**Project: Visualising music with processing**

**Background**

The basis of this project is replicating the effects of a specific type of synesthesia (chromesthesia) using the programming library known as Processing, which is an “built for the electronic arts, new media art, and visual design communities”[[1]](#footnote-2) and works on top of Java. It is described on its website as a “flexible software sketchbook”[[2]](#footnote-3) and is suited to this project, due to its integrated GUI and vast array of libraries for working with sound and graphics.

I have previously had experience working with Java in the first year of my degree, so the familiarity with the language should prove useful in working on this project, as the majority of the code is likely to be vanilla Java, with the specific graphical and audio elements written using functions provided in the Processing library. The Processing website provides a large number of tutorials for the basic functions provided in the Processing library and other associated libraries[[3]](#footnote-4), which should be useful in getting to grips with some of the languages fundamentals that I may be required to use.

**Introduction**

After some initial researching around the project and the Processing language, I recognised that it would be sensible to split the work for this project into some sub-areas, which would make it easier to break down the complex task at hand. The original idea would be to split the project as follows:

1. Compiling statistics in real-time from music
2. Investigating the desired visual output for the program, in respect to imitating the effects of synaesthesia
3. Implementing the visual output of the program in real-time, based on the compiled statistics

Part 1 would involve producing the back-end code to turn an audio sample into a set of usable statistics, part 2 would involve mainly research, while part 3 would involve producing the front-end of the program, resulting in a visualisation that could accurately imitate the effect of synaesthesia.

In terms of a time scale, parts 1 and 2 should be relatively quick to complete, since the code for part 1 will presumably be more generic and draw from readily available libraries and part 2 will simply involve compiling research about the condition. Part 3 will involve producing the more specialist code to give the visualisations desired and therefore I believe this will constitute the majority of the code produced and time spent on this project.

**Part 1: Compiling statistics in real-time from music**

The deliverable for the project is specified as “*A standalone system, written using Processing, that will take as input an arbitrary* ***MP3 file****, and visualise it, where the visualisation features can be clearly correlated with properties of the music.”*, with the properties of the music that are given as an example being *“frequency, amplitude, key, and beat detection”.* Therefore, for this part of the project, I will be required to produce code to extract these features from a given MP3 file.

The fact that I am required to use an MP3 file may prove an issue, since the audio is compressed and I believe that it will be impossible to implement a system that will work with variable bit-rate files in the time required – I will begin by looking at the theory behind extracting these properties from an uncompressed WAV file, then separately approach the issue of converting this code to work with constant bit-rate MP3 files.

Initially, I will intend to implement the extraction of these properties as functions that can be applied to a short sample of sound, rather than attempting to apply it to a real-time piece of music that is playing. Once the functions are produced, it should be simple enough to adapt this to a piece of code that samples the playing sound at regular intervals and applies this function repeatedly, giving the output in real-time.

I will therefore look at each of these properties that I wish to extract individually and determine if there is a simple known method to perform this that can be written in Processing, or even if Processing contains a library to perform the given function:

Frequency (and key)

Since the program is intended to work with music files, rather than a recording of a single note being played, it is likely that any short sample of audio will contain multiple frequencies of sound being played at the same time. Therefore, it would make sense to convert a short audio sample from the time domain to the frequency domain to give all the frequencies present in that sample of recording and the amplitude of these. Once this has been done, background noise could be removed by using a threshold, which could be absolute or dynamically calculated based on the amplitudes given, or I could even simply remove all except a given number (e.g. 3) of the highest frequency bands present in the sample. This would also allow the key of the music given to be calculated, by taking the frequencies given, determining the notes these correspond to and comparing these to the notes that make up each musical key.

(<https://processing.org/tutorials/sound/> - FFT method derived from here)

The conversion of a sample from the time domain to the frequency domain is something I have previously performed using Python in COMP28512, using an FFT (Fast Fourier Transform). Conveniently, Processing also contains an FFT method, which similarly converts a sample of audio from the time domain to the frequency domain in a given number of bands.

