 6230, 'data1': 64, 'data3': 0, 'data2': 127})>
<Event(34-Unknown {'status': 128, 'vice_id': 1, 'timestamp': 6268, 'data1': 52, 'data3': 0, 'data2': 127})>
<Event(34-Unknown {'status': 144, 'vice_id': 1, 'timestamp': 6363, 'data1': 65, 'data3': 0, 'data2': 114})>
<Event(34-Unknown {'status': 128, 'vice_id': 1, 'timestamp': 6481, 'data1': 59, 'data3': 0, 'data2': 127})>
```

Die Informationen, die hier wichtig sind, beginnen ab der 5. Zeile.

Man bekommt für jedes events auf dem midi-keyboard ein dictionary mit den Werten *status, timestamp, vice_id, data1, data3* und *data2* zurückgegeben.

status gibt an, ob die Taste gedrückt oder losgelassen wird, wobei 144 = drücken und 128 = loslassen ist.

vice_id bestimmt, von welchen Gerät diese Eingabe kommt.

timestamp, bedarf keiner nähren Erläuterung.

data1 gibt an, welche Taste gedrückt wurde, über einen int mit einem Index, wobei data1 >= 0 ist.

data3 ist eine Variable, die ich weder nutze noch weiß ich, welche Bedeutung sie hat,

data2 gibt an, mit welcher Geschwindigkeit man die Taste gedrückt hat.

Das Abfangen der Daten läuft über diese Funktion ab:

```
    def generate_tasten(self):
        """liefert midi input"""

        events = self.event_get()

        for e in events:
            if e.type in [pygame.midi.MIDIIN]:
                ton=e.data1
                if e.status==144:#ton objekt init
                    ver.generiere_spur_key(ton)
                elif e.status==128:#delete-funktion
                    try:
                        ver.generiere_zombie_spur(ton)
                        ver.DEL_spur(ton)
                    except KeyError:
                        pass
```

Wie man sehen kann, benutze ich von all diesen Variablen zwei für den Synthesizer, jedoch könnte man für *data2* noch eine sinnvolle Anwendung finden.

Durch die Funktion, die oben abgebildet ist, werden alle Events abgefangen und ausgewertet. Wobei *data1* den Ton definiert und auch einen Index für die daraus neu generierte oder gelöschte Spur, der Begriff Spur benötigt einer Erklärung, und *status,* ob der Ton entsteht oder gelöscht wird, eine Zombiespur benötigt auch eine Erklärung.

<u>Ton-Generation:</u>

Ich zeige hier nur das simpelste Beispiel, da sich die anderen aus diesem ableiten.

Für die Generation eines Tones benötigt man die Frequenz, das ist die Variable, die einen Ton von anderen unterscheidet, wie vorher in Pygame-midi dargestellt, wird jeder Ton durch einen integer definiert.

Man weiß, dass ton(n) und ton(n+1) Frequenzen in dieser Beziehung stehen:
Frequenz(ton(n+1))=Frequenz(ton(n))* 2**(1/12)

Jetzt benötigt man nur noch einen bestimmten Ton(integer) mit einer bestimmten Frequenz, hier bietet sich der Kammerton: A an, A hat eine Frequenz von 440 Hz, wobei man hier aber für Diskussionen offen sein kann.

Man kann den Ton also bestimmen, wenn man diese beiden Dinge weiß, daraus leitet sich auch ab dass A1, also A um eine Oktave höher, die doppelte Frequenz von A haben muss, oder A die halbe Frequenz von A1.

Also berechne ich die Frequenz für jeden Ton mit dieser Funktion:

```
def frequenz(n):
    '''berechnet die frequenz'''
    if n==0:
        return (27.5)*2**(1/6)
        #der faktor 2**(1/6) ist leider keyboard spetzifisch
        #die start frequen von 27.5 Hz ist nicht nötig,
        #man kann auch bei einem höheren ton anfangen
    else:
        return 2**(1/12)*frequenz(n-1)
```

Bei der Berechnung der Frequenzen von diesem Synthesizer hat der niedrigste Ton eine Frequenz von 27.5 * 2**(1/6) Hz. Der Mensch kann natürlich nur in einem bestimmten Frequenzspektrum hören, deshalb lohnt es sich auch nicht sonderlich, tiefere Töne als 20 Hz zu erzeugen, wobei auch diese schwer oder gar nicht zu hören sind. 27.5 Hz ist ein 16tel von 440 Hz (Kammerton).

Jetzt berechnet man eine Sinuskurve wie folgt:

