UNIVERSITY OF
WEST LONDON

TC60077E
Sound Engineering Project

Image Sonification Synthesis

BSc (Hons) Sound Engineering

21440503

Abstract

This project asks if a novel sound synthesis implementation that harnesses image sonification can be created. To achieve this, Python was used to create software, ‘main.py’, that raster scanned an image into a one-dimensional array and processed it into audio. Convolution and an oscillator were incorporated to make a synthesis framework that could create new sounds from digital image input. Taking this further, a second Python program was created to facilitate sonification and saving to file, ‘imagesave.py’. This was used in combination with GNU Image Manipulation Program (GIMP) and Ableton Live to apply image transformations and observe their effects on the resultant audio. The outcome of combining ‘imagesave.py’ with GIMP and Ableton Live shows interesting results when image transformations such as rotation, reflection or translation are applied to images that represent basic audio waveforms. These image manipulation techniques could have implications for sound synthesis and waveform manipulation if expanded upon. Further research could explore 16-bit audio compatibility, more image effects, and a wider selection of basic waveform images to perform investigations with. This stronger base of knowledge would lead to a superior sonification implementation for sound synthesis.

Acknowledgements

Firstly, I would like to thank my project supervisor Eugenio Donati for keeping me on track throughout the course of the project and helping me manage my expectations appropriately. His commitment to being helpful and available has aided my research project and implementation greatly. I would like to thank other members of the Sound Engineering BSc: Christos Chousidis, for encouraging me from the earliest stages of the idea and pushing me so I could reach a strong concept that would support my final project; Nino Auricchio, for helping me develop valuable skills that have stood me in great stead for this research project; and my classmates, for their encouragement, friendship, and support throughout the three years we have spent together. Thanks also to Jamie Newman for being a great sounding board for ideas.

Table of Contents

Table of Figures

List of Equations

List of Tables

Nomenclature

AD	Auditory Display
CC	Creative Commons
DAW	Digital Audio Workstation
DSP	Digital Signal Processing
FFT	Fast Fourier Transform
FM	Frequency Modulation
FVC2002	Fingerprint Verification Competition 2002
GIMP	GNU Image Manipulation Program
GNU	‘GNU’s Not Unix’ (recursive)
GUI	Graphical User Interface
OOP	Object-Oriented Programming
PIL	Python Image Library
PMSon	Parameter Mapping Sonification
PNG	Portable Network Graphics
RGB	Red Green Blue
USB	Universal Serial Bus
WAV	Waveform Audio File Format (shorthand)

1 Introduction

Solid State Logic, Universal Audio, and Neve are among other audio companies who have spent the last quarter-century digitising famous hardware into software emulations that can be opened within a digital audio workstation (DAW). Although analogue audio equipment will always be the gold standard, software is the affordable alternative. It opens music production, sound synthesis, and mixing & mastering possibilities to a whole new generation of artists and engineers who can't justify the money, space, and energy costs associated with owning analogue studio equipment. Although software is often used to try and emulate an analogue method in the name of replicating 'that sound', it also allows for a new way of treating audio: purely as the ones and zeroes used to store and represent them. This idea allows for a broader concept of what could be considered digital audio. It exists as a string of numbers within a specified value range, played back one after the other at an audio-rate speed (i.e., 48kHz). This implies that other forms of data stored digitally could be played back as audio if processed appropriately.

The creation of this research project is a result of recently discovered reading on the topic of image sonification, notably: 'Voice of Sisyphus' (McGee, Dickinson and Legrady, 2012); 'Raster Scanning: A New Approach to Image Sonification, Sound Visualization, Sound Analysis And Synthesis' (Yeo and Berger, 2006); and 'The Sonification Handbook' (Hermann, Hunt and Neuhoff, 2011). This reading introduced the research field of auditory display (AD), described as an examination of how the human auditory system can be used as the primary interface channel for communicating and transmitting information. Image sonification is described as the technique of rendering sound in response to data and interactions, a core component of auditory display (Hermann, Hunt and Neuhoff, 2011). As described in The Sonification Handbook, sonification is used across a broad range of fields including chaos theory; biomedicine; data mining; seismology; and accessible interface design to name a few. Some of the core disciplines that contribute to sonification research include physics, acoustics, psychoacoustics, sound engineering, and computer science. From this initial reading, a project proposal was created. The research question posed:

How can image sonification be harnessed to create a novel software implementation for sound synthesis?

2 Project Development and Methodology

2.1 Proposal Stage

The project aimed to produce a sound synthesis implementation that would use digital image as a form of input and explore image sonification applied to sound synthesis, using the implementation as a tool to do so. Objectives were created to achieve this: create a way for the user to generate an image to be sonified; write code that can read and store the pixel data in one dimension; process the data into a format that allows for typical digital audio signal processing conventions and techniques to be employed; design a system that facilitates sound synthesis through manipulation of the sonified image signal; and build on existing research using elements of the implementation in combination with other software.

From initial reading, the concept of raster scanning formed the basis of the idea for an image sonification implementation. Raster scanning is originally a technique for generating or recording video image by means of line-by-line sweep (Yeo and Berger, 2006). It is a data mapping between one and two-dimensional spaces. Existing research has shown how this data mapping can be applied to sonification, specifically with image. The pixels in a time-independent, two-dimensional image can be mapped into one dimension (Hermann, Hunt and Neuhoff, 2011). A raster scan commonly starts on the top-left pixel of the image and scans through every row top-to-bottom in alternating directions. This one-dimensional representation of the image's pixel data can be amplitude-scaled and read at audio rate, treating each pixel's value as an audio sample. As an example, a 400x400 single-channel greyscale image is made up of 160,000 pixels, each with an associated discrete value. Played back at 48kHz, this sonified data would have a duration of 160,000/48,000, or 3.33 seconds. This is the fundamental method that underpins this research project.

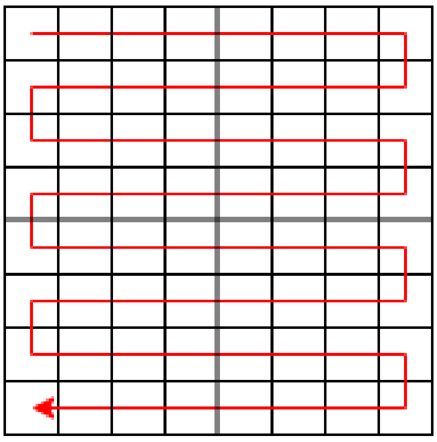


Figure 1 - Bidirectional raster scan diagram

The literature review showed that sonification has many classifications, such as Parameter Mapping Sonification (PMSon), audification, and model-based sonification. It also displayed that image or any other data sonification was not broadly available within the market of consumer audio production. Some examples such as MetaSynth and StatSonVST exist for consumers, but nothing that employed raster scanning to sonify data. The literature review also searched for a creative application to help potential users relate with the tool. Biometric data, specifically a fingerprint image, was chosen to connect users to the tool that they would be using by having them input their own fingerprint to be sonified. Combining biometric data with image sonification to achieve sound synthesis is the concept that formed from the literature review. This project would produce a software prototype, or proof-of-concept, for something that could potentially be developed into a digital audio tool for consumer or professional audio purposes.

2.2 Artefact Build

2.2.1 Initial Build with Hardware

The initial hardware design for the artefact used an Adafruit 4690 fingerprint sensor; connected to an ESP32 S2 Feather microcontroller running CircuitPython (CircuitPython, 2023), a framework built on Python designed for microcontroller functionality; and a 2013 MacBook Air running Mu Editor, connected to the microcontroller via Universal Serial Bus (USB). The objective was to have the sensor scan a user's fingerprint using a Python file on board the microcontroller, then transmit the image across USB. The MacBook would then receive the image in a Python file that handles processing, sonifying, and synthesising sound.

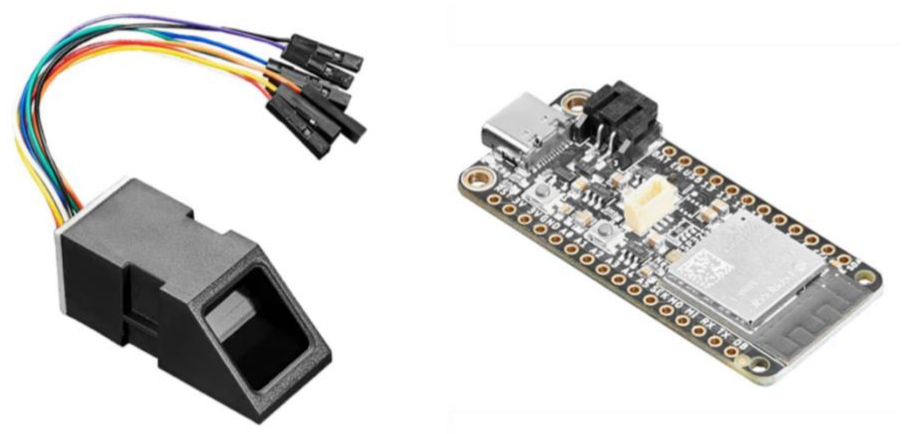


Figure 2 - 4690 fingerprint sensor and ESP32 S2 Feather microcontroller

Having a hardware element to the artefact proved challenging. This is primarily because each piece of hardware had to be integrated with one another to interact in the desired way. This proved difficult for a few reasons: the CircuitPython fingerprint sensor package made by Adafruit did not have functionality to export fingerprint images once created; the microcontroller board selected for the project did not have sufficient flash memory to utilise the required Python packages to extract images; and using USB to synchronise the hardware implementation with the Python file on the MacBook was difficult too. To counter this last problem, using the ESP32 S2's Wi-Fi capabilities was considered. These issues are discussed in greater detail in the Critical Appraisal chapter.

