Automated Visual Mixing System

A thesis submitted to University College Cork
in partial fulfilment of the requirements for the degree of
Master of Science in Interactive Media

October 2019

Conor Muldowney
School of Computer Science & Information Technology

Contents

List of Tables

List of Figures

Abstract

This thesis presents a novel approach to record and mix audiovisual streams based on audio signal analysis. The associated project implements signal processing techniques in order to selectively display a desired video feed based on voice activity detection, providing the user with the ability to record and broadcast two audio visual streams without the need for human resources to switch between feeds. The proposed system is a web based application, making it accessible to a wide range of users without the need to download or purchase specific hard/software. It was written in JavaScript/Node.js and utilises the MediaDevices, WebAudio, MediaRecorder and MediaSource APIs in conjunction with Socket.IO to capture, process, record and deliver streams of audio visual segments to a broadcast page. The challenging aspects of the project were isolating the characteristics of an audio signal similar to those of a human voice and providing a system capable of broadcasting live content to a large audience. This thesis illustrates the details of the aforementioned challenges and solutions.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Signed:
Conor Muldowney

Acknowledgements

Many thanks to Dr. John O' Mullane, Pat Muldowney and Colette Muldowney.
Special thanks to Neil O Carroll, Eamon Muldowney, James Madden and Ariel Travis.

Chapter 1

Introduction

1.1 Background

Modern media service providers such as YouTube and Twitch provide millions of content creators with a means to record and broadcast to a global audience. This results in the emergence of podcasting and visual radio, which are becoming increasingly popular due to the subscribers ability to view or listen to content on demand, regardless of geographical location. Statistics show that 37% of the Irish population are monthly listeners.[16] From an artistic standpoint, perhaps the most appealing aspect to content creators is that they have minimal restrictions and constraints. From a business and marketing position, podcasts are both easily accessible and cheap, while the content creator stands to earn an income from advertising. In the US alone, advertising revenue from podcasts is estimated to reach 659 million dollars by 2020. This equates to a 1000 per cent increase over the last five years.[27]

At present, using studio editing software in conjunction with online video hosting platforms not only make it easy to create and broadcast to a massive audience, but also can provide professional looking/sounding content.[4] However, the majority of podcasts are comprised of only utilising an audio element. The removal of a video component reduces the human resources necessary. Nevertheless, utilising a video component in dialogue-driven content, such as interviews, provides huge benefits for listeners. Currently, the majority of podcasts only offer a transcript for listeners with impaired hearing. The implementation of a video component captures aspects such as the expressed emotion between speakers, nuances in conversation, facial expressions and body language. By providing content creators with a cheap and efficient means to produce material in a live setting, the proposed system enables users to create a live broadcast with minimal requirements for human resources such as a camera/desk operators.

1.2 Target User and Proposed Solution

The proposed system is aimed to facilitate a user with the desire to record and broadcast an audiovisual representation of a conversation between two speakers with minimal human resource requirements.

1.2.1 Proposed System

The proposed system should:

- Be easily accessible to a user with minimal hard/software requirements.
- Be capable of taking two audiovisual input feeds and broadcast a mixed output feed to a remote webpage.
- Be capable of automatically switching between camera feeds according to a specific set of rules.
- Perform without interaction while recording is in process.
- Produce an output similarly equivocal to the quality produced by professional camera operator crew.

Chapter 2

Analysis

The previous chapter describes the current state of online video broadcasting. It outlines the requirements for a cost efficient, automated camera switching system and current limitations in the existing technology. This chapter illustrates the current available solutions to producing live and pre-recorded content. This chapter also demonstrates how signal processing techniques are used to implement an automated camera switching solution.

2.1 Smart Video Editing Software

Currently, frame-based editing tools such as Final-Cut and the Adobe Premiere software, are the popular solution when editing content. Though this method of editing may be tedious and requires a certain amount of experience, it produces the most desirable results for most productions. The aesthetic quality of the resulting composition is subjective and therefore requires an element of human influence to achieve certain styles. To reduce certain menial tasks in the editing process, a number of solutions have been created which lessen the amount of human resources necessary. This section compares existing technologies for pre-recorded and live audiovisual mixing solutions.

2.1.1 Pre-Recorded Editing Solutions

Aside from the frame-based editing tools, current solutions to aid in the pre-recorded video editing process operate on the basis of retroactively analysing the audio/video/transcript components, then extracting and utilising the pertinent information to decide on an edit. These approaches are of either fully or semi-automatic nature, depending on the systems level of control over the final edit. Semi-automatic software produces a number of edits to choose from based on labelling and segmenting the input data, utilising various scene cut detection techniques, then outputting multiple choices for the final edit.[20][32][11] Solutions such as IMPACT take a video input, analyse motion within the scene and generate a structural description of the video. The user is then provided with a means to organise and edit by manipulating the categorised shots. The Hitchcock solution similarly utilises motion detection to determine the

suitability of a video segment for consideration in the edit. The suitability of each clip is provided to the user as a numerical representation, to aid in categorising and deciding clip length and positioning. The video sequence is organised in a hierarchical view, providing grouped segments of similar nature and thus reducing the task load of the editor.[29]

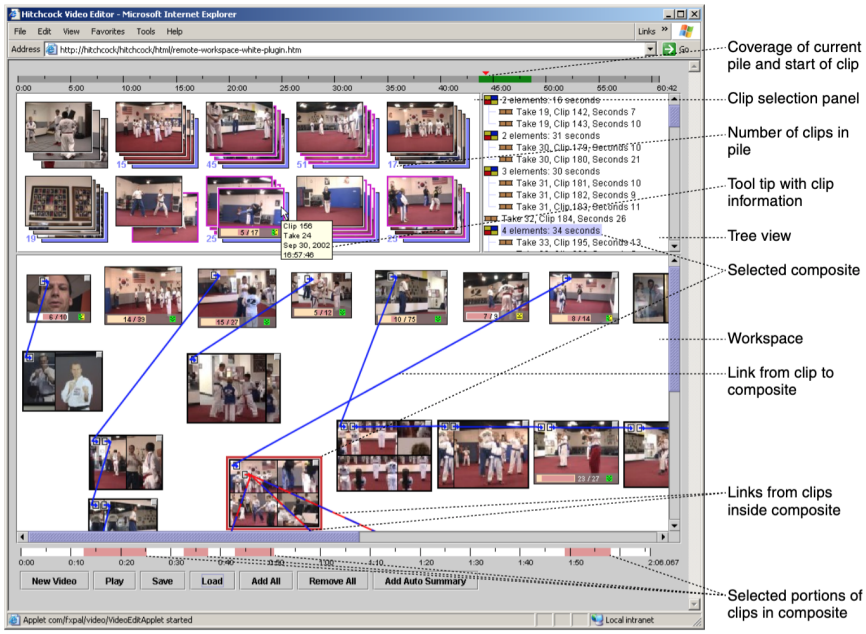

Figure 2.1: Screenshot of Hyper-Hitchcock software[29]

More modern approaches use statistical models to create multiple edits based on a transcript of the dialogue in the scene.[20] Mackenzie's statistical model decides clip selection based on various common film editing idioms in order to convey the narrative in a particular style. Film-

Idiom Name	Description
Avoid jump cuts	Avoid transitions between clips that show the same visible speakers to prevent jarring transitions.
Change zoom gradually	Avoid large changes in zoom level that can disorient viewers and instead change zoom levels slowly.
Emphasize character	Do not cut away from shots that focus on an important character unless another character has a long line. This focuses the audience's attention on the more important character.
Intensify emotion	Use close ups for particularly emotional lines to provide more detail in the performer's face.
Mirror position	Select clips for one performer that most closely mirror the screen position of the other performer to create a back-and-forth dynamic for a two person conversation.
Peaks and valleys	Zoom in for more emotional lines and zoom out for less emotional lines to allow the audience to see more detail in the performer' faces for emotional lines.

Figure 2.2: List of common filming idioms[20]

makers typically combine idioms such as those seen in figure 2.2, to create different styles. Compared to the previously mentioned methods, Mackenzie's technique provides the user with a wide variety of stylised options with minimal effort. The editing process is modelled using a Hidden Markov Model. Using this approach, the system is capable of combining idioms by arithmetic operations. As a result, the software creates an intelligent and stylised edit.

Each of the systems mentioned above are considered to be semi-automatic, in that they

require an amount of human interaction within the decision process while still reducing the number of human resources required to perform the menial tasks. In a fully automatic approach, the system has absolute control over the editing process, allowing little to no user influence on the operation.[30][8] Commercial products such as *Muvee*, *Pond5*, *LIFE.FILM* and *Filmora* are used mainly for social media marketing and slideshows but have applications in creative productions also. These solutions utilise machine learning algorithms to cut and transition between inputted media. This solution makes sense for certain scenarios however the editors abilities to personalise edits are reduced.

A fully automatic system is the sought after solution for news on demand and interview segments, as typically, there is only one output format and thus has no need for a human element to influence the systems editing process.

2.1.2 Live Solutions

The retroactive approach produces a much smoother and concise edit compared to a live solution. The main reason being that the analysis of existing footage can account for the occurrence of unexpected events in the environment during recording and, as a result, disregard instances of undesirable audio/video when producing the final composition. In contrast, it is impossible to fully anticipate an unwanted noise, gesture or event in a live setting.

Various techniques to improve speaker detection and prediction show promising results through the analysis of a persons gaze. This method is effective when more than two subjects are being recorded. Research shows that the gaze direction of subjects can indicate to where their focus of attention lies.[15][31][33] Gaze direction can be approximated by tracking head orientation. Developing on this hypothesis, systems implement head tracking software to estimate a popular gaze target. However, the system can produce false positives or a series of quick and unexpected cuts. For example, attention can shift at a fast pace during a heated debate resulting in undesirable cuts between camera feeds. Issues also occur in two person dialogue driven scenarios. Both subjects typically have a fixed gaze on each other making it difficult for the system to discern who deserves the focus of attention. In such a scenario, a possible use case for this technology would be to switch to external media being referenced in the dialogue, for example a whiteboard or YouTube clip. If the speakers both shift their gaze to a particular object, the system could recognise it as the popular gaze target and make an intelligent switch.

To estimate where the focus of attention lies in a two speaker scenario, this study proposes the implementation of voice activity detection to calculate which speaker deserves the focus of attention. A number of systems utilise speaker detection algorithms to determine a subject of interest.[14][21][28] *vMix*, *Wirecast* and *vidBlaster* each provide solutions for live automated vision mixing by using utilising various input signal analysis techniques. While analysis of a video input can provide the system with useful information such as body movement and gaze tracking, the most logical method for discriminating between speakers is to analyse audio input. By determining which speaker is articulating significant speech, the system can determine who currently deserves the focus of attention.

2.2 Voice Activity Detection

Methods for determining the presence of speech in audio signals have been well explored.[9][7][13] Voice activity detection (VAD) software is essential to a number of applications including speech recognition and speech coding. The main purpose of VAD technology is to distinguish between human speech and background noise. This is accomplished by analysing the acoustic signal and locating attributes that are unique to human speech. Conversely, noise by definition is non specific and occurs in an unexpected manner. As a result, it has no uniquely defining characteristics and cannot be determined as easily as speech. In order to accurately recognise the attributes unique to the human voice, it is important to understand how we articulate sounds.

A sound is produced through a process known as phonation. A column of air pressure passes through the vocal folds causing a sequence of vibratory cycles to occur. These cycles produce resonance frequencies, which form the basis of a voiced sound. The frequency of vibrations of the vocal folds while generating voiced sounds is known as the fundamental frequency. This frequency and associated harmonics are filtered by various vocal organs and result in a distinctly audible unit of speech (phoneme) emitted through the lips. The timbre and pitch of the phoneme varies depending on a number of factors.

Timbre is determined according to vocal tract length and the combined shape of an individuals resonators such as the mouth, nasal passages and throat. These resonators determine which frequencies/harmonics will be amplified or attenuated. The combination of the produced frequencies determine the timbre of an individual's recognisable voice. Pitch is determined by the physical dimensions of the vocal cords. For example, longer vocal cords result in a lower pitch. By contracting and expanding the vocal folds, the speaker alters the pitch and prosody of the output.[19]

Due to the uniqueness of an individual's speech system and the varying characteristics of speech across languages, it can be difficult to develop a robust system capable of recognising each speaker. For example, the English language can be represented using a set of roughly forty two different phonemes while Mandarin Chinese holds a set of sixty six. On top of this, languages such as Mandarin and Cantonese are considered to be tonal languages wherein the pitch of the speaker significantly changes during the pronunciation of phonemes.

Nevertheless, all phonemes produced can be characterised by the properties of the excitation source and the transfer function of the system, making it possible for a system to recognise the human voice.