```
def generiere_sinuskurve(freq):
    """eine beispiel"""
    yy=np.arange(int(T*RATE))*(2*np.pi*freq)/vars.RATE\
        +vars.phaseshift*np.ones(int(CHUNK))
    vars.phaseshift+=T*2*np.pi*self.freq

    yy=(np.sin(yy))
    return yy
```

Diese Funktion kommt nicht im Code vor es ist nur ein Bespiel.

Hier werden auch andere Variablen als nur die Frequenz genutzt*: T, RATE* und *CHUNK* sind schon erklärt, *T* ist die Variable, die vom callback weitergegeben wird, *phaseshift* und *freq* sind es noch nicht.

freq ist die Frequenz.

phaseshift ist eine Variable, die einspeichert, in welcher Phase sich die Sinuskurve befindet. Sie sorgt dafür, dass man alle Teile der Sinuskurve hintereinander sampeln kann, wenn man diese linear verändert vergrößert proportional zur der Frequenz.

Jede Spur braucht eine dieser Generatorfunktionen, um ein Signal zu erzeugen. Eine Spur ist ein Ton, der abgespielt wird, also entsteht für jeden Tastendruck eine Spur, die konstant den gleichen Ton abspielt.

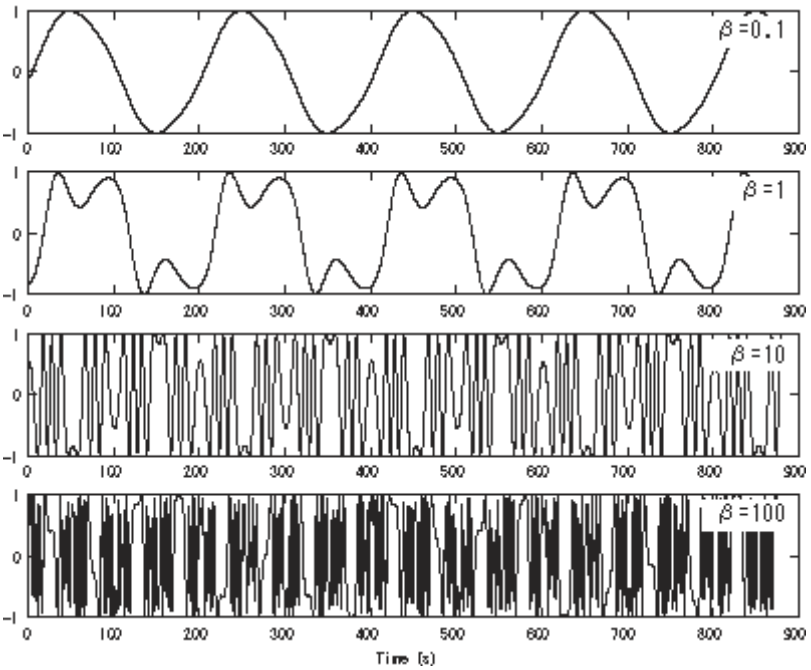
FM-Synthese:

Die FM-Synthese eröffnet die Möglichkeit, viele unterschiedliche Töne mit nur zwei Sinuskurven erzeugen zu können, anstatt eine simple Sinuskurven zu verwenden und diese zu addieren, wird bei der FM-Synthese die Ton-Kurve moduliert mit einer anderen Ton-Kurve. Die Mathematik dahinter sieht folgend aus:

$$s_{fm}(t) = A \cdot \cos(\omega_c t + \beta \cdot \cos(\omega_m t))$$

Durch die FM-Synthese werden 2 neue Variablen in die Tongeneration eingebaut (zuvor war es nur eine pro Ton), *BETA* und *fm-frequenz. Beta* gibt an, wie stark das Verhältnis zwischen den beiden Sinuskurven ist, durch die FM-Synthese sind beide Sinuskurven voneinander abhängig geworden. *fm-frequenz* gibt die Frequenz der Modulationsschwingung an. Durch globale Variablen wird