By this point, progress was stalling, and adjustments needed to be made to the project so that development could stay on track. It was important that the timeframe of the research project was respected, and this was a moment where this premise was recognised and upheld. Following discussion with my project supervisor, some next steps were suggested. Mainly, development of the software side of the implementation would commence independent of the completion of the original implementation's hardware. Working in this manner allowed for research outcomes to be met. The decision was made to move on from the hardware element and focus on developing the core Python implementation.

2.2.2 Software Build

The first iteration of the algorithm took image input, measured its dimensions, used these dimension values to raster scan through each pixel, and store its data in a one-dimensional array. It also scaled the recorded data from its 8-bit integer resolution (256 values) to float values between 1 and -1. It then auditioned the sound to the user of the program. The algorithm deals with 8-bit single channel greyscale images. The greyscale channel value is scaled and mapped to amplitude. Using multi-channel images, such as red-green-blue (RGB), could be advantageous for artistic purposes, such as mapping the channels to audio parameters in an audio plug-in. This would have taken the project away from the pseudo-intrinsic, periodic nature of mapping single channel colour value to audio amplitude, so it was decided not to implement colour image handling.

```
# Iterates through every horizontal pixel position, top to bottom
for y in range(height):


    # If row is at an even index, traverse from left to right
    if (y+2) % 2 == 0 or ((y+2) % 2 == 1 and bi_sel == 2):


        # Iterates through each pixel in the row
        for x in range(width):
            pixel_value = image.getpixel((x, y))


            # Add the pixel
            pixel_values.append(pixel_value)


    # If odd index AND bidirectional = 1 (True), traverse right to left
    # This 'snaking' through the pixel rows is raster scanning
    else:
        for x in range(width):
            pixel_value = image.getpixel(((width-x-1), y))
            pixel_values.append(pixel_value)
```

Around 10 fingerprint image samples were selected from a larger data set to be used when testing the initial algorithm. Work was carried out with sample fingerprint images sourced from the University of Bologna's 2002 Fingerprint Verification Competition (FVC2002) website (University of Bologna, 2002). Their databases were available for anybody to download to participate in the competition. Conveniently, the website still exists today so the images could be utilised. This allowed for input testing of the algorithm to commence during development.

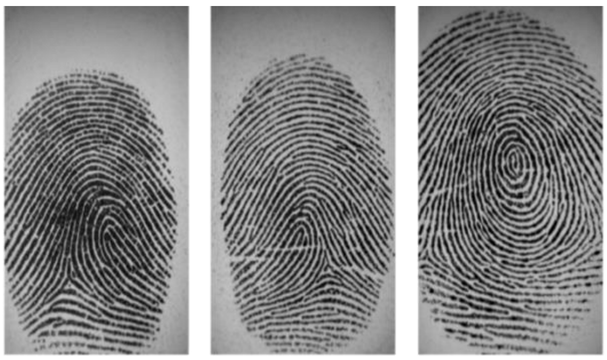


Figure 3 - Three fingerprints from the FVC2002 dataset

Once experimentation with fingerprint image input began, it was clear that these sonified fingerprint images largely sounded the same in terms of timbre, with the only difference being felt due to positioning differences in the images. Gaps at the top or bottom of the image would lead to quieter audio at the start or end respectively. Experimentation started with other digital images to observe the variations displayed by a wider range of input (Art, photography, pixel images such as gradients, block gradients and textures).

This expansion showed a much more varied range of sonified sounds. It also led to the image input section of the algorithm being optimised so that it could robustly handle a single channel greyscale image of any height or width. It did so by measuring and storing the dimensions of the image on opening. These were utilised when iterating through each pixel in the image.

```
# Get the dimensions of the image
width, height = image.size
```

Next, a framework was built on top of the core raster scanning algorithm focused on convolving the extracted image signal with a waveform such as a sine, square or sawtooth wave of a selected frequency. It also included menu functionality using ‘while true’ loops and exit conditions.

```
while True:


    try:
        bidir = int(input(
```

The menus in the terminal allow the user to navigate functionalities such as selecting image and oscillator properties, audition the result of the convolution, export the audio to Waveform Audio File Format (WAV) file in a dedicated folder, or quit the program. Greyscale images to be sonified are placed by the user in a folder in the same directory as ‘main.py’.

```
    # Validates user input is an integer
    except ValueError:
        print("Please enter either 1 or 2.")
        continue
```

Many of the functionalities of the first version of the core raster scan algorithm were abstracted into functions employed many times throughout the 512 lines of code that make up ‘main.py’. This includes plotting waveforms and Fast Fourier Transforms (FFT), the core raster scanning algorithm, user selection menus, data normalisation, audio fading, and waveform generation.

```python
def my_fft(data):
    # Create an FFT of the image data
    data_fft = fft.rfft(data)
    # Create a list of positions
    positions = np.linspace(10, 22050, 22040)
    # Create the plot (x,y)
    plt.semilogx(positions, np.absolute(data_fft[10:22050]))
    plt.xlabel('Frequency')
    plt.ylabel('Amplitude')
    plt.title('FFT')
    plt.show()
```

At this point, the implementation was running as expected, except for some simple quick-fix bugs that were squashed as they were encountered. Minor optimisations were made, such as: adding user choice to whether the raster scan occurred in one direction or two; displaying input metadata on the menus to inform the user when they are modifying parameters; using the ‘os’ and ‘pathlib’ packages to generalise file paths throughout the implementation; and modifying the oscillator signal’s duration to match that of the sonified image signal. This was changed to create more natural sounding convolution output, because the length of a convolution in samples is equal to the sum of both input signal lengths minus one sample.

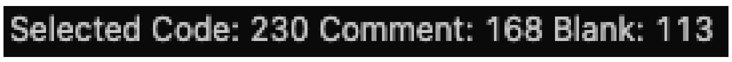
$$f(n) = N_f \ \& \ g(n) = N_g$$
$$N_h = N_f + N_g - 1$$

Equation 1 - Length of the output of a convolution

2.3 Code Testing

All testing that was performed is classed as ‘white-box testing’. This refers to the tester’s ability to access the source code while they perform tests. This contrasts with ‘black-box testing’, which focuses on the potential user’s experience by testing without access to the source code (Hamilton, 2023).

Of the 511 lines in the implementation, 398 are code or comment lines. Using this value, a code coverage of 87.2% was attained. This was calculated by identifying code lines or blocks that were tested and summing them to gain a percentage value.



```
Selected Code: 230 Comment: 168 Blank: 113
```

Figure 4 - VS Code Counter output

2.3.1 Unit Testing and Validation

Unit testing is focused on testing individual sections of code independent of the system. This was performed by testing all employed functions on their own before integrating into ‘main.py’. This is to ensure that all software meets requirements and expectations. It is cheaper and easier for developers to troubleshoot bugs at component level because it makes integration and system testing more straightforward later.

Validation was performed on elements of the implementation that handle user input. It is important to make sure that all input is of the expected datatype and appropriate value range. The code validated the user input by attempting to cast the input as the desired datatype. If the user entered an input of the correct datatype, the employed ‘try’ statement would pass. If not, an ‘except’ statement would act on a detected value error and print an error message to the user. This is combined with ‘while true’ loops with exit conditions that check the range of the input. This creates code flow where input menus print repeatedly until the user input is of the correct datatype and value range. This was used for menu and filename input validation.

Waveform and FFT graph functions were optimised at unit level to be more informative by adding a main title and axis labels. Also, the FFT graph was modified to show only the absolute values of the real component. Combined with an appropriate x-axis range (10-22,050Hz), the FFT graphs became much more aesthetically pleasing and useful to read from.

