Prototyping a Rhythm Game that Provides Automatic Level Generation Options

by

JAKE ROBERTS

Department of Computer Science

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Master of Science by advanced study in Computer Science in the Faculty of Engineering.

September 2023

Declaration

I declare that the work in this dissertation was carried out in accordance with the requirements of the University's Regulations and Code of Practice for Research Degree Programmes and that it has not been submitted for any other academic award. Except where indicated by specific reference in the text, the work is the candidate's own work. Work done in collaboration with, or with the assistance of, others, is indicated as such. Any views expressed in the dissertation are those of the author.

Jake Roberts, September 2023

This project fits within the scope of blanket ethics application 15765, as reviewed by my supervisor Peter Bennett.

Executive Summary

A standard rhythm game features pre-designed levels set to a limited catalogue of music. In the age of music streaming and with ongoing developments in the field of MIR (Music Information Retrieval), it makes sense to explore frameworks that allow such levels to be generated by the game itself to any given track.

In this project I present the design and implementation of both 'Autorhythm', a game built in Unity, and an accompanying Python server. Together these provide a framework for users to select generation options and have a rhythm level made automatically for a given piece of music. Users can then play, save and edit these levels.

Notably, this project:

- Gives an overview of the history of rhythm games and explores promising existing solutions to auto-generating levels for them using neural networks.

- Implements a general method for generating rhythm levels using musical onsets, featuring a technique to filter onsets relative to local strength based on given parameters.

- Suggests four methods of quantization in the interest of controlling level difficulty, and evaluates how users perceive the changes they bring.

- Looks at common existing formats for saving rhythm level scores, and proposes and implements a new save format in line with them.

- Designs, implements, and evaluates a generalised note-playing rhythm game which features an extensible system of automatic level generation, an input binding system with support for MIDI devices and most PC input devices, a system for saving and loading levels, and an editor view that allows for deletion, insertion and movement of notes in the level's score.

Acknowledgements

I would like to thank my personal tutor, Sarah Connolly, and my supervisor, Peter Bennett, for their guidance and motivation throughout both this project and the academic year.

I would also like to thank my family for their support, and my friend Bella Cooke for giving me the idea.

Contents

1 Introduction

The worldwide release of 'PaRappa the Rapper' in 1997 marked the start of a new genre of video game in which the core mechanic is getting users to tap along to an onscreen 'score' of notes in time with the game's music. This basic yet highly effective gameplay concept was taken and adapted into a plethora of highly successful titles, most notably the 'Dance Dance Revolution' and 'Guitar Hero' franchises.

Different games in this genre generally feature variations on this theme, such as notes that must be held down, different numbers of inputs available, and different styles of score ('Guitar Hero' scores will emulate guitar parts, while 'Rock Band' has different scores to emulate different instruments). Many games must be bought with their own specific controller, ranging from plastic guitars and drum sets to dance mats.

Despite these differences, these titles are all based around playing notes in-time with a given track, and so are often functionally the same. While it might not provide optimal player experience, there is no reason why a Dance Dance Revolution level could not be played using Guitar Hero controls, should a fan of the latter want to perform a song only available on DDR with a controller they have already purchased and are familiar with. This sort of interoperability is not available in these commercial games, however. Bespoke, often expensive controllers can even become defunct within their own game series [1]. This poses a serious sustainability issue, and this (combined with the sheer number of similar games being released) would eventually contribute to a saturation of the market around 2010 [2][3].

Beyond interoperability and sustainability concerns, the main drawback common to games of this genre is the limited catalogue of songs they come with. Once a user grows bored of the levels available to them, they must purchase an expansion pack or a new title to gain access to more. The specific genres and songs people enjoy vary, but it is generally impossible to buy specific tracks and users are bound by what the developers release.

With these core problems in mind, this project prototypes 'Autorhythm', a rhythm game that implements the pivotal idea of playing notes in-time with a given track, but allows the user to play along to any song from their library with any controller that can connect to PC. This is aimed primarily at two user groups: 'casual' fans of the genre who simply desire more content facilitated by freedom of music and controller choice, and 'experienced' fans who have a deeper interest in rhythm games and would look to use 'Autorhythm' as a tool for the development and testing of new rhythm levels.

While the creation of a rhythm level is always informed by the underlying music, it is still ultimately a subjective task and so this project's aims have a strong focus on providing the system with adaptability and room for future growth, rather than setting out to implement a single solution to level auto-generation.

1.1 Aims and Objectives

The specific aims for the game and reasoning for them are as follows:

Provide a user-customisable, expansible system of level generation.

Different songs lend themselves well to different generation techniques. Existing controllers are built with different level styles and difficulties in mind. Users need to be able to tailor generation options to produce the type of level they desire.

Additionally, the most promising current solutions to rhythm game level generation (discussed in the next chapter) are built/trained with specific song styles, controllers, and gameplay in mind. An expansible generation system allows solutions like these to be added as specialised options as and when they are developed in the future.

Provide a high level of controller support.

Returning players of other rhythm games may have old controllers they wish to reuse to play along to new songs. Controllers for rhythm games generally have no use beyond the title they were designed for, so allowing for their use in 'Autorhythm' gives them new purpose. New users may be happy playing with their computer keyboard, but the option purchase unwanted controllers from other rhythm games and use them provides a sustainable option.

The more comprehensive the controller support, the wider the appeal and usefulness of the application. Allowing for support of controllers not typically used to play rhythm games also paves the way for innovation into new level styles.

Save files in a format that encourages both interoperability with other games and future analysis.

Saving files as a basic requirement lets users replay and share levels they are particularly happy with. The format requirements come from thinking about these files as a resource outside the game. For 'Autorhythm' to be playable with any input device, it cannot feature any gameplay elements that could not be performed by simply pressing buttons. This includes Guitar Hero's 'hammer-ons' and 'pull-offs', which are dependent upon strumming technique as well as the notes pressed, and any games that feature 'hold notes', where a user must hold a button down for a period of time rather than just pressing it (which could not be performed by a MIDI drum set, for example, or any other device where the user cannot control when an input is released).

Since this is the case, a user may want to use 'Autorhythm' as a tool to create a new level for a song, and then use the resulting file in a secondary application where they can add these additional features manually, such as 'StepMania' or 'Clone Hero'. As such, the save files produced by 'Autorhythm' should lend themselves to conversion into other popular file formats for rhythm game scores and contain all the information necessary for this conversion.

Additionally, it is useful for output files to contain information on what generation options were used to create the level as this could be later analysed to improve generation methods and user experience.

Finally, as noted in 'Dance Dance Convolution', user-made online scores for rhythm games "represent an abundant and under-recognized source of annotated data for MIR research"[4]. MIR models should not be mistakenly trained on data that was itself generated using MIR techniques, and so steps should be taken to ensure that the scores created by 'Autorhythm' are not a hindrance to this informal dataset. The generation method used should be clear and it should be possible to see any changes or additions made by a human user.

Provide a way for users to edit levels.

What constitutes a 'good' rhythm game level is subjective, and there can be no generation method that outputs levels which every user always perceives as perfect. An editing function will allow casual users to correct perceived mistakes such as timing errors or difficult bunches of notes in a score they are otherwise happy with and will allow more experienced users to adapt and change a level to better suit a desired style or controller without having to create the entire score from scratch.

2 Background and Context

The term 'rhythm game' is used to broadly cover any game in which gameplay is dictated by the underlying soundtrack. This relation can come in many forms. In games such as 'Metal: Hellsinger' and 'Crypt of the NecroDancer', the user inputs for attacking/moving must occur in-time with the music, but there is no further influence of the music on the gameplay beyond this tempo-matching requirement. This paper is more concerned with 'note-playing' rhythm games, that is, games where the user must use a given input device to 'play' notes that are designed to correspond in some way to the accompanying soundtrack.

Many note-playing games, such as 'Friday Night Funkin', 'Rhythm Heaven' and 'Rhythm Doctor', utilise a call-and-response gameplay structure. The levels are designed so that the player's inputs must match the rhythm of the music to a more complex degree than the previous games, but the game will generally play these rhythms to the user first with some visual feedback to familiarize them, before they are prompted to play back the same or similar rhythm. Such games often rely on a bespoke soundtrack featuring songs that are composed with this back-and-forth in mind.

More crucial to this report are note-playing rhythm games that do not rely on an original soundtrack for the gameplay to function, such as 'Dance Dance Revolution', 'Guitar Hero' and 'Taiko no Tatsujin'. These games instead capitalise on the popularity of commercially available music and set their levels to existing songs. Tracks may be trimmed down and adjusted to emphasise tempo or a certain instrument, but the song itself is not specifically intended for use in a rhythm game and level designers must analyse it and attempt to place notes against it in a cohesive fashion. However, it is rare to find a game of this nature that lets users input their own music, and so they are generally limited to a certain number of pre-designed levels set to a certain number of licensed songs.

2.1 History of Note-Playing Rhythm Games

In this section we will investigate note-playing rhythm games, and work related to the automatic generation of content for these games.

2.1.1 Dance Dance Revolution (DDR)

Konami's 'Dance Dance Revolution' was first released in 1998 for Japanese arcades. Arcade versions feature a physical 'dancefloor' alongside the cabinet which acts as the input device, with 4 directional buttons for the user to step on in time with the music and visual 'score' onscreen.

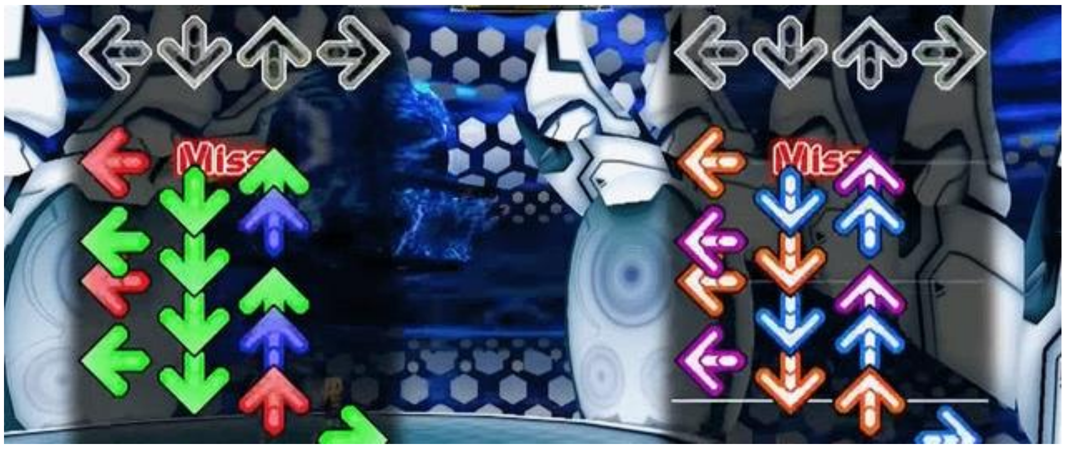


Figure 1: Example of how a DDR level appears onscreen [5].

The popularity of DDR is so high that new arcade versions and updates to existing cabinets are still being released [6]. Fans are also not restricted to these cabinets – following the success of the original game it was brought out on Playstation, and the following 20 years saw entries to the series available on Xbox, Nintendo consoles, mobile devices, TV/DVD and Windows, allowing the games to reach a large and varied audience.

The notes in DDR are not attempting to emulate any specific instrument and tend to follow the prominent rhythms of the song, as a dancer might. A stream a note falls in is not chosen based on the pitch of the underlying melodies but is instead chosen to create a cohesive 'dance' – for example, as noted in [4], level designers will try to avoid a sequence of notes that would cause players to turn around, away from the screen, on a dance mat.

2.1.2 Guitar Hero

In 1999, Konami released another note-playing rhythm game for arcade called GuitarFreaks. Similarly to Dance Dance Revolution's dancefloor, GuitarFreaks featured a controller custom to the game, modelled after an electric guitar with users holding buttons on the fretboard and strumming to play the onscreen score along with the music.

Using the same gameplay mechanics the company that helped create this hardware, RedOctane, would release the first game in what would become the Guitar Hero series in 2005. This series would become hugely successful, with Guitar Hero III passing $1 billion in sales just over a year after release [7].

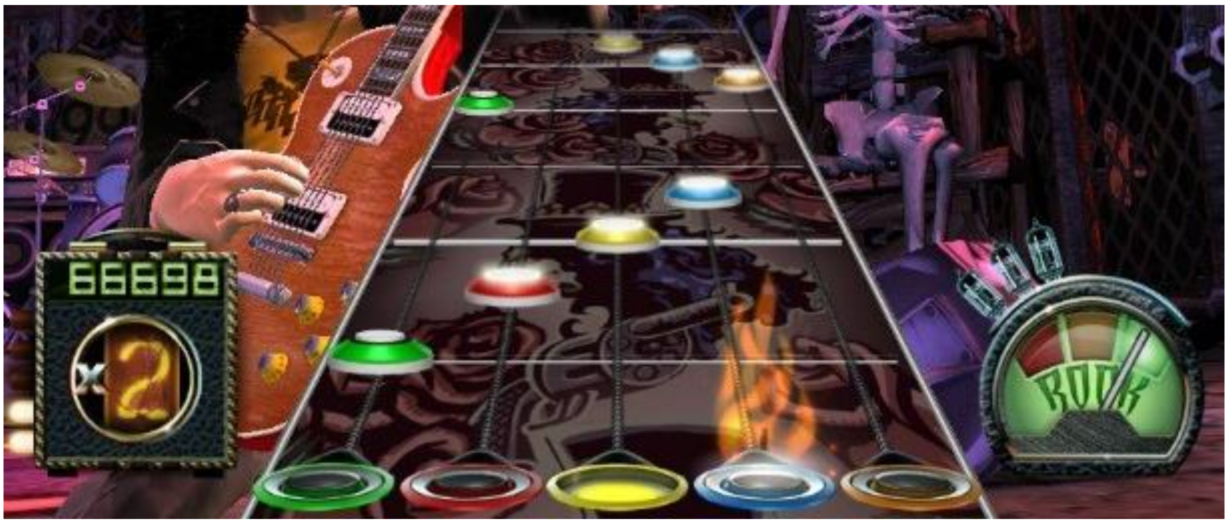


Figure 2: Example of how a Guitar Hero level appears onscreen [8].

Where DDR has 4 'note streams' (left, right, up, down), Guitar Hero tends to feature 5 with 'Guitar Hero Live' using 6. The notes that the user performs are also placed and selected differently. The notes presented to the user follow the guitar parts of the song being played. An ascending phrase in the guitar melody on the track will tend to be reflected in-game by an 'ascending' pattern of notes through the different note streams, though with a limited number of inputs each stream does not correspond to one exact pitch.

2.1.3 Others

While DDR and Guitar Hero are generally the most well-known note-playing rhythm games, many others exist. 'Taiko no Tatsujin' uses one large drum as the controller, with the user hitting either the rim or the centre to play the two different note streams. 'Rock Band' offers controllers modelled after drums and keyboards in addition to guitars. 'Beatstar' is a mobile game offering 3 note streams that the user must directly tap via the touchscreen.

While note placement, note selection and the number of note streams differ, it's important to note that these are all, at their cores, variations of the same game. A user is provided with an input device of some sort and is asked to use this controller to perform a score of scrolling notes as they reach a certain point onscreen. All these scores have been written so that the notes shown match or mirror the underlying music in some way.

2.2 Commercial Rhythm Games That Explore Generative Levels

Open-source options for generating levels often exist for popular rhythm games, and these are explored later in this chapter. However, it is rare for a commercial release to feature any such system, so it is worth looking at some examples.

An early rhythm game that experimented with letting the player generate levels from their own music was 'Vib-Ribbon', released for the original PlayStation in 1999. Players must use 4 different buttons on the PlayStation controller to dodge 4 different corresponding obstacle types that appear onscreen, in time with the music. While presented differently, this does fit into the same category of note-playing rhythm games as the other titles we have just discussed; the obstacles are the 'notes' that match the underlying soundtrack, and must be 'played' via user input when they reach a certain point on the screen.

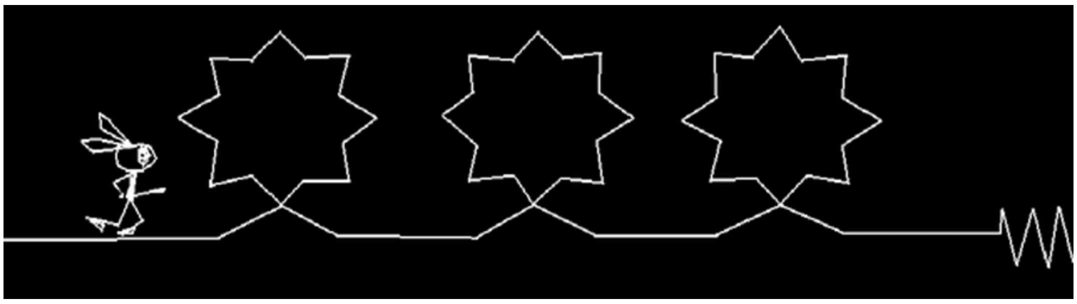


Figure 3: Example of how a Vib-Ribbon level appears onscreen [9].

The game is designed to generate levels from any CD the user inserts into the PlayStation. Obstacles are generated as a level progresses by analysing the proceeding 8 seconds of audio [10, pg 90]. While creative, level generation was naturally limited by the PlayStation's hardware and the MIR techniques of the time, and reviews often indicate that generating a level "usually results in really ridiculous obstacle design" [11] or can sometimes produce "utterly impossible and infuriating" [12] gameplay. Music information retrieval is constantly improving, and the generation options available to us now are much more sophisticated.

A more modern rhythm game that allows users to map levels to their own music library is 'Crypt of the NecroDancer', first released for PC in 2015. In this title, players hop in cardinal directions around a procedurally generated dungeon, defeating enemies and collecting coins. This can be classed as a rhythm game because every enemy the player faces moves in-time with the music, and the user is also expected to move on-beat if they want to achieve a good score. This does have elements of a note-playing rhythm game – lines converge at the bottom of the screen to give the player some visual feedback on when their inputs should be falling. These lines only represent the tempo of the song, however, and therefore have no rhythmic diversity or strict guidance on which input the user should press.

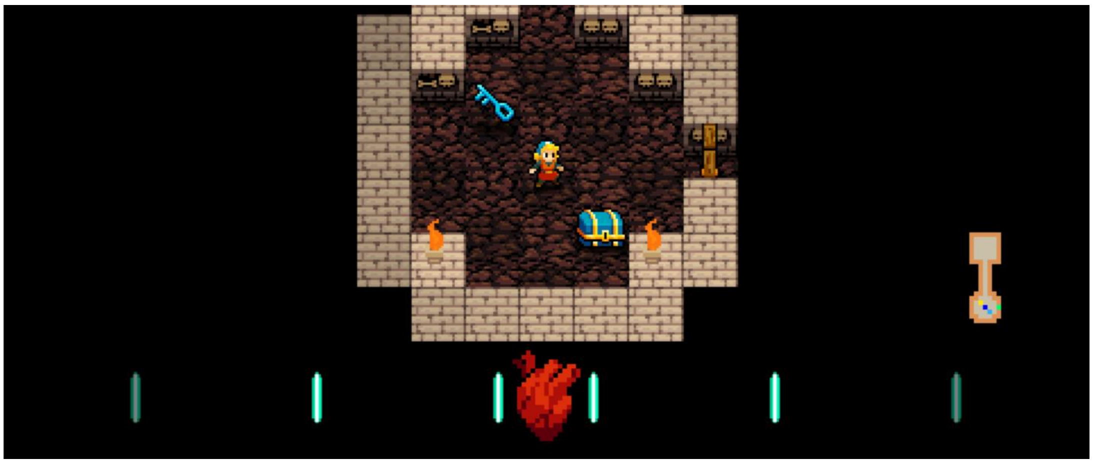


Figure 4: Screenshot of 'Crypt of the NecroDancer' gameplay.

Users are given the option to load songs from their own library to play along to, but since gameplay does not reflect any rhythmic elements of the background music this feature extraction is limited to detecting the tempo of the uploaded song and syncing the game with it. Again, there is much more feature extraction that could be done to create a rhythm game that makes more detailed use of user-uploaded audio files.

2.3 Onset Detection

In music, the 'onset' of a note refers to its perceived starting point. A highly studied field of music information retrieval is getting a computer to automatically chronicle every onset in a piece of music, dubbed 'onset detection'. There are numerous methods of doing this, and all have advantages and disadvantages that make them suitable for different tasks. These methods traditionally involve reducing the audio waveform in some way to accentuate a specific signal feature related to the magnitude or phase spectra, and if the reduction method is well suited to the music being analysed this should result in some identifiable features that indicate the positions of onsets [13].

A common example of a way that features of music can be represented is with a mel-frequency spectrogram.

