VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

DIPLOMOVÁ PRÁCE

Brno, 2023

Bc. Daniel Lazorčák

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV TELEKOMUNIKACÍ

DEPARTMENT OF TELECOMMUNICATIONS

SYNTÉZA ZVUKU Z VIDEO DAT

SEMESTRÁLNÍ PRÁCE

SEMESTRAL THESIS

AUTOR PRÁCE — **Bc. Daniel Lazorčák**

AUTHOR

VEDOUCÍ PRÁCE — **doc. Ing. Kamil Říha, Ph.D.**

SUPERVISOR

BRNO 2021

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Audio inženýrství**
specializace Zvuková produkce a nahrávání
Ústav telekomunikací

Student: Bc. Daniel Lazorčák ***ID:*** 203738
Ročník: 2 ***Akademický rok:*** 2022/23

NÁZEV TÉMATU:

Syntéza zvuku z video dat

POKYNY PRO VYPRACOVÁNÍ:

Nastudujte pokročilé možnosti převodu obrazových a video dat na zvuková. Naprogramujte aplikaci, která umožní interaktivní provádění různých druhů tohoto převodu s možností volby libovolného obrazu či videa a s poslechem vytvořeného zvuku. Je předpokládáno, že aplikace umožní několik možností analýzy obrazového obsahu, např. s respektováním barev, výskytu detailů, výskytu obličejů, započítání časové osy snímků apod. Tato aplikace bude obsahovat minimálně tři různé možnosti analýzy.

DOPORUČENÁ LITERATURA:

[1] GONZALEZ, R. C.; WOODS, R. E.: Digital Image Processing, Prentice Hall, New Jersey, 2002.
[2] BRADSKI, G.; KAEHLER, A.: Learning OpenCV: Computer Vision with the OpenCV Library, O'Reilly Media, Inc. USA 2008, ISBN: 978-0-596-51613-0.

Termín zadání: 6.2.2023 ***Termín odevzdání:*** 19.5.2023

Vedoucí práce: doc. Ing. Kamil Říha, Ph.D.

doc. Ing. Jiří Schimmel, Ph.D.
předseda rady studijního programu

UPOZORNĚNÍ:

Autor diplomové práce nesmí při vytváření diplomové práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

ABSTRAKT

V tejto diplomovej práci je vyhotovená rešerš metód syntézy zvuku z obrazových a video dát na dáta zvukové a relizovaná je implementácia troch nových metód syntézy. Prvá časť práce poskytuje prehľad o existujúcich prístupoch k tvorbe zvuku z obrazu, identifikuje ich výhody, obmedzenia a prípadné možnosti rozšírenia. V druhej časti je popísaná implementácia aplikácie Vsyntha, ktorá syntetizuje zvuk z videa v reálnom čase s možnosťou ovládania hudobných parametrov. V tretej časti je opísaná aplikácia ReAmper, ktorá ozvučuje scény pomocou zvukových objektov a hudobných tónov na základe detekcie a sledovania objektov v obraze. Vo štvrtej časti je popísaná aplikácia SegMentor, ktorá vytvára MIDI súbory z videa pomocou rôznych techník segmentácie obrazu. Implementované metódy poskytujú nové nástroje pre tvorbu zvuku a multimediálnych diel, otvárajú priestor pre ďalší výskum a vývoj v oblasti syntézy zvuku z obrazu a poskytujú užitočné nástroje pre tvorbu zvukového obsahu a interakciu s vizuálnymi dátami vo forme zvuku. Výsledky tejto práce poskytujú prehľad o súčasnom stave výskumu a praxe v tejto oblasti a ponúkajú možnosti pre ďalší rozvoj a aplikácie v praxi.

KĽÚČOVÉ SLOVÁ

aplikácie, detekcia objektov, mapovanie dát, metódy syntézy, MIDI, obrazové dáta, ozvučovanie scény, python, segmentácia obrazu, sledovanie objektov, syntéza zvuku, video dáta

ABSTRACT

In this thesis, a survey of audio synthesis methods from image and video data to audio data is performed and the implementation of three new synthesis methods is reviewed. The first part of the thesis provides an overview of existing approaches to sound from image, identifying their advantages, limitations and possible extensions. The second part describes the implementation of VSyntha, an application that synthesizes audio from video in real-time with the ability to control musical parameters. The third section describes the ReAmper application, which performs soundscaping using sound objects and musical cues based on the detection and tracking of objects in the image. The fourth section describes the SegMentor application, which creates MIDI files from video using various image segmentation techniques. The implemented methods provide new tools for the creation of audio and multimedia works, open the way for further research and development in the field of sound-from-image synthesis, and provide useful tools for creating audio content and interacting with visual data in the form of audio. The results of this work provide an overview of the current state of research and practice in this area and offer opportunities for further development and applications in practice.

KEYWORDS

applications, audio synthesis, data mapping, image data, image segmentation, MIDI, object detection, object tracking, python, soundscaping, synthesis methods, video data.

Vysadené pomocou balíčku `thesis` verzie 4.07; `http://latex.feec.vutbr.cz`

LAZORČÁK, Daniel. *Syntéza zvuku z video dat*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2023, 85 s. Semestrálna práca. Vedúci práce: doc. Ing. Kamil Říha, PhD.

Vyhlásenie autora o pôvodnosti diela

Meno a priezvisko autora: Bc. Daniel Lazorčák

VUT ID autora: 203738

Typ práce: Semestrálna práca

Akademický rok: 2022/23

Téma závěrečnej práce: Syntéza zvuku z video dat

Vyhlasujem, že svoju záverečnú prácu som vypracoval samostatne pod vedením vedúcej/ceho záverečnej práce, s využitím odbornej literatúry a ďalších informačných zdrojov, ktoré sú všetky citované v práci a uvedené v zozname literatúry na konci práce.

Ako autor uvedenej záverečnej práce ďalej vyhlasujem, že v súvislosti s vytvorením tejto záverečnej práce som neporušil autorské práva tretích osôb, najmä som nezasiahol nedovoleným spôsobom do cudzích autorských práv osobnostných a/alebo majetkových a som si plne vedomý následkov porušenia ustanovenia § 11 a nasledujúcich autorského zákona Českej republiky č. 121/2000 Sb., o práve autorskom, o právach súvisiacich s právom autorským a o zmene niektorých zákonov (autorský zákon), v znení neskorších predpisov, vrátane možných trestnoprávnych dôsledkov vyplývajúcich z ustanovenia časti druhej, hlavy VI. diel 4 Trestného zákonníka Českej republiky č. 40/2009 Sb.

Brno ....

podpis autora*

*Autor podpisuje iba v tlačenej verzii.

POĎAKOVANIE

Rád by som poďakoval vedúcemu diplomovej práce pánovi doc. Ing.Kamilovi Říhovi, Ph.D. za odborné vedenie, konzultácie, trpezlivosť a podnetné návrhy k práci.

Obsah

Úvod **13**

1 Rešerš metód syntézy zvuku z obrazových alebo video dat **14**
1.1 Syntéza zvuku z obrázkov videa v reálnom čase . . . 14
1.2 Filtrácia šumu videa na audio s detekciou pohybu . . . 15
1.3 Vytváranie zvukov z obrazových dát . . . 16
1.4 Photosounder . . . 18
1.5 Syntéza zvuku zo získaného tvaru na obrázku . . . 19
1.6 Aplikácia obrazových sonifikáčných metód na hudbu . . . 20
1.7 Prostredie SONIFYD . . . 21
1.8 Sonifikácia pomocou rastrového skenovania . . . 23
1.9 Iné práce . . . 24

2 VSyntha **25**
2.1 Prostredie . . . 25
2.2 Knižnice . . . 25
2.2.1 Knižnica PyQT5 . . . 26
2.2.2 Knižnica PYO . . . 26
2.2.3 Knižnica OpenCV . . . 26
2.2.4 Knižnica NumPy . . . 26
2.3 Grafické užívateľské prostredie . . . 27
2.4 Prehrávanie videa . . . 29
2.5 Syntéza . . . 30
2.5.1 Mapovanie dát farebného priestoru na dáta zvukového signálu 31
2.5.2 Mapovanie dát histogramu na dáta ekvalizačných filtrov . . . 32
2.6 Generátor zvuku . . . 33
2.7 Rýchlosť algoritmu . . . 35
2.8 Zhodnotenie a možné vylepšenia . . . 35

3 ReAmper **37**
3.1 Knižnice a moduly . . . 37
3.1.1 SciPy . . . 37
3.1.2 Difflib . . . 37
3.1.3 Random . . . 38
3.1.4 Soundfile . . . 38
3.2 Grafické užívateľské prostredie . . . 38
3.3 Objekty . . . 41

3.3.1 Detekcia objektov . . . 41
3.3.2 Sledovanie objektov a detekcia pohybu . . . 42
3.4 Zvuková banka . . . 46
3.4.1 Zvuky objektov . . . 46
3.4.2 Hudobná banka . . . 47
3.5 Spracovanie zvuku . . . 47
3.5.1 Korekcia objektových máp . . . 48
3.5.2 Nastavenie dĺžky signálu . . . 50
3.5.3 HRTF panoráma . . . 50
3.5.4 Sčítanie zvukov . . . 52
3.6 Rýchlosť metódy . . . 54
3.7 Zhodnotenie a možné vylepšenia . . . 54

4 SegMentor **55**
4.1 Knižnice a moduly . . . 55
4.1.1 HMMlearn . . . 55
4.1.2 Mido . . . 55
4.2 Grafické užívateľské prostredie . . . 56
4.3 Segmentácie . . . 58
4.3.1 Prahová segmentácia . . . 59
4.3.2 Watershed segmentácia . . . 60
4.3.3 Segmentácia na báze detekcie hrán . . . 63
4.3.4 Segmentácia pomocou skrytého Markovovho modelu . . . 64
4.3.5 Segmentácia pomocou K-Means . . . 66
4.3.6 Diskrétna kosinusová transformácia . . . 67
4.4 Mapovanie dát z výstupného tenzora na hudobné parametre . . . 69
4.5 Hudobné parametre . . . 69
4.5.1 Vytvorenie rytmickej mapy . . . 70
4.5.2 Vytvorenie melodickej mapy . . . 70
4.5.3 Vytvorenie harmonicko-melodickej mapy . . . 71
4.5.4 Vytvorenie oktávovej mapy . . . 71
4.5.5 Vytvorenie intervalovej mapy . . . 71
4.6 MIDI . . . 72
4.7 Rýchlosť metódy . . . 76
4.8 Zhodnotenie a možné vylepšenia . . . 77

Záver **79**

Literatúra **81**

Zoznam symbolov a skratiek **83**

Zoznam príloh **84**

A Obsah elektronickej prílohy **85**

Zoznam obrázkov

1.1 Proces získania hodnôt amplitúd spektrálnych zložiek 14
1.2 Grafické užívateľské prostredie aplikácie *Bzučák* 16
1.3 Grafické užívateľské prostredie aplikácie *Granulka* 17
1.4 Grafické užívateľské prostredie aplikácie *Sólo pro obraz* 18
1.5 Grafické užívateľské prostredie aplikácie *Photosounder* 19
1.6 Princíp separácie stôp z tvaru aplikácie *Pyc2Sound* 20
1.7 Užívateľské prostredie aplikácie *SonART* 21
1.8 Princíp aplikácie *Sonifyd* 22
1.9 Princíp rastrového skenovania 23
2.1 Grafické užívateľské prostredie aplikácie *Vsyntha* 27
2.2 Vývojový diagram grafického užívateľského prostredia aplikácie *Vsyntha* 28
2.3 Vývojový diagram bloku *Video Player* aplikácie *VSyntha* 29
2.4 Vývojový diagram bloku *Synthesis* 30
2.5 Vývojový diagram mapovania obrazových dát na zvukové 31
2.6 Vývojový diagram bloku mapovania dát histogramu 32
2.7 Vývojový diagram bloku *Syntéza* 33
3.1 Grafické užívateľské prostredie aplikácie ReAmper 39
3.2 Diagram GUI aplikácie ReAmper 40
3.3 Výstup kombinácie YOLOv4 + DeepSORT 43
3.4 Diagram detekcie objektov a spracovania dát 46
3.5 Diagram spracovania dát 53
4.1 Grafické užívateľské prostredie aplikácie SegMentor 57
4.2 Diagram GUI aplikácie SegMentor 58
4.3 Výstup prahovacej funkcie 60
4.4 Výstup segmentovania watershed 62
4.5 Výstup segmentačnej funkcie na báze detekcie hrán 64
4.6 Výstup segmentačnej funkcie na báze HMM 65
4.7 Výstup segmentačnej funkcie na báze K-means clustering 67
4.8 Vývojový diagram funkcie Segmentor() 76

Zoznam tabuliek

2.1 Namerané hodnoty pre jednotlivé testovacie videá 35
4.1 Namerané hodnoty pre jednotlivé implementované funkcie 77

Zoznam výpisov

2.1 Metóda filtering() . . . 34
2.2 Metóda calculate_pitch() . . . 34
3.1 Štýl PyQT aplikácie ReAmper . . . 39
3.2 Funkcia CalculatePath() . . . 44
3.3 Funkcia CalculateSize() . . . 44
3.4 Funkcia CreateObjectMap() . . . 45
3.5 Časť súboru scale.py . . . 47
3.6 Funkcia path_interpolation() . . . 49
3.7 Funkcia size_interpolation() . . . 49
3.8 Funkcia getHRTFdata() . . . 52
4.1 Funkcia thresholding() . . . 60
4.2 Funkcia watershed_segmentation() . . . 62
4.3 Funkcia edge_based_segmentation() . . . 63
4.4 Funkcia HMM() . . . 65
4.5 Funkcia kmeans() . . . 67
4.6 Funkcia kmeans() . . . 68
4.7 Funkcia sinc_interpolation() . . . 73
4.8 Funkcia set_min_distance() . . . 73
4.9 Funkcia the_sum() . . . 75

Úvod

Video je neoddeliteľnou súčasťou moderných audiovizuálnych umeleckých diel a inštalácií. Téma prepojenia obrazu a zvuku je preberaná azda už od doby, keď tieto média vznikli. Od tej doby boli vytvorené rôzne aplikácie ponúkajúce prepojenie obrazu a zvuku, každá fungujúca na rôznych princípoch a metódach. V tejto diplomovej práci sa preto aj my venujeme téme syntézy zvuku z dát videa.

V prvej časti je vypracovaná rešerš, ktorá poskytuje prehľad existujúcich prístupov a aplikácií zaoberajúcich sa syntézou zvuku z obrazových a video dát. V rámci tejto rešerše sme analyzovali články, práce a programy, ktoré sa zaoberajú mapovaním obrazových dát na zvukové. Tieto aplikácie fungujú na rôznych princípoch syntézy zvuku z obrazu a poskytujú zaujímavé možnosti pre tvorcov audiovizuálnych diel.

V práci sú implementované tri nami vytvorené metódy syntézy zvuku z video dát. Prvou z týchto implementácií je aplikácia *VSyntha*, ktorá je založená na mapovaní dát farebného priestoru na parametre oscilátorov. Druhá z týchto metód je implementovaná v aplikácií *ReAmper*, ktorá vykonáva ozvučovanie scén vďaka detekcii a sledovaní objektov scény, priraďovaní a spracovávaní zvukov zo zvukovej knižnice. Treťou implementovanou metódou je aplikácia *SegMentor*, ktorá vykonáva rôzne druhy segmentácie obrazu, ktoré následne využíva pri mapovaní hudobných parametrov MIDI protokolu. Výstupmi týchto aplikácií sú zvuková syntéza v reálnom čase, video súbor so syntetizovaným zvukom a súbor MIDI.

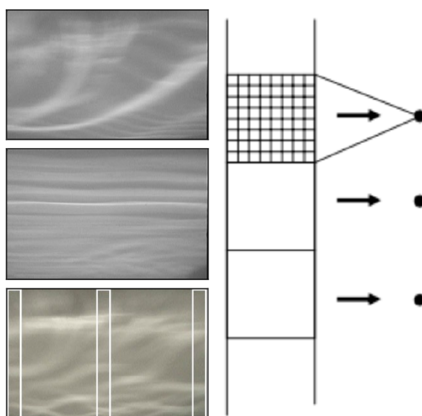
1 Rešerš metód syntézy zvuku z obrazových alebo video dat

V tejto kapitole sme vytvorili rešerš z dostupných zdrojov zaoberajúcich sa podobnou tématikou. Popísané sú rôzne články, práce a programy, ktorých súčasťou je premena obrazových dát na dáta zvukové.

1.1 Syntéza zvuku z obrázkov videa v reálnom čase

Článok *Sound Synthesis from Real-Time Video Images*[1] hovorí o technike syntézy zvuku využívajúcej obrázky videa na kontrolu zvukového spektra. Svetelná intenzita moduluje amplitúdy tridsiatich dvoch harmonických zložiek v syntetizovaných zvukoch. Článok pojednáva o problémoch mapovania dát videa na dáta zvukové, variáciách úrovní svetla, malých snímkových frekvenciách videa v porovnaní s vysokým vzorkovacím kmitočtom audia a celkovej systémovej implementácií.

Využíva sa technika, kde je intenzita videa použitá ako zdroj terénu pre syntézu, konkrétne sú používané videá vodnej hladiny. Autor prirovnáva techniku k pohybu fotorezistoru v kruhoch po obrázku videa, pričom ak je tento pohyb dostatočne rýchly, je generovaná kvaziperiodická vlna. Pri zmene obrázka nastáva aj zmena generovaného zvuku. Autor pri tejto metóde však narazil na problémy s výsledným zvukom, keď pohyb vody nevytváral dostatočný pohyb pre generovaný zvuk. Metódu preto modifikoval tak, že obraz rozdelil na obdĺžniky o veľkosti 8×256 pixelov (pri použití vstupného obrázka o veľkosti 256×256 pixelov). Každý z týchto obdĺžnikov je následne rozdelený na bloky s rozmerom 8×8 pixelov, ktorých hodnoty sú ďalej spriemerované a vytvárajú tak tridsaťdva hodnôt, ktoré reprezentujú amplitúdy tridsiatich dvoch harmonických zložiek.[1]



Obr. 1.1: Proces získania hodnôt amplitúd spektrálnych zložiek [1]

1.2 Filtrácia šumu videa na audio s detekciou pohybu

Článok *Filtering Video Noise as Audio with Motion Detection to Form a Musical Instrument [2]* hovorí o metóde tvorby harmonických zvukov s použitím video signálu ako vstupu. Navrhnutý je hudobný nástroj, ktorý využíva video na zvukovú syntézu a taktiež pre ovládacie prostredie na výber hudobných tónov. Nástroj je vytvorený z 3 častí:

- Syntetizátor zvuku
- Ovládacie prostredie
- Vizualizácia výstupu

Audio je vytvorené zo snímok videa, ovládanie funguje na báze detekcie pohybu a vizualizácia je vložená nad vstupné video pre zvýraznenie ovládania pohybom.

Metóda syntézy zvuku z videa spočíva v troch krokoch. V prvom kroku je každá snímka videa konvertovaná na monochromatický obraz pomocou výpočtu priemernej hodnoty troch kanálov RGB (farebný priestor Red - Green - Blue) pre každý pixel. Ďalej sú tieto hodnoty konvertované do audio bufferu so škálovaním na hodnoty v intervale **<-1;1>**. V poslednom kroku je buffer poprehadzovaný z dôvodu vytvorenia zlomu súvislých plôch snímok videa. Z tohto procesu je pre farebnú snímku produkované ticho, a pre obrazový šum je generovaný biely šum. Pre formovanie harmonického signálu autor použil veľké množstvo band-pass filtrov s úzkou šírkou, kde jednotlivé pásma odpovedajú hudobným tónom. Tieto band-pass filtre sú napojené na ASR (Obálka typu Attack-Sustain-Release) obálku, ktorá manipuluje zosilnením filtra a akosťou filtra.

Ovládanie je realizované pomocou komparácie aktuálnej snímky s predchádzajúcou, kde dochádza k výpočtu rozdielu snímok. Ak je takýchto zmenených pixelov, detegovaných na určitej ploche viac, je determinovaná pozícia plochy na snímke. Táto pozícia je následne použitá pri určení tónu .

Pri implementácií tejto metódy autor využil *HTML5, JavaScript* a API (Application programming interface - rozhranie pre programovanie aplikácií) *Web Audio, Media Capture* a *Stream*. Výstupom je webová aplikácia - hudobný nástroj dostupný na adrese `https://carlthome.github.io/motion-synth/` Výstup je podľa autora zvukovo zaujímavý, no náročný na ovládanie.[2]

1.3 Vytváranie zvukov z obrazových dát

Diplomová práca *Vytváření zvuků z obrazových dat* [3] sa venuje téme vytvárania zvuku z obrazových dát. Autor v práci uvažuje o vzťažnosti medzi obrazovými a zvukovými dátami a vytvára metódy pre tvorbu zvuku a zmenu jeho parametrov. Výsledkom sú dve aplikácie vytvorené v prostredí MATLAB a jedna aplikácia vytvorená v jazyku C++ s využitím softvérového rámca JUCE.

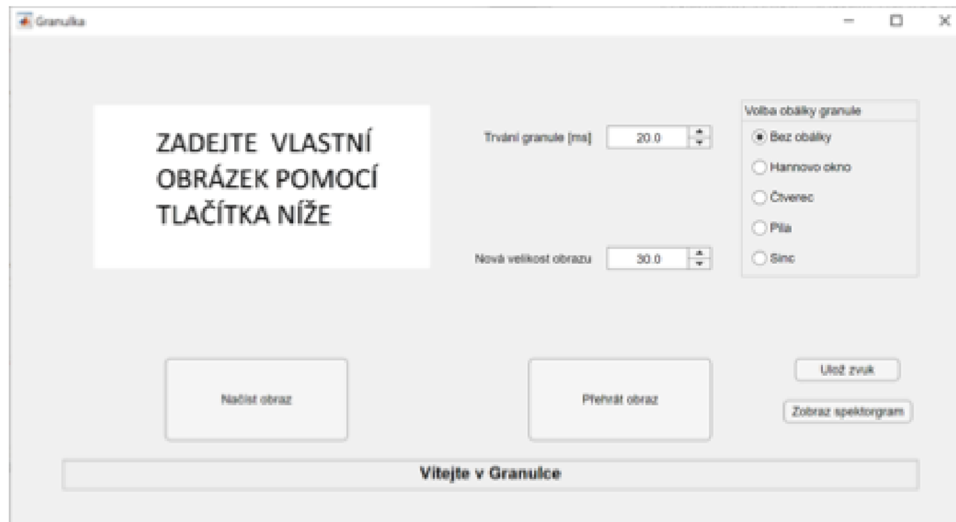
Prvá metóda nazvaná *Bzučák* je postavená na rozdelení obrazu na riadky. Vyššie riadky obrazu reprezentujú vyššie frekvencie harmonickej rady, kde počet harmonických zložiek je možné meniť ručne. Amplitúdy jednotlivých harmonických zložiek sú závislé na jasovej zložke jednotlivých riadkov, pričom fundament[1] je odvodený z celkového jasu obrazu. Táto metóda generuje jeden zvuk s nastaviteľnými harmonickými zložkami a fixnou výškou základného tónu. Výhodou algoritmu je jednoduchosť a nízka výpočtová náročnosť.