2.2.1 Phonemes

In its simplest form, VAD software is used to detect the presence of vocal utterances as opposed to speech in a signal. Considering this thesis's proposed primary use case, it is not necessary to accurately determine if the speech signal contains complex language units. Instead, the use case proposes a system capable of recognising if a speaker has produced a sound, or stream of sounds for a specified amount of time. For this reason, this study ignores lexical or morphological language units and only focuses on the presence of phonemes within the signal. Phonemes can be divided into two distinct categories: Vowels and Consonants.

Consonants

Consonants are formed by restricting air flow using various vocal organs located above the larynx such as the tongue, lips and teeth. These are known as known as articulators. Consonants can be further classified by a number of factors, specifically, the method and location of articulation and as being voiced or unvoiced. The method and location refer to how and where the consonants are produced.

- Plosive consonants are formed as a result of air flow being blocked in the vocal tract. These include the /t/, /d/, /p/, /b/, /k/ and /g/ phonemes.
- Fricative Consonants, such as /f/, /s/, /sh/, /h/, /v/, /th/ and /z/ are formed by producing turbulence as a result of constricted air flow.
- Nasal Consonants include /m/, /n/ and /ng/ which are created by lowering the soft palate allowing the nasal cavity to engage in the transmission and broadcast.
- Liquid Consonants, such as /l/ and /r/ are produced by lifting the tip of tongue to a point without completely blocking air flow.
- Semi-vowels such as /w/ and /y/ are intermediates between a vowel and consonant.
- Voiced consonants involve the vibration of the vocal folds, an example being the phoneme /v/ in the pronunciation of the word 'voice'.
- During the pronunciation of consonants such as /f/ in 'fish', there is no vibration of the vocal folds and are considered to be unvoiced.

The varied characteristics of a consonant makes differentiating them from noise to be inefficient. Instead, they can be characterised by periods of silence or their effect on the frequencies of adjacent vowels. Although some studies show promising results when detecting glottal vibration and turbulence noise as consonant landmarks, the majority of VAD software utilise other methods.[25] As a result, consonants do not provide an efficient means to recognise the human voice.[26][17]

Vowels and Formants

On the other hand, vowels prove to be much more efficiently distinguished. Extensive research shows how the classification of the voice can be determined by formant tracking alone.[13][34] This is due to the presence of distinct spectral peaks in the spectrum, which are known as formants. Extracting formants from an audio signal can indicate if a vowel is present, while preceding and proceeding segments of speech can be captured through the use of a windowing algorithm. This algorithm can be used to account for consonants.

Spectral peaks occur when there is a high concentration of energy at a specific band. Studies show that even with corruption from additive white noise, where the energies of the noise and the source signals are equal, most of the spectral peaks of the original signal remain high. In particular, additive noise that spans across all spectral bands has a very small impact on a

given spectral peak.[13][10] In order to recognise the presence of a formant in an audio signal, 3 defining parameters can be examined.

- Formant Frequency: Frequency at which spectral amplitudes in formant region reaches maximum point.
- Formant Width: Double distance from formant frequency to the frequency where the height of formant envelope is decreased by 3dB.
- Formant Intensity: Maximum height of formant envelope.

Typically, the quality of the recognised vowel can be categorised by the first 3 formant frequencies, each of which are generally located within a 1000 Hz of each other. For example, F1 is typically located between 0 Hz and 1000 Hz, F2 is located within 1000 Hz and 2000 Hz, F3 is within 2000 Hz and 3000 Hz. Spectrograms represent the strengths of the component fre-

	65+		50–55		35–40		20–25	
Vowel	F1	F2	F1	F2	F1	F2	F1	F2
iː	285	2283	269	2355	269	2312	276	2338
ɪ	382	2024	341	2074	374	2115	393	2174
ɛ	454	1962	489	1920	512	1888	600	1914
æ	644	1678	693	1579	696	1574	917	1473
ɑː	665	1085	639	1041	608	1062	604	1040
ɒ	518	875	522	865	496	833	484	865
ɔː	391	619	360	604	382	626	392	630
ʊ	376	990	371	975	381	984	413	1285
uː	301	994	283	1112	288	1336	289	1616
ʌ	630	1213	643	1215	629	1160	658	1208
ɜː	475	1321	511	1340	497	1419	494	1373

Figure 2.3: Mean frequencies of formants across four speaker groups[12]

quencies over time, which are typically utilised in data sets for training voice detection models. Because the spectrogram represents the waveform over time it can be used to recognise a wide variety of characteristics such as accent, prosody, intonation, emotion and fatigue. In order to measure changes in the vocal tract during speech, a spectrogram is used to plot the time-frequency-intensity of the short time spectrum. Both vocal tract resonances and fundamental frequency of excitation are apparent on a spectrogram.[24]

However, two major problems must be addressed in order to utilise formants. Firstly, noise which introduces significant spectral peaks can severely affect a peak-based algorithm. An instance of background noise with significant amplitude at certain bands would cause specific peaks, which can influence the algorithms ability to recognise the formant. Secondly, formants may vary across different speakers and dialects. The exact frequencies of each spectral peak may vary even for the same speaker as frequencies change while speaking. This is very prominent in the aforementioned tonal languages. Therefore, short analysis window sizes are required to

capture variances. However, such short window sizes can result in confusion with impulsive background noises containing non-relevant spectral peaks.

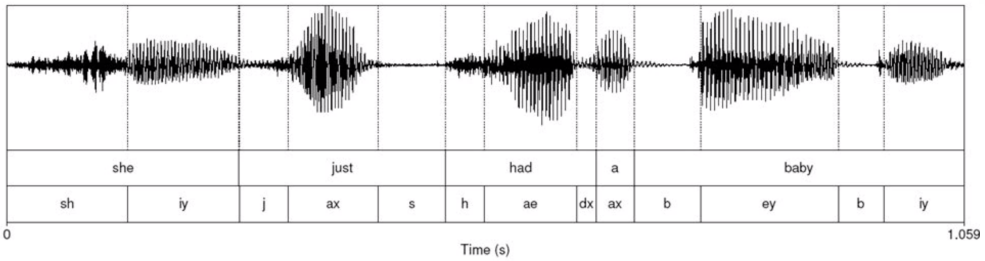

Figure 2.4: Waveform of individual phonemes within a sentence

In order to attain the necessary frequency information, an amount of signal processing should be performed on the input signal. The spectral analysis of signals is a common technique which involves transforming the acoustic signal into a spectral representation through Fourier analysis. Given that waveforms can be decomposed into individual sine waves, the spectrum represents a waveform's frequency components and their respective amplitudes at a single point in time.

2.2.2 Conclusion

From the analysis gathered, formant tracking proves to be an efficient method of determining the presence of a human voice within a signal. Using the formant characteristics presented in this section, the presence of formants can be determined through spectral analysis. This provides VAD software with a means to identify the presence of a speakers fundamental frequency, associated harmonics and formants.

2.3 Analysis Conclusion

This chapter describes the obstacles that arise when using speaker disambiguation in media capture systems. When implementing heuristics to accurately discern a speaker of interest, the primary difficulty lies in isolating what determines a speaker to have the audience's focus of attention, and how to implement a system that can recognise those characteristics. Overall, this project attempts to find a solution for identifying a person of interest based on the presence of speech utterances, and using that solution, provide an automated camera switching system with capabilities to record and broadcast a dialogue between two speakers. The next chapter will present the project design, which will illustrate the potential means for achieving the research aim.

Chapter 3

Design

The previous chapter described the challenges of identifying a speaker of interest in a media capture system. A solution for calculating the speaker of focus, and how to record and broadcast the resulting media is presented in this project. This chapter will discuss the possible methods that can be applied in an effort to discover a solution that can provide sufficient performance.

3.1 Web Based Application versus Native Application

The first step in the design process examines the advantages and disadvantages of a native application against a web based solution. The design should deliver a proficient application to a wide selection of users, with minimal machine requirements. To achieve this, a number of factors needed examination, namely, software capabilities, developer communities and support available. After much consideration, FFMPEG, HTML5, JavaScript and Node.js were identified as the most fitting candidates for consideration.

3.1.1 Native Solution

A native solution provides many benefits such as improved speed and responsiveness. This is due to the code for the application being stored locally, and as a result, allows the application to run more predictably. Conversely, a web based application requires the device to have a strong connection to the network for optimum performance. However, locally stored code also has drawbacks. It requires the user to manually update the software, as opposed to a web based solution, which updates without user interaction. Another large drawback to the native model is the differences in code bases across platforms. To deploy the application on a wide selection of platforms, multiple versions need to be developed. This factor restricts the accessibility of the application.

Other benefits include device access. The native solution provides the developer with access to the full set on a selected operating system, providing a more personalised product to the end user. As this project requires access to the users microphone and camera, this benefit is of particular interest.

3.1.2 Web Based Solution

From a development and deploying standpoint, the web based solution is an easier route. There is less requirement to adhere to the operating systems standards. Unlike Google Play and the Apple App Store, the web solution has less regulations as there is no quality control authority in regards to the marketplace. Due to the lack of regulations, the onus is completely on the developer to make sure the application is secure. Possible issues regarding security arise when dealing with access to user media devices. Therefore in order to provide an acceptable system the web application will require HTTPS certification in order to be hosted publicly. This is an absolute necessity when using the MediaDevices API proposed in the proceeding section.

From a user experience point of view, more effort is required to run a web based app as it is not directly stored on the device. As web apps are not tailored to each system, they result in being less interactive and intuitive compared to native apps. However, the most appealing aspect to the web based solution is the ability to cater for a wide variety of users regardless of user system requirements.

3.1.3 Conclusion

The research shows that native applications can provide powerful systems. However, due to the importance of this system being widely accessible, a web based platform is considered to be more desirable from the solutions offered in this section.

3.2 Candidate Technologies

3.2.1 FFMPEG

FFMPEG is a well documented, open source command line utility written in C and Assembly. It offers a vast suite of libraries and provides the user with various multimedia capabilities. A robust set of tools gives the user the ability to record, manipulate, compress and broadcast audiovisual content regardless of codec or format. It is a significant candidate for this project as it encapsulates a number of processes in the recording pipeline. It provides the user with access to connected media streaming devices. The acquired inputs can produce a specified output RTMP stream which can be aimed at a streaming platform of the users choice. It also provides a means to package the captured data to user specifications such as container format and various encoding options. A huge benefit is the capability to encode to MPEG DASH, allowing for high quality adaptive streaming. However, in order to broadcast a live stream, FFMPEG must be used in conjunction with a multimedia streaming server. FFMPEG is used to push media to a server which transcodes the stream and finally distributes to the end clients. Although alternatives exist (RedPro), FFMPEG no longer supports FFSERVER. As a result, compatibility issues arise and adjustments would need to be made in order to broadcast a live stream. Another option would be to avail of external delivery platforms such as Facebook Live or YouTube. FFMPEG provides capabilities for piping streams to a web page by specifying a destination in the output parameter.

For live production, processes are typically divided over a number of machines. For example, an individual machine for media capture pipes the captured feed to another machine which is used to broadcast the live stream.

3.2.2 JavaScript and HTML5

A hugely beneficial characteristic of designing a web based system is its accessibility. Provided the user has sufficient internet access, a web based solution would allow users an easily accessible application without the need to download or install any software. In order to achieve this, the system has a number of requirements that need to be fulfilled. The proposed system requires access to the user's camera and microphone, which could be achieved through the MediaDevices API. For signal processing, JavaScript provides the WebAudio API and for recording, the system can avail of the MediaStream Recording API. On the delivery side, HTML5 provides numerous options for media playback. For example, iframe, video and audio elements have support for MP4, WebM, Ogg, MP3 and WAV formats whereas the canvas element, when paired with JavaScript, can be a powerful tool for creating animated graphics. A benefit from using the canvas element in this project is its capability to create a 2d graphical representation of the audio input signal frequencies. This provides useful visual feedback during the signal processing stage.

3.2.3 Node.js and NPM

Node.js is a run-time environment which utilises a simplified model of event-driven programming operating on a single thread event loop. The Node.js task architecture operates through non-blocking I/O calls which allows for asynchronous I/O and support for a large number of concurrent connections. The design optimises throughput and scalability in real-time web applications with numerous input/output operations. Through the node package manager (NPM) software, users have access to publish or install JavaScript modules from the NPM registry. Although research has unveiled a number of security issues with the NPM and Node platform, the NPM registry incorporates almost 500,000 modules to aid the development of applications through Node.[23]

3.2.4 Express, WebSockets and Socket.io

Examples of NPM packages available are Express and Socket.IO. The Socket.IO library can act as a wrapper for the WebSockets protocol which enables bi-directional communication between client and server. It is intended for real time web applications and is comprised of two components; A server-side library for Node.js and client-side counterpart which runs in the browser. The benefits of using Socket.IO over WebSockets include the fallback option of polling, broadcasting to numerous sockets and asynchronous I/O. The implementation of a two way connection grants both the server and the client the ability to initiate communication, allowing a continuous connection for data to flow from either side as opposed to the traditional web response paradigm wherein the client always initiates communication. The bi-directional paradigm is desirable for the proposed system to allow the flow of data between the input

recording web page and broadcast output page. Express works as a server framework aiding in the management of the server and provides route handling capabilities. Another factor of interest is its ability to suit various API requirements and it is well documented.