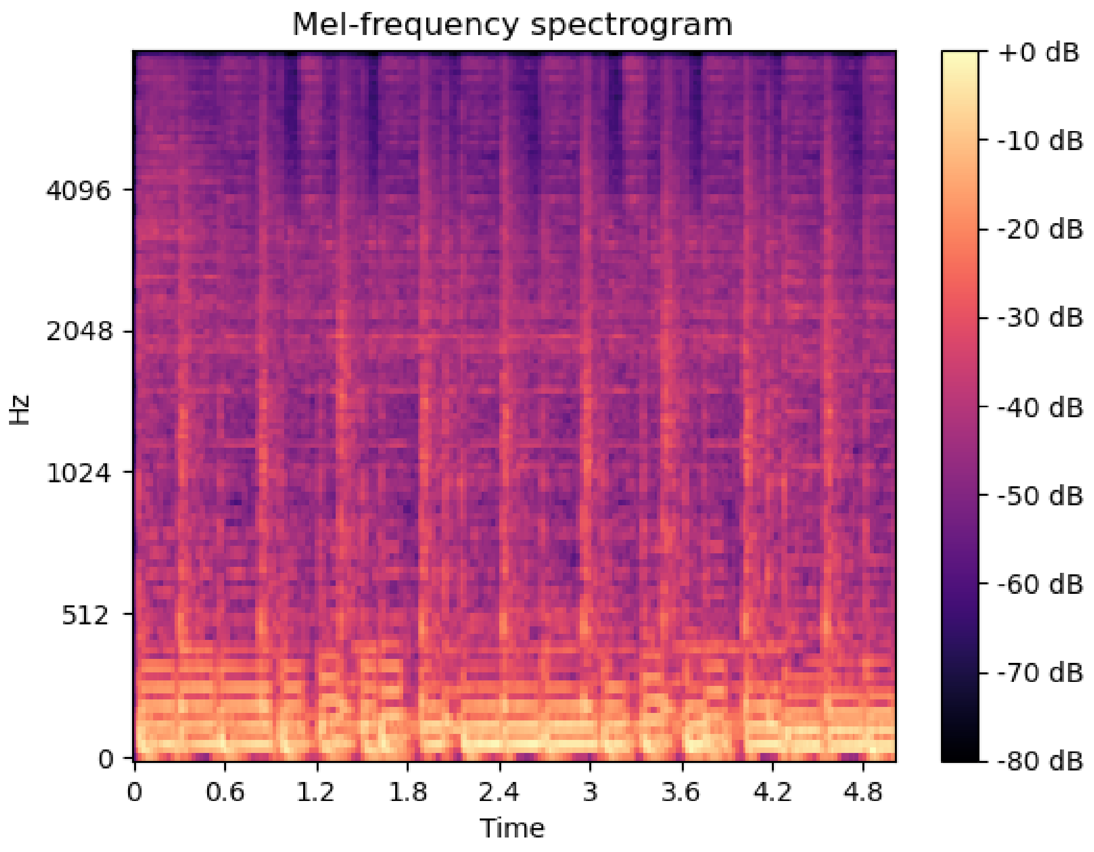


Figure 5: Mel-frequency spectrogram of the first 5 seconds of 'La' by Shane Iver [14], made using librosa [15]

The x-axis represents time while the y-axis represents frequencies under the mel scale, which is a unit of pitch based on human perception of sound rather than the actual frequency of the soundwaves. Humans find it much easier to hear differences in two low pitch sounds of similar frequencies than two high pitch ones of the same frequency difference, and so this scale gives a higher resolution at low Hertz [16].

With these axes, each pixel represents the combined magnitude of a small range of adjacent frequencies (or a 'frequency bin') over a frame of time. In the above diagram warmer colours represent larger magnitudes, and just by looking you can begin to tell where the onsets of some notes might be.

In 2014, Schlüter and Bock published a method of onset detection using CNNs in 'Improved Musical Onset Detection With Convolutional Neural Networks' [17]. CNNs are commonly used in computer vision solutions, often taking an image as input and returning labels based on what it 'sees'. However, as we have just seen, features of music that we wish to analyse can often be neatly conveyed as an image. It follows that you could try and train a CNN to label onsets on a spectrogram. In Schlüter and Bock's successful method, they took vertical slices of the spectrogram with the time frame to be analysed in the centre. The CNN was trained to attribute binary labels to these slices, based on whether an onset occurred at the central time or not. They found that this method outperformed other techniques at the time, and this methodology forms the basis for many models that attempt to autogenerate levels for existing rhythm games.

2.4 Adaptation of Onset Detection with CNNs for Automated Level Design

Popular rhythm games will often spawn fan-made clone games for PC, such as ‘StepMania’, which emulates Dance Dance Revolution, ‘Clone Hero’, which emulates guitar hero, and ‘Taikojiro’, which emulates ‘Taiko no Tatsujin’. These are born in part from users’ desire to play all the songs released for the respective series in one place, but also from the desire for new, user-made levels.

There are a large number of original levels available online for all these clones, and the creators of ‘Dance Dance Convolution’ [4] realised that these could be used as a dataset for training a neural network. In their work they use the same method as Schlüter and Bock, but rather than training the CNN to predict whether an onset lies in the central frame of a spectrogram slice, they have it predict whether a note would be placed there in a StepMania score. This process is improved by combining the CNN with a long-short term memory network, enabling the model more context for decision making. This addresses the problem of ‘note placement’, creating a strong model that determines at what times notes should be placed with a level. To solve the problem of ‘note selection’, which would be to take these note timings and determine which type of note they should be (which ‘note stream’ they should belong to), they employed a second LSTM.

This work was generally successful at producing accurate and playable levels for ‘StepMania’ and, using their techniques, similar models for ‘Clone Hero’ and ‘Taikojiro’ were developed [18][19]. The researchers for the ‘Clone Hero’ auto generator were unable to create a neural network for note placement, mainly since the guitar melodies they were attempting to set notes to are more difficult to detect than the general prevalent rhythm of a song, which is what ‘Dance Dance Convolution’ needed. They did create a successful network for note selection using an RNN which took into account ‘chords’ (multiple simultaneous notes) and the motion of sequential notes in regard to the motion of the underlying music. The developer for the ‘Taikojiro’ auto-generator found that the C-LSTM method of note placement used in ‘Dance Dance Convolution’ did not work when trained on ‘Taiko no Tatsujin’ scores, but after balancing the data by only using spectrogram excerpts from near the notes in the scores they were able to train the CNN model proposed in [17] to place notes.

3 Initial Design

3.1 Software Choices

3.1.1 Game Engine

A simple and accessible way for a solo developer to create a game is with the use of a game engine. This streamlines the process by providing tools and libraries for handling audio, rendering and inputs, which are crucial to this project.

The two most popular free-to-use game engines are 'Unity' and 'Unreal Engine'. Unreal is renowned for being able to render detailed 3D graphics and as such much of its functionality is geared towards the creation of advanced 3D games. Unity, on the other hand, has a more comprehensive 2D engine and is reportedly more accessible for beginners [20][21].

Beyond this, the main motivation for using Unity is the new input system it introduced as an optional library in 2019 [22][23]. This provides a way for Unity to listen for user inputs at runtime and bind these to a preconfigured control scheme for the game. Additionally, a public plugin called 'Minis' (MIDI Input for New Input System)[24] allows Unity to listen for and bind MIDI inputs to a control scheme.

Building 'Autorhythm' in Unity will therefore ensure it access to a large range of potential controller support with a simple way of integrating these controls.

3.1.2 External Server

While Unity is able to load audio from file at runtime and provides some methods to manipulate this music, it does not feature any MIR techniques as standard, such as beat tracking and onset detection. While these features can be implemented manually or with the use of open-source libraries [25], 'Autorhythm' aims to deliver an expansible and relatively future-proof method of level generation, that allows outdated methods of generation to be updated and new generation methods added as MIR technologies improve.

If initial generation methods are implemented into the Unity application directly, changing these methods requires updating the entire build of the game. As such, I opted to use an external server to handle the note placement aspect of level generation. The Unity application can send requests to this server, which will respond with a newly generated score. With this setup, the server could be entirely overhauled and all that would need updating in-game are the options available to the user.

While using a server hosted on the web opens lots of possibilities, discussed in the 'Future Work' section, for this prototype build I decided to create a python server to process requests that must run locally on the user's machine. This is not the most user-friendly option, as it requires an install of Python 3 and some additional packages on the user's computer and must be manually launched from a command line, but it has been chosen as it gives me access to the 'librosa' library [15] which allows for a wide range of audio analysis.

Python also supports machine learning networks such as PyTorch and TensorFlow, which could be used to provide generation options with neural networks like those discussed in the previous chapter. The implementation and development of such networks is beyond the scope of this project, but their importance should be recognised and an extensible system of level generation should be able to support them.

Another reason I chose to use a local server is because users will be providing audio files local to their machine to play with. The Unity application needs access to this file to load and play the audio in-game, and the server needs access to this file to perform analysis on it. Having both the server and application hosted on the user's computer means the user can copy their audio file into a designated local folder and both will be able to easily access it.

3.2 Default Note Placement

While a more experienced user might be happy to fiddle with generation options and controller settings before they can start playing, a casual or first-time user will likely expect to be able to quickly load a song from their library and play along to it. To this end, it is important to think about what generation options the user will be provided with by default, and most importantly how the system will place notes to the music by default. This method needs to produce generic levels which are not specialised to a particular input device and should be applicable to any style of song.

In order to achieve this generalised note placement, I decided to try and use the onsets of a song. It is noted in 'Dance Dance Convolution' that of the 'StepMania' levels they looked at, "while not every onset in our data corresponds to a DDR step, every DDR step corresponds to an onset"[4]. This logically makes sense; rhythm games task the user with performing actions in-time with music, so it follows that these actions will generally occur at the same time as a note in the track. By setting the in-game notes to the onsets of a song they should reflect its performance.

The onsets of a song can be easily computed with librosa. The onset envelope is calculated based on spectral flux, which is a measure of how the energy distribution across different frequencies changes over time in a given audio signal. The envelope gives a representation of how the spectral flux changes over time that accentuates significant changes, as these correlate with onset events and can be used to calculate where onsets are likely to lie.

Placing notes in this way left me with the problem of controlling the difficulty of the level. This does not tend to happen in practice, but if every single onset in a song was correctly identified and labelled then the resulting score would be the hardest possible for that song (beyond adding unfounded notes that do not correlate with onsets). The initial method I used to tackle reducing the number of notes in a level was via quantization.

3.3 Quantization

Quantization is the process of attributing rhythmic subdivisions to musical events, and correcting their timings so that they lie directly in-time with this subdivision. When considering the notes a user plays in a rhythm game, there is no strict requirement that the rhythmic subdivision of these notes need be known; the only truly necessary information for syncing notes to the music is the exact time at which they need to be played in relation to the song. However, I decided to quantize the notes generated by the server for several reasons.

Firstly, one of the aims of this project is to produce a level editor. Having the notes in the level's score limited to set rhythmic subdivisions means the user can only move notes on discrete time steps, which while potentially restrictive should be simpler and more intuitive to use than allowing the user to move notes over continuous time.

Secondly, another aim of the project is to provide save files for levels that encourage interoperability with other games of similar nature. I look in detail at other filetypes for rhythm games later, and all the examples studied have some indication of rhythmic subdivision associated with the notes in them. Conversion to these file formats would still be possible without the inclusion of this information, but in the interest of encouraging interoperability it is useful information to include.

Thirdly, of the three neural network solutions to score generation looked at in the background and context chapter, none attribute rhythmic subdivisions to the notes in their output scores. For example, 'Dance Dance Convolution'[4] considers sequential 10ms excerpts of an input song when placing notes, so can represent the output by setting the score to 125bpm with a resolution of 192 possible steps per 4 beats (or 1 measure, assuming 4/4 time signature). Four beats equate to 1.92 seconds at 125bpm, so having a resolution of 192 for each 1.92 seconds of music allows a clear mapping from their labelled 0.01-second sections of music to this score.

They do note that providing the network with beat phase and knowledge of how many beats lie between the previous and next note improved their step selection process, but they did not implement a way to determine these so left them out of the final model. Having the server quantize notes after note placement means that this information is now available for any note selection methods incorporated into ‘Autorhythm’ in future.

Lastly, as mentioned in the previous section, quantization can be used to simplify a level by being selective about which rhythmic subdivisions we allow and discarding notes that don’t map to them, thereby reducing the total number of notes and reducing difficulty.

3.3.1 Controlling Difficulty with Quantization

To design the quantizing functions, I first wrote functions to parse SM files (score files for ‘StepMania’) and find which rhythmic subdivisions notes were commonly placed on at each difficulty: ‘Beginner’, ‘Easy’, ‘Medium’, ‘Hard’ and ‘Challenge’.

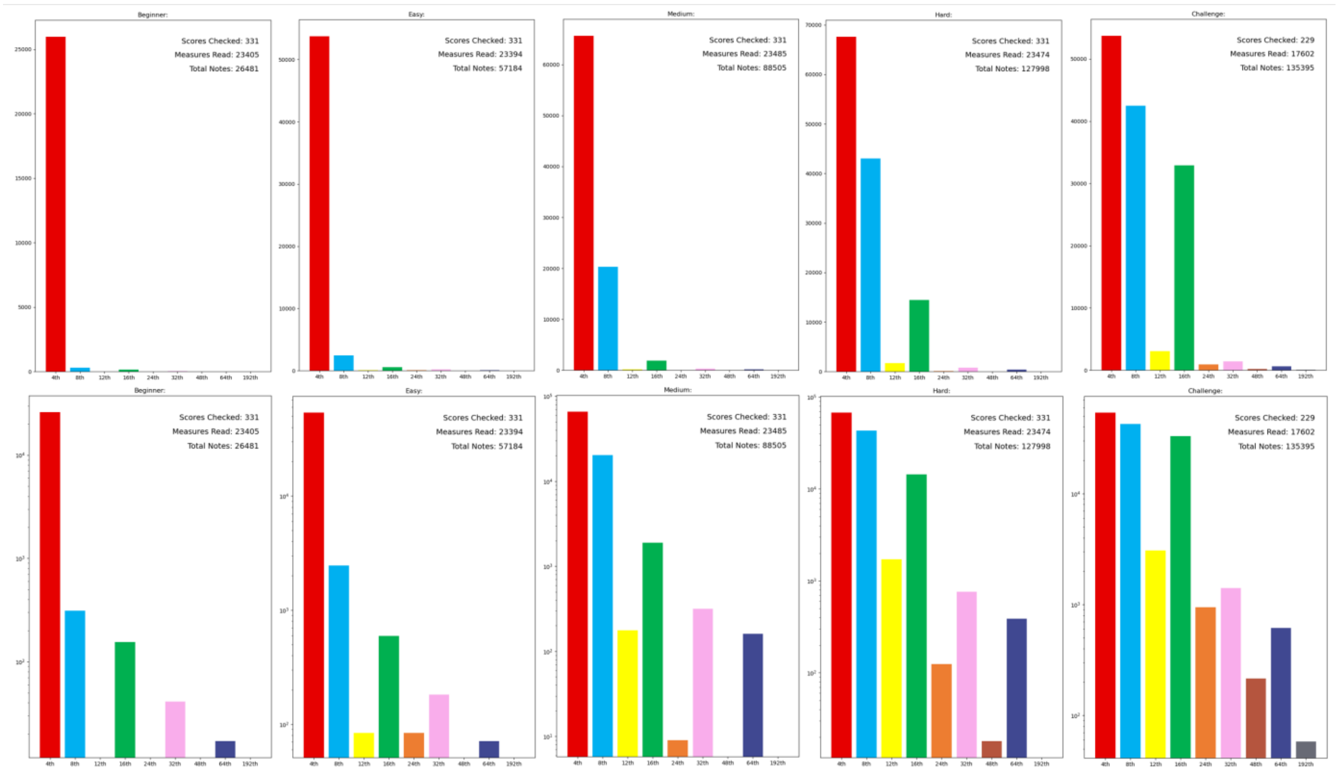


Figure 6: Rhythmic subdivision frequency by difficulty. Linear and logarithmic scales for each are included for clarity.

	BEGINNER	EASY	MEDIUM	HARD	EXPERT
% 4th notes	98	94	74	52.8	39.7
% 8th notes	1.2	4	23	33.6	31.4
% 12th notes	0	0.15	0.2	1.3	2.3
% 16th notes	0.6	1	2	11.3	24.3
% 24th notes	0	0.15	0.01	0.1	0.7
% 32nd notes	0.2	0.3	0.36	0.6	1
Average notes per measure	1.1314	2.4444	3.7686	5.4528	7.692
Note density relative to ‘Expert’	0.15	0.32	0.49	0.71	1

Figure 7: Numeric display of relevant note frequencies.

331 scores were included, sourced from the online collections ‘Fraxtil’s Beast Beats’, ‘Tsunamix III’, ‘Fraxtil’s Arrow Arrangements’, ‘In The Groove 1’ and ‘In The Groove 2’. This gave me a guideline on the common densities of the different beat phases used for notes at each difficulty.

Following this, I decided to only quantize notes to 4th, 8th, 16th, 32nd, 12th, and 24th notes. This gives an irregular resolution but captures the most common rhythmic subdivisions. While 12th and 24th notes occur less often and their exclusion would result in a regular resolution, they are important to include as they provide better compatibility for songs that are swung or have a time signature that gives 3 (or 6) counts per beat.

3.3.1.1 Expert

My first quantization method is standard, and simply snaps each note to its nearest rhythmic subdivision.

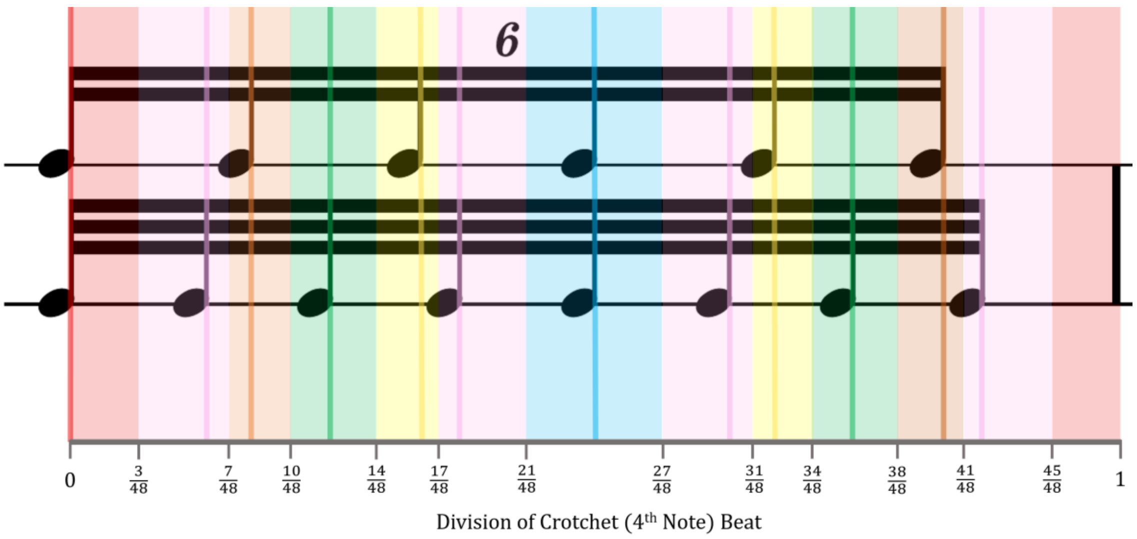


Figure 8: Visualisation of 'Expert' quantizing method.

If a note is nearest a beat phase which another note has already been snapped to, it is removed. Additionally, any notes lying before the first or after the last detected beat in a song are ignored since rhythmic subdivision timings can only be calculated if the length of the beat they belong to is known, and calculating this requires both the start and end time of a beat.

Even more notes could be retained by having all notes snap to their nearest available beat phase, rather than discarding notes whose nearest phase is taken, but this causes notes to move further from their true timings and could result in large clusters of notes that are difficult to play and do not truly represent the underlying music. As such, this quantization represents the maximum possible number of notes we can allow and so is designated as method for producing 'Expert' levels.

3.3.1.2 Hard

My analysis showed there are approximately 30% fewer notes per beat in the hard scores studied compared to the expert ones. To try and reflect this, I designed a quantizing method that ignores a third of each beat.

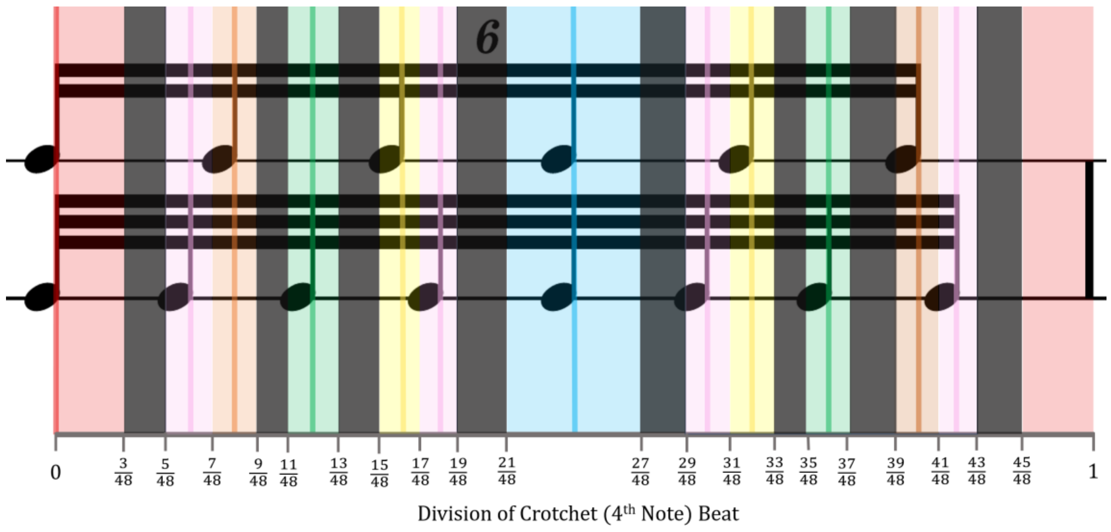


Figure 9: Visualisation of 'Hard' quantizing method.

