LONDON'S GLOBAL UNIVERSITY

UCL

Generating 3D Hand Animations For Virtual Guitar Lessons

Toby Best

MEng Mathematical Computation

Supervisor:

Yuzuko Nakamura

25th May 2020

Abstract

This dissertation presents the development of a virtual guitar tutor software using motion captured 3D hand animations, to allow musicians to learn the necessary hand movements to play songs at their own comfort.

The project aims to explore the usage of motion capture in the context of a learning environment, in order to gain better insight into the generation and application of realistic 3D animations, as well as its application for more mundane, everyday contexts as opposed to predominantly in the entertainment industry. To this end, the project provides insight into its usage toward teaching musical instruments, using a software platform that can convert guitar chords into their respective hand movements to play.

The project also aims to provide an easily accessible platform for potential musicians who may lack full usage of both of their hands to play (whether it be due to upper-limb deficiency, loss of finger mobility, for example), such as myself.

The end result, written in C# using the Unity engine, is a program that can take an input song and its respective musical score and MIDI file, and outputs a series of animations in 3D space in accordance with the song being played. The main challenge faced was the creation of the platform used to join the animations together in time with the song being played, as well as recording and capturing the animations themselves.

Acknowledgements

Firstly, I would like to offer my sincere gratitude to my supervisor, Dr Yuzuko Nakamura, for allowing me the opportunity to participate in such an exciting and unique project. Yuzuko was always helpful and willing to provide assistance and guidance whenever I was uncertain on how to proceed, and her readiness to talk about any issues that cropped up was an invaluable aid throughout the year.

I would like to thank the numerous members of the UCL Virtual Reality department, including Sebastian Friston, David Swapp and Felix Thiel, for their helpful advice and keen interest in the project, and for allowing me the opportunity to use their newly-acquired Manus VR Gloves to capture and record the animations used in the VR Labs. Without their support, this project would likely have never had the animations captured in real time and might not have made completion.

I would also like to thank my personal tutor and director of Mathematical Computation, Professor Robin Hirsch for his guidance and advice over my four years of study at UCL, as well as the countless lecturers, tutors and other staff members of the Computer Science department. Though as I write this paper it seems the everyone in the world has lost their mind due to the current global pandemic, the CS department has been incredibly clear and thorough with their guidance for students, and I feel they should be acknowledged for their work.

Lastly, I would like to thank my parents for always believing in me and supporting me during my studies, as well as taking care of me during these particularly unprecedented times.

Contents

I: Introduction

This chapter provides a brief overview of the project's purpose and what it aims to deliver in the final product.

I.1) Motivation

Although it has been in usage as early as the 1990s, motion capture is a relatively novel and cutting-edge technology used to create realistic animations and movement tracking by recording the movements of its actor. Since its conception, motion capture has predominantly been used for entertainment purposes; whether it be to provide realistic and dynamic animations for characters within a video game, tracking the user's movements within virtual reality to help sell the illusion they are interacting directly with the environment, or mapping computer-generated imagery over actors' facial movements in cinematography. This project thus aims to investigate and explore the usage of motion capture software for more mundane, day-to-day usage, particularly in the context of a learning environment, in order to better understand its potential applications.

One such example would be its application as a teaching tool for how to play a musical instrument. The main challenge within the project is to create a virtual guitar learner program, which can act as a substitute 'virtual tutor', enabling musicians to continue to learn new songs at their own rate during situations where a personal tutor or teacher is unavailable.

At minimum, the project aims to deliver an easily-accessible piece of software that can animate the hand gestures on a guitar to match the notes played, though potential expansions include improving the quality of the teaching by allowing the user to freely adjust the tempo and speed of the music, as well as create their own checkpoints and loops to practise specific points in the song. Additionally, to make it more accessible to everyone (and drawing upon my own personal experiences), the project aims to be able to accommodate for any users who might lack full use of both of their hands, whether that be due to an upper-limb deficiency, loss of full finger movement, or something more serious.

As an aside, when the project was initially being planned out back in the summer of 2019, it was impossible to fully predict the impact a global pandemic such as the current COVID-19 outbreak would cause; however, with lockdowns in place in almost every country, thus preventing opportunities for in-person musical lessons, the ability to learn instruments with virtual lessons would become even more important.

