AUDIO-PROJEKT WS 17/18:

Teilnehmer: Arpad Krause

# Einführung:

Mein Projekt war es einen Synthesizer zu programmieren der auch ein wenig komplexere töne abspielen konnte, während dem programmieren habe ich mich über Dinge wie Sound-technik gelernt dieses Wissen habe ich dann angewendet um z.B den Kompressor zu bauen. Ansonsten habe ich mich mit Pygame, Tkinter und Pyaudio beschäftigt um den Synthesizer zu bauen und zu visualisieren, ich möchte noch erwähnen das ich kein Interesse daran hatte einen schönes oder praktisches interface auf zu bauen dies ist auch in der jetzigen Version nicht vorhanden.

# Gliederung:

1. Arbeitsverlauf
2. Grundlegende Struktur
3. Struktur

# 1. Arbeitsverlauf:

Zum Start der Projektarbeitsphase gab mir Stefan ein Programm welches mit Hilfe von Pyaudio eine simple Sinuskurve abspielen kann, Die Struktur mit dem dieses Programm den Ton herstellte verwende ich immer noch. Meine ersten Schritte bestanden also daraus zu verstehen wie dieses Programm funktioniert und wie ich dafür sorgen kann das man den abgespielten ton verändern konnte. Hier gab Stefan mir den Tipp Tkinter zu verwenden um eine Eingabemöglichkeit zu haben.

Als nächstes versuchte ich dafür zu sorgen das mehrere Töne gleichzeitig abgespielt werden konnten, allerdings war die Eingabe über die GUI immer noch sehr sperrig und unpraktisch. Ich nutze dann ein MIDI-keyboard welches dann dafür sorgte das die Eingabe erleichtert und mehrere töne gleichzeitig zu spielen tatsächlich auch eine praktische Sache war, da es mehr Möglichkeiten lieferte Töne zu komplizieren (variablere Töne), was ich jedoch bis Heute nicht nutzte.

Nun gab es natürlich immer noch große Probleme Bspw. hört man ein knacken wenn man ein Ton beendete, des weiteren gab es keinen Lautstärkeverlauf ein ton war dirket so laut wie es geht und wurde auch nicht leiser, also fehlte die Dynamik, hier kommt die Hull-Kurve in spiel die dafür sorgte dass, das knacken verschwand und eine Dynamik existierte.

Um bis dahin große probleme zu verhinderen hatte ich ein mischer eingebaut der dafür sorgte das es nichts zu störungen kommt auf grund der art wie Pyaudio die information liest, dazu wird später noch etwas gesagt. Brauchte ich etwas was den pegel kontrolliert, das war dann ein Kompressor.

Nun waren alle grundlegenden Strukturen vorhanden, jetzt konnte man noch dafür sorgen das es eine Soundsynthese mit mehr Möglichkeiten oder ein paar Soundeffekte gibt, das was ich darauf einbaute war ein Generator der mit der FM-synthese arbeitete, ein Resonanzgenerator und ein Delay.

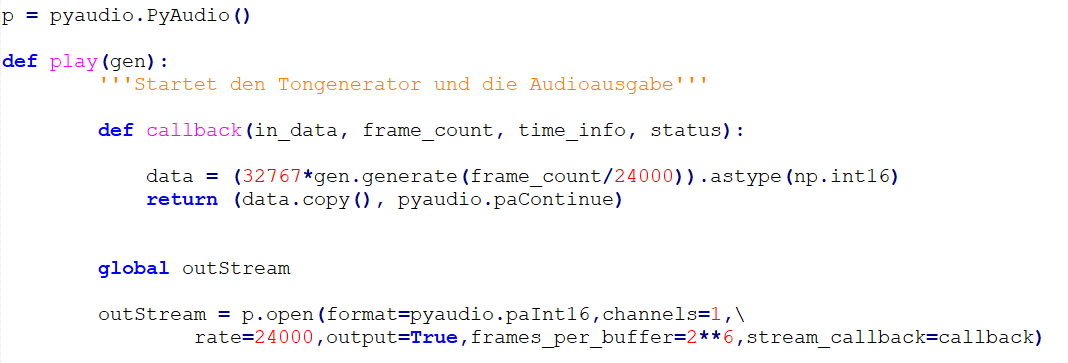
Das ist der momentane Standpunkt.

## 2.grundlegende Strukturen:

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

Pyaudio:

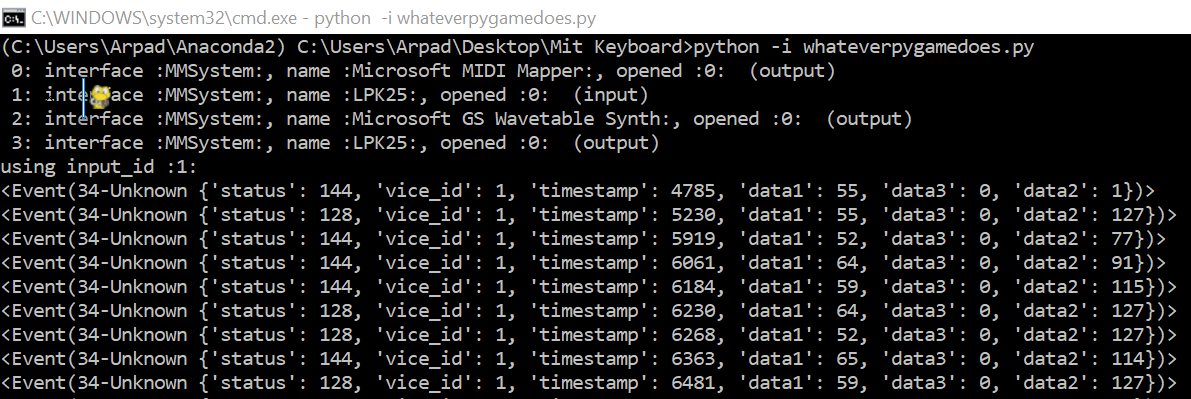
Pyaudio ist ein Modul von Python welches Informationen an die Soundkarte weiter geben kann, die dann abgespielt werden, dies Funktioniert über ein callback



Mit der Hilfe einer callbackfunktion wird dann das callback aufrecht gehalten. Im code wird man oft die Variable RATE finden diese speichert den integer 24000 der hier auch in dem Pyaudio object eingespeichert wurde. Diese RATE könnte auch bspw. 44100 Hz sein was dann der samplingrate von einer Wave-Datei entsprechen würde. Die frames\_per\_buffer variable definiert wie groß ein chunk ist die Größe des chunks bestimmt aller arrays die erzeugt werden müssen. Die arrays dürfen nur floats mit einem Wert zwischen 1 und -1 enthalten.

Pygame-midi-eingabe:

In pygame gibt es ein modul „pygame-midi“ welches daten von eine Midi keyboard abfangen kann. Hier sieht man ein Beispiel von den daten die man abfängt.

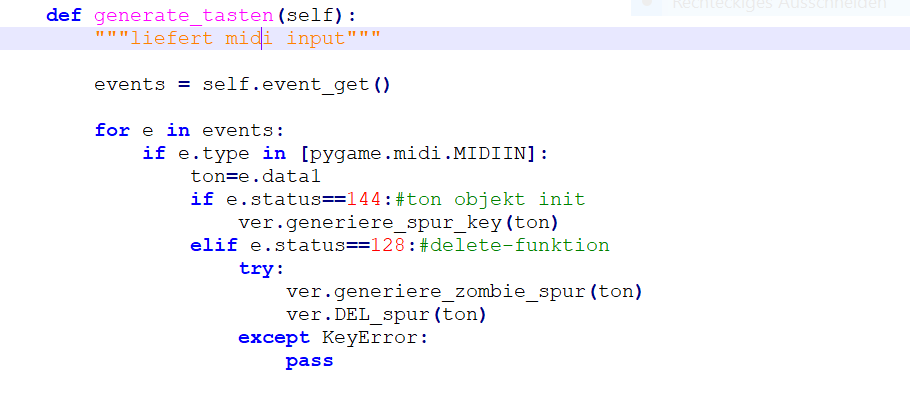


Die Informationen die hier wichtig sind beginnen ab der 5. Zeile.

Man bekommt für jede Eingabe,jedes event auf dem midi-keyboard, ein dictionary mit den werten. „status“, „timestamp“, „vice\_id“, „data1“, „data3“ und „data2“ zwei zurück gegeben die variable „status“ gibt an ob die Taste gedrückt oder losgelassen wird, 144 = drücken und 128 = loslassen, „vice\_id“ bestimmt von welchen gerät diese Eingabe kommt, „timestamp“ war für mich irrelevant und bedarf keiner nähren Erläuterung, „data1“ gibt an welche taste gedrückt wurde, „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.

Für mich waren nur zwei Variablen, „status “ und „data1“ , relevant allerdings hätte man „data2“ auch noch abfangen lassen können und mit nutzen können um mehr Dynamik einzubauen, jedoch habe ich noch kein Konzept entwickelt welches ich gut fand um diesen Wert einbauen zu können.

Das abfangen der Daten läuft über diese Funktion ab:



Pygame-midi fängt alle Ereignisse ab und während das Programm läuft wird nach den events gefragt dieses im folgenden ausgewertet: wenn die Taste gedrückt wird, wird eine neue Spur generiert und wenn die Taste losgelassen wird wir eine „zombie-spur“ , dazu später mehr (im währen Hull-kurve), generiert und eine Spur gelöscht die den selben ton spielt, wie die neue „zombie-spur“.

Ton-generation:

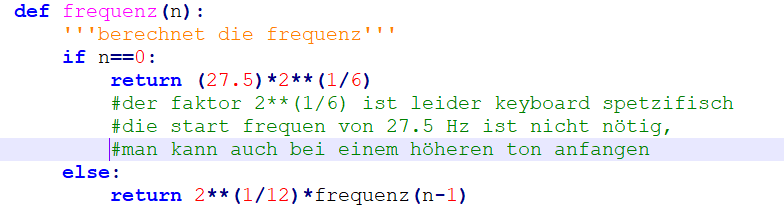
ich zeige hier nur das simpelste Beispiel:

Für die Generation eine Tones benötigt man die Frequenz, das ist die Variable die einen Ton von anderen unterscheidet, von Pygame midi wurde ein Index für jeden Ton zurück gegeben, ein integer mit einem positiven wert z.b 40, Null mit inbegriffen.