3.2.5 Conclusion

The web solution combines JavaScript, HTML5, CSS, Node and associated NPM packages. This provides the proposed system with the aforementioned resources capable of achieving a sufficient result.

3.2.6 Peer-to-Peer, WebRTC and Media Servers

The Peer-to-Peer (P2P) model is a distributed system of nodes that are directly connected to each other. Each client in the network is considered a node which can both request and provide services. WebRTC enables real time data sharing and communication between clients over a P2P network through the RTCPeerConnection API. A connection between peers is achieved by exchanging session description protocols (SDP) over signalling servers. Once the connection is made, the peers are capable of streaming data directly, allowing both clients to bypass a centralised server. The proposed design can benefit from this system as it allows for faster data

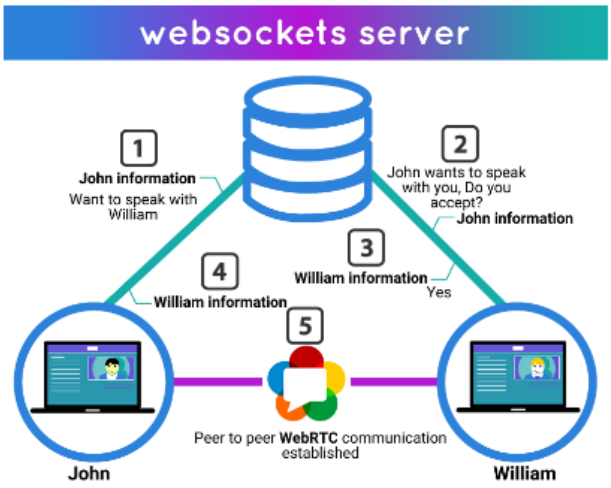

Figure 3.1: Establishing a P2P connection for WebRTC[2]

transfer. It is also possible to have multiple interconnected peers in a network providing chat room capabilities. However, the proposed system requires a solution capable of broadcasting to vast numbers of clients. Current implementations of WebRTC rooms only provide capabilities to host a max of 256 participants.[22].

The current implementations of WebRTC are to connect remote peers. Conversely, the proposed system aims to provide capabilities of sending two individual local media streams. This requires the studio page to create two RTCPeerConnections with each viewer, complicating the

architecture. Suggested solutions include the use of media servers such as Kurento and Janus. These provide capabilities to broadcast on a larger scale, however multistream broadcast from a single client is not supported.[6][18]

3.2.7 Conclusion

The WebRTC API aims to connect remote users which would greatly benefit this project. However, the proposed system aims to broadcast multiple local streams to an outside client. Although the centralised server solution produces a slower infrastructure for data transfer, it is robust and has little limitations to the number of clients who can access and view the broadcast content. Although very recent developments propose a solution to stream to vast audiences using Janus, a centralised server approach appears to be a more appropriate solution for the proposed system.[6]

3.3 Audio Routing

Depending on the users input hardware, it may be necessary to install external audio routing software. In order to facilitate input for multiple audio and visual input devices, the user may require an external audio interface. There are a number of options available to accommodate such instances.

3.3.1 LoopBack, iShowU, VB Audio Cable and Jack/JackPilot

LoopBack provides sufficient capabilities for creating virtual audio devices. These devices can pass audio from input devices to audio processing software. The appealing characteristics of the LoopBack solution is the easy to use interface. However the LoopBack solution comes with a drawback. As LoopBack is platform dependant and only available for MacOS, it obstructs a key objective of the proposed system. While similar software exists for both MacOS (Jack-/JackPilot, iShowU) and Windows (VB Audio Cable), there does not appear to be a cross platform solution. For a proof of concept, a free trial of the LoopBack software was used in the design of the proposed system.[3]

3.4 Signal Processing

The proposed system requires access to the user's camera and microphone in order to capture an audiovisual stream. The acquired stream must then undergo a level of processing in order to determine if speech is present and, ultimately if the speaker is of interest. A number of libraries and technologies were considered to complete the task based on their accessibility and capabilities.

3.4.1 MediaDevices API

The MediaDevices API is a JavaScript library that is intended to be used in conjunction with the MediaStream or WebAudio API. It provides a system with access to connected media capture

devices. A benefit of this API is that it provides the system with access to connected device information, such as the ID. In regards to the proposed system, this information is beneficial as it can be used to distinguish between capture devices in a complex system. The API also has methods to acquire information regarding device capabilities, such sample rate. This is required when determining the Nyquist frequency, necessary for calculating the bandwidth of the frequency bins in the proceeding stage.

3.4.2 MediaStream API

The MediaStream API provides methods to create and control streams of multimedia. The API is built around two components, the MediaStreamTrack and MediaStream interfaces. MediaStreamTracks are isolated streams that originate from a single source, such as a microphone. They can be of either audio or video nature. The MediaStream object is comprised of MediaStreamTracks and acts as a single unit which can either be recorded or rendered within a media element. This API, combined with MediaDevices, provides the proposed system with an infrastructure to capture input streams for processing.

3.4.3 WebAudio API

WebAudio is a widely supported, high level JavaScript API for processing and synthesising audio. It is currently published as a W3C candidate recommendation with extensive documentation available. WebAudio is supported across most browsers with the exception of Safari and Internet Explorer. The API is built around an AudioContext, which acts as a directed graph of connected AudioNodes. The AudioNodes determine the flow of an audio stream from a source

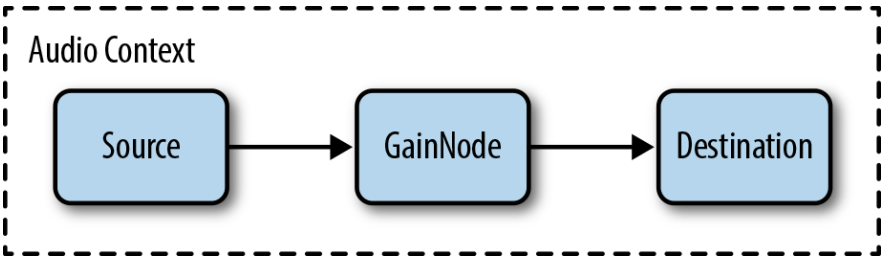

Figure 3.2: AudioContext[5]

to a destination. Each node in the processing chain allows the audio stream to be manipulated and inspected. The variety of AudioNodes can be categorised as follows:

- Source nodes: Input
- Modification nodes: Processors, filters, panners, etc.
- Analysis nodes: Analysers and processors used for signal analysis
- Destination nodes: Output

Although the processing takes place on the underlying code (C/C++/Assembly), direct script processing is permitted through the use of the AudioWorklet node.

The AnalyserNode provides support for frequency analysis of an audio signal. It is capable of analysis over the time or frequency domain and outputs results based on FFT analysis. The frequency data is calculated by first computing the current time-domain data. A Blackman window is then applied follwed by a FFT. The results are smoothed over time and finally converted to decibel. The API provides the ability to alter the FFT buffer size, which allows a more detailed analysis of the signal but at the cost of performance. Another benefit provided is the *smoothingTimeConstant* property, which provides capabilities to alter the amount of smoothing performed on the window.

3.4.4 DSP.js

DSP.js is a widely used digital signal processing library. The library provides similar capabilities to WebAudio in terms of Fourier transforms and filters. However, the number of tools provided are not as extensive. This being said it offers some benefits not available through WebAudio including various window functions (Hann, Hamming). A factor worth considering is the support available. Compared to WebAudio, the DSP library is not as robust with less documentation available.

3.4.5 Conclusion

This section introduced a means to capture user media and provided two signal processing libraries for consideration, DSP.js and WebAudio. Due to the capabilities and extensive documentation provided, WebAudio is considered to be more desirable for this project's needs. The task to be carried out by the candidate technology is to recognise characteristics unique to the human voice, utilising knowledge gathered from the previous chapter. The implementation of the chosen library will be described in the next chapter.

3.5 Recording and Delivery

To create an infrastructure capable of transporting captured content to a remote viewing page, a number of technologies were considered. An aim of the project is to deliver content with minimal latency. A proof of concept is required to demonstrate that the system is capable of representing a broadcast signal at a location separate to the web page from which the feed was recorded. As the P2P method constricted the number of possible viewers, the WebRTC and Media Server approach was not considered in the design.

3.5.1 MediaStream Recording API

MediaStream Recording provides capabilities to record media streams over various container formats which can be specified by the APIs *mimeType* attribute. A list of accepted MIME types can be found in the appendix. A downfall to this API is the inability to record audio without data loss. As the proposed system requires analysis of the input audio signal, It is important that the signal retains as much information as possible before it is analysed. In order to accomplish this, analysis should be performed before compression of the signal. The

API provides a means to record a media stream as an array of Blobs. The *dataavailable* event returns the contents of the recording, the data of which is a Blob. This can be acquired through the *ondataavailable* event handler or *requestData* function. Using these methods, it is possible to retrieve chunks of audiovisual data from the recorded stream while recording is still in progress. The chunk size can be specified by the user, allowing the system handle recorded data as it pleases before the recording has finished.

3.5.2 Recorder.js

Recorder.js is intended to be used in conjunction with the WebAudio API. It acts as a plugin to export the output from an AudioNode. A large benefit of the library is its capability to record as uncompressed PCM audio, allowing for a more accurate frequency analysis of the recorded signal and a means to efficiently package recorded audio for server side VAD methods. Although recorder.js is widely used and supported, the documentation is not as extensive as other candidates, such as MediaStreamRecorder. What's more, the recorder.js github repository is no longer being actively maintained.

3.5.3 MediaSource Extensions (MSE) API

The MSE API is built around the concept of a MediaSource, which acts as a container for information. This provides the user with a means to assign a MediaSource object as the source value of a media element. A benefit of doing so is the users capability to implement a SourceBuffer to dynamically append and remove chunks of data to the MediaSource without changing the source value of the media element. This could prove beneficial to the proposed system as in a live recording scenario, the audiovisual data is constantly being updated. Conversely, when broadcasting pre-recorded content, the MSE solution is unnecessary. The recorded material, in its entirety, is encapsulated within a single chunk of data and thus the source value can be assigned to a single source. There are however, other benefits to utilising MSE. By dynamically appending and removing chunks to a media element there is minimal space being used on the server at a given time. The proposed live broadcast method treats recorded chunks as transient units of data as they are deleted after consumption.

3.5.4 Conclusion

MediaStream Recorder provides sufficient capabilities to record and package multimedia content for the purposes of this project. However, as the MediaRecorder API is incapable of recording audio in a lossless format, the frequency analysis should be undertaken on the client side before media is recorded. This can be achieved through the use of the aforementioned AnalyserNode from the WebAudio API, allowing signal processing to occur before the signal has been compressed. When used in conjunction with MSE, the proposed system can minimise excess chunks occupying space by dynamically disregarding the viewed segments. However, this is only possible for live broadcasts.

3.6 Broadcast Methods

There are a number of viable options to examine when designing a project such as this. Two possible systems present themselves. A choice needs to be made whether the system should provide capabilities to record content and post later or to stream a live broadcast. This section examines two possible designs.

3.6.1 Live Solution

The proposed system's live broadcast architecture is comprised of two web pages, both of which are connected to a Node server via a socket. The studio web page is dedicated to first obtaining multimedia streams from the users inputs, then performing frequency analysis on the acquired streams and finally recording both streams. The studio page aims to simultaneously record two separate audiovisual feeds in chunks of specified length. The chunks are simultaneously sent to the server. From there, the chunks are sent to the broadcast page over a socket. The broadcast page is comprised of two video elements, which will serve as an output for each recorded stream. Upon the chunks arrival at the broadcast page, the source of the video elements are assigned to their own respective recorded chunk. This provides the broadcast page with two separate video feeds at a given time. Using HTML, CSS and JavaScript, the visibility of each media element will be manipulated depending on the results found from the frequency analysis of the respective media element's designated stream.