I.2) Aims and Objectives

I.2a) Aims

- To investigate and learn more about motion capture and its potential applications, particularly in a learning environment.
- To explore in greater detail the potential usage of motion capture for a more day-to-day case that is accessible for the general public to use.
- To gain a better understanding about how to generate 3D animations, specifically those obtained from motion capture, for future personal usage as part of a career.
- To create a tool that can allow people to learn how to play a musical instrument (in this case, the guitar) from the comfort of their own home, without the need of going out and hiring a tutor to teach them.

I.2b) Objectives

- To construct a platform for a working software animation that demonstrates, step-by-step, the necessary hand positions required for guitar chords in a piece of music, within fully rotatable 3D space. This virtual tutor would guide the user through the musical piece in real-time and can be easily adjusted to suit the user's comfort (e.g. speed of the animation/tempo of the song, repeat specific parts of the song, etc.).
- To create a wide library of animations for the 3D animations required, that can smoothly transition from one to another, and can be adapted to cover almost all chord shapes and note patterns in comfortable, easy-to-follow lesson.
- To provide an easily accessible method of teaching for musicians who, like myself, might lack full use of both hands (e.g. upper-limb deficiencies, loss of finger mobility, etc.). Accessibility for the users is a key component that ideally needs to be kept in mind throughout the project.

I.3) Deliverables

- An interactive program that acts as a virtual tutor to teach the user the hand movements required to play a song on guitar.
- A collection of 3D motion captured animations used within the program to animate the virtual tutor.
- The ability to create lesson files, generated from the 3D animations, that will teach the user how to play a specific song.
- A collection of music files containing songs, that can have lessons generated by pairing them with a musical score file (time and resources permitting).
- A collection of musical score sheets, that can be used by the animation database to generate a lesson when paired with the corresponding music file (time and resources permitting).
- Test and bug reports of the strategy used for testing the program, any errors that were discovered, and how they were subsequently resolved.

Figure 1.1 below shows a visualisation of the potential output structure.

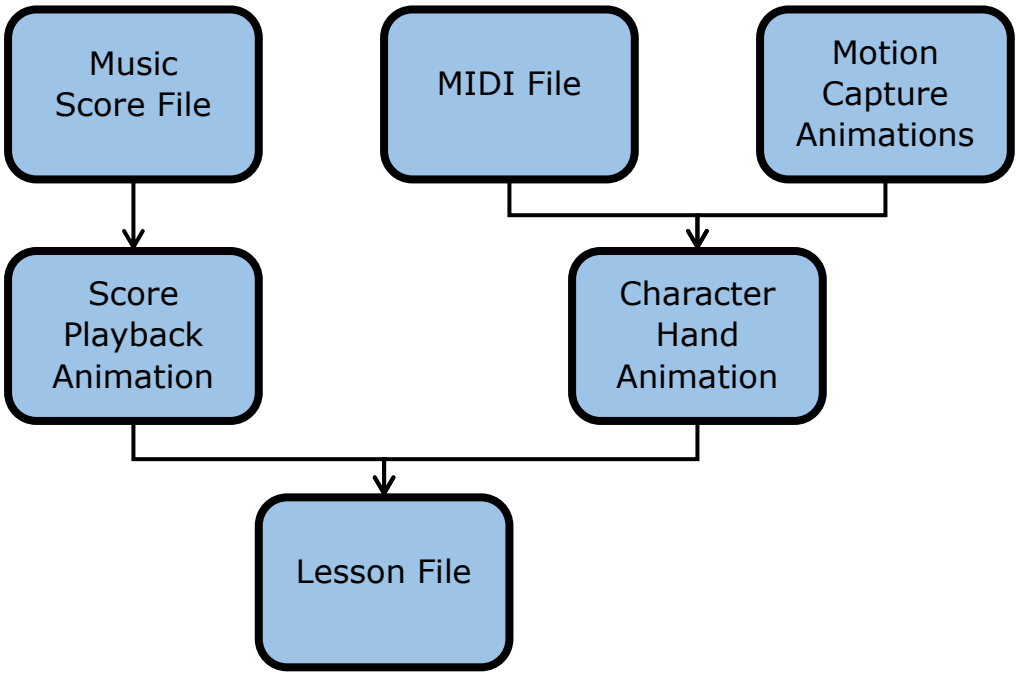

Figure 1.1: A diagram representation of the relationships between the various files within the project.

I.4) Project Approach

In accordance with the aims and objectives listed above, the project was split into four key stages: literature research and review on related work and similar projects; obtaining and recording the motion capture animations; constructing the software platform that utilises the 3D animations for the virtual guitar tutor program; and lastly testing and analysing the success and usefulness of the finished product.

