Department of Computer Science and Engineering
University of Dhaka
14th June 2021

Audio Visualizer

Fundamentals of Programming Lab(CSE 1211)

Team Members:

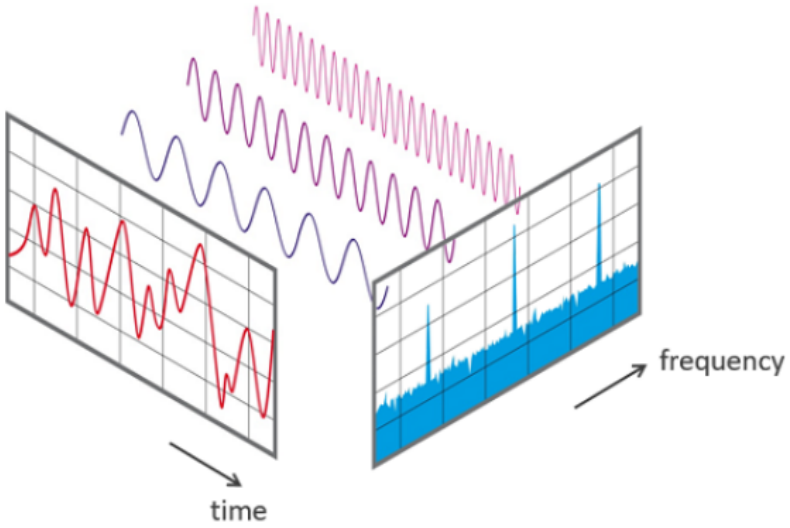
Yeamin Kaiser, Roll: 01

Mahadi Hasan, Roll: 03

Bhola Nath Das, Roll: 22

Introduction

Our project in essence is an audio visualizer that plays a wav file and divides the incoming audio stream into very short time slices called samples, and visualizes the signal in two different forms. The "Wave" visualization is simply the time-domain graph of the current stream. The "Bars" mode on the other hand does a Fourier Transform on each slice, extracting the frequency components, and updates the visual display using the frequency information.



Objectives

Our objectives were simple enough we wanted to make a robust audio visualizing application. It had to be able to take a stream of audio data and convert it into its simplest form for us to be able to use that information to do as we like. We also desired to have 2 modes in which we visualize the audio data, the time-domain mode and the frequency mode. It was also our plan to have this application run on as many platforms as could be possible, Currently to our knowledge it runs on most Unix operating systems and we have also made it so that it is possible to run on the windows 10 operating system. We have also strived to provide a method through which one may acquire wav files, since they are not the norm in music listening. As such we have made it possible to download wav files from YouTube eliminating, having the user go through the process of having to find it themselves.

Features

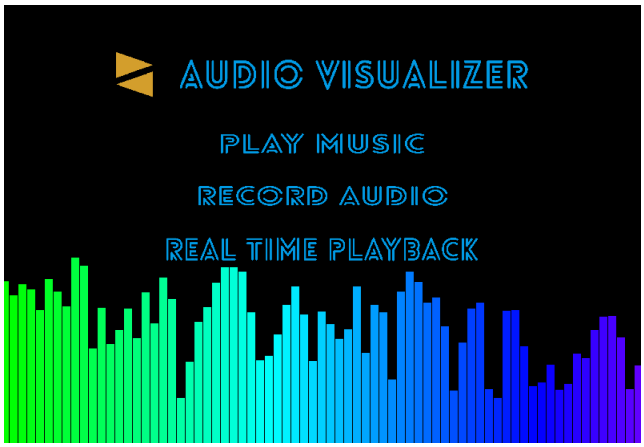
We have tried to incorporate multiple features which utilize our main objective into our application.

Command Line Interface:

For users who are accustomed to using CLI and does not prefer GUIs we have implemented this feature. This allows the user to run the whole application from the command line without ever having to worry about prompting the GUI.