das in meinem Programm von dem Interface übernommen. Durch dieses Verfahren entstehen viele Nebenschwingungen, die den Ton komplizierter machen, aber jedem Ton einen Charakter geben kann. Bspw. kann man durch eine niedrige *fm-frequenz* dafür sorgen, dass es immer sehr tiefe Nebenschwingungen gibt. Mit dem Beta-faktor kann man diese Charakteristik dann verstärken, es ergeben sich sehr interessante Töne.

Allein bei Änderung eines Faktors, hier Beta, können sehr unterschiedlich Klänge entstehen.



https://upload.wikimedia.org/wikipedia/commons/f/fd/Frequencymodulationdemo-td.png

Diese Töne könnten natürlich auch additiv erzeugt werden, da man alle sich periodisch wiederholenden Graphen durch addierte Sinuskurven darstellen kann, jedoch wäre die benötigte Rechenleistung dafür enorm, deshalb ist die FM-Synthese so interessant gewesen zu Zeiten, wo Computer nicht so schnell waren wie heute.

Resonanz:

In Klavieren schwingen, beim Anschlag eines Tones, andere Saiten immer mit. Es schwingen alle Oktaven mit. Klaviere haben aufgrund der Resonanz sehr komplexe Töne, die sich sehr natürlich anhören, auch für Synthesizer wurde das Prinzip der Resonanz aufgegriffen. Bei Synthesizern ist das Prinzip natürlich sehr viel „unorganischer“. Man berechnet dann die Nebenfrequenz und deren Lautstärke. Die Anzahl und die Lautstärke der Töne sind abhängig vom Resonanzfaktor, letztendlich entstand folgende Funktion, um die Nebenfrequenzen und deren Lautstärke zu berechnen.

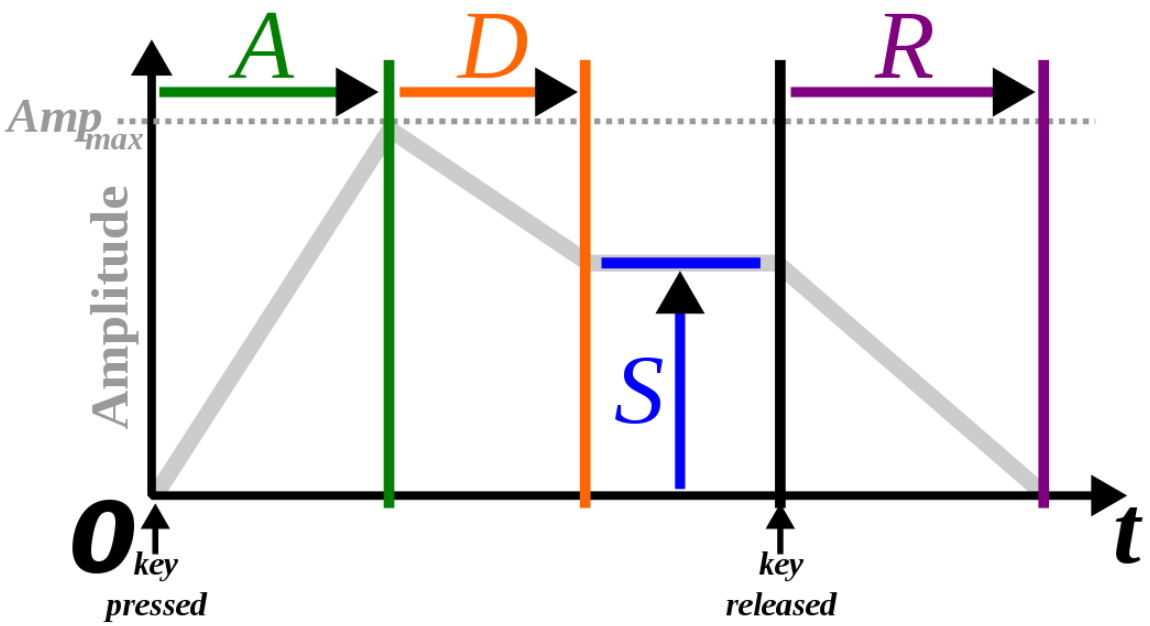
```
def bestimmung_der_neben_frequenzen(self):
    """eigentliches herz der resonanz, ist aber nicht sehr akurat und er skizziert """
    self.array.update({self.freq:[1,0,True]})
    for i in range(int(np.log(self.grenzwert)/np.log(self.resonanzfaktor))):
        copy=self.array.copy()