I.5) Report Structure

Chapter I: Introduction

The current chapter details a brief overview of the purpose of the project and what it aims to deliver.

Chapter II: Project Context

This chapter details the initial research into the background of the project and related work on the topic at hand. It explores the motivation behind the project and an initial briefing of its implementation.

Chapter III: Requirements and Analysis

This chapter details the requirements and expectations of the project, including the MoSCoW Criteria (Must Have, Should Have, Could Have, Won't Have) that should be considered when creating the software platform.

Chapter IV: Design and Implementation

This chapter details the design and implementation choices, challenges and complexities encountered during the project's course, how and why these decisions were made, and what was done to resolve the issues that cropped up.

Chapter V: Testing

This chapter details the testing strategy used throughout programming the software platform, and the testing and debugging results are specified.

Chapter VI: Conclusion and Evaluation

This chapter details the overall effectiveness of the finished project, comparing what from the MoSCoW Criteria in Chapter III was met, what was not, why it was not met, and what could be done to improve upon the project if time were permitting.

Appendix

The appendix includes various miscellaneous additions, such as the system manual, the user manual, the use cases, the test cases, the project plan, the interim report, an explanation of the code listing, and a bibliography of citations.

II: Project Context

This chapter investigates previous research papers on automatic hand animations for playing instruments, relevant existing products for teaching how to play music, as well as any personal decisions and thoughts on the start of the project.

II.1) Literature Review

The origins of motion capture go back to rotoscoping in 1915, and modern motion capture technology has been in regular use since the 1990s. However, outside its usage in cinematography, it remains a relatively novel technology.

In terms of everyday applications for such animations, studies into representing the human hand in motion have revealed the anthropometrical and biomechanical constraints, such as muscles and bone structures, for a learning approach to generate natural-looking movement of the hand [1], as well how to simulate muscles to manipulate bone movement and deform skin tissue to render anatomically correct finger movements [2]. Research has been undertaken into applying these findings to play musical instruments; in particular, simulating natural movements for playing the piano, guitar and violin.

For virtual piano tutors, Lin and Liu explained how they generated realistic-looking piano fingering frames through an algorithm known as Slicing Fingering Generation, which slices through the note list in a MIDI file to create the finger positions, then sums the cost of each generated pattern to find the best fingering possible [3]. Meanwhile, Yamamoto et al. performed analysis on inputting piano-playing hand motions to output fully computer-generated animations of natural hand movement emulating its common features [4] and how realistic the end results compare to the original motions. Zhu et al. created an optimisation algorithm to refine computer-generated animations for piano playing to make them more natural [5], in an attempt to minimise the necessary hand movements for the user between the outputted poses.

For animating guitar hand movements, ElKoura and Singh present an educational tool for generating the animations needed to play a given piece of music on the guitar's frets [6]. This uses a complex, realistic model of the human hand that recreates the muscles, tendons and bone structure to emulate natural hand motion as the fingers reposition themselves on the frets to play the music.

For violin tutoring, Kim et al. used neural networks to optimally generate physically and musically feasible 3D hand animations [7], while Yin et al. produced a Digital Violin Tutor that creates a 3D avatar teacher to better show the motions needed to play the music, whilst also visualising any

mistakes that the learner might have made in following the teacher's performance [8].

Ultimately, the decision was made for the project to create a virtual guitar tutor, as this was an instrument that I am familiar with, having learnt to play the guitar for 9 years and only stopping when I came to university. As a result, I would have stronger prior knowledge of what sort of animations would be required for a guitar tutor than for what would be required for a piano or violin tutor.

Speaking from personal experience, there are many barriers to regularly practicing an instrument. Tutoring is generally done in person, which makes distance learning difficult. Finding a tutor is both a time-consuming process and expensive, especially after moving away from a tutor who you have built a strong rapport over the years and who understands how you learn and play. A virtual guitar tutor would therefore be useful to keep students engaged with instrument practice in the absence of a tutor. This is the main motivation driving this project.

Additionally, having been born with an upper-limb deficiency in that I lack beyond the elbow on my right arm, the guitar is the easiest of the three instruments researched to adapt for one-handed playing. Consequently, something that I wish for the project is that its final software development is easily accessible for people such as myself who may lack full usage of either of their hands. The software should accommodate for both left-handed and right-handed players and, if possible, should have alternative animations available for those who do not possess full finger mobility to make it easier for them to follow along with the lessons.