Initially it seems that this method will produce levels with 33% less notes than the 'expert' quantization method does while reducing the precedence of the finer beat phases, but in practise the drop will be less than this.

The notes this method ignores could have also been ignored by the previous method due to their nearest beat phase being occupied. Additionally, the 33% reduction assumes that notes are distributed randomly which is not the case, as our note placement method places on onsets which in theory should fall on rhythmic subdivisions, meaning the probability of notes appearing in the grey zones above is likely lower than one third.

However, for this project I am using librosa's 'beat_track' feature to detect where the beats lie in the user's music. This method and the method of onset detection I am using are not perfect and can lead to notes being placed in the dead zones shown above. As such this quantization method should result in a reduction of the overall notes in a score to some degree by removing notes that are not near a common rhythmic subdivision.

I have also noted that the beat tracker can often place the beats on the 8th notes of a song, rather than the 4th notes, especially if the music is syncopated. As such I have chosen to leave the 'capture zones' for 4th and 8th notes the same size as each other, regardless of the difficulty I am aiming to emulate through the quantization process.

3.3.1.3 Easy

Both beginner and easy difficulties feature predominantly 4th notes. Since I am giving the same precedence to 8th notes as 4th notes in this prototype, I decided to create an easy difficulty that just considers 4th and 8th notes and to not implement a 'beginner' difficulty.

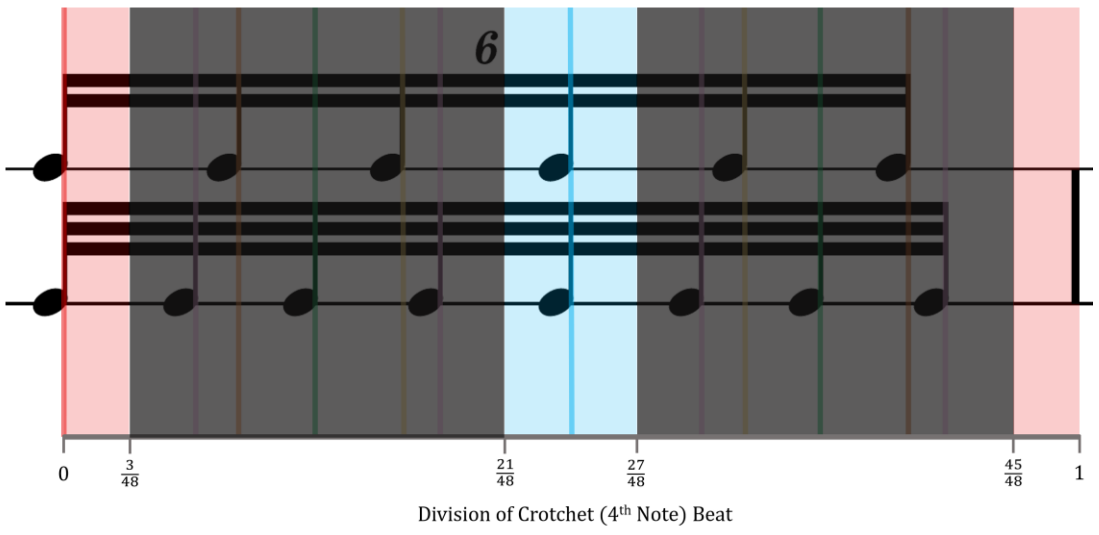


Figure 9: Visualisation of 'Easy' quantizing method.

3.3.1.4 Medium

To produce medium difficulty scores I decided to reuse the method for hard difficulty. However, the medium version only allows for one note to exist between each potential 4th and 8th note, shown by the red boxes below.

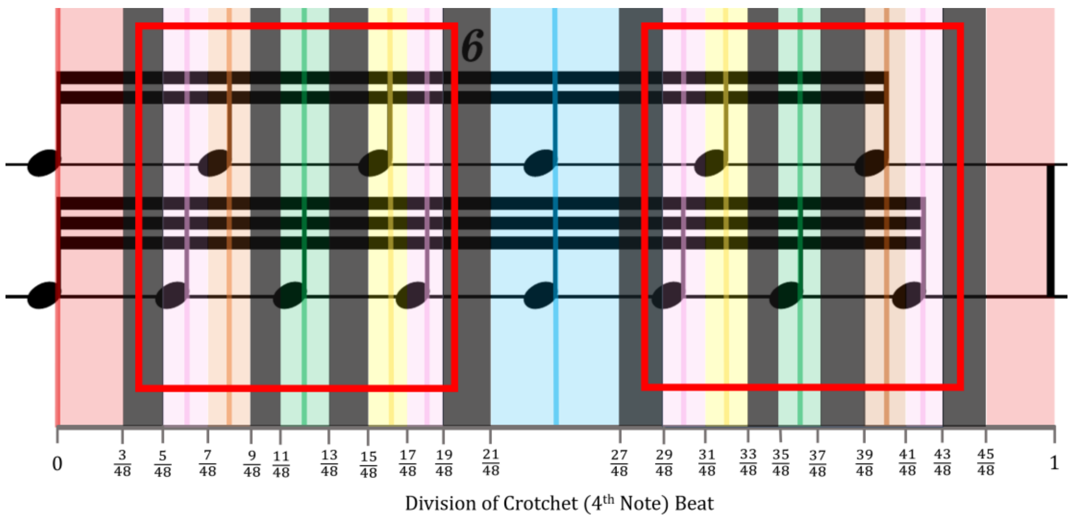


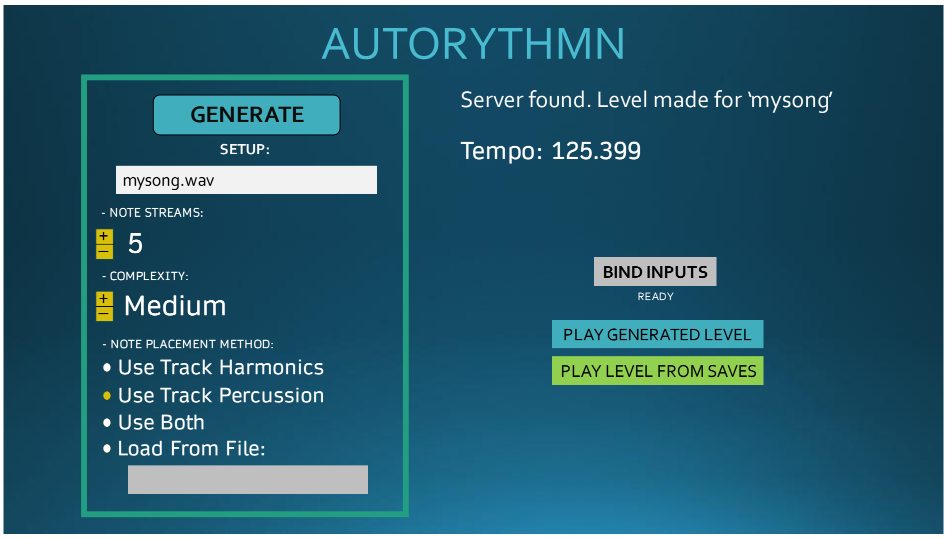
Figure 10: Visualisation of 'Medium' quantizing method.

This constricts the maximum number of notes per beat to 4 and reduces the total number of notes with beat phases smaller than an 8th, which are quite rare in the medium levels analysed. If more than one note could be quantized in these regions, the note whose raw timing is closest to its quantized timing is kept. This is used as the deciding factor as it does not favour any specific subdivision and minimises the overall amount note timings are shifted by.

The code implementing this can be found in the appendix, which encapsulates the logic used for all 4 quantizing methods.

3.4 Wireframing

With software selected and an initial method of note placement designed, I began to plan how 'Autorhythm' would be operated from a user's perspective and created an early wireframe design.


AUTORYTHMN

GENERATE

SETUP:

mysong.wav

- NOTE STREAMS:

5

- COMPLEXITY:

Medium

- NOTE PLACEMENT METHOD:

- Use Track Harmonics
- Use Track Percussion
- Use Both
- Load From File:

Server found. Level made for 'mysong'

Tempo: 125.399

BIND INPUTS

READY

PLAY GENERATED LEVEL

PLAY LEVEL FROM SAVES

Figure 11: Homepage wireframe design.

The intended interaction with this home page for level generation is as follows:

- User places an audio file into the 'autorhythm_songs' folder on their machine.
- User inputs the name of this song in-game, which the game loads. On success, the song begins playing in the background to indicate it has been retrieved. Hearing the song while moving

through the next stages of the generation process may also help a more experienced user decide which options will be best for it.

- User selects the number of note streams they wish to play with, between 1 and 6.
- User selects a difficulty, 'easy', 'medium', 'hard', or 'expert'. This determines the quantization method used.
- User selects the note placement method for level generation. Librosa has an implementation for harmonic-percussive source separation, allowing me to decompose an audio signal into its approximate harmonic and percussive components. I use this to provide additional note placement options; selecting 'Use Both' will cause the server to perform onset detection on the unaltered audio signal, while 'Use Track Harmonics' and 'Use Track Percussion' will instead use the respective filtered signal. This gives the user more control over which aspects of their song the generated level will try to follow.
- User clicks 'GENERATE', and a level is created by the server and loaded into the game.

An additional note placement option, 'Load From File', is also included. This is intended as an early way to let players use other note placement methods, such as the one described in 'Dance Dance Convolution' [4], by letting them import the result via file.

The right-hand side of this home screen gives the player a button to open the input binding menu, and once bound allows the player to play either a level they have just generated or a level from their save files. The latter option does not need a server connection, because the song and all information on the level will have been saved locally. Both options then take the player to the game view.

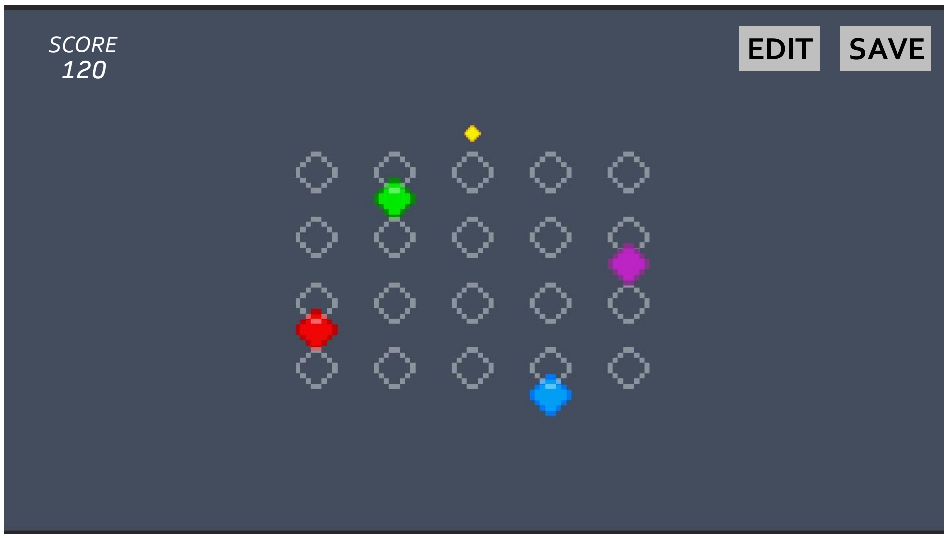


Figure 12: Game view wireframe design.

In the game view, the song restarts and notes begin to scroll upwards. Notes appear 4 beats before the time they are meant to be played at, based on the estimated tempo of the song, and should be 'played' when they align with the top row of note outlines. Four rows of note outlines are displayed, with the intention of giving the user a reference for distance so they can better judge how far a note is from the top row and how far notes are from one another. For example, in the above design you can tell that the next 4 notes are equally spaced one beat apart from each other and will need to be played at a constant rhythm.

The user is offered a score in the top left to gauge their performance, and buttons for saving and editing. The 'save' button writes the level to file and saves it in the 'autorhythm_scores' folder so the user can play it again later. The 'edit' button takes the user to the editor view.

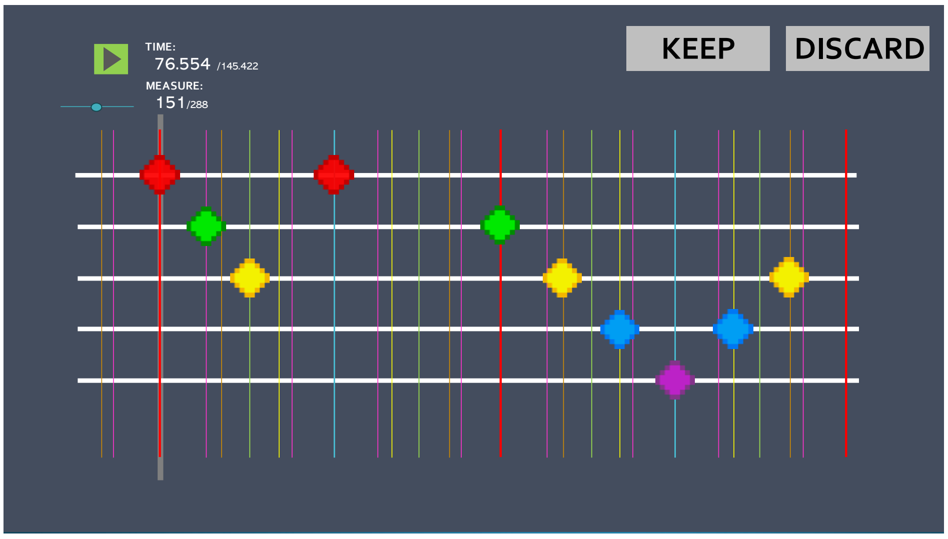


Figure 13: Editor view wireframe design.

In the editor view, the user can navigate the entire score, placing, deleting, and moving notes where they see fit. Notes can only exist on the rhythmic subdivisions discussed earlier, indicated by the coloured lines in the editor.

When a note is moved, its timing is recalculated and its new beat phase is updated in a temporary copy of the score. When the user is finished editing, they can choose either to keep their changes, replacing the old score with the new one, or discard the changes and continue playing with the old score.

4 Initial Evaluations

4.1 Note Placement

In my initial designs, I have proposed the use of onsets for note placement and the use of four different functions to quantize these notes and reduce the difficulty of a level by selectively removing some. To evaluate this solution, I wanted to gain an understanding of user perceptions on the following:

- To what extent do onsets found using librosa's onset detection implementation match/mirror the prominent perceivable rhythms of the song/music analysed?
- To what extent (if any) is quantizing these onsets via my methods detrimental to how in-time with the music they sound?
- Do the different quantizing methods produce distinct difficulties?

To gain some indication of the answers to these questions, I took four songs and detected the onsets for each with librosa. Applying my quantizing functions, I was left with five arrays of timestamps representing potential note placements for each song (easy, medium, hard, expert, and unquantized). I was also interested in how this method of note placement performs in comparison to the note placement method used in 'Dance Dance Convolution' (DDC), so I used a version of their model hosted online [26] to generate 'challenge' difficulty scores for these four songs, then parsed and converted these into timestamps as well.

To facilitate user testing on the above metrics, I converted these timestamps to 'click tracks' using the clicks() function from the librosa library, and combined each with its respective song. This gives an intuitive way to check the timing, rhythm and difficulty of the placed notes as you can hear where they fall in relation to the music, though in-game these notes only serve to beckon user input and do not produce sound when played.

Nine participants were tested on thirty short audio clips taken from the songs, each with an accompanying click track from one of the six note placement methods. For each, they were asked to rate, subjectively, how in-time the clicks were (that is, if the clicks were themselves an instrument in the song, would the participant consider their performance to be on tempo), how well the clicks matched or mirrored the underlying music (that is, regardless of whether the clicks are in-time or not, to what extent does the user feel they copy the music's rhythm), and how difficult the participant feels the click track would be to input, in the context of a rhythm game. The averaged scores are given below.

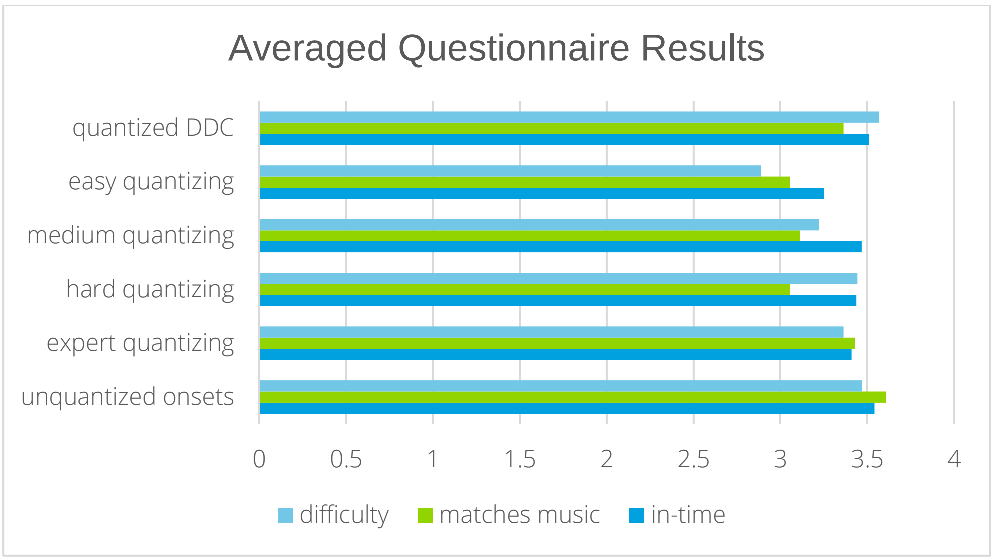


Figure 14: Averaged Questionnaire Results. Full data can be found in Appendix B.

The three criteria were measured on a scale of 1 to 5, with 5 being 'completely in-time', 'very good match' and 'very difficult', respectively.

On average, each of the six techniques performs similarly in all areas. The onsets of a song overall prove to be perceived as most in-time with the music and the best match for it, however the quantizing process does not show a significant reduction in the first metric until we reach 'easy' difficulty, showing that quantization is not overly detrimental to how in sync the notes are with the tempo of the music.

The participants perceived less of a match between the notes and the songs' rhythms at 'hard', 'medium' and 'easy' difficulties, however this is to be expected; with fewer possible notes, it is harder to emulate intricate rhythms. There is some drop in perceived difficulty between 'hard' and 'medium', and again between 'medium' and 'easy', however this drop is not large and at this stage I was concerned that the difference between the four difficulties would be negligible. A solution to this is discussed in the next chapter.

The average perception of DDC's note placement was similar to the perception of expert-quantized and unquantized note placement using onsets. However, when comparing the performance clip-by-clip (seven different audio clips were used, total), DDC consistently outperformed expert quantized onsets in the 'matches music' and 'in-time' categories. The exception to this was one clip that only featured a drum machine. This is easy to detect onsets for, and lends itself well to quantizing, so the onset-based methods performed well while DDC performed badly, likely due to the limitations of the data it was trained on.

DDC was across the board better at placing notes in highly polyphonic clips, which the spectral flux onset detection method struggles with due the high energy levels in many frequency bins making it difficult to isolate distinct changes. These differences in performance reaffirm the need for an extensible note placement system that can provide multiple options for generation.

4.2 Application Design

Participants were also asked to evaluate the wireframe design by talking though how they would approach the generation and editing of a level. This information was used to help evaluate the design before implementation.

Many participants expressed that the 'GENERATE' button looked like a title, or suggested that it should be pressed first to enable the setup options. This button is moved to the bottom of the setup box to avoid this confusion.

It was also suggested that selecting the number of note streams has more relevance to input binding than it does to generation, since it is not required for note placement. The note stream option is therefore moved to coincide with the binding step in the final design.

Participants did not expect to have to type the name of a song file verbatim to load it into the game, instead expected that either suggestions would come up as they typed or clicking on the box would allow them to browse their local files. To reflect this, the input box of audio is replaced by a dropdown box that offers options from a dedicated 'songs' folder which users must copy their audio files into.

In the game view, some participants pointed out that there was no on-screen way to pause or quit to home, and no input to do either of these things had been communicated to them. Two participants offered that they would press ESC to attempt to pause, restart or otherwise interrupt gameplay. Others indicated that they would click 'EDIT' or 'SAVE' to try and achieve this. In an attempt to remove this ambiguity, the design was edited to explicitly include a clickable pause button in the game view, and 'pause' was added as a binding alongside the note stream bindings and is preset to 'ESC'. 'Resume', 'Restart', and 'Return To Home' were added as options in this pause menu, and 'Edit' and 'Save' were removed from the unpaused screen and added to the menu as well.

Participants generally also expressed that they would expect feedback on how well they played each individual note, via popups or a clear change in the score. I worried that a single score would be too ambiguous to provide this information properly and that popups might distract a user during play, so the score was changed to multiple clear counts of how many notes the user had played at different accuracies, which increase as the user plays.

For the editor view, some participants expressed that they thought there should be some explanation of what the vertical lines and their respective colours represented specifically, beyond just where notes can go. Most also said that they would expect to be able to move, add and delete notes, but that explanation for how to do these things was also lacking. Text to explain all these features is added in the implementation.

Finally, participants all either said that on opening the editor they would expect it to show the start of the score or whatever part of the score reflected what was previously on-screen in game view. I think the latter would be more intuitive and useful from a user perspective, and so this is implemented as well.