Obr. 1.2: Grafické užívateľské prostredie aplikácie *Bzučák* .[3]

Druhá metóda nazvaná *Granulka* spočíva v pohľade na pixely obrazu ako na granuly zvuku. Pixely sú čítané od stredu obrázku smerom k okrajom. Pri generovaní zvuku používa dáta z troch obrazových rovín - RGB, HSV a YCbCr. Každá zložka RGB reprezentuje frekvenciu podľa analógie vlnovej dĺžky svetelnej vlny a zvukovej vlny. Dáta z roviny HSV sú použité hlavne pre stredné a nižšie frekvencie pomocou hodnoty saturácie **S**. Rovina YCbCr je použitá pre zložku **Y**, ktorá reprezentuje amplitúdu jednotlivých harmonických zložiek.

[1] Základná frekvencia signálu (tón)

Aplikácia funguje ako granulárny syntetizátor, ktorý tvorí zvuk z granulí získaných z hodnôt pixelov obrázka. Aplikácia narazila na výpočtovú náročnosť a pri veľkosti obrazu 100x100 pixelov už bolo potrebné čakať na výsledok aj niekoľko sekúnd. Nevýhodou tejto metódy je takmer nulová kontrola nad výsledkom.



Obr. 1.3: Grafické užívateľské prostredie aplikácie *Granulka*[3]

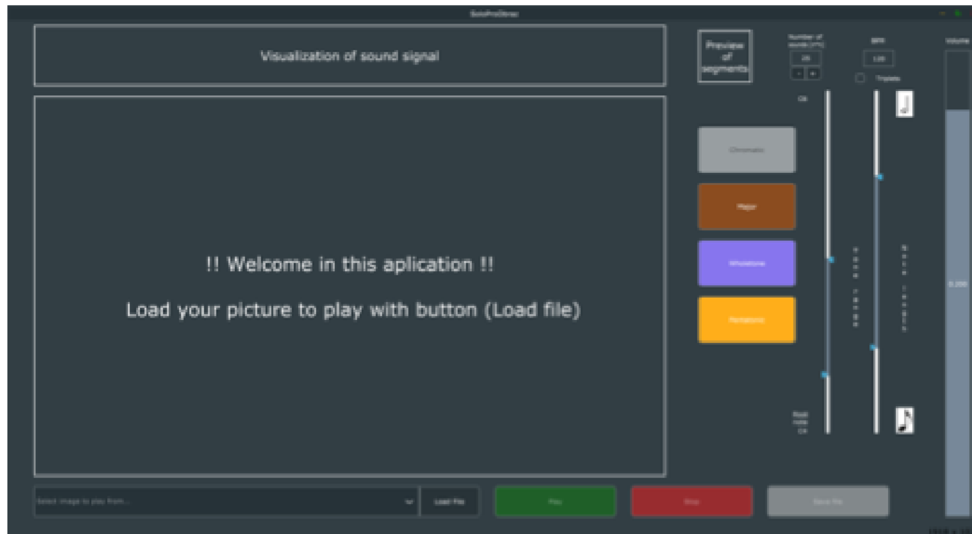
Poslednou metódou vytvorenou v tejto práci je metóda s názvom *Sólo pro obraz*, ktorá je postavená na myšlienke vytvoriť signály, ktoré vychádzajú z hudobnej teórie s možnosťou ich následného využitia v rámci klasickej hudobnej kompozície. Aplikácia je postavená v jazyku C++ s využitím softvérového rámca JUCE a knižnice OpenCV pre spracovanie obrazu. Jednotlivé tóny sú generované podľa vlastností pixelov obrázka, pričom sa v tejto aplikácií dá nastaviť:

- Načítanie obrázka
- Dĺžka generovanej skladby
- Použitý tónový rozsah skladby v jednotlivých stupniciach
- Tempo skladby
- Dĺžka nôt

Podobne ako pri predchádzajúcej metóde, táto metóda využíva tri farebné priestory HSL, HSV a YCbCr. Využíva jasovú zložku **Y** pre výšku tónu, zložku saturácie **S** pre nastavenie počtu harmonických zložiek, zložku **V** pre dĺžku jednotlivých tónov, zložku **L** pre výpočet pomeru medzi harmonickými zložkami.

Aplikácia je založená na syntéze zvuku z dát obrázka s použitím hudobnej teórie pre kompozíciu. Pri používaní sa dajú nastaviť použité stupnice a tóniny. Autor

popisuje výsledný zvuk ako tóny hudobnej kompozície zmäteného autora, pretože sa výsledný zvuk nedrží žiadnych konvencií pri tvorbe skladby.[3]



Obr. 1.4: Grafické užívateľské prostredie aplikácie *Sólo pro obraz* [3]

1.4 Photosounder

Aplikácia *Photosounder* je založená na koncepte, že zvukové spektrum v čase môže byť interpretované ako obrázok, a tento proces môže byť inverzný - obrázok môže byť interpretovaný ako zvukové spektrum v čase. Po načítaní zvuku je vytvorený spektrogram[2], ktorý tvorí obrazovú reprezentáciu zvuku. Rovnaký princíp je použitý pri načítaní obrázka - jas každého pixelu definuje amplitúdu a poloha frekvenciu v čase.

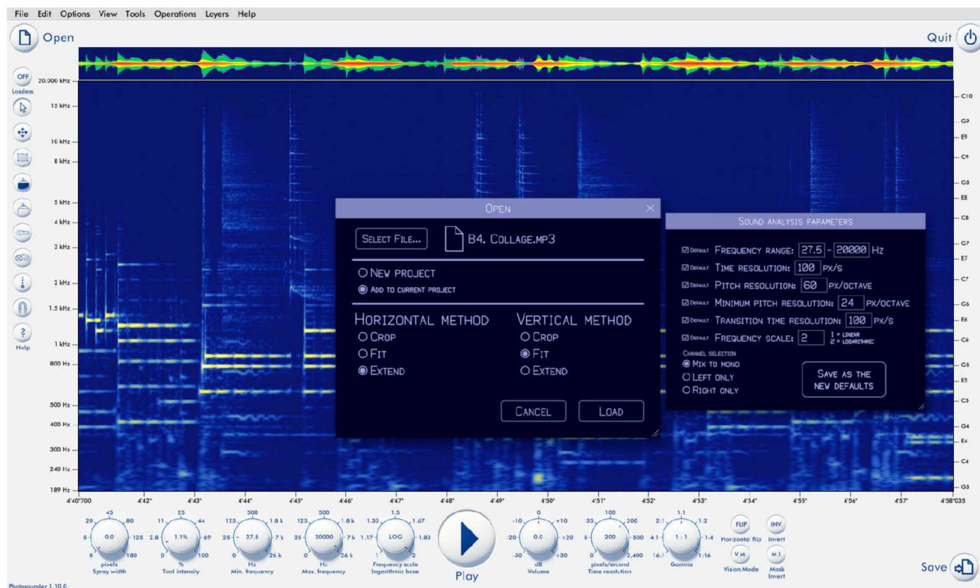
Aplikácia ponúka viacero módov syntézy:

- **Noise synthesis** - obraz syntetizovaný touto metódou je rozdelený na riadky pixelov. Každá línia popisuje, ako sa vyvíja hlasitosť v čase v závislosti od jasu jednotlivých pixelov. Táto hlasitosť je mapovaná pre šumové pásma, ktorých stredná frekvencia je determinovaná výškou riadku v obrázku. Biely obraz by touto metódou vytvoril ružový šum.
- **Lossless mode** - funguje na podobnom princípe ako *Noise synthesis*, ale namiesto ružového šumu používa originálny zvukový súbor. Vhodné použitie na zosvetľovanie obrázka, ekvalizáciu zvuku, zmenu dynamiky, izolovanie zvukov.

[2]vizuálna reprezentácia frekvenčného obsahu zvukového signálu v čase

- **Live Synthesis** - Funguje ako spomínané módy, s tým rozdielom, že je zvuk syntetizovaný v reálnom čase a užívateľ má možnosť ovládať prehrávanie svojimi zásahmi.

Aplikácia ponúka rôzne nástroje na manipuláciu s obrazovými a zvukovými dátami ako *Move Tool, Spray Tool, Rectangle Tool, Harmonics Modifier, Magnet Modifier, Zoom* a kontrolu nad parametrami ako *Min. Frequency, Max. Frequency, Frequency Scale, Volume, Tume resolution, Spray Width, Tool Intensity, Horisontal flip, Invert, Mask Inver, Vision Mode, Gamma, Layer Intensity*. Demo verzia tejto aplikácie je dostupná na adrese `https://photosounder.com/download.php`. [8]



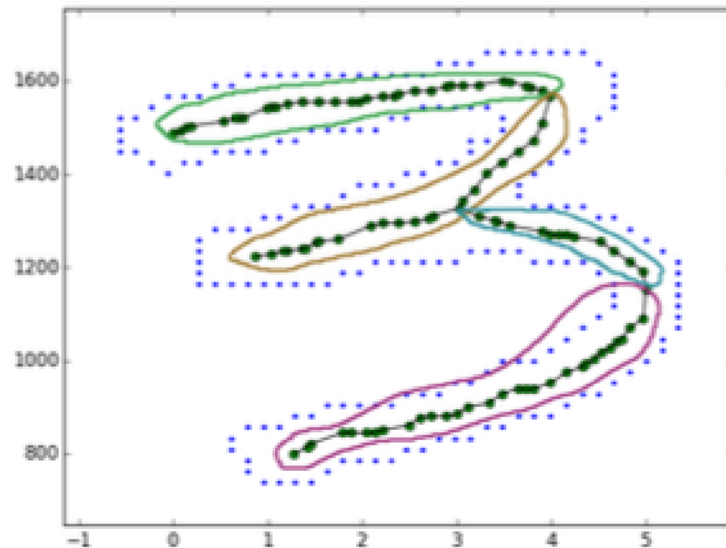
Obr. 1.5: Grafické užívateľské prostredie aplikácie *Photosounder*[8]

1.5 Syntéza zvuku zo získaného tvaru na obrázku

Článok *Pyc2Sound: a Python tool to convert images into sound* pojednáva o metóde transformácie obrázku na zvuk, s účelom vytvoriť nástroj pre umelecké alebo edukačné účely. Prezentovaná metóda nepoužíva obrázok na zvukovú syntézu priamo, ale extrahuje z neho kostru tvarov, ktoré obrázok obsahuje pomocou Voroného diagramu. Aplikácia obsahuje grafické užívateľské prostredie, v ktorom môže užívateľ nakresliť rôzne tvary, ktoré vytvárajú základ pre zvukovú syntézu. Metóda vychádza z predpokladu, že tvary objektov sú vnímané ako prvé, zľava doprava a to, že výšku obrázku korelujeme k výške tónu. Obrázky tak tvoria časovo-frekvenčnú

reprezentáciu zvuku, kde je na jednej dimenzií obrázka reprezentovaný čas a na druhej dimenzií frekvencia. Metóda využíva separáciu tvaru na lineárne stopy, ktoré sú následne podkladom pre generovanie zvuku pomocou aditívnej syntézy.

Autor v závere spomína možné modifikácie metódy pre obohatenie generovaného zvuku. Spomenuté je ovládanie amplitúdy jednotlivých stôp za pomoci získavania hrúbky tvaru, ovládanie sily harmonických zložiek pomocou farieb tvaru a spomína aj použitie spektra obrázka ako základ pre syntézu. [4]



Obr. 1.6: Princíp separácie stôp z tvaru aplikácie *Pyc2Sound* [4]

1.6 Aplikácia obrazových sonifikáčných metód na hudbu

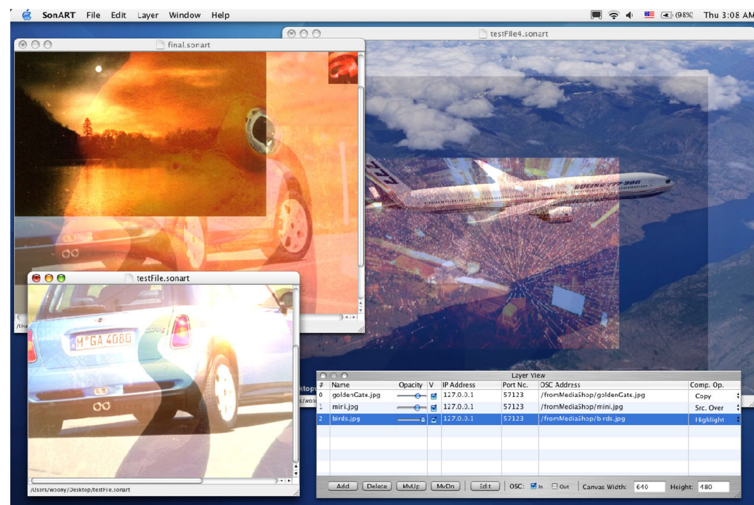
V článku *Application of Image Sonification Methods to Music* autor prezentuje dva koncepty mapovania statických dát na dáta časovej domény. Autor v článku popisuje 5 metód sonifikačného mapovania:

- **Mapovanie pomocou inverzného spektogramu** - proces skenovania[3] riadkov obrázka pozdĺž jednej z ôs. Tento proces vytvorí 3D reprezentáciu farby alebo jasu obrázka, ktorý sa dá použiť ako spektrogram zvuku.
- **Mapovanie pomocou inverzného rastrového skenovania** - pri rastrovom skenovaní sú dvojdimenzionálne dáta obrazu reprezentované ako sériový tok dát. Tieto dáta môžu byť použité pri sonifikácií ako melodická sekvencia.

[3]Prechádzanie *pixel po pixely* v riadkoch alebo stĺpcoch, po lineárnych cestách

- **Mapovanie pomocou virtuálnych ciest pomocou farebnej hĺbky** - Horizontálna a vertikálna pozícia každého pixelu je namapovaná na priestorový parameter a výšku, jasová alebo farebná zložka pixelu rozhoduje o dĺžke zvuku.
- **Mapovanie pomocou virtuálnych ciest pomocou imaginárnej osi** - Technika využíva podobný princíp ako technika pomocou farebnej hĺbky, no používa cestu *do* a *z* obrázka a pohybuje sa tak po imaginárnej osi.
- **Mapovanie pomocou pointerov na arbitrárnych cestách** - Kombinácia probingu[4] a lokálneho skenovania. Pre sonifikáciu má užívateľ možnosť vytvárať cesty pre skenovanie alebo ukladať body pre probing.

Autor ďalej v článku popisuje možné aplikácie týchto metód v hudobnej praxi - napríklad využitie skenovania ako notový zápis alebo improvizácie na základe probingu. Kombinácie metód probingu a skenovania autor označuje za nový model pre hudobné aplikácie a spomína SonART[5] ako ideálne prostredie na implementáciu.[5]



Obr. 1.7: Užívateľské prostredie aplikácie *SonART* [6]

1.7 Prostredie SONIFYD

V článku *SONIFYD: A GRAPHICAL APPROACH FOR SOUND SYNTHESIS AND SYNESTHETIC VISUAL EXPRESSION* autor popisuje *Sonifyd* - prostredie pre sonifikáciu multimédií na báze konverzie farby na zvuk s manipuláciou v reálnom čase. Sonifyd skenuje obraz horizontálne alebo vertikálne, generuje zvuk

[4] Skenovanie pixelov v okolí bodu

[5] Prostredie pre sonifikáciu dát, ich vizualizáciu a tvorbu sieťových multimediálnych aplikácií

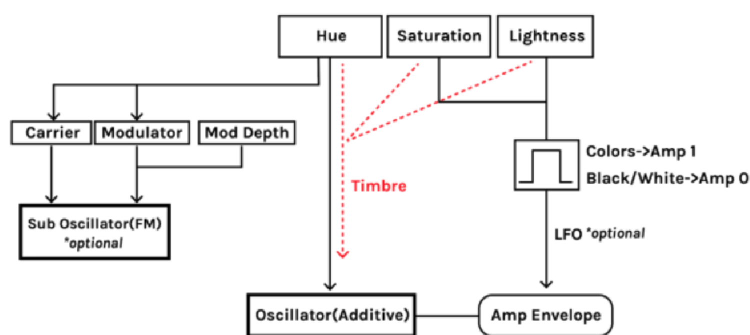
[6] Obrázok prevzatý z: `https://ccrma.stanford.edu/~woony/software/sonart/screenshot.01.jpg`

a ovplyvňuje farbu na báze aditívnej syntézy. Primárnym cieľom tohto projektu je vytvoriť funkčný nástroj pre nový druh vizuálneho umenia, vývoj sonifikácie, rôzne umelecké inštalácie a audiovizuálne vystúpenia. Autor pre implementáciu aplikácie využíva:

- **Processing** - Grafická knižnica s integrovaným vývojovým prostredím, určená pre ľudí, ktorí nemajú skúsenosti s programovaním vďaka svojmu vizuálnemu prístupu
- **Syphon** - Open source technológia pre MacOS umožňujúca aplikáciám zdieľanie snímok v reálnom čase.
- **MaxMSP** - Vizuálny programovací jazyk pre hudobné a multimediálne aplikácie.
- **Lemur** - Aplikácia pre kontrolu MIDI[7] a OSC[8] za pomoci grafického užívateľského prostredia

Aplikácia využíva metódu sonifikácie farby na zvuk, kde využíva farebný priestor HSL pre interpretáciu farebných hodnôt. Využíva hodnotu **H** na kontrolu frekvencie a hodnoty **S** a **L** na kontrolu pitch-shifting[9]. Mapovanie hodnoty **H** na frekvenciu prebieha v rámci jednej oktávy a vytvára tak mikrotonálne spektrum. Autor využíva hodnoty bielej a čiernej farby na generovanie pulzu, ktorý moduluje amplitúdovú obálku oscilátora. Ponúka aj možnosť vytvorenia *"sub-oscilátora"* ktorý je založený na syntéze pomocou frekvenčnej modulácie.

Autor hodnotí *Sonifyd* ako prostredie pre improvizovaný a experimentálny zvukový dizajn, a ako dobrý základ s vysokým potenciálom pre tvorbu unikátnych a imerzívnych audiovizuálnych zážitkov.[6]

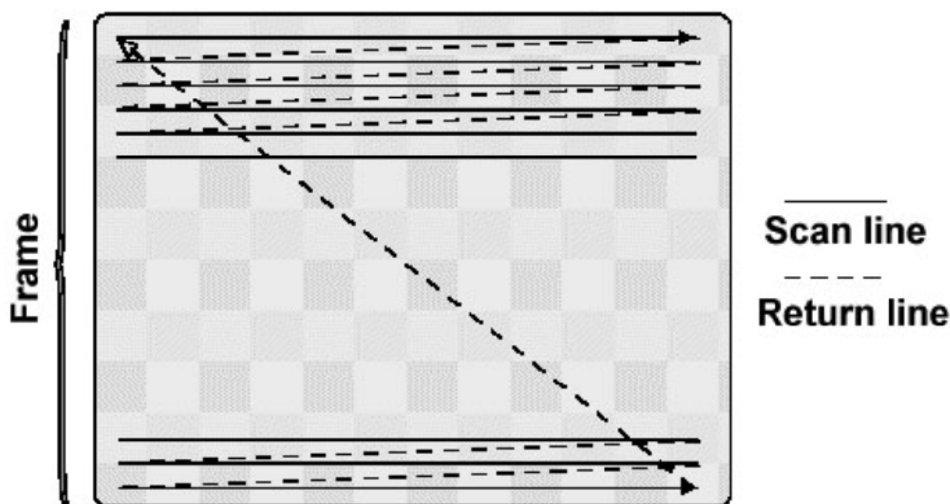


Obr. 1.8: Princíp aplikácie *Sonifyd* [6]

[7]Protokol Musical Instrument Digital Interface
[8]Protokol Open Sound Control
[9]Zmena výšky tónu

1.8 Sonifikácia pomocou rastrového skenovania

Článok *Raster Scanning: A New Approach to Image Sonification* rieši problematiku obrazovej sonifikácie a syntézy pomocou rastrového skenovania. Autor popisuje metódu pre sonifikáciu obrazu a vizualizáciu zvuku. Rastrové skenovanie je technika, pri ktorej sa prechádzajú pixely obrázka riadok po riadku, zľava doprava a zhora dole.



Obr. 1.9: Princíp rastrového skenovania [7]

Metóda používa mapovanie hodnôt jasu pixelov šedotónových obrázkov, ktoré sú normalizované na interval **<-1;1>** aby mohli reprezentovať vzorky audia - jeden pixel tak zodpovedá jednej vzorke.

Autor sa v časti článku venuje aj zvukovej vizualizácií pomocou rastrogramu. Rastrogram definuje ako inverzný proces k rastrovej sonifikácií a proces tak vytvorí zo zvuku obrázok.

V článku je predstavená metóda rastrovej sonifikácie a zvukovej vizualizácie. Autor hodnotí rastrovú sonifikáciu ako silnú metódu pre vytváranie zvuku s pocitom originálneho obrázka. Rastrogram hodnotí ako potenciálny nástroj vizuálneho prostredia pre audio. [7]

1.9 Iné práce

Medzi ďalšie práce a aplikácie, ktoré sa venujú podobnej tématike, môžeme spomenúť:

- **Variofon** - Fotoelektronický hudobný nástroj vyvinutý v roku 1932, ktorého autormi sú Yevgeny Alexandrovitch Sholpo a Georgy Rimsky-Korsakov. Metóda využívana týmto nástrojom spočívala v optickom nahrávaní zvuku, čo bolo dosiahnuté strihaním zvukových vĺn na kartónové disky, ktoré rotovali synchrónne s 35mm filmovým pásom. Tento proces bol natočený a následne prehraný na normálnom filmovom projektore a reproduktore. [9]
- **ANS synthesizer** - Je to fotoelektronický hudobný nástroj z roku 1957, ktorého autor je Evgeny Murzin. Funguje na báze kreslenia zvukovej vlny, pričom na zápis využíva rotujúci sklenený disk, na ktorý je možné zapísať 144 optických fonogramov.[10]
- **Oramics** - Technika kreslenia zvuku vyvinutá v roku 1957, ktorej autor je Daphne Oram. Technika je založená na kreslení tvarov na 35mm film, vytvárajúc masku, ktorá moduluje svetlo dopadajúce na fotočlánky a ovládajú tak výšku, hlasitosť a farbu tónu. [11]
- **Metasynth** - Nástroj pre zvukový dizajn a hudobnú produkciu, ktorý využíva vytváranie zvuku z obrazu, vyvíjaný spoločnosťou *U&I Software*. Najznámejšie použitie tohto softvéru bolo pri produkcií filmu *The Matrix*.[12]
- **Audiopaint** - Aplikácia tvorí zvuk z obrázkov formátov JPEG, GIF, PNG a BMP. Z každého pixelu extrahuje jeho polohu a farebnú informáciu, ktoré mapuje na frekvenciu, amplitúdu a panoramovanie. Funguje na báze aditívnej syntézy.[13]
- **Coagula** - Aplikácia tvorí zvuk z obrázka na základe použitia sínusoidy na každom riadku obrázka, pričom farba každého pixelu definuje panorámu a množstvo úzkopásmového šumu.[14]
- **Kaleidoscope** - Nástroj pre zvukový dizajn využívajúci konverziu obrazu na zvuk. Využíva skenovanie obrázka zľava doprava, pričom horizontálna rovina reprezentuje čas a vertikálna rovina reprezentuje frekvenciu, podobne ako spektogram. Jasová zložka každého pixelu determinuje hlasitosť výstupného zvuku.[15]

2 VSyntha

V tejto kapitole sa venujeme implementácií prvej metódy syntézy zvuku z video dát, a to vo forme samostatnej aplikácie, ktorá používa video ako vstupné dáta a syntetizuje z neho zvuk.