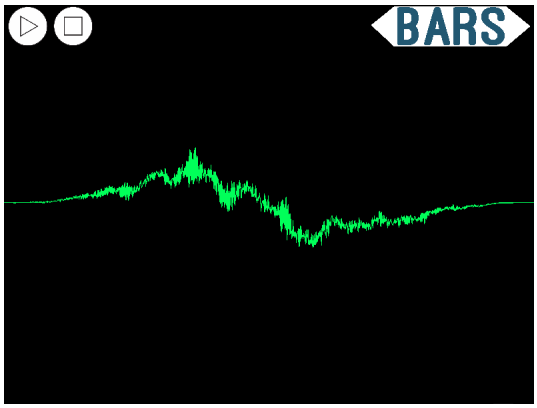
Graphic User Interface:

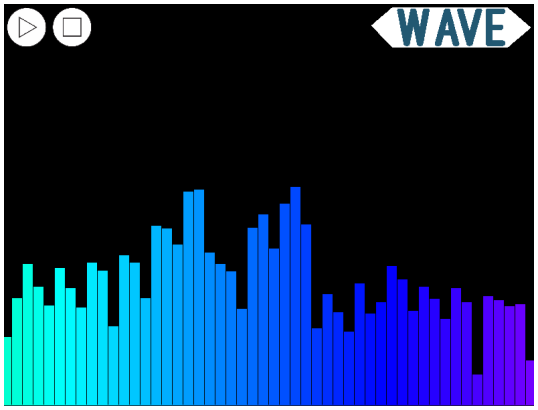
For users who require a GUI, we have implemented an attractive and feature-rich GUI. It provides the users with multiple options to interact with our application. This makes our application convenient for users who have no previous experience using command line interfaces.



Music Player:

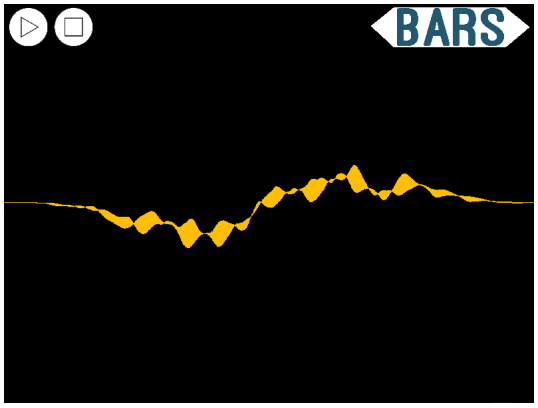
Our application can play and visualize any WAV file of any length using our custom callback functions without distorting the actual audio data.





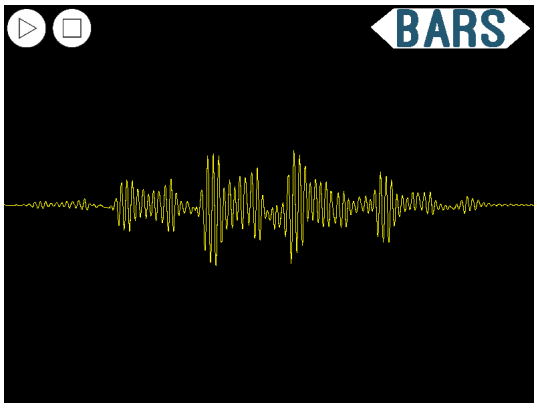
Audio Recorder:

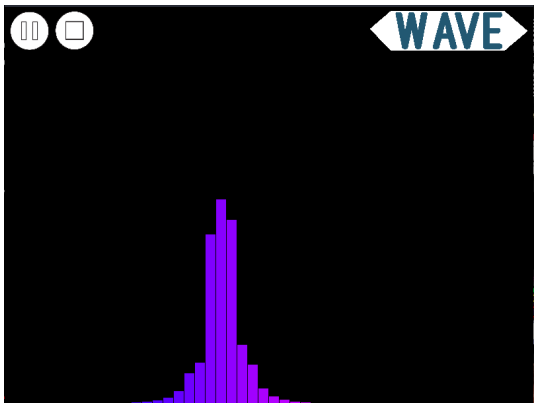
The user can record his/her own audio and then hear and visualize it. This feature allows the user to store his audio as not only an audio, but also with a visual representation of the user's own voice.