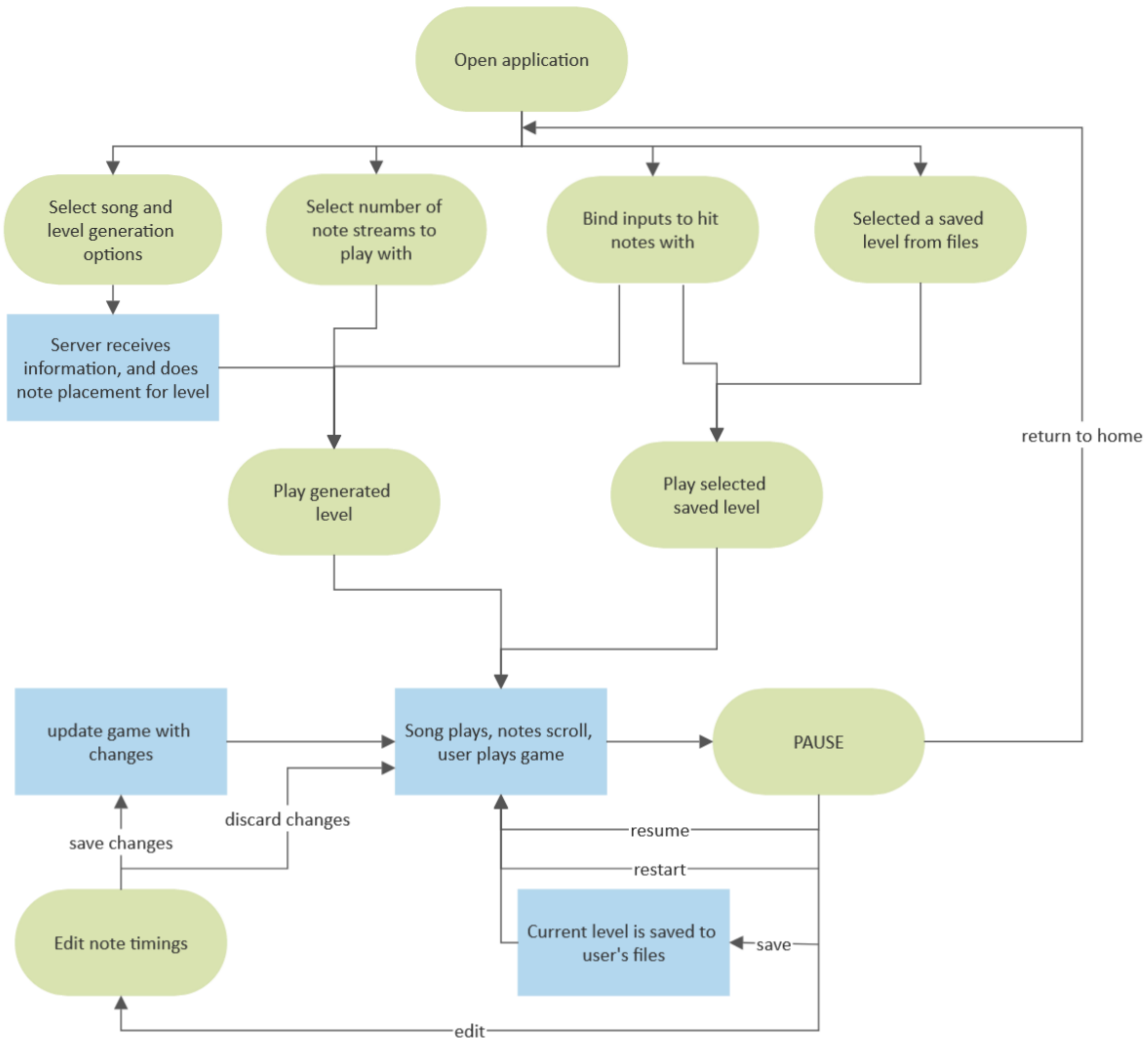

Open application
Select song and level generation options
Select number of note streams to play with
Bind inputs to hit notes with
Selected a saved level from files
Server receives information, and does note placement for level
return to home
Play generated level
Play selected saved level
update game with changes
Song plays, notes scroll, user plays game
PAUSE
discard changes
save changes
resume
restart
Edit note timings
Current level is saved to user's files
save
edit

Figure 15: Flowchart showing updated application interaction.

5 Further Design and Implementation

5.1 Onset Filtering

I believe that the main reason that there is little distinction between the difficulties created using quantization is due to the sheer number of onsets a typical song will contain. In general, a song will contain many instruments playing simultaneously, and an onset detector will attempt to label every note played by every one of them. For example, take the first 20 seconds of ‘Boys Don’t Cry’ by The Cure:

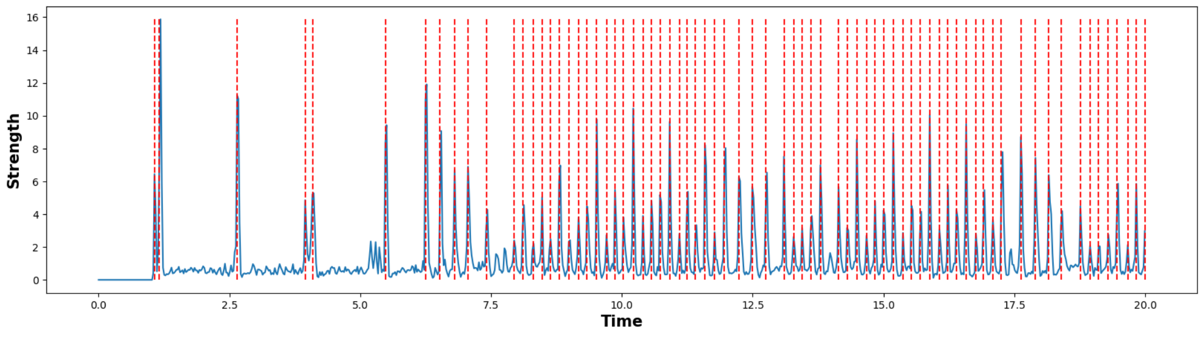


Figure 16: Example onset envelope and related onsets.

Here we have plotted the spectral flux onset strength envelope in blue and shown where the detected onsets lie with red dotted lines. In this 20 second clip, 74 onsets were found. ‘Expert’ quantizing reduces this to 63, ‘hard’ to 58, ‘medium’ to 55 and ‘easy’ to 31. Onsets are occurring very regularly in the latter two thirds of the clip, and while these onsets have different strengths and therefore prominence, they are all treated equally.

High energy onsets (which are generally the most discernible to a listener as well as to the onset detector) are not prioritised and the system simply seeks to quantize as many onsets as possible. ‘Expert’, ‘hard’ and ‘medium’ therefore have little difference between them, and if these options were the only ways to edit difficulty then the user would not have much control over how hard a level is.

To improve upon this, I developed functions to edit the array of onsets based on given parameters to filter out those with low energy, giving the user a way to reduce difficulty on a continuous scale, from retaining all the onsets to retaining none, while prioritizing the highest energy onsets.

An onset envelope is calculated from an audio signal, ‘y’, and its sampling rate, ‘sr’. Onset detection is performed on this envelope as normal, giving an array of indexes corresponding to timeframes in the audio.

```
def createFilteredOnsets(y, sr, limit, window_width_in_frames):
    onset_env = librosa.onset.onset_strength(y=y, sr=sr)
    onset_frames = librosa.onset.onset_detect(onset_envelope=onset_env, sr=sr)
```

However, not each onset frame coincides directly with the frame in the envelope containing the peak strength of the onset. You may be able to tell from the diagram above (or from Figure 17 on page 27) that onsets are often placed slightly before the zenith of a peak in the envelope. As such, to find the strength of each onset we need to find the nearby frame with the greatest strength.

```
def getLocalPeaks(onset_env, onset_frames):
    peak_frames = []
    length = len(onset_env)
    for onset_frame in onset_frames:
        strongest = -1
        for frame in range(onset_frame-1, onset_frame+2):
            if(frame >= 0 and frame < length and strongest < onset_env[frame]):
                frame_to_add = frame
                strongest = onset_env[frame]
        peak_frames.append(frame_to_add)
```

I have chosen to check one frame either side of the onset frame for the peak, as I have yet to find an onset that was placed further than one frame away from its peak strength frame, however this range can be easily changed.

The danger with increasing this range is that if multiple energy peaks lie within it, the related onsets will be assigned the strength of the tallest peak rather than the closest. Keeping the range as small as possible means this is unlikely to happen, and onsets that do lie within a frame of each other are likely to be quantized to the same subdivision later regardless.

Once we have the frames containing the true strength of each onset, we can begin to filter out those with low energy. We cannot use the total average onset strength to determine this. Some sections of a song may contain loud, clear notes that cause lots of spectral flux, while quieter and/or noisier sections produce less.

If we were to average every onset, entire sections of a song may therefore be ignored. Instead, we compare each onset only to the onsets nearby. The exact distance checked either side of each onset can be changed by the user and is given as ‘w’ below. By this method we associate a local mean and standard deviation to each onset based on nearby onset strengths.

```
def getRollingMeanAndStd(onset_env, peak_frames, w):
    frames = len(onset_env)
    p_frames = len(peak_frames)
    rolling_mean = []
    rolling_std = []

    startindex = 0
    endindex = 0
    for peak_frame in peak_frames:
        while(peak_frames[startindex] < peak_frame - w):
            startindex += 1
        while(endindex < p_frames-1 and peak_frames[endindex+1] < peak_frame + w):
            endindex += 1

        rolling_mean.append(np.mean(onset_env[peak_frames[startindex:endindex+1]]))
        rolling_std.append(np.std(onset_env[peak_frames[startindex:endindex+1]]))

    return rolling_mean, rolling_std
```

Which onsets are kept is based on a threshold which can also be edited by the user. This threshold should not be measured in fixed units of strength, as the differences between nearby onset strengths can also vary in different parts of the song, so I measure it in standard deviations. A threshold of -1.5, for example, will keep all onsets with a strength larger than the local mean minus 1.5 local standard deviations, while a threshold of 2 would only keep onsets at least 2 standard deviations stronger than the local mean strength.

```
For i in range(len(onset_frames)):
        if(onset_env[peak_frames[i]] >= local_means[i] + local_stds[i]*threshold):
            filtered_onsets.append(onset_frames[i])
```

Below are four examples of this filtering method being used with different 'window' and 'threshold' parameters on the same five seconds of audio.

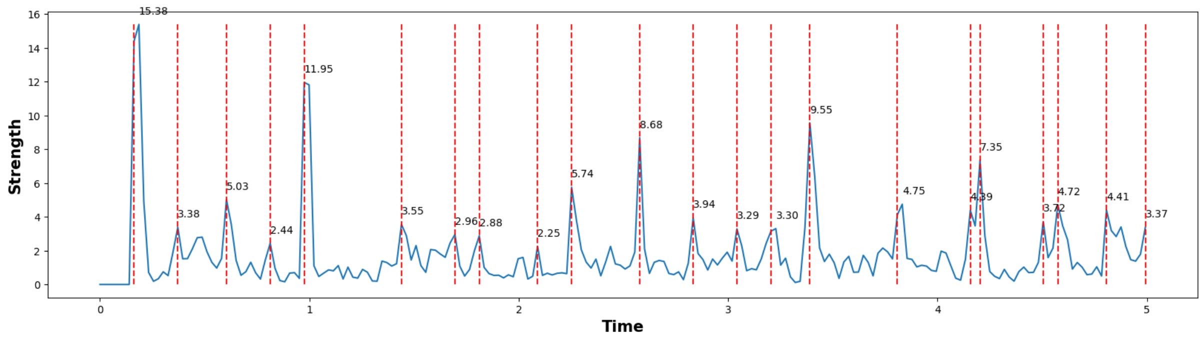


Figure 17: Filtered onsets using window = 1, threshold = -100.
All 22 onsets were kept.

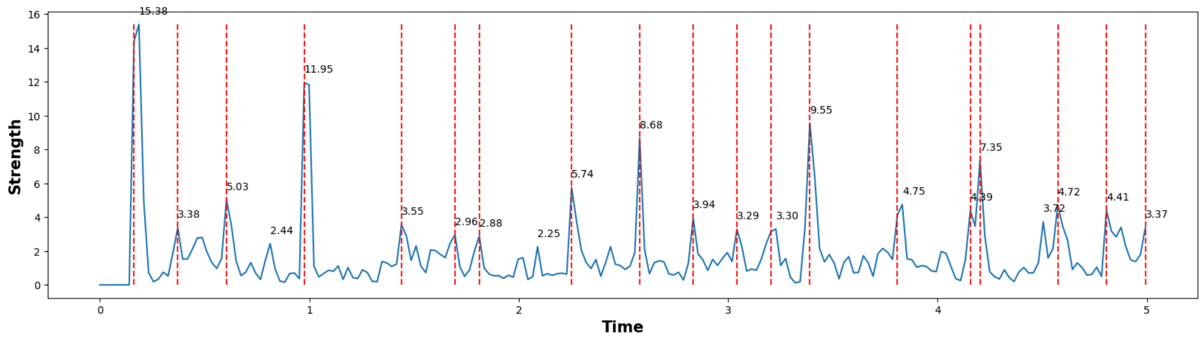


Figure 18: Filtered onsets using window = 8, threshold = -1.
19 of 22 onsets were kept.

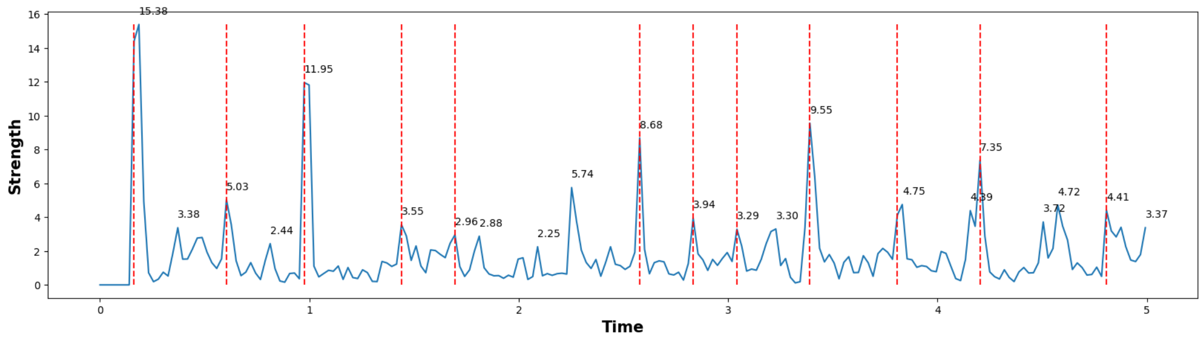


Figure 19: Filtered onsets using window = 8, threshold = 1.
12 of 22 onsets were kept.

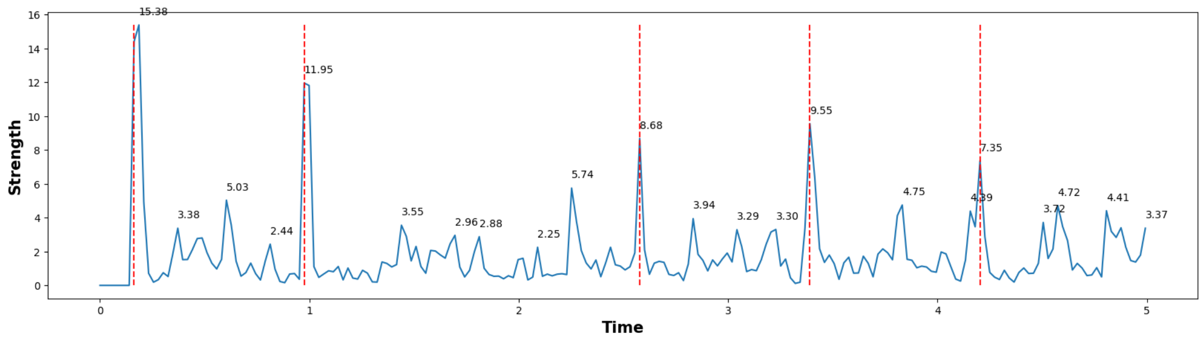
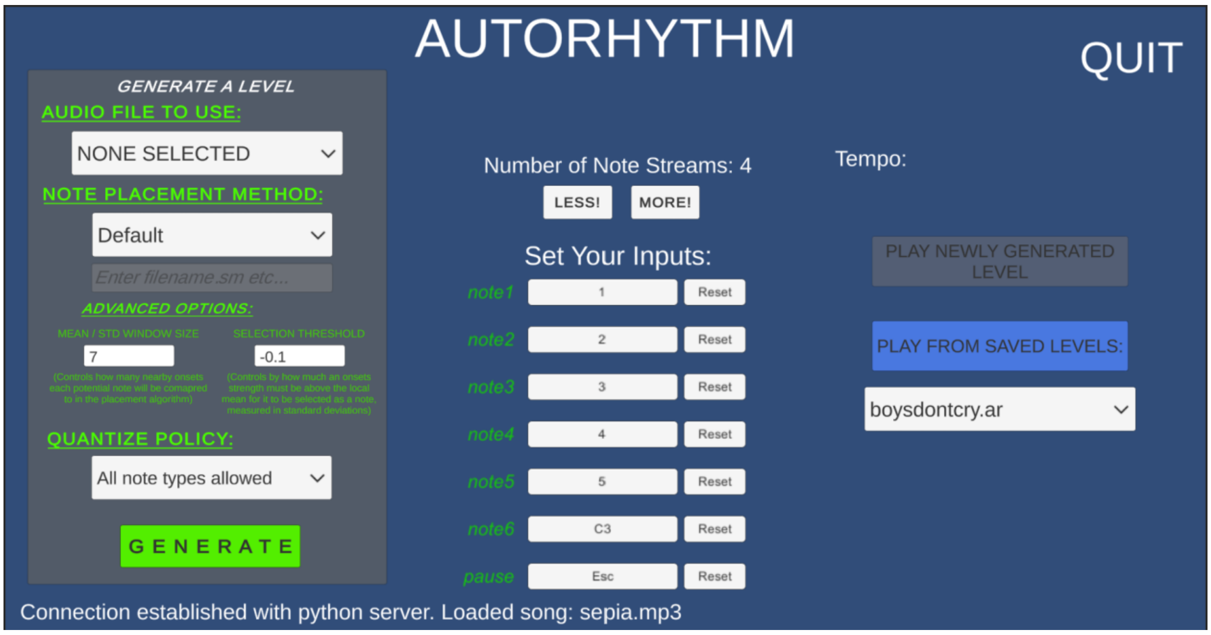


Figure 20: Filtered onsets using window = 20, threshold = 1.
5 of 22 onsets were kept.

5.2 Homepage


AUTORHYTHM

QUIT

GENERATE A LEVEL

AUDIO FILE TO USE: NONE SELECTED

NOTE PLACEMENT METHOD: Default

Enter filename.sm etc...

ADVANCED OPTIONS:

MEAN / STD WINDOW SIZE: 7

(Controls how many nearby onsets each potential note will be comapred to in the placement algorithm)

SELECTION THRESHOLD: -0.1

(Controls by how much an onsets strength must be above the local mean for it to be selected as a note, measured in standard deviations)

QUANTIZE POLICY: All note types allowed

Number of Note Streams: 4

Set Your Inputs:

note1	1	Reset
note2	2	Reset
note3	3	Reset
note4	4	Reset
note5	5	Reset
note6	C3	Reset
pause	Esc	Reset

Tempo:

PLAY FROM SAVED LEVELS: boysdontcry.ar

Connection established with python server. Loaded song: sepia.mp3

Figure 21: Completed homepage.

The homepage of 'Autorhythm' is where the majority of game setup takes place. It was created using Unity's UI elements and two Unity scripts, one to handle audio and the other to handle the rest of game setup.

5.2.1 Generation

The first step to generation is for the user to select a song that they have placed in the autorhythm_songs folder, which should exist in the same directory as the application and python script for the server. On homepage load, the 'AUDIO FILE TO USE' dropdown menu is populated with filenames from this folder found using C#'s Directory.GetFiles method.

```
private void LoadSongs()
    {
        List<string> audioFiles = new List<string>();
        string[] fileNames = Directory.GetFiles(pathToSongFolder, "*");
        string name;

        // add default first
        audioFiles.Add("NONE SELECTED");

        foreach (string file in fileNames)
        {
            // remove path to file
            name = file.Split("autorhythm_songs")[1];
            // chop off file separator
            audioFiles.Add(name.Substring(1));
        }

        songDropdown.AddOptions(audioFiles);

    }
```

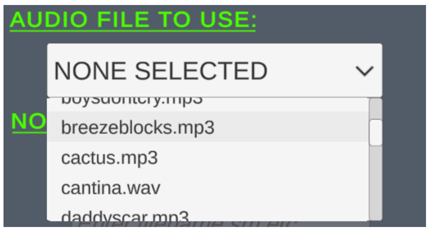
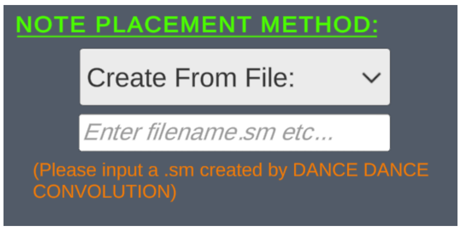


Figure 22: 'Audio File To Use' dropdown menu.

Unity allows for automatic function calls when a dropdown (or any input box) is updated. When a dropdown option is selected, a function passes the filename to the audio script which loads the song and begins to play it in the background.

The second step is to select a note placement method. Four methods are currently implemented: default, harmonic and percussive (which use onsets), as well as 'from file'. The index of the selected generation option is sent to server along with the rest of the generation options when 'GENERATE' is clicked. 'From file' additionally needs to have a file specified (which should be contained in the autorhythm_scores folder), so when selected the input box becomes interactable.


NOTE PLACEMENT METHOD: Create From File:

Enter filename.sm etc...