II.2) Other Guitar Learning Tools

II.2a) Songsterr

Songsterr [9] is an online archive of community-submitted guitar, bass and drum tablatures. It is a free and popular tool for learning musicians to teach themselves how to play new songs, thanks to its online tab player allowing them to play in real time along a MIDI-file version of the song. As it is an open practice tool for learning how to play the guitar, it has many features that would be considered desirable for this project.

Although the tab player on the website can be used without creating an account, users can register either a free or paid account to allow them to submit tabs to the community and suggest adjustments to existing tabs, akin to how Wikipedia allows users to submit revisions to its articles. The paid subscription also allows for additional features, including printing out the tablature, adjusting the tempo control, and looping specific parts of the

song. This greatly enhances the features offered by the tab player, improving the user's experience; however, it is a costly subscription at $9.90 per month, and not all budding musicians may have the expendable funds to afford that.

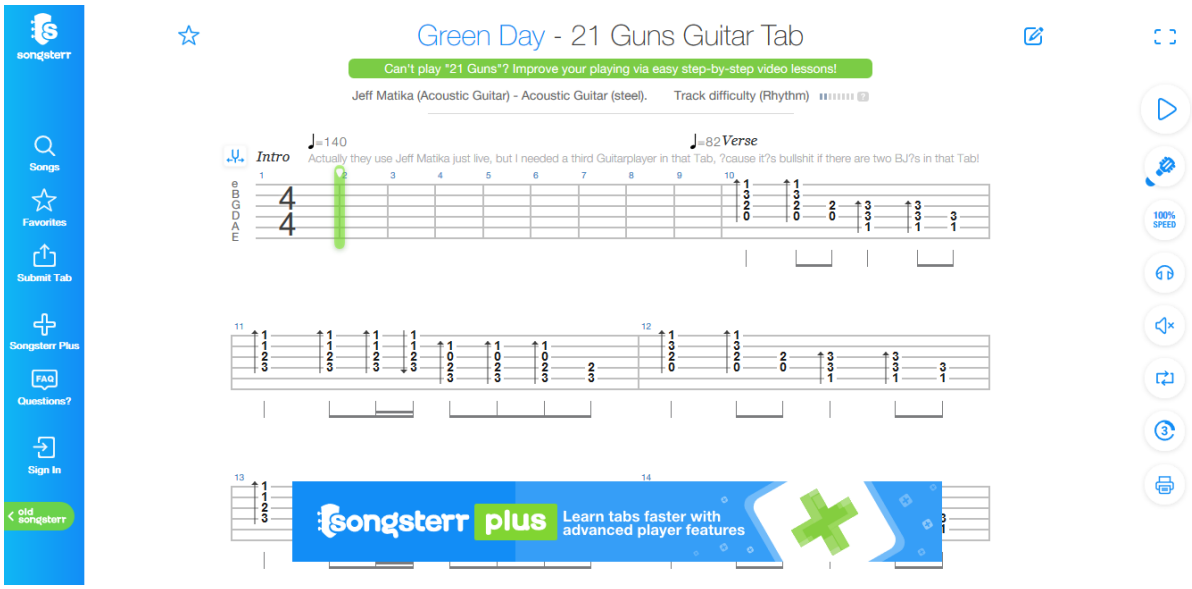

Figure 2.1: A screenshot of Songsterr's website layout for guitar tabs. The green bar represents the song's progress throughout the tab player.

Additionally, although the tabs can track for multiple instruments, such as a leading guitar, a backing guitar, a bass guitar and drums – even sometimes substituting vocals with a brass or woodwind instrument such as an oboe – the size of the music bars in the tabs can vary depending upon how often they are used. Figure 2.2 below shows an example of this. These inconsistencies can potentially cause confusion for the user if they are trying to keep track of multiple instruments at the same time.

Drawing from this, the program developed for this project needs to contain many of the features present on Songsterr for it to be considered a worthwhile alternative to the already successful platform. The paid account features offered by Songsterr improve the quality and experience of the teaching, in particular the option to adjust the song's tempo and create set loops within the song to better practice particular parts. The end result for this project will be freely distributed and can be used without the need for a paid subscription, so to ensure the best experience it can provide it needs to include functionality that would otherwise be considered premium in its base form. It will also include motion captured hand animations for better teaching of notes, which Songsterr does not include.