Rozhodli sme sa vytvoriť aplikáciu s názvom *VSyntha* na báze aditívnej syntézy, s využitím skenovacej metódy a mapovania obrazových dát na dáta zvukové. Implementovaná je aj filtrácia signálu pomocou využívania hodnôt histogramu šedotónového obrázka na ovládanie parametrov ekvalizačných filtrov. Základná myšlienka tejto metódy sonifikácie je čítanie videa ako hudobnej notácie - z každého videa tak dokážeme extrahovať jedinečnú hudobnú skladbu.

2.1 Prostredie

Pre tvorbu tejto aplikácie sme sa rozhodli pre využitie programovacieho jazyka **Python**, ktorého hlavnými prednosťami sú množstvo dostupných knižníc, jednoduchá a čitateľná syntax a portabilita medzi platformami. Hlavnou nevýhodou tohto jazyka môže byť rýchlosť, keďže sa Python radí medzi pomalšie jazyky, kvôli svojej univerzálnosti a tým, že patrí medzi dynamicky typované programovacie jazyky[1].

Pri výbere vývojového prostredia sme sa rozhodli pre *Visual Studio Code*[2], pre inštaláciu verzií Pythonu a knižníc sme využili distribútor *Anaconda*[3]. Aplikácia tak beží na verzii **Python 3.7**, pretože na tejto verzii sú dostupné všetky použité knižnice. Jednou z výhod využitia distribútora *Anaconda* je možnosť vytvorenia a exportovania pythonového prostredia *environment* pre ľahkú migráciu medzi systémami. Vývoj tejo aplikácie, rovnako aj ďalších aplikácií v tejto práci, prebiehal na operačnom systéme macOS Ventura 13.3.1 (22E261).

2.2 Knižnice

Pri návrhu tejto aplikácie sme vyberali z množstva dostupných knižníc pre programovací jazyk Python. Hlavným cieľom pri voľbe knižníc bolo nájsť kompatibilné knižnice pre spracovanie obrazu, generovanie a filtrovanie zvuku, ako aj knižnicu pre vytvorenie grafického užívateľského prostredia. Nakoniec sme sa rozhodli pre knižnice *PyQT5, PYO, OpenCV* a *Numpy*, na ktoré sa pozrieme v nasledujúcich podkapitolách.

[1] Jazyky, ktoré nedefinujú typ premennej v kóde, ale až v kompilácií

[2] `https://code.visualstudio.com`

[3] `https://www.anaconda.com`

2.2.1 Knižnica PyQT5

Pri výbere knižníc pre tvorbu užívateľského prostredia sme mali na výber z viacerých možností, ako napríklad *Tkinter, Kivy, wxPython, Libavg*. Nakoniec sme sa rozhodli pre **PyQT5**, ktorý je pythonovou extenziou grafického softvérového rámca **Qt**[4]. Pre grafický dizajn užívateľského prostredia je dostupná aj aplikácia *QtDesigner*, ktorú sme sa ale rozhodli pri implementácií nepoužívať. Knižnica je dostupná pre všetky unixové systémy, ako aj pre platformu MS Windows. Knižnica je dostupná pod GNU/GPL-2 a pod komerčnou licenciou.[16]

2.2.2 Knižnica PYO

Ako ďalšiu sme vybrali knižnicu pre spracovanie a generovanie zvuku, kde sme vyberali z knižníc ako *pyAudioAnalysis, Dejavu, Mingus, hYPerSonic, PYO*. Rozhodli sme pre použitie knižnice **PYO**[5], ktorá obsahuje mnoho nástrojov pre manipuláciu so zvukom od základných matematických operácií s audio signálom až po pokročilé algoritmy. Pri využívaní tejto knižnice sa lokálne spúšťa audio server, na ktorom prebieha generovanie zvuku, ktorý má možnosť generovať užívateľské prostredie pre ovládanie parametrov generovaného signálu a servera. Knižnica je dostupná pod licenciou GNU LGPL.[17]

2.2.3 Knižnica OpenCV

Spomedzi knižníc na spracovanie obrazu sme zvažovali výber z knižníc ako *Scikit-image, OpenCV, Mahotas, SimplelTK, Pillow*. Rozhodli sme pre knižnicu **OpenCV**[6], ktorá obsahuje nástroje pre spracovanie obrazu ako napríklad rozpoznávanie objektov, rozpoznávanie tvárí, segmentáciu obrazu. Knižnica je od verzie 4.5. licencovaná pod Apache 2, pričom všetky predošlé verzie sú pod licenciou BSD. [18]

2.2.4 Knižnica NumPy

Pri implementácií sme využili aj knižnicu **NumPy**[7] pre optimalizáciu výpočtov a prácu s maticami a vektormi. Táto knižnica ponúka rôzne dátové štruktúry a veľký počet matematických funkcií, ktoré uľahčujú a zásadne zrýchľujú chod programu. Táto knižnica patrí medzi open-source pod licenciou BSD.[19]

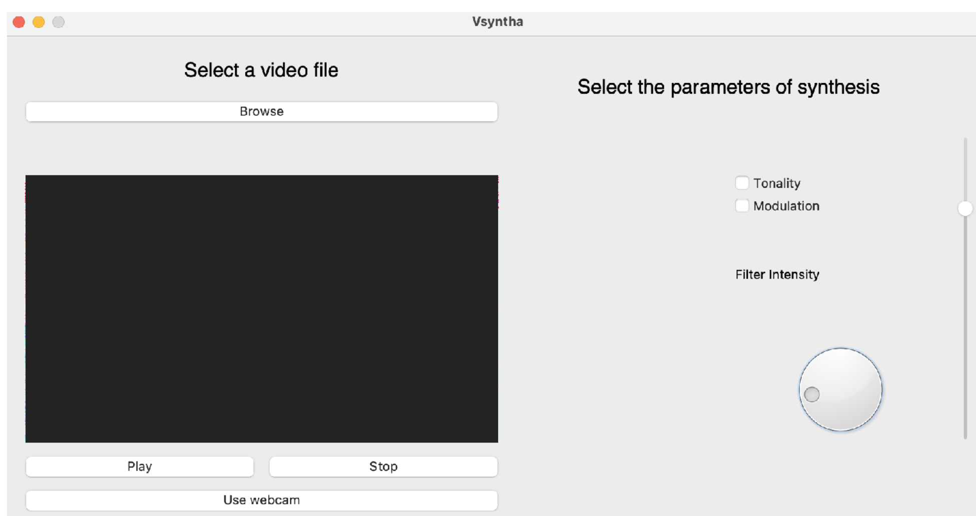
---

[4] `https://www.qt.io`
[5] `http://ajaxsoundstudio.com/software/pyo/`
[6] `https://opencv.org`
[7] `https://numpy.org`

2.3 Grafické užívateľské prostredie

Ako prvé sme vytvorili grafické užívateľské prostredie pre aplikáciu. Pri návrhu tejto aplikácie sme rozhodli, že toto prostredie by malo obsahovať prvky, ktoré by ponúkali nasledujúce možnosti:

- Načítanie videa zo súboru
- Prehrávač videa s možnosťou spustiť a zastaviť prehrávanie videa
- Použiť webkameru ako vstupné video
- Parametre ovládania syntézy
- Parametre ovládania výstupnej hlasitosti

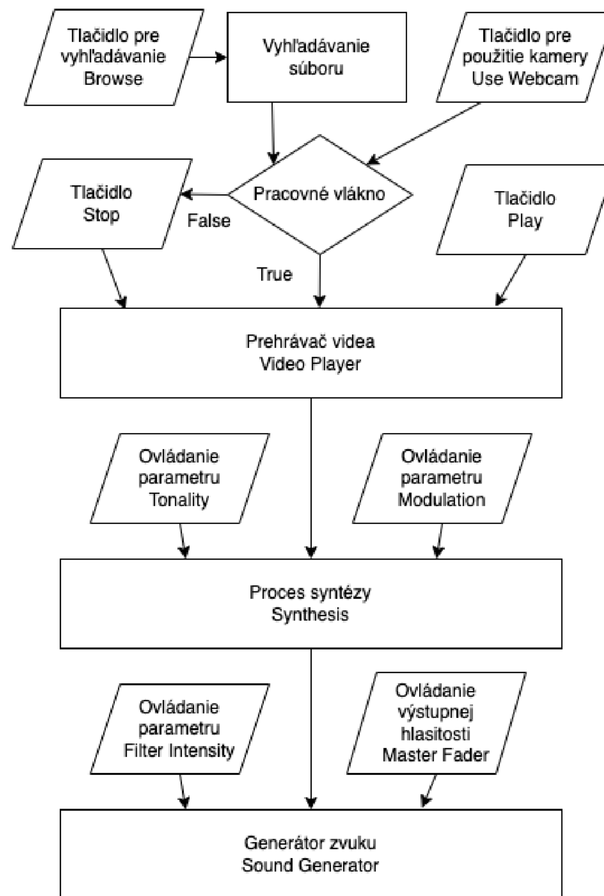


Obr. 2.1: Grafické užívateľské prostredie aplikácie *Vsyntha*

Pre načítanie videosúboru sme vytvorili tlačidlo *Browse*, ktoré po stlačení otvorí dialógové okno pre výber súboru. Po otvorení sa cesta k súboru zobrazí pod tlačidlom a rovnako sa spustí prehrávanie videa. V spodnej časti sa nachádzajú tlačidlá *Play* a *Stop*, ktoré ovládajú prehrávanie videa (popísané v časti 2.4) a tým pádom zároveň ovládajú aj generovanie hudby (popísané v časti 2.6). Na tomto mieste sa nachádza aj tlačidlo *Use webcam*, ktoré nastaví vstupné video na prúd dát z webkamery. V pravej časti sa nachádza ovládanie parametrov tejto metódy syntézy, a to ovládanie pre generovanie tonálneho (implementovaná durová stupnica) alebo atonálneho (implementovaná chromatická stupnica) zvuku, ovládanie pre využívanie modulácie[8] v priebehu generovania a ovládací potenciometer pre intenzitu spektrálnej filtrácie. Posledným objektom v užívateľskom prostredí je vertikálny posuvník, ktorý slúži na tlmenie výstupného zvuku. Vývojový diagram môžeme vidieť na obr. 2.2.

[8]Zmena tonálneho centra skladby

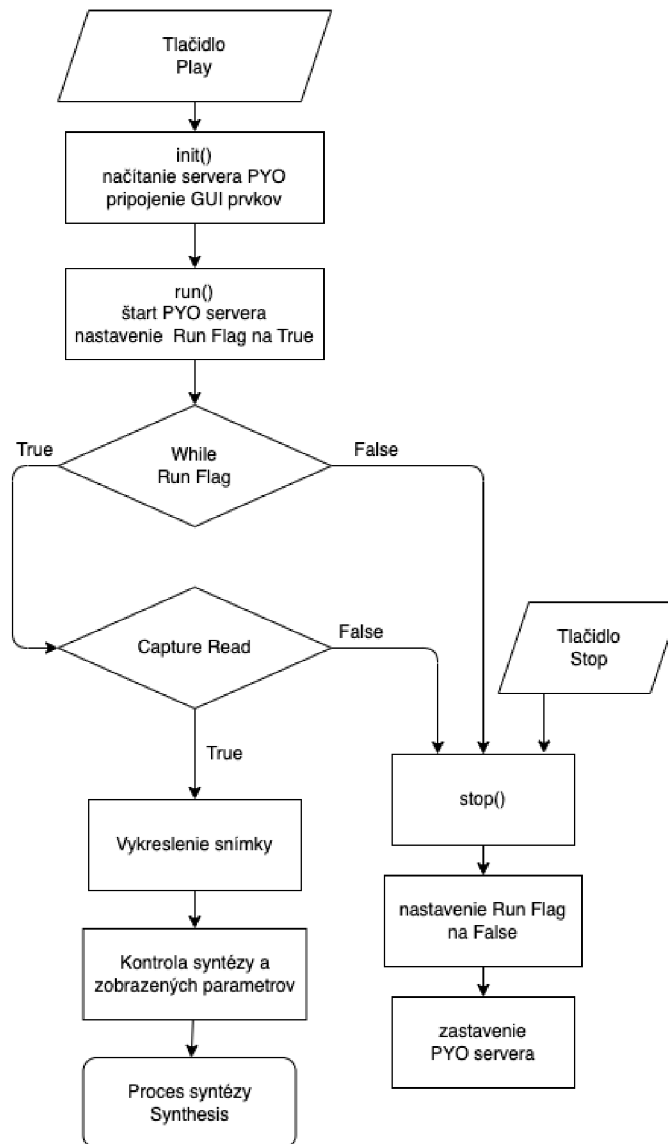
Každý zo vstupných parametrov ovláda jeden z troch hlavných blokov *Video Player*, kde prebieha vykresľovanie snímok videa, *Synthesis*, kde prebieha výpočet pre *Sound Generator*, ktorý generuje syntetizovaný zvuk pomocou aditívnej syntézy. Celý zdrojový kód tohto prostredia sa nachádza v prílohe A, v súbore `main.py`.



Obr. 2.2: Vývojový diagram grafického užívateľského prostredia aplikácie *Vsyntha*

2.4 Prehrávanie videa

Pre prehrávanie videa sme využili knižnicu *OpenCV*, pomocou ktorej získavame zo súboru (alebo z webkamery) snímku po snímke a zobrazujeme ju do štítku vytvorenom v užívateľskom prostredí. Celý tento proces beží na triede pracovného vlákna, ktorý vykonáva skript programu a zabraňuje tak zamrznutiu hlavného grafického užívateľského prostredia počas syntézy a prehrávania. Vývojový diagram môžeme vidieť na obr. 2.3. Zdrojový kód tohto bloku sa nachádza v prílohe A, v súbore `main.py`.

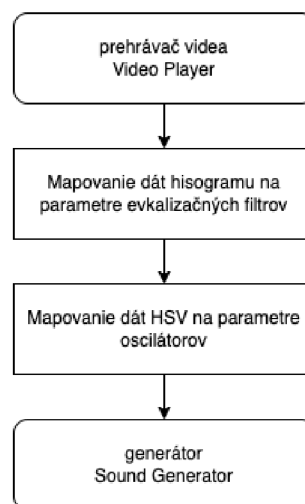


Obr. 2.3: Vývojový diagram bloku *Video Player* aplikácie *VSyntha*

Po stlačení tlačidla *Play* sa inicializuje trieda **MethodThread(QThread)**, pričom sa načíta server knižnice *PYO* a pripoja sa všetky prvky užívateľského prostredia pre získavanie alebo vypisovanie hodnôt. Zároveň sa spustí metóda triedy **run()**, ktorá spomínaný server spustí (umožní tak prehrávanie zvuku) a nastaví premennú **Run Flag**, ktorej hodnota vypovedá o stave prehrávania. V tele tejto metódy prebieha cyklus **while**, ktorý číta dáta z videosúboru snímku po snímke. Ak sa podarilo získať snímku zo súboru, je vykreslená do pripraveného štíku v grafickom užívateľskom prostredí, prebehne zisťovanie hodnôt nastaviteľných parametrov a tieto hodnoty sa spolu so snímkom pošlú do bloku pre syntézu *Synthesis*. Ak sa ale nepodarí získať ďalšiu snímku alebo užívateľ klikne na tlačidlo *Stop*, je volaná metóda triedy **stop()**, ktorá nastaví premennú **Run Flag** na FALSE a zastaví tak prehrávanie videa a zároveň vypne zvukový server. Užívateľ tak môže ľubovoľne zastavovať a prehrávať video.

2.5 Syntéza

Samotný výpočet pre našu metódu syntézy prebieha v bloku *Synthesis*, ktorý je v skripte volaný funkciou **M1Comp()**. V tomto kroku prebieha získanie dát z obrázku a ich mapovanie na parametre zvukovej syntézy. Tento proces sa skladá z dvoch podprocesov - mapovanie dát pre oscilátory, ktoré generuje zvukový signál pomocou aditívnej syntézy a mapovania dát pre ekvalizačné filtre, ktoré ovplyvňujú spektrálnu obálku výsledného zvuku. Namapované parametre sa následne posielajú do bloku *Sound Generator*, kde prebieha generovanie zvuku. Vývojový diagram tohto bloku môžeme vidieť na obr. 2.4. Tento proces je volaný na každej snímke videa.

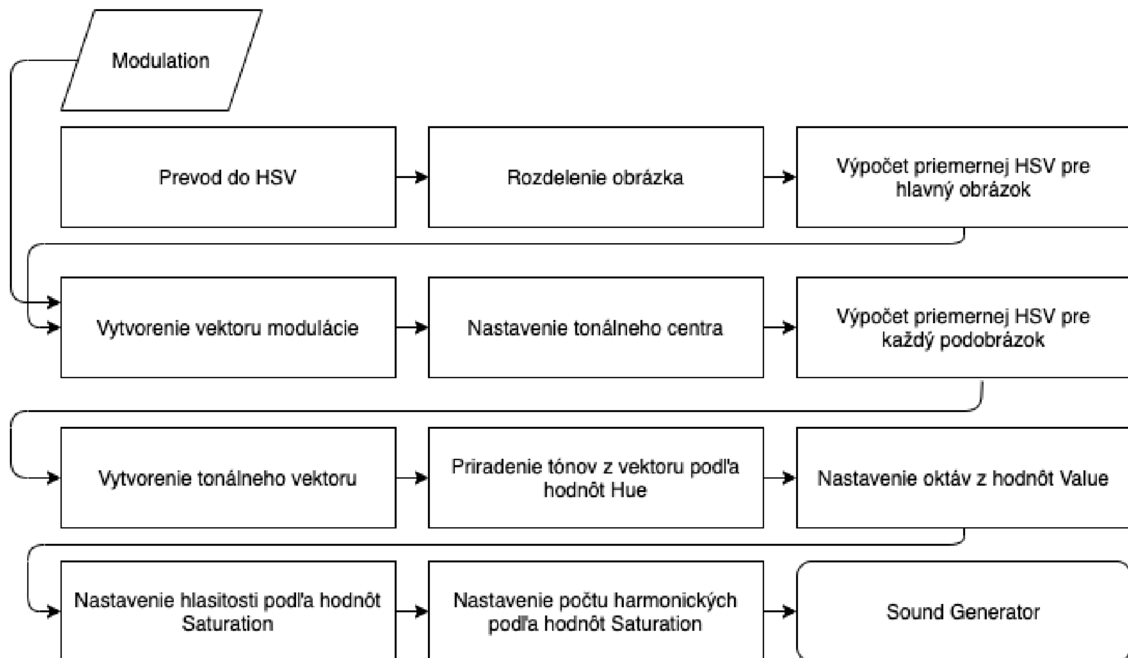


Obr. 2.4: Vývojový diagram bloku *Synthesis*

Zdrojový kód tohto bloku sa nachádza v prílohe A, v súbore `m_1.py`.

2.5.1 Mapovanie dát farebného priestoru na dáta zvukového signálu

Proces mapovania dát farebného priestoru **HSV** na parametre oscilátora je bližšie popísaný na vývojom diagrame na obr. 2.5.

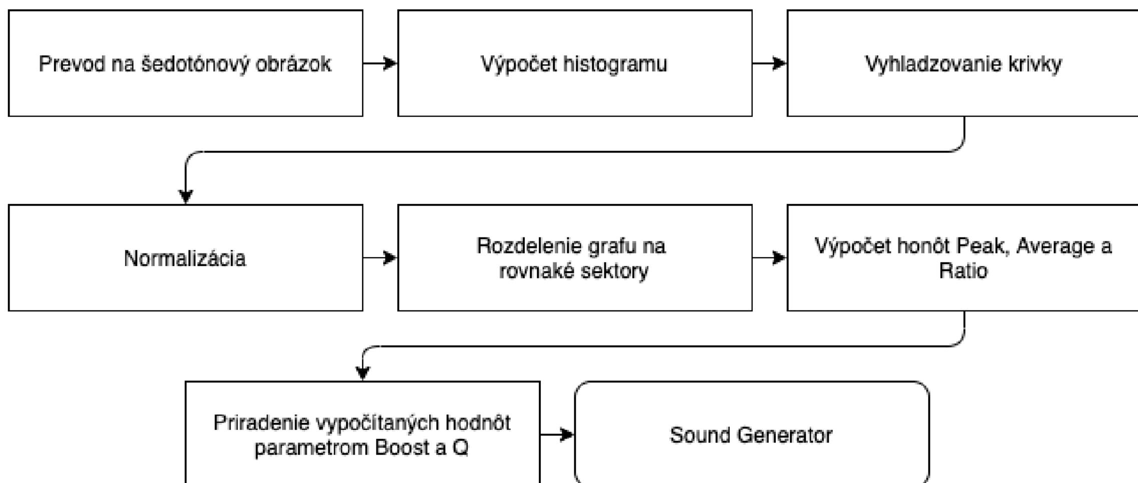


Obr. 2.5: Vývojový diagram mapovania obrazových dát na zvukové

Tento proces začína prevodom farebného priestoru snímky na priestor **HSV**, následne sa táto snímka rozdelí na päť podobrázkov. Toto rozdelenie je na báze vytvorenia stĺpcov z obrázka - šírka obrázku je rozdelená na päť častí, výška ostáva rovnaká. Podľa vstupnej možnosti *Modulation* sa vytvorí modulačný vektor, ktorý sa používa na hudobnú moduláciu zvuku podľa **H** pôvodnej snímky. Tento vektor je buď naplnený tónmi oktávy, alebo obsahuje iba jeden prvok a generovaná hudba má tonálny základ v tóne *a (440Hz)*. Následne sa na každom podobrázku vyrátajú priemerné hodnoty **HSV**, ktoré sa využívajú na mapovanie. Vytvorí sa tonálny vektor podľa vstupnej možnosti *Tonality* a generujú sa tak koeficienty buď chromatickej alebo durovej stupnice. Podľa hodnôt **H** každého podobrázka a tonálneho vektora sa mapujú hodnoty výšky generovaných tónov jednotlivých oscilátorov. Pomocou hodnôt **V** sa mapuje výška oktávy pre daný tón. Z hodnôt **S** sa mapuje amplitúda generovaného signálu a počet generovaných harmonických zložiek. Všetky tieto namapované parametre sa odosielajú do generátora zvuku *Sound Generator*.

2.5.2 Mapovanie dát histogramu na dáta ekvalizačných filtrov

Proces mapovania dát získaných z histogramu šedotónovej snímky na parametre ekvalizačných filtrov je popísaný na vývojovom diagrame na obr. 2.6. V prvom rade sa snímka zmení na šedotónový obrázok, z ktorého sa pomocou funkcie z knižnice *OpenCV* **calcHist()** vyrátajú hodnoty histogramu. Následne sa pomocou *Savitzky–Golayovho* filtru[9] krivka vyhladzuje, aby pripomínala reálnu ekvalizačnú krivku. Hodnoty tejto krivky sú následne normalizované medzi hodnoty na intervale **<-1;1>** a krivka je lineárne rozdelená na päť sekcií. V každej z týchto sekcií je následe vyrátaná maximálna hodnota *Peak*, priemerná hodnota *Average* a pomer medzi hodnotami *Peak/Average*. Maximálna hodnota *Peak* sa mapuje na parameter ovládajúci zosilnenie *boost* ekvalizačného filtra a hodnota pomeru sa využíva pri mapovaní hodnoty Q ekvalizačného filtra. Tieto hodnoty sa následne odošlú do zvukového generátora *Sound Generator*, kde sa nimi ovláda päť ekvalizačných filtrov.