(Please input a .sm created by DANCE DANCE CONVOLUTION)

Figure 23: 'Create From File' display.

```
scoreFileInput.interactable = true;
scoreInputWarning.gameObject.SetActive(true);
advancedOptions.SetActive(false);
```

Additionally, the advanced options are hidden when this placement method is selected as they only pertain to note placements that use onsets.

The only files that can currently be used for note placement are .sm files generated by Dance Dance Convolution [4], as in this prototype I wanted to have an early example of the compatibility of note placement methods that use neural networks, without actually implementing one.

```
    times = []
    for x in file:
        x = x.strip()
        # 0000 indicates no step
        if(x == "0000"):
            timer += 0.01
        # if not ‘,’ or ‘0000’, then step has been placed
        elif(x != ","):
            times.append(timer)
            timer += 0.01
```

Scores created by DDC are relatively easy to parse since they have all have the same bpm, no offset and constant resolution, so we can continuously check whether a note has been placed (represented by 1s) and add 10ms to the timer before moving to the next timestep. The ‘from file’ method could be extended to support more filetypes in future, though, such as Clone Hero’s .chart files and all .sm files for ‘StepMania’.

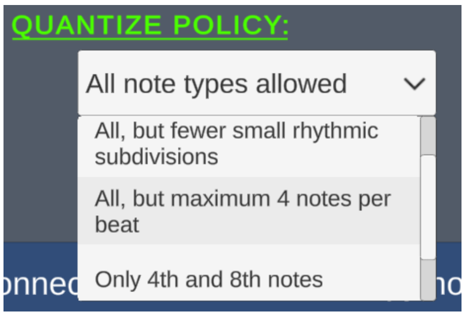
The third step in generation is to select ‘advanced options’, which are the window size and threshold for filtering onsets that were explained in the last section. By default, these are set to 7 and -0.1 respectively, as I have observed that these settings produce reasonably difficult levels but help prevent clusters of notes caused by frequent onsets that are unintuitive and difficult to play. Restrictions are placed on what can be entered here – window size is measured in frames, and so must be a positive integer, while the threshold can be set to any decimal. If at any time the contents of the input boxes are not valid, they revert to the last value they held.

```
public void WindowSizeUpdate(string str)
    {
        if (int.TryParse(str, out int result) && result > 0)
        {
            windowSize.text = str;
            windowSizeVal = result;
        }
        else
        {
            windowSize.text = windowSizeVal.ToString();
        }

    }
```

```
public void SelectionThresholdUpdate(string str)
    {
        if (double.TryParse(str, out double result))
        {
            threshold.text = str;
            thresholdVal = result;
        }
        else
        {
            threshold.text = thresholdVal.ToString();
        }
    }
```

The last step for generation is selecting a quantization method. These were initially referred to in terms of difficulty (‘expert’, ‘hard’, ‘medium’ and ‘easy’), however since the introduction of the advanced options for controlling difficulty I instead opt to describe what the quantizing methods result in. This is so they are not misled into thinking they are the only way to control difficulty, and so that they can understand what each method actually does and make an informed decision on which to use. Additionally, for some note placement methods quantizing will be a distinct task


QUANTIZE POLICY:
All note types allowed
All, but fewer small rhythmic subdivisions
All, but maximum 4 notes per beat
Only 4th and 8th notes
Figure 24: ‘Quantize Policy’ dropdown.

from controlling difficulty. If separate placement methods trained to do hard, medium and easy levels were introduced in the future, for example, it would not make sense to then have the quantizing methods named after difficulties – quantizing an already easy level with the 'hard' quantizing algorithm would not then make the level hard.

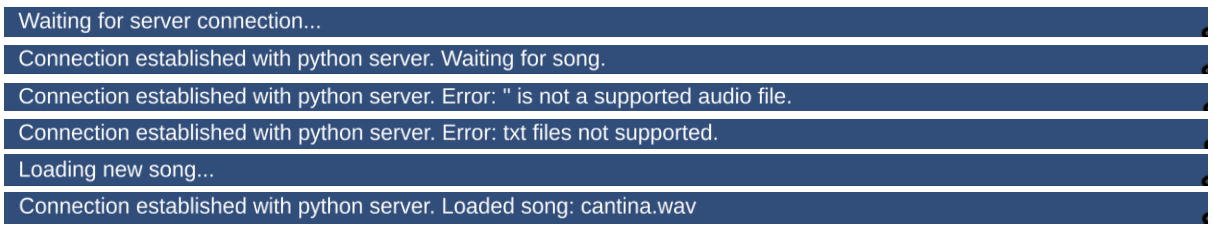
When the user presses the 'GENERATE' button, all these selected options are sent to the Python server, provided there is one running. For this communication I use 'Two-way communication between Python 3 and Unity (C#)' by Y. T. Elashry [27], which implements a way for a Python server and Unity application to send each other strings over a UDP socket. Unity sends the server the name of the song file, the quantizing and note placement methods (or rather integers representing them), the name of the score file for placing from file (if this is selected), as well as the window width and threshold from the advanced options, all separated with commas as the delimiter.

On new data being received the server splits and parses it to the correct datatypes, then runs checks to make sure everything is usable. Most checks would only fail due to implementation error in the Unity application, but two represent user error and are caused by attempted use of unsupported files. Check failures return a related error message.

```
# ensure the file specified from autorhythm_songs is audio
    try:
        soundfile.SoundFile(songpath)
    except Exception:
        return "Error: '" + song + "' is not a supported audio file."
# ensure specified file for note placement is a supported type
    if(placement_method == 3):
        if(not os.path.exists(scorepath)):
            return "Error: no file at '" + scorepath + "'?"

        file_ext = scorepath.strip().split(".")[1]
        if(file_ext != "sm"):
            return f"Error: {file_ext} files not supported."
```

Leaving these unchecked could cause the server to crash, and additionally the user needs to be informed if something goes wrong so they can take steps to correct it. All user interaction takes place within the Unity application, so all updates from the server need to be displayed there. The server sends a status message every second for Unity to display to the user. By default, this reads "Waiting for song...", but should the data check function return an error message this can set as the string to send. These messages display at the bottom of the homepage.



```
Waiting for server connection...
Connection established with python server. Waiting for song.
Connection established with python server. Error: '' is not a supported audio file.
Connection established with python server. Error: txt files not supported.
Loading new song...
Connection established with python server. Loaded song: cantina.wav
```

Figure 25: Server status messages as they appear in-game.

If no error is thrown, the server extracts the tempo and beat times of the song, and performs the relevant note placement method and quantizes these timings. Each quantize method takes the raw note timing as the beat timings as input and returns the quantized note timings and phase information, so the relevant quantization function can be passed as argument to a higher-order note placing function.

```
def placeNotes(path, mode, snap_function, window_width, threshold):
    sf = soundfile.SoundFile(path)
    y, sr = librosa.load(sf)
    tempo, beat_frames = librosa.beat.beat_track(y=y, sr=sr)
    beat_times = librosa.frames_to_time(beat_frames)

    if(mode == 1):
        y = librosa.effects.harmonic(y)
    if(mode == 2):
        y = librosa.effects.percussive(y)

    onset_frames = createFilteredOnsets(y, sr, threshold, window_width)
    onset_times = librosa.frames_to_time(onset_frames, sr=sr)
    note_times, phases = snap_function(onset_times, beat_times)
    return tempo, beat_times, note_times, phases
```

Regardless of methods, the result is a double representing tempo, arrays of doubles containing beat times and note times, and an array of integers providing phase information.

The tempo returned is an estimate of the global tempo of a given audio file, and I use this to set the movement speed of all notes in-game. The tempo of a song often fluctuates throughout and so will not always match this global tempo, however the beat times generated by librosa's beat_track function are not fixed to a constant width apart and will become less or more frequent to reflect these changes. A more accurate way so set the speed of notes would be to calculate the tempo at the start of every beat based on the distance to the next beat, however this leads either to more computation at runtime or requires more data to be stored so using the average tempo is deemed sufficient for this prototype.

The note times are the only component necessary for the basic game to function, as they define when exactly each note should be played in relation to the music. The phase information is used when displaying the notes of a score in the level editor.

All this information is sent back to Unity as a string.

```
sock.SendData("INFO;" + str(tempo) + ";" + str(beat_times)[1:-1] + ";" +
str(np.array(note_times))[1:-1] + ";" + str(np.array(phases))[1:-1])
```

This message is preceded with "INFO", which tells the Unity application that the received data is for parsing into game data rather than displaying as a status message.

5.2.2 Parsing data from the server

The arrays passed to Unity are NumPy arrays represented as strings. The string formatting for these arrays separate entries with a variable number of spaces so that they appear in human-readable columns. I need to be able to parse these arrays into C# arrays of either doubles or integers, depending on the datatype. I achieve this by implementing a generic method that accepts a delegate for parsing as an argument.

```
delegate bool TryParse<T>(string s, out T result);
private T[] ParseNumpyArray<T>(string npArray, TryParse<T> tryParse)
{
    // we need to remove empty entries because beat times can be seperated by more than one space due to python's
    formatting
    string[] stringList = npArray.Split(' ', StringSplitOptions.RemoveEmptyEntries);
    T[] parsedList = new T[stringList.Length];
    for (int i = 0, n = stringList.Length; i < n; i++)
    {
        if (tryParse(stringList[i], out T val))
        {
            parsedList[i] = val;
        }
        else
        {
            Debug.Log("Could not parse '" + stringList[i] + "'");
        }
    }

    return parsedList;

}
```

The string array is first split on spaces. With this method, consecutive spaces in the array lead to empty entries in the resulting array of strings, so these are ignored by using 'StringSplitOptions.RemoveEmptyEntries'. Each entry can then be parsed into the given type using the given parsing method. The return type and parameters for the delegate have been defined in such a way that the 'TryParse' methods built into primitive datatypes in C# can be used:

```
double[] beatTimes = ParseNumpyArray<double>(splitStrs[2], double.TryParse);
double[] noteTimes = ParseNumpyArray<double>(splitStrs[3], double.TryParse);
int[] notePhases = ParseNumpyArray<int>(splitStrs[4], int.TryParse);
```

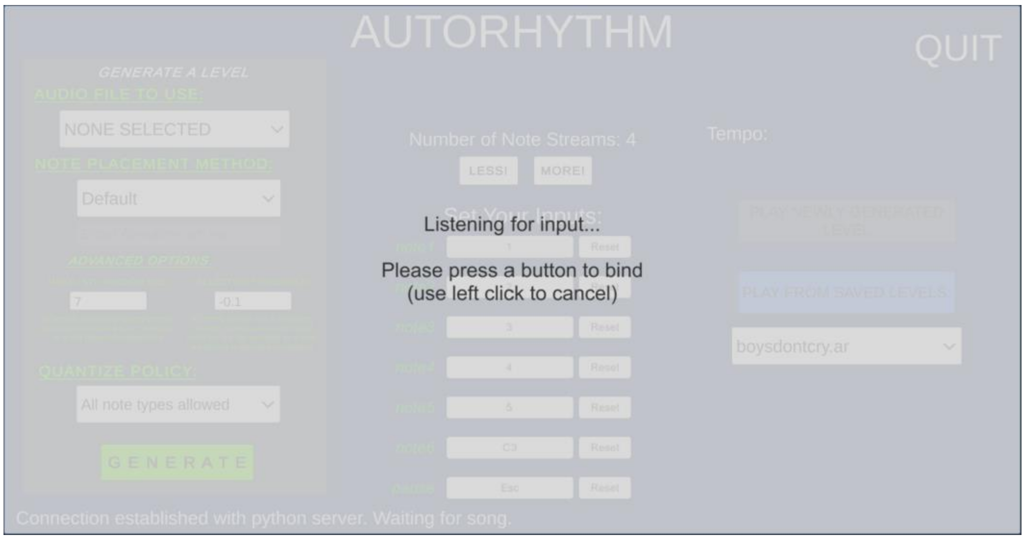
With the data loaded, the 'PLAY NEWLY GENERATED LEVEL' button becomes interactable.

5.2.3 Note Streams and Input Binding

A user can play with one to six note streams, and this can be incremented or decremented with two buttons on the homepage. This is only relevant when playing a newly generated level, as pre-saved levels have the number of note streams used and which stream each note belongs to specified. This prototype of 'Autorhythm' has no method of note selection implemented, and so every note is randomly assigned a stream when 'PLAY NEWLY GENERATED LEVEL' is clicked.

Relevant for both generated and saved levels, though, is which input each stream is bound to. For input binding I use Unity's new input system [22], which allows me to set up a 'control scheme' for in-game actions and dynamically change which inputs correspond to which action at runtime. I gave the control scheme actions for playing notes in each of the six possible streams, as well as one for pausing, all of which can be rebound using buttons from Unity's rebinding UI package. On click, the application waits for the user to press an input and binds it to the relevant action. The user can bind any supported input devices [28], which includes all USB HID class devices, as well as MIDI inputs thanks to the 'minis' plugin [24].

The only limitation is that the user cannot bind the left mouse button to any input, and clicking it instead cancels the rebind operation. It is useful to give the user a way to cancel in case they press a rebind button by accident or change their mind, and left click was chosen as it is used for all in-game navigation on the homepage and in the pause menu later. This would mainly pose an issue only if left click was bound to the 'pause' control, as attempting to click a button in the pause menu would cause the game to unpause rather than selecting the desired option, however the user is prevented from using left click for any game inputs to leave a clear distinction between navigating the application and playing it.



Listening for input...

Please press a button to bind (use left click to cancel)

Connection established with python server. Waiting for song.

Figure 26: Homepage with input binding overlay active.

By default, the six note streams are set to number keys 1 through 6 on the keyboard, and pause is bound to the escape key.

5.2.4 Playing from saved levels

Should the user want to play a saved 'Autorhythm' level instead of a newly generated one, they can do so by selecting an 'AR' file (discussed later) from the dropdown on the right and clicking 'PLAY FROM SAVED LEVELS'. The name of the required audio file from the songs folder and all the data needed populate the beatTimes, noteTimes and notePhases arrays that allow the game to function are contained within the AR file, so playing from saved does not require a connection to the external server.

The dropdown is populated in the same way as the dropdown for picking a song in the generation options, however now only files with the '.ar' extension are displayed. This filtering is necessary because the autorhythm_scores folder is not intended to only contain save files, but also foreign score files for the 'load from file' note placement option.

```
string[] scoreNames = Directory.GetFiles(pathToScoreFolder,"*.ar");
```

When 'PLAY FROM SAVED LEVELS' is clicked, the given AR file is parsed for the required data. This consists of:

- The name of a song file, which is then loaded to the audio driver from the songs folder.
- The average tempo of a level, which is used to determine note speed.
- The number of note streams the level uses.
- The timings of the beats of the given song, which are used to optionally provide beat guidelines later.
- The timings of each note in the score.
- The stream each note belongs to.
- The phase of each note in the score.

Should the song file specified not exist in the songs folder, the level will begin as normal but no music will play in the background.

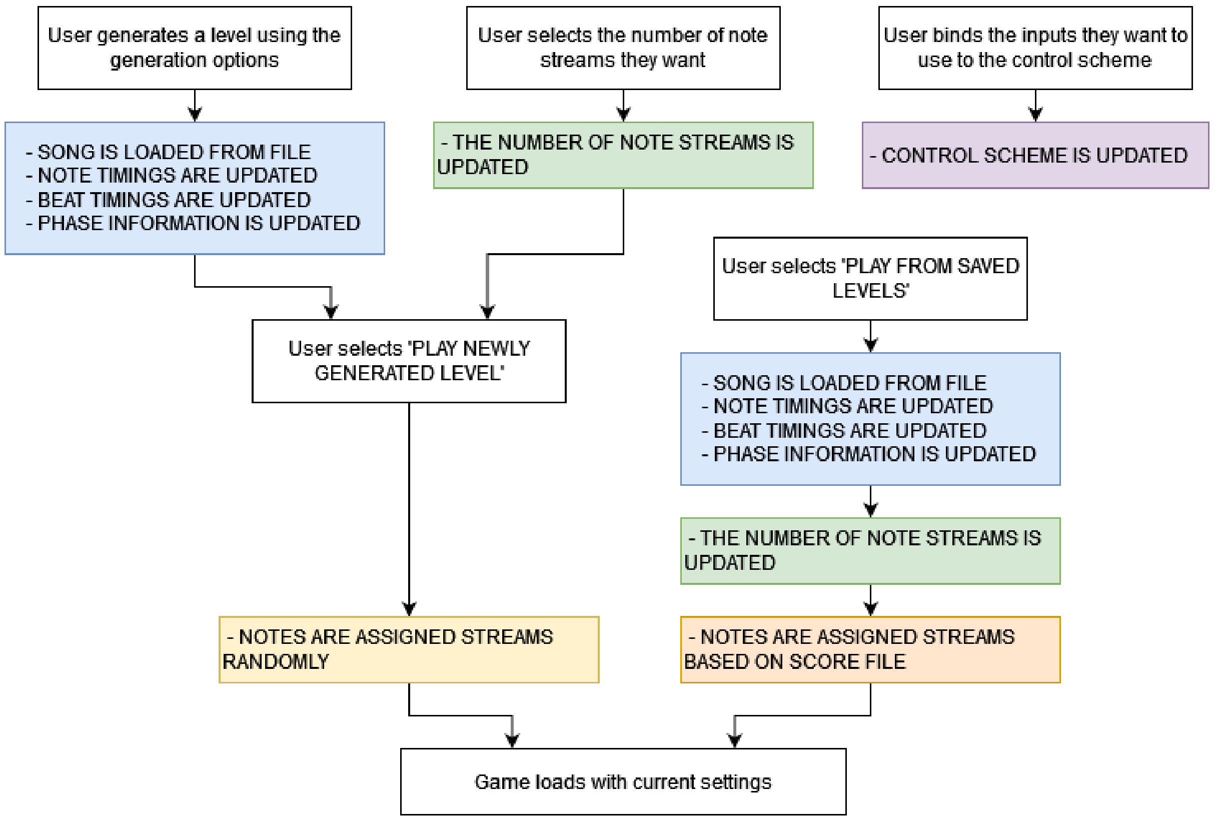

User generates a level using the generation options
User selects the number of note streams they want
User binds the inputs they want to use to the control scheme
- SONG IS LOADED FROM FILE
- NOTE TIMINGS ARE UPDATED
- BEAT TIMINGS ARE UPDATED
- PHASE INFORMATION IS UPDATED
- THE NUMBER OF NOTE STREAMS IS UPDATED
- CONTROL SCHEME IS UPDATED
User selects 'PLAY FROM SAVED LEVELS'
User selects 'PLAY NEWLY GENERATED LEVEL'
- SONG IS LOADED FROM FILE
- NOTE TIMINGS ARE UPDATED
- BEAT TIMINGS ARE UPDATED
- PHASE INFORMATION IS UPDATED
- THE NUMBER OF NOTE STREAMS IS UPDATED
- NOTES ARE ASSIGNED STREAMS RANDOMLY
- NOTES ARE ASSIGNED STREAMS BASED ON SCORE FILE
Game loads with current settings

Figure 27: Flowchart describing how interaction with the homepage can set the data required to play a level.

5.3 Playing a Level

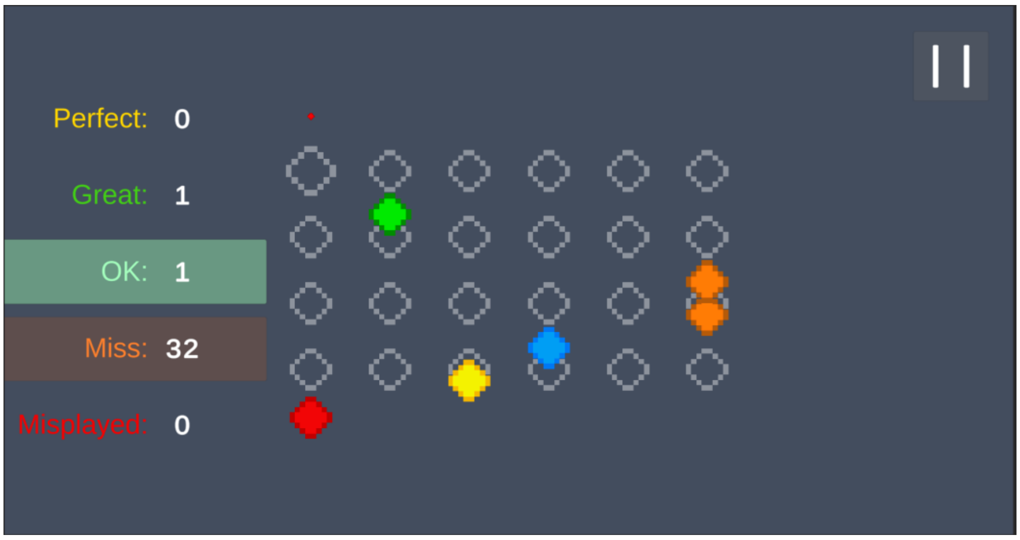


Figure 27: Gameplay still, featuring six note streams.

Once either 'PLAY NEWLY GENERATED' or 'PLAY FROM SAVED' is selected, a new Unity scene is loaded where the user can play the level. Notes begin to spawn at the bottom of the screen, and move upwards over the span of 4 beats, reaching the top row of note outlines at the time they are meant to be played. Notes that pass this line begin to shrink, and disappear after one beat unless played late. As the user plays the accuracy of their performance is tracked via scores on the left of the screen. Gameplay is controlled mainly by two scripts, a 'spawn' script to handle note creation and a 'logic' script to handle all other important functions.