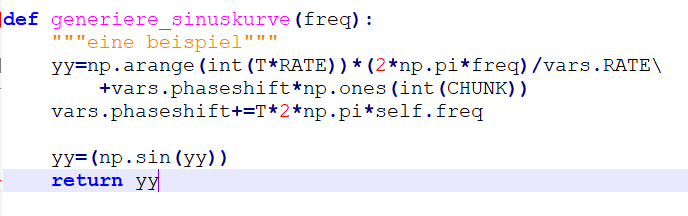
Man weiß, das ton(n) und ton(n+1) Frequenzen in einer bestimmten Beziehnung stehen: Frequenz(ton(n+1))=Frequenz(ton(n))\* 2\*\*(1/12)

Jetzt benötigt man nur noch einen Ton mit einer bestimmten Frequenz, hier bietet sich der Kammerton: A an, A hat eine Frequenz von 440 Hz.

Man kann den Ton also bestimmen wenn man diese beiden Dinge weiß, daraus leitet sich auch ab das 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: 

Die für Pyaudio wichtige Sinuskurve wird wie folgt berechnet:



Die Variablen RATE und CHUNK sind variablen die für Pyaudio wichtig sind. Rate entspricht den samplen pro Sekunde und Chunk der Größe eines arrays und ist eine variable die vom callback geliefert wird.

Phaseshift sorgt dafür das nicht immer der gleiche Bereich der Sinuskurve betrachtet wird sondern die Sinuskurve „fortschreitet“, also sich die Phase verschiebt.

Nun habe ich während mehrere Generatoren gebaut, die alle dieses Prinzip benutzen, zwei von den dreien die ich bisher gebaut habe benutzen dabei keine einfache Sinuskurve sondern andere Syntheseverfahren.

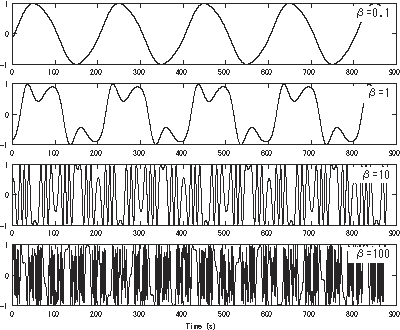
FM-synthese:

Die FM-synthese eröffnet einem die Möglichkeiten viele unterschiedliche töne mit nur zwei Sinuskurven erzeugen zu können, anstatt eine simple ton-kurven zu verwenden,und diese zu addieren, wir bei der FM-synthese die ton-kurve moduliert mit einer anderen ton-kurve,. Die Mathematik dahinter sieht folgend aus:



Dadurch werden 2 neue variablen in die Tongeneration eingebaut im Vergleich zu davor war es nur eine pro Ton. Man muss nur den Faktor: beta und die Frequenz der modulationskurve durch globale Variablen das wird in meinem Programm von dem Interface übernommen.

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 addierten Sinuskurven darstellen kann jedoch wäre die benötigte Rechenleistung dafür enorm, deshalb ist dieses Verfahren so interessant gewesen zu Zeiten wo Computer nicht so schnell wahren wie heute.

Resonanz:

In Klavieren usw. schwingen andere Saiten immer mit, wenn ein Ton angeschlagen wird, es schwingen alle Oktaven mit, Klaviere usw. haben aufgrund der Resonanz sehr komplexe Töne die sich sehr natürlich anhören, auch für Synthesizer wurde dieses Prinzip aufgegriffen, bei Synthesizern ist das Prinzip natürlich sehr viel „unorganischer“ und man errechnet einfache eine über den Resonanzfaktor bestimmte Menge an neben Frequenzen aus, mit einer vom Resonanzfaktor bestimmten Lautstärke wodurch man dann ein erhöhtes klang Spektrum hat ähnlich das eines Klavier.