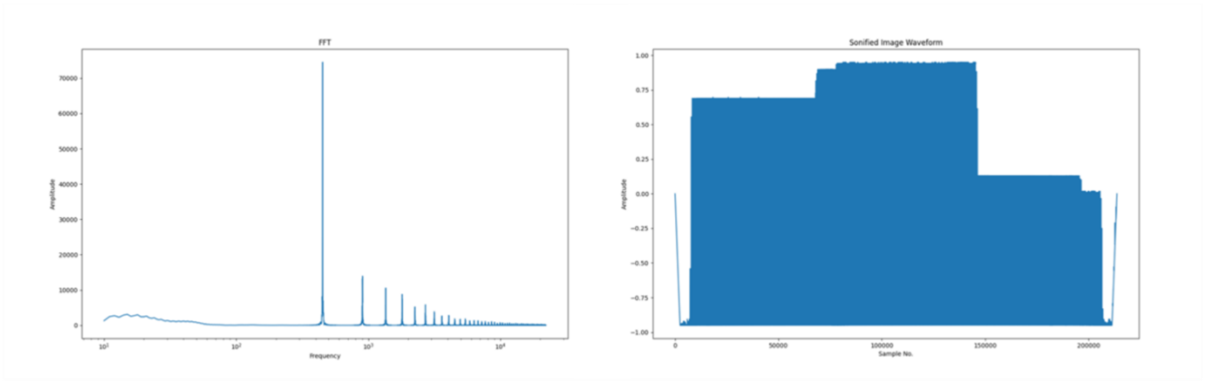


Figure 5 - Optimised FFT and waveform graphs

2.3.2 Integration Testing

Integration testing makes sure that elements of the code, such as functions, interact in a desired way. This is made easier by performing unit testing on all software components. The areas that required testing were the ‘dir_path’ variable when called in an f-string; conditional statements that followed menu inputs; and menu display f-strings.

```
dir_path = os.path.dirname(os.path.realpath(__file__))
```

```
    # Validates that input is within desired range
    if select > 4 or select < 1:
        pass
    else:
        break
```

```
IMG   Name      : {Path(img.filename).stem}
      Format    : {img.format}
      Dimensions: {width}x{height}
      Pixels    : {osc_d}
      Duration  : {round(osc_d/sample_rate, 2)} seconds (2 d.p.)

OSC   Waveform  : {wf_type}
      Frequency : {osc_f} Hz
      Samples   : {osc_d}
      Duration  : {round(osc_d/sample_rate, 2)} seconds (2 d.p.)
```

2.3.3 System Testing

System testing is an important part of the testing process. It makes sure that all components of the implementation interact in the desired way. Usually, this is performed in a testing environment for larger systems. For a project of this scale, system testing takes the form of debugging the ‘main.py’ file. This mainly consisted of making sure that the heavily abstracted implementation ran correctly, as around half of ‘main.py’ consists of function definitions. System testing was largely made easier by carrying out unit and integration testing on software elements before implementing them into ‘main.py’.

2.3.4 Regression Testing

Regression testing is performed to ensure that any changes made to the implementation (such as modifying the function that handles the duration of the oscillator waveform) do not affect the functionality or running of the implementation in unintended ways. Regression testing happened any time that the implementation was altered.

2.3.5 Usability Testing

A form of usability testing was mentioned in the interim report. It was planned to get potential end-users (music producers, composers, etc) to give feedback on the implementation. As the project developed, it was realised that usability testing is not important for a prototype artefact. This is because a prototype is an early version of a product that is used to test and validate design concepts and features. It is typically not a fully functional product suitable for use by end-users. During the development of a prototype, the focus is on testing and iterating on design concepts and features. Usability, along with security and performance testing, is beyond the scope of this research project.

2.4 Arrival at the Finished Artefact

The finished implementation opens an 8-bit greyscale image; raster scans it to create a one-dimensional array of pixel colour data; scales the data values between 1 and -1; and then convolves this signal with a typical audio waveform such as a sine, square, or sawtooth wave.

It also utilises the ‘matplotlib’ library to output waveform and FFT plots; prints terminal menus so that users can edit various parameters and use functionalities; and saves convolved signals to file upon user request.

The implementation in its finished form was used to explore a range of image input beyond fingerprint images. This includes digital photography, artworks, geometric patterns, gradients, and textures, all of which were converted to greyscale using GIMP. The audio output, a convolution of the image signal and a typical audio waveform, was often a complex, metallic sounding signal, like the output produced by frequency modulation (FM) synthesis.

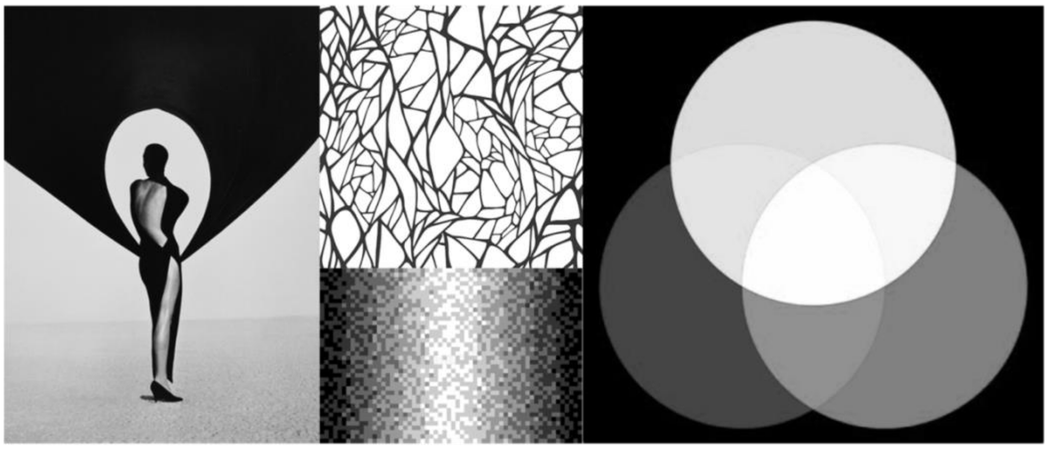


Figure 6 - Sample of greyscale images used for input trialling

2.5 Image/Audio Characteristics

A simpler version of the implementation, named ‘imagesave.py’ was created to sonify images and save them to file. This was used as a tool to explore image sonification and raster scanning without further sound synthesis.

It was found that complex images such as photos and artworks were very noisy due to the immense detail encoded in the image data. Some artworks with bolder lines and larger solid blocks of colours had an audio signal that was somewhat easier to discern characteristics from. Simpler images such as geometric patterns, gradients and shape combinations were much better suited for learning from. It was important to try with a wide range of image input, despite many not having a discernible educational value. This is to test the resilience of the implementation and to reflect the freedom that potential users would enjoy, being able to use any greyscale image as input.

Following discussion with my project supervisor, we agreed that using simplistic images would be more useful when it came to discerning audio characteristics. From this, greyscale images that represented sine, square, and sawtooth waves were developed.

2.5.1 Image Dimensions

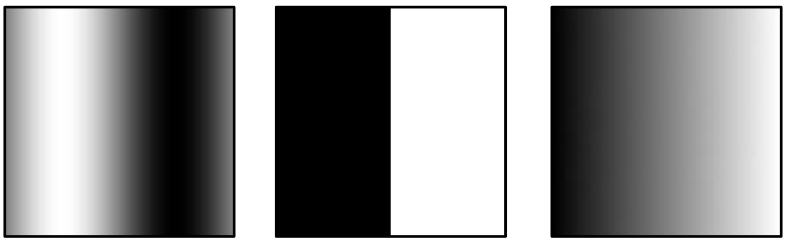
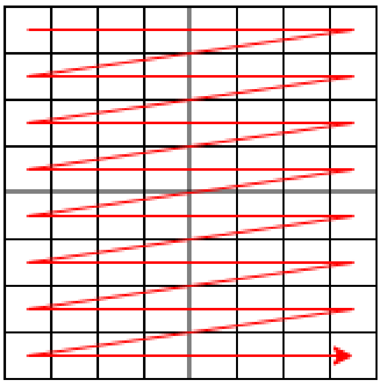


Figure 7 – Sine, square, and sawtooth waveforms represented as greyscale images

Figure 7 shows three 200x200 8-bit greyscale images. From left to right, they represent a sine wave, a square wave, and a sawtooth wave. The 256 unique pixel colour values represent amplitude, with a value of 0 representing pure black, and 255 representing pure white. Bidirectional scanning is typically how raster scanning works, but unidirectional scanning is more useful for discerning the effects of image transformations on resultant audio. Treating each pixel row as one cycle of a waveform, unidirectional scanning reads every cycle in the same way.



For greyscale images that represent audio waveforms, image width is analogous to the period

Figure 8 - Unidirectional raster scan diagram

of the resultant audio. For example, if an image is 200 pixels wide and the resultant audio is auditioned with a sample rate of 48kHz, then the period of the image's waveform is equal to 200 samples, or 4.17 milliseconds. This corresponds to a frequency of 240Hz. The image's height is analogous to the number of cycles in the resultant audio. A 200x200 image auditioned at 48kHz would complete 200 cycles at a frequency of 240Hz, having a total duration of 40,000 samples, or 0.83 seconds. It should be noted that images that represent static audio waveforms in this way have identical pixel data in every row. If these pixels rows were different, the resultant waveform would develop over time. This would be unhelpful as a starting point for this research.

2.5.2 Image Transformations

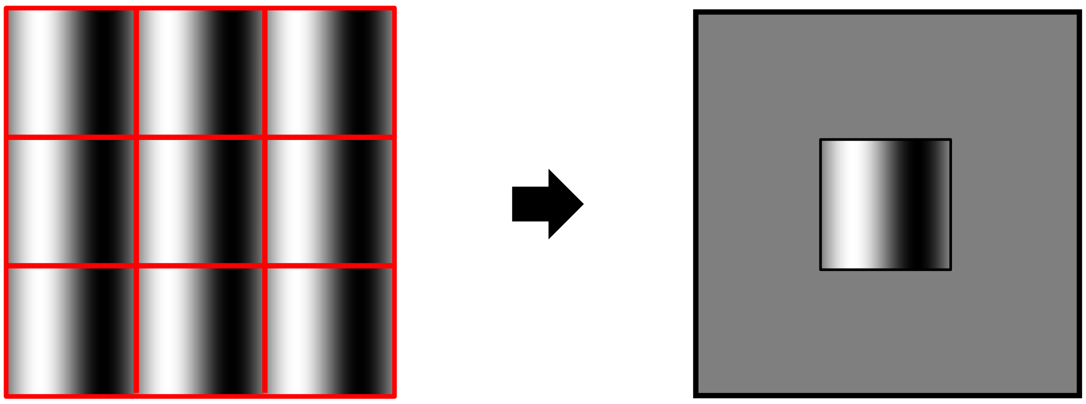


Figure 9 - A 200x200 pixel sine wave image tiled 3x3, then recanvassed

By applying transformations to the image input, fundamental effects on the resultant audio can be observed. Unidirectional raster scanning is employed to achieve this. The images in Figure 7 were tiled into 3x3 grids (600x600 pixels) but with a centred 200x200 pixel canvas, showing only the original image in the centre tile of the grid. These tiled grids are useful for observing the effects of translation or rotation on an image's resultant audio because they provide repetition of the pixel data on all sides of the image. The canvas size boxes the tiled and transformed image into the dimensions of the original image, preserving its periodicity and duration. This tiling could be applied to any regular rectangular image. Figure 9 shows the 600x600 pixel tiled sine wave image before and after the 200x200 pixel canvas is overlaid.