        for i in copy.copy():
            if self.array[i][0]*self.resonanzfaktor<=self.grenzwert:
                self.array[i][2]=False
                pass
            else:
                if self.array[i][2]:
                    x=i*2
                    if x in self.array:
                        self.array[x][0]+=self.array[i][0]*self.resonanzfaktor
                        self.array[x][2]=True
                    else:
                        self.array.update({x:[self.array[i][0]*self.resonanzfaktor,0,True]})
                    y=i/2
                    if y in self.array:
                        self.array[y][0]+=self.array[i][0]*self.resonanzfaktor
                        self.array[y][2]=True
                    else:
                        self.array.update({y:[self.array[i][0]*self.resonanzfaktor,0,True]})
                    self.array[i][2]=False
                else:
                    pass
```

Das ist eine skizzierte Version der Resonanz.

Hullkurve:

Die Hullkurve ist eine Art, den Tonverlauf, Lautstärkeverlauf eines Tones, zu gestalten. Dabei teilt man den Tonverlauf in vier Phasen ein; *Attack, Decay, Sustain* und *Release*, deswegen wird die Kurve auch ADSR-Kurve genannt. Durch die Hullkurve entsteht eine gewisse Dynamic, die für interessantere Töne sorgen kann.

Die ersten beiden Phasen, Attack und Decay, sorgen dafür, dass man so etwas wie einen Anschlag vernimmt, sobald der Attack vorbei, also die maximale Lautstärke des Anschlages erreicht wurde, startet der *Decay*. Durch den *Decay* wird die Lautstärke auf das Level des *Sustians* gesenkt, sobald das *Sustian* erreicht wurde, bleibt der Level konstant. Der *Release* wird aktiviert, sobald die Taste, die den Ton aktivierte, losgelassen wird. Der *Release* lässt das Level auf 0 sinken. Im Synthesizer ist der *Release* von den anderen Phasen getrennt. Der Release wird in einer anderen Spur durchgeführt , der „Zombiespur“.



https://upload.wikimedia.org/wikipedia/commons/e/ea/ADSR_parameter.svg

Damit die Hullkurve einen Einfluss auf das Signal hat, legt der Generator ein timer-array an, welches dann mit dem signal-array multipliziert wird, wodurch Lautstärke und Signal unabhängig voneinander bearbeitet werden bis zur Weitergabe. Die Variablen *Decay*, *Attack* und *Release* bestimmen die Dauer von jeder Phase bis auf den *Sustain*, der *Sustain* gibt seine eigene Lautstärke an, also hat man jeweils einen Punkt, zu dem der Graph linear steigen muss, wenn man sich die Hullkurve in einen zweidimensionalen Koordinatensystem vorstellt, wobei die x-Achse die Zeit in Sekunden darstellt und die y-Achse die Lautstärke in [0,1], die Lautstärke hat einen float Wert in diesem Intervall. Also kann man, wenn man einen Punkt gegeben hat, den man erreichen möchte, die Steigung errechnen. Wenn man dieses hat, kann man alle Punkte zwischen Anfang einer Phase und Ende einer Phase berechnen, dies führte zu diesem Code:

```
'''timer für ton Hull kurve'''
if self.status==1 or self.status==0:
    self.timer=np.arange(int(T*vars.RATE*(self.counter-1)),int(T*vars.RATE*self.counter))
"""form der ton Hull kurve"""
if self.status==0:#ATTACK
    if self.counter==1:
        self.steigungswinkel=(self.max_Amplitude/vars.ATTACK)
    if self.counter<=(int(vars.ATTACK*int(vars.RATE/vars.CHUNK))):
        self.timer=((self.timer/vars.RATE)*self.steigungswinkel)
        if self.counter==(int(vars.ATTACK*int(vars.RATE/vars.CHUNK))):
            self.DECLINE=vars.ATTACK*(self.max_Amplitude-(self.max_Amplitude*0.7))