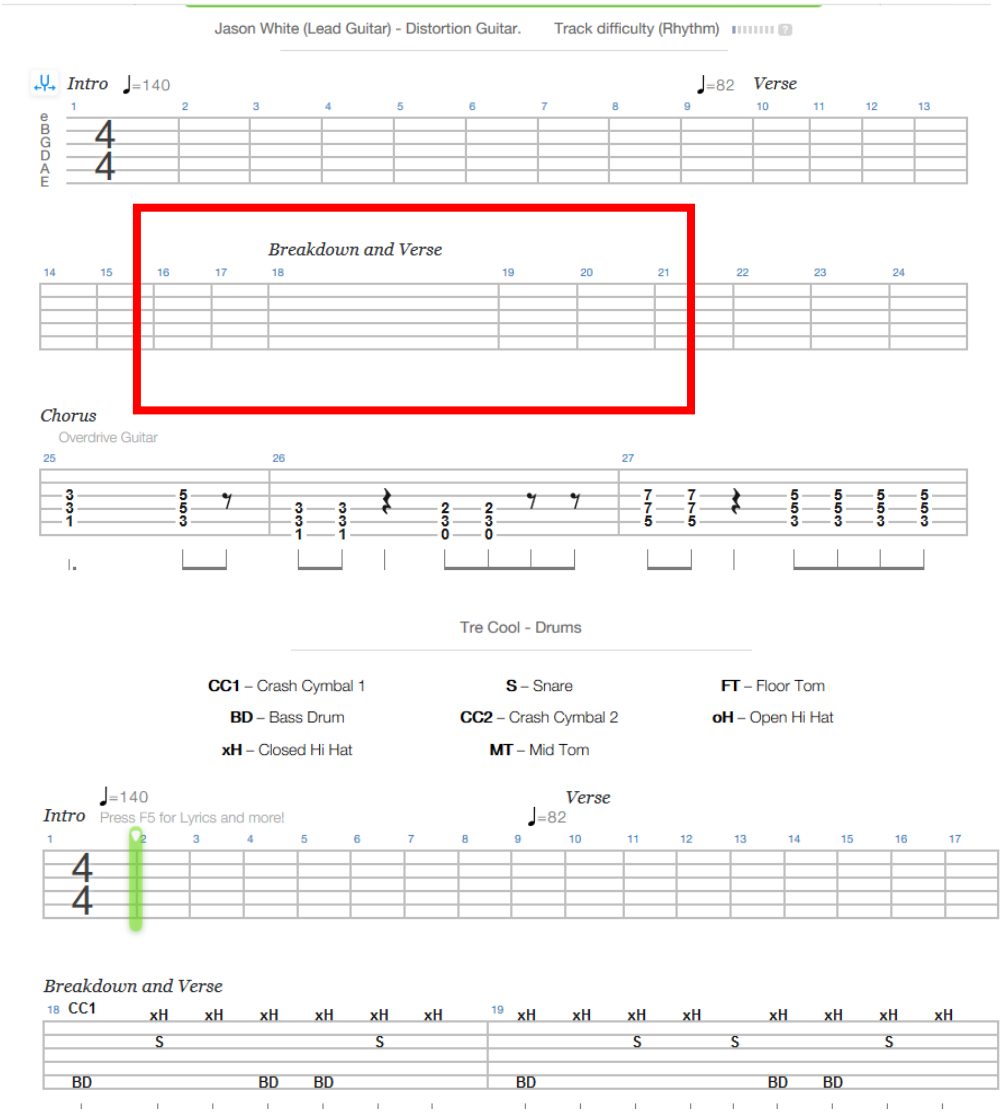

Figure 2.2: A comparison between two tabs for the same song (above: lead guitar; below: drums). Note how the empty bars do not keep a consistent size, particularly Bar 18 for the lead guitar (highlighted in red).

II.2b) Clone Hero

Released in 2017, Clone Hero [10] is a faithful, fan-made recreation of the rhythm-based Guitar Hero video game series, programmed with the Unity game engine. Rather than requiring the user to play a real guitar, the game uses a special guitar-shaped controller, with a simplified five button note system and a strum bar. As notes fly towards the player from the top of the screen to the bottom, the player must press the correct note button and hit the strum bar in time with the song.

Although the game itself is not of much use for this project, its online utility and community is worth looking into. Though it lacks the single-player story mode and currently only supports guitar gameplay (later Guitar Hero

games included functionality for other song components, such as drums, keyboard and vocals), Clone Hero has allowed its community to create game files based upon any song they wish, and share it online for all to access. As such, the player has access to all of the songs that were present within the Guitar Hero games, as well as those released after the last game in 2015.