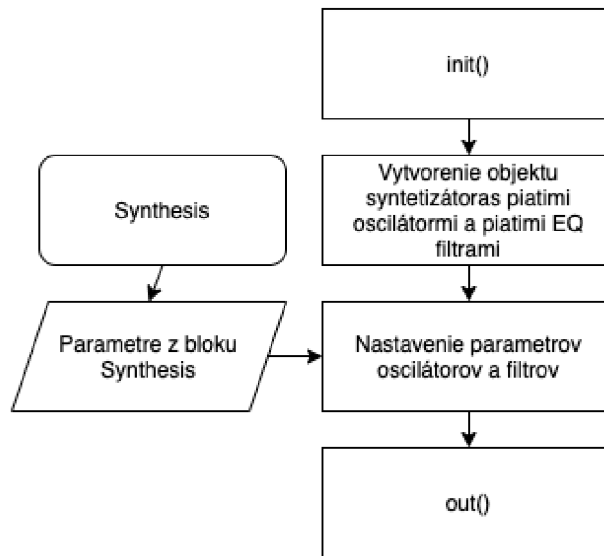


Obr. 2.6: Vývojový diagram bloku mapovania dát histogramu

[9]Filter sme použili z knižnice SciPy: `https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.savgol_filter.html`

2.6 Generátor zvuku

Pre generovanie zvuku sme využili knižnicu *PYO* a jej signálový generátor, pomocou ktorého generujeme signály typu **Blit** (generuje harmonickú radu s variabilným počtom vyšších harmonických zložiek) na piatich oscilátoroch, na ktorých meníme frekvecie a počet generovaných harmonických zložiek. Výsledný zvuk je filtrovaný piatimi ekvalizačnými filtrami, na ktorých sa mení ich zosilnenie a parameter Q. Celý proces generovania zvuku a zmeny parametrov prebieha v bloku *Sound Generator*, ktorého vývojový diagram môžeme vidieť na obr. 2.7.



Obr. 2.7: Vývojový diagram bloku *Syntéza*

Pri inicializácii syntézy sa v triede **SynthPlayer()** vytvorí päť oscilátorov a päť ekvalizačných filtrov. Všetky vytvorené objekty sú nastavené pre chod tohto zvukového prehrávača, o ktorý sa starajú metódy triedy **out()** a **sig()**. Táto trieda obsahuje aj metódy na zmenu parametrov jednotlivých objektov.

Metóda **filtering()**, zobrazená na výpise kódu 2.1, má na vstupe parametre získané v bloku *Synthesis* a mení nastavenie ekvalizačných filtrov podľa týchto parametrov. Zosilnenie filtrov sa v tomto bode násobí s hodnotou prvku užívateľského prostredia *Filter intensity*, a získavame tak možné zosilnenie filtra na intervale **<-6;6>** [dB].

```
...
    def filtering(self, boost, Q, intensity):
        for x in range(len(boost)):
            exec(f'self.eq{x+1}.q = Q[x]*10')
            exec(f'self.eq{x+1}.boost = boost[x]*intensity')
...
```

Výpis 2.1: Metóda filtering()

Metóda **calculate_pitch()**, zobrazená na výpise kódu 2.2, má na vstupe taktiež parametre z bloku *Synthesis* a mení nastavenie oscilátorov, konkrétne frekvenciu a počet harmonických zložiek. Frekvencia sa vyráta vynásobením získaných namapovaných hodnôt z prvkov **H** a **V**. Počet harmonických zložiek získavame z hodnôt saturácie **S**, rovnako ako aj hodnotu amplitúdy. Zakomponovali sme do skriptu aj pamäť predošlých frekvencií, aby ku zmene dochádzalo iba v prípade zmeny tónu.

```
...
def caculate_pitch(self, pitch_array, octave,
                   volnumber, saturation):
    for x in range(len(pitch_array)):
        if self.prev_freq[x] != pitch_array[x]:
            exec(f'self.osc{x}.harms = int(saturation[x])')
            exec(f'self.osc{x}.freq =
                                pitch_array[x]*octave[x]')
            exec(f'self.osc{x}.mul = volnumber')
        else:
            pass
    self.prev_freq = pitch_array
...
```

Výpis 2.2: Metóda calculate_pitch()

Táto syntéza je volaná na každej snímke videa, čo pre tempo skladby znamená, že je závislá na počte snímok za sekundu a samotnom obsahu videa, no najmenší možný časový interval je $1/FPS$[10]. Výsledkom je viac-hlasný zvuk s maximálnym počtom piatich generovaných hlasov v priebehu skladby. Celý zdrojový kód tohto bloku sa nachádza v prílohe A, v súbore `m_1.py`.

[10] Frames per second - počet snímok za sekundu

2.7 Rýchlosť algoritmu

Pre správny chod programu sme potrebovali zistiť rýchlosť výpočtu a zaistiť tak, aby nedochádzalo k chybám a nesprávnej rýchlosti prehrávania videa. Do hlavného cyklu programu sme preto vložili časovač, ktorý zisťuje dobu vykonávania výpočtov syntézy a porovnáva tento čas s časovačom snímok. Týmto princípom sme otestovali aplikáciu na piatich rôznych videách. Výsledky tohto merania môžeme vidieť na tabuľke 2.1.

Tab. 2.1: Namerané hodnoty pre jednotlivé testovacie videá

Video č. [-]	Priemerný čas algoritmu $T_a[s]$	Priemerný rozdiel časov $T_{fps} - T_a[s]$	Pomer medzi T_a a T_{fps}[%]	FPS $[s^{-1}]$	Kvalita [px]
1	0.0097	0.0097	29.24	30	1280×720
2	0.0096	0.0096	28.85	29,97	1280×720
3	0.0167	0.0167	41.85	25	1920×1080
4	0.0166	0.0182	41.57	25	1920×1080
5	0.0578	-0.0178	144.44	25	4096×2160

Na základe výsledkov merania môžeme vidieť, že algoritmus stíha syntetizovať zvuk v reálnom čase pri všetkých rozlíšeniach, no pri rozlíšení 4K@25 (4096×2160 pri FPS = 25) už čas na spracovanie tvorí 1,4-násobok času čakania medzi jednotlivými snímkami a nestíha sa tak syntéza v reálnom čase. Algoritmus tak zvláda syntézu pri rozlíšeniach do Full HD (vrátane). Keďže bol test rýchlosti pre rozlíšenia menšie ako 4K úspešný a algoritmus aplikácie tak prebieha v dostatočnej rýchlosti, použili sme vypočítaný čas v skripte a odčítali sme ho od časovača snímok - dostali sme tak uniformné prehrávané videa s pevným časom medzi snímkami.

2.8 Zhodnotenie a možné vylepšenia

Aplikácia *Vsyntha* je založená na aditívnej syntéze zvuku z dát videa v reálnom čase. Zvuk je generovaný ako hudobná skladba, pripomínajúca skôr avantgardný alebo post-modernistický štýl kompozície.

Pri zhodnotení tejto aplikácie sa javia aj možnosti vylepšenia algoritmu, ako napríklad:

- **Optimalizácia kódu** - úprava cyklov použitých v skriptoch pre rýchlejší výpočet;
- **Pridanie ďalších stupníc** - momentálne je dostupný iba variant medzi chromatickou a durovou stupnicou. Aplikácia by sa mohla rozšíriť o molové, cirkevné a exotické stupnice;
- **Tempo** - pridanie možnosti ovládania rýchlosti prehrávania skladby, respektíve videa, čo by mohlo byť dosiahnuté napr. zmenou počtu snímok za sekundu;
- **Vyššie rozlíšenia** - pre výpočet pri vyšších rozlíšeniach by mohlo pomôcť celkové zrýchlenie kódu a to by bolo možné napríklad pri prepise aplikácie do jazykov ako *C, C++* alebo *Fortran*. Alternatívne by sme mohli škálovať snímky videa na vstupe;
- **Variabilný počet hlasov** - pridanie možnosti výberu počtu hlasov podľa užívateľa, alebo podľa určeného parametru videa. Taktiež pridanie možnosti výberu smeru rozdelenia snímky.

3 ReAmper

V tejto kapitole sa pozrieme na ďalšiu metódu syntézy zvuku, ktorú sme implementovali do aplikácie s názvom *ReAmper*. Táto metóda je založená na *soundscapingu* pomocou detekcie objektov nachádzajúcich sa na snímkoch videa. Soundscaping je proces vytvárania zvuku kombináciou rôznych zvukov, ktoré spolu vytvárajú zvukové prostredie alebo zvukovú atmosféru. Zahŕňa výber a manipuláciu s rôznymi zvukmi s cieľom vytvoriť špecifický zvukový zážitok, ktorý dopĺňa alebo umocňuje vizuálny projekt ako napríklad film, videohru alebo umeleckú inštaláciu.

3.1 Knižnice a moduly

Pri návrhu tejto aplikácie sme sa znovu rozhodli pre vytvorenie grafického užívateľského prostredia pomocou knižnice *PyQT5*. Rovnako sme znova použili knižnice *NumPy* a *OpenCV*. Novo použitými knižnicami boli *SciPy*[1], *difflib*[2], *random*[3], *soundfile*[4] a *moviepy*[5].

3.1.1 SciPy

SciPy je knižnica Pythonu s otvoreným zdrojovým kódom, ktorá je určená pre vedecké a technické výpočty. Obsahuje moduly na optimalizáciu, lineárnu algebru, integráciu, interpoláciu, špeciálne funkcie, FFT, spracovanie signálov a obrazov, riešenie ODE a mnohé ďalšie úlohy, ktoré sú bežné v oblasti vedy a techniky. Vývoj SciPy prebieha verejne na platforme *GitHub*[6], pod dohľadom komunity *SciPy* a širšej vedeckej komunity.

3.1.2 Difflib

Difflib je modul jazyka Python, ktorý poskytuje funkcie na porovnávanie textových súborov alebo reťazcov a následné zistenie rozdielov medzi nimi. Modul difflib

[1] `https://scipy.org/`
[2] `https://docs.python.org/3/library/difflib.html`
[3] `https://docs.python.org/3/library/random.html`
[4] `https://pysoundfile.readthedocs.io/en/latest/`
[5] `https://zulko.github.io/moviepy/`
[6] Webová platforma pre správu a zdieľanie kódu

umožňuje získanie základných informácií o rozdieloch, ako sú vložené, vymazané alebo nahradené znaky v porovnávaných súboroch. Tento modul sa používa na porovnávanie verzií súborov alebo na hľadanie podobností v súboroch s podobným obsahom.

3.1.3 Random

Random je modul jazyka Python, ktorý poskytuje funkcie na generovanie náhodných čísel. Tento modul umožňuje generovať náhodné čísla z rôznych distribúcií, vrátane rovnomerného rozdelenia, normálneho rozdelenia a ďalších iných distribúcií. Okrem toho, umožňuje aj nastavovať generátor náhodných čísel a pracovať s ním. *Random* modul je výhodný pre mnohé aplikácie, napríklad pri tvorbe simulácií, testovaní algoritmov, ale aj pri generovaní náhodných hesiel a iných bezpečnostných kódov.

3.1.4 Soundfile

Soundfile je knižnica jazyka Python, ktorá poskytuje funkcie na čítanie a zápis zvukových súborov rôznych formátov. Táto knižnica umožňuje prácu so súbormi vo formátoch ako **WAV**, **FLAC**, **OGG** a **AIFF**. *Soundfile* poskytuje jednoduchý a intuitívny spôsob práce so zvukovými súbormi, vrátane čítania a zápisu zvukových dát, zisťovania informácií o súbore a poskytuje taktiež konverzie medzi rôznymi formátmi. Knižnica soundfile je často používaná v aplikáciách pre zvukové spracovanie a analýzu, ako aj v projektoch, ktoré vyžadujú prácu so zvukovými súbormi.

3.2 Grafické užívateľské prostredie

Pri návrhu tejto aplikácie sme rozhodli, že grafické užívateľské prostredie aplikácie by malo obsahovať prvky, ktoré by ponúkali nasledujúce možnosti:

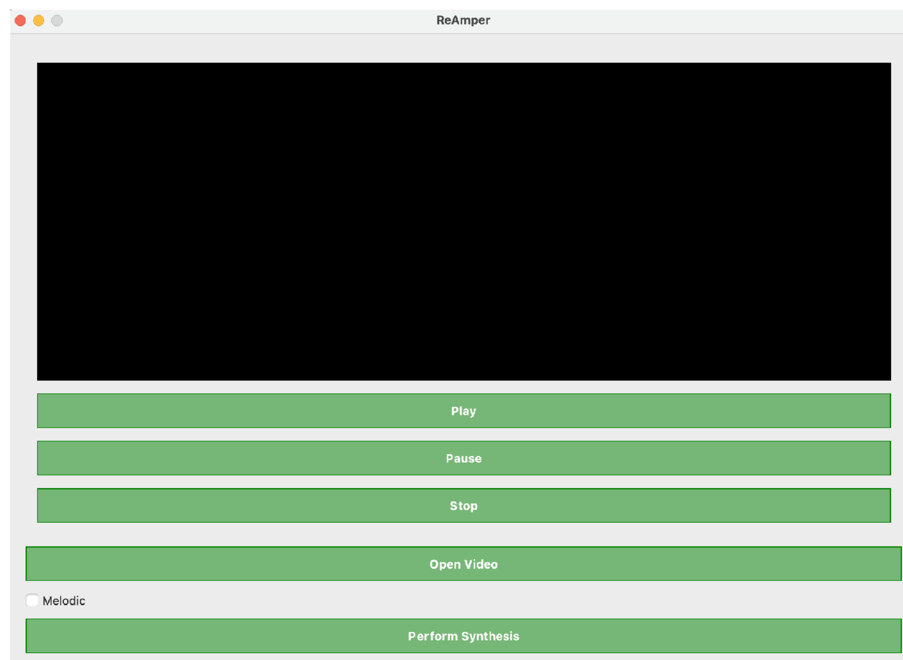
- Tlačidlo pre načítanie videa zo súboru
- Tlačidlo pre spustenie syntézy
- Tlačidlo pre uloženie vygenerovaný zvuk do súboru na disku
- Zaškrtávacie pole, ktoré určuje, aký typ sytnézy sa má aplikovať[7]
- Prehrávač videa s možnosťou spustiť a zastaviť prehrávanie

[7]Možnosť výberu medzi dvoma zvukovými bankami, viac v 3.4

Pre grafickú úpravu aplikácie sme použili metódu *setStyleSheet* knižnice PyQT, ktorá syntaxom pripomína jazyk *CSS* používaný na štylizáciu HTML webových aplikácií. Upravili sme vizuál jednotlivých tlačidiel a textu, čo môžeme vidieť na výpise 3.1. Výsledné grafické užívateľské prostredie môžeme vidieť na obr. 3.1.

```
app.setStyleSheet(" QPushButton {
    background-color: rgba(0, 128, 0, 0.5);
    color: white;
    font-weight:
    bold; border: 1px solid green;
    padding: 10px; }")
```

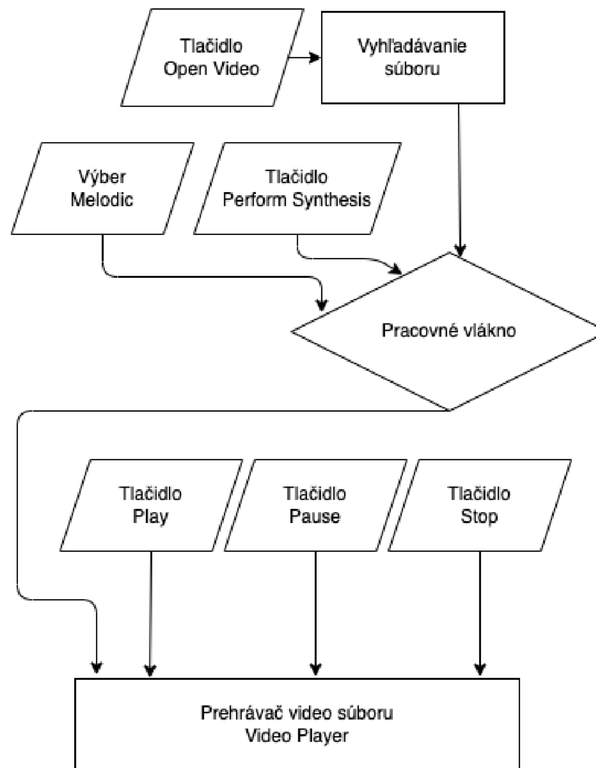
Výpis 3.1: Štýl PyQT aplikácie ReAmper



Obr. 3.1: Grafické užívateľské prostredie aplikácie ReAmper

Pre načítanie videosúboru sme vytvorili tlačidlo **Open Video**, ktoré po stlačení otvorí dialógové okno pre výber súboru z disku. Po výbere video súboru sa uvoľní tlačidlo **Perform Synthesis**, ktoré spúšťa algoritmus syntézy, ďalej popísaný v časti 3.3.2 a 3.5. Pred spustením syntézy máme na výber zaškrtnutie políčka **Melodic**, ktoré určuje, ktorá zvuková banka sa má pri syntéze použiť. Po úspešnom vykonaní procesu syntézy sa do prehrávača načíta video so syntetizovaným zvukom a uvoľnia sa tlačidlá **Play**, **Pause** a **Stop**, ktoré slúžia na prehrávanie spojeného videa

s vygenerovaným zvukovým signálom. Vývojový diagram grafického užívateľského prostredia môžeme vidieť na obr. 3.2



Obr. 3.2: Diagram GUI aplikácie ReAmper

Načítanie videa je vykonané pomocou objektu **QFileDialog()**, vďaka ktorému dokážeme vyvolať grafické dialógove okno a nájsť tak cestu k nášmu súboru. Výber súboru v tomto dialógovom okne je obmedzený na video súbory typu **MP4**, **MOV** a **AVI** pomocou modifikátora *setNameFilter*.

Prehrávanie vygenerovaného zvukového súboru je vykonané pomocou objektu **QMediaPlayer()**, ktorý umožňuje prehrávať média vrámci balíčka PyQT v module *QtMultimedia*. Po vygenerovaní zvukového súboru sa tomuto objektu priradí jeho cesta pomocou metódy *QUrl.fromLocalFile* a pomocou funkcií *play()* a *pause()*, pripojenými k odpovedajúcim tlačidlám, je zvuk zo súboru prehrávaný.

Samotná syntéza je vykonaná vrámci funckie *perform_prediction()*, ktorá spúšťa pracovné vlákno **MethodThread()** a využíva funkcie zo súboru `process.py`, ktorý môžeme nájsť v prílohe A, v zložke aplikácie *ReAmper*. Výstup vo forme videa sa potom načíta do prehrávača, ktorý môžeme ovládať prostredníctvom funkcií *play()*, *pause()* a *stop()*. Tento proces je zahrnutý v súbore `main.py`, nachádzajúcom sa v prílohe A v zložke aplikácie *ReAmper*.

3.3 Objekty

Táto metóda je založená na objektoch, ktoré sa nachádzajú na snímkach videa, preto sme potrebovali nájsť spôsob, ako by sme dokázali tieto objekty detegovať, a kedže sa jedná o video, potrebujeme tieto objekty aj sledovať a identifikovať na rôznych obrázkoch - snímkach videa. Preto sme sa rozhodli využiť možnosti detekcie objektov pomocou dostupných neurónových sietí a implementácií.

3.3.1 Detekcia objektov

Pre detekciu objektov existujú možnosti ako:

- **Faster R-CNN**[8] - široko používaný algoritmus detekcie objektov, ktorý sa skladá zo siete návrhu regiónu (RPN) na generovanie návrhov potenciálnych ohraničení objektov, po ktorej nasleduje konvolučná sieť na klasifikáciu navrhnutých regiónov a spresnenie súradníc ohraničenia.
- **YOLO**[9] - algoritmus na detekciu objektov v reálnom čase, ktorý spracúva celý obraz v jednom priechode a priamo predpovedá ohraničujúce polia a pravdepodobnosti tried, rozdeľuje obraz na mriežku a každej bunke mriežky priradí ohraničujúce boxy a pravdepodobnosti tried.
- **SSD**[10] - algoritmus detekcie objektov v reálnom čase, ktorý používa sériu konvolučných vrstiev s rôznymi stupnicami na predpovedanie ohraničujúcich polí a pravdepodobností tried.
- **RetinaNet**[11] - jednostupňový model detekcie objektov, ktorý rieši problém nevyváženosti tried pri detekcii objektov zavedením fokálnej straty. Na predpovedanie tried objektov a súradníc ohraničujúcich boxov používa sieť pyramídových prvkov FPN a klasifikačnú podsieť.
- **Mask R-CNN**[12] - rozširuje algoritmus Faster R-CNN tak, aby okrem ohraničujúcich polí a pravdepodobností tried predpovedal aj masky objektov. Do siete zavádza vetvu na generovanie masiek na úrovni pixelov pre každý zistený objekt.

Z týchto dostupných riešení sme sa rozhodli pre prácu s YOLO (You Only Look

[8] https://arxiv.org/abs/1506.01497
[9] https://www.v7labs.com/blog/yolo-object-detection
[10] https://arxiv.org/abs/1512.02325
[11] https://paperswithcode.com/method/retinanet
[12] https://github.com/matterport/Mask_RCNN

Once) vo verzií 4 (YOLOv4)[13]. Výstupom modelu je ohraničenie objektu s klasifikáciou objektu a istotou predikcie, čo sú parametre, ktoré môžeme využiť pri implementácií našej metódy.

3.3.2 Sledovanie objektov a detekcia pohybu

Ako vstup používame dáta videa, teda jednotlivé snímky, preto sme potrebovali nájsť spôsob, akým skombinovať detekciu objektov s ich sledovaním. Rozhodli sme sa použiť niektorý z dostupných algoritmov a neurónových sieti pre toto využitie, medzi ktoré patria:

- **EfficientDet**[14] - architektúra detekcie objektov, ktorá kombinuje efektívne chrbticové siete s efektívnymi detekčnými hlavami. Dosahuje dobrú rovnováhu medzi presnosťou a výpočtovou efektívnosťou.
- **SORT**[15] - populárny algoritmus na sledovanie viacerých objektov. Využíva Kalmanov filter na odhad stavu a maďarský algoritmus na asociáciu údajov. Zameriava na sledovanie objektov vo videosekvenciách namiesto ich detekcie.
- **CenterTrack**[16] - algoritmus na sledovanie objektov, ktorý integruje detekciu a sledovanie objektov do jednotného rámca. Predpovedá stredové body objektov a spája ich v čase s cieľom generovať stopy objektov. Dosahuje vysokú presnosť sledovania a výkon v reálnom čase.
- **DeepSort**[17] - populárny algoritmus na sledovanie viacerých objektov, ktorý rozširuje základný algoritmus SORT o funkcie hlbokého učenia. Jeho cieľom je spájať detekcie naprieč snímkam a generovať spoľahlivé stopy objektov vo videách.