Applying an x-direction (left/right) translation to a tiled image is analogous to shifting the phase of each cycle. For example, translating the image to the right by 100 pixels, half of the image width, would cause a 180º phase shift. Applying a y-direction (up/down) translation to a tiled image alters the pixel row at which raster scanning begins. This transformation applies an offset to the start position of the raster scan and could be called a 'cycle shift'. Shifting a tiled 200x200 pixel image upwards by 50 pixels would cause the raster scanning to start on the 51st pixel row and finish on the 50th. In the case of the example images shown in Figure 7, no audible effect would be observed with y-direction translation alone as every row of pixels is identical.

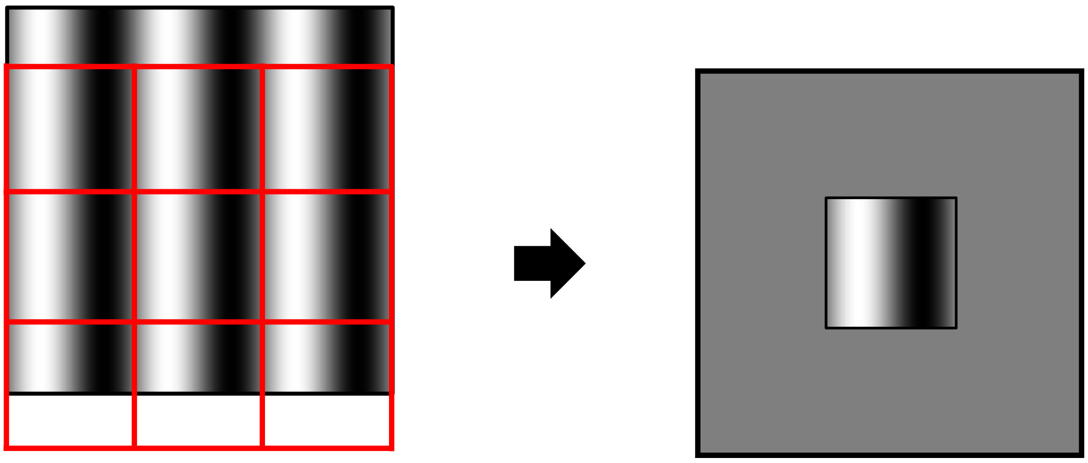


Figure 10 - 100 pixel shift upwards (start offset)

Applying an x-axis reflection to an image reverses the playback order of each cycle while leaving each individual cycle unchanged in terms of phase. This would seemingly reverse the resultant audio, but not exactly: reversing the audio would reverse the order of every sample, whereas this method reverses group of samples, the size of the group defined by the image width, or cycle size in samples. This preserves the order of each cycle's samples while reversing the order in which the cycles are played back. Applying a y-axis reflection to an image will do the inverse of an x-axis reflection: each cycle will be played in reverse, while their playback order is maintained.

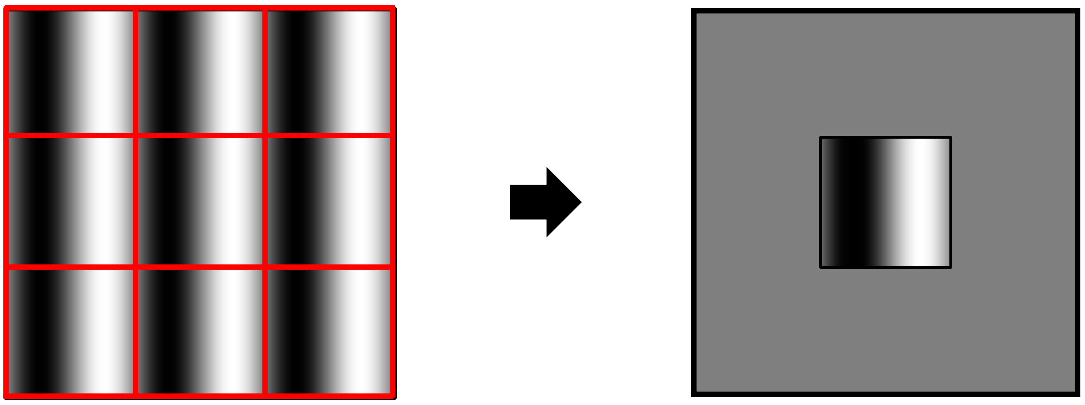


Figure 11 - y-axis reflection

Rotating an image has a range of effects depending on the angle of rotation. Rotating an image by 180º will result in the audio output being completely reversed. This transformation could also be viewed as two reflections, once around the x-axis and once around the y-axis. This composite transformation reverses the sample playback order within each cycle (x-reflection), then reverses the cycle playback order (y-reflection), having the same outcome as reversing the order of the total list of samples (180º rotation). This shows that reversing audio generated from image raster scanning can be split into two components: in-cycle sample reversal (x-reflection) and in-image cycle reversal (y-reflection).

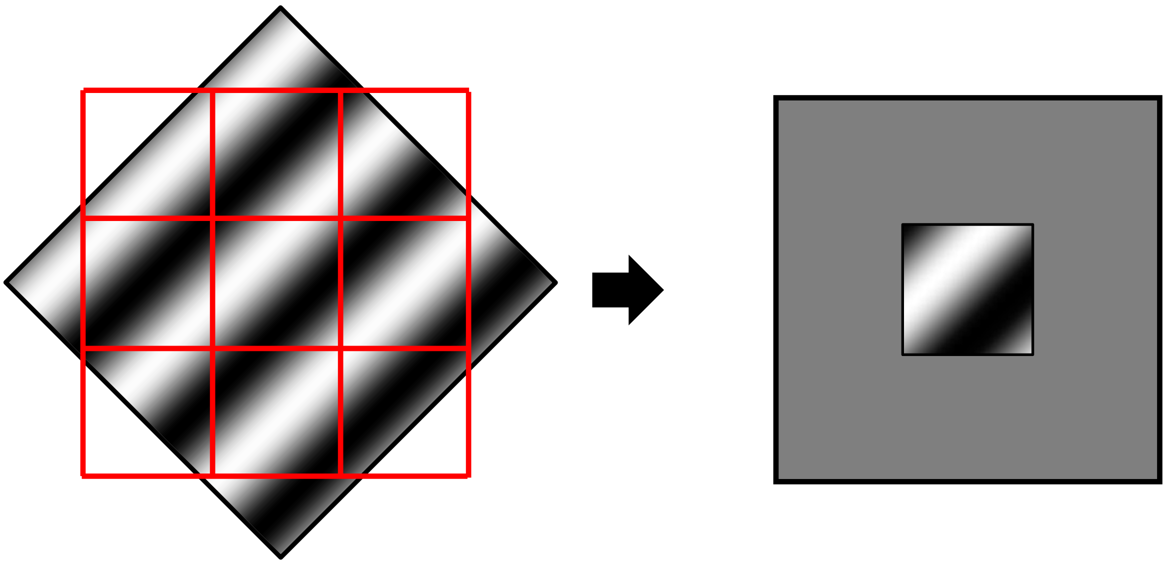


Figure 12 - 45º rotation

Rotating an image from 0º to 90º has interesting effects on the resultant audio. This is largely due to the canvas size of the original image being applied to the 3x3 tiled image. Applying this canvas on top of the transformed image applies the original period and duration on to it. As the image is rotated and recanvassed, each cycle of the original image is seemingly stretched more and more as it rotates. The cycles are refreshed every 200 samples in this case, which has the effect of cutting each stretched cycle short to a period of 200 samples. This looks like hardsync: a synthesis option available on some synthesisers, but it is achieved in a different way.