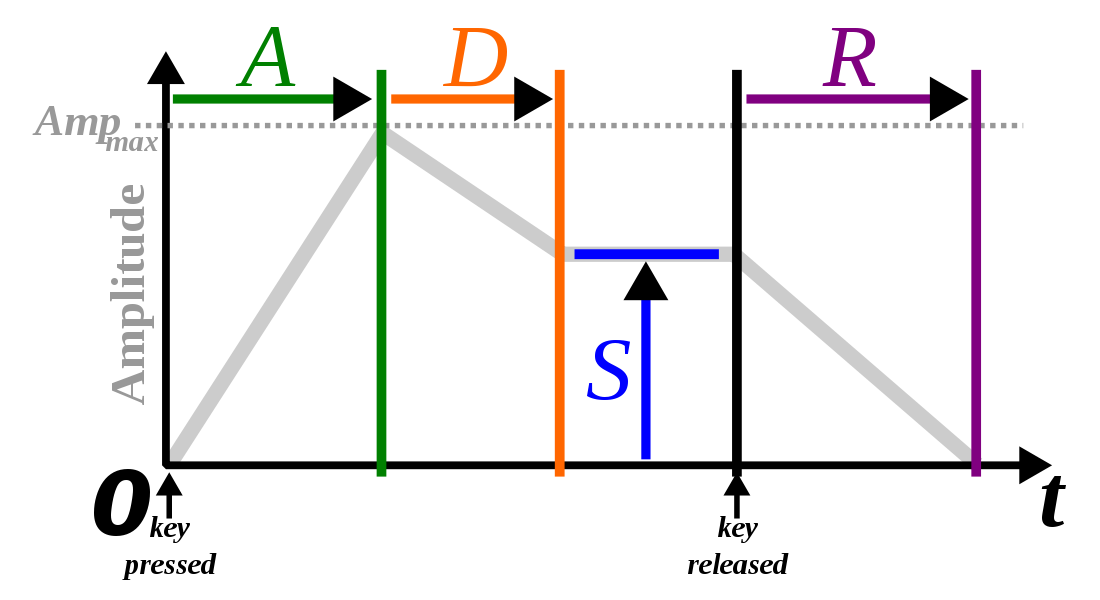
Kommentar:

Die Simulation der Resonanz ist theoretisch relativ komplex während der Arbeit an dieser Art von Soundsynthese stellte sich heraus das die „Resoanzberechenung zu enorm ist anstatt nur eine Sinuskurve pro Ton oder zwei bei der FM-synthese hat man auf einmal n, wird über den Resonanzfaktor bestimmt, viele Spuren pro Ton, das führte letztendlich dazu das ich eine stark vereinfachte Art der Resonanz programmieren konnte, allerdings für töne die aus der FM-synthese entstanden oder eine zu große Anzahl von neben Frequenzen reicht die Rechenleistung und Speicherkapazität eines(meines) Computers nicht aus.

Hullkurve:

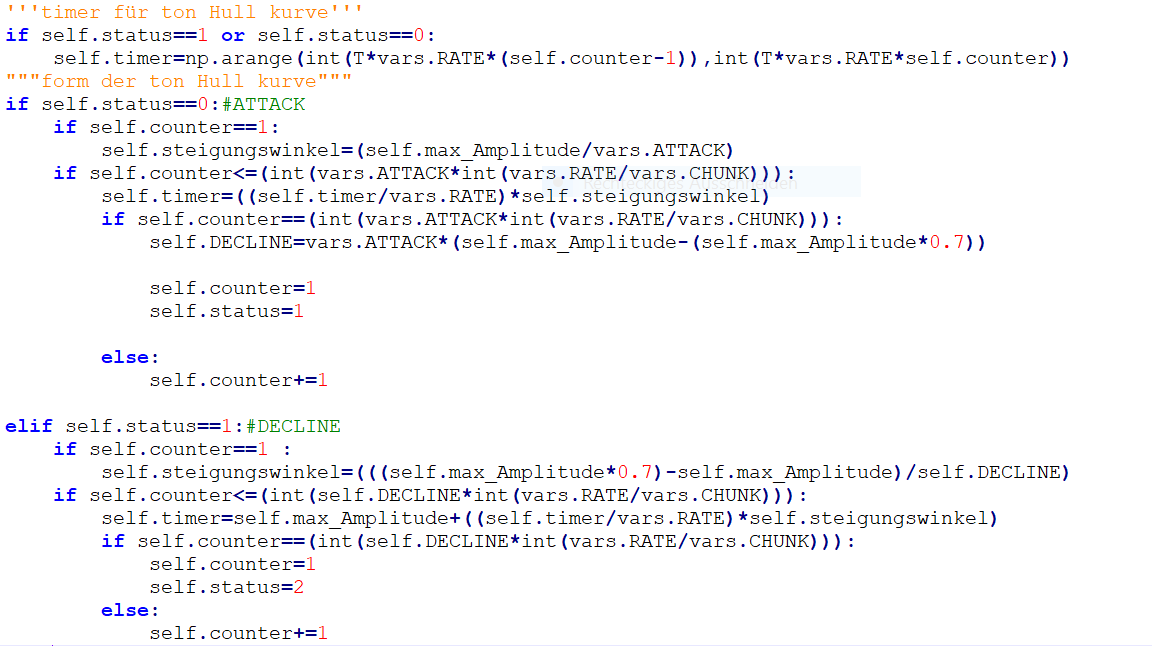
Die Hullkurve ist eine Art den Ton verlauf, Lautstärke Verlauf eines Tones, zu gestalten. Dabei gibt teilt man den Ton verlauf in 4 Phasen ein Attack, Decay, Sustain und Release, deswegen wird die kurve auch ADSR-Kurve genannt, damit eine gewisse Dynamik entsteht.