            self.counter=1
            self.status=1

        else:
            self.counter+=1

elif self.status==1:#DECLINE
    if self.counter==1 :
        self.steigungswinkel=(((self.max_Amplitude*0.7)-self.max_Amplitude)/self.DECLINE)
    if self.counter<=(int(self.DECLINE*int(vars.RATE/vars.CHUNK))):
        self.timer=self.max_Amplitude+((self.timer/vars.RATE)*self.steigungswinkel)
        if self.counter==(int(self.DECLINE*int(vars.RATE/vars.CHUNK))):
            self.counter=1
            self.status=2
        else:
            self.counter+=1
```

self.status ist eine Variable, die angibt, in welcher Phase man sich befindet, 1 steht für *Attack*, 2 für *Decay* und 3 für *Sustain*.

Kommentar:

Der Grund, weshalb ich den *Release* von den anderen Phasen trennte, ist mir im Nachhinein schleierhaft.

Kompressor:

Um den Output zu kontrollieren und für Pyaudio lesbar zu machen, musste ich eine Struktur einbauen, die dies ermöglichte. Ich entschied mich dann für einen dynamic range Kompressor. Diese versuchen, obwohl es eine Kompression gibt, noch dafür zu sorgen, dass ein wenig Dynamik vorhanden ist, ohne dieses Konzept würden beim Drücken von mehreren Tasten langsam alle Töne, egal wie laut sie eigentlich sind, gleichmäßig leiser werden, wodurch kein Anschlag mehr wahrzunehmen wäre. Mein Kompressor braucht die Variablen*: Threshold, Ratio, Kneewidth, Attacktimer* und *Make up gain*.

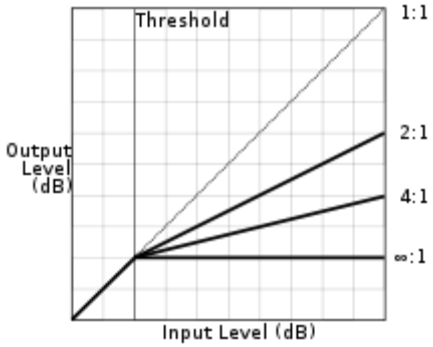
Der *Threshold* bestimmt die Lautstärke, bei der der Kompressor anfangen soll, die Lautstärke zu senken.

Die *Ratio* ist der Faktor, der bestimmt, wie stark die Kompression ist, wobei 1 keiner Kompression und 60 nahezu limitting entspricht.

Die *Kneewidth* bestimmt die Größe des softknees, welches dafür sorgt, dass der Unterschied zwischen Kompression und Nicht-Kompression geschmeidiger verläuft. Die Kompression wird dann auch als das *Hardknee* bezeichnet.

Der *Attacktimer* ist ein Faktor, der bestimmt, wie schnell der Kompressor angreift.

Mit dem *Make up Gain* kann man den Gesamt-Pegel anheben, damit wird ein Ausgleich zur Kompression geschaffen.



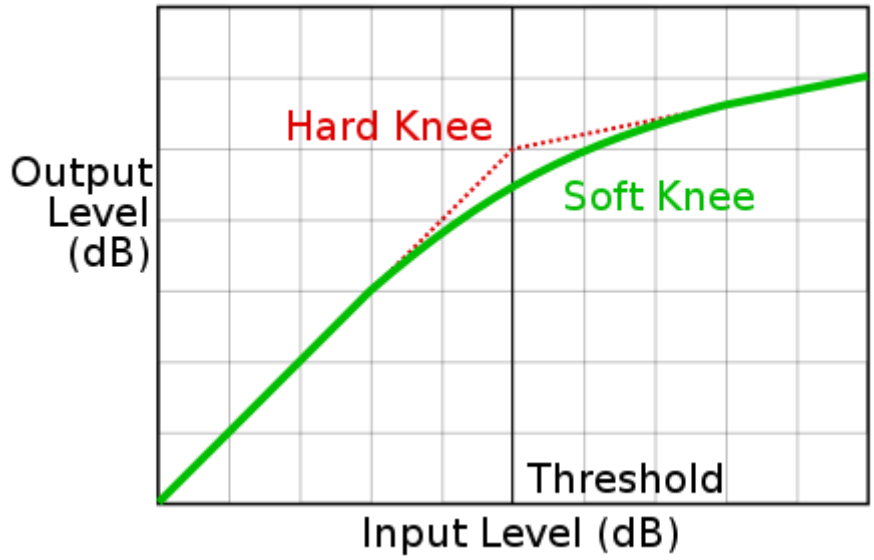
https://en.wikipedia.org/wiki/Dynamic_range_compression

Diese Graphik visualisiert, wie sich ratio und threshold verhalten sollen.

Wenn man einen Kompressor baut, muss man dafür zwei grundlegende Strukturen beachten. Die wichtigste ist die Funktion, die dafür sorgt, dass der Pegel gesenkt wird wie oben dargestellt, diese Funktion nannte ich *Reduktion*. Die zweite Funktion soll eine gewisse Toleranz für schnell lauter werdende Töne in den Kompressor einbauen, diese Funktion nannte ich *level detection*.

Reduktion:

Die Reduktion erfolgt, indem man zuerst bestimmte Wertebereiche festlegt, in denen der Kompressor reduziert, und wie er dies tut. Dies geschieht durch den threshold und die Kneewidth. Durch diese Variablen werden dann drei Bereiche festgelegt: einer ohne Kompression, das Softknee und das Hardknee.