Real Time Audio Visualizer:

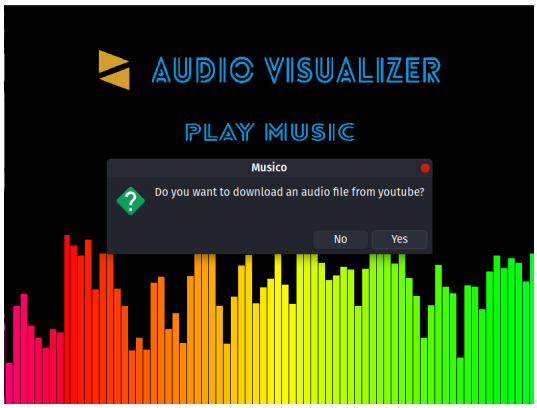
This feature lets the user see his own voice as an actual wave in real time. More so, the user can also see how large of frequency range is encompassed by his/her voice. This feature does this with negligible intermediate delay.

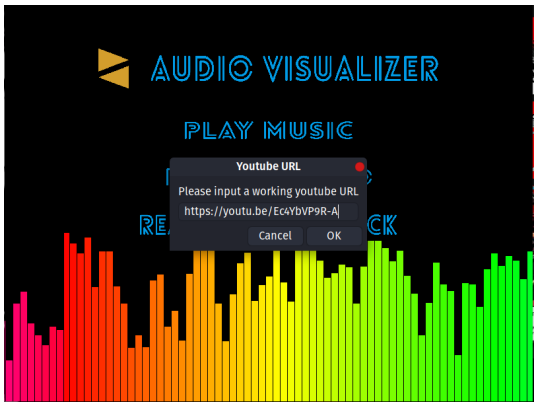




YouTube-dl Integration:

As the SDL language allows customization of audio playing with only WAV files, the user needs their audio files in that format. But for our application, the user does not need any previously owned WAV files. The user can download the audio file of any video present on the online platform, YouTube, using this feature. Our application downloads that audio file and converts it into a usable WAV file which the user can later play and enjoy.





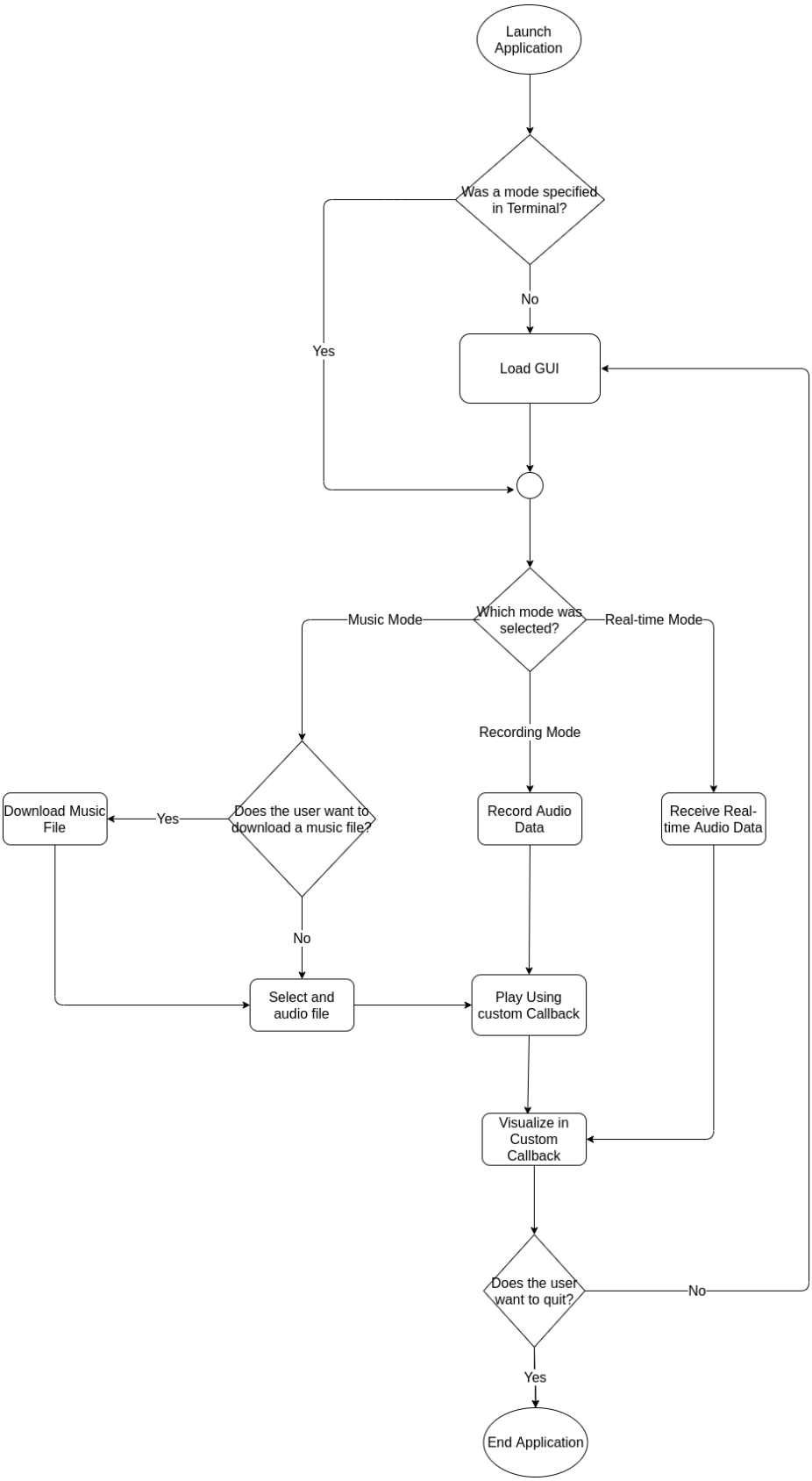
Dual Visualizer Mode:

After accessing audio data from the user our application can not only visualize the actual audio wave in a time domain mode as it actually exists, but also split it into multiple component frequency waves in the frequency domain mode and visualize the amplitude of these frequency wave in their respective ranges.

Cross Platform:

We have made it possible for the application to be compatible on multiple platforms. Through our testing we can confidently say it runs on most Unix Operating systems and the Windows Operating system.

Modules


Launch Application
Was a mode specified in Terminal?
No
Yes
Load GUI
Which mode was selected?
Music Mode
Real-time Mode
Recording Mode
Does the user want to download a music file?
Yes
Download Music File
No
Record Audio Data
Receive Real-time Audio Data
Select and audio file
Play Using custom Callback
Visualize in Custom Callback
Does the user want to quit?
No
Yes
End Application

Our code base relies on the following custom headers.

`preprocessor.h`

In this header, we have included all the C, SDL and the fftw3 libraries that we require. We used preprocessor directives to use maximum and minimum functionality and constant value assignemnt to be used throughout the codebase. We also enumerated different states of the program to further simplify the development process for anyone who is likely to contribute.

`utils.h`

We have declared extern variables that we need to use across other modules. The functions to start the application, change of use mode and quitting the program are declared. The three main use mode of our program works via corresponding functions that are also declared here. Similarly, anyone wanting to add their own different layer of features should add to corresponding functions in this particular header.

`audio_struct.h`

This header contains custom structs that contain information about the stored audio. The WavData structure holds the required audio information from an wav file to be loaded. The RecordedData is used to store data recorded from the input device and thus contains a bit more information as the buffer size is prefixed.

`audio_data.h`

The key function used in different parts of our program to access the audio data in buffers and sync the corresponding visuals with it is declared in this very header, namely the Callback() functions and the Get16bitAudioSample() function.

*Callback()

The SDL callback signature is a stand-alone function whose return type and parameters are fixed. This function can be specified as a member of different SDL Audio structures. The Callback() function will be repeatedly called in very small time intervals to interact with the audio driver. The second parameter of this function, Uint8 *stream, holds information of the current audio buffer which we pass to our visualizerOutput() function. This passing method is the same in all of our use modes of the program.

In the music mode, we copy the audio buffer from the playback device to the stream variable to be visualized and decrease the byte size of the playing audio. Whenever the music ends or equivalently the byte size of the audio reaches 0 we return.