V našej implementácií tejto metódy sme sa rozhodli pre využitie **DeepSORT** v kombinácií s **YOLO**. Využili sme preto dostupnú implementáciu tohto kombinovaného algoritmu[18], ktorú sme implementovali do našej aplikácie. Modifikovali sme tak existujúci súbor `object_tracker.py`, z ktorého sme vytvorili súbor `my_object_tracker.py`, oba sa nachádzajú v prílohe A v zložke *ReAmper*. Táto modifikácia spočívala hlavne v tom, že sme vytvorili funkciu *YOLO_DeepSort_Tracking()*, ktorej vstupné argumenty sú cesty k vstupnému a výstupnému videu, a ktorá vracia pole detegovaných objektov. Toto pole obsahuje informácie o každom detegovanom

[13] `https://paperswithcode.com/method/yolov4`
[14] `https://arxiv.org/abs/1911.09070`
[15] `https://arxiv.org/abs/1602.00763`
[16] `https://arxiv.org/abs/2004.01177v2`
[17] `https://learnopencv.com/understanding-multiple-object-tracking-using-deepsort/`
[18] `https://github.com/theAIGuysCode/yolov4-deepsort`

objekte vo formáte [číslo snímky, ID objektu, názov triedy, bod ohraničenia 1, bod ohraničenia 2, bod ohraničenia 3, bod ohraničenia 4]. Výstup kombinácie YOLOv4 a DeepSORT sú ohraničené detegované objekty s ich klasifikáciou a priradeným identifikačným číslom. Takýto výstup môžeme vidieť na obrázku 3.3, kde sme ako vstup použili naše testovacie video.

Obr. 3.3: Výstup kombinácie YOLOv4 + DeepSORT

Týmto spôsobom získavame v **MethodThread()** pole detegovaných objektov *detected_objects*, ktoré následne spracúvame pomocou funkcie triedy **TrackedObjects()** *CreateObjectMap()*.

Trieda TrackedObjects() obsahuje funkcie pre výpočet cesty objektu po obrazovke *CalculatePath()*, výpočet veľkosti objektu *CalculateSize()* a vytvorenie mapy objektov *CreateObjectMap()*. Pri inicializácií triedy sú vytvorené vlastné premenné *self.id*, *self.name*, *self.path*, *self.size*, do ktorých budú ukladané dáta z vlastných funkcií.

Funkcia *CalculatePath()*, ktorú môžeme vidieť na výpise 3.2 vypočítava polohu detegovaného objektu na obrazovke, ktorú vypočítava z hraničných bodov, ktoré sú výstupom detekcie. Tieto hodnoty následne vracia vo forme [číslo snímky, poloha objektu].

```python
1   def CalculatePath(object_id: int, detected_objects):
2       path = []
3       for i in range(len(detected_objects)):
4           if int(detected_objects[i][1]) == object_id:
5               box = detected_objects[i][3]
6               x = box[0]+(box[2]-box[0])/2
7               y = box[1]+(box[3]-box[1])/2
8               path.append([int(detected_objects[i][0]),
9                                               (x,y)])
10          else:
11              path.append([int(detected_objects[i][0]),
12                                              (0,0)])
13      return path
```

Výpis 3.2: Funkcia CalculatePath()

Funkcia *CalculateSize()*, ktorú môžeme vidieť na výpise 3.3 vypočítava veľkosť detegovaného objektu na obrazovke, ktorú taktiež vypočítava z hraničných bodov, ktoré sú výstupom detekcie. Tieto hodnoty následne vracia vo forme [číslo snímky, veľkosť objektu].

```python
1   def CalculateSize(object_id: int, detected_objects):
2       size = []
3       for i in range(len(detected_objects)):
4           if int(detected_objects[i][1]) == object_id:
5               box = detected_objects[i][3]
6               size.append([int(detected_objects[i][0]),
7                       (box[3]-box[1])*(box[2]-box[0])])
8           else:
9               size.append([int(detected_objects[i][0]), 0])
10      return size
```

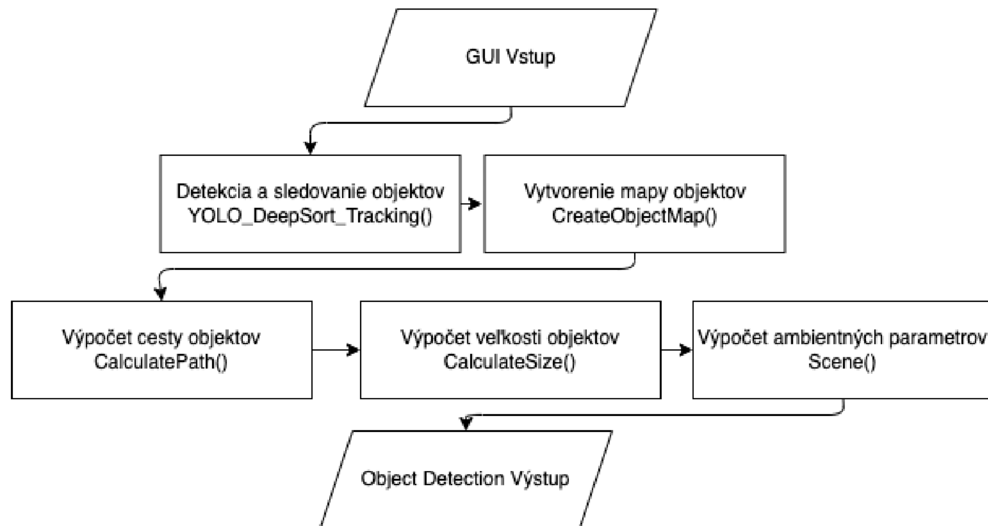
Výpis 3.3: Funkcia CalculateSize()

Funkcia *CreateObjectMap()*, ktorú môžeme vidieť na výpise 3.4 riadi výpočty cesty a veľkosti jednotlivých detegovaných objektov. Iteruje po všetkých detegovaných objektoch a realizuje vyššie spomínané funkcie *CalculateSize()* a *CalculatePath()*. Rovnako napĺňa premenné pre ID objektu a názov triedy objektu. Funkcia vracia pole takto spracovaných objektov. Túto funkciu môžeme vidieť na výpise 3.4.

```
def CreateObjectMap(detected_objects):
    ids = []
    objects = []
    for i in range(len(detected_objects)):
        if int(detected_objects[i][1]) in ids:
            pass
        else:
            ids.append(detected_objects[i][1])
            temp_object = TrackedObject()
            temp_object.id = int(detected_objects[i][1])
            temp_object.name =
                                str(detected_objects[i][2])
            temp_object.path =
        TrackedObject.CalculatePath(temp_object.id,
                                    detected_objects)
            temp_object.size =
        TrackedObject.CalculateSize(temp_object.id,
                                    detected_objects)
            objects.append(temp_object)
    return objects
```

Výpis 3.4: Funkcia CreateObjectMap()

Po vytvorení tohto poľa objektov *objects*, nastáva zisťovanie typu scény pomocou funkcie *Scene()*. Táto funcia rozdeľuje typ scény na tri základné kategórie - *ľudia*, *premávka* a *príroda*. Cieľom tejto funkcie je vytvoriť ambientný zvuk prostredia, a preto je ku každej tejto kategórií neskôr priradený ambientný zvukový súbor. Funkcia vypočítava počet jednotlivých prvkov scény a podľa ich príslušnosti k jednotlivým kategóriám vracia koeficient zosilnenia každej tejto kategórie. Táto funkcia sa nachádza v súbore `main.py` v prílohe A. Ďalej sa pristupuje k iterácií po detegovaných objektoch, kde už pristupujeme k spracovaniu zvuku, ktoré je opísané v časti 3.5. Vývojový diagram detekcie a procesu prípravy dát pred spracovaním zvuku môžeme vidieť na obrázku 3.4.



Obr. 3.4: Diagram detekcie objektov a spracovania dát

3.4 Zvuková banka

Pre priraďovanie zvukov objektom sme museli vytvoriť zvukovú banku. Táto zvuková banka obsahuje zvukové súbory, ktorých obsah odpovedá zvukom objektov, ktoré detegujeme pomocou algoritmu. Rozhodli sme pre vytvorenie takej zvukovej banky, ktorá obsahuje nielen zvuky objektov, ale aj tóny pre generáciu hudobného zvukového materiálu.

3.4.1 Zvuky objektov

Všetky zvuky objektov sme čerpali z webovej databázy **Free Sound**[19]. Priraďovali sme tak viacero zvukov pre triedy *car*, *person*, *truck*, *bus*, *traffic light*, *bicycle* a podobne. Načítanie tejto zvukovej banky je vykonané pomocou triedy **SoundLibrary()**. Táto trieda vo svojej inicializačnej triede načítava cesty k zvukovým súborom podľa užívateľom vybranom type syntézy - teda objektová alebo hudobná knižnica. Vytvorená je vlastná premenná *self.samples* typu *dict*, ktorej sú priradené zvukové súbory podľa názvu detegovanej triedy. Z tohto zoznamu sa ďalej pri spracovaní zvuku priradzujú jednotlivé zvukové súbory objektom, čo je popísané v časti 3.5.

[19] `https://freesound.org/`

3.4.2 Hudobná banka

Zvuková banka pre možnosť *melodic* je zložená zo štyroch oktáv durovej stupnice. Zvukové súbory tejto knižnice sme vytvorili pomocou súboru `scale.py`, ktorý generuje jednotlivé zvukové súbory v cykle **for** pomocou knižnice *NumPy* a *soundfile*. Časť tohto procesu, ktorý tvorí samotný signál, môžeme vidieť na výpise 3.5. Jednotlivé zvukové súbory sú pomenované podľa tried detekcie, aby bolo možné uskutočniť priraďovanie z výstupu detekcie.

```
1  ...
2  for i in range(0, octaves):
3      for j in range(len(t)):
4          n = (t[j]*(i+1))
5          period_duration = 1/n
6          x = np.linspace(0, period_duration, int(44100
7                      * period_duration), endpoint=False)
8          y = (np.sin(2 * np.pi * n * x))
9          if t[j] == c:
10             sf.write(f"soundlibrary/melodic/car{i+1}.wav",
11                                                   y, 44100)
12         if t[j] == d:
13             sf.write(f"soundlibrary/melodic/person{i+1}.wav",
14                                                   y, 44100)
15 ...
```

Výpis 3.5: Časť súboru scale.py

3.5 Spracovanie zvuku

Proces spracovania zvuku je založený na priradení zvuku zo zvukovej banky jednotlivým objektom, úprave daného zvukového signálu podľa získaných dát pohybu a veľkosti objektu. Konkrétne aplikujeme HRTF[20] panorámu podľa pohybu objektu a ovplyvňujeme intenzitu daného zvuku podľa veľkosti objektu. Všetky funkcie a triedy pre spracovanie zvuku sa nachádzajú v súbore `sound_proces.py`, ktorý sa nachádza v zložke *ReAmper* v prílohe A.

[20] Head-related transfer function - odozva, ktorá charakterizuje, ako ucho prijíma zvuk z bodu v priestore

Pokračovanie algoritmu tiredy **MethodThread()** je iterovanie po jednotlivých objektoch, pričom v každej iterácii je daný objekt použitý pre funkciu *place_audio()*, výstupom ktorej je následne naplnené pole *output_data*.

3.5.1 Korekcia objektových máp

Funkcia *place_audio()*, nachádzajúca sa v súbore *sound_process.py*, vytvára na báze vstupných argumentov výstup - umiestnený zvukový signál. V prvom kroku sú využité funkcie *interpolate_path()* a *interpolate_size()*.

Funkcia *path_interpolation()* slúži na interpoláciu cesty objektu. Cesta objektu ako vstupný parameter je reprezentovaný ako zoznam bodov, táto funkcia odstráni body s nulovou hodnotou a následne interpoluje medzery medzi zvyšnými bodmi. Funkcia prechádza každý bod v ceste objektu a ak jeho druhý prvok nie je (0, 0), pridá ho do poľa *interpolated_path*. Následne je vytvorené prázdne pole s názvom *result*. Funkcia prechádza každý bod v *interpolated_path* a pridáva ho do *result*. Ak je index bodu menší ako dĺžka (*interpolated_path* - 1), vypočíta sa rozdiel medzi súradnicami nasledujúceho bodu a aktuálneho bodu. Ak je tento rozdiel väčší ako 1, znamená to, že medzi týmito bodmi existuje medzera, a preto sa vykoná interpolácia. V rámci interpolácie sa postupne generujú nové body, ktoré sa pridávajú do *result*. Výsledkom je zoznam bodov, ktorý obsahuje pôvodné body cesty objektu a interpolované body, ktoré vypĺňujú medzery medzi týmito bodmi. Túto funkciu môžeme vidieť vo výpise 3.6

Funkcia *size_interpolation()* slúži na interpoláciu veľkosti objektu. Veľkosť objektu ako vstupný parameter je reprezentovaný ako zoznam hodnôt, v ktorom táto funkcia odstráni hodnoty s nulovou hodnotou a následne interpoluje medzery medzi zvyšnými hodnotami. Funkcia prechádza každý index v objekte a ak hodnota príslušnej veľkosti nie je 0, pridá ju do zoznamu *interpolated_size*. Tento krok slúži na odstránenie hodnôt, ktoré majú nulovú hodnotu a nemajú vplyv na interpoláciu. Následne je vytvorený prázdny zoznam s názvom *result*. Funkcia prechádza každú hodnotu v *interpolated_size* a pridáva ju do poľa *result*. Ak index hodnoty je menší ako dĺžka (*interpolated_size* - 1), vypočíta sa rozdiel medzi hodnotami nasledujúcej hodnoty nasledujúceho prvku a aktuálneho prvku. Ak je tento rozdiel väčší ako 1, znamená to, že medzi týmito hodnotami existuje medzera, a preto sa vykonáva interpolácia. V rámci interpolácie sa postupne generujú nové hodnoty, ktoré majú rovnakú hodnotu veľkosti ako aktuálna hodnota. Tieto nové hodnoty sa pridávajú do poľa *result*. Túto funkciu môžeme vidieť vo výpise 3.7

```
def path_interpolation(object, info: dict):
    interpolated_path = []
    for i in range(len(object.path)):
        if object.path[i][1] != (0,0):
            interpolated_path.append(object.path[i])
    result = []
    for i in range(len(interpolated_path)):
        result.append(interpolated_path[i])
        if i < len(interpolated_path) - 1:
            diff =
    interpolated_path[i+1][0] - interpolated_path[i][0]
            if diff > 1:
                for j in range(1, diff):
                    result.append((interpolated_path[i][0]+j,
                                interpolated_path[i][1]))
    object.path = result
```

Výpis 3.6: Funkcia path_interpolation()

```
def size_interpolation(object, info: dict):
    interpolated_size = []
    for i in range(len(object.size)):
        if object.size[i][1] != 0:
            interpolated_size.append(object.size[i])
    result = []
    for i in range(len(interpolated_size)):
        result.append(interpolated_size[i])
        if i < len(interpolated_size) - 1:
            diff =
    interpolated_size[i+1][0] - interpolated_size[i][0]
            if diff > 1:
                for j in range(1, diff):
                    result.append((interpolated_size[i][0]+j,
                                  interpolated_size[i][1]))
                    if len(interpolated_size[i]) > 2:
                      result[-1] += interpolated_size[i][2:]
```

Výpis 3.7: Funkcia size_interpolation()

3.5.2 Nastavenie dĺžky signálu

Ďalej sa vo funkcií *place_audio()* priradzuje zvukový súbor objektu, a následne je tento zvukový súbor načítaný pomocou funkcie *read_audio_samples()*. Následne prebieha blok kódu, ktorý sa stará o úpravu dĺžky signálu, aby odpovedal dĺžke videa. Využívajú sa funkcie knižnice NumPy *np.concatenate* a *np.tile*. V prípade, že je video kratšie ako zvukový súbor, výsledný signál je skrátený, a ak je video dlhšie ako zvukový súbor, signál je nakopírovaný pomocou *np.tile*. Na začiatok každého toho signálu je náhodne vybraný počet nulových prvkov v rozmedzí 0 - 200, ktoré slúži na rozlíšenie jednotlivých, viackrát použitých zvukových súborov.

Ďalej je zvukový signál rozdelený podľa celkového počtu snímok a realizovaná je iterácia po týchto čiastkových zvukových signáloch. Vrámci tejto iterácie je vypočítané zosilnenie podľa veľkosti objektu a panoráma pomocou cesty objektu, ktorej sa venujeme v časti 3.5.3. Tieto hodnoty sú vypočítané na aktuálnej a nasledujúcej snímke, a následne je pomocou týchto hodnôt realizovaný crossfade[21], aby bol prechod medzi zvukmi plynulý.

3.5.3 HRTF panoráma

HRTF (Head-Related Transfer Function) panoráma je technika zvukového spracovania, ktorá sa používa na vytvorenie priestorovej lokalizácie zvukových zdrojov. Vytvára ilúziu zvuku, ktorý sa pohybuje v priestore okolo poslucháča z rôznych smerov, na čo využíva vlastnosti ľudských uší a hlavy. Každé ucho zachytáva zvuky z rôznych smerov a prenáša ich do vnútorného ucha cez vonkajší zvukovod, čo vytvára odchýlky vo zvukovej energii a fázovej štruktúre, ktoré sa využívajú pri tejto technike. Proces HRTF panorámy spočíva v zaznamenaní a reprodukcii zvukových vzoriek s rôznymi hodnotami pre rôzne smerové polohy zvukového zdroja. Tieto vzorky sa potom prehrávajú v závislosti od polohy zvukového zdroja v priestore. Pri správnom použití HRTF panorámy je dosiahnutý efekt, kde zvuky vychádzajúce z rôznych smerov a vzdialeností sú vnímané poslucháčom ako presne lokalizované v priestore. Táto panoráma sa často využíva v aplikáciách virtuálnej reality a rozšírenej reality, kde presné lokalizácie zvukových zdrojov prispievajú k realistickejšiemu zážitku. Tiež sa používa pri nahrávaní zvuku v stereo alebo viac kanálových formátoch, kde je snaha zachytiť priestorovú polohu zvukových zdrojov a vytvoriť realistickú akustickú scénu pre poslucháča.

[21] technika, ktorá sa používa pri prechode medzi dvoma zvukovými stopami

V našej implementácií sa o HRTF panoramovanie stará funkcia *getHRTFdata()*. Táto funkcia slúži na získanie dát HRTF databázy na základe súradníc zvukového zdroja v priestore a rozmerov priestorového modelu. Funkcia prijíma dve vstupné parametre: dvojicu súradníc zvukového zdroja v priestore (x, y), a dvojicu určujúcu rozmery priestorového modelu (šírka, výška). Najskôr sa definuje cesta k priečinku obsahujúcemu HRTF databázu, konkrétne sa jedná o databázu CIPIC HRTF[22]. V tomto priečinku sa nachádzajú zvukové súbory HRTF. Následne sa určuje azimut a elevácia zvukového zdroja na základe súradníc xy a rozmerov priestorového modelu. Azimut udáva uhol horizontálnej polohy zvukového zdroja a elevácia udáva uhol vertikálnej polohy zvukového zdroja. Pre každé ucho sa načíta príslušný zvukový súbor HRTF, ktorý najlepšie zodpovedá danému azimutu a elevácii. Načítaný zvukový súbor je transformovaný na impulznú odozvu, ktorá je reprezentovaná ako pole hodnôt. Nakoniec sa vráti pole impulzných odoziev pre ľavé a pravé ucho, čo predstavuje HRTF dáta pre dané súradnice zvukového zdroja v priestore. Túto funkciu môžeme vidieť na výpise 3.8.

Po získaní týchto dát funkcia *place_audio()* vykoná konvolúciu medzi pôvodným singálom a impulznou odozvou získanou *getHRTFdata()*. Po vykonaní týchto úkonov sú naplnené polia *gL* a *gR*, ktoré funkcia vracia ako ľavý a pravý kanál umiestneného zvukového signálu objektu.

[22]Oficiálna stránka v čase písania už nedostupná, dostupné na: `https://github.com/amini-allight/cipic-hrtf-database`

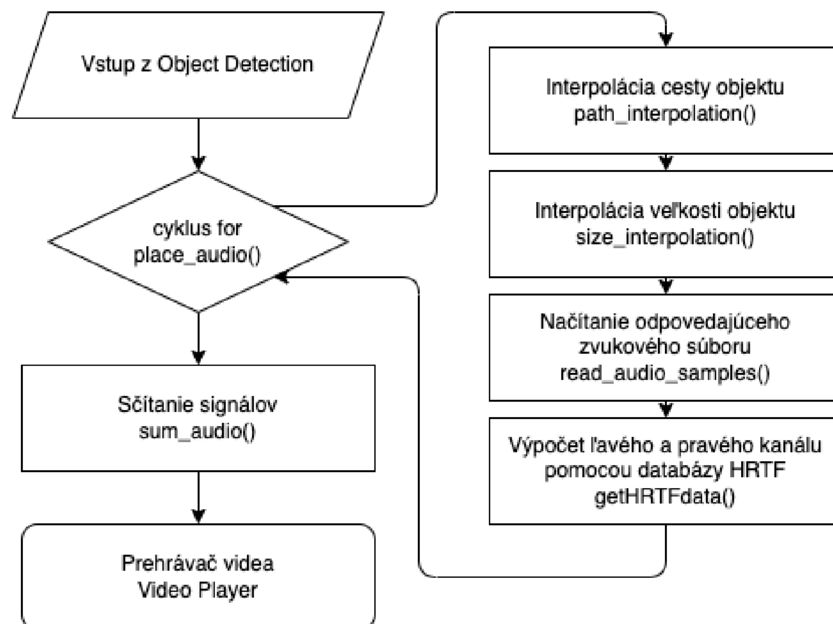
```python
def getHRTFdata(xy: tuple, dimensions: tuple):
    path = "./HRTF/data/"
    list = [f for f in os.listdir(path)]
    ear = ["left","right"]
    ir = []
    x = (xy[0] - dimensions[0]/2)/dimensions[0]/2
    y = (xy[1]/dimensions[1]) # rozsah 0:1
    if x>=0:
        azimuth = int(x*90)
        xx = ""
    else:
        azimuth = 360 - int(x*90 + 360)
        xx ="neg"
    elevation = int(y*45)
    for i in range(2):
        filename = f"{xx}{azimuth}az{ear[i]}.wav"
        match = difflib.get_close_matches(filename, list)
        fs, samples = wav.read(path+match[2])
        response=[]
        # for j in range(len(samples)):
        #     response.append(samples[j][elevation])
        response = np.array(samples[elevation])
        ir.append(response)
    return ir
```

Výpis 3.8: Funkcia getHRTFdata()

3.5.4 Sčítanie zvukov

Ďalším krokom v **MethodThread()** je volanie funkcie *sum_audio()*, ktorej výstup je syntetizovaný zvuk scény. Táto funkcia slúži na sčítanie audio dát z umiestnených zdrojov a generovanie výstupného zvukového súboru. Najprv funkcia inicializuje dve prázdne premenné *outputL* a *outputR*, ktoré budú slúžiť na sčítanie zvukových dát. Pomocou funkcie *SetAmbientSound()* je získaný ambientný zvuk pre danú scénu. V cykle for prechádza jednotlivé zvukové dáta a sčíta ich hodnoty pre ľavý a pravý kanál. K *outputL* a *outputR* sa prIráta aj ambientný zvuk, aby sa zohľadnila celková atmosféra prostredia. Následne sa vypočítajú maximálne hodnoty pre ľavý a pravý kanál *mL* a *mR*, vypočíta sa maximálna hodnota z týchto maximálnych hodnôt. Zvukové dáta sa normalizujú delením ich hodnôt touto maximál-

nou hodnotou. Ďalej je vytvorené výstupné dvojkanálové pole *out* pomocou funkcie *np.column_stack*. Výstup sa ďalej zapíše do súboru *output.wav* so vzorkovacou frekvenciou 44100 Hz pomocou funkcie *sf.write*. Na záver je nahradená pôvodná zvuková stopa syntetizovanou vo vstupnom video súbore pomocou funkcie *replace_audio()*. Táto funkcia slúži na nahradenie zvukovej stopy vo videu novou zvukovou stopou. Najprv sa načíta vstupné video pomocou objektu **VideoFileClip** a nový zvukový súbor pomocou objektu **AudioFileClip**. Ďalej je vytvorený nový video klip, kde sa pôvodná zvuková stopa nahradí novým zvukovým súborom pomocou metódy *set_audio()*. Nový video klip je zapísaný do výstupného súboru pomocou metódy *write_videofile()*. Toto nové video je následne priradené prehrávaču v grafickom užívateľskom prostredí. Vývojový diagram spracovania zvuku môžeme vidieť na obrázku 3.5.