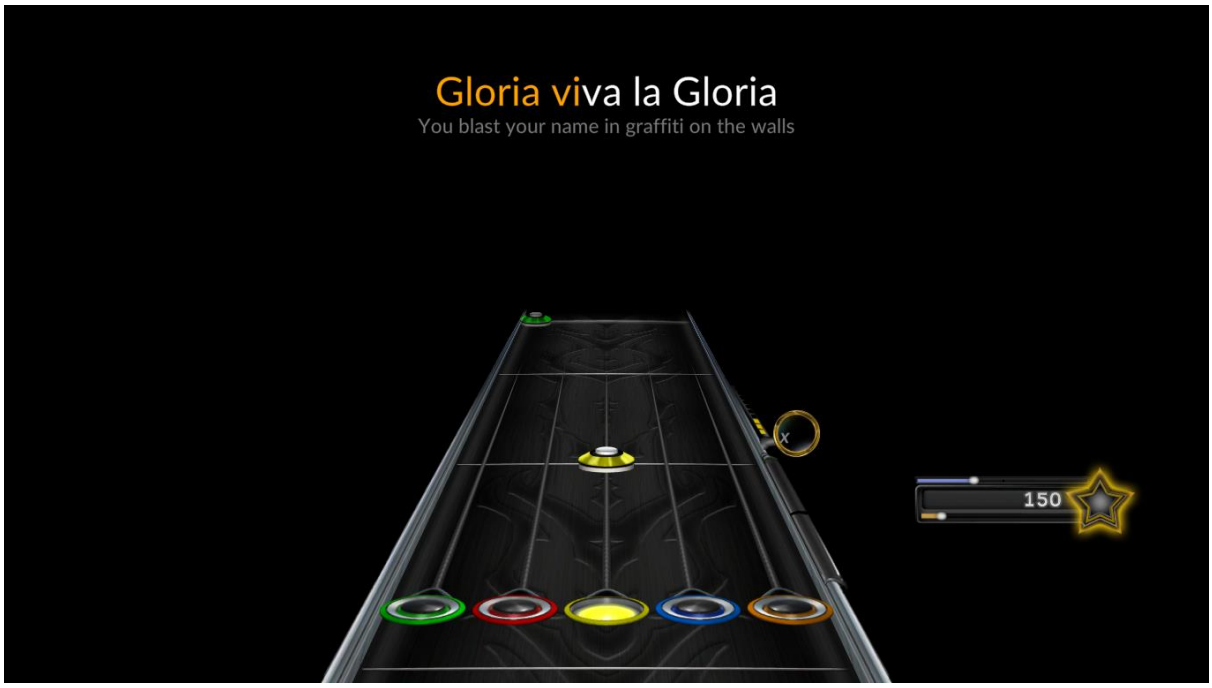

Figure 2.3: An in-game screenshot of Clone Hero. The player's five note buttons are at the bottom of the screen, and each note to play travels down the 'note highway' towards them. The player must press the correct note button and the strum button to play the note and score points.

Internally, the game splits up the song into separate sound files [11], with an example shown in Figure 2.4 below. It then then lays these files on top of each other to recreate the song. Certain songs may not have all of the files listed:

- album.png – The thumbnail of the song's album during selection.
- crowd.ogg – Sounds of the crowd cheering you as you play. Rarely used in Guitar Hero, and almost never used in Clone Hero.
- drums.ogg – Three distinct sound layers of the drums keeping the rhythm. A song might have multiple drums.ogg layers or none at all.
- guitar.ogg – The lead guitar. This is what the player is in control of. If the player misses a note, this layer is cut off until the next hit.
- keys.ogg – The lead keyboard. Only present if the MIDI file has notes for the player to play, otherwise it is ignored.
- preview.ogg – A 30 second extract of the full song that might be randomly selected on the main menu.

- notes.mid – A MIDI rendition of the song used to control the notes that the player needs to hit in order to score. This combines with guitar.ogg (or rhythm.ogg) to represent the player's performance.
- ps.dat – A leftover relic from a game called PhaseShift, a similar rhythm-based music game. Otherwise useless to Clone Hero.
- rhythm.ogg – The supporting instrument layer. Normally a rhythm guitar or bass but can also be other instruments such as violins. At the song selection, the player can choose to play this instead of the lead guitar.
- song.ini – A text file that includes details of the song, including its name, artists, album name, genre and duration, among other information.
- song.ogg – Any remaining instruments not covered in the other layers. This includes pianos, violins and backing vocals that lack MIDI notes for guitar.ogg and rhythm.ogg.
- vocals.ogg – The lead vocals.