My concern with working in real-time was alleviated once I started looking at the way sound works in the Processing sound library, which conveniently allows an FFT to be produced for a sound sample in real-time. I chose Processing's own Sound library over the 3rd party librarys Minim and Beads, since it is a more recent development, with lots of support and seems to include all of the features that I might require for this project.

The FFT takes 2 parameters – the SoundFile, which is produced from the MP3 file and the number of bands to split the audio into. Since the lowest musical notes (C0 and C#0) are separated by just under 1Hz and all higher musical notes are separated by a larger amount than this, it seems reasonable to split the audio into bands of 1Hz and that this will allow accurate rounding of significantly amplified samples to the nearest musical note.

This was my initial plan for the FFT, but I disregarded the fact that the FFT must have a number of bands that was a power of 2. Since the majority of MP3 files have a sampling frequency of 44100, we can use this to work with a number of bands that should give us approximately the precision of 1Hz per band, but the exact precision will not be possible. Thus, we can set the number of bands separately and work out the frequency of the sound given by taking half the sampling frequency (the range of the FFT), dividing it by the number of bands and multiplying by the band that has the highest amplitude, giving the lower bound of the band, which should be accurate enough, given that the bands are relatively small.

I tested this with an audio sample with notes of known frequency and it seemed relatively accurate, with the values being accurate enough at sensible pitches even with only 4096 bands – I will stick with this number now for testing. The next stage is to convert this from a frequency value into a note name and number. fn = f0 \* (a)n is the equation used to determine notes from a single given note, where f0 is the given note, a = 21/12 and n is the number of steps away the note is. Therefore, we can start with the lowest given note C0 and work up in single steps, until we find the note given, simplifying the formula to f1 = f0 \* 21/12. Since we must also include sharps and flats, initially I will work out how many semitones displacement there is, then divide it by 12 to get the octave and use the remainder as the index into an array holding the note names. In order to improve the accuracy, I also checked if the note was closer to the note after the one given and changed if necessary.

Rather than using a for loop to test values of n to find the closest note, we can actually rearrange the formula using logarithms to find the value of n, then round this to its nearest value. Since fn = f0 \* 2n/12, we can rearrange to find n, then use the Math.round function to get the nearest note.

I believe that while finding the highest amplitude note is useful, it may also be very important to find the octave band which has the most total amplitude, to determine whether the overall sound of the piece of music is high or low pitched. I therefore intend to add up the amplitude of the bands pertaining to each octave, find the maximum and also present this.

Amplitude

There are two different types of amplitude that I initially thought to measure, which could both be combined to describe the sound at any given moment. The first is the amplitude of the single frequency that is selected during the previous part – this is easy to obtain, as it is simply the value of the band of the selected frequency. The second is the total amplitude of the FFT, i.e. the total amplitude of all of the bands that make up that sound signal. It would be useful to consider the maximum and reasonable values that these might take and express the amplitude as a percentage, either relative to the rest of the given track or in general.

One way to achieve this would be to add another circular buffer that contains the amplitude of a number of previous samples, then calculate the mean and variance of the samples already given. These can then be compared to the current value to determine how 'loud' or 'quiet' it is.

After considering this further, I decided that the mean and variance were less important than the 'minimum' and 'maximum' amplitudes encountered during the previous samples. We can then determine where the current sample sits within this window and determine how 'loud' it is based on this.

Key

I considered one method for determining the key for a song would be to record the note that is being expressed on the screen in an array, then once a number of frames has passed, determine the most frequent notes and calculate the key of the music from this. This involves holding an array, which will correspond to each musical note, where the corresponding value is incremented every frame that a note is selected for. After a certain number of frames, we can go through and determine which set of 7 notes in the array has the greatest number of hits and this can be decided as the key. Considering that a song may change key, it may make sense to reset the array once a certain time has passed or only consider a certain number of previous frames. I will attempt to achieve this through the use of a circular buffer in addition to the total amount of each note – when adding a note, I will first check in the circular buffer for the note to remove, then overwrite this in the circular buffer, so that there is always a total of n values in the array, where n is the size of the circular buffer.