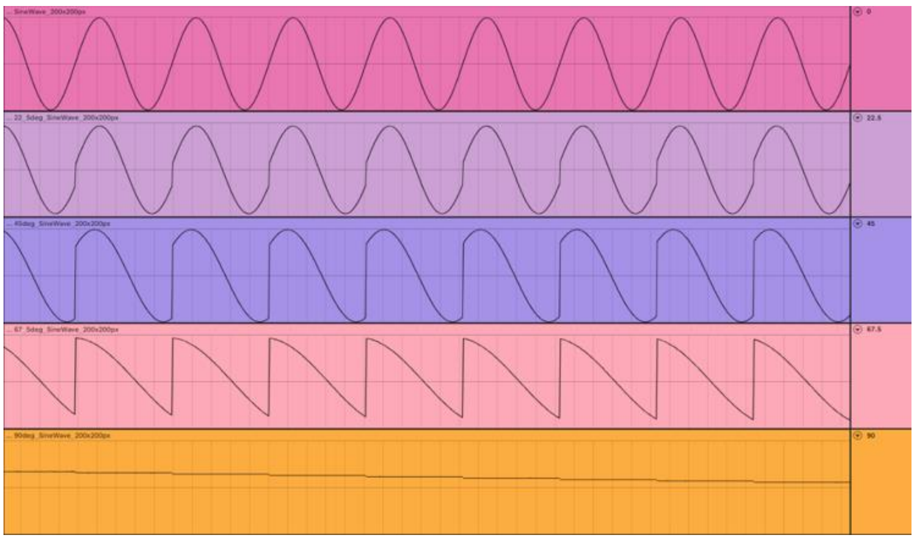


Figure 13 - Hardsync-style distortion observed as rotation increases from 0º to 90º

A form of audio distortion can be observed, and as the rotation angle increases, complete breakdown of the resultant audio occurs. In the case of images that represent static audio waveforms, a 90º rotation creates one cycle of the respective waveform which has a period equal to the duration of the original audio signal. Tests were performed with a sine wave. The breakdown into a single-cycle audio signal can be observed as rotation approaches 90º.

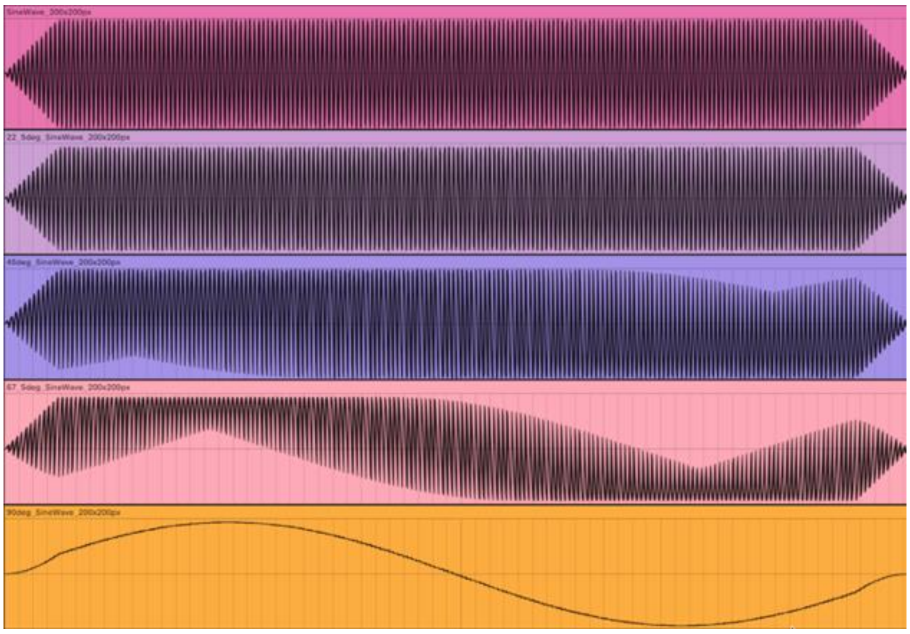


Figure 14 - Macro view of waveform distortion

These transformations could have implications for audio manipulation, allowing for new synthesis methods to be created by using waveform cycles as a subset of the dimension of time. Beyond this research, image transformation sonification could be an interesting area to expand the project into.

3 Sustainability

3.1 Environmental Concerns

With any audio project, noise pollution should be considered as a potential environmental concern. The project in its current state does not have large implications for noise pollution. If the research resulted in an electro-acoustic installation, for example, where an array of speakers was used to exhibit the created sounds in a physical space, noise pollution and other acoustic considerations would need to be made.

A genuine concern for any electrical or electronic device (hardware or software) is its power consumption. As a rule of thumb, software is less energy intensive than hardware. As this implementation runs using Python and some additional packages to complete non-real-time audio processing tasks, its computational cost should be manageable for most personal computers. It should be noted that the implementation was developed entirely on a 2013 MacBook Air, which has limited computational power compared to a modern device.

3.2 Economic Concerns

If this prototype were to be developed into a product to be sold as an audio plug-in, further development would be required, developing beyond Python into C++, a more robust audio programming language. A price could be applied to the product if it were to see release, maybe around £20-40 depending on current demand for sonification tools.

Innovation can and has been achieved by applying a business mindset to audio, but this would require a high level of capital to back a team of researchers, developers and project leads with the resources required to make and maintain a truly innovative product. Furthermore, all the provided capital within the private sector is underpinned by the promise of a return on investment. If the customer appeal is not great enough, market demand will not be sufficient to meet fiscal expectations. This could lead to decisions being made with vested interests in mind, as opposed to truly trying to create the best product possible.

An alternative option is to commit the future of this project to being community-based and open-source. From a business point of view, this may seem like a poor choice. However, from a research and development point of view, an open-source project would almost definitely provide for more interesting findings, developments, and experiments along the way. Open-source projects require community, co-operation, and most importantly, vast interest and knowledge of the subject at hand. As a result, progress may be slower than that of industry, but the product lifecycle and design process can be scrutinised at a closer level with less potential for conflicts of interest to affect development.

Thanks to prior research undertaken as part of this project, competition would not be a large issue with this implementation if it were to make it to market. There are limited consumer audio products that incorporate image sonification in some form into their workflow, especially not digital image raster scanning which has been utilised as a core part of this project.

The software implementation is built on Python and some external packages for Python, all of which are free at the point of use. Any user could install the required packages and run the prototype implementation in its current state on a home computer device.

3.3 Social Concerns

The main social concern of this research project regards intellectual property and fair use. Every effort has been made throughout to only use digital image either sourced under Creative Commons (CC) license or created for the project. Using the intellectual property of others without authorisation could lead to later problems with ownership of synthesised sounds. Despite this, if the prototype was developed into a consumer product, there would be no way to regulate which image a user decides to use as input, and no easy way to identify if an image had been used in the process of synthesising a sound.

From this, an interesting debate arises. Using film photography as an example, does a photographer believe that a raster-scanned list of derived pixel colour-values holds the same value as the image they originally captured on film? Can it be protected in the same way? A digital scan of an image can be. However, the pixel colour-values list and the digital image are only intrinsically linked to one another if you understand the somewhat arbitrary process of acquisition, scaling, and normalisation that link the two. Reassembling the data in a different way will not produce anything like the original image portrayed. This processing could provide enough abstraction from the original spirit of the image that there is no intellectual property left to protect. The image simply provides a texture that colours or filters the sound it is convolved with. There is no meaningful link that users can make between an image and the sound of that image as there is no way to preserve the intrinsically two-dimensional visual nature of an image when mapping the data to the one-dimensional time domain. Arbitrary decisions must be made, such as sample rate and how the data is read,

Another social concern that indirectly affects this research project is accessibility. Sonification is about making data more accessible through sound, so considerations should be made towards making this software accessible to as wide a range of users as possible. For example, this could take shape through screen-reading functionality that reads menu options to visually impaired users, combined with other assistive technologies such as braille keyboards. Although accessibility is not a key objective of this research, factoring it into any further development would strengthen the project's alignment with the wider ethos of data sonification.

4 Critical Appraisal

4.1 Project Reflection

Reflecting on the evolution of the research project allows for further learning to take place beyond the knowledge gained from the research itself. This project evolved substantially from its conception. Sometimes this was in reaction to problems that needed navigating. Other times, inspiration was found, and relevant functionality made its way into the final artefact. These two scenarios were not always mutually exclusive; sometimes, a solution or a project evolution was found by sourcing inspiration from the problem itself.

A large shift in the project's direction occurred because the Adafruit fingerprint sensor library that was being used did not provide image export functionality. In reaction to this, research was carried out regarding image data handling on board the microcontroller. Examples were found for other microcontrollers and fingerprint scanner models that utilised the Python Image Library (PIL) package. The microcontroller in question, the ESP32 S2 Feather, only has 2MB flash memory on board. More research in the project's preliminary stages would have shown that the PIL package requires 12.2MB of memory to store. Unfortunately, this ruled out processing the sensor data into an image on board the microcontroller. While other potential solutions were being explored, work was carried out with sample fingerprint images sourced from the FVC2002 website. Using the images showed that sonified fingerprint images were generally noisy and sounded very similar to each other. This lack of variation within the sonics of fingerprint images led to broadening the input data to practically any digital image. The limitation placed on the project by an incomplete Python library and limited microcontroller flash memory led to the project leaving hardware behind and evolving into something beyond just biometric image data. This ultimately led to more meaningful outcomes while making sure that the project's development did not stagnate.

Finding greyscale images that had an actual grey colour model, as opposed to RGB, proved difficult. Many images were found that appeared as greyscale, but upon further inspection were using the RGB colour model. This led to research being undertaken on whether RGB images could be converted to greyscale within Python. Some potential solutions were found, but none of them managed to solve the specific problem at hand: converting any RGB image into single channel 8-bit greyscale. The solution found was a free software called GIMP. It was used to convert colour images into single channel greyscale images. This formatting was attempted within the code implementation, but nothing attempted or provided gave a general solution to handle a wide range of image inputs. GIMP was a very straightforward solution that could convert any image into Portable Network Graphics (PNG) format with the pixel colour model required. This also led to GIMP being used for image transformations later in the project.