5.3.1 Note Behaviour

Notes are created from 'prefabs' – a reusable asset that can be easily instantiated from a script. Each note has a script attached that handles its behaviour. On instantiation, this script calls methods from the 'logic' script to calculate from the current tempo how quickly the note should move and how long it will take to reach the top row.

Notes spawn 'startDist' units away from the top row and must take exactly 4 beats (based on the current tempo, measured in beats per minute) to reach it.

Seconds to reach = 4*(60/tempo), units per second = startDist/(time to reach) = startDist*tempo/240

```
public float CalculateMoveSpeed()
{
    return (startDist * tempo) / 240;
}

public float CalculateNoteLifespan()
{
    return 240 / tempo;

}
```

Once calculated, notes need to use this information to move at a constant rate and arrive at the top row at the correct timing. Unity provides all classes that inherit from the 'MonoBehaviour' class with an Update() method, which is called every frame. However since framerate is variable depending on a user's computer, we cannot move notes by a fixed amount each frame and must take into account how much time has passed since the last. Unity provides this in seconds 'deltaTime' in its Time class, so we can incorporate this when calculating how far a note must move each frame.

```
transform.position = transform.position + (Vector3.up * moveSpeed) * Time.deltaTime;
```

Time.deltaTime is also used by a note's script to keep track of how long the note has existed for. Once this value exceeds the calculated lifespan of the note (indicating the note has passed the top row), a coroutine is triggered that begins to shrink it.

```
timer += Time.deltaTime;
        if (timer > liveFor && !startedDeleting)
        {
            StartCoroutine(ShrinkDelete());
            startedDeleting = true;

        }
```

In Unity, a coroutine is a function that can execute over multiple frames, which is useful for achieving time-based tasks such as this one. 'Yield return null' is used to indicate where the function should pause, and execution will continue from this line in the next frame.

```
protected IEnumerator ShrinkDelete()
    {
        Vector3 initialSize = transform.localScale;
        Vector3 endSize = new Vector3(0, 0, 0);


        float timer = 0, duration = liveFor/4;
        while(timer < duration)
        {
            transform.localScale = Vector3.Lerp(initialSize, endSize, timer / duration);

            if(logic.paused == false) timer += Time.deltaTime;
            yield return null;
        }

        logic.BumpMiss();
        Destroy(gameObject);

    }
```

Every script in Unity is attached to a GameObject (which in this case is the note), and every GameObject has a 'transform' component that determines its location and scale. Scale is represented by a 3D vector (although since this game is 2D, the last value does not matter), and we can use linear interpolation to gradually reduce the scale from its initial size to nothing. Vector3.Lerp(a, b, t) accepts two 3D vectors and values of t from 0 to 1, and will return a*(1-t) + b*t. By setting t to timer / duration, the returned vector starts at initalSize (as the timer starts at 0) and approaches endSize as the timer approaches the duration set. The timer is incremented by Time.deltaTime, which again serves to make the duration of this process independent of framerate.

When the timer reaches the set duration, which in the code above is the duration of one beat, the loop finishes and the note is deleted. The BumpMiss() function is also called to update the users score, reflecting that the note went unplayed.

5.3.2 Note Creation

As mentioned, the creation of note is handled by a dedicated 'spawn' script. This script has its own internal timer to track how much of the background song has elapsed, and also tracks the index of the next note to be spawned. Together, these are used to instantiate notes at the correct times.

```
movementTime = logic.CalculateNoteLifespan();
while (noteIndex != -1 && timer > GameSetup.noteTimes[noteIndex] + Logic.gameStartWaitFor - movementTime)
{
    SpawnNote(GameSetup.streamInfo[noteIndex]);
    noteIndex++;
    if (noteIndex == noteTimesLength) noteIndex = -1;

}
```

As we have seen, notes take 4 beats at local tempo to reach the top row, where they are meant to be played, and so must be spawned 4 beats early. However, this means that notes occurring in the first four beats of the music do not have time to reach the top row should the music start playing immediately on level load. To remedy this, a delay relative to the speed of four beats at the starting tempo of the song is implemented on both the spawn script and the audio driver, allowing these early notes to appear and begin moving in the initial silence so they can reach the correct positions at the correct timings.

5.3.3 User Interaction

For the game to function, notes must disappear when 'played' by the user. This comes with some design considerations.

Firstly, I only allow notes to be played if they are within one beat of the top row. During play, a user will generally be focused on the notes that are on or nearing the top row, as these are the ones that need to be played next. If a user accidentally presses the wrong input when trying to play a note and this restriction is not in place, it could result in a note in another stream much further down the screen being deleted without the user noticing. Take figure 28 (right). If the user's next input is to play a red note rather than a yellow note, the error is likely to be that the user pressed the wrong input rather than the user was attempting to play a red note two beats early. The user was not intending to remove a red note and can still play them in-time later. Deleting one does not correctly reflect the user's error and may make the level feel disjointed.

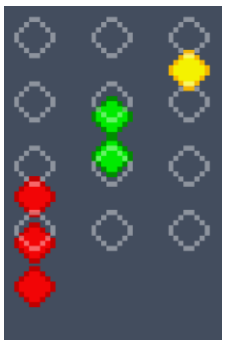


Figure 28

Secondly, if there is more than one note of the correct stream within one beat of the top row when the corresponding input is pressed, the note closest to top row is 'played', rather than the oldest. Although this may not always be the note they were intending to play, especially if notes are closely bunched together, this prevents small, shrinking notes that are about to disappear from being played in place of a newer note which is lined up well with the top row.

5.3.3.1 Implementation

The spawn script is attached to a GameObject called 'NoteSpawner'. A 'PlayerInput' component is also attached, and broadcasts messages when the user's bound inputs are pressed. Through this, six methods in the script to deal with playing notes in each of the six possible streams can be automatically called when the respective inputs are activated. Notes are named based on which stream they belong to, and so the respective name is passed by these methods to a method that handles users attempting to play notes.

```
foreach (Transform child in transform)
        {
            if (child.name.StartsWith(name))
            {
                moveScript = child.GetComponent<T>();
                currentDist = Math.Abs(moveScript.timer - moveScript.liveFor);
                if(currentDist < moveScript.liveFor/4 && currentDist < closestDist)
                {
                    closestNote = child.gameObject;
                    closestDist = currentDist;
                }
            }

        }
```

All notes are instantiated by the spawn script at runtime, and so their Transform components are children of NoteSpawer's Transform component. By iterating through these children we can find whether they belong to the correct stream based on their name. If they do, their internal timer and lifespan can be used to calculate how far from the top row they are, allowing us to determine whether they are within one beat of it and which is closest. If a note is found, it is deleted and the user's score is updated based on the note's distance.

5.3.4 Scoring

Each time the user inputs to play a note, the 'performance' is rated as either 'Perfect', 'Great', 'OK', 'Miss', or 'Misplayed' and the counters on the left of the screen tick up, allowing them to gauge their accuracy. The rating is dependent on how close a note is to the top row, as shown in figure 29. If a note passes the top row and shrinks until it disappears, this is counted as a 'Miss'. If the user tries to play a stream that has no notes near the top row in it, this is counted as a 'Misplayed' note instead (since in this instance, no note is deleted).

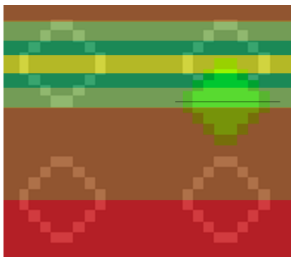


Figure 29: Scoring bands

Initially the only feedback to the user was the scores counting up as they played. This is fine if the user only wants to look at their score at the end, however since during the game the user will be constantly focused on the notes coming up the screen it is very difficult tell which score they are receiving for each note. One way to amend this is by making it easier for the user to tell which scores are increasing out of their peripheral vision, which I achieved by making each score flash its respective colour whenever its value changes. This is implemented by placing transparent panels over each score and using a coroutine to increase its opacity on score increase before quickly decaying it back to transparent over 0.2 seconds.

```
float timer = 0, duration = 0.2f, startingOpacity = 0.6f;
        while (timer < duration)
        {
            panel.color = new Color(panel.color.r, panel.color.g, panel.color.b,
                                    startingOpacity - (startingOpacity * timer / duration));
            timer += Time.deltaTime;
            yield return null;

        }
```

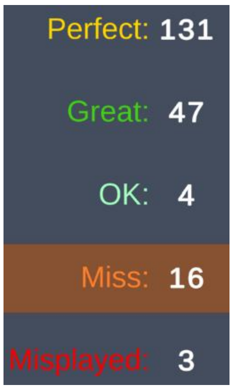


Figure 30: 'Miss' score total highlighted in response to missed note.

There was also initially no feedback on which note stream a user had pressed, other than notes disappearing from that stream if possible. New users do not necessarily know which stream is which, and any user may forget which streams are associated with which input, so providing clearer and consistent feedback is useful. This is done by having the note outlines in the top row increase in scale slightly when the input bound to their stream is pressed, and then shrinking them back to their original size with a coroutine similar to the one used for shrinking notes.

The first implementation of this had the current size of an outline increase by a factor of 1.2 then shrink over 0.2 seconds by a factor of 5/6 to return it to its original size. However, if the user triggers this method again during the 0.2 second shrinking period, the note grows from whatever size it is currently at by a factor 1.2 again and will not return to its original size. This was fixed by saving the 'true size' of a note outline on scene load and using this as a reference rather than the current size of an outline.

5.4 Pause Menu

At any time the game can be paused, either by pressing the input bound to pausing or by left clicking the pause button in the top right. The second option was included to give the user a failsafe way to pause in case they cannot do it by other means, for example if their controller disconnects or they forget which button they bound to pausing, which is necessary as the only way to return to the homepage is via the pause menu, and the only way to rebind the pause input is via the homepage.

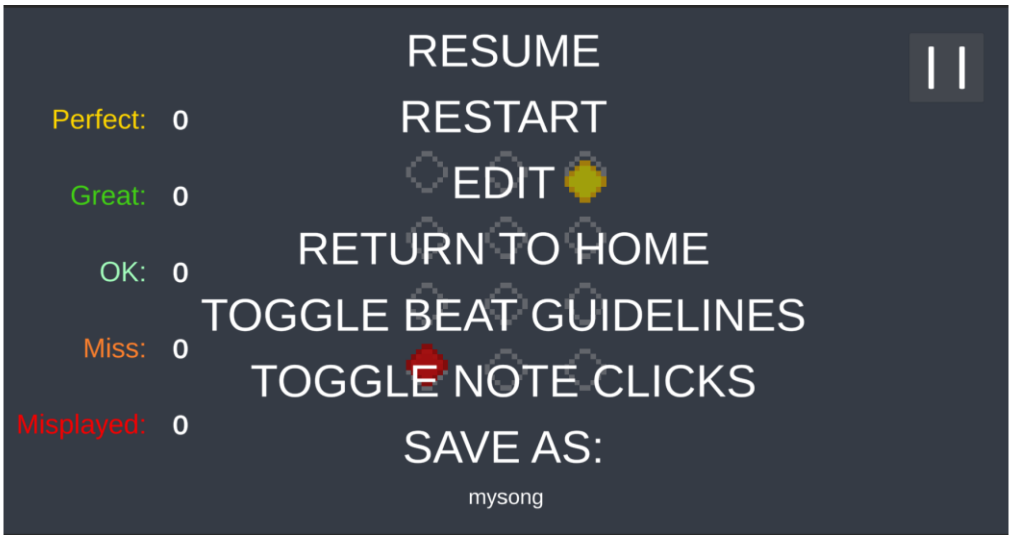


Figure 31: Pause menu overlay in game view.

5.4.1 Resume

The resume button unpauses the game and deactivates the pause menu overlay. This can also be achieved with either of the two methods used to pause the game in the first place.

5.4.2 Restart

The restart button resumes the game from the beginning. The internal timer and note index in the spawn script are reset to 0, as are the scores, and audio is restarted by a coroutine after the required 4 beat delay.

5.4.3 Return To Home

The return to home button changes the Unity scene back to the homepage. The audio driver used to play the music for the game is created when the homepage is loaded, but is not destroyed when the scene changes to the game view since the music it holds is needed. To stop multiple instances being created, 'return to home' also destroys the current audio driver.

5.4.4 Beat Guidelines

Existing rhythm games will generally give some indication of the rhythmic subdivision of each note. In 'StepMania', different subdivisions have different colours, and in some 'Guitar Hero' games lines indicating where beats lie move with the notes. The four rows of note outlines in 'Autorhythm' are useful as reference points for working out the approximate distance between notes, but they give no indication of where beats lie in the music. As such, a user can optionally toggle beat guidelines, horizontal lines that reach the top row in time with the detected beats in the music. This gives the user more frame of reference to work out the phases and timings of notes. This is useful when playing a level and useful when used in tandem with the level editor as it makes the rhythmic subdivisions that the editor relies on easier to determine in-game.

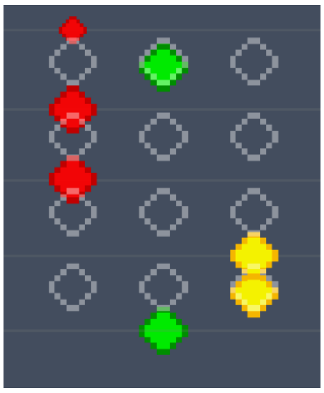


Figure 32: Beat guidelines.

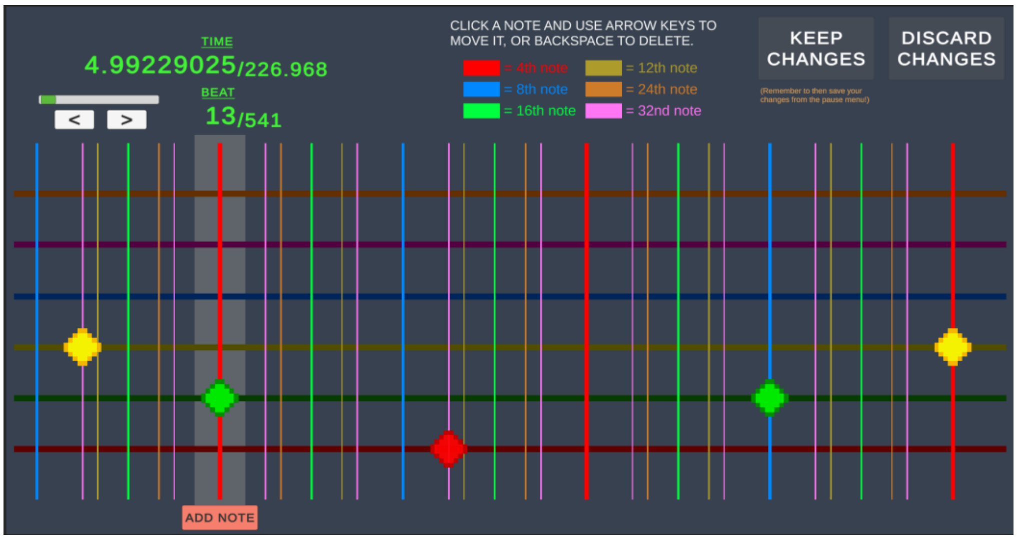
5.4.5 Note Clicks

The note clicks option was also added as a way to help with editing. Unless a user can play a level perfectly, it is hard to know whether the notes in a score are slightly out of time, or if the user simply is not playing them at the correct timings. Activating note clicks causes a click to sound whenever a note reaches the exact spot it is meant to be played. This way, a user wanting to edit a level does not have to attempt to play the notes perfectly or try to visually determine whether notes are in time or not, and can instead just listen to the clicks against the song to immediately hear where exactly the notes should be played in relation to it, and whether they are in-time or not.

When a note is played it is deleted, and each note contains its own 'AudioSource' to play these clicks, so if a note is played before it reaches the centre of the top row its click will not sound. This makes this option less useful for users who are genuinely playing the game, and so it is mainly just intended as a tool to help with editing.

5.5 Editor

5.5.1 Features



CLICK A NOTE AND USE ARROW KEYS TO MOVE IT, OR BACKSPACE TO DELETE.

🟥 = 4th note
🟦 = 8th note
🟩 = 16th note
🟨 = 12th note
🟧 = 24th note
🟪 = 32nd note

(Remember to then save your changes from the pause menu!)

Figure 33: Editor view.

The edit button brings up the editor for the level. This does not use a scene change, but instead activates a hierarchy of UI elements which cover the whole screen. Beneath this the game simply remains paused, and cannot be unpaused until the overlay is deactivated.

The top left of the editor displays the index and timing of the 4th note beat highlighted by the grey box. When the editor first opens, this index is set to the index of the last beat to cross the top row in-game. This is so that the first notes displayed to the user match the notes that were near the top row when they paused and opened the editor.

The editor displays all notes from the preceding 8th note to the 4th note two beats away, giving a total view of 2.5 beats. Every accepted rhythmic subdivision in this range is displayed by coloured vertical lines. A key at the top of the screen explains which beat phases are referenced by each colour. Coloured horizontal lines in the background represent the six streams that notes can belong to. Together these lines make a grid, showing all the possible positions notes can occupy.

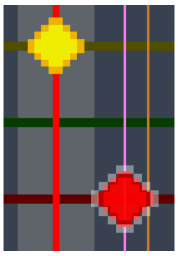


Figure 34: Example of a selected note (red) and an unselected note (yellow)

Notes displayed in the editor have a 'Toggle' component, which allows the user to click on notes to select or unselect them. A reference to the currently selected note is held in the logic script, and when a new note is selected the script will use this reference to deselect the old note and target the newly selected one, ensuring only one note can be selected at a time.

A selected note can be moved around the grid using the arrow keys, or removed from the score entirely by pressing backspace. In general notes can be moved anywhere on the displayed grid, with a couple of exceptions. Firstly, notes cannot be moved up onto note streams that do not exist in the current level. The number of note streams in a given level is fixed as the note outlines for each stream are drawn on scene load, so if the user wants to introduce more they need to return to the homepage first and either regenerate the level with more streams or edit the number of streams defined in a saved AR file. Secondly, notes cannot be moved backwards past the beginning of the first beat, since this may not be in the bounds of the audio and since I cannot quantize notes that are not in-between two beats. To stop users placing notes past the last beat in a score, the highlighted beat cannot go past (total beats – 2), so subdivisions beyond the last beat can never be displayed.

If the user wants to add a note to the score, there is a button in the bottom left. This will create a new note belonging to the first stream (red) at the highlighted beat. The red note stream will always exist regardless of the number of note streams, which is why this position was chosen.

The user can navigate the score by pressing the arrow buttons in the top left to move to the previous or next beat. Moving by just one beat produces overlap since the editor displays 2.5 beats, allowing the user to better keep track of where notes are in relation to each other. If the user wants to move the display quite far from the current beat, they can use the scrollbar above the arrow buttons to navigate the entire breadth of the score, rather than having to repeatedly click through each beat to get somewhere.

To return to the game, a user can select to either keep or discard changes. Discard changes will cause no change to the level, while keep changes will update the local 'noteTimes', 'notePhases' and 'streamInfo' arrays that define what notes are created and when in the level.

If the user discards changes, they are returned to the pause menu with no effect on the game. However, if the user keeps changes the game is forwarded or rewound so that they are returned four beats before the highlighted beat in the editor. This is so that the notes in the editor view are the first ones to appear on screen, allowing the user to immediately appraise their changes without having to remember where they were and play through the level to get to them. This then lets the user go quickly back and forth between game view and editor view to experiment and refine changes more easily.

5.5.2 Implementation

The editor needs to be able to change the timings and beat phases of notes in the score, as well as the stream they belong to. During gameplay, this information is stored in three arrays of equal length, 'noteTimes', 'notePhases' and 'streamInfo'. We cannot manipulate this data directly as the user may choose to discard changes. Additionally, the user may add or delete notes, and the standard arrays in C# do not support this. As such, I define a new class 'NoteInfo' to hold all data associated with a given note, and then create a 'List' of these objects to represent the level's score in a way that is easier to manipulate.

```
public class NoteInfo
    {
        public double time;
        public int phase;
        public int stream;

        public NoteInfo(double t, int p, int s)
        {
            time = t; phase = p; stream = s;
        }

    }
    private List<NoteInfo> score;
```

Whenever the editor is opened, this list is cleared and populated.

The notes in the score that occur in the range shown on screen must be displayed on screen. I created prefabs for 'clickable notes' to display, and instanciate them on screen in the correct positions with a method called 'DrawNotes'.

```
public void DrawNotes(int beat, int selectedNotePhase)
```

Given an integer beat, this function will search the score list for notes that lie in the correct range, calculate the relevant x and y co-ordinates they should be placed at based on their phase and stream, and instantiate them.

```
//get the 'phase number' of the 8th note before this beat, and the 4th note 2 beats on
// (because this is the width the editor can display)
int startPhase = beat * 12 -6;
int endPhase = (beat + 2) * 12;

int index = 0, x, y;
GameObject newNote;
// skip through score
while (index < score.Count && score[index].phase < startPhase) index++;

while(index < score.Count && score[index].phase <= endPhase)
{
    x = CalculateX(score[index].phase - startPhase);
    y = CalculateY(score[index].stream);
    newNote = GetPrefab(score[index].stream);
```

```
            newNote = Instantiate(newNote);
            //tell the note its overall index
            newNote.GetComponent<ClickableNoteScript>().myIndex = index;
            //add to Canvas
            newNote.transform.SetParent(canvas.transform);
            // give position local to Canvas
            newNote.transform.localPosition = new Vector3(x, y, 0);
            // give size relative to Canvas
            newNote.GetComponent<RectTransform>().sizeDelta = new Vector2(50f, 50f);
            // give the toggled-on outline a size realtive to Canvas
            newNote.transform.GetChild(0).GetComponent<RectTransform>().sizeDelta = new Vector2(60f, 60f);
            //if newNote was the selected note, re-select it
            if (score[index].phase == selectedNotePhase)
            {
                newNote.GetComponent<Toggle>().isOn = true;
            }
            shownNotes.Add(newNote);
            index++;

        }
```

References to all the created note GameObjects are added to a 'shownNotes' list, so they can be destroyed next time DrawNotes is called. If the user had selected a note prior to DrawNotes being called, and this note is still on screen, then this note can be found again and reselected by specifing its phase as 'selectedNotePhase'.