Beat Detection (determining BPM)

(derived from method described at <http://mziccard.me/2015/05/28/beats-detection-algorithms-1/>)

1. Produce cyclic buffer, i.e. an array, with a certain number of blocks (i), each containing a number of samples (s).
2. Add the total amplitude of each sample (or of a certain subset of the bands, could also be a 2D array containing multiple bands of audio), to the given block, filling all of the blocks until the array is full – this results in initial delay
3. Now start filling block 1 (with the i+1 block considered in total), then consider if each filled block is bigger than C \* average block energy. C will be discussed later
4. If a beat is detected, record the time stamp of this sample (middle of the block?) in another circular buffer. Compute the average time between the samples in the second circular buffer (obviously ignoring empty cells initially) and use this to determine the BPM

Consider that the BPM of a song is expected to be in the range of approximately 24 to 250 BPM <https://en.wikipedia.org/wiki/Tempo>

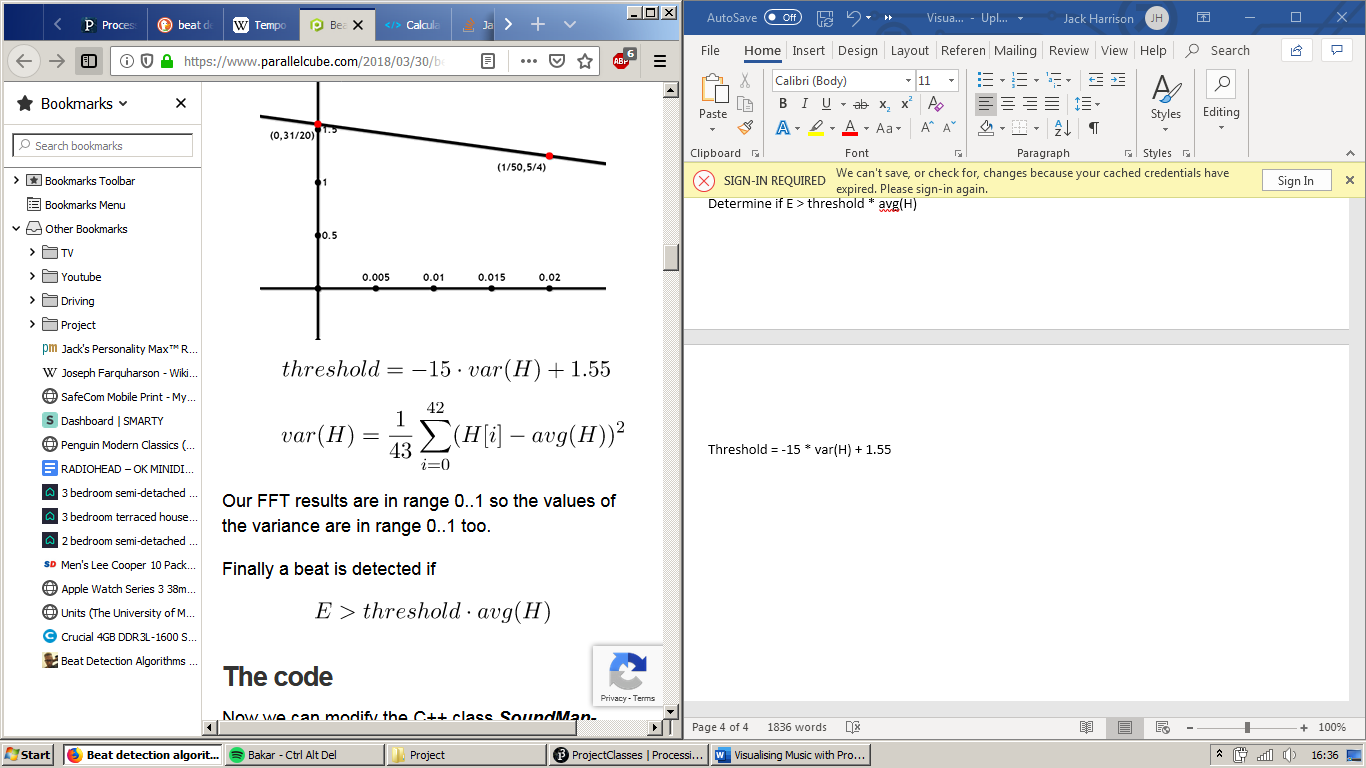
Therefore, in order to detect the slowest beats, we must consider a total time in the circular buffer of around 3 seconds. The fastest beats will occur at a maximum of 5 times per second, so in order for the beat to occur at more times than the average, we must sample at least 2 non-beats for 1 beat, therefore I propose to sample 20 times per second in order to be safe, so give the circular buffer 60 blocks, which at minimum should contain a single sample each. The number of samples in each can be linked to the framerate, so that if the framerate is increased, the number of samples in each block increases accordingly.