Die Ersten beiden Phasen: Attack und Decay sorgen dafür das man so etwas wie einen Anschlag vernimmt sobald der Attack vorbei also die maximal Lautstärke des Anschlages erreicht wurde wird die Phase zum Decay geändert in dieser Phase sinkt die Lautstärke auf das level des Sustian, die nächste Phase, diese Phase ist die einzige Phase ohne Zeitbeschränkung in dieser Phase gibt es auch keine Veränderung der Lautstärke wenn die taste los gelassen wird setzt dann das Release ein, welches dafür sorgt das der Ton ausklingt. Was noch zu erwähnen ist, der Release wird in einem extra Generator erzeugt, dieser brauchte eine Deletefunktion.



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 praktisch Lautstärke und Signal Unabhängig von ein andere bearbeitet werden bis zur Weitergabe. Die variablen Decay, Attack und Release bestimmen die Dauer von jedem abschnitt bis auf den Sustain, der Sustain gibt seine eigen Lautstärke an also hat man jeweils ein 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:

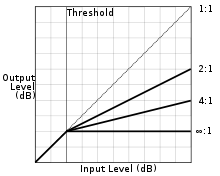


self.status ist eine Variable die angibt in welcher Phase man sich befindet, 1 steht für Attack, 2 für Decay und 3 für sustain, release ist in einem anderen Generator ,diese Generatoren haben den Namen zombie generatoren. Momentan wir die Dauer jeder Phase über eine Variable bestimmt.

Kommentar: In der Musik ist das natürlich einer der wichtigstem Unterschiede zu den „herkömmlichen“ Instrumenten, aufgrund der höheren Kontrolle von der Dynamic, hier habe ich, bis her, die Dynamic von einzelnen Tönen für nicht besonders wichtig gehalten, deshalb die tatsächlichen Möglichkeiten hier sehr gering.

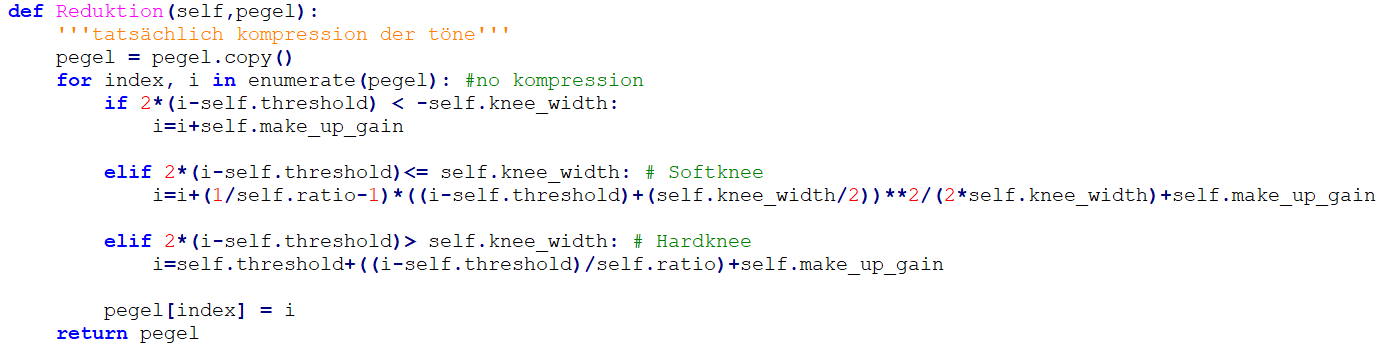
Kompressor:

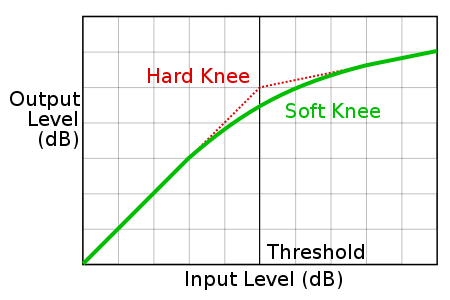
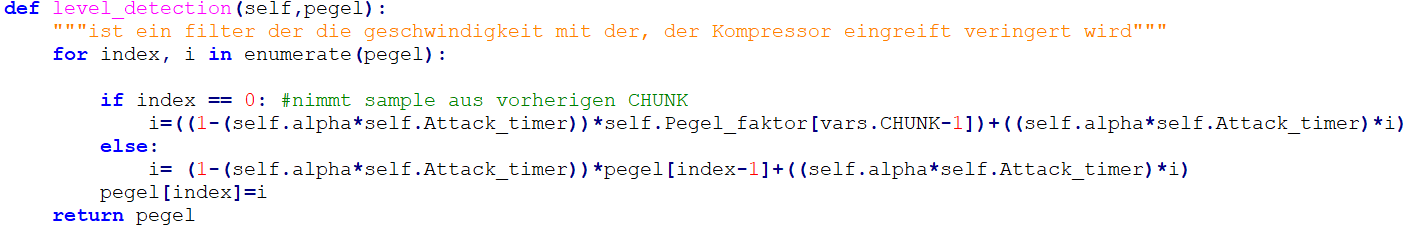
Um den Output zu kontrollieren und für Pyaudio lesbar zu machen habe ich einen Kompressor einbauen müssen. Ich entschied mich dann für einen dynamic range kompressor diese versuchen obwohl es eine Kompression gibt noch dafür zu sorgen das es ein wenig dynamic gibt. Ohne dieses Konzept würde beim drücken von mehreren tasten langsam alle töne egal wie laut sie eigentlich sind gleich mäßig leiser werden wodurch kein attack , Anschlag, mehr war zu nehmen wäre. Mein Kompressor braucht die Variablen: Threshold, Ratio, Kneewidth ,Attacktimer und Make up gain, wobei der Threshold die Lautstärke bestimmt bei der , der Kompressor anspringt, die Ratio den Faktor angibt mit wie Stark die Kompression ist wobei 1 keiner kompression und bspw: 60 nahezu limitting entspricht, also einer sehr starke kompression, die Kneewidth bestimmt die größe des softknees welches dafür sorgt das der unterschied zwischen Kompression und nicht Kompression geschmeidiger verläuft, kompression wird dann auch als das Hardknee bezeichnet, Attack timer ist ein faktor Herz der Dynamic range kompression dardurch wird ermöglich das peeks im Lautstärke verlauf möglich sind und das Make up Gain diese sorgt dafür das man den gesamten Pegel nutzen kann und leise töne im vergleich zu lauten tönen immer noch leise wirken auch wenn die durch den Kompressor leiser gemacht wurden.