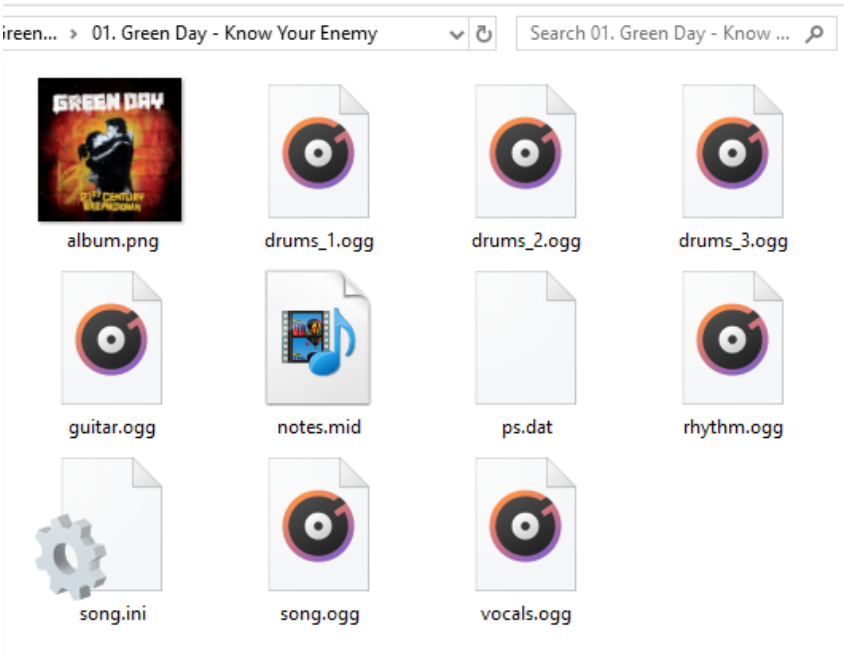

Figure 2.4: Example game files tied to one song entry.

Although Clone Hero is not strictly a tool for learning guitar tabs due to its oversimplified nature, I found its ability to read MIDI files to determine what notes need to be played an inspiration for controlling the hand animations in the project's final program. Rather than using the song's .mp3 or .wav file to control the animations, the MIDI file would provide a more accurate reading of what note has been played, and thus determine what hand position is needed and where in relation to the guitar's neck the hand's model should be.

III: Requirements And Analysis

This chapter details the requirements gathered as a result of the research done into previous published papers and the similar existing programs examined in Chapter II. The requirements are prioritised by the MoSCoW Criteria approach, detailed below.

III.1) Problem Statement

The purpose of this project is to plan out, design, implement and test a fully 3D virtual guitar tutor that can take any song and create motion captured hand animations for how to play it. With how expensive and time-consuming it can be to find a new music tutor, it can be quite impractical in everyday life to work with someone unfamiliar and begin rebuilding a rapport. The desired final product from this project will allow budding musicians to learn how to play their favourite songs from the comfort of their home, using motion captured realistic hand animations to teach the hand motions required to play the song.

The system will take a sound file (e.g. .mp3/.wav) of a song and a MIDI file relating to that song as an input, and then show the hand's movement up and down the guitar neck and what frets needs to be played for the correct note as its output. It must be robust and user-friendly enough to understand how to use it without hassle or complication, and ideally be accessible for any potential users (such as myself) who may lack full usage of both of their hands.

III.2) Requirements

The table below lists the planned requirements for the project, organised by a MoSCoW Criteria system. Each goal has been categorised into five lists, as to whether it relates to the system as a whole, the animation handling, the user interface, customisation options, or miscellaneous. The criteria uses the following grading system for each goal to determine its priority:

- Must Have: This requirement is of the highest priority and must be completed in order to consider the project a success.
- Should Have: This requirement is of a high priority and ideally should be implemented if possible, though are not mandatory for completion.
- Could Have: This requirement is of a low priority and not necessary for completion, but it is desirable for the project if possible.
- Won't Have: This requirement will not be implemented for this project but may be considered for future plans if sufficient time and resources are available.