The second circular buffer size will have to be tested, I will initially give a size of 10 values, but this may be too short for high bpm tracks.

Considering the value of C, one case is given at <https://www.parallelcube.com/2018/03/30/beat-detection-algorithm/>

Determine if E > threshold \* avg(H)

Threshold = -15 \* var(H) + 1.55



So if X > (-15 \* var(array) + 1.55) \* avg(array), then the sample is a beat

The initial attempt with taking single samples in each block did not work at all, since each beat in a general song was made up of multiple samples and was causing the BPM to register as once per sample for the time during the beat and then for the actual time for the beat to be skewed based on the averaging from the previous ‘beat’ times. Therefore, to maintain the capacity to read the beat of a piece of music that is 250 BPM, we must keep the blocks less than around 1/600th of a minute, or 1/10th of a second. Therefore, I propose to raise the frame-rate of the sound from 30 to 60 frames per second and produce blocks of 6 samples in length. I also think it may be useful to average over a longer number of ‘beats’ to hopefully avoid the wrong value caused by 2 samples that are part of the same beat getting interpreted as separate beats causing problems with the final value.

Additionally, the article above also considers the fact that the beat of a song is produced mainly by the kick and snare drum, thus we should only be considering the frequency bands that contain these instruments. This is from 60Hz to 130Hz for the bass drum and 301Hz to 750Hz for the snare drum. Therefore, rather than using the total amplitude of a sample, as we have previously been, I will now calculate simply the total amplitude of these frequency bands and pass this into the function instead.

In order to prevent samples that are part of the same beat being detected as multiple beats, if we are only allowing a BPM of up to 250BPM but consecutive above average samples would suggest a BPM of much higher than this, we can simply ignore consecutive values in the table and only calculate differences where the difference is greater than one.

If there are consecutive values, the beat will actually be the length of the beat between the two end values plus the number of consecutive pairs (assuming that there are the same number at the end of each consecutive values, though this can be calculated using an average). Therefore, the length of time between beats is in fact (distance between non-consecutive values + number of consecutive values) / number of non-consecutive values). This is slightly skewed however, since it is likely that the circular array will hold consecutive values at the start and the end, where in reality we actually want to add the average number of consecutive values between non-consecutive. Therefore, I modified the algorithm to work this out, which also required expanding the BPM circular buffer to hold more values, since there are regularly a large number of consecutive values. Using this algorithm, for an extremely clear piece of audio, the BPM is almost perfect. The challenge, however, is when a less clear song is used, which causes the beat to be wildly off. This may require fine-tuning the frequencies selected for the beat or the equation used to determine if a sample is classified as a beat, but this can now be determined through trial and error, since we know that the code can give an accurate value with a clear signal.