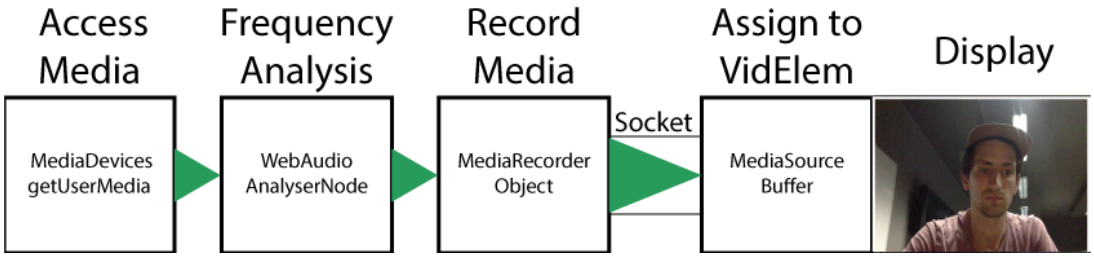

Figure 3.3: Flow of data in proposed live solution

3.6.2 Pre-Recorded Solution

The proposed system's pre-recorded broadcast architecture is comprised of a single studio web page hosted on a node server. Similarly to the live solution, the studio page is dedicated to acquiring, processing and recording multimedia streams. Each multimedia stream is to be recorded in chunks however, the streams are not to be recorded simultaneously. Each recorded chunk length is determined by the presence of the human voice in the signal. The recorder is to commence recording speakerA as soon as the streams are initialised. From that instance, whenever a speaker begins to talk, the recorder stops recording current chunk and begins recording the subsequent chunk. The newly recorded chunk is sent to the server via HTTP request. After the session has concluded, the final chunk is posted to the server and all chunks are concatenated to create a single media file.

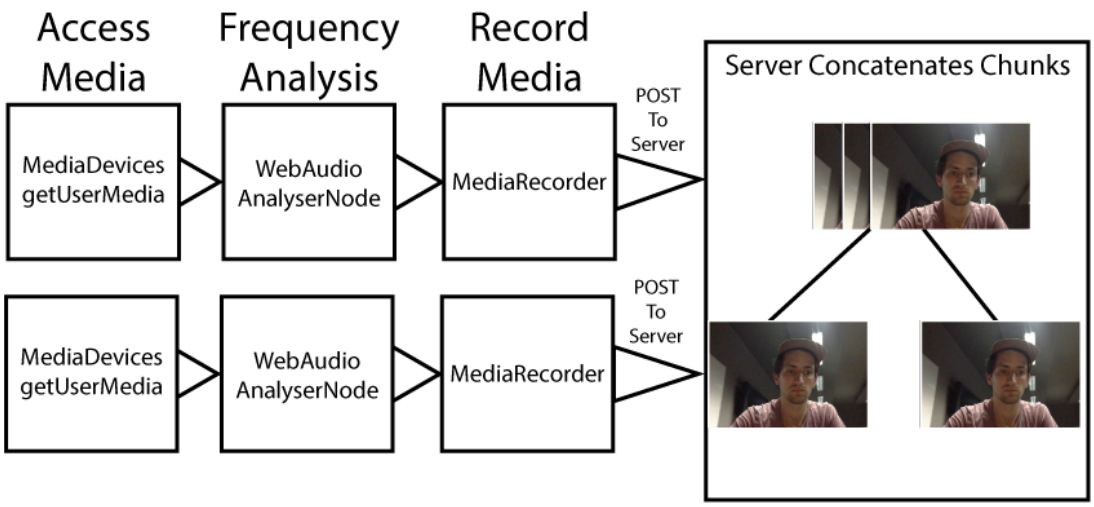

Figure 3.4: Flow of data in proposed pre-recorded solution

3.6.3 Conclusion

Through the examination of various technologies, two potential frameworks have been considered. The first design proposes a fully automatic system capable of real time recording and broadcast, whereas the second design proposes a "Record now, post later" system capable of outputting a recorded edit for the user to upload after the recording process has finished. The decision was made to achieve a fully automatic live broadcasting system as it enables users provide a more interactive experience. This project will employ Node, JavaScript, CSS and HTML5 in conjunction with the WebAudio, MediaStream and MediaRecorder APIs to implement the proposed live system. All proposed libraries and APIs are supported across all browsers except for Internet Explorer and Edge.

Chapter 4

Implementation

The previous chapter illustrated the proposed system's components. This chapter aims to detail how they are implemented. As the project processes a stream of data in stages, each stage should be isolated and tested separately.

4.1 Project Architecture

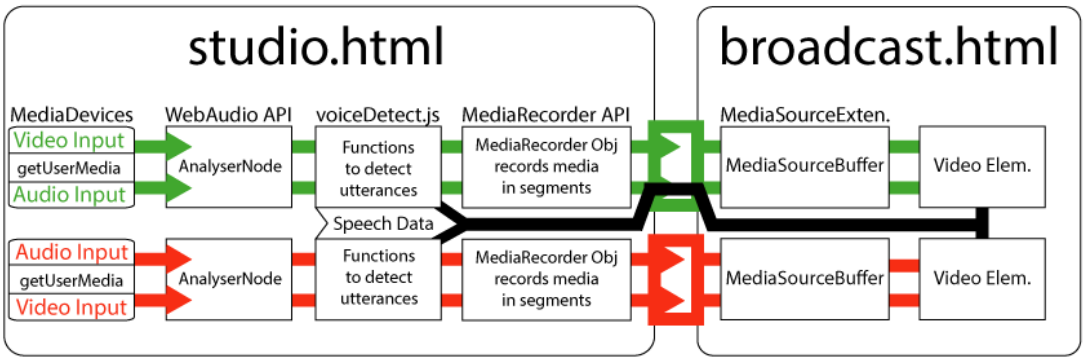

Figure 4.1: Diagram of Proposed Project Architecture

4.2 Hardware

The hardware used for the implementation of the proposed design are as follows:

- Focusrite 2i4 USB Audio-Interface
- Shure SM58 Dynamic Microphone
- MacBook Pro Built-in Mic
- Apple FaceTime WebCam

- Logitech HD Pro Webcam C920

The audio interface provided two physical inputs for microphones. Although this may not be necessary for other audio interface devices, 2 virtual devices needed to be created for the Focusrite 2i4 using LoopBack. This allowed the browser to recognise the audio-interface as having 2 separate mono inputs.

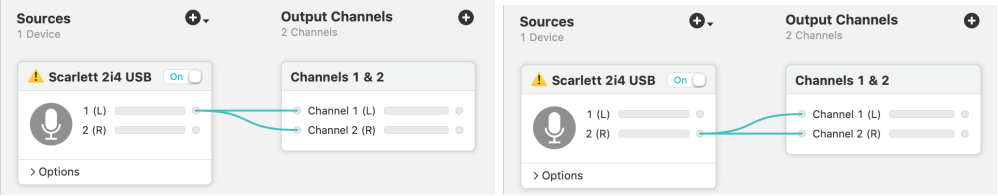

Figure 4.2: LoopBack Interface

Figure 4.2 illustrates how the LoopBack interface is utilised to create two virtual machines, providing the system with SpeakerA and SpeakerB input channels.

4.2.1 Conclusion

The systems design was implemented and tested with a number of different audio and video input devices. This was done in order to present a system that could cater to a wide variety of users, regardless of their hardware. The audio capture devices produced dissimilar results when the captured frequency data was analysed. This is due to the different frequency responses of each device. As the method for voice activity detection presented here depends heavily on the analysis of the captured frequencies, the proposed system needed to be calibrated differently for each audio capture device.

4.3 Mediastream Architecture

This section outlines how the system implements the MediaDevice and MediaStream APIs to obtain a users audiovisual feed, and transform it into a usable MediaStream object.

4.3.1 Media Capture

To make sure that the system is capable of communicating with a users media device, the application first calls the *getUserMedia* function. This function creates a reference to *getUserMedia* that can be recognised across all compatible browsers.

In order to acquire media input from the user agent, the system avails of the MediaDevice's *getUserMedia* function. By calling this function, the system prompts the user for permission to utilise a media device input. This asynchronous operation returns a promise, which ultimately resolves to a MediaStream object, providing that the promise has been fulfilled. A benefit to JavaScript providing promise capabilities, is that subsequent chained events can be fired upon fulfilment of the preceding promise. This is accomplished through the use of the *then* method. The system employs this method to call a function which provides UI elements with connected

device information. The *initStreams* function is then attached, which initialises AnalyserNodes, necessary for signal processing to occur once a media stream has been obtained. Finally, a function to catch errors is attached providing more informative feedback as to why a function failed.

getUserMedia takes a MediaStreamConstraints object as an argument, which specifies the tracks, and their settings, to be included in the acquired stream. The track settings include information such as device ID, channel count, sample rate, sample size and frame rate. Although some track information is read-only, it is possible for the user to alter certain settings such as frame rate. The resulting MediaStream object provides the system with a signal for processing.

The proposed system obtains two individual input streams and therefore implements an individual function to accommodate each. The *getMediaWithConstraints* and *getSecondMediaWithConstraints* functions each take two arguments: an audio source and a video source. These arguments correspond to the device IDs of the desired capture devices, which the system can be made aware of through the *MediaDevice.enumerateDevices* function. Once the system is made aware of connected devices, the user is provided with an option to select the desired device through UI elements on the client page.

When the promises resolve, two media streams are created which are passed to the *gotStream* and *gotSecondStream* functions. These functions assign each stream to an associated video element's *srcObject*, which provides the user with visual feedback of the acquired video feeds. Both functions also make the system aware of the track IDs of both audio and video tracks from each stream. This information is useful when combining selected tracks to create new MediaStream objects. For instance, a scenario wherein the user requires a single stream to have multiple audio tracks and a single video track.

4.4 Signal Processing

The previous section illustrates how the system acquires a MediaStream object. From this point, the system implements the WebAudio API to perform signal analysis on the obtained signal. This section details how the signal is decomposed into usable data to be utilised in the voice detection process.

4.4.1 AudioContext and Connections

The *initAnalyserNodes* function creates an infrastructure to conduct audio signal analysis. It creates an AudioContext with a specified sample rate for each audio track, and inserts an AnalyserNode into each context. The FFT size for each AnalyserNode is specified and frequency bin width is calculated by dividing the sample rate by FFT size. At this point, cutoff markers are approximated to determine a crude bandwidth of the human voice, while still allowing a wide enough band for utterance recognition. This is achieved by the following function.

```
1  lowCutoff = Math.floor(300/(sampleRate/FFTsize));
2  highCutoff = Math.floor(3400/(sampleRate/FFTsize));
```

The 300 Hz and 3400 Hz values are coherent with the figures documented for the usable voice band in telephony.[**?**] Although audio quality is lost, this method is only used to filter analysis

data. The AnalyserNode allows the output signal to remain unchanged and thus the quality of the broadcast content will not be affected. An 8bit unsigned integer array is constructed for each AnalyserNode, with the intention of holding an audio signals frequency data. The contents of the created arrays will later be filled with the results of the FFT analysis performed by the array's corresponding AnalyserNode. At this stage a decision was made to represent the

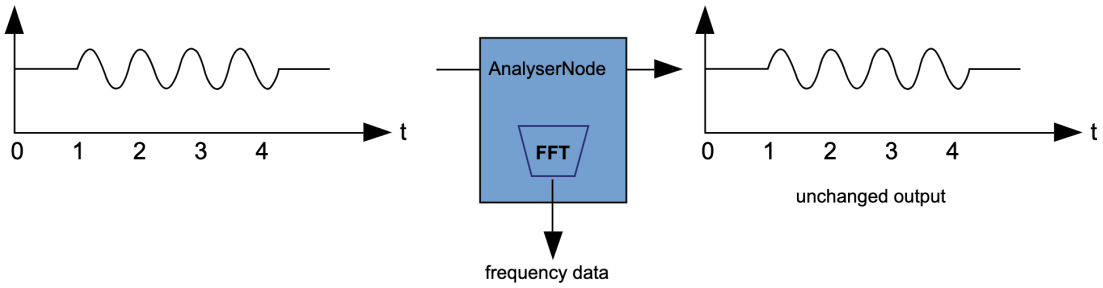

Figure 4.3: AnalyserNode[5]

frequency data as 8bit unsigned integers instead of 32bit floats. The AnalyserNode provides capabilities to obtain audio signal frequency data as a float or byte. The *getByteFrequency-Data* method copies the frequency data values and represents the results as a normalised value between 0 and 255. *getFloatFrequencyData* does the same but provides a representation with a higher resolution. An AudioNode is created from each MediaStream object and placed in

$$b[k] = \left\lfloor \frac{255}{\mathrm{dB}_{max} - \mathrm{dB}_{min}} (Y[k] - \mathrm{dB}_{min}) \right\rfloor$$

Figure 4.4: getByteFrequencyData formula[5]

a corresponding AudioContext. This is achieved using the *createMediaStreamSource* method. The method takes an argument of a MediaStream object and returns an AudioNode which acts as a source node for the AudioContext. This source node is connected to an AnalyserNode, providing an audio signal to perform analysis on. From this point, an event loop is triggered which provides the system with a means to update the input values obtained from the source node.