DrawNotes is called anytime the highlighted beat is changed via the arrow buttons or scrollbar, displaying the notes in the new range. It is also used whenever a note is moved, added or deleted, which is easier than implementing logic to carry out these tasks without destroying and re-creating all the notes. Instead, the relevant note can just be edited in, added to, or deleted from the score list, and this change will be immediately reflected when DrawNotes is called again.

Notes must be ordered by timing in the score list for the logic of the main game and DrawNotes to function, so any time a note is added or moved horizontally the list must be sorted to ensure this ordering is preserved.

```
        //resort and redraw
        score.Sort((a, b) => a.time.CompareTo(b.time));

        DrawNotes(currentBeat, score[i].phase);
```

If the user chooses to keep changes, the sorted score is converted to three arrays to replace the old 'noteTimes', 'notePhases' and 'streamInfo' arrays. Methods in the spawn script and audio driver are also called that change the internal timers for both to four beats before the currently shown beat in the editor.

```
    public void SaveEdits()
        {
            int returnToBeat = currentBeat - 4;
            if (returnToBeat < 0) returnToBeat = 0;
            float timeToReturnTo = (float) GameSetup.beatTimes[returnToBeat];
            GameObject.FindGameObjectWithTag("Spawner").GetComponent<SpawnScript>().SkipToTime(timeToReturnTo);
            audioScript.SkipToTime(timeToReturnTo);

            //update noteTimes, notePhases, streamInfo with any changes
            List<double> newTimes = new List<double>(score.Count);
            List<int> newPhases = new List<int>(score.Count);
            List<int> newStreams = new List<int>(score.Count);
            foreach (NoteInfo note in score)
            {
                newTimes.Add(note.time);
                newPhases.Add(note.phase);
                newStreams.Add(note.stream);
            }
            GameSetup.noteTimes = newTimes.ToArray();
            GameSetup.notePhases = newPhases.ToArray();
            GameSetup.streamInfo = newStreams.ToArray();

            LeaveEditor();
            Resume();

        }
```

5.6 Design of Save File Format

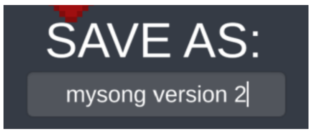


Figure 35: 'Save As' input box.

Finally, the save as button saves the current level to the autorhythm_scores folder as an AR file, so it can be played again later. The name that it will save under is shown under the 'SAVE AS' option. By default this is the same name as the audio file used to provide the background music for the level, however the user can replace it with whatever name they want. This allows for multiple levels to exist for the same audio file, since saving will overwrite files of the same name.

Deciding how to represent a score when it is saved to file comes with a number of considerations. It would be relatively easy just to save all the required data in a condensed format, however this project aims to encourage interoperability and future use of these scores as a MIR resource. To this end, I first looked at the structure of score files used in 'StepMania', 'Clone Hero' and 'Taikojiro', as all of these feature a human-readable format so that anyone can create a level, and all have been used to train neural networks, as mentioned in the 'Background and Context' chapter.

5.6.1 SM files

SM files are text files used to represent scores for 'StepMania'.

5.6.1.1 Metadata

The start of the file includes data about the score, indicated with hashtags:

```
#TITLE:MySong;
#SUBTITLE:;
#ARTIST:foo;
#TITLETRANSLIT:;
#SUBTITLETRANSLIT:;
#ARTISTTRANSLIT:;
#GENRE:Rock;
#CREDIT:bar;
#BANNER:banner.png;
#BACKGROUND:bg.png;
#LYRICSPATH:;
#CDTITLE:cd.png;
#MUSIC:MySong.ogg;
#OFFSET:-0.023;
#SAMPLESTART:35.768;
#SAMPLELENGTH:25.993;
#SELECTABLE:YES;
#DISPLAYBPM:95.000;
#BPMS:0.000=94.745
,15.000=95.100
,16.000=94.745
,31.000=95.100
,32.000=94.800
,47.750=93.000
,48.000=94.800
,63.833=92.000
,64.000=94.800
,79.750=92.000
,80.000=94.800
,95.500=93.700
,96.000=94.745
,111.000=95.100
,112.000=94.745
;
```

Figure 36: Example SM metadata.

Lots of this data is decorative and has no impact on gameplay beyond changing display, such as 'title', 'artist', 'background', 'cdtitle' and 'displaybpm'. The important information contained here is:

- 'music', which specifies an audio file to use,

- ‘offset’, which allows the score to sync with the music by defining when the first note in the score should be played in relation to the start of the audio,
- ‘samplestart’, optionally defining where to begin playback of the audio file from,
- ‘samplelength’, optionally defining how long the audio should be played for
- ‘bpms’, which defines the tempo of the song, as well as the position and value of all changes to this tempo

5.6.1.2 Score

Different difficulty scores for the level are all contained within the same sm file. The score is divided up into 4-beat measures, and each measure can have a resolution of either 4, 8, 12, 16, 24, 32, 48 or 192 possible notes per measure, independent of other measures.

```
0100
0000
0010
0000
,
1001
0000
0001
1000
0100
1000
0010
0000
0100
0000
0010
1000
0001
0010
0100
0000
,
```

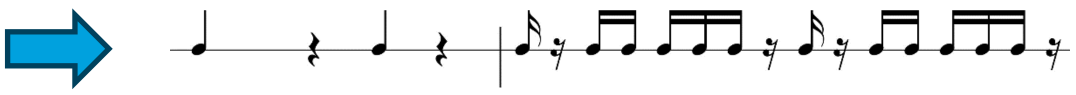


Figure 37: Two example measures formatted in sm-style translated rhythmically.

Using a resolution of 192 for every measure would be able to represent any score, however use of the other resolutions reduces file size and increases readability.

5.6.2 CHART files

CHART files are used to represent scores for ‘Clone Hero’.

5.6.2.1 Metadata

Again, the start of the file includes information about the song and how to sync the score with it:

```
[Song]
{
  Name = "mysong"
  Artist = "foo"
  Charter = "bar"
  Album = "myalbum"
  Year = ", 2023"
  Offset = 0
  Resolution = 192
  Player2 = bass
  Difficulty = 3
  PreviewStart = 0
  PreviewEnd = 0
  Genre = "Swing Trance"
  MediaType = "cd"
  MusicStream = "song.ogg"
}
```

```
[SyncTrack]
{
  0 = TS 4
  0 = B 135000
  26112 = B 270000
  38400 = B 135000
}
```
Figure 38: Example CHART metadata.

The key information to facilitate gameplay here is:

- 'MusicStream', specifying the song file to use,
- 'Offset', defining the time difference between the start of the song and start of the score,
- 'Resolution', which provides the resolution used for every beat,

Tempo and changes to tempo are defined in the 'SyncTrack' section and are marked with a 'B'. Tempo is provided as bpm*1000 and the timings are given in relation to the chosen resolution. For example, in the CHART file shown the tempo change to 270bpm would occur at '26112', corresponding to the 136th beat at 192 resolution, which given the previous bpm of 135 would put the change at just past one minute into the song.

5.6.2.2 Score

Notes are given in the same way as tempo, with a number defining where they should lie at the given resolution.

```
3264 = N 3 0
3312 = N 4 0
3744 = N 1 144
3744 = N 2 144
3744 = N 3 144
3936 = N 1 0
3936 = N 2 0
3936 = N 4 0
```
Figure 39: Example note data as formatted in CHART files.

The first number after the 'N' refers to the stream the note belongs to (between 1 and 5), and the second number refers to game-specific information such as whether the note should be held down and for how long, or whether the note can be tapped to play rather than strummed.

Like with SM files, all difficulties for a certain level are contained in the same CHART file.

5.6.3 TJA files

TJA files are used to represent scores for 'Taikojiro'.

5.6.3.1 Metadata

The start of the file again contains metadata:

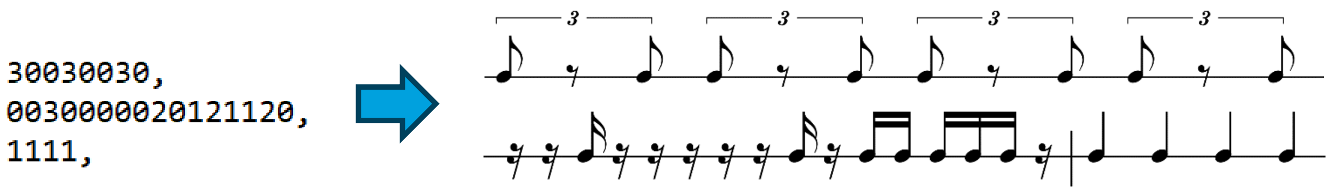
```
TITLE:mysong
SUBTITLE:--artist
BPM:227
WAVE:mysong.ogg
OFFSET:-3.481
SONGVOL:100
SEVOL:100
DEMOSTART:64.323
SCOREMODE:2
BGMOVIE:mysong.wmv
NEWSONG:1
MOVIEOFFSET:0
```

Figure 40: Example TJA metadata.

Key gameplay related information here is:

- ‘WAVE’, which specifies the song file,
- ‘OFFSET’, again defining the time difference between the start of the song and start of the score,
- ‘SONGVOL’ and ‘SEVOL’, defining the volume for the song and sound effects respectively, as a percentage,
- ‘BPM’, which sets the tempo for the entire song

5.6.3.2 Score



```
30030030,
0030000020121120,
1111,
```

Figure 41: Three example measures formatted in tja-style translated rhythmically.

Like in SM files, TJA scores are organized by 4-beat measure and each measure can use a different resolution to produce different note types. ‘0’ represents a rest, ‘1’ and ‘2’ represent the two different note streams, and larger numbers represent game-specific features such as hitting both note streams simultaneously. Unlike SM files, measures are organised horizontally which condenses the score further. This is possible as there are less than 10 possible inputs in ‘Taikojiro’ so they can all be represented as single digits. In ‘StepMania’ there are four note streams which can be pressed one at a time, two at once or not at all at each rhythmic step, already providing 11 possible inputs without accounting for additional inputs such as held notes.

5.6.4 AR files

After learning from the designs of similar filetypes, I designed my own format of text-based score representation for ‘Autorhythm’. These use ‘.ar’ as their extension, so the application can find and load them.

5.6.4.1 Metadata

```
#TITLE: sepia
#SONGFILE: sepia.mp3
#AVERAGEBPM: 123.0469
#NOTESTREAMS: 3
#NOTEPLACEMENT: [0] Default
#QUANTIZEPOLICY: [3] All note types allowed
#WINDOWSIZE: 7
#THRESHOLD: -0.1
```

The start of the file contains metadata on the score, as this is standard in the other filetypes looked at. These are marked with hashtags to make it easier for a parser to ignore this information and reach the score itself, which might be the case if the file is being used for MIR research. The information included consists of:

- ‘TITLE’, which is not strictly necessary to include but gives a reference to the file’s name from within the file, and it is standard to include a title in all of the other filetypes looked at.
- ‘SONGFILE’, which gives the name of the relevant song file required to play the level. This is both necessary for ‘Autorhythm’ to function and is standard in the other filetypes.

- 'AVERAGEBPM', which is used to set note speed in 'Autorhythm'. This is also not strictly necessary information as beat times are included later in the file and an average bpm can be extracted from them, but including an average bpm here removes the need for this and also helps with conversion into score file formats such as TJA that use a static bpm.
- 'NOTESTREAMS', which defines the number of note streams the level uses. Variable note streams is not a feature of any other rhythm game considered, but 'Autorhythm' needs this information when loading a level from file.

The last four entries all pertain to how the level was generated. These are not required by 'Autorhythm', or for conversion into other file formats, however preserving this data could be useful for future analysis. This is discussed later.

5.6.4.2 Score

The score section of an AR file provides information on beat timings, note timings, note number and note stream.

```
#(STREAM/NOTE NUMBER AT 192 RESOLUTION/TIMING)
#SCORE
                                  #0.116099773
3,         48,        0.60371882
                                  #0.603718821
                                  #1.13777778
                                  #1.67183673
3,         144,       1.67183673
3,         160,       1.84985639
3,         184,       2.11688587
                                  #2.13623583
1,         200,       2.21363568
2,         216,       2.36843537
                                  #2.62385488
3,         240,       2.62385488
```

The timings of beats are provided on their own lines and are marked with a hashtag to distinguish them from note information. Note information is given as an integer value for stream, an integer value for note number, and a decimal value for timing, seperated by commas and tabs.

The 'note number' of a note is essentially a measure of how far away a note is from the first beat in the score, and this value can be used to work out the rhythmic subdivision of the note without having to do calculations with the note and beat timings. 'Autorhythm' uses an irregular note resolution, as there are 12 available places a note can be placed every beat but these are not equally spaced.

Resolutions used in the different filetypes looked at are all regular, so for ease of conversion this note number is represented at a 192 possible notes per 4-beat measure resolution, even though this resolution must be awkwardly converted back into Autorhythm's resolution when loading from file. For example, a note placed on the first beat would have note number 0, while a note placed on the last 32nd note of the first beat would have note number 42, but when the level is loaded in 'Autorhythm' these notes would be numbered 0 and 11 repectively.

192 notes-per-measure resolution is accepted by all both SM and TJA files, and equates to a 48 notes-per-beat resolution in CHART files, so using this resolution should lead to much simpler conversion.

All of the filetypes studied featured an 'offset' to define where the score starts in relation to the music. AR files contain note timings so do not need this information, however if the file was converted into another format then the offset would be the timing of the first beat, which will always be the first value under the ‘#SCORE’ tag.

‘Autorhythm’ reflects changes in tempo in its beat placement. The other rhythm games studied that allow changes in tempo both explicitly define them in their score files, however. Conversion is still possible as the tempo at each beat in an AR file can be calculated using the time between it and the next beat. Alternatively, the average bpm can be used as an approximation.

6 Reflection and Future Work

In this chapter we will evaluate the extent to which the prototype version of 'Autorhythm' described in this paper met its aims, and discuss potential improvements and future directions.

Deliver a generalised note-playing rhythm game.

At its most basic aim, the application does indeed function as a basic, generalised rhythm game. Everything described in the implementation chapter works as intended; users are able to generate or load a level consisting of notes set against locally saved music, then play along and receive feedback on their performance as these notes scroll.

Existing rhythm games will often produce a single score to rate the user's performance, which is calculated differently from game to game and can take into account individual note accuracy, streaks of consecutively well-performed notes and game-specific features such as 'star power' in 'Guitar Hero'. Since the aim was to produce a generalised rhythm game, in order to generalise this scoring system I opted to keep individual tallies of each 'accuracy band' (perfect, great, OK, miss, misplayed) and provide no overall score.

This is, in essence, just a new game-specific scoring system though, as I have hard coded the size of each band and give the user no customisation options to tailor scoring to different level styles.

A potential improvement could be functionality for editing the size of each accuracy band to make a level harder or easier depending what the user feels is appropriate, and additionally adding a total score with customisable calculation options.

This scoring information can be saved with the rest of the metadata in a level's AR file, and could also be saved as a user preference that can be selected from the homepage when generating new levels.

Provide a user-customisable, expansible system of level generation.

The current level generation system is, to an extent, user-customisable and expansible. The system currently contains four options for both note placement and quantizing, and steps were taken to ensure that this system could be easily extended.

Both these processes take place in an external server, so edits or additions to the system can be done here with little effect on the game. Adding options would require some in-game adjustments, however, as these options would need to be added to dropdowns and may have to hide the 'advanced options' when selected, like the 'Create From File' placement option does.

Since the server contains all of the functions to perform generation, and the Unity application simply reflects these options, this system could be improved by having the server send all data needed to populate the options in the Unity application when connection between them is established. This way updates made to the server would be automatically reflected in-game without the need for UI edits.

The main drawback of the current generation system is that there is no method of note selection implemented. Once a level is generated, notes are assigned their streams randomly. This was done as, in my opinion, the most promising current methods of note selection are from 'Dance Dance Convolution'[4] and 'Tensor Hero'[18], both of which use neural networks and are specialised to specific styles of rhythm games. However, I have noticed in my own testing that it is hard to see the distance between and play notes that are in non-adjacent streams if they are very close to each other timing-wise. Even without the use of neural networks, a useful early note selection method might be to increase the probability of notes spawning in adjacent streams based on how close their timings are.

Additionally, this lack of note selection means that I have implemented no extensible method of choosing a note selection method. A fully expansible system of level generation should have this implemented, even if the only option currently is 'random selection'.

I am pleased, however, with the currently implemented default note placement method that uses onsets. All music with discernible rhythm contains beats and onsets, and so if beat and onset detection were to theoretically work perfectly then this method will produce levels for any song that could feasibly be used for a rhythm game. Early testing has produced a variety of levels that follow the underlying rhythms to various degrees of complexity, and the addition of advanced options to operate window size and threshold make this method highly user customisable.

As expected, the performance of onset-based placement methods is directly limited by the strength and accuracy of the onset detection technique used to facilitate them. I have noticed, for example, that librosa's detection technique that I use struggles to find onsets in very noisy sections of music. However, this note placement method can be easily adapted to use any improved or specialised onset detection function that generates an onset envelope, and so can be extended and refined in future.

Provide a high level of controller support.

Autorhythm currently supports any USB HID class input device as a controller, which encapsulates most modern USB input devices capable of connecting to a computer, so gives a fairly exhaustive level of controller support.

Increasing this further is the additional support of MIDI input. MIDI instruments are a natural fit to rhythm games, and the hope is their inclusion will lead to new and varied player experience, beyond just expanding the controllers available to them.

The current binding system for controllers has positive aspects and potential room for improvement. The listening function, which allows the user to press an input on any device connected to their computer to bind it, is very quick and intuitive, and so far the system has been tested with a 'Guitar Hero Live' controller, a MIDI keyboard, a MIDI pad controller and a standard computer keyboard with no issues. However, the user is currently able to bind inputs to all six input streams regardless of how many note streams are selected. A clearer system may be to hide the input binding boxes for streams that will not appear in game.

Provide a way for users to edit levels.

The in-game editor is fully functional and can be used to change, add and delete notes as required.

There has been thought put into user needs and interaction, and solutions implemented to make the task of editing more manageable. Users can play the level with note clicks and/or beat guidelines activated to gain a better understanding of how in-time the currently placed notes are and how they relate to each other rhythmically.

On opening the editor, the user is shown the section of the score corresponding to where they paused the game and, if changes are kept, on closing the editor they are returned to the section of the game corresponding to where they last edited the score, providing a streamlined way to flip back and forth between the two modes. However, in hindsight, I think it may be more intuitive and provide even quicker editing if there was no separate editor screen at all, and the user could simply activate an edit overlay and move notes as they appear in game view. This way a user could just scroll up or down to find the note they wanted to change, move it, hit play and hear it in context even quicker than with the current method.

Another potential improvement would be to allow the user to edit beat timings. One intended use for the editor is to correct perceived mistakes in the timings of notes, and the discrete spots where users can place and move notes to are determined by the beat timings. However it is worth noting that the beat tracking method used is imperfect, and it should not be assumed that the beats detects are always perfectly in-time themselves.

The current design still works, as the system and user can place notes on the 'wrong' rhythmic subdivision to counteract imperfections in beat tracking, for instance a note that should objectively be a 4th note could be placed slightly late as a 32nd note if the detected beat was slightly early. However, building levels in this way should not be encouraged, as notes should be placed on their true rhythmic subdivision if scores are to be useful for future analysis. It was mentioned in the aims and objectives section that some step selection methods benefit from knowledge of beat phase, but this is only the case if beat phases are correct.

Similarly, beat timings should be at the correct times, so overall I feel it would be useful to allow users to edit beat timings on a continuous scale in addition to note timings on a discrete one.

Save levels to file in a format that encourages both interoperability with other games and future analysis.

The application can successfully read and write to file to both save and load levels. As outlined in the previous section, thought has been put into interoperability and how these files might lend themselves to conversion into other file formats. I have not, however, implemented any of these proposed conversions, and although it goes beyond promoting interoperability with just file format it may be worth adding extensible options for file conversion in-game.

The format of AR files has attempted to make it trivial to extract timing, stream and phase information on each note in the score. The neural network note placement solutions looked at use the timing information for training, and the note selection solutions benefit from training on all three. Such solutions benefit from training on user-made or validated data rather than autogenerated data, but it is possible to determine how much a score has been edited due to the generation options contained in the metadata.

Re-generating a level with the same options will always produce the same outcome for phase and timing information, so this can be compared to a score to identify where notes have been moved by the user. A more explicit method of seeing how edited a score is from initial generation may be more useful however, as this method requires access to the server used to generate a level, so later versions may want to annotate user changes.

6.1 Web Server

The current local python server was only implemented for ease of use, and a promising direction for future work that would play into the strengths of the design of the current version of 'Autorhythm' would stem from replacing this local server with a web server.