**Part 2: Investigating the desired visual output for the program, in respect to imitating the effects of synaesthesia**

Synesthesia is a phenomenon where a person experiences “the subjective sensation of a sense other than the one being stimulated.”[[4]](#footnote-5) The condition is thought to exist in 1 in 2000 people and has been described as far back as 1690[[5]](#footnote-6). There are many different types of synesthesia, some of which I will outline below:[[6]](#footnote-7)

* Grapheme-colour synesthesia – associating specific colours with letters/numerical digits
* Chromesthesia – associating sounds with certain colours, affects around 1% of the population
* Hearing-touch synesthesia – developing tactile feelings when hearing certain sounds
* Lexical-gustatory synesthesia – associating words with a certain taste
* Mirror-touch synesthesia – feeling exactly what others feel, i.e. an extreme form of empathy
* Number-form synesthesia – having an involuntary mental map of numbers

The type of synesthesia we will focus on is chromesthesia and the idea that we should be able to replicate what a person suffering from this condition sees using a computer program. The issue with replicating this is that the condition is highly idiosyncratic, in other words, each person with the condition associates different colours with different notes/pitches/amplitudes of music. I would suggest that it is therefore reasonable to allow the program to accept a number of parameters for colours that the individual associates with a certain note, which the program can then use.

Additionally, since we are working with complex pieces of music, rather than single notes, we have to consider how a synaesthete may perceive a range of notes being played at once. One study states that “If several different notes are played simultaneously, as in a chord, most synaesthetes report experiencing several colours rather than a fusion of colours”[[7]](#footnote-8), yet this implies that some do experience a fusing of colours. Yet, since we cannot expect to represent the experience of every individual who experiences chromesthesia, I suggest that it is reasonable to present a piece of software that represents the experiences of a “majority”.

To keep the initial development simple, I believe it is reasonable to initially work with a single set of colours that are described from a single individual who experiences chromesthesia. The study that I have chosen to base my initial program on[[8]](#footnote-9) centres on the case of a middle-aged American women who claims to suffer from chromesthesia and gives clear pitch-colour pairings that she experiences:

* A – Lavender - ABC
* B – Orange - ABC
* C – Red - ABC
* D – Blue - ABC
* E – Green - ABC
* F – Brown – ABC
* G – Black - ABC

She states that “high octaves of a tone tend to evoke a lighter colour value, and lower octaves a darker value”, as well as that “'Black key' pitches were reported to elicit a greater colour intensity”, representing a merging of the colours on either side, e.g. D# giving a blue-green.

This seems to be enough information to create a basic template for the program, initially focussing only on the note being played, then expanding to also consider the amplitude of the sound and the BPM/speed of the music.

Since the effect of amplitude and BPM on synaesthetes is not widely reported, I will initially decide myself how these will be represented in the program. Amplitude is roughly representing the amount of noise given by the music, so I believe that it would be sensible to use this to determine how much of the screen is taken up by the colour given i.e. the size of the colour. The BPM represents how fast paced a song is and therefore I think it should be used to determine how quickly the colours change on screen. Representing every single 'frame' of sound would generally be too quickly changing for the user, so within a certain range of speed the BPM can determine how quickly one frame fades into the next.