Obr. 3.5: Diagram spracovania dát

3.6 Rýchlosť metódy

Rýchlosť metódy je ovplyvnená hlavne algoritmom YOLO + DeepSORT. Pri našom testovaní dosahovala rýchlosť detekcie okolo 1,49 snímky za sekundu, čo zďaleka nieje akceptovateľné pre nasadenie v reálnom čase. Dôvodom je nevyužívanie akcelerácie grafickej karty, keďže testovací systém nemá k dispozícií dostatočne silnú grafickú kartu s podporou CUDA[23]. Detekcia a sledovanie objektov tak pre 13 s video trvá 244.38 s. Potenciálne zapojenie grafického procesora s podporou CUDA by mohlo výrazne zrýchliť tento proces a vytvoriť tak prostriedky pre použitie tejto metódy v reálnom čase. Vytvorenie objektových máp pre celé video trvá 4,53 s a spracovanie zvuku trvá v priemere 2.58 s na objekt pre celé video. Tieto merania sme vykonali pomocou knižnice *time* a metódy *time()*.

3.7 Zhodnotenie a možné vylepšenia

Aplikácia *ReAmper* vykonáva syntézu zvuku na báze detekcie a sledovania objektov. Priradzuje každému detegovanému objektu zvukovú stopu, ktorá je procesovaná v závislosti na svojej veľkosti a polohe. Vrámci možných vylepšení tejto aplikácie a metódy syntézy zvuku z dát videa, môžeme spomenúť:

- **Optimalizácia aplikácie na systéme s grafickou kartou NVidia** a podporou CUDA, čo by zrýchlilo algoritmus a pravdepodobne vytvorilo podmienky pre využitie tejto metódy v reálnom čase;
- **Výber podrobnejšej databázy HRTF** zameranej presnejšie na náš rozsah použitej elevácie a azimutu, čo by malo za následok kvalitnejší proces HRTF panoramovania;
- **Využitie alebo vytvorenie neurónovej siete**, ktorá by generovala zvukový signál podľa výstupu algoritmu YOLO + DeepSORT;
- **Výmena algoritmu DeepSORT** za iný, napríklad spomínaný algoritmus CenterTrack.

Všetky tieto navrhované zmeny by prispeli k skvalitneniu a zrýchleniu výstupu aplikácie a optimalizáciou rýchlosti tak, aby sa dala metóda nasadiť v reálnom čase.

[23] platforma vyvinutá spoločnosťou NVIDIA, ktorá umožňuje využívať výpočtovú silu grafických procesorov na rýchle a paralelné spracovanie úloh

4 SegMentor

Ďalej sa pozrieme na tretiu, poslednú metódu syntézy zvuku, ktorú sme implementovali do aplikácie s názvom *SegMentor*. Táto metóda je založená na obrazovej segmentácií s využitím protokolu **MIDI**[1]. Metóda sa dá rozdeliť do troch krokov: segmentácia snímok videa, vytvorenie mapovacích tenzorov a generovanie MIDI súboru. Na tieto jednotlivé kroky sa pozrieme v nasledujúcej kapitole.

4.1 Knižnice a moduly

Pri návrhu tejto aplikácie sme sa znovu rozhodli pre vytvorenie grafického užívateľského prostredia pomocou knižnice *PyQT5*. Rovnako sme znova použili knižnice *NumPy* a *OpenCV*. Novo použitými knižnicami boli *hmmlearn*[2] a *mido*[3].

4.1.1 HMMlearn

HMMlearn je knižnica pre Python, ktorá poskytuje implementácie skrytého Markovovho modelu - Hidden Markovov Model - **HMM**. HMM sú štatistické modely, ktoré sa používajú na analýzu sekvenčných dát, ako sú napríklad rečové signály, genetické sekvencie alebo pohyby objektov. *HMMlearn* poskytuje nástroje na trénovanie a evaluáciu, ako aj na dekódovanie a predikciu dát na základe modelu HMM. Okrem toho poskytuje aj rôzne typy HMM, ako diskrétne HMM, GaussianHMM a Hidden Semi-Markovov Model. Knižnica je distribuovaná ako open-source softvér s licenciou BSD a je k dispozícii na stiahnutie z *GitHub-u* alebo prostredníctvom *pip*.

4.1.2 Mido

Mido je knižnica pre spracovanie a generovanie súborov MIDI pre Python, ktorá poskytuje jednoduchý a prehľadný spôsob práce s MIDI dátami. Umožňuje pripojiť sa k MIDI vstupným a výstupným portom na počítači, môžete čítať vstupné MIDI udalosti zo zariadení a posielať MIDI správy na MIDI výstupné porty. Umožňuje

[1] Musical Instrument Digital Interface - protokol, ktorý umožňuje elektronickým hudobným nástrojom a zvukovým zariadeniam komunikovať a vymieňať si hudobné informácie.

[2] `https://hmmlearn.readthedocs.io/en/latest/`

[3] `https://mido.readthedocs.io/en/latest/messages.html`

čítať a zapisovať MIDI súbory, načítať existujúci MIDI súbor do pamäte a spracovať jeho udalosti alebo vytvoriť nový MIDI súbor. Poskytuje jednoduché rozhranie pre vytváranie, čítanie a modifikovanie MIDI správ, pričom sa dajú jednoducho nastaviť parametre správ, ako sú časový údaj, typ správy ako NoteOn, NoteOff, ControlChange a podobne, kanál a hodnoty jednotlivých parametrov. Umožňuje aj prácu s rôznymi druhmi hudobných udalostí, ako sú tóny, akordy, rytmické vzory a sekvencie. Môžete vytvárať a spravovať hudobné sekvencie pomocou jednoduchého a zrozumiteľného rozhrania. Poskytuje aj funkcie na synchronizáciu s MIDI hodinami a správu časových údajov ako tempo, začiatok a koniec sekvencie, synchronizácia s inými MIDI zariadeniami.

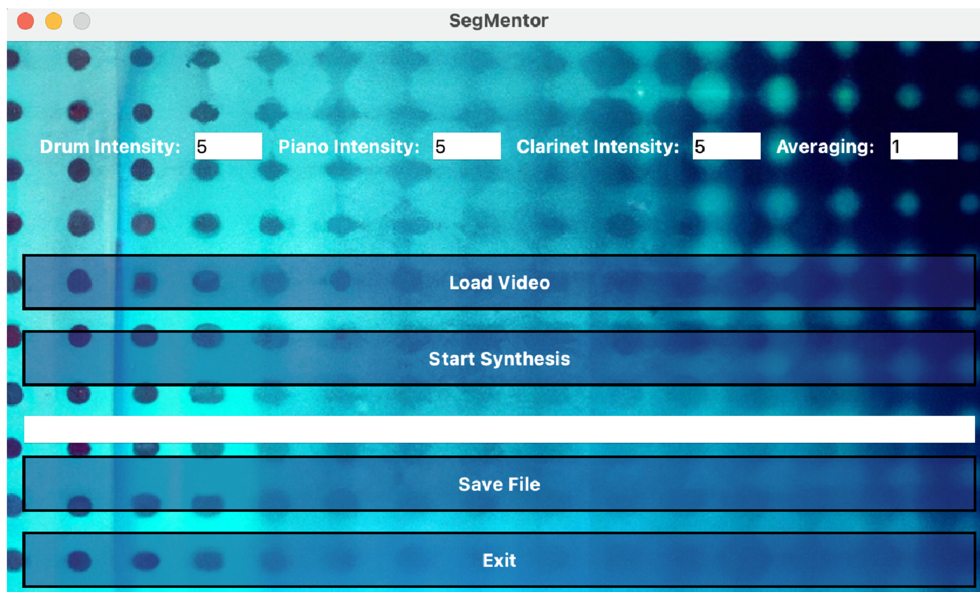
4.2 Grafické užívateľské prostredie

Pre ovládanie tejto aplikácie sme použili prvky, ktoré ponúkajú tieto možnosti:

- Tlačidlo pre načítanie videa zo súboru
- Tlačidlo pre spustenie syntézy
- Textové polia, kde sa budú dať nastaviť parametre syntézy
- Textové pole, kde by sa zobrazoval priebeh syntézy
- Tlačidlo pre uloženie vygenerovaného súboru na disk
- Tlačitlo na ukončenie aplikácie

Pre grafickú úpravu aplikácie sme znova použili metódu *setStyleSheet* knižnice *PyQT*. Upravili sme vizuál jednotlivých tlačidiel, textu a pridali sme obrázok pozadia, ktorý sme vygenerovali pomocou Dall-E 2[4]. Grafické užívateľské prostredie aplikácie SegMentor môžeme vidieť na obr. 4.1.

[4]https://openai.com/product/dall-e-2

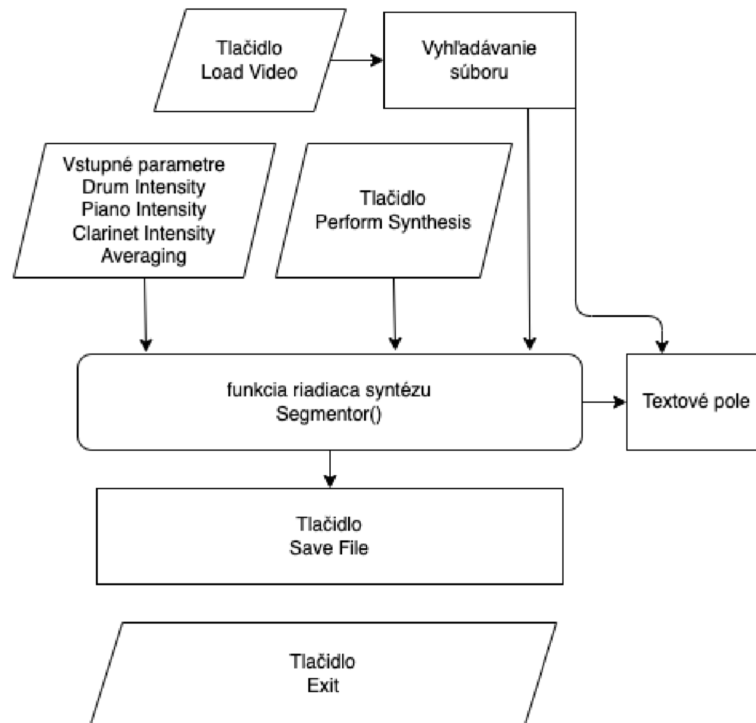


Obr. 4.1: Grafické užívateľské prostredie aplikácie SegMentor

Pre načítanie videosúboru sme vytvorili tlačidlo **Load Video**, ktoré po stlačení otvorí dialógové okno pre výber súboru z disku. Po výbere video súboru sa uvoľní tlačidlo **Synthesis**, ktoré spúšta algoritmus syntézy, ďalej popísaný v časti 4.3 a v textovom poli sa zobrazí priebeh syntézy. Užívateľ môže ovládať štyri parametre syntézy - *Drum Intensity*, *Piano Intensity*, *Clarinet Intesity* a *Averaging* pomocou štyroch textových polí, do ktorých môže užívateľ zadať číslené hodnoty týchto parametrov viac opísaných v časti 4.4. Po úspešnom vykonaní procesu syntézy sa v textovom poli vypíše potvrdzujúca hláška a tlačidlo pre uloženie súboru **Save Output** sa uvoľní. Po kliknutí na tlačidlo **Exit** sa aplikácia vypne pomocou systémovej funkcie *sys.exit()*. Vývojový diagram grafického užívateľského prostredia môžeme vidieť na obr. 4.8

Načítanie videa je vykonané pomocou objektu **QFileDialog()**, pomocou ktorého dokážeme vyvolať grafické dialógove okno a nájsť tak cestu k zvolenému súboru. Výber súboru v tomto dialógovom okne je obmedzený na video súbory typu **MP4**, **MOV** a **AVI** pomocou modifikátora *setNameFilter*.

Celkový proces syntézy je spustený po kliknutí na príslušné tlačidlo pomocou funckie *Segmentor()*. Najprv sa vykoná načítanie snímky videa, následne sa vykonajú segmentácie týchto snímok, z nich sa vytvoria mapovacie tenzory a následne je vytvorený MIDI súbor. Proces syntézy je podrobne opísaný v ďalších častiach tejto kapitoly, a to konkrétne 4.3 a 4.4.



Obr. 4.2: Diagram GUI aplikácie SegMentor

4.3 Segmentácie

Prvým krokom tejto metódy syntézy zvuku z videa, je segmentácia obrazu. Využili sme segmentácie:

- Prahová segmentácia
- Watershed segmentácia
- Segmentácia na báze detekcie hrán
- Segmentácia pomocou skrytého Markovovho modelu (HMM)
- Segmentácia pomocou metódy K-Means

Okrem týchto segmentácií používame aj diskrétnu kosínusovú transformáciu **DCT**[5], ktorú využívame na vytvorenie kontrolného signálu, viac popísaného v časti 4.6.

Pre analýzu a segmentáciu videa sme vytvorili triedu **VideoAnalyzer()**, v ktorej sme definovali funkcie pre jednotlivé segmentácie. Ďalej sa pozrieme na tieto segmentácie a našu implementáciu triedy.

[5]Discrete Cosine Transform

4.3.1 Prahová segmentácia

Prahová segmentácia je jednoduchá a často používaná technika segmentácie obrazu, ktorá rozdeľuje obraz na oblasti s rôznymi úrovňami intenzity na základe stanoveného prahu. Algoritmus prahovej segmentácie spočíva v tom, že sa určí prahová hodnota, ktorá sa použije na rozdelenie obrazu. Všetky pixely s intenzitou nižšou ako stanovená prahová hodnota sa označia ako čierna farba a všetky pixely s intenzitou vyššou ako prahová hodnota sa označia ako biela farba. Pixely tak nabúdajú hodnoty 0 alebo 255. Tento postup môže byť použitý na jednoduché binárne segmentovanie obrazu, napríklad na extrakciu objektu z pozadia alebo na rozpoznávanie znakov v textových obrázkoch. [20]

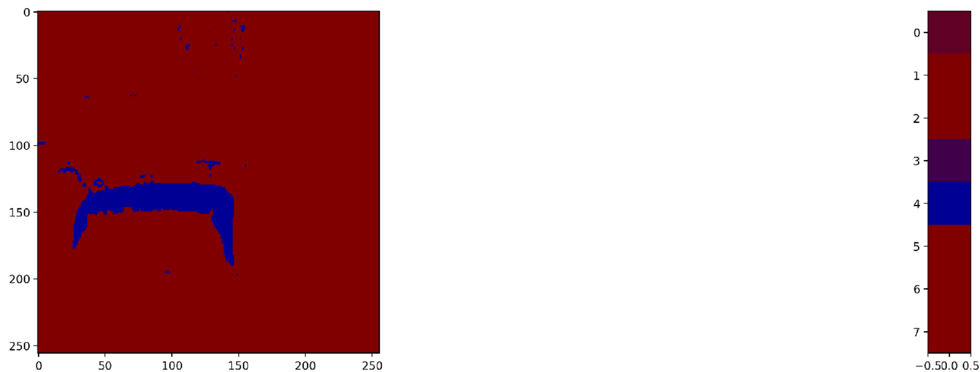
V našej implementácií využívame túto segmentáciu vo funkcií *thresholding()*, ktorej vstupný argument je pole prahových hodnôt. Táto funkcia najprv oznámi svoj priebeh do textového poľa a následne pristupuje k segmentácií snímky po snímke. Každá snímka je najprv prevedená do šedotónového farebného priestoru a následne je aplikovaná funkcia OpenCV *threshold()*, ktorá berie ako vstupné argumenty našu šedotónovú snímku, spodný a horný prah a typ prahovania, v našom prípade použitý **cv.THRESH_BINARY**[6]. Následne na tento prahovaný snímok aplikujeme farebný priestor **cv.COLORMAP_JET**[7] pomocou funkcie *applyColorMap()*. Na záver je z takto sprocesovanej snímky vytvorený mapovací tenzor o veľkosti (1, 8), ktorý je ďalej využívaný pri syntéze. Každý takto sprocesovaný snímok je pripájaný do výstupného poľa **out**, ktorý je funkciou vracaný. Tento proces môžeme vidieť na výpise kódu 4.1. Výstup takto segmentovaného snímku môžeme vidieť na obr. 4.3 na ľavej časti obrázka a na pravej môžeme vidieť výstupný tenzor, zobrazený pomocou funkcie *display_images()*, ktorá využíva knižnicu *matplotlib* a modul *pyplot*.

[6] `https://docs.opencv.org/4.x/d7/d4d/tutorial_py_thresholding.html`

[7] preddefinovaná farebná mapa, ktorá sa často používa na vizualizáciu dát

```python
def thresholding(self, limits):
  self.textfield.setText("thresholding...")
  out = []
  for img in self.frames:
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
    ret, thresh = cv.threshold(gray, limits[0], limits[1],
                                                cv.THRESH_BINARY)
    thresh = cv.applyColorMap(thresh, cv.COLORMAP_JET)
    out.append(cv.resize(thresh, (1, 8)))
  return out
```

Výpis 4.1: Funkcia thresholding()



Obr. 4.3: Výstup prahovacej funkcie

4.3.2 Watershed segmentácia

Segmentácia Watershed je technika segmentácie obrazu, ktorá sa používa na oddelenie objektov od seba. Základná myšlienka tejto segmentácie je založená na analógii vodného toku na topografickej mape. V tejto technike obraz považujeme za topografickú mapu, pričom úrovne jasu každého pixelu zodpovedajú výške príslušného bodu na mape. Potom identifikujeme lokálne minimá obrazu, ktoré zodpovedajú najnižším bodom na topografickej mape. Tak je každé lokálne minimum vyplnené značkami, ktoré považujeme za východiskové body segmentácie. Následne prebieha simulácia zaplavenia obrazu z týchto značiek, kde hladina vody stúpa, až kým nedosiahne susedné značky. Čiary rozvodia sú definované ako hranice medzi oblasťami, ktoré sú naplnené vodou z rôznych značiek. Algoritmus Watershed

je užitočný pri segmentácii obrazov, kde majú objekty jasné hranice a môžu byť oddelené. Jednou z hlavných nevýhod segmentácie pomocou watershed je problém nadmernej segmentácie, keď sa malé objekty segmentujú do samostatných oblastí. Tento problém sa dá vyriešiť použitím segmentácie riadenej značkami, pri ktorej sa značky umiestňujú ručne, aby definovali oblasti záujmu v obraze.[21]

V našej implementácií využívame túto segmentáciu vo funkcií *watershed_segmentation()*. Táto funkcia najprv vypíše do textového poľa hlásenie o svojej inicializácií a následne prebehne iterácia po snímkach videa, kde sa uskutočňuje segmentácia. Počas každej iterácie sa najprv prekonvertuje snímka na snímku šedotónovú, následne je aplikovaná funkcia *trheshold()*, kde využívame dva typy prahovacích funkcií. Hodnota 0 je dolným prahom, hodnota 255 je horným prahom a kombinácia **cv.THRESH_BINARY_INV** a **cv.THRESH_OTSU** určuje, že sa použije inverzná binarizácia s automaticky vypočítaným prahom metódou *Otsu*[8]. Ďalej sú použité morfologické operácie na vyhladenie binárnej masky, pričom je vytvorený jadrový element o veľkosti 3×3. Následne sa aplikuje operácia otvorenia **cv.MORPH_OPEN** s jadrovým elementom a počtom iterácií 2. Otvorenie pomáha odstrániť šumy a prepojiť prerušené objekty. Ďalej je aplikovaná transformácia vzdialenosti a vytvorená je predpoveď popřednej časti. Vzdialenostná transformácia **cv.distanceTransform** sa používa na výpočet vzdialeností od pozadia ku kontúram objektov na binárnej maske. Následne sa pomocou prahu získajú predpovedané popredné oblasti. Následne nastáva identifikácia neznámych oblastí a spojených komponentov. Neznáme oblasti sa získajú odčítaním predpovedaných popredných oblastí od binárnej masky a spojené komponenty pomocou **cv.connectedComponents**, kde je každej oblasti priradené jedinečné číslo. Spojeným komponentom sa pridá hodnota 1 a neznámym oblastiam sa priradí hodnota 0. Potom sa aplikuje funkcia OpenCV *cv.watershed* na vstupný obrázok, prebehne normalizácia v intervale **<0;255>** a aplikuje sa farebný priestor *cv.COLORMAP_JET*. Nakoniec je vytvorený mapovací tenzor o veľkosti **<1;7>** a funkcia vracia pole takýchto odpovedajúcich tenzorov k jednotlivým snímkam.

Tento proces môžeme vidieť na výpise kódu 4.2. Výstup takto segmentovanej snímky môžeme vidieť na obr. 4.4 na ľavej časti obrázka a na pravej môžeme vidieť výstupný tenzor, zobrazený pomocou funkcie *display_images()*.