4.4.2 The Event Loop

The evtLoop function acts as an event loop which successively triggers events repeatedly. This is achieved by placing a *requestAnimationFrame* function inside the evtLoop operation. The purpose of the loop is to update the contents of the frequency data arrays so the system can continuously analyse a signal. From this function, a number of processes are given the capability to perform in real time.

```
1 function evtLoop(time) {
2   let detectUtterance = false;
3
4   analyser.getByteFrequencyData(freqData);
```

```
5
6   // definition of an 'utterance':
7   if ( hasStrength && withinSimilarBand && formantsMatch ){
8     detectUtterance = true;
9   } else {
10    detectUtterance = false;
11  }
12
13  // shifts elements of analysedStream array
14  slideWindow(analysedStream);
15
16  // fill first index with most recent input value
17  analysedStreamA[0] = detectUtterance;
18
19  window.requestAnimationFrame(evtLoop);
20 }
```

Listing 4.1: Event loop to allow continuous analysis

4.4.3 Voice Detection

The voice detection software implemented in the proposed project operates around a set of assumptions presented in the analysis chapter. The functions to assess if the frequency data contains voice information are encapsulated within the *voiceDetect.js* file. Each function returns a boolean value to signal if the data has met a specific requirement. A number of factors which affect the efficiency of the following operations should be considered. The FFT size and sample rate directly correlates to the bandwidth of the frequency bins. Therefore, the frequency data array from which the systems decisions are based upon, will depict crude or precise results, based on FFT size and sample rate. Frequency bins are derived from the resolution and sampling frequency of the transform. The width of each bin can be calculated by dividing the sample rate by FFT size. A visual representation of the collected bins contained in the frequency data array can be seen from 4.5. Each bar in the graph represents a bin. These bins hold the various frequency amplitudes of the input audio signal.

Lower FFT sizes and sample rates result in better system performance. The WebAudio AnalyserNode allows for a minimum FFT size of 256, while the typical sampling rate of a device is 44.1 kHz. This results in a frequency bin width of 172.265625Hz. While these settings pose a less cumbersome task than, let's say an FFT size of 1024, the resulting data is less precise. In light of these findings, the operations were executed with different FFT parameters in order to achieve a capable solution with optimum performance. The proposed voice activity detection model estimates the presence of human voice in a signal by taking an array of frequency data input and outputting a single boolean value.

Has the signal sufficient strength?

The initial requirement states that the input signal should be of sufficient strength. The threshold of acceptance is defined by an interactive UI element which can be manipulated by the user. To calculate the input signal strength, the system computes the root mean square average of all

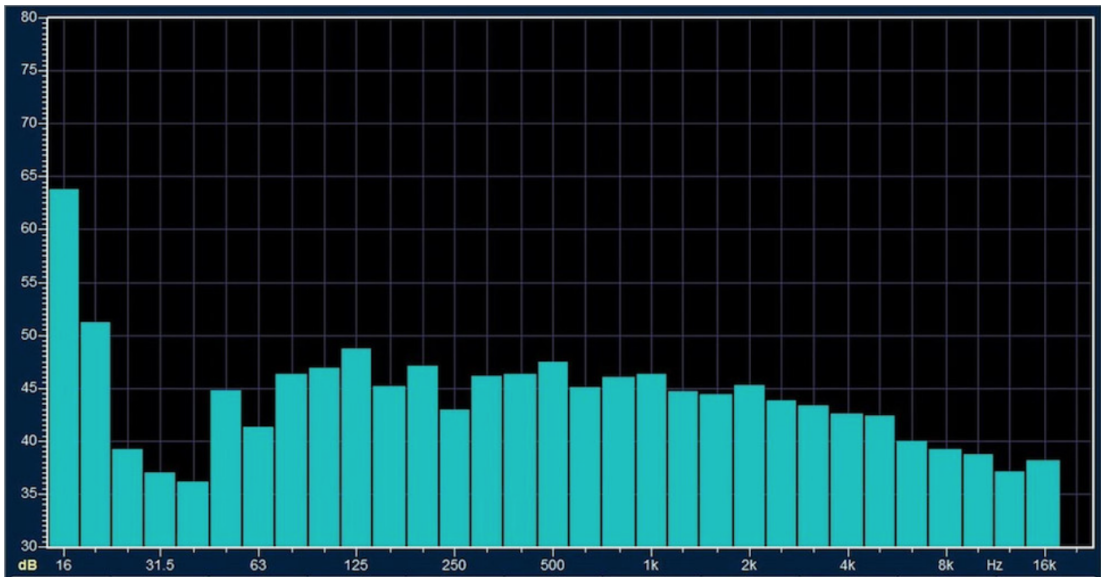

Figure 4.5: Frequency Amplitudes of a given input signal

frequency amplitudes in the frequency data array. The RMS value should be greater than the user defined threshold in order for the requirement to be met. This function does not require a large FFT size as using the minimum FFT size provides results similarly equivocal to those of a higher FFT size.

```
let calculateRMS = (frequencyData) => Math.sqrt(
                    frequencyData.map( val => (val * val))
                    .reduce((acum, val) => acum + val)
                    /frequencyData.length);
```

Listing 4.2: Function to Calculate RMS of Data

Does the signal resemble the voice band?

The second requirement states that the average frequency amplitude within the human voice band (approximately 300 Hz - 3400 Hz) is greater than the average frequency amplitude inside of the noise bands (all frequency bins outside of voice band). This provides the system with the assumption that the most powerful frequencies within the signal are of a similar bandwidth to the human voice. This is calculated by getting the average of all frequency bins between the lowCutoff and highCutoff points. The average frequency amplitude from frequency bins outside of the voice band is also calculated. If the ratio of the average frequency amplitude within the voice band is greater than three times the average frequency amplitude within the noise band, the function returns a true value. As the human voice band is an approximation, the frequency bin width does not need to be precise and the minimum FFT size provided adequate results.

```
function getAvgByteVoiceBins(frequencyData){
  var avg = 0;
```

```
3   var n = 0;
4   for(var i = lowCutoff; i < highCutoff; i++){
5     avg += frequencyData[i];
6     n += 1;
7   }
8   avg = Math.floor(avg/n);
9   return avg;
10 }
```

Listing 4.3: Get Average of Voice Bins

```
1  function getAvgByteNoiseBins(frequencyData){
2    var avg = 0;
3    var n = 0;
4    for(var i = highCutoff; i < frequencyData.length; i++){
5      avg += frequencyData[i];
6      n += 1;
7    }
8    for(var i = 0; i < lowCutoff; i++){
9      avg += frequencyData[i];
10     n += 1;
11   }
12   avg = Math.floor(avg/n);
13   return avg;
14 }
```

Listing 4.4: Get Average of Noise Bins

Does the frequency data contain spectral peaks similar to formants?

The third requirement states that the frequency data should contain information indicating the presence of spectral peaks. If a number of peaks are located, the signal is considered as a candidate for containing human voice. In order to accurately determine the presence of the peaks, a number of operations are tested. The first method tested iterates through the frequency data array, locating the 3 indexes containing the highest frequency amplitudes. These frequency bin values are tested to see if the contained frequency amplitudes are above a specified threshold.

The second method searches for a peak in each formant band. This is based on the assumption that a formant occurs inside each 1000 Hz band.

```
1  function findFormants(frequencyData){
2    let f1 = 0, f2 = 0, f3 = 0;
3
4    for(var i = 0; i < f1Band[0]; i++){
5      if(frequencyData[i] > f1){
6        f1 = i;
7      }
8    }
9    for(var i = f1Band[0]; i < f2Band[0]; i++){
10     if(frequencyData[i] > f2){
11       f2 = i;
12     }
13   }
```

```
14    for(var i = f2Band[0]; i < f3Band[0]; i++){
15      if(frequencyData[i] > f3){
16        f3 = i;
17      }
18    }
19    let formants = [f1, f2, f3];
20    return formants;
21  }
```

Listing 4.5: Function to find F1 F2 and F3

Are the formants similar to the human voice?

The fourth and final requirement states that the spectral peaks should resemble the formant characteristics of the human voice. Because the variance of formant frequencies depends on gender, age and language, the proposed system is required to encompass a wide range of speakers. More robust systems utilise machine learning techniques to achieve more accurate detection. This system however employs more rudimentary techniques to attain formant recognition. A number of methods were implemented and tested to achieve this goal.

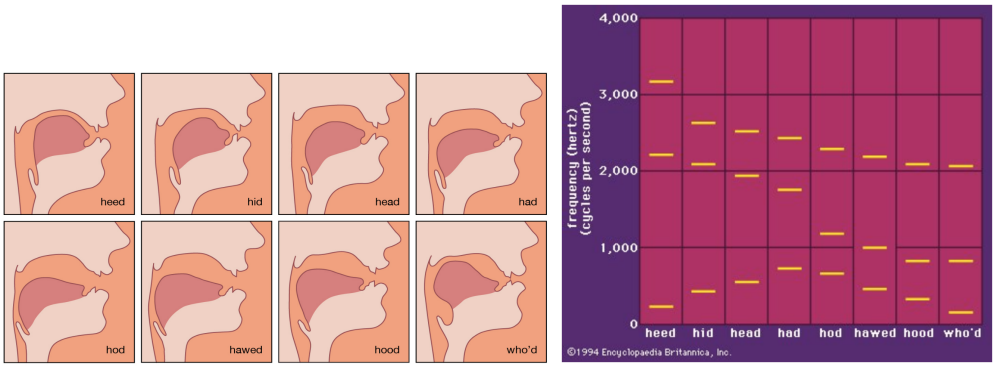

Figure 4.6: Mouth shape and formant frequencies across vowel pronunciations

The first implementation to determine if the peaks resemble formants, allows the user to calibrate the system to their own formant characteristics. This is achieved through the implementation of 3 double handled UI range sliders, which can be manipulated to specify the bandwidth where F1, F2 and F3 are located. This solution requires the user to pronounce specified phonemes into the system (heed, hid, head, had, hod, hawed, hood, and who'd). Visual feedback is given to the user when a formant falls within the specified bandwidth.

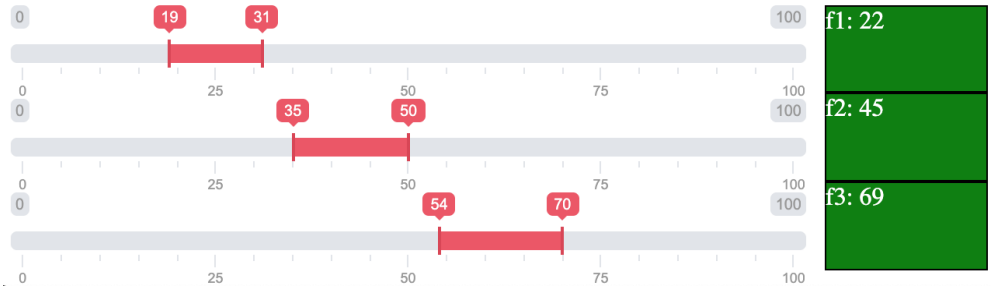

Figure 4.7: Implemented UI elements to calibrate users individual formants

To symbolise that the formant is within the user defined bandwidth, a formant's associated red box turns green. It should be noted that this method requires a larger FFT size than the previous three requirements as the disambiguation formants from noise requires a more precise analysis. Although the manual calibration method produces a result tailored to each individual formant range of the speaker, the calibration process is tedious. To provide a more acceptable user experience, a proposed solution requiring less user interaction was implemented. The solution uses information regarding the relative distance between formant locations in the frequency bin array. Similarly to the manual calibration method, in order for this solution to provide adequate results, it is essential to have a sufficient FFT size. The distance between each formant was easily calculated by subtracting the frequency bin indexes of the previously acquired spectral peaks. For example:

```
f2Distance = frequencyData[spectralPeak2] - frequencyData[spectralPeak1]
```

This solution also requires user interaction in order to produce adequate results. However, as the user needs only calibrate an acceptable range for two distances, the amount of interaction is significantly reduced. The number of interactive UI elements can be reduced from 3 to 2.