 https://en.wikipedia.org/wiki/Dynamic\_range\_compression

diese Graphik visualisiert wie sich ratio und threshold verhalten sollen.

Wenn man eine kompressor baut muss man dafür zwei grundlegende strukturen, ein mal die funktion die dafür sorgt das das soft knee und hard knee genutzt werden und das, dass eingreifen des Kompressors verlangsamt wird so das eine gewisse toleranz für peeks gibt man betrachtet in einem Kompressor nur die Lautstärke also den gesamt Pegel, dieser ist die addition aller Hull-Kurven, das signal ist erst einmal nicht interesant, es gibt auch kompressoren die direkt auf das Signal angewendet werden jedoch sorgt man dann dafür das Nebenschwingungen entstehen die nicht geplant sind.





Dieses Graphik soll darstellen was soft und hardknee sind.

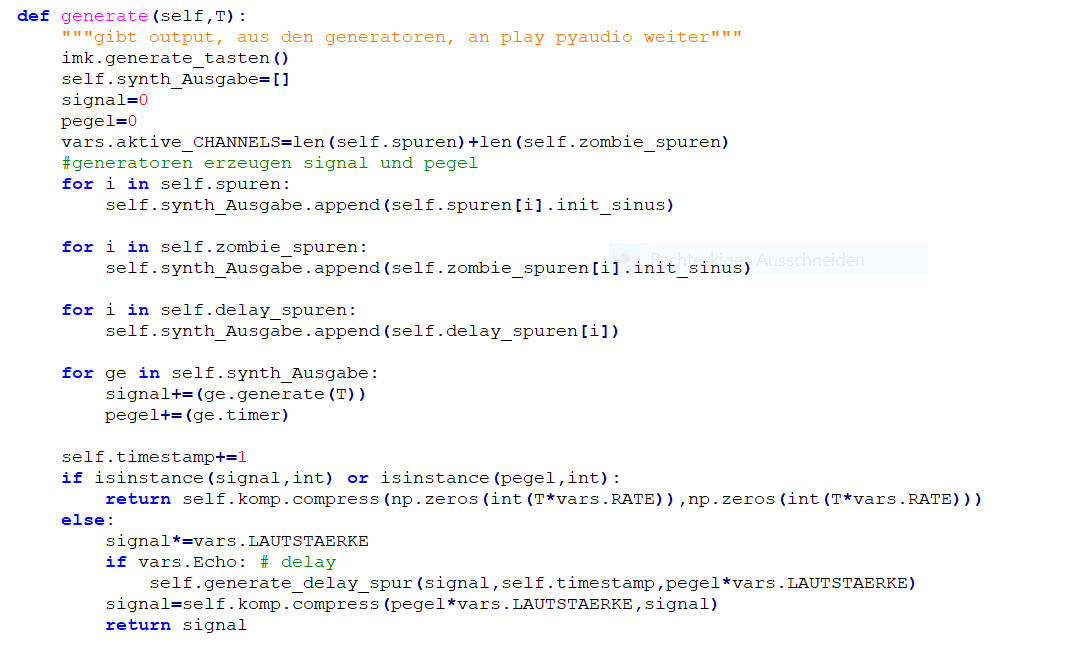
Hier wird dafür gesorgt das alle Töne die sich im Bereich des softknees und hardknees dem entsprechend komprimiert werden, des weiteren wird das make\_up\_gain auf die Lautstärke addiert. Bei dieser Funktion wird allerdings nicht die dynamic beachtet. Dafür benötigt man einen Filter der das ansteigen der kompression vom vorherigen ton abhängig macht:

Hier kommt auch der Attacktimer ins spiel, in vielen Kompressoren findet man auch einen Decay timer dieser wird aktiviert so bald die Lautstärke abfällt damit kann man den kompressor noch besser kontrollieren.