Beyond making server maintenance easier and removing the requirement that the user needs to launch the server themselves, this opens up new possibilities. Levels can be optionally published and uploaded rather than just saved locally, allowing users to play levels other people have created. The audio files required for the levels should not be uploaded, but it is possible that references to songs hosted on music streaming services such as Spotify or Apple Music could be used in their place when playing an online level, and the application edited to facilitate playback through these platforms.

Uploaded levels could be rated by other users, and given tags based on genre, difficulty, intended controller and number of note streams. From a user's perspective, this rating and tagging system should make it easier to find and play good quality levels that they are interested in. From a scientific perspective, this creates a system in which users are encouraged to create new information for musical datasets. The result of this interaction will be a library of scores of the same format with a record of generation methods and amount of user editing used, pre-sorted by user-made tags into different categories and peer-reviewed for accuracy via the user rating system.

Well-reviewed user edited scores could then be used to train new generation methods similar to those outlined in [4], [18] and [19], better specialised to different controllers, difficulties and music genres, and these new methods can be incorporated back into the game as new generation options. Each such improvement may attract more users and help retain existing ones, who may then edit and publish more levels, expanding the data set further and allowing for more refined generation methods to be trained.

7 Conclusion

To conclude, I have created a functional prototype game that implements the core mechanics common to rhythm games but provides a user with the freedom to apply any song and controller to these mechanics, and the freedom to edit levels as they see fit.

In the context of current work in rhythm level auto-generation, I have provided a general method of note placement using onsets, given and evaluated options for quantizing placed notes to common rhythmic subdivisions, and have developed a system that allows for the inclusion of specialised note placement methods and lends itself to the development of such methods by encouraging user edits and providing a standard text-based save file format.

Appendix A

Code Extracts

‘Medium’ quantizing function:

```python
def stg_medium(gen_times, beat_times):
    # define how the beat will be divided up
    divisions = [0,3,5,7,9,11,13,15,17,19,21,27,29,31,33,35,37,39,41,43,45,48]
    divisions = [x/48 for x in divisions]
    # define which rhythmic subdivision lies between each pair of divisions
    grid_labels = [4,0,32,24,0,16,0,12,32,0,8,0,32,12,0,16,0,24,32,0,4]
    # provide which beat fraction each subdivision occupies
    grid_placements = [0,0,6,8,0,12,0,16,18,0,24,0,30,32,0,36,0,40,42,0,48]
    grid_placements = [x/48 for x in grid_placements]
    label_count = len(grid_labels)
    n = len(gen_times)

    snapped_times = []
    phases = []
    last = beat_times[0]
    index = 0
    available_spot = True
    closest_extra = 1000
    phase_index = 0
    #ignore notes before first detected beat event
    while gen_times[index] < last and index < n:
        index += 1

    for t in beat_times:
        beat_duration = t - last
        for i in range(label_count):
            while(index < n and gen_times[index] < t + divisions[i+1]*beat_duration):
                # this is only diff:
                if(grid_labels[i] > 8):
                    dist = t + grid_placements[i]*beat_duration - gen_times[index]
                    if(closest_extra > dist):
                        closest_extra = dist
                        extra_beat = t + grid_placements[i]*beat_duration
                        extra_beat_phase = phase_index

                elif(grid_labels[i] != 0):
                    # if we are on 4th or 8th note, snap an extra beat if there is one and resolve as normal
                    if(closest_extra != 1000):
                        snapped_times.append(extra_beat)
                        phases.append(extra_beat_phase)
                        closest_extra = 1000

                    if(available_spot):
                        snapped_times.append(t + grid_placements[i]*beat_duration)
                        phases.append(phase_index)
                        available_spot = False
                index += 1

            if(i != label_count-1):
                available_spot = True
                if(grid_labels[i] != 0):
                    phase_index += 1

        last = t

    return snapped_times, phases
```

Code for parsing SM files for rhythmic subdivision frequencies:

```python
import os, glob
import numpy as np

#global variables to hold counts for each subdivision
count4 = 0
count8 = 0
count12 = 0
count16 = 0
count24 = 0
count32 = 0
count48 = 0
count64 = 0
count96 = 0
count192 = 0
# set this to either 'Beginner:', 'Easy:', 'Medium:', 'Hard:', 'Expert:'
diff = "Beginner:"

def count_rhythmic_subdivisions(stepmap, file):
    resolution = len(stepmap)
    if(resolution == 0):
        return
    elif(resolution == 4):
        crs4(stepmap)
    elif(resolution == 8):
        crs8(stepmap)
    elif(resolution == 12):
        crs12(stepmap)
    elif(resolution == 16):
        crs16(stepmap)
    elif(resolution == 24):
        crs24(stepmap)
    elif(resolution == 32):
        crs32(stepmap)
    elif(resolution == 48):
        crs48(stepmap)
    elif(resolution == 64):
        crs64(stepmap)
    elif(resolution == 192):
        crs192(stepmap)
    else:
        print("BAD MEASURE RESOLUTION: "+str(resolution)+" FOUND IN: "+str(file))
    return


# 4 x4
def crs4(stepmap):
    global count4
    count4 += stepmap.count(True)
    return

#4/8 x4
def crs8(stepmap):
    global count4
    global count8
    for i in range(8):
        if(stepmap[i] == True):
            if(i%2 == 0):
                count4 += 1
            else:
                count8 += 1
    return

# 4/12/12 x4
def crs12(stepmap):
    global count4
    global count12
    for i in range(12):
        if(stepmap[i] == True):
            if(i%3 == 0):
                count4 += 1
            else:
                count12 += 1
    return

# 4/16/8/16 x4
def crs16(stepmap):
    global count4
    global count8
    global count16
    for i in range(16):
        if(stepmap[i] == True):
            if(i%4 == 0):
                count4 += 1
            elif(i%2 == 0):
                count8 += 1
            else:
                count16 += 1
    return

# 4/24/12/8/12/24 x4
def crs24(stepmap):
    global count4
    global count8
    global count12
    global count24
    for i in range(24):
        if(stepmap[i] == True):
            pos = i%6
            if(pos == 0):
                count4 += 1
            elif(pos == 3):
                count8 += 1
            elif(pos == 2 or pos == 4):
                count12 += 1
            else:
                count24 += 1
    return

# 4/32/16/32/8/32/16/32 x4
def crs32(stepmap):
    global count4
    global count8
    global count16
    global count32
    for i in range(32):
        if(stepmap[i] == True):
            pos = i%8
            if(pos == 0):
                count4 += 1
            elif(pos == 4):
                count8 += 1
            elif(pos == 2 or pos == 6):
                count16 += 1
            else:
                count32 += 1
    return

# 4/48/24/16/12/48/8/48/12/16/24/48 x4
def crs48(stepmap):
    global count4
    global count8
    global count16
    global count12
    global count24
    global count48
    for i in range(48):
        if(stepmap[i] == True):
            pos = i%12
            if(pos == 0):
                count4 += 1
            elif(pos == 6):
                count8 += 1
            elif(pos == 3 or pos == 9):
                count16 += 1
            elif(pos == 4 or pos == 8):
                count12 += 1
            elif(pos == 2 or pos == 10):
                count24 += 1
            else:
                count48 += 1
    return

# 4/64/32/64/16/64/32/64/8/64/32/64/16/64/32/64 x4
def crs64(stepmap):
    global count4
    global count8
    global count16
    global count32
    global count64
    for i in range(64):
        if(stepmap[i] == True):
            pos = i%16
            if(pos == 0):
                count4 += 1
            elif(pos == 8):
                count8 += 1
            elif(pos == 4 or pos == 12):
                count16 += 1
            elif(pos%4 == 2):
                count32 += 1
            else:
                count64 += 1
    return

# 4/192/96/64/48/192/32/192/24/64/96/192/16/192/96/64/12/192/32/192/48/64/96/192/
# 8/192/96/64/48/192/32/192/12/64/96/192/16/192/96/64/24/192/32/192/48/64/96/192 x4
def crs192(stepmap):
    global count4
    global count8
    global count12
    global count16
    global count24
    global count32
    global count48
    global count64
    global count96
    global count192
    for i in range(192):
        if(stepmap[i] == True):
            pos = i%48
            if(pos == 0):
                count4 += 1
            elif(pos == 24):
                count8 += 1
            elif(pos == 12 or pos == 36):
                count16 += 1
            elif(pos%12 == 6):
                count32 += 1
            elif(pos%6 == 3):
                count64 += 1
            elif(pos == 8 or pos == 40):
                count24 += 1
            elif(pos == 16 or pos == 32):
                count12 += 1
            elif(pos%2 == 1):
                count192 += 1
            else:
                count96 += 1
    return

def containsInput(string):
    if(string.find('1') == -1 and string.find('2') == -1):
        return False
    return True

def get_type(difficulty_code, file):
    dance_type = file.readline().strip()
    if(dance_type != "dance-single:"):
        return 0

    author = file.readline().strip()
    difficulty = file.readline().strip()

    if(difficulty == ""):
        if(difficulty == "Beginner:" or difficulty == "Easy:" or difficulty == "Medium:" or difficulty == "Hard:" or difficulty
== "Challenge:"):
            return 1
    elif(difficulty == difficulty_code):
        return 1

    return 0

def parse_measure(step, file):
    stepmap = []
    i = 0
    while i < 1000:
        i += 1
        if(step.startswith(";")):
            return stepmap, False
        if(step.startswith(",")):
            return stepmap, True
        if(step != ""):
            stepmap.append(containsInput(step))
        step = file.readline().strip()

    print("TIMEOUT ERROR VIA "+str(file))
    return [], False


path = os.path.join("existing_scores","sm")
x = 0
found_score = 0
scores_read = 0
divisions = []
# OPTIONS AFTER '#NOTES:' ARE:
# dance-single:
# dance-double:
# lights-cabinet:
# dance-couple:

for filename in glob.glob(os.path.join(path, '*.sm')):
   with open(filename, 'r') as f:
        for line in f:
            if(found_score > 0):
                if(len(line.strip()) == 4):
                    stepmap, more = parse_measure(line.strip(), f)
                    x += 1
                    count_rhythmic_subdivisions(stepmap, f)
                    while more:
                        stepmap, more = parse_measure("", f)
                        x += 1
                        count_rhythmic_subdivisions(stepmap, f)
                    found_score = 0
                    scores_read += 1


            elif(line.strip() == "#NOTES:"):
                found_score = get_type(diff, f)
```

Appendix B

Full Questionnaire answers

CLIP / PLACEMENT METHOD	in time	matching	difficulty	in time	matching	difficulty	in time	matching	difficulty	in time	matching	difficulty	in time	matching	difficulty	in time	matching	difficulty	in time	matching	difficulty	in time	matching	difficulty	matching	difficulty
Clip 1, onsets expert quantized	4	4	4	5	3	3	5	4	4	5	4	5	4	3	1	3	2	2	5	5	4	5	5	5	4	4
Clip 1, onsets hard quantized	4	2	5	5	4	4	5	4	4	5	4	5	4	3	1	3	2	2	4	5	4	5	5	4	5	4
Clip 1, onsets medium quantized	4	3	4	5	4	4	4	4	4	5	4	5	3	2	4	4	2	2	5	5	4	5	3	5	4	4
Clip 1, onsets easy quantized	3	3	4	5	3	4	4	4	4	5	3	4	1	2	5	4	3	2	4	5	4	5	5	3	3	4
Clip 1, DDC challenge expert quantized	1	2	5	3	2	3	4	4	4	3	2	4	1	1	5	3	2	2	3	3	5	4	4	5	3	5
Clip 1, unquantized onsets	3	3	4	5	4	5	3	2	4	4	3	3	3	4	3	3	2	2	4	4	4	4	4	5	3	4
Clip 2, onsets expert quantized	4	4	4	4	4	3	4	2	4	3	2	4	5	5	5	2	2	2	3	3	3	5	4	4	5	2
Clip 2, DDC challenge expert quantized	4	4	3	5	5	4	4	3	4	4	4	4	4	5	3	4	3	2	4	5	3	5	5	5	5	5
Clip 2, DDC hard expert quantized	2	3	3	4	4	4	4	3	4	5	4	4	4	4	4	4	3	1	4	4	4	4	3	5	4	5
Clip 2, DDC medium expert quantized	4	4	3	4	2	3	4	2	4	4	2	3	3	3	4	4	4	2	3	5	3	5	3	4	2	3
Clip 2, DDC easy expert quantized	3	4	3	4	2	2	4	3	4	2	3	2	2	4	3	2	2	1	3	4	2	4	3	4	2	3
Clip 2, DDC beginner expert quantized	4	4	3	3	2	1	3	2	4	4	4	1	4	4	1	2	2	1	3	4	2	5	1	3	3	1
Clip 3, onsets expert quantized	3	3	3	3	3	2	3	2	4	3	3	2	4	5	2	3	4	2	3	3	3	5	5	4	3	3
Clip 3, unquantized onsets	1	2	2	3	4	2	2	2	5	5	4	2	5	5	1	4	2	1	5	4	2	5	1	3	2	4
Clip 3, DDC challenge expert quantized	1	2	2	2	3	2	2	2	4	5	5	3	4	5	2	3	1	1	5	2	2	5	1	3	2	3
Clip 4, onsets expert quantized	1	1	2	3	3	3	3	3	5	2	2	3	3	4	3	2	3	1	3	3	3	5	5	5	3	4
Clip 4, unquantized onsets	1	2	2	2	3	2	3	3	5	4	4	3	3	5	3	3	3	1	5	5	2	5	4	4	4	4
Clip 4, DDC challenge expert quantized	2	2	2	3	3	2	3	3	5	5	4	4	3	2	3	4	2	2	5	5	2	5	5	5	2	5
Clip 4, onsets hard quantized	1	1	2	2	2	2	3	3	5	4	3	3	1	1	5	2	2	1	3	3	2	4	4	5	2	4
Clip 4, onsets medium quantized	2	2	2	3	1	2	3	4	4	3	3	3	1	2	2	1	2	1	2.5	3	2	5	5	3	3	3
Clip 4, onsets easy quantized	2	1	3	2	1	1	2	2	4	2	2	1	3	4	2	3	3	1	2	2	2	5	5	2	4	2
Clip 5, DDC challenge expert quantized	1	1	5	1	2	4	3	3	3	2	4	3	3	2	1	1	1	1	3.7	4	5	3	3	5	4	5
Clip 5, unquantized onsets	1	2	5	2	2	4	3	3	3	2	3	4	1	1	5	1	2	1	3	3	5	4	4	5	5	5
Clip 5, onsets expert quantized	3	3	2	2	2	2	3	3	3	2	4	2	1	1	2	2	1	2	4	2	1	5	4	4	5	2
Clip 6, unquantized onsets	4	4	4	4	4	4	4	4	5	5	5	4	4	4	2	2	3	2	5	5	3.5	5	5	5	5	4
Clip 6, DDC challenge expert quantized	4	4	5	2	3	3	4	4	5	5	5	4	5	3	3	3	3	2	5	5	4	5	5	5	5	5
Clip 6, onsets expert quantized	4	4	5	3	4	4	3	3	5	5	5	5	3	2	2	1	3	2	5	5	5	5	5	5	5	5
Clip 7, unquantized onsets	5	5	5	4	5	5	4	4	5	5	5	4	4	5	3	3	4	1	5	5	3	5	5	5	5	4
Clip 7, DDC challenge expert quantized	4	4	5	3	4	4	4	4	5	5	5	4	3	3	2	4	4	1	5	5	3	5	5	5	4	5
Clip 7, onsets expert quantized	3	3	5	3	4	3	4	3	5	4	4	4	1	2	5	2	4	2	3	4	3	5	5	5	3	5

References

[1] WikiHero, “Controllers and controller compatibility” [Online]. Available: **https://guitarhero.fandom.com/wiki/Controllers_and_controller_compatibility** [Accessed 08 2023]

[2] Z. Millsap, (2020, May 12), “Why the Guitar Hero and Rock Band Series Failed”, CBR [Online]. Available: **https://www.cbr.com/why-guitar-hero-rock-band-series-failed/** [Accessed 08 2023]

[3] Screen Play, (2012, June 20), “Pop goes the music games”, The Age [Online]. Available: **https://www.theage.com.au/technology/pop-goes-the-music-games-20120618-20k7b.html** [Accessed 08 2023]

[4] C. Donahue, Z.C. Lipton, J McAuley, (2017), “Dance Dance Convolution”, ICML'17: Proceedings of the 34th International Conference on Machine Learning, Vol. 70, pp. 1039–1048.

[5] Konami, (2019), “The 8th Konami Arcade Championship” [Online]. Available: **https://www.youtube.com/watch?v=X8k1Jggh9aE&t** [Accessed 08 2023]

[6] Wikipedia, “List of Dance Dance Revolution video games”, [Online]. Available: **https://en.wikipedia.org/wiki/List_of_Dance_Dance_Revolution_video_games** [Accessed 08 2023].

[7] Forbes, (2009, Apr. 27), “Will The Cradle Rock? Activision Picks Van Halen For Next Guitar Hero”, [Online]. Available: **https://www.forbes.com/2009/04/27/activision-van-halen-guitar-hero-technology-paidcontent.html?sh=84a63db615f1** [Accessed 08 2023]

[8] C. Belland, (2015, June 10), “Reunion Tour: The Best And Worst Of Guitar Hero”, GameInformer [Online]. Available: **https://www.gameinformer.com/b/features/archive/2015/06/10/reunion-tour-the-best-and-worst-of-guitar-hero.aspx** [Accessed 08 2023]

[9] C. Olivia, (2014, Dec. 13), ”Vib Ribbon”, THINK magazine, [Online]. Available: https://thinkmagazine.mt/vib-ribbon/ [Accessed 08 2023]

[10] S. Parkin, (2015, July), “The Making Of... Vib-Ribbon”, Edge Gaming Magazine, Vol 281, pp. 88-91.

[11] M. Estrada, (2014, Oct. 9), “Review: Vib-Ribbon”, Hardcore Gamer, [Online]. Available: **https://hardcoregamer.com/reviews/review-vib-ribbon/110688/** [Accessed 08 2023]

[12] C. Weatherley (2012, Nov. 12), “Vib Ribbon (PS)”, The Pixel Empire, [Online]. Available: **https://www.thepixelempire.net/vib-ribbon-ps-review.html** [Accessed 08 2023]

[13] J.P. Bello, L. Daudet, S. Abdallah, C. Duxbury, M. Davies, M.B. Sandler, (2005, Aug. 15), “A tutorial on onset detection in music signals”, IEEE Transactions on speech and audio processing, Vol 13, No 5, pp. 1035-1047.

[14] Music: La by Shane Ivers - **https://www.silvermansound.com**

[15] B. McFee, C. Raffel, D. Liang, D.P.W. Ellis, M. McVicar, E. Battenberg, O. Nieto, (2015) “librosa: Audio and music signal analysis in python.” In Proceedings of the 14th python in science conference, pp. 18-25.

[16] P. Pedersen, (1965, Dec.), “The mel scale”, Journal of Music Theory, Vol 9, No 2, pp. 295-308.

[17] J. Schlüter, S. Böck, (2014, May 4), “Improved musical onset detection with convolutional neural networks”, 2014 IEEE international conference on acoustics, speech and signal processing (icassp), pp. 6979-6983.

[18] E. Waissbluth, S. Carr, A. Popescu, J. Hu, “Tensor Hero: Generating Playable Guitar Hero Charts from Any Song.”, [Online]. Available: **https://www.ischool.berkeley.edu/sites/default/files/sproject_attachments/tensorhero_capstone.pdf** [Accessed 08 2023]

[19] S. Nakamura, (2018, Nov. 17), “Automatic Note Generator for a rhythm game with Deep Learning”, DataDrivenInvestor, [Online]. Available: **https://medium.datadriveninvestor.com/automatic-drummer-with-deep-learning-3e92723b5a79** [Accessed 08 2023]

[20] Old School Gamers, “Can Unreal Engine Make 2D Games?”, [Online]. Available: **https://osgamers.com/frequently-asked-questions/can-unreal-engine-make-2d-games** [Accessed 08 2023]

[21] A. Andrade, (2015, Nov. 5), “Game engines: a survey”, EAI Endorsed Transactions on Serious Games, Vol 2, Issue 6.

[22] Documentation: “Input System”, Unity Technologies, [Online]. Available: **https://docs.unity3d.com/Packages/com.unity.inputsystem@1.7/manual/index.html** [Accessed 08 2023]

[23] R. Damm, (2019, Oct. 14), “Introducing the new Input System”, Unity Blog, [Online]. Available: **https://blog.unity.com/technology/introducing-the-new-input-system** [Accessed 08 2023]

[24] keijiro, (2021, Aug. 5), “Minis: MIDI Input for New Input System”, [Online]. Available: **https://github.com/keijiro/Minis** [Accessed 08 2023]

[25] Documentation: “RhythmTool Documentation”, [Online]. Available: **https://hellomeow.net/rhythmtool/documentation/html/53f2927b-71fd-4719-aae5-34b7ff45a9ad.htm** [Accessed 08 2023]

[26] “Dance Dance Convolution”, [Online]. Available: **https://ddc.chrisdonahue.com/** [Accessed 08 2023]

[27] Y. T. Elashry, (2022, June 19), “Two-way communication between Python 3 and Unity (C#)”, [Online]. Available: **https://github.com/Siliconifier/Python-Unity-Socket-Communication** [Accessed 08 2023].

[28] Documentation: “Supported Input Devices”, Unity Technologies, [Online]. Available: **https://docs.unity3d.com/Packages/com.unity.inputsystem@1.0/manual/SupportedDevices.html** [Accessed 08 2023]