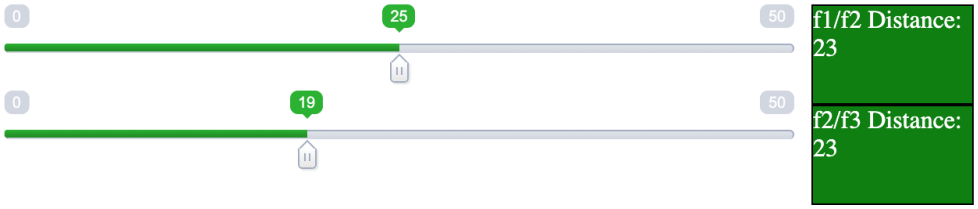

Figure 4.8: Implemented UI elements to calibrate users formant distances between bins

A crude, automatic method was implemented utilising the data gathered from figure 2.3. Using a switch statement the *checkFormantDist* function compares the distances returned from *findFormantDist*.

```
let dist = findFormantDist(freqData)

function checkFormantDist1(dist){
  let isF1;

  switch (true) {
    case dist == 1:
```

```
8         isF1 = true;
9         break;
10      case dist == 4:
11        isF1 = true;
12        break;
13      case dist == 5:
14
15        ...
16
17      }
18      return isF1
19  }
```

Listing 4.6: Automatically compares formant distances

A similar function is used to test the distance between F2 and F3. However, due to the wide variety of the human voice, the data returned from *checkFormantDist* is error prone and not reliable.

4.4.4 Conclusion

As manually calibrating the system is less desirable than an automatic approach, the proposed system makes use of formant distance properties in order to implement a solution with minimal user interaction. However, the formant distance solution does not provide results as exact as the manually calibrated format location solution. As a result, a compromise is proposed, wherein the user manually calibrates distances using two UI range slider elements. A margin is implemented wherein matching values north and south of the UI slider value are accounted for. This can bee seen in listing 4.7

```
1   let uiSliderMaxValF1 = f1Dist + 8,
2       uiSliderMinValF1 = f1Dist - 15,
3       uiSliderMaxValF2 = f2Dist + 8,
4       uiSliderMinValF2 = f2Dist - 15,
5
6   if(formantArray[0] > uiSliderMinValF1 && formantArray[0] < uiSliderMaxValF1){
7
8       // F1 peak is matched
9
10      if(formantArray[1] > uiSliderMinValF2 && formantArray[1] <
            uiSliderMaxValF2){
11
12        // F2 peak is matched
13
14      } else {
15
16        // F2 peak is not matched
17
18      }
19    } else {
20
21        // F1 peak is not matched
22
```

```
23 }
```

Listing 4.7: Code to accommodate peaks near the user selected distance

4.4.5 Analysed Stream Output

The previously discussed operations determine whether the input signal contains characteristics similar to the human voice. In order for the system to calculate the detection of speech more efficiently, the returned values are used as flags to determine if a single *isUtterance* boolean value is true or false. Each VAD operation's returned value is utilised in a conditional statement to determine the *isUtterance* value, which is updated with each execution of the evtLoop function. The *isUtterance* boolean represents the models overall analysis of the frequency data.

As the proposed system can only identify voiced phonemes, a number of factors need to be considered when identifying speech over time. Originally, a countdown timer was implemented to cater for non detectable utterances and gaps within a stream of speech. To implement this concept, a *setTimeout* function was utilised in conjunction the throttle method from the underscore.js library. When the system initially detects a vocal utterance, an *isSpeaking* variable is assigned the value true and a countdown timer is initialised. Each subsequent detected utterance resets the timer, keeping the *isSpeaking* boolean value set to true. When the timer runs out, *isSpeaking* is set to be false. This method proves to be inefficient as it does not account for brief speaker interjections during conversation. It also is not scalable to suit the varied pace between speakers. For example, speaker A speaks at a quicker pace to speaker B. If the system has allocated the same amount of time between utterances for both speakers, two possible unwanted scenarios can occur. In scenario A, The system considers speaker A to be speaking for an unnecessary amount of time after their last utterance has been emitted. In scenario B, speaker B isn't allocated enough time in between utterances and as a result, is prematurely deemed to be not speaking. As a result, a more efficient solution was explored.

After obtaining a single value to denote the presence of an utterance, the system requires a means to analyse a string of *isUtterance* values over time. This is achieved through implementing the proposed *analysedStream* array. The array is of fixed length and contains boolean values which are updated with the most recent *isUtterance* value. The most recent *isUtterance* value replaces the oldest value in the array. By doing so, the original stream of complex audio frequency data is now reduced to a simplified string of true or false values which indicate the presence of utterances over time. The *analysedStream* array acts similarly to a sliding window.

```
1  function createWindow(size) {
2    for (var i = 0; i <= size; i++) {
3      analysedStream.push(false);
4    }
5  }
6
7  function evtLoop(){
8      let detectUtterance = false
9
10     // if data contains an utterance, functions update detectUtterance value;
```

```
11
12      analysedStream[0] = detectUtterance;
13  }
14
15  function slideWindow(analysedStream) {
16    let temp = analysedStream[0];
17
18    for(var i = 0; i < analysedStream.length; i++)
19    {
20      analysedStream[i] = analysedStream[i + 1];
21    }
22
23    analysedStream[analysedStream.length-1] = temp;
24  }
```

Listing 4.8: createWindow() and slideWindow()

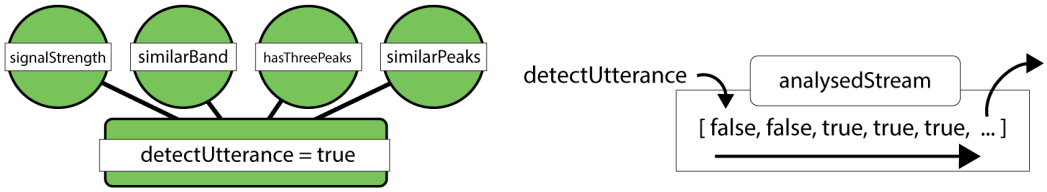

Figure 4.9: Voice detection process

A number of methods were tested to assess the contents of the analysed stream. These were achieved through various JavaScript array methods executed at regular intervals. The first implementation utilised the *Array.prototype.includes* method, which checks if the *analysedStream* array contains a single 'true' value. This method, although more efficient, gave results similar to those of the countdown timer method. The method didn't provide a means to estimate if the stream contained non pertinent utterances or interjections such as "hmm". As a result, if the *.includes* method located a single true value within a speakers analysed stream, that speakers importance would be regarded equivalent to that of another speaker with 80 true values.

```
1  let includesMethod = () => {
2    if (analysedStream.includes(true)){
3      // show speakers video feed;
4    } else {
5      // hide speakers video feed;
6    }
7  };
```

Listing 4.9: analysedStream.includes() method

The Array.prototype.filter method provides a means to calculate the ratio of true/false values. By regularly computing the percentage of true values in the stream, the system can determine if the speaker is continuously emitting utterances in succession. This operation accounts for instances of non pertinent utterances such as interjections. For example, speaker A is uttering a string of pertinent speech and as a result has the focus of attention. If speaker B interjects occasionally with utterances such as "hmm" and "yeah", the system can disregard such utterances by assessing the *analysedStream*. This provides the system with capabilities to more

efficiently estimate which speaker is of more importance, and in this case, maintain the focus of attention on the desired speaker. By introducing a means to compare the ratio of utterances between speakers, this method provides many more approaches to determine which speaker deserves the focus of attention.

```
let isSpeakingA = (analysedStream) => {
  let count = analysedStream.filter(v => v).length;
  windowRatioA = (count/analysedStream.length)*100;
};
```

Listing 4.10: analysedStream.filter() method

4.4.6 Conclusion

After implementing various methods to calculate how to determine if a speaker is consecutively producing utterances, the decided implementation is to populate an array with boolean values obtained from the previous voice detection section. This array is updated continuously. The *isSpeaking* function, as seen above, provides a means to calculate if a speaker is producing relevant speech data over time. By performing the filter method on the *analysedStream* array, an integer representing the percentage of true values is returned. This allows the system to compare ratios between speakers to determine who deserves focus of attention

4.5 Recording

The method in which media is recorded and delivered is implemented using the MediaRecording API in conjunction with MediaSourceExtensions. A MediaRecorder object takes a MediaStream and MIME type as arguments to determine the stream, container format and codec of the recording. The MediaRecorder records data in chunks, the length of which can be specified (in milliseconds) by the user. When a chunk has been recorded, an *ondataavailable* event is triggered. This event can be attached to a handler, allowing the system to handle individual chunks of media while recording is still in progress.

4.5.1 Single MediaRecorder Solution

Originally the proposed system implemented a single MediaRecorder object, which began recording a particular media stream upon the detection of voice within the stream. This solution proved to be inefficient as the gap between stopping a recording and initiating a new recording resulted in the loss of media. Although the loss of visual data is not extremely noticeable, the gap between audio chunks is. A solution to this was to treat each stream as it's own individual camera feed and send both streams to a broadcast page. The decision as to which feed is visible at a given time is discussed later.

4.5.2 Multiple MediaRecorder Solution

The proposed system simultaneously records two audiovisual streams in chunks of fixed length, using two individual MediaRecorder objects. The *ondataavailable* event triggers a function

which sends each completed chunk to a broadcast page over a socket. This provides the broadcast page with two continuous, time regulated streams of media data. The broadcast page is composed of two video elements. Each video element has an associated stream of chunks which arrives from the recording page.

4.6 Delivery

Initially, in order to play the recorded chunks, the proposed system assigned the video element source attribute to each chunk upon arrival. This method gives extremely undesirable results as it requires the video element to update the source value every time a new chunk arrives. This causes the video element to constantly refresh every few seconds, resulting in a perceived flickering or periods of black frames. To overcome this issue, another video element and MediaRecorder object was created for each stream, resulting in 4 video elements and 4 MediaRecorders. Figure 4.11 illustrates the flow of data from each MediaRecorder (MR) to

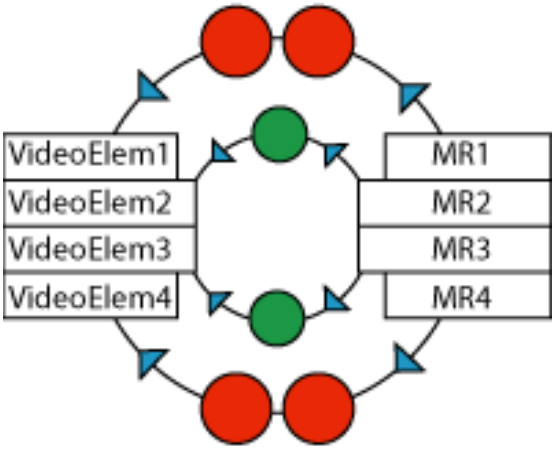

Figure 4.10: Flow of data between each MediaRecorder and their corresponding video element

their respective video element. In order to hide the black frames caused by the video element refreshing, an overlapping chunk was sent in between the arrival of the two original chunks which can be seen in figure 4.10. The opacity of each video element is manipulated to alleviate the perceived black frame. This is achieved by firstly regulating the time at which each chunk is recorded and sent to the broadcast page and secondly by regulating the time at which each video element is visible. This solution is extremely inefficient as it requires recording and sending twice as much data. This solution also produces other issues regarding browser policies.

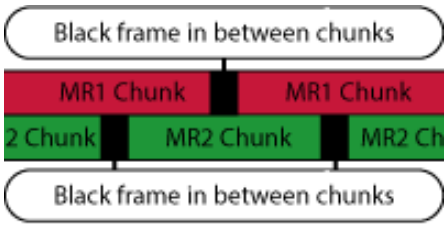

Figure 4.11: Relative locations of chunks in flow of data

4.6.1 Autoplay Policy

Web browsers are implementing stricter autoplay policies, in an attempt to enhance user experience. The autoplay policies aim at reducing the need for ad blockers, and making data consumption more efficient. This affects the proposed systems requirement for the playback of each chunk to automatically occur. Chromes autoplay policies for browser media elements are as follows [1]

- "Muted autoplay is always allowed."
- Autoplay with sound is allowed if:
  - "User has interacted with the domain (click, tap, etc.)"
  - "On desktop, the user's Media Engagement Index threshold has been crossed, meaning the user has previously played video with sound."
  - "The user has added the site to their home screen on mobile or installed the PWA on desktop."
- "Top frames can delegate autoplay permission to their iframes to allow autoplay with sound."

Furthermore, the Media Engagement Index (MEI) measures the users disposition to consume media from a particular site. Chrome calculates this as a ratio of the user's interaction with media elements under specific headings:

- "Consumption of the media (audio/video) must be greater than 7 seconds."
- "Audio must be present and unmuted."
- "Tab with video is active."
- "Size of the video (in px) must be greater than 200x140."

If the delivery method does not act in accordance with the policy, a promise is left unfulfilled or rejected when a chunk is called to play. For example, the system assigns a chunk length to be 1 second in duration. In order for a continuous feed of video to be visible, the user must interact with the broadcast page or unmute the video element each time a chunk arrives (once per second). This results in unplayable streams on the broadcast page.