When convolving the two input signals (image and oscillator), if they were of vastly different lengths, the result of the convolution would have a disjointed sound. The length disparity of the inputs could be heard clearly within the output signal. This is due to the larger signal completely enveloping the other during the crossover multiplication that is convolution. To improve the quality of the output signal, the oscillator's duration value was no longer obtained from user input. Instead, the duration of the sonified image signal informed the duration of the oscillator-generated waveform. Adding this effective limitation to the algorithm provided much more satisfying output signals due to the input signals being of equal duration.

4.2 Meeting Project Criteria

The project's aims and objectives were reworked following the interim report to improve the foundation of the project. This was in reaction to advice from my project supervisor. This helped focus the project and allowed for its subsequent success.

The research project met most of the requirements laid out in the aims and objectives. The implementation successfully read and stored the pixel data in a one-dimensional format using raster scanning (O2); processed the data into a format that allowed for audio digital signal processing (DSP) manipulation (O3); and offered sound synthesis functionalities using a 3-waveform oscillator and convolution (O4). The only objective that was not fully achieved was creating a way for the user to generate an image to be sonified (O1). However, by using GIMP to transform images, it was shown how users could manipulate an input to modify the resultant audio.

	Number	Description	Achieved
Aim	1	Produce a sound synthesis implementation that uses digital image as a form of input.	Y
	2	Explore image sonification applied to sound synthesis using the created implementation and other software to do so.	Y
Objective	1	Create a way for the user to generate an image to be sonified.	N
	2	Write code that can read and store pixel data in one dimension.	Y
	3	Process the data into a format that allows for typical audio DSP conventions and techniques to be employed.	Y
	4	Design a system that facilitates sound synthesis through manipulation of the sonified image signal.	Y
	5	Build on existing research using elements of the implementation in combination with other software.	Y

Table 1 - Achieving aims and objectives

Both aims were met, firstly with the application of convolution and an oscillator to show how new sounds can be created using image sonification as the conduit; and secondly with the raster scanning Python code that was used in combination with GIMP to explore the effects of basic transformations on a sonified image's audio output.

4.3 Further Development

8-bit greyscale images were used in this research project. Naturally, 8-bit audio was generated from this data. 8-bit is standard image bit-depth for viewing on computers and other screen-based media. This meant that they were readily available and a standard fit for most devices. Ideally, images with a bit-depth of 16 would have been used as input for the implementation, as this would have allowed for higher resolution 16-bit audio to be generated from these images. If this project is developed further, 16-bit image handling and audio output would be a design focal point.

The fundamental frequency of the sonified image signal is partially dependent on the playback sample rate used, along with the image width. Variable playback sample rate could be utilised to create ‘vari-speed’ functionality when used in conjunction with resampling before convolving with a typical audio sample. This development would abstract the sonified signals beyond fundamental pitch, but limitations would present themselves. For example, if 48kHz was the default sample rate, there would be no issue with pitching up the audio by increasing the sample rate then resampling to 48kHz. However, the signal would only be able to be slowed by around 8% before the sample rate was below 44.1kHz and frequency information would begin to be lost. This could be negated by using a higher default sample rate, such as 96kHz, allowing for around 50% slowdown with no concerns regarding lost frequency information, but this trade-off incurs an expensive increase in computational requirements. Images would need to be twice as large for the same duration of audio, and the resulting audio would be processed at twice the cost too. It remains to be seen whether this would even be a viable feature that could be worked into a real-time audio implementation due to the high computational cost.

Another function that could be added is the option of multiple raster scan modes beyond unidirectional and bidirectional raster scanning. This would allow for multiple sonified signals to be synthesised from one image.

Adding stereo functionality to the implementation is another potential further development. This could allow for two images to sonify the left and right channels respectively. Or, one image could be scanned in two different ways, using one signal for each channel. This could take shape by creating a class for the implementation and using objects to represent the left and right channels. Using object-oriented programming (OOP) could allow for expansion beyond stereo into surround or multichannel audio.

Using a more appropriate programming language will allow for real-time audio functionality to be developed. This will allow for users to audition and modify audio in a much more satisfying way. The most likely option for this is C++ due to its efficiency advantages over Python. Also, the JUCE plug-in development framework which utilises C++ may be a useful tool to help realise this development area.

Adding a barrage of features to the implementation, such as multiple raster modes or multichannel support with OOP, could detriment the workflow of the user because currently choices are made through a series of menus in the terminal. More functionalities could have the potential to slow the implementation down too. With a graphical user interface (GUI) and real-time audio support using C++, more functionalities could be added without tedious user experience.

5 Conclusions

This project asked if a novel sound synthesis implementation that harnessed image sonification could be created. It aimed to produce a software implementation that used digital image as input; and to explore image sonification applied to sound synthesis with the created implementation and growing knowledge on the research area. To meet the objectives of the project, Python was used to create software, 'main.py', that raster scanned an image into a one-dimensional array and processed it into audio; convolution and an oscillator were incorporated to make a synthesis framework that could create new sounds from digital image input; and a second Python program was created, 'imagesave.py', which was used in combination with GNU Image Manipulation Program (GIMP) and Ableton Live to apply image transformations and observe their effects on the resultant audio.

The audio created with 'main.py' was noisy but had similarities to frequency modulation (FM) synthesised audio when using square or sawtooth waves. This was mostly observed in detailed digital images such as photographs or artworks, often creating dissonant harmonics. Images that represented static basic waveforms (sine, square, and sawtooth) were used to discern the effects of image transformations on the sonified audio. GIMP was used in combination with 'imagesave.py' and Ableton Live to find that: image rotation modulated hardsync-style waveform distortion reaching signal destruction at 90°/270°, and total signal reversal at 180°; image translation modulated phase shift (x-direction) and raster scan start offset (y-direction); and image reflection flipped the playback order of the samples within each cycle (y-reflection), or the playback order of the complete cycles (x-reflection). Image transformations and their respective audio manipulation techniques have strong implications for sound synthesis when combined with raster scanning and image tiling.

Further development of the two implementations would include 16-bit resolution image and resultant audio, vari-speed functionality, stereo raster scanning, real-time audio support, and a graphical user interface (GUI) instead of terminal menus. Using C++ and the JUCE development framework could enable this, and it would lead the way towards an audio plug-in for use in digital audio workstations (DAW). Future research within this area would explore more image effects that are analogous to audio effects, using a wider selection of basic waveforms to observe audio effects further. This would allow for a broader understanding of how sonification and image manipulation interact. Overall, this would lead to a stronger knowledge base that would contribute to a superior sonification plug-in for sound synthesis.

Reflecting on the research project as a whole, some observations have been made: more preliminary research would have been required to successfully carry out a hardware component of the implementation as many issues were encountered; and the most interesting findings were discovered using images that represented sounds, such as the 200x200 pixel images of sine, square and sawtooth waveforms. Overall, leaving the hardware behind to focus on the Python implementation allowed the project to evolve and follow a path to unexpected findings relating sonification to image transformations.

Reference List

CircuitPython (2023) *CircuitPython Documentation.* Available at: https://docs.circuitpython.org/en/latest/README.html (Accessed: .

Hamilton, T. (2023) *White Box Testing – What is, Techniques, Example & Types.* Available at: https://www.guru99.com/white-box-testing.html(Accessed: May 31, 2023).

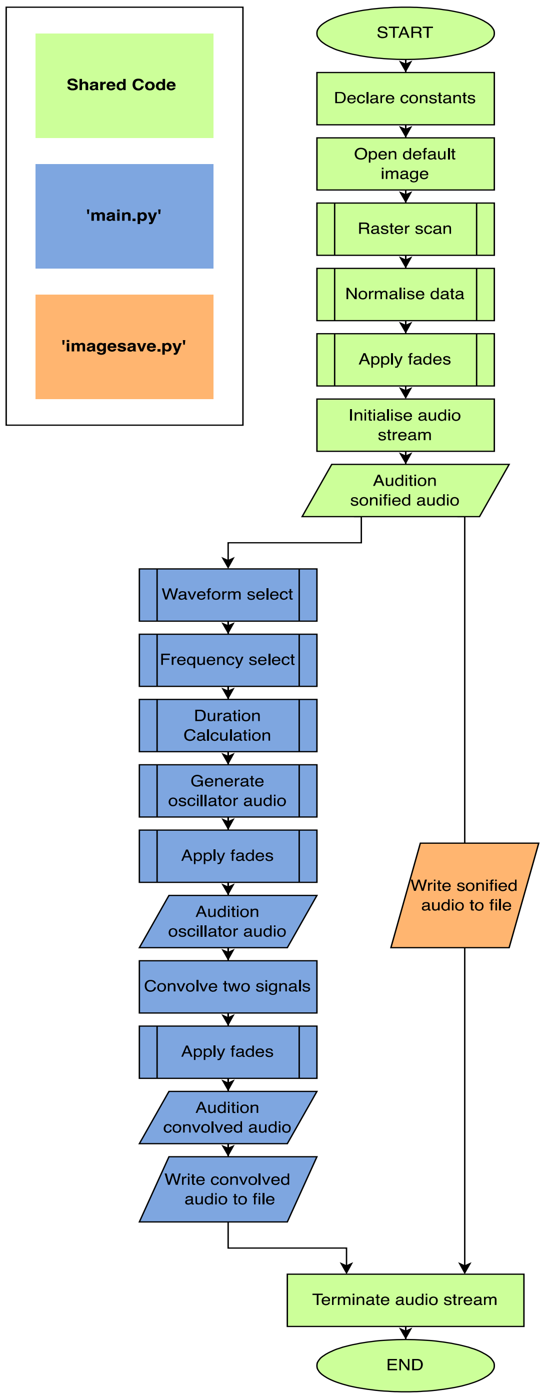
Hermann, T., Hunt, A. and Neuhoff, J.G. (2011) *The Sonification Handbook.* Logos Publishing House.