```
def Reduktion(self,pegel):
    '''tatsächlich kompression der töne'''
    pegel = pegel.copy()
    for index, i in enumerate(pegel): #no kompression
        if 2*(i-self.threshold) < -self.knee_width:
            i=i+self.make_up_gain

        elif 2*(i-self.threshold)<= self.knee_width: # Softknee
            i=i+(1/self.ratio-1)*((i-self.threshold)+(self.knee_width/2))**2/(2*self.knee_width)+self.make_up_gain

        elif 2*(i-self.threshold)> self.knee_width: # Hardknee
            i=self.threshold+((i-self.threshold)/self.ratio)+self.make_up_gain

        pegel[index] = i
    return pegel
```

Hier wird auch das Make up gain zur Lautstärke hinzugefügt.

Level detection:

```
def level_detection(self,pegel):
    """ist ein filter der die geschwindigkeit mit der, der Kompressor eingreift veringert wird"""
    for index, i in enumerate(pegel):

        if index == 0: #nimmt sample aus vorherigen CHUNK
            i=((1-(self.alpha*self.Attack_timer))*self.Pegel_faktor[vars.CHUNK-1])+((self.alpha*self.Attack_timer)*i)
        else:
            i= (1-(self.alpha*self.Attack_timer))*pegel[index-1]+((self.alpha*self.Attack_timer)*i)
        pegel[index]=i
    return pegel
```

Die level detection arbeitet mit einem Filter. Dieser sorgt dafür, dass die Reduktion verlangsamt wird. Dazu wird immer die Lautstärke aus dem vorherigen sample berücksichtigt, über den Faktor alpha wird dann festgelegt, wie schnell der Attack des Kompressors ist. Mit Hilfe der Attack timer variable kann alpha beeinflusst werden. Diese Funktion macht die dynamic range Kompression erst möglich.

3. Struktur:

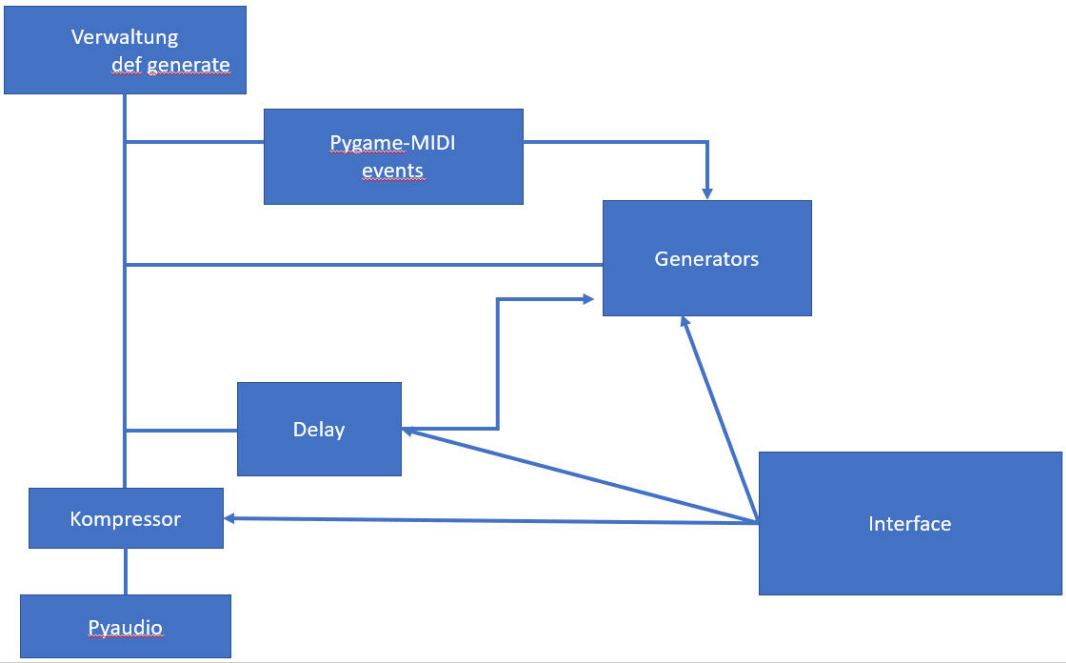
In diesem Kapitel geht es darum, wie die einzelnen Module ineinandergreifen. Beim Programmieren war das der Teil, der am aufwendigsten war. Dabei ist die Struktur relativ leicht zu verstehen, sobald man sich die Klasse „Verwaltung“ angeschaut hat. In dieser Klasse kommen alle Generatoren und andere Funktionen zusammen.