Kommentar: Ein Kompressor ist eine sehr Komplexe sache man kann in unterschiedlichen Genres und sogar im gleichen genre auf unterschiedliche einstellung die den Sound komplett verändern. Der Kompressor den ich hier Programmiert habe, mit Hilfe von einem Skript von Stefan und Stefan, ist immer noch sehr simple die wichtigsten Sachen sind beachtet allerdings könnte man noch ein „besseren“ Kompressor bauen wobei hier allerdings genau das Problem der Individualität dar ich noch nicht weiß was hier wirklich benötigt wurde habe ich noch keine großen extras eingebaut werden, auch jetzt bin ich mir immer noch nicht sicher ob der ganze Kompressor so funktioniert wie er funktionieren soll.

# 3. Struktur:

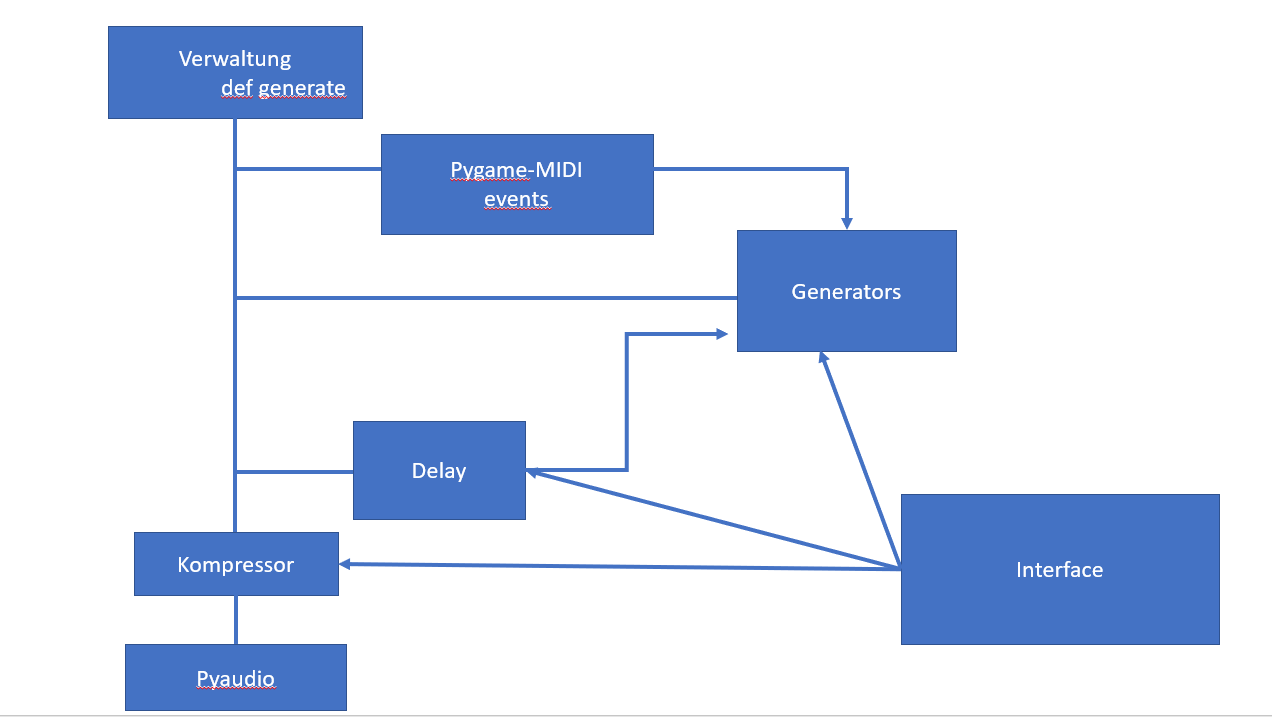
In diesem Kapitel geht es darum wie die einzelnen Module zusammen greifen, beim programieren war das der Teil an dem ich am meisten Zeit verbrauchte, dabei ist die Struktur relativ leicht zu verstehen sobald man sich die Klasse „Verwaltung“ angeguckt hat in dieser Klasse kommen alle Generatoren und andere Funktionen zusammen.



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 Lautstärke kurve und das Tonsignal eingespeichert diese werden dann an den Kompressor weiter gegeben nach dem die Masterlautsärke beachtet wurde, ein delay kann dann optional auch noch hinzugefügt werden.

In diesem Fließschema wird die Struktur des Synthesizers graphisch dargestellt.