McGee, R., Dickinson, J. and Legrady, G. (2012) 'Voice Of Sisyphus: An Image Sonification Multimedia Installation', .

University of Bologna *Fingerprint Verification Competition 2002.* Available at: http://bias.csr.unibo.it/fvc2002/# (Accessed: May 31, 2023).

Yeo, W.S. and Berger, J. (2006) *Raster Scanning: A New Approach to Image Sonification, Sound Visualization, Sound Analysis And Synthesis.* 18 Oct 2022. Available at:(Accessed: .

Appendix 1 – ‘main.py’ and ‘imagesave.py’ block diagram



Appendix 2 – main.py

```
### Package import ###

import numpy as np
from scipy import signal, fft
from scipy.io.wavfile import write
import pyaudio
from PIL import Image
import matplotlib.pyplot as plt
import os
from pathlib import Path


### Defining functions ###

def my_fft(data):
    # Create an FFT of the image data
    data_fft = fft.rfft(data)

    # Create a list of positions
    positions = np.linspace(10, 22050, 22040)

    # Create the plot (x,y)
    plt.semilogx(positions, np.absolute(data_fft[10:22050]))

    # Add axis labels and title
    plt.xlabel('Frequency')
    plt.ylabel('Amplitude')
    plt.title('FFT')

    # Show the plot
    plt.show()

def raster_scan(image, bi_sel):
    # Get the dimensions of the image
    width, height = image.size

    # Create an array to store the pixel values
    pixel_values = []
    # Iterates through every horizontal pixel position, top to bottom
    for y in range(height):

        # If row is at an even index, traverse from left to right
        if (y+2) % 2 == 0 or ((y+2) % 2 == 1 and bi_sel == 2):

            # Iterates through each pixel in the row
            for x in range(width):
                pixel_value = image.getpixel((x, y))

                # Add the pixel
                pixel_values.append(pixel_value)

        # If odd index AND bidirectional = 1 (True), traverse right to left
        # This 'snaking' through the pixel rows is raster scanning
        else:
            for x in range(width):
                pixel_value = image.getpixel(((width-x-1), y))
                pixel_values.append(pixel_value)

    return pixel_values

def bidirectional_select():
    while True:

        try:
            bidir = int(input(
"""
------------------------------------------------
Please select bidirectional (L->R then R->L)
or unidirectional (just L->R) raster scanning
------------------------------------------------

1. Bidirectional
2. Unidirectional


------------------------------------------------
Select 1/2
------------------------------------------------
"""))

        # Validates user input is an integer
        except ValueError:
            print("Please enter either 1 or 2.")
            continue

        # Validates that input is within desired range
        if bidir > 1 or bidir < 3:
            break
        else:
            pass

    return bidir

def waveform_select():
    while True:

        try:
            waveform = int(input(
"""
-------------------------------------------------
Please select a waveform for the oscillator
-------------------------------------------------

1. Sine wave
2. Square wave
3. Sawtooth wave

-------------------------------------------------
Select 1/2/3
-------------------------------------------------

"""))

        # Validates user input is an integer
        except ValueError:
            print("Please enter an integer between 1 and 3.")
            continue
        # Validates that input is within desired range
        if waveform > 3 or waveform < 1:
            pass
        else:
            break

    return waveform

def frequency_select():
    while True:

        try:
            osc_freq = int(input(
"""
------------------------------------------------
Please input a frequency value in Hertz for the
oscillator between 10Hz and 22,000Hz.
------------------------------------------------


"""))

        except ValueError:
            print("Please enter an integer between 10 and 22,000.")
            continue

        if osc_freq > 22000 or osc_freq < 10:
            pass
        else:
            break

    return osc_freq

def duration_select():
    osc_duration = width*height
    return osc_duration

def normalise(data, max_min):
    # Find max and min values of all pixel colour data
    min_value = np.min(data)
    max_value = np.max(data)

    # Normalise pixel values between 0.95 and -0.95
    normalised_data = ((2 * (data - min_value) / (max_value - min_value)) - 1)*max_min

    # Validate the edge values to ensure no clipping will occur
    print(f"""MaxValue: {np.max(normalised_data)}
MinValue: {np.min(normalised_data)}""")

    return normalised_data

def fade_audio(unfaded_data):
    # Variable fade_length, 2400, is number of samples for fade in (50ms at 48kHz)
    fade_length = 2400

    # Create lists that go from 0 to 1 (fade in) or 1 to 0 (fade out)
    fade_multiplier = np.linspace(0, 1, fade_length)

    # Apply the fades to the audio data
    for i in range(0, fade_length, 1):

        # Image data
        unfaded_data[i] *= fade_multiplier[i]
        unfaded_data[-1-i] *= fade_multiplier[i]

def my_plot(data, plot_name):
    # Create a list of positions
    positions = list(range(len(data)))

    # Create the plot (x,y)
    plt.plot(positions, data)

    # Add axis labels and title
    plt.xlabel('Sample No.')
    plt.ylabel('Amplitude')
    plt.title(f'{plot_name}')

    # Show the plot
    plt.show()
```

```
def sine_gen(audio_rate, f, l):
    # Create time values
    t = np.linspace(0, l/audio_rate, l, dtype=np.float32)

    # Generate y values for signal
    y = np.sin(2 * np.pi * f * t)

    # Returns a sine waveform of the desired frequency and duration
    return y

def square_gen(audio_rate, f, l):
    # Create time values
    t = np.linspace(0, l/audio_rate, l, dtype=np.float32)

    # Generate y values for signal
    y = signal.square(2 * np.pi * f * t)

    # Returns a square waveform of the desired frequency and duration
    return y

def saw_gen(audio_rate, f, l):
    # Create time values
    t = np.linspace(0, l/audio_rate, l, dtype=np.float32)

    # Generate y values for signal
    y = signal.sawtooth(2 * np.pi * f * t)

    # Returns a sawtooth waveform of the desired frequency and duration
    return y


### Declaring parameters ###

dir_path = os.path.dirname(os.path.realpath(__file__))
sample_rate = 48000
channels = 1
format = pyaudio.paFloat32


### Image intake ###
```

```
# Open the image
img = Image.open(f'{dir_path}/Greyscale Images/rgb.png')
width, height = img.size

# Display information about the image
print(img.format)
print(img.mode)
print(img.size)

# Show the image
img.show()


### Image raster scanning ###

bi_select = bidirectional_select()
pixel_values = raster_scan(img, bi_select)

# Using the normalise() function to scale the pixel data values appropriately
image_data = normalise(pixel_values, 0.95)

# Use the fade_audio() function to apply short fades
# at the start and the end to avoid any audible distortion
fade_audio(image_data)


### Sound synthesis ###

# Create a PyAudio object
p = pyaudio.PyAudio()

# Initialise the audio stream
stream = p.open(format=format,
                channels=channels,
                rate=sample_rate,
                output=True)

# Audition the image audio waveform
stream.write(image_data.astype(np.float32).tobytes())
```

```
# Plot the image waveform and FFT
my_plot(image_data, "Sonified Image Waveform")
my_fft(image_data)

# Waveform selection
wf = waveform_select()

# Oscillator frequency selection
osc_f = frequency_select()

# Oscillator duration selection
osc_d = duration_select()

# Select the function to generate the chosen waveform
if wf == 1:
    osc_signal = sine_gen(sample_rate, osc_f, osc_d)
    wf_type = "Sine Wave"

elif wf == 2:
    osc_signal = square_gen(sample_rate, osc_f, osc_d)
    wf_type = "Square Wave"

else:
    osc_signal = saw_gen(sample_rate, osc_f, osc_d)
    wf_type = "Sawtooth Wave"

fade_audio(osc_signal)
stream.write(osc_signal.astype(np.float32).tobytes())
my_plot(osc_signal, "Oscillator Waveform")
my_fft(osc_signal)



# Create a convolution of the image and oscillator signal
conv_signal = signal.convolve(osc_signal, image_data)

# Normalise it
conv_signal = normalise(conv_signal, 0.95)
```

```
# Apply fades
fade_audio(conv_signal)
# Audition the convolved audio
stream.write(conv_signal.astype(np.float32).tobytes())
# Plot the waveform and FFT
my_plot(conv_signal, "Image/Oscillator Convolution Waveform")
my_fft(conv_signal)




# Exit condition for looping menu
quit = False

while quit == False:
    while True:
        # Menu allows user to: view details about the image and the oscillator,
        # modify them, audition the audio they have synthesised, save the
        # audio they have created to file, or quit the program.
        try:
            select = int(input(
f"""
------------------------------------------------
Modify IMG/OSC parameters, audition the convolved
audio, export audio, or quit.
------------------------------------------------

1. Modify IMG/OSC parameters
2. Audition convolved audio
3. Export audio
4. Quit

------------------------------------------------
Select 1/2/3/4
------------------------------------------------
Current IMG/OSC Parameters
------------------------------------------------

IMG   Name      : {Path(img.filename).stem}
      Format    : {img.format}
    Dimensions: {width}x{height}
    Pixels    : {osc_d}
    Duration  : {round(osc_d/sample_rate, 2)} seconds (2 d.p.)

OSC   Waveform  : {wf_type}
    Frequency : {osc_f} Hz
    Samples   : {osc_d}
    Duration  : {round(osc_d/sample_rate, 2)} seconds (2 d.p.)

------------------------------------------------


"""))

        # Validates user input is an integer
        except ValueError:
            print("Please enter an integer between 1 and 4.")
            continue

        # Validates that input is within desired range
        if select > 4 or select < 1:
            pass
        else:
            break

    # Parameter modify menu
    if select == 1:
        while True:

            try:
                param_select = int(input(
f"""
------------------------------------------------
Please select a parameter to modify
------------------------------------------------

1. Select new image
2. Select oscillator waveform
3. Select oscillator frequency


------------------------------------------------
```

```
Select 1/2/3
------------------------------------------------
Current IMG/OSC Parameters
------------------------------------------------

IMG   Name       : {Path(img.filename).stem}
      Format    : {img.format}
      Dimensions: {width}x{height}
      Pixels    : {osc_d}
      Duration  : {round(osc_d/sample_rate, 2)} seconds (2 d.p.)

OSC   Waveform  : {wf_type}
      Frequency : {osc_f} Hz
      Samples   : {osc_d}
      Duration  : {round(osc_d/sample_rate, 2)} seconds (2 d.p.)

------------------------------------------------

"""))

            # Validates user input is an integer
            except ValueError:
                print("Please enter an integer between 1 and 3.")
                continue

            # Validates that input is within desired range
            if param_select > 3 or param_select < 1:
                pass
            else:
                break

        # Select new image
        if param_select == 1:
            while True:

                try: img_select = input("""
Please place the image you want to sonify in the same directory as main.py
Type the full name of the image, i.e 'gradient.png': """)

                except FileNotFoundError:
```

```
            print("File not found.")

        if os.path.isfile(f'{dir_path}/GIMP Images/{img_select}'):
            print('File found.')
            break

        else:
            print("File not found. Please make sure the correct file name is entered.")

    img = Image.open(f'{dir_path}/GIMP Images/{img_select}')
    img.show()
    width, height = img.size
    osc_d = duration_select()
    bi_select = bidirectional_select()
    pixel_values = raster_scan(img, bi_select)
    image_data = normalise(pixel_values, 0.95)
    fade_audio(image_data)
    stream.write(image_data.astype(np.float32).tobytes())
    my_plot(image_data, "Sonified Image Waveform")
    my_fft(image_data)


else:
    # Select waveform
    if param_select == 2:
        wf = waveform_select()

    # Select oscillator frequency
    else:
        osc_f = frequency_select()

    if wf == 1:
        osc_signal = sine_gen(sample_rate, osc_f, osc_d)
        wf_type = "Sine_Wave"

    elif wf == 2:
        osc_signal = square_gen(sample_rate, osc_f, osc_d)
        wf_type = "Square_Wave"

    else:
```

```
                osc_signal = saw_gen(sample_rate, osc_f, osc_d)
                wf_type = "Sawtooth_Wave"

            fade_audio(osc_signal)
            stream.write(osc_signal.astype(np.float32).tobytes())
            my_plot(osc_signal, "Oscillator Waveform")
            my_fft(osc_signal)

        # Create a new convolution of the image and oscillator signal
        conv_signal = signal.convolve(osc_signal, image_data)
        conv_signal = normalise(conv_signal, 0.95)
        fade_audio(conv_signal)

    # Audition convolved audio
    elif select == 2:
        stream.write(conv_signal.astype(np.float32).tobytes())
        my_plot(conv_signal, "Image/Oscillator Convolution Waveform")
        my_fft(conv_signal)

    # Save convolved audio to file
    elif select == 3:
        write(f"WAV Exports/{Path(img.filename).stem}_{wf_type}_{osc_f}Hz_{round(osc_d/sample_rate*2,2)}s.wav",
sample_rate, conv_signal.astype(np.float32))
        print(f"File saved as {Path(img.filename).stem}_{wf_type}_{osc_f}Hz_{round(osc_d/sample_rate*2,2)}s.wav")

    # Quit
    else:
        quit = True
        print("Shutting down...")

# Close the stream and PyAudio object
stream.stop_stream()
stream.close()
p.terminate()
```