```
def generate(self,T):
    """gibt output, aus den generatoren, an play pyaudio weiter"""
    imk.generate_tasten()
    self.synth_Ausgabe=[]
    signal=0
    pegel=0
    vars.aktive_CHANNELS=len(self.spuren)+len(self.zombie_spuren)
    #generatoren erzeugen signal und pegel
    for i in self.spuren:
        self.synth_Ausgabe.append(self.spuren[i].init_sinus)

    for i in self.zombie_spuren:
        self.synth_Ausgabe.append(self.zombie_spuren[i].init_sinus)

    for i in self.delay_spuren:
        self.synth_Ausgabe.append(self.delay_spuren[i])

    for ge in self.synth_Ausgabe:
        signal+=(ge.generate(T))
        pegel+=(ge.timer)

    self.timestamp+=1
    if isinstance(signal,int) or isinstance(pegel,int):
        return self.komp.compress(np.zeros(int(T*vars.RATE)),np.zeros(int(T*vars.RATE)))
    else:
        signal*=vars.LAUTSTAERKE
        if vars.Echo: # delay
            self.generate_delay_spur(signal,self.timestamp,pegel*vars.LAUTSTAERKE)
        signal=self.komp.compress(pegel*vars.LAUTSTAERKE,signal)
        return signal
```

Imk.generate_tasten sorgt dafür, das callback vom midi keyboard zu bekommen.

Self.synth_ausgabe wird mit den Generatoren gefüllt und dann werden in den Variablen Pegel und Signal die Lautstärkekurve und das Tonsignal eingespeichert. Diese werden dann an den Kompressor weitergegeben, nachdem die Masterlautstärke beachtet wurde, ein delay kann dann optional auch noch hinzugefügt werden.

In folgendem Fließschema wird die Struktur des Synthesizers graphisch dargestellt.



Also checkt die generate-Funktion in Verwaltung zuerst nach allen Pygame-Midi Events. Sind solche vorhanden, beeinflussen diese die Generatoren. Jetzt werden alle Generatoren in synth ausgabe gesammelt und das Gesamtsignal und der Gesamtpegel ermittelt. Nun wird, wenn Echo True ist, ein

delay generator erzeugt. Jetzt geht das Signal durch den Kompressor und dann zu Pyaudio. Das interface hat Einfluss auf den Kompressor, das delay und die Generatoren.

4. Quellen:

zum Kompressor:

Titel: Digital dynamic Range Compressor Design - A Tutorial and Analysis, AES Memeber
authors: Dimitios Giannoulis, Micheal Massberg, Joshua D. Reiss
Publisher Queen Mary University of London, London, UK
-Stellt dar wie Digitale kompressoren Gebaut werden und Analysiert diese.

zum interface:

Tkinter Dokumentation:
https://docs.python.org/2/library/tkinter.html
Time: 07.04.2018/15:38
-Generelle Übersicht über Tkinter.

Tkinter Einführung:
https://www.python-kurs.eu/python_tkinter.php
Time: 07.04.2018/15:41
-Beispiele von Tkinter Nutzung.

pygame-midi:

pygame-midi Dokumentation:
https://www.pygame.org/docs/ref/midi.html
Time: 07.04.2018/15:51
-generelle Übersicht über pygame-midi

pygame:

pygmame Dokumentation:
https://www.pygame.org/docs/
Time: 07.04.2018/15:54
-generelle Übersicht über pygame

Fm-synthese:

Wikipediaseite:
https://de.wikipedia.org/wiki/FM-Synthese
Time: 07.04.2018/16:56
-bilder und Grundwissen