In the recording mode, similarly we copy buffer from the recording device to the stream until the user stops the recording or 600 seconds are passed. This time we just save this data to be visualized later similar to how we did in music mode.

Lastly in the real time mode, we do the same thing but visualize the stream in sync.

Get16bitAudioSample()

We took the current buffer of the audio data and converted it to represent relative amplitudes in the interval [0, 1] and then fed it to the visualizer function.

As a detailed explanation:
The current audio stream is basically an array of unsigned 8-bit integers, each of which represent the "loudness" or more precisely the amplitudes of the sound. But Stereo audio samples are stored in a "LRLRLR" ordering so we combined adjacent values into a single 16-bit integer.
To achieve this, we first checked the byte order of the buffer and applied simple bitwise operations. To map it to a fraction value as required, we divided the value by the maximum of values it can take.

`audio.h`

Here we prepare to utilize input audio data using SDL audio library functionalities. It is expected that functions that could manipulate the audio data further should be in this header. For our part, we have declared some required SDL variables and functions that handle basic recording, playing and can set default audio specs.

`visual_struct.h`

The visualization consists of color and complex number values. In this header, we decided to create struct named complexData having input and output variables and other necessary members required to visualize the data. Furthermore, to ease the use of colors we mapped the much used HSV format to RGB using a constructor in our custom made RGB struct. If someone wants to visualize our audio data in a different color scheme they only need to modify the constructor.

`visual_data.h`

There are countless ways to visualize an audio data, from eye catching fireballs to more statistics oriented ones like our frequency mode. So we decided to have a dedicated header for this particular task. Anyone who wants to visualize a stream of audio data in any way possible to imagine, can directly use this header file. For instance, our visualizerOutput() function can be a reference on how to achieve this.

visualizerOutput()

This is the main function that visualizes our audio data stream by stream. The data is stored in an array of **complexData** which is a custom struct. If we are in the **time-domain mode** we present this data as the time-domain graph of the current stream. Otherwise we apply the Discrete Fourier Transform to extract the frequency components present in the current stream.

To elaborate more on this:
After receiving the audio data, we smooth it out using the Hann Windowing Function, which we visualize directly in the time-domain mode.
And in the frequency mode we perform a Fourier Transform on the audio data. The Fourier transform is a mathematical operation that is based on the premise that any real (or complex) valued function can be represented as a linear combination of sinusoids. The Fourier transform can transform can extract the coefficients of the sinusoids that compose any real-valued function. Each coefficient can be represented as:

$$X(\omega) = \int_{-\infty}^{\infty} f(t) e^{-i\omega t} dt$$

$f(t)$ represents the time domain signal

$X(\omega)$ represents the complex number output

A Discrete Fourier transfrom(DFT) approximates this at N different intervals. For a time domain signal of length N, the kth coefficient can be expressed as:

$$X[k] = \sum_{n=0}^{N-1} f[n] e^{\frac{2\pi i}{N} n}$$

The $f[i]$ are thought of as the values of a function, or signal, at equally spaced times $t = 0, 1, \ldots, N-1$. The output $X[k]$ is a complex number which encodes the amplitude and phase of a sinusoidal wave with frequency $\frac{k}{N}$ cycles per time unit. The effect of computing the $X[k]$ is to find the coefficients of an approximation of the signal by a linear combination of such waves.

Instead of storing the coefficients directly in an array, what is often done (as was done in our project) was to store the magnitudes of each coefficient $|X[k]|$. This is slightly more space efficient because it can be shown that $X[k]$ and $X[N-1-k]$ are complex conjugates, which further implies that they have the same magnitude. Therefore, you only need to store 50% of the actual computed magnitudes if you don't care about the complex component of each coefficient. In our project, we use these magnitudes to determine the state of the music that is playing.

`visual.h`

This header is to load all graphical elements of the program. Fonts, Sufaces, Textures and other variables are declared here. Our UI is entirely dependent on this particular header, that is customize-able to any extent imaginable.

We have both terminal and graphic user interfaces in our application.