4.6.2 Canvas Solution

An alternative approach utilising a canvas element to display a feed was implemented in order to avoid the aforementioned flicker and policy issues. This was achieved using the following operation.

```
canvas.getContext('2d').drawImage(videoFeed, 0, 0, canvas.width, canvas.height);
```

The drawImage method takes arguments of 'what' and 'where' to draw. In order to update the contents of the canvas the following function was called.

```
startDraw = setInterval(drawToCnvs, 33.33333333);
```

This allows the canvas to update every 33.3 milliseconds (30fps). However, this information needs to be send over a socket and more latency is introduced between received frames, resulting in an inefficient solution.

4.6.3 MSE Solution

The most appealing solution was to utilise a MediaSource object, which could be assigned to the *videoElement.src* attribute. This method allows the broadcast video element to implement a buffer to accommodate the incoming media chunks. Bearing in mind that the proposed project aims to be widely accessible, it should be noted that the MediaSource Extensions API is supported across all browsers except for Safari on iOS.

```
  mediaSource.addEventListener('sourceopen', function(e) {
    mseSourceBuffer = mediaSource.addSourceBuffer(RECORDER_MIME_TYPE);

    // Listen for errors
    mseSourceBuffer.addEventListener('error', function(e) {
      // catch error and log e.message
    });
    // Look for more bytes to append when last append completes
    mseSourceBuffer.addEventListener('updateend', function(e) {

      // buffer status
      if (mseSourceBuffer.updating) { // kill buffer and log error }

      if (mseVideo.buffered.length > 1) { // buffered has a gap }

      if (pendingAppendBytes.length > 0) { appendPendingBytes(); }
    });
  }, false);
}
```

Listing 4.11: SourceBuffer solution

On the broadcast page, the arriving chunk is converted into a Blob, which is placed into an unsigned 8bit integer array. A SourceBuffer is created to handle the stream of Blobs by appending each new arrival and removing the oldest Blob in the array. This is achieved through the *MediaSource.appendBuffer* method, which appends media chunk data from the ArrayBuffer to the SourceBuffer. This results in a perceived continuous stream of media chunks.

```
  let reader = new FileReader();
  reader.addEventListener("load", function() {
    pendingAppendBytes.push(new Uint8Array(reader.result));
    appendPendingBytes();
  });

  reader.readAsArrayBuffer(chunk);
});
```

Listing 4.12: Method to start reading the contents of a specified Blob

This method allows the broadcast to be viewed only on the condition that the viewer is on the page before the broadcast has been initiated. This method does not account for users accessing the broadcast page after the broadcast has already commenced.

The initial recorded segment for each MediaSource object contains vital information regarding track ids, codec initialisation data and timestamp offsets shown in 4.12. Each viewer needs to be present on the broadcast page to receive these initialisation segments. In order to accommodate a newly arriving viewer, the initialisation segments for each stream would need to be stored and made available to the viewer on arrival. Other issues arise regarding network usage

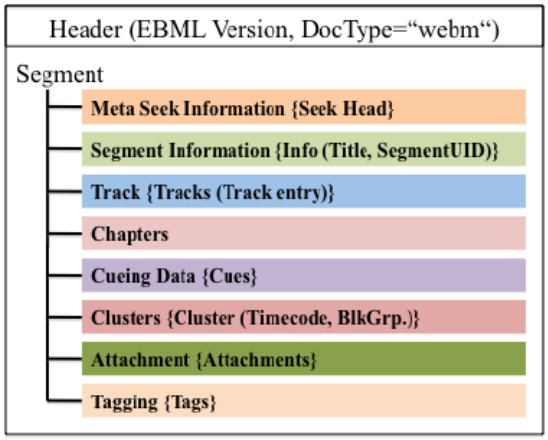

Figure 4.12: WebM Segment

and latency between the recording and delivery stages of the pipeline. Gaps between segments in the MediaBuffer results in errors on the broadcast page. A solution to overcome these gaps is to provide the media buffer with a surplus amount of media segments. This means introducing an increased delay between recording and playback of the recorded media. For example, the current solution allows each MediaSourceBuffer to contain only one media chunk at a time. This means that when the current chunk is finished playing, the next chunk needs to be present and ready for playback. As the length of each chunk is the same, any amount of time between recording and playback increases the possibility of noticeable gaps in the buffer and, as a result, introduces errors in the playback.

In order to alleviate this issue, an excess chunk should be stored in the buffer. This provides the system with an accessible chunk the moment the current chunk has been viewed. The solution is to hold the initial media segment from each stream on the server for a time equivalent

to a single segments playback length. Before the arrival of the proceeding segment, the initial segment is sent to the broadcast page. This allowed two successive segments to be available in the buffer at a given time.

4.6.4 Conclusion

The most effective method proposed utilises the MediaSource Extensions library to achieve a continuous stream of media segments. This method allows for two MediaRecoder objects with two associated video media elements to be utilised. This is a far more efficient solution to the multiple media recorder solution for a number of reasons. The video elements only need to be interacted with once in order to adhere to the autoplay policies. Furthermore, the resulting visual stream does not contain black frames or flicker. The canvas solution proved to exhibit too much latency between frames and as a result did not provide adequate user experience.

4.7 Switching Between Feeds

As previously discussed, the proposed system sends two simultaneous streams of media data to the broadcast page. In order to determine which stream is to be visible and audible at a given time, the proposed system utilises the acquired *analysedStream* information to make a decision. The decision triggers the opacity of the undesirable video element to be reduced while conversely changing the opacity of the desired video element to become visible. This is easily achieved by attaching the contents of each speakers utterance data when sending their associated media segment over the socket. Upon arrival at the broadcast page, a function compares both data variables and determines adjusts the necessary video elements opacity. Although this solution causes two streams to be simultaneously playing, it caters for two audio streams to be audible and a single video stream to be visible at a given time.

4.8 Conclusion

This discussed implementation provides a proof of concept that the proposed system can capture media, perform acceptable signal analysis on the acquired streams and utilise the data obtained to deliver a system capable of mixing media streams. The next chapter evaluates the quality of the broadcast along with the project as a whole.

Chapter 5

Evaluation

The previous chapter illustrated the implementation of the proposed design in detail. This chapter evaluates the results from tests carried out on the completed system. The evaluation is presented under three headings: Performance, accuracy of voice detection and finally an overall evaluation of the system.

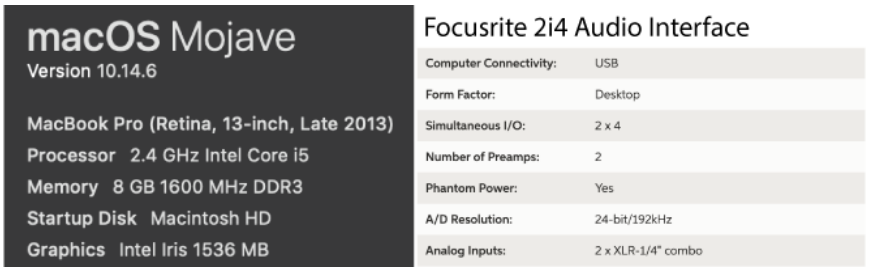

Figure 5.1: Testing hardware specifications

In order to sufficiently test the system on a public server, a HTTPS certifcate is required. The reason being, is that the method in which the system obtains a users media feed requires the *getUserMedia* function. Security concerns prevent this method from being executed unless the domain is secure. As a result, all testing was performed by running a node server on the laptop specified above in figure 5.1.

5.1 Performance

The proposed system presents a number of issues regarding performance. These issues result in latency within the pipeline. The latency impedes the system to operate predictably and as a result, affects the systems ability to broadcast a reliable stream.

5.1.1 Client Side Processing

Implementing *window.requestAnimationFrame* in conjunction with *getByteFrequencyData* to update the contents of the frequency data array poses a number of issues. According to the

WebAudio specifications “The most recent fftSize frames are used in computing the frequency data” which points to a FFT being performed with each call of the *getByteFrequencyData* function.[5] The current systems implementation performs an FFT with each animation frame. This occurs approximately every 3 milliseconds. This was calculated in the function illustrated in figure 5.1.1

```
function evtLoop(time) {
  var t0 = performance.now();

  // update array, perform frequency analysis functions

  var t1 = performance.now();
  console.log("evtLoop took", (t1 - t0), "ms");
  window.requestAnimationFrame(evtLoop);
}
```

Listing 5.1: Event loop performance test

The returned results vary from 0.3ms - 6.5ms. As large FFTs are computationally expensive, the system should regulate the size or rate at which FFTs performed.

The proposed system performs signal analysis on the client side. This requires the clients device to employ an unnecessary amount of processing power to perform the frequency analysis. A more efficient solution would be to outsource the signal processing work to the server, allowing the client to perform the recording and transport of media more efficiently. A benefit of Node is its ability to process multiple tasks asynchronously. This provides an infrastructure capable of performing frequency analysis on multiple inputs simultaneously.

In order to achieve a more efficient use of the server, a means to implement server side frequency analysis is required. A possible solution would be to capture the media segment on the client, send segment to the server for analysis and ultimately send the segment with analysis results to the broadcast page. For this to be achieved, capturing the audio in a lossless form is necessary. Implementing server side digital signal processing on node would also need to be researched further.

5.1.2 Broadcast page

Media playback on the broadcast page is significantly prone to errors. Issues arise when updating the contents of the MediaSource buffer which can cause gaps in the broadcast streams. This occurs from latency issues within the pipeline between recording, transport and delivery of the media segments. A possible cause of the latency is a slow network, which impedes the rate at which segments arrive. The current buffer allows each WebM segment (roughly 1Mb), 900ms to reach the broadcast page which should account for slow networks. If network speeds are very slow an even larger buffer size between recording and playback could be implemented.

Among the three proposed methods for detecting speech input, the formant distance and individual formant calibration methods cause the most significant issues during playback. This could be a result of the considerable amount processing being performed on the tested laptop.

5.2 Testing

In order to optimise system performance, the operations to detect speech were tested in isolation. This aims to achieve a solution which provides optimum performance while not substantially compromising the systems ability to detect voice. The hardware used to carry out the following experiments include:

- Focusrite 2i4 USB Audio-Interface
- Shure SM58 Dynamic Microphone
- t.Bone MB85 Supercardioid Dynamic Microphone
- MacBook Pro Built-in Mic
- Bose QuietComfort Inline Mic
- Budget Headphones Inline Mic
- The Box Pro DSP 115 Speaker
- Logic Pro X 10.4

5.2.1 Strength of signal

The strength of signal method only compares the ratio of frequency amplitudes within the voice band to the frequency amplitudes outside of the voice band. Because a lower FFT size can be implemented, the system's performance is increased. However, the resulting VAD capabilities are weak. By itself, the strength of signal method provides little to no differentiation between speech and non-speech. As a result, this method is only effective if the user is using microphones with precise polarity such as a *shotgun* or *lavalier* mic. Mic placement in relation to each speakers location should also be considered in order to reduce cross talk.

5.2.2 Individual Formant Calibration

When comparing both formant recognition methods, the manual individual formant calibration method provides the most effective means to encapsulate a speaker. This method is more computationally expensive than the strength of signal as it requires a higher FFT size. However, it provides the user with the most personalised application. Due to the varying nature of the human voice across speakers, this method is the most adaptable and as a result the most desirable.

In order to reduce the rate at which the computer performs an FFT, the interval between iterations of *evtLoop* was increased. This greatly improves the systems performance. The lowest FFT size that still produced acceptable results allowed for 1024 frequency bins, each having a bandwidth of roughly 43Hz.

5.2.3 Manual Formant Distance Calibration

Similarly, the manual distance calibration method requires a relatively larger FFT than the signal strength method, as it needs to analyse the frequency data more precisely. Though this method allows for better automatic recognition than other methods, the manual distance calibration method is more accurate. The benefit of the distance method is that the user only needs to manipulate 2 UI slider element handles in order to calibrate the system (see figure 4.8) as opposed to the 6 individual formant handles (see figure 4.7).

5.2.4 Conclusion

From testing each of the frequency data analysis methods, a decision was made to regulate rate at which *evtLoop* was fired. This was achieved by monitoring the time that had passed between each iteration of the *evtLoop* function.

```
// initialize the timer variables and starts evtLoop
function initEvtLoopOne(fps) {
    evtOneInterval = 1000 / fps;
    evtOneThen = Date.now();
    evtOneStartTime = evtOneThen;
    evtLoopFeedOne();
}

function evtLoopFeedOne() {
  // the animation loop calculates time elapsed since the last loop
  // and only draws if specified fps interval is achieved

  // if true request another frame
  if(stop == false){
    window.requestAnimationFrame(evtLoopFeedOne);

    // perform signal analysis functions

    // calc elapsed time since last loop
    evtOneNow = Date.now();
    evtOneElapsed = evtOneNow - evtOneThen;

    // if enough time has elapsed, draw the next frame
    if (evtOneElapsed > evtOneInterval) {

      // Get ready for next frame by setting then = now
      evtOneThen = evtOneNow - (evtOneElapsed % evtOneInterval);
    }
  } else {
  window.requestAnimationFrame(evtLoopFeedOne)
};
```

Listing 5.2: Improved evtLoop method

This allows the system to be tailored to different machine capabilities. By adjusting the frame rate when initialising the *evtLoop*, the rate at which each operation is performed can be increased or decreased to suit the users machine. Although the individual formant calibration method

provides more accurate results, the formant distance calibration method requires less effort to achieve adequate results.