**Part 3: Implementing the visual output of the program in real-time, based on the compiled statistics**

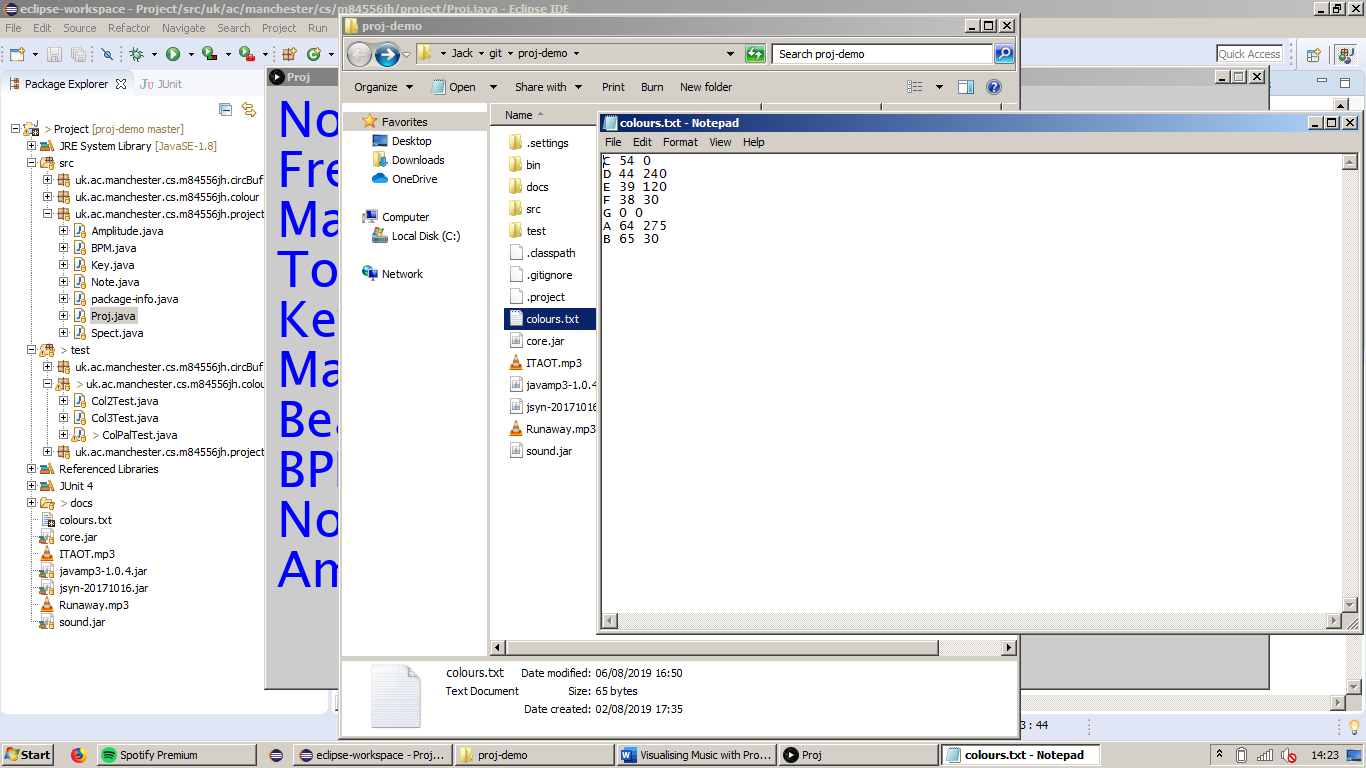
Before any visual representation can be created for the GUI of the program, the first step will be to convert the compiled statistics from the first part into values that can be used to create the colour representations on screen. There are 3 conversions that need to be made – pitch → colour, amplitude → size percentage and BPM → fade speed, each of which should be able to be accomplished using a function containing a number of calculations. I will detail the process of determining each of these below:

Pitch → colour

Processing allows us to consider colour as RGB values or HSB (Hue, saturation and brightness) values[[9]](#footnote-10). Since each note has a certain colour and the brightness needs to be adjusted depending on the octave, I believe that the most sensible way to represent colours in this application will be using HSB.

The program can initially be set up by defining the hue and saturation values for the notes A, B, C, D, E, F and G, with a function calculating the values for the sharps and flats between these by averaging the hue and saturation values for the values either side. An array of hue and sat values can then be stored, with the indexes corresponding to the note indexes already used in the program.