[8] https://muthu.co/otsus-method-for-image-thresholding-explained-and-implemented/

```python
def watershed_segmentation(self):
    self.textfield.setText("Watersheding...")
    out = []
    for img in self.frames:
        gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
        ret, thresh = cv.threshold(gray, 0, 255, cv.
                    THRESH_BINARY_INV+cv.THRESH_OTSU)
        kernel = np.ones((3,3),np.uint8)
        open = cv.morphologyEx(thresh,cv.MORPH_OPEN,
                             kernel, iterations = 2)
        dtransform = cv.distanceTransform(open,cv.DIST_L2,5)
        ret, sfg = cv.threshold(dtransform,
                                     0.1*dtransform.max(),255,0)
        sfg = np.uint8(sfg)
        unknown = cv.subtract(open,sfg)
        ret, markers = cv.connectedComponents(sfg)
        markers = markers+1
        markers[unknown==255] = 0
        markers = cv.watershed(img,markers)
        markers = cv.normalize(markers,markers, 0, 255,
                                    cv.NORM_MINMAX, cv.CV_8UC1)
        markers = cv.applyColorMap(markers, cv.COLORMAP_JET)
        out.append(cv.resize(markers, (1, 7)))
    return out
```

Výpis 4.2: Funkcia watershed_segmentation()

Obr. 4.4: Výstup segmentovania watershed

4.3.3 Segmentácia na báze detekcie hrán

Segmentácia na báze detekcie hrán sa zameriava na identifikáciu hraníc alebo rozhraní medzi rôznymi objektmi na základe intenzity pixelov obrazu. Detekcia hrán, alebo tiež Edge Detection sa zvyčajne vykonáva pomocou rôznych metód spracovania obrazu, ako sú Sobel, Prewitt alebo Canny. Tieto metódy sa zameriavajú na identifikáciu zmien v intenzite pixelov v rôznych smeroch, a tým vytvárajú hrany alebo rozhrania. Po detekcii hrán sa môže aplikovať algoritmus segmentácie na rozdelenie obrazu na jednotlivé časti. To môže zahŕňať napríklad prahovanie na základe intenzity pixelov, kde pixely s hodnotou vyššou ako určitý prah sa považujú za súčasť objektu a pixely s hodnotou nižšou ako tento prah sa považujú za pozadie. V praxi sa segmentácia na báze Edge Detection používa v mnohých oblastiach, ako sú napríklad medicína, spracovanie obrazu a robotika.[20]

V našej implementácií sme vytvorili funkciu *edge_based_segmentation()*, ktorá využíva tento typ segmentácie. Najprv sa vypíše v textovom poli hláška o prebiehajúcej funkcií a následne prebehne iterácia po snímkach videa. Každá snímka je konvertovaná na šedotónovú a následne je aplikovaná funkcia *cv.Canny*, s dolným prahom 100 a horným prahom 200. Ďalej je aplikované farebné mapovanie *cv.COLORMAP_JET*. Na záver je vytvorený mapovací tenzor o veľkosti (1,7) a je priradený k poľu, ktoré je vracané funkciou.

Tento proces môžeme vidieť na výpise kódu 4.3. Výstup takto segmentovanej snímky môžeme vidieť na obr. 4.5 na ľavej časti obrázka a na pravej môžeme vidieť výstupný tenzor, zobrazený pomocou funkcie *display_images()*.

```
1    def edge_based_segmentation(self):
2        self.textfield.setText("Edge...")
3        out = []
4        for img in self.frames:
5            gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
6            edges = cv.Canny(gray, 100, 200)
7            edges = cv.applyColorMap(edges, cv.COLORMAP_JET)
8            out.append(cv.resize(edges, (1, 7)))
9        return out
```

Výpis 4.3: Funkcia edge_based_segmentation()

Obr. 4.5: Výstup segmentačnej funkcie na báze detekcie hrán

4.3.4 Segmentácia pomocou skrytého Markovovho modelu

Skrytý Markovov model **HMM** (Hidden Markovov Model) je stochastický model, ktorý sa používa na modelovanie systémov, ktoré vykazujú stavovú závislosť. V HMM existujú dva typy procesov - skrytý proces a pozorovaný proces. Skrytý proces je reprezentovaný Markovovovým reťazcom, kde každý stav v reťazci predstavuje nejaký skrytý stav. Pozorovaný proces je výstup, ktorý sa generuje na základe skrytého stavu. Hlavnou myšlienkou je, že pozorovaný proces nie je priamo pozorovateľný, ale je generovaný skrytým stavom. HMM má preto schopnosť modelovať náhodné signály s neznámymi alebo skrytými štruktúrami. HMM sa používa v rôznych oblastiach, ako napríklad rozpoznávanie reči, rozpoznávanie písma, strojové prekladanie, rozpoznávanie obrazu a biometrické identifikácie.[22]

V našej implementácií sme vytvorili funkciu *HMM()*, ktorá vykonáva túto segmentáciu. Funkcia iteruje po snímkach videa a mení ich rozlíšenie na 24×24. Ďalej je takáto snímka konvertovaná do šedotónového farebného priestoru a hodnoty jej pixelov sú normalizované na rozsah $<\mathbf{0;1}>$. Ďalej je táto snímka transformovaná do jednorozmerného vektoru. Inicializovaný je Gaussovský skrytý Markovov model so siedmimi skrytými stavmi. Model generuje postupnosť skrytých stavov, ktoré sú následne transformované do obrazu s rovnakým rozmerom ako pôvodná šedotónová snímka. Ďalej nasleduje cyklus pre transformáciu zo šedotónového obrazu na farebný pomocou priraďovania hodnôt RGB z poľa do príslušných pixelov. Následne je vytvorený mapovací tenzor o veľkosti (1,7) a vracané je pole s týmito tenzormi.

Tento proces môžeme vidieť na výpise kódu 4.4. Výstup takto segmentovanej snímky môžeme vidieť na obr. 4.6 na ľavej časti obrázka a na pravej môžeme vidieť výstupný tenzor, zobrazený pomocou funkcie *display_images()*.

```
def HMM(self):
    self.textfield.setText("HMM...")
    out = []
    colors = np.array([[255, 0, 0],
            [0, 255, 0], [0, 0, 255],
            [255, 255, 0], [255, 0, 255]])
    for frame in self.frames:
        frame = cv.resize(frame, (24, 24))
        gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY)
        gray = gray / 255.0
        features = gray.reshape(-1, 1)
        model = hmm.GaussianHMM(n_components=7,
                        covariance_type='diag')
        model.fit(features)
        state_sequence = model.predict(features)
        label_image = state_sequence.reshape(gray.shape)
        colored_image = np.zeros((label_image.shape[0],
                        label_image.shape[1], 3),
                        dtype=np.uint8)
        for i in range(colors.shape[0]):
            olored_image[label_image == i, :] = colors[i, :]
        out.append(cv.resize(colored_image, (1, 7)))
    return out
```

Výpis 4.4: Funkcia HMM()

Obr. 4.6: Výstup segmentačnej funkcie na báze HMM

4.3.5 Segmentácia pomocou K-Means

Segmentácia pomocou K-Means je algoritmus strojového učenia, ktorý sa používa na rozdelenie dát do skupín, ktorý nevyžaduje predchádzajúce označenie dát a patrí do kategórie nesupervízneho učenia[9]. Algoritmus sa používa na riešenie problému zoskupenia alebo segmentácie dát na základe podobnosti ich vlastností. Funguje tak, že najprv vyberie počet skupín, na ktoré sa majú dáta rozdeliť a náhodne sa vyberú zastupujúce prvky pre každú skupinu. Potom sa vypočíta vzdialenosť medzi každým prvkom dát a každým zastupujúcim prvkom. Prvky dát sa priradia do skupiny, ktorej zastupujúci prvok je najbližšie. Potom sa vypočítajú nové zastupujúce prvky pre každú skupinu ako priemer prvkov v skupine. Tento postup sa opakuje, kým sa neustáli poloha zastupujúcich prvkov a teda aj skupiny, alebo dokedy nenastane maximálny počet opakovaní. Segmentácia pomocou K-Means sa používa napríklad na rozdelenie obrazu na základe farby alebo na zoskupenie podobných zvukov v hudobnom zázname.[23]

V našej implementácií sme takúto segmentáciu vytvorili prostredníctvom funkcie *kmeans()*. Funkcia najprv oznámi hlášku do textového poľa o svojom priebehu a následne prebehne iterácia po snímkach videa. V každej tejto iterácií sú najprv získané informácie o rozlíšení obrázka a následne je obrázok transformovaný do jednorozmerného poľa. Následne je definovaný počet zhlukov, na ktoré sa bude obrázok segmentovať. Ďalej definujeme kritéria ukončenia algoritmu K-means clustering a nastavujeme spôsob inicializácie centroidov v algoritme. Ďalej vykonáva samotný algoritmus K-means clustering a priradzuje každý pixel do jedného zo hlukov, čoho výsledkom je pole *labels*, kde každý prvok označuje príslušnosť pixelu ku zhluku. Ďalej je toto pole transformované do rozmeru pôvodného obrázka a prebieha normalizácia hodôt na iterval **<0;255>**. Následne je aplikované farebné mapovanie *cv.COLORMAP_JET*. Vytvorený je mapovací tenzor o veľkosti (1,7) a vracané je pole týchto tenzorov.

Tento proces môžeme vidieť na výpise kódu 4.5. Výstup takto segmentovanej snímky môžeme vidieť na obr. 4.7 na ľavej časti obrázka a na pravej môžeme vidieť výstupný tenzor, zobrazený pomocou funkcie *display_images()*.

[9]typ strojového učenia, ktorý sa vyznačuje tým, že nevyžaduje anotované trénovacie dáta s preddefinovanými výstupmi

```python
def kmeans(self):
 self.textfield.setText("KMeans...")
 out = []
 for img in self.frames:
    w, h, d = img.shape
    image_array = img.reshape((w * h, d)).astype(np.float32)
    K = 7
    criteria = (cv.TERM_CRITERIA_EPS +
            cv.TERM_CRITERIA_MAX_ITER, 10, 1.0)
    flags = cv.KMEANS_RANDOM_CENTERS
    _, labels, _ = cv.kmeans(image_array, K, None,
                criteria, attempts=10, flags=flags)
    label_image = np.reshape(labels, (w, h))
    o = (label_image * 50).astype(np.uint8)
    o = cv.applyColorMap(o, cv.COLORMAP_JET)
    out.append(cv.resize(o, (1, 7)))
 return out
```

Výpis 4.5: Funkcia kmeans()

Obr. 4.7: Výstup segmentačnej funkcie na báze K-means clustering

4.3.6 Diskrétna kosinusová transformácia

Diskrétna kosínusová transformácia **DCT** je matematická transformácia, ktorá prevádza signál z časového do frekvenčného spektra. Podobne ako Fourierova transformácia, DCT rozkladá signál na jednotlivé frekvencie, avšak na rozdiel od Fourierovej transformácie, DCT pracuje s reálnymi číslami a je vhodnejšia na analýzu signálov, ktoré sú symetrické a obmedzené. DCT prekladá signál do koeficientov,

ktoré reprezentujú rôzne frekvencie signálu, pričom najnižšie frekvencie sú reprezentované prvými koeficientmi a najvyššie frekvencie sú reprezentované poslednými koeficientmi. DCT má rôzne varianty, napríklad DCT I a II, ktoré sa líšia výpočtom prvých koeficientov. Aplikácie DCT sú mnohé a zahrňujú zobrazovacie technológie, kompresiu dát, spracovanie zvukových a obrazových signálov a mnoho ďalších oblastí.[20]

V našej impletentácií sme vytvorili funkciu *dct()*, ktorá túto transformáciu vykonáva na jednotlivých snímkach. Najprv je do textového poľa GUI vypísaná hláška o priebehu funkcie a následne prebieha iterácia po snímkach videa. Počas tejto iterácie sú snímky prevedené na šedotónový obraz a následne je na nich vykonaná diskrétna kosinusová transformácia pomocou funkcie OpenCV *cv.dct()*. Výsledky tejto operácie sú uložené v poli *dct_array*. Ďalej je vypočítaný priemer pre všetky snímky v poli a výsledok je uložený do premennej *averaged*. Následne je toto pole rozdelené do troch sektorov pomocou premenných *upsum*, *midsum* a *lowsum*, ktorých jednotlivé riadky sú ščítané a vytvorené sú tak jednorozmerné polia, pričom sú následne normalizované na rozsah **<-1;1>**. Funkcia nevracia žiadnu hodnotu, no premenné sú uložené ako vlastné premenné triedy, a prístup k nim je realizovaný pomocou incializovanej triedy. Tento proces môžeme vidieť na výpise kódu 4.6.

```
def dct(self):
    self.textfield.setText("Performing DCT...")
    dct_array = []
    for i in self.frames:
        i = cv.cvtColor(i, cv.COLOR_RGB2GRAY)
        dct = cv.dct(np.float32(i))
        dct_array.append(dct)
    averaged = np.sum(dct_array, axis=0) / len(dct_array)
    upsum = np.sum(averaged[0:100], axis=0)
    midsum = np.sum(averaged[100:200], axis=0)
    lowsum = np.sum(averaged[200:255], axis=0)
    self.top_control = upsum / max([max(upsum),
                                    abs(min(upsum))])
    self.mid_control = midsum / max([max(midsum),
                                    abs(min(midsum))])
    self.low_control = lowsum / max([max(lowsum),
                                    abs(min(lowsum))])
```

Výpis 4.6: Funkcia kmeans()

4.4 Mapovanie dát z výstupného tenzora na hudobné parametre

Táto metóda syntézy zvuku z videa spočíva vo vytvorení mapovacieho tenzoru z každej segmentácie a následného namapovania tenzorov na hudobné parametre. Vytvorenie výstupných tenzorov pomocou rôznych typov segmentácií sme si ukázali v časti 4.3. V tejto časti sa pozrieme na mapovanie dát týchto výstupných mapovacích tenzorov na hudobné parametre, ktoré môžeme využiť pri syntéze zvuku.

Pri návrhu a implementácií tejto metódy sme rozhodli, že výstupný signál by mal byť založený na hudobných základoch a mal by obsahovať:

- Rytmickú zložku vo forme sady bicích nástrojov
- Melodickú a harmonickú zložku a to vo forme melodickej línie klavíra a klarinetu

4.5 Hudobné parametre

Kedže sme implementovali 5 rôznych druhov segmentácie, rozhodli sme pre namapovanie piatich rôznych hudobných parametrov. Jedná sa konkrétne o parametre:

- Rytmus
- Intervaly
- Oktáva
- Melodicko-harmonická linka klavíra
- Melodická linka klarinetu

Tieto parametre sme namapovali na jednotlivé segmentácie, a to konkrétne:

- Prahová segmentácia - rytmus
- Watershed segmentácia - intervaly
- Edge - oktávy
- K-means - melodická linka
- HMM - harmonicko-melodická linka

Pre tieto účely sme vytvorili triedu **Synthesis()**. Inicializačná funkcia tejto triedy má vstupné argumenty *textfield* a *inlist*, ktoré obsahujú referenciu na textové pole a vstupné parametre nastaviteľné v grafickom užívateľskom prostredí. V tejto incializačnej funkcií priradzujeme tieto vstupné dáta k vlastným premenným triedy a vytvárame polia potrebné pre mapovanie hudobných dát, a to konkrétne

polia *self.scale*, *self.intervals* a *self.drums*. Rozhodli sme sa pri syntéze využiť protokol MIDI, a tak sme do týchto polí priradili korešpondujúce hodnoty MIDI správ. Pole *self.drums* obsahuje 8 celočíselných hodnôt, ktoré reprezentujú MIDI správy pre jednotlivé prvky súpravy bicích nástrojov, a to konkrétne 36 - veľký bubon, 38 - malý bubon, 42 - hi-hat, 43- malý tom, 45 - stredný tom, 48 - veľký tom, 51 - ride cymbal a 57 - crash cymbal. Pole *self.scale* obsahuje hodnoty MIDI správ pre tóny základnej stupnice C dur, a to konkrétne 36 - tón C, 38 - tón D, 40 - tón E, 41 - tón F, 43 - tón G, 45 - tón A, 47 - tón H.

Pre priradenie týchto hodnôt a vytvorenie máp jednotlivých parametrov pre neskoršie vytvorenie MIDI správ, sme vytvorili funkcie *create_melody_map()*, *create_harmony_map()*, *create_rythm_map()*, *create_interval_map()* a *create_octave_map()*. Ďalej sa pozrieme na tieto funkcie bližšie.

4.5.1 Vytvorenie rytmickej mapy

Funkcia *create_rythm_map()* vytvára rytmickú mapu z výstupného tenzora segmentácie prahovania. Vo funkcií prebieha iterácia po prvkoch poľa tenzorov získaných pomocou algoritmu popísanom v časti 4.3.1. V prvom kroku je každý prvok konvertovaný z trojrozmerného priestoru na jednorozmerný pomocou konverzie na šedotónový farebný priestor. Ďalej nasleduje normalizácia na interval **<0;8>** a zmena dátoveho typu na celé číslo, kedže vyberáme z ôsmich prvkov množiny *self.drums* definovanej vyššie. Následne výstup normalizujeme na hodnoty v intervale **<0;127>**, kedže MIDI protokol podporuje navyššiu možnú hodnotu 127. Ďalej prebehne ďalšia iterácia vrámci tenzoru, ktorej výsledkom je vytvorenie prvku mapy v podobe [číslo MIDI noty, hodnota, index]. Pole s takto vytvorenými prvkami je priradené k vlastnej premennej triedy *self.rmap*.

4.5.2 Vytvorenie melodickej mapy

Funkcia *create_melody_map()* vytvára melodickú mapu z výstupného tenzora segmentácie K-means. Vo funkcií prebieha iterácia po prvkoch poľa tenzorov získaných pomocou algoritmu popísanom v časti 4.3.2. Ako aj pri rytmickej mape, je v prvom kroku každý prvok konvertovaný z trojrozmerného priestoru na jednodimenzionálny pomocou konverzie na šedotónový farebný priestor. Ďalej nasleduje normalizácia na interval **<0;7>** a zmena dátoveho typu na celé číslo, kedže teraz vyberáme zo siedmych prvkov množiny *self.scale* obsahjúcej tóny stupnice. Následne výstup znova normalizujeme na hodnoty v intervale **<0;127>**. Následne je vybraný

prvok s najväčšou hodnotou a prebehne ďalšia iterácia vrámci tenzoru, ktorej výsledkom je vytvorenie prvku mapy v podobe [číslo MIDI noty, hodnota, index]. Pole s takto vytvorenými prvkami je priradené k vlastnej premennej triedy *self.mmap*.

4.5.3 Vytvorenie harmonicko-melodickej mapy

Funkcia *create_harmony_map()* vytvára melodickú mapu z výstupného tenzora segmentácie pomocou HMM. Vo funkcií prebieha iterácia po prvkoch poľa tenzorov získaných pomocou algoritmu popísanom v časti 4.3.4. Každý prvok je konvertovaný z trojdimenzionálneho priestora na jednodimenzionálny pomocou konverzie na šedotónový farebný priestor. Ďalej nasleduje normalizácia na interval **<0;7>** a zmena dátoveho typu na celé číslo, kedže znova vyberáme zo siedmych prvkov množiny *self.scale*. Výstup znova normalizujeme na hodnoty v intervale **<0;127>**. Následne je vybraný prvok s najväčšou hodnotou a prebehne ďalšia iterácia vrámci tenzoru, ktorej výsledkom je vytvorenie prvku mapy v podobe [číslo MIDI noty, hodnota, index]. Pole s takto vytvorenými prvkami je priradené k vlastnej premennej triedy *self.hmap*.

4.5.4 Vytvorenie oktávovej mapy

Funkcia *create_octave_map()* vytvára mapu oktávových hodnôt z výstupného tenzora segmentácie pomocou detekcie hrán. Vo funkcií prebieha iterácia po prvkoch poľa tenzorov získaných pomocou algoritmu popísanom v časti 4.3.3. Znova je každý prvok konvertovaný do jednodimenzionálneho priestoru pomocou konverzie na šedotónový farebný priestor. Ďalej nasleduje normalizácia na interval **<0;7>** a zmena dátoveho typu na celé číslo, kedže oktávový modifikátor budeme priradzovať jednotlivým tónom. Následne prebehne ďalšia iterácia vrámci tenzoru, ktorej výsledkom je vytvorenie prvku mapy v podobe [číslo oktávy, index]. Pole s takto vytvorenými prvkami je priradené k vlastnej premennej triedy *self.omap*.

4.5.5 Vytvorenie intervalovej mapy

Funkcia *create_interval_map()* vytvára intervalovú mapu z výstupného tenzora segmentácie watershed. Vo funkcií prebieha iterácia po prvkoch poľa tenzorov získaných pomocou algoritmu popísanom v časti 4.3.2. Každý prvok je konvertovaný na jednodimenzionálny priestor pomocou konverzie na šedotónový priestor. Ďalej nasleduje normalizácia na interval **<0;7>** a zmena dátoveho typu na celé číslo, kedže

vyberáme zo siedmych prvkov množiny *self.intervals*. Výstup je normalizovaný na hodnoty v intervale **<0;127>**. Ďalej prebehne ďalšia iterácia vrámci tenzoru, ktorej výsledkom je vytvorenie prvku mapy v podobe [číslo intervalu, hodnota, index]. Pole s takto vytvorenými prvkami je priradené k vlastnej premennej triedy *self.imap*.

4.6 MIDI

MIDI (Musical Instrument Digital Interface) je protokol pre prenos hudobných dát medzi digitálnymi zariadeniami. Bol vytvorený v roku 1982 a umožňuje komunikáciu medzi rôznymi hudobnými nástrojmi, počítačmi a ďalšími zariadeniami. MIDI dáta sa skladajú z rôznych typov správ, ako sú notové správy, kontrolné správy a systémové správy. Notové správy obsahujú informácie o notách a ich parametroch, akými sú dĺžka, hlasitosť a tón. Kontrolné správy umožňujú ovládanie rôznych parametrov hudobných nástrojov, ako sú napríklad hlasitosť, tón alebo efekty. Systémové správy sú určené na kontrolu a synchronizáciu MIDI zariadení. Vďaka MIDI protokolu je možné vytvárať, upravovať a prehrávať hudbu pomocou počítača alebo iných zariadení. MIDI je využívané v rôznych oblastiach, ako nahrávanie, produkcia, výuka hudby, tiež sa dá použiť na riadenie osvetlenia a vizuálnych efektov na koncertoch či v divadelných predstaveniach. MIDI je stále populárne a používané v súčasnosti vďaka svojej univerzálnosti, jednoduchosti použitia a flexibility.[24]

Pre prácu s MIDI v našej implementácií využívame knižnicu *mido*. Táto knižnica ponúka vytvorenie MIDI súboru s možnosťou viacerých stôp a MIDI správ pomocou správ Mido, objektov jazyka Python s metódami a atribútmi, ktoré sa líšia v závislosti od typu správy. Vytvorené mapy z časti 4.4 sme pomocou týchto správ použili pri vytvorení takéhoto MIDI objektu, a následnom zápise MIDI súboru na disk. Tento proces je vykonávaný vo funkcií triedy **Synthesis()** *create_midi_file()*.