The terminal usage works with the help of the of the function getopt() from the unistd.h library. This is a C library function which can parse arguments from command line. It was initially used for testing purposes since we didn't have GUI working. But we decided to keep it as it was for those who prefer CLI.

We tried to make an attractive graphic user interface using both images and text rendered using the SDL_ttf library. The interface was built on clickable buttons which worked on the basis of custom-made circular and rectangular intersection functions. The buttons based which state of the application the user is in.

Team Member Responsibilities

The idea of making an application that visualizes audio data was conceptualized by Mahadi Hasan and our team leader, Bhola Nath Das. All three members contributed in the coding the source code of our application. The terminal uses were added by Mahadi Hasan and Yeamin Kaiser. Mahadi Hasan coded the audio_struct.h and audio_data.h module. The audio.h header was jointly done by Bhola Nath Das and Mahadi Hasan. The visualizer modules consisting of visual_struct.h and visual_data.h was done by Yeamin Kaiser. Bhola Nath Das created the Graphical User Interface and the main control flow by coding the visual.h and utlis.h header files. Other than the coding parts, all three members had to do extensive research to figure out how each module would work and how to tie it all together into one single coherent application.

Platform, Library & Tools

Platform:

Our application runs on a variety of platforms. To our current knowledge it runs well on most Unix based systems and windows.

Library:

We have used open source tools such as SDL for providing a robust library through which the core parts of our applications runs. We have also used the fftw3 library in our visual processing modules. We have also used tinyfiledialogs library to integrate OS style file selection.

Tools:

The only tool we have used as executable is an open source program known as YouTube-dl. It can download files from YouTube and convert them to any format of our liking.

Limitations

Being inefficient on memory usage and lack of multi-threading are the main drawbacks of our program. What we fail to achieve is being able to edit and manipulate audio data similar to how is done in full-fledged applications. But to do that, we would require much more modern APIs so decided not move that way.

We store the audio data in memory in its RAW format requiring a lot of memory as the stream continues. For this reason, we decided to limit our recording capacity at 600s for optimum balance. Real-time visualization also suffers from this issue as memory is constantly allocated during the process. We integrated youtube-dl by calling a system script because forking youtube-dl's python based code to our program is beyond our ability now. This is why, the program being single threaded, when youtube-dl script runs for a substantial amount time, the main UI stops and waits for the script to complete. But this behavior causes unresponsive UI on both unix and win32 platforms as they constantly check if the the main thread is alive.

Window re-sizable feature causes problem under Windows host due to an OpenGL issue mentioed in our github repository.

Conclusions

Initially we had a glimpse of what we are trying to achieve but we had no idea about Digital Audio and Signal Processing. We assumed that Fourier Transformation was sufficient knowledge for us to start but eventually we realized the project required quite extensive knowledge on Digital audio, how it is stored, how SDL handles audio, how to map audio data to feed as inputs for DFT and some other implementation tricks. Given the internet resources mostly use modern APIs, since SDL is a very old library, we we had to dig deeper to find what we actually needed and in some cases we had to figure it out ourselves through hours trial and error.

Future plan

Our future plan with this application is to make it a full fledged audio software for all sorts of editing purposes along with many other uses. We believe, that audio visualization could be used as tool for security purposes and educational application. Though our application currently works on desktop operating systems. We plan to give it a wider range by enabling cross platform usage in systems like Android and iOS.

Repositories: Github , YouTube Link

References

[1] The SDL Library.
https://www.libsdl.org/

[2] The SDL wiki.
https://wiki.libsdl.org/

[3] The fftw3 Library.
http://www.fftw.org/

[4] The basics of digital audio
https://youtu.be/9z5HOZ4C1KY

[5] Information on the discrete fourier transform.
https://en.wikipedia.org/wiki/Discrete_Fourier_transform

[6] Many articles from Stack Overflow.
https://stackoverflow.com/

[7] Tiny File Dialog.
https://github.com/native-toolkit/tinyfiledialogs

[8] Youtube-dl
https://youtube-dl.org/

[9] Typesetting was done on Overleaf
https://www.overleaf.com/read/mgcpchtcjhdv