Pip Feeze data:

alabaster==0.7.10

anaconda-client==1.6.5

anaconda-navigator==1.6.8

anaconda-project==0.8.0

asn1crypto==0.22.0

astroid==1.5.3

astropy==2.0.2

babel==2.5.0

backports-abc==0.5

backports.functools-lru-cache==1.4

backports.shutil-get-terminal-size==1.0.0

backports.ssl-match-hostname==3.5.0.1

beautifulsoup4==4.6.0

bitarray==0.8.1

bkcharts==0.2

blaze==0.11.3

bleach==2.0.0

bokeh==0.12.7

boto==2.48.0

Bottleneck==1.2.1

CacheControl==0.12.3

cdecimal==2.3

certifi==2017.7.27.1

cffi==1.10.0

chardet==3.0.4

click==6.7

cloudpickle==0.4.0

clyent==1.2.2

colorama==0.3.9

comtypes==1.1.2

conda==4.3.30

conda-build==3.0.22

conda-verify==2.0.0

configparser==3.5.0

contextlib2==0.5.5

cryptography==2.0.3

cycler==0.10.0

Cython==0.26.1

cytoolz==0.8.2

dask==0.15.2

datashape==0.5.4

decorator==4.1.2

distlib==0.2.5

distributed==1.18.3

docutils==0.14

entrypoints==0.2.3

enum34==1.1.6

et-xmlfile==1.0.1

fastcache==1.0.2

filelock==2.0.12

Flask==0.12.2

Flask-Cors==3.0.3

funcsigs==1.0.2

functools32==3.2.3.post2

futures==3.1.1

gevent==1.2.2

glob2==0.5

greenlet==0.4.12

grin==1.2.1

h5py==2.7.0

heapdict==1.0.0

html5lib==0.999999999

idna==2.6

imageio==2.2.0

imagesize==0.7.1

ipaddress==1.0.18

ipykernel==4.6.1

ipython==5.4.1

ipython-genutils==0.2.0

ipywidgets==7.0.0

isort==4.2.15

itsdangerous==0.24

jdcal==1.3

jedi==0.10.2

Jinja2==2.9.6

jsonschema==2.6.0

jupyter-client==5.1.0

jupyter-console==5.2.0

jupyter-core==4.3.0

jupyterlab==0.27.0

jupyterlab-launcher==0.4.0

lazy-object-proxy==1.3.1

llvmlite==0.20.0

locket==0.2.0

lockfile==0.12.2

lxml==3.8.0

MarkupSafe==1.0

matplotlib==2.0.2

mccabe==0.6.1

menuinst==1.4.8

mistune==0.7.4

mpmath==0.19

msgpack-python==0.4.8

multipledispatch==0.4.9

navigator-updater==0.1.0

nbconvert==5.3.1

nbformat==4.4.0

networkx==1.11

nltk==3.2.4

nose==1.3.7

notebook==5.0.0

numba==0.35.0+10.g143f70e.dirty

numexpr==2.6.2

numpy==1.13.1

numpydoc==0.7.0

odo==0.5.1

olefile==0.44

openpyxl==2.4.8

packaging==16.8

pandas==0.20.3

pandocfilters==1.4.2

partd==0.3.8

path.py==10.3.1

pathlib2==2.3.0

patsy==0.4.1

pep8==1.7.0

pickleshare==0.7.4

Pillow==4.2.1

pkginfo==1.4.1

ply==3.10

progress==1.3

prompt-toolkit==1.0.15

psutil==5.2.2

py==1.4.34

PyAudio==0.2.11

pycodestyle==2.3.1

pycosat==0.6.2

pycparser==2.18

pycrypto==2.6.1

pycurl==7.43.0

pyflakes==1.5.0

pygame==1.9.3

Pygments==2.2.0

pylint==1.7.2

pynput==1.3.7

pyodbc==4.0.17

pyOpenSSL==17.2.0

pyparsing==2.2.0

PySocks==1.6.7

pytest==3.2.1

python-dateutil==2.6.1

pytz==2017.2

PyWavelets==0.5.2

pywin32==221

PyYAML==3.12

pyzmq==16.0.2

QtAwesome==0.4.4

qtconsole==4.3.1

QtPy==1.3.1

requests==2.18.4

rope==0.10.5

ruamel-yaml==0.11.14

scandir==1.5

scikit-image==0.13.0

scikit-learn==0.19.0

scipy==0.19.1

seaborn==0.8

simplegeneric==0.8.1

singledispatch==3.4.0.3

six==1.10.0

snowballstemmer==1.2.1

sortedcollections==0.5.3

sortedcontainers==1.5.7

Sphinx==1.6.3

sphinxcontrib-websupport==1.0.1

spyder==3.2.3

SQLAlchemy==1.1.13

statsmodels==0.8.0

subprocess32==3.2.7

sympy==1.1.1

tables==3.4.2

tblib==1.3.2

testpath==0.3.1

toolz==0.8.2

tornado==4.5.2

traitlets==4.3.2

typing==3.6.2

unicodecsv==0.14.1

urllib3==1.22

wcwidth==0.1.7

webencodings==0.5.1

Werkzeug==0.12.2

widgetsnbextension==3.0.2

win-inet-pton==1.0.1

win-unicode-console==0.5

wincertstore==0.2

wrapt==1.10.11

xlrd==1.1.0

XlsxWriter==0.9.8

xlwings==0.11.4

xlwt==1.3.0

zict==0.1.2