V prvom kroku je vykonaná interpolácia kontrolných signálov, ktorých tvorba je popísaná v časti 4.3.6. Pre interpoláciu sme vytvorili funkciu *sinc_interpolation()*, ktorá vykonáva doplnenie vzoriek nášho kontrolného signálu na dĺžku polí máp. Túto funkciu môžeme vidieť na výpise 4.7.

```python
def sinc_interpolation(signal, new_length, threshold):
    old_length = len(signal)
    t_old = np.linspace(0, 1, old_length)
    t_new = np.linspace(0, 1, new_length)
    kernel = np.sinc((t_new[:, None] - t_old[None, :])
                                        * old_length)
    interpolated = np.dot(kernel, signal)
    interpolated = (interpolated + 1) / 2
    return (interpolated >= threshold).astype(int)
```

Výpis 4.7: Funkcia sinc_interpolation()

Táto interpolácia vráti kontrolný signál prahovaný na hodnoty 1 alebo 0. Následne je aplikovaná funkcia *set_min_distance()*, ktorej parametre sú kontrolované užívateľom v grafickom užívateľskom prostredí. Tieto parametre určujú najmenšiu vzdialenosť medzi jednotlivými prvkami s hodnotou 1 v riadiacom signále. Tento proces má vplyv na výsledné množstvo MIDI správ. Túto funkciu môžeme vidieť na výpise 4.8.

```python
def set_min_distance(arr, mindist):
    prev_idx = -mindist - 1
    for i in range(len(arr)):
        if arr[i] == 1:
            if i - prev_idx < mindist:
                arr[i] = 0
            else:
                prev_idx = i
    return arr
```

Výpis 4.8: Funkcia set_min_distance()

Ďalej funkcia vytvorí objekt knižnice mido *MidiFile*, priradí mu stopy bicích *drum_track*, klavíra *piano_track* a klarinetu *clarinet_track*. Následne je každej stope priradená MIDI správa typu **Program Change**, pomocou ktorej nastavujeme priradený zvuk. Pre *drum_track* a *piano_track* ostáva hodnota 0 (rozlíšenie pre biciu súpravu vykonáme neskôr pomocou zmeny MIDI kanálu, a to konkrétne na kanál 9) a pre *clarinet_track* je priradená hodnota 73. Jednotlivé hodnoty reprezentujú rôzne nástroje štandardizované v MIDI[10]. Následne je nastavené tempo vo

[10] https://www.recordingblogs.com/wiki/midi-program-change-message

formáte BPM[11] a je priradené MIDI objektu pomocou správy *set_tempo*.

V ďalšom kroku prebehnú tri cykly for, ktoré objektu priradzujú MIDI správy podľa vytvorených máp parametrov. Prvý cyklus priradzuje stope bicích nástrojov správu **Note On** na kanáli 9, ktorý slúži pre správy súpravy bicích nástrojov. Parametru **note** je priradený prvok z poľa *self.rmap*, **velocity** je vypočítaný ako druhý prvok poľa *self.rmap* vynásobený kontrolnou krivkou *mid_control*. Posledným parametrom je **time**, ktorému je priradený posledný prvok poľa *self.rmap*. Prvky poľa self.rmap tak využívame ako [hodnota noty, velocity, time]. V ďalšom cykle postupujeme obdobne, no tentokrát s parametrom *self.hmap*, ktorý priradzujeme klavíru. Nota je znovu priradená z prvého prvku poľa *self.hmap*, pričom je prirátaný oktávový modifikátor z poľa *self.omap*, **velocity** z druhého prvku vynásobeného kontrolným signálom *top_control* a *time* je priradený podľa posledného prvku tohoto poľa. Znova pole využívame vo forme [hodnota noty, velocity, time]. V poslednom cykle vytvárame MIDI správy pre melodickú líniu klarinetovej linky, pričom pre parameter **note** je použitý prvý prvok poľa *self.mmap*, ku ktorému je pričítaný prvý prvok poľa intervalov. Ďalej je prvku **velocity** pridaný druhý prvok poľa *self.mmap*, vynásobený kontrolnou krivkou *low_control* a rovanko je pridaný tretí prvok poľa parametru **time**.

Následne dokážeme tento objekt MIDI uložiť ako súbor na disk, pomocou vstavanej metódy MIDI objektu *save()*. Funkcia taktiež vracia tento MIDI objekt *midi_file*, ktorý sa ďalej priradzuje funkcií pre uloženie súboru na disk v grafickom užívateľskom prostredí. Výsledný súbor si užívateľ môže prehrať v rôznych DAW[12], MIDI syntetizátoroch alebo prehrávacích programov ako VLC[13], QuickTime Player, Windows Media Player a podobne.

Celý proces syntézy ovláda funkcia *Segmentor()*. Táto funkcia vytvára objekt triedy **VideoAnalyzer()** a vykonáva všetky druhy segmentácie, následne využije funkciu *the_sum()*, ktorej vstupný parameter má užívateľ možnosť ovládať v grafickom užívateľskom prostredí, a to konkrétne parameter *average*. Táto funkcia znižuje počet snímok, respektíve prvkov polí segmentovaných snímok podľa čísla zadaného užívateľom. Túto funkciu môžeme vidieť na výpise 4.9.

[11] Beats per minute - počet úderov alebo štvrťových nôt za minútu

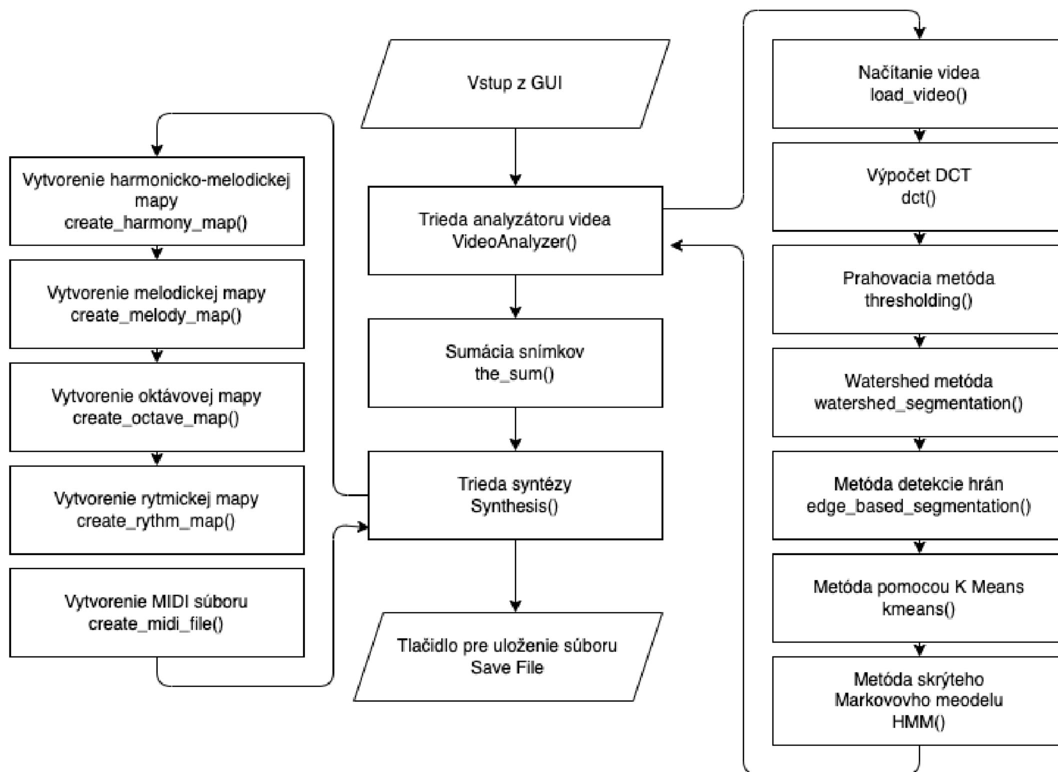
[12] Digital Audio Workstation - softvérová aplikácia, ktorá umožňuje nahrávanie, editáciu, mixovanie a produkciu hudby a zvuku. DAW sa používa na profesionálne nahrávanie hudobných skladieb, tvorbu zvukových efektov

[13] `https://www.videolan.org/vlc/`

```
def the_sum(arr, interval):
    summed = [[], [], [], [], []]
    a = 0
    for i in range(len(arr)):
        j=0
        while j < len(arr[i]):
            for x in range(interval+1):
                try:
                    a += arr[i][j+x]
                except:
                    pass
            summed[i].append(a)
            a = 0
            j += interval
    return summed
```

Výpis 4.9: Funkcia the_sum()

Následne je vytovrený objekt triedy **Synthesis()** a vypočítané sú mapy hudobných parametrov. Funkcia vracia objekt MIDI, ktorý je ďalej využitý vo funkcií pre uloženie súboru na disk. Celý tento proces môžeme vidieť na vývojovom diagrame na obrázku 4.8.



Obr. 4.8: Vývojový diagram funkcie Segmentor()

4.7 Rýchlosť metódy

Rýchlosť jednotlivých procesov sme merali pomocou knižnice *time*. Výsledky merania pre video o dĺžke 17 *s* môžeme vidieť v tabuľke 4.1.

Ako môžeme vidieť, najviac času zaberajú metódy HMM a K-means, a to skoro rovnaký čas ako samotné video. Výsledný čas je 36 *s*, metódu tak nieje možné používať v reálnom čase pre generovanie MIDI nôt z videa. Najrýchlešie sú funkcie pre vytvorenie máp hudobných parametrov, a spomedzi funkcií segmentácií je najrýchlejšia metóda prahovania thresholding.

Zrýchlenie metódy môžeme dosiahnuť zrýchlením čiastkových procesov, a to hlavne optimalizáciou segmentácie K-Means a HMM. Kedže naše testovacie video obsahuje 17 sekúnd záznamu pri FPS = 24, celkový počet snímkov je 408. V tom prípade nám vychádza čas sprocesovania jedného snímku 0,088 *s*. Aby sme dosialhli periódu T = 0,04 potrebujeme rýchlosť metódy viac ako zdvojnásobiť. To môžeme dosiahnúť jedine optimalizáciou segmentačných metód K-Means a HMM, no to bude mať za efekt zníženie procesnej kvality daných segmentačných metód.

Tab. 4.1: Namerané hodnoty pre jednotlivé implementované funkcie

Funkcia [-]	Priemerný čas algoritmu[s]
1 /FPS	0,04
DCT	0.318
thresholding	0.200
Watershed	1.274
Edge	0.246
K-Means	16.232
HMM	16.624
Harmony Map	0.012
Interval Map	0.011
Melody Map	0.015
Octave Map	0.009
Rythm Map	0.012

4.8 Zhodnotenie a možné vylepšenia

Aplikácia SegMentor je založená na využití segmentačných algoritmov obrazu, ktorých výsupy sú následne využité pri syntéze zvuku vo forme MIDI súboru. Táto aplikácia nedokáže syntetizovať zvuk v reálnom čase, a tak je použiteľná iba na existujúce video súbory, z ktorých je zvuk syntetizovaný a uložený na disk vo forme súboru MIDI. Výstup sa dá klasifikovať, podobne ako výstup aplikácie *VSyntha*, ako hudobnú kompozíciu pripomínajúcu moderný alebo avantgardný štýl kompozície.

Pri zhodnocovaní tejto aplikácie sme zvážili aj možné vylepšenia tejto aplikácie, respektíve tejto metódy, a to konkrétne:

- **Zrýchlenie procesov segmentácie** obrazu za účelom vytvoriť možnosť aplikáciu používať v reálnom čase pomocou vstupu pripojenej webkamery alebo videokamery. To môžeme realizovať zjednodušením výpočetnej náročnosti jednotlivých algoritmov, čo ale bude mať dopad na kvalitu procesu segmentácie, prepisom kódu do rychlejších jazykov ako C alebo C++ alebo zmenou metód segmentácie, a to hlavne metódy K-Means a HMM, ktorých nahradením za rychlejšie varianty dokážeme razantne zrýchliť proces syntézy.
- **Pridanie zvukového prehrávača MIDI súboru** - prehrávač, ktorý by prehrával vygenerované MIDI súbory, buď to pomocou vstavanej MIDI knižnice systému, alebo s možnosťou výberu vlastnej zvukovej banky. To môžeme do-

siahnúť napríklad pomocou knižníc *pygame*, *python-rtmidi*, *FluidSynth* a podobne.

- **Pridanie možnosti voľby počtu stôp a nástrojov** - voľba nástrojov podľa užívateľa alebo pomocou analýzy obrazu, ktorá by nastavovala parametere **Program Change**, a obdobným spôsobom ovplyvňovať množstvo jednotlivých MIDI stôp a teda počtu nástrojov.

Záver

Cieľom tejto diplomovej práce bolo vytvoriť rešerš metód syntézy zvuku z obrazových a video dát na dáta zvukové a implementácia troch, nami vytvorených metód takejto syntézy.

V prvej časti práce je vyhotovená rešerš, kde sú popísané rôzne práce, články a aplikácie, ktoré sa venujú podobnej tématike. Táto rešerš poskytuje prehľad o súčasnom stave výskumu a praxi v oblasti tvorby zvuku z obrazu. V rámci nej je opísaných pätnásť rôznych riešení, ktoré sa zameriavajú na túto problematiku, ktoré sa líšia vo svojom prístupe a technikách použitých na tvorbu zvuku z obrazu. Cieľom tejto časti práce je poskytnúť komplexný prehľad o existujúcich prístupoch k tvorbe zvuku z obrazu a identifikovať ich výhody, obmedzenia a prípadné možnosti rozšírenia. Tieto informácie následne slúžia ako základ pre ďalšie experimenty a vývoj nových metód a aplikácií v tejto oblasti.

V druhej časti práce je popísaná implementácia našej prvej metódy syntézy zvuku z video dát. Jedná sa o aplikáciu s názvom *VSyntha*, ktorá v reálnom čase prehráva video a syntetizuje z neho zvuk. Aplikácia funguje, podobne ako práce spomínané v prvej časti práci, na mapovaní video dát na dáta zvukové. Aplikácia tvorí z dát videa hudobnú kompozíciu s možnosťou ovládať harmonické parametre tonality a modulácie. Aplikácia sa ovláda prostredníctvom grafického užívateľského prostredia, s možnosťou prehrávania syntetizovaného videa, ako aj vstupu webkamery alebo videokamery.

V tretej časti tejto práce je popísaná implementácia našej druhej metódy syntézy zvuku z dát videa. Aplikácia s názvom *ReAmper* je založená na princípe soundscapingu - tvorby zvuku scény, zvukovej atmosféry. Táto aplikácia využíva detekciu a sledovanie objektov sćeny pomocou kombinácie algoritmov DeepSORT a YOLO. Získané dáta využíva na syntetizáciu a umiestňovanie zvukov objektov zo zvukovej banky do scény, pričom je využívaná technika HRTF. Pri tomto ozvučovaní scény má užívateľ na výber medzi použitím dvoch typov zvukových bánk - zvuky objektov a hudobné tóny. Túto metódu syntézy kvôli rýchlosti nieje možné použiť v reálnom čase.

Vo štvrtej časti tejto práce je popísaná implementácia našej tretej metódy syntézy zvuku z video dát, a to vo forme aplikácie *SegMentor*. Táto aplikácia využíva rôzne druhy segmentácie obrazu pre generovanie mapovacích tenzorov pre hudobné parametere, ktoré sú následne využité pri tvorbe MIDI súboru. Aplikácia využíva prahovú segmentáciu, watershed segmentáciu, segmentáciu na základe detekcie hrán, segmentáciu pomocou skrytého Markovovho modelu a segmentáciu na základe algo-

ritmu K-means clustering. Vytvorené sú tri stopy bicej súpravy, klavíra a klarinetu, ktorých hudobné dáta generujú segmentované obrazy. Výstup je MIDI súbor, ktorý je možno naimportovať do rôznych aplikácií alebo prístrojov pre prácu s MIDI.

Na základe vykonanej práce a implementácie troch metód syntézy zvuku z obrazových a video dát sme dosiahli nasledujúce výsledky. Naša prvá metóda, aplikácia *VSyntha*, úspešne prehráva video a generuje z neho zvukovú kompozíciu s možnosťou ovládania rôznych hudobných parametrov. Táto aplikácia poskytuje jednoduchý a intuitívny spôsob tvorby zvuku z obrazu a môže mať široké využitie. Druhá metóda, aplikácia *ReAmper*, sa zameriava na ozvučovanie scén pomocou zvukových objektov a hudobných tónov. Využíva detekciu objektov v obraze a ich následnú syntézu a umiestnenie do zvukovej scény. Táto metóda prináša možnosti vytvárania zvukovej atmosféry a zvukových efektov v interaktívnom prostredí. Naša tretia metóda, aplikácia *SegMentor*, využíva rôzne techniky segmentácie obrazu pre generovanie mapovacích tenzorov pre hudobné parametre. Táto metóda umožňuje vytváranie komplexných MIDI súborov z video dát, čo otvára možnosti pre ich ďalšie spracovanie a úpravu v rôznych hudobných aplikáciách. Všetky tri implementované metódy predstavujú prínos v oblasti syntézy zvuku z obrazových a video dát. Poskytujú nové nástroje a možnosti pre tvorcov zvuku a multimediálnych umelcov a otvárajú priestor pre ďalší výskum a vývoj v tejto oblasti. Súčasne vytvorené aplikácie predstavujú užitočné nástroje pre tvorbu zvukového obsahu a umožňujú interakciu s vizuálnymi dátami vo forme zvuku.

Výsledky tejto práce nám poskytujú pohľad o súčasnom stave výskumu a praxe v oblasti syntézy zvuku z obrazových a video dát a ponúkajú možnosti pre ďalší rozvoj v tejto oblasti.

Literatúra

[1] DANNENBERG, R.B.; NEUENDORFFER, T.: Sound Synthesis from Real-Time Video Images, School of Computer Science, Carnegie Mellon University. Dostupné na: https://www.cs.cmu.edu/~rbd/papers/videosound-icmc2003.pdf

[2] THORNÉ, C.: Filtering Video Noise as Audio with Motion Detection to Form a Musical Instrument (2016). Dostupné na: https://www.researchgate.net/publication/301842969_Filtering_Video_Noise_as_Audio_with_Motion_Detection_to_Form_a_Musical_Instrument

[3] ZAVŘEL, Petr. Vytváření zvuků z obrazových dat. Brno, 2019, 70 s. Diplomová práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací. Dostupné na: https://www.vut.cz/www_base/zav_prace_soubor_verejne.php?file_id=129481

[4] BRAGARD, V; PELLEGRINI, T; PINQUIER, J.: Pyc2Sound: a Python tool to convert images into sound, Universite de Toulouse, IRIT (2015). Dostupné na:https://www.researchgate.net/publication/301463518_Pyc2Sound_a_Python_tool_to_convert_imags_into_sound

[5] YEO, W. S.; BERGER, J.:Application of Image Sonification Methods to Music, Stanford University, Center for Computer Research in Music and Acoustics (2005). Dostupné na: https://www.researchgate.net/publication/239416537_Application_of_Image_Sonification_Methods_to_Music

[6] JOO, W.: SONIFYD: A GRAPHICAL APPROACH FOR SOUND SYNTHESIS AND SYNESTHETIC VISUAL EXPRESSION, Virginia Polytechnic Institute and State University (2019) . Dostupné na: https://smartech.gatech.edu/bitstream/handle/1853/61519/icad2019_045.pdf

[7] YEO, W. S.; BERGER, J.: Raster Scanning: A New Approach to Image Sonification, Sound Visualization, Sound Analysis And Synthesis, Department of Music, Stanford University. Dostupné na: https://ccrma.stanford.edu/~woony/publications/Yeo_Berger-ICMC06.pdf

[8] ROUZIC, M.: Photosounder: User Guide. Dostupné na: https://photosounder.com/documentation.php

[9] Sound Art Team:The 'Variophone' Yevgeny Sholpo. Russia, 1932,Dostupné na: https://soundart.zone/the-variophone-yevgeny-sholpo-russia-1932/

[10] KREICHI S.: The ANS Synthesizer, Composing on a Photoelectronic Instrument . Dostupné na: https://www.theremin.ru/archive/ans.htm

[11] FRY P.; Sarah COX S.: Student builds Daphne Oram's unfinished 'Mini-Oramics' . Dostupné na: https://www.gold.ac.uk/news/mini-oramics/

[12] UI Software: Metasynth . Dostupné na: https://uisoftware.com/MetaSynth/

[13] FOURNEL, N.: AUDIOPAINT . Dostupné na: http://www.nicolasfournel.com/?page_id=125

[14] EKMAN, R.: Coagula Industrial Strength Color-Note Organ . Dostupné na: https://www.abc.se/~re/Coagula/Coagula.html

[15] Kaleidoscope, 2caudio . Dostupné na: https://www.2caudio.com/products/kaleidoscope#_overview

[16] PyQT5 Documentation, Qt . Dostupné na: https://doc.qt.io/qtforpython/

[17] PYO Documentation, Ajax Sound Studio . Dostupné na: http://ajaxsoundstudio.com/pyodoc/

[18] OpenCV Documentation, OpenCV . Dostupné na: https://docs.opencv.org/4.x/d6/d00/tutorial_py_root.html

[19] NumPy Documentation, Numpy . Dostupné na: https://numpy.org/doc/stable/

[20] R.C. GONZALEZ, R.E. WOODS.: Digital Image Processing, Prentice Hall, 2007.

[21] ROERDINK, J. B., MEIJSTER, A.: The watershed transform: definitions, algorithms and parallelization strategies. Fundamenta Informaticae, 41(1-2), 187-228. 2000

[22] JURAFSKY, D., MARTIN, J. H.: Speech and Language Processing (3rd ed.). 2019

[23] T. HASTIE, R. TIBSHIRANI, J. FRIEDMAN.: The Elements of Statistical Learning: Data Mining, Inference, and Prediction, 2nd edition, Springer, 2009

[24] HOLMES, T., HOLMES, T.: MIDI Demystified (3rd ed.). Hal Leonard Corporation. 2018

Zoznam symbolov a skratiek

ASR Obálka typu Attack-Sustain-Release
RGB farebný priestor Red - Green - Blue
HSV Farebný priestor Hue - Saturation - Value
HSL Farebný priestor Hue - Saturation - Level
YCbCr Farebný priestor YCbCr
API Application programming interface - rozhranie pre programovanie aplikácií
DCT Diskrétna kosinusová transformácia
HMM Hidden Markov Model - skrýtý Markovov model

Zoznam príloh

A Obsah elektronickej prílohy **85**

A Obsah elektronickej prílohy

V elektronickej prílohe sú priložené súbory so zdrojovými kódmi implementovaných aplikácií *Vsyntha*, *ReAmper* a *SegMentor*. Zložka aplikácie *ReAmper* prekračovala veľkostný limit elektronickej prílohy, preto je v zložke umiestnený súbor s odkazom na stiahnutie dodatočných súborov.

Priložený je aj súbor prostredia *Anaconda* `environment.yml` s exportovanými verziami Pythonu a všetkými použitými knižnicami a `.pdf` súbor tejto práce.

- `.1 /` ..... koreňový adresár priloženého archívu
  - `VSyntha` ..... Zložka implementácie prvej metódy
    - `main.py` ..... Hlavný súbor aplikácie obsahujúci GUI a pracovné vlákno
    - `m_1.py` ..... Súbor metódy syntézy, obsahujúci výpočet parametrov a generátor zvuku
  - `ReAmper` ..... Zložka implementácie druhej metódy
    - `main.py` ..... Hlavný súbor aplikácie obsahujúci GUI a pracovné vlákno
    - `sound_process.py` ..... Súbor obhsahujúci triedy a funkcie pre spracovanie zvuku
    - `my_object_tracker.py` ..... Súbor obsahujúci funkciu pre detekciu objektov
    - `link.rtf` ..... Súbor s odkazom na dodatočné súbory pre aplikáciu
  - `SegMentor` ..... Zložka implementácie tretej metódy
    - `main.py` ..... Hlavný súbor aplikácie obsahujúci GUI, triedy a funkcie
  - `environment.yml` ..... Exportované prostredie Anaconda
  - `diplomová_práca.pdf` ..... PDF súbor diplomovej práce