To allow the program to be personalised with a user’s choice of colours, it makes sense to read in the hue and saturation values from a file. The reading of the file can be achieved in the function to set up the array of colours, using standard Java code for file management. The file will be read from C to B and will take the Hue value, then the Saturation value. I have set up the basic file using preset colour values[[10]](#footnote-11), which can be seen in the screenshot below:



**These values are based on using a range of 0-255 for Hue, 0-255 for Sat and 0-100 for Bri.**

I produced three different classes in a new package, all relating to colour – Col2, Col3 and ColPal. ColPal stores the colour palette computed from a given file, which includes an array of Col2 – these simply contain two integers (hue and saturation), so represent the colour of a note in any octave. Col3 contains all three colour properties (hue, saturation and brightness) and is used to store the colour of an exact note.

Since we are also determining the overall key of the music, this can be seen as the overall 'colour' of a piece of music at a given moment. Some ways to use this would be to use the root note of the key as the background colour of the window or alternatively as the colour at the 'centre' of the visualisation.

Amplitude → size percentage

One way to produce a visualisation based on the colours of a set number of samples would be to have a circular buffer, containing a number of pixels. This could be added to each frame with a set number of pixels from the colour of the most recent frame, then read out backwards to the screen. It seems that it would be sensible to add 5 pixels of colour to the buffer for each frame added, since this will allow a maximum of 100 samples in a 500 pixel buffer.

Since the size of the visualisation is dependent on the amplitude, the number of pixels read will grow and shrink with each frame. If the amplitude is bigger than the previous frame, it seems sensible to just extend the size with the new colour, so that the amplitude of a frame is represented by the number of pixels corresponding to the colour of that frame. If the amplitude is smaller than the previous frame, we can simply add the 5 pixels of colour for that colour and read out the number of pixels required.

Due to the fact that I am now using circular buffers in multiple places in my code, it seems sensible to produce a circular buffer generic class, which can be read forwards and backwards and keep track of the front pointer. This is quite complex to implement and can be confusing, so many suitable tests will need to be added for this class.

After many attempts at different implementations of the circular buffer, I finally settled on an abstract generic circular buffer class, which would provide a write function. The subclasses that extend this would be for each circular buffer 'type' required (Col3, Integer, Double) and include their own read function, depending on whether the specific type of circular buffer should be read forwards or backwards. I then moved all of these circular buffer classes into their own package, to keep them separate from the rest of the program code.

**Parts to add**

- BPM -> Speed of sound changing

- Produce visualisation from stats

- Add dialog box for user to enter location of colour.txt file

- Add help file

- Add dialog box for user to enter location of music file

- Use <http://www.sojamo.de/libraries/controlP5/> or <https://github.com/brendanberg/interfascia> or <http://www.lagers.org.uk/g4p/> for GUI?

1. https://en.wikipedia.org/wiki/Processing\_(programming\_language) [↑](#footnote-ref-2)
2. https://processing.org/ [↑](#footnote-ref-3)
3. https://processing.org/tutorials/ [↑](#footnote-ref-4)
4. https://www.thefreedictionary.com/synaesthesia [↑](#footnote-ref-5)
5. https://www.telegraph.co.uk/culture/art/3653012/The-man-who-heard-his-paintbox-hiss.html [↑](#footnote-ref-6)
6. http://synesthesia-test.com/ [↑](#footnote-ref-7)
7. Ward, J (2006). Sound-color Synaesthesia: to what extent does it use cross-modal mechanisms common to us all? Cortex, 42, 264-280. [↑](#footnote-ref-8)
8. Haack, P. A., & Radocy, R. E. (1981). A Case Study of a Chromesthetic. *Journal of Research in Music Education*, *29*(2), 85–90. <https://doi.org/10.2307/3345016> [↑](#footnote-ref-9)
9. https://processing.org/tutorials/color/ [↑](#footnote-ref-10)
10. http://www.workwithcolor.com/color-chart-full-01.htm [↑](#footnote-ref-11)