5.3 User Experience

The following tests were conducted in order to determine the systems frequency analysis capabilities using various media capture apparatus. Each test was conducted utilising varied audio input devices.

5.3.1 Frequency Analysis Test

The following test shows an objective analysis of the input frequency array when the system is used within in a target users environment. The test utilises a variety of audio capture equipment, ranging from cheap (Budget Earphones Inline Mic, Macbook Built-in Mic) to mid-range/expensive (Shure SM58, t.Bone MB). The aim of this test is to measure the systems ability to analyse frequencies captured from the audio input. This is calculated with the following functions.

```
function getFrequencyValue(frequency, context, freqDomain) {
  var nyquist = context.sampleRate/2;
  var index = Math.round(frequency/nyquist * freqDomain.length);
  var arr = [index, freqDomain[index]]
  return arr;
}
function getPeak(frequencyData){
  let peak = 0;
  for (let i = 0; i < frequencyData.length; i++){
    if(frequencyData[i] > peak){
      peak = i;
    }
  }
  return peak
}
// firstTestControl = getFrequencyValue(329.63, audioCxtA, freqDataA);
// firstTestCaptured = getPeak(freqDataA);
console.log(firstTestControl, firstTestCaptured);
```

Listing 5.3: Functions to compare capture devices

This test assumes the user to be recording in a carpeted room with no background chatter or noise. The systems capturing sample rate is 44.1kHz with an FFT size of 1024. This works out at a 43.06Hz bandwidth for each frequency bin. Both inline mics were tested simultaneously. SM58 and t.Bone mics were later tested simultaneously in the same setting. Finally the Macbook mic was tested. Each input device recorded a number of sine wave tones across 329Hz - 3000Hz. The tones were produced using Logic Pro's Test Oscillator plugin. The frequency data arrays of each input device were examined to monitor variation across devices. The results illustrate which frequency bin the system calculated to have the highest frequency amplitudes.

Test Tone:	Bin	SM58	t.Bone	Bose Inline	Budget Inline	Macbook Mic
329.63Hz:	8	8	8	9	9	8
392.00Hz:	9	9	10	10	9	10
440.00Hz:	10	10	12	11	11	12
1000.00Hz:	23	27	25	26	27	27
1500.00Hz:	35	37	34	38	34	36
2500.00Hz:	58	60	60	58	55	60
3000.00Hz:	70	71	71	71	72	71

Table 5.1: Captured frequency test

5.3.2 Conclusions

From the results gathered (Table 5.1), variation can be seen between capture devices. This outlines the current systems need for adaptability to suit various users. For this reason, it is important for the system to incorporate a margin for error when determining voice detection. The results show the need for a margin of +/- 5. However, other capture devices may require a wider margin.

5.3.3 Dialogue Test

A test to examine the systems capability at recognising speech across various speakers was performed using the same apparatus. The audio was captured at 44.1kHz sample rate using the SM58, t.Bone, Bose inline mic, budget inline mic and the built in Macbook mic. Both inline mics were tested simultaneously in a controlled setting (carpeted room, no background chatter or noise, mouth 8cm from mic). The SM58 and t.Bone mics were tested simultaneously and finally the Macbook mic was tested under the same parameters. The speaker read the words 'heed, hid, head, had, hod, hawed, hood, who'd' in succession. Each reading was repeated 5 times. The aim of the test is to capture a varied range of vowel shapes. A green box on screen notified the user if the system deemed the utterance as a voice. The results were averaged and the systems accuracy was calculated. The tests were carried out over a group of 10 speakers comprised of both male and female aged 20 - 60. The tests were performed on both formant detection methods

Word	SM58	t.Bone	Bose Inline	Budget Inline	Macbook Mic
Heed	82%	84%	88%	82%	88%
Hid	84%	88%	90%	88%	92%
Head	88%	90%	82%	82%	90%
Had	92%	90%	90%	84%	90%
Hod	90%	84%	82%	84%	88%
Hawed	88%	84%	90%	82%	90%
Hood	88%	84%	84%	90%	90%
Who'd	84%	90%	88%	82%	88%

Table 5.2: Voice Detection: Manual Formant Distance Calibration

Word	SM58	t.Bone	Bose Inline	Budget Inline	Macbook Mic
Heed	90%	88%	90%	82%	88%
Hid	82%	84%	88%	88%	80%
Head	80%	82%	80%	78%	82%
Had	90%	84%	84%	82%	88%
Hod	90%	90%	82%	82%	84%
Hawed	88%	84%	94%	84%	88%
Hood	92%	90%	88%	90%	88%
Who'd	82%	84%	80%	84%	88%

Table 5.3: Voice Detection: Manual Individual Formant Calibration

5.3.4 Conclusion

Both dialogue tests showed promising results across all devices for capturing the voice. However, the tests do not consider the systems capability to discriminate between a voice and common noises within a similar band, such as a door closing or a dog barking.

5.3.5 Noise Test

A test to examine the systems ability to disambiguate noise from voice was tested in the same environment as the previous tests. In each setting, a speaker was placed 8cm from the capture device. White noise, pink noise and a sweeping test tone from 20Hz to 20kHz were played through the speaker individually. More complex sound forms were accounted for by testing a guitar and piano as they produce a waveform with multiple peaks. To capture the piano and guitar, the capture device was placed 8cm from the origin of the instruments sound.

Noise	SM58	t.Bone	Bose Inline	Budget Inline	Macbook Mic
Pink Noise	false	false	false	false	false
White Noise	false	false	false	false	false
Test tone	false	false	false	false	false
Piano	true	true	false	false	true
Guitar	true	true	false	false	true

Table 5.4: Noise Detection: Manual Formant Distance Calibration

Noise	SM58	t.Bone	Bose Inline	Budget Inline	Macbook Mic
Pink Noise	false	false	false	false	false
White Noise	false	false	false	false	false
Test tone	false	false	false	false	false
Piano	true	true	false	false	true
Guitar	true	true	false	false	true

Table 5.5: Noise Detection: Manual Individual Formant Calibration

5.3.6 Conclusions

The noise test results show that the system is not able to differentiate between certain sounds and the human voice. A noise with multiple distinct peaks in the frequency spectrum is falsely recognised as a human voice. Conversely, a sound with distributed power across the whole spectrum, or a sound that exhibits a single peak at a given time is disregarded. This is true for both proposed formant identification methods.

Chapter 6

Conclusion

6.1 Concluding Statement

This project implemented audio signal processing techniques to discern a speaker of interest for automatic visual mixing. Through various experiments, the conclusions are as follows: Firstly, in order to accommodate a wide variety of speakers, an adaptable system needs to configured. This can be accomplished through incorporating machine learning capabilities to provide a more robust system. Secondly, due to the computational cost to perform sufficient processing of an audio signal, a server side solution is recommended. Finally, the proposed system provides evidence that a web based system, capable of intelligently switching between two speakers is achievable using rudimentary audio analysis.

6.2 Future Work

By implementing other methods to monitor subject and environmental factors it is possible to further develop the proposed system. Advances in mobile device sensors provide the public with access to ambient light meters and proximity sensors. By incorporating these various technologies it is possible to consider lighting, motion and location when implementing a camera switching system. The proposed technology can also be expanded to reach live entertainment and security industries. The proposed system provides a means to manipulate a visual feed based on an audio input. This can provide establishments with sound activated cameras or a live performer with a dynamic visual projection system capable of reacting differently based on the audio input data.

Bibliography

[1] Autoplay policy changes, google developers — developers.google.com.

[2] Custom web mobile apps for broadcasting, medical, and more — webrtc.ventures/.

[3] Loopback: Cable-free audio routing for mac — rogueamoeba.com/loopback.

[4] Open source broadcasting studio (obs) — obsproject.com.

[5] Web audio api — w3.org/tr/webaudio, Sep 2018.

[6] AminArria. Aminarria/janusv_erlang_wrapper_demo.

[7] B. S. Atal, S. L. H. Speech analysis and synthesis by linear prediction of the speech wave. *The Journal of the Acoustical Society of America 23*, 12 (2015), 2238–2245.

[8] Christel, M. G., Smith, M. A., Taylor, C. R., and Winkler, D. B. Evolving video skims into useful multimedia abstractions. *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI 98* (1998).

[9] Daniel Jurafsky, J. H. M. *Speech and Language Processing: An introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, vol. 23. IEEE, California, 2015.

[10] Fatima, N., and Zheng, T. F. Vowel-category based short utterance speaker recognition. *2012 International Conference on Systems and Informatics (ICSAI2012)* (2012).

[11] Girgensohn, A., Boreczky, J., Chiu, P., Doherty, J., Foote, J., Golovchinsky, G., Uchihashi, S., and Wilcox, L. A semi-automatic approach to home video editing. *Proceedings of the 13th annual ACM symposium on User interface software and technology - UIST 00* (2000).

[12] Hawkins, S., and Midgley, J. Formant frequencies of rp monophthongs in four age groups of speakers. *Journal of the International Phonetic Association 35*, 2 (2005), 183–199.

[13] In-Chul Yoo, Hyeontaek Lim, D. Y. Formant-based robust voice activity detection. *IEEE/ACM Transactions on Audio, Speech, and Language Processing 23*, 12 (2015).

[14] Inoue, T., Okada, K.-I., and Matsushita, Y. Learning from tv programs: application of tv presentation to a videoconferencing system. *Proceedings of the 8th annual ACM symposium on User interface and software technology - UIST 95* (1995).

[15] Jenkin, T., McGeachie, J., Fono, D., and Vertegaal, R. Eyeview: Focus+context views for large group video conferences. pp. 1497–1500.

[16] Kelleher, L. Article on podcast consumption in ireland, Jun 2019.

[17] Kovacevic, B., Milosavljevic, M. M., Mladen, V., and Milan, M. *Robust Digital Processing of Speech Signals*. Springer International Publishing, 2018.

[18] Kurento. Multistream in webrtcendpoint? · issue 62 · kurento/bugtracker.

[19] Lammert, A., and Narayanan, S. On instantaneous vocal tract length estimation from formant frequencies.

[20] Leake, M., Davis, A., Truong, A., and Agrawala, M. Computational video editing for dialogue-driven scenes. *ACM Transactions on Graphics 36*, 4 (2017), 1–14.

[21] Liu, Q., Rui, Y., Gupta, A., and Cadiz, J. J. Automating camera management for lecture room environments. *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI 01* (2001).

[22] Muaz-Khan. muaz-khan/webrtc-experiment, May 2019.

[23] Ojamaa, A., and Düüna, K. Assessing the security of node.js platform. In *2012 International Conference for Internet Technology and Secured Transactions* (Dec 2012), pp. 348–355.

[24] Oppenheim, A. V. Speech spectrograms using the fast fourier transform. *IEEE Spectrum 7*, 8 (1970), 57–62.

[25] Park, C. Consonant landmark detection for speech recognition.

[26] Rabiner, S. *Digital Processing of Speech signals*, vol. 23. IEEE, California, 2015.

[27] Research, E. Edison research, Sep 2019.

[28] Rui, Y., Gupta, A., and Cadiz, J. J. Viewing meeting captured by an omni-directional camera. *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI 01* (2001).

[29] Shipman, F., Girgensohn, A., and Wilcox, L. Hyper-hitchcock: Towards the easy authoring of interactive video.

[30] Smith, M. A., and Kanade, T. Video skimming and characterization through the combination of image and language understanding techniques.

[31] Takemae, Y., Otsuka, K., and Yamato, J. Automatic video editing system using stereo-based head tracking for multiparty conversation. In *CHI '05 Extended Abstracts on Human Factors in Computing Systems* (New York, NY, USA, 2005), CHI EA '05, ACM, pp. 1817–1820.

[32] Ueda, H., Miyatake, T., and Yoshizawa, S. Impact. *Proceedings of the SIGCHI conference on Human factors in computing systems Reaching through technology - CHI 91* (1991).

[33] Vertegaal, R., Weevers, I., Sohn, C., and Cheung, C. Gaze-2. *Proceedings of the conference on Human factors in computing systems - CHI 03* (2003).

[34] Welling, L., and Ney, H. Formant estimation for speech recognition. *IEEE Transactions on Speech and Audio Processing 6*, 1 (1998), 36–48.