Appendix 3 – imagesave.py

```
### Package import ###

import numpy as np
from scipy import signal, fft
from scipy.io.wavfile import write
import pyaudio
from PIL import Image
import matplotlib.pyplot as plt
import os
from pathlib import Path


### Defining functions ###

def my_fft(data):
    # Create an FFT of the image data
    data_fft = fft.rfft(data)

    # Create a list of positions
    positions = np.linspace(10, 22050, 22040)

    # Create the plot (x,y)
    plt.semilogx(positions, np.absolute(data_fft[10:22050]))

    # Add axis labels and title
    plt.xlabel('Frequency')
    plt.ylabel('Amplitude')
    plt.title('FFT')

    # Show the plot
    plt.show()

def raster_scan(image, bi_sel):
    # Get the dimensions of the image
    width, height = image.size

    # Create an array to store the pixel values
    pixel_values = []
    # Iterates through every horizontal pixel position, top to bottom
    for y in range(height):

        # If row is at an even index, traverse from left to right
        if (y+2) % 2 == 0 or ((y+2) % 2 == 1 and bi_sel == 2):

            # Iterates through each pixel in the row
            for x in range(width):
                pixel_value = image.getpixel((x, y))

                # Add the pixel
                pixel_values.append(pixel_value)

        # If odd index AND bidirectional = 1 (True), traverse right to left
        # This 'snaking' through the pixel rows is raster scanning
        else:
            for x in range(width):
                pixel_value = image.getpixel(((width-x-1), y))
                pixel_values.append(pixel_value)

    return pixel_values

def bidirectional_select():
    while True:

        try:
            bidir = int(input(
"""
------------------------------------------------
Please select bidirectional (L->R then R->L)
or unidirectional (just L->R) raster scanning
------------------------------------------------

1. Bidirectional
2. Unidirectional


------------------------------------------------
Select 1/2
------------------------------------------------
"""))

        # Validates user input is an integer
        except ValueError:
            print("Please enter either 1 or 2.")
            continue

        # Validates that input is within desired range
        if bidir > 1 or bidir < 3:
            break
        else:
            pass

    return bidir

def normalise(data, max_min):
    # Find max and min values of all pixel colour data
    min_value = np.min(data)
    max_value = np.max(data)

    # Normalise pixel values between 0.95 and -0.95
    normalised_data = ((2 * (data - min_value) / (max_value - min_value)) - 1)*max_min

    # Validate the edge values to ensure no clipping will occur
    print(f"""MaxValue: {np.max(normalised_data)}
MinValue: {np.min(normalised_data)}""")

    return normalised_data

def fade_audio(unfaded_data):
    # Variable fade_length, 2400, is number of samples for fade in (50ms at 48kHz)
    fade_length = 2400

    # Create lists that go from 0 to 1 (fade in) or 1 to 0 (fade out)
    fade_multiplier = np.linspace(0, 1, fade_length)

    # Apply the fades to the audio data
    for i in range(0, fade_length, 1):
        # Image data
        unfaded_data[i] *= fade_multiplier[i]
        unfaded_data[-1-i] *= fade_multiplier[i]

def my_plot(data, plot_name):
    # Create a list of positions
    positions = list(range(len(data)))

    # Create the plot (x,y)
    plt.plot(positions, data)

    # Add axis labels and title
    plt.xlabel('Sample No.')
    plt.ylabel('Amplitude')
    plt.title(f'{plot_name}')

    # Show the plot
    plt.show()



### Declaring parameters ###

dir_path = os.path.dirname(os.path.realpath(__file__))
sample_rate = 48000
channels = 1
format = pyaudio.paFloat32


### Image intake ###

# Open the image
img = Image.open(f'{dir_path}/Greyscale Images/rgb.png')
width, height = img.size

# Display information about the image
print(img.format)
print(img.mode)
print(img.size)
```

```
# Show the image
img.show()


### Image raster scanning ###

bi_select = bidirectional_select()
pixel_values = raster_scan(img, bi_select)

# Using the normalise() function to scale the pixel data values appropriately
image_data = normalise(pixel_values, 0.95)

# Use the fade_audio() function to apply short fades
# at the start and the end to avoid any audible distortion
fade_audio(image_data)

### Sound synthesis ###

# Create a PyAudio object
p = pyaudio.PyAudio()

# Initialise the audio stream
stream = p.open(format=format,
            channels=channels,
            rate=sample_rate,
            output=True)

# Audition the image audio waveform
stream.write(image_data.astype(np.float32).tobytes())

# Plot the image waveform and FFT
my_plot(image_data, "Sonified Image Waveform")
my_fft(image_data)

write(f"WAV Exports/{Path(img.filename).stem}.wav", sample_rate, image_data.astype(np.float32))
print(f"File saved as {Path(img.filename).stem}.wav")

# Close the stream and PyAudio object
stream.stop_stream()
stream.close()
```

```
p.terminate()
```