ID	Description	Priority
System Overview		
S1	The system will provide a teaching platform for playing the guitar.	Must have
S2	The system will use lifelike 3D animations to teach the user in real time.	Must have
S3	The system will be programmed in C#, using the Unity engine.	Should have
S4	The system will cater to both left-handed and right-handed users.	Should have
S5	The system will cater to users with upper- or lower-limb deficiencies.	Could have
S6	The system will have online functionalities.	Won't have
S7	The system will be available on iPhone and Android phones.	Won't have
S8	The system will provide animations and tutorials for learning other instruments, such as the piano or violin.	Won't have
Animations		
A1	The animations will transition fluidly and smoothly between each other.	Must have
A2	The animations will be completely rotatable in 3D space.	Must have
A3	The animations will remain constant every time the same song is played.	Must have
A4	The animations will adapt, depending upon the chord positions, to minimise the required hand movements.	Should have
A5	The animations will be compatible for people without full usage of both hands (missing hand, loss of finger mobility, etc.)	Could have
A6	The animations will be compatible for people without full usage of either hands (i.e. there are problems with both hands)	Could have
User Interface		
U1	The user interface will be easy to use and understand.	Must have
U2	The user interface will allow the user to pause, resume, rewind and fast-forward the song.	Should have
U3	The user interface will allow the user to easily skip between parts of the song.	Should have
U4	The user interface will allow the user to create certain loops in the song to better practise certain parts.	Should have
U5	The user interface will be professionally-made.	Could have
U6	The user interface will have multiple options for languages.	Won't have
Customisation		
C1	The user can adjust the tempo of the music and speed of the animations to practise at a more comfortable level.	Should have
C2	The user can import music files (.mp3, .wav, etc.) and have the system teach them songs of their own choice, provided there is a matching music sheet uploaded.	Could have
C3	The user can create their own music sheets for the system to teach them, much like similar free programs (MuseScore, PowerTab, and more primitive software such as Mario Paint Composer).	Could have
C4	The user can scan in and import music sheets, which the system will convert into animations.	Could have
C5	The user can edit any imported music sheets, in case of errors that might have been created from conversion.	Could have
Miscellaneous		
M1	The final version will be free to distribute and download.	Must have
M2	The final version will have a wide variety of demo songs available to test with.	Could have
M3	The final version will make you a pro at Guitar Hero.	Won't have
M4	The final version will solve the Riemann Hypothesis.	As if!

III.3) Requirements Analysis

The core component of this project is understanding the applications for realistic motion captured hand animations in the context of a virtual guitar tutor, so naturally this will be of the highest priority. The code will also be written in C#, using the Unity engine, as this is what Clone Hero was written in and how it handles the MIDI files; it is also the programming language and engine used in my Virtual Environments module earlier in the year, so I am very familiar with the tools being used.

The final platform's accessibility is something that needs to be kept in mind, however accommodating for an unpredictable amount of variation in the user's hand mobility (such as if there are any fingers or a whole hand that is missing or immobile) means that this is not as important. At the very least, the animations should be able to be mirrored so it can teach both left-handed and right-handed players.

The platform will be restricted to teaching only the guitar, as piano and violin tutors already exist as shown by the research papers previously explored. Additionally, there are no plans to release a compatible version of the platform for mobile devices, nor to implement online connectivity to download lessons directly. If the scope of the project was larger, with a suitable community where this would be advantageous, then perhaps direct downloads within the program would be worth considering.

For the user interface, the project must be simple enough for anyone to understand how to use the product – though making it look exciting and have a unique design would be nice, it is safer to have a plain, bland design that does not sacrifice user-friendliness. The UI should also allow the user to perform functions present in Songsterr's subscription accounts, such as tempo control and setting checkpoints, to improve the teaching experience and give it the edge over its competition. However, having the function to customise the lessons directly within the program, such as what notes are needed to be played or generating new lesson files, is not as important a feature to include; this would be more suited for a larger scope project, such as one with online connectivity, so that the community could freely share the generated files with each other directly.

Last of all, as both Songsterr and Clone Hero are free applications in their basic form, the project's software will also need to be free to download and distribute, otherwise there will be little purpose in using it as a learning tool while other free alternatives exist. That said, the number of songs present in the basic form is not a necessary component for distribution – it is more important to ensure the tool is of high quality for a small quantity of songs.

III.4) Use Cases

Use Cases are a way to define the interactions between the user and the system, by highlighting the necessary action that must be taken to achieve the goal and the system's response to these actions. Figure 3.1 below shows an example diagram.

Please refer to Appendix A.C for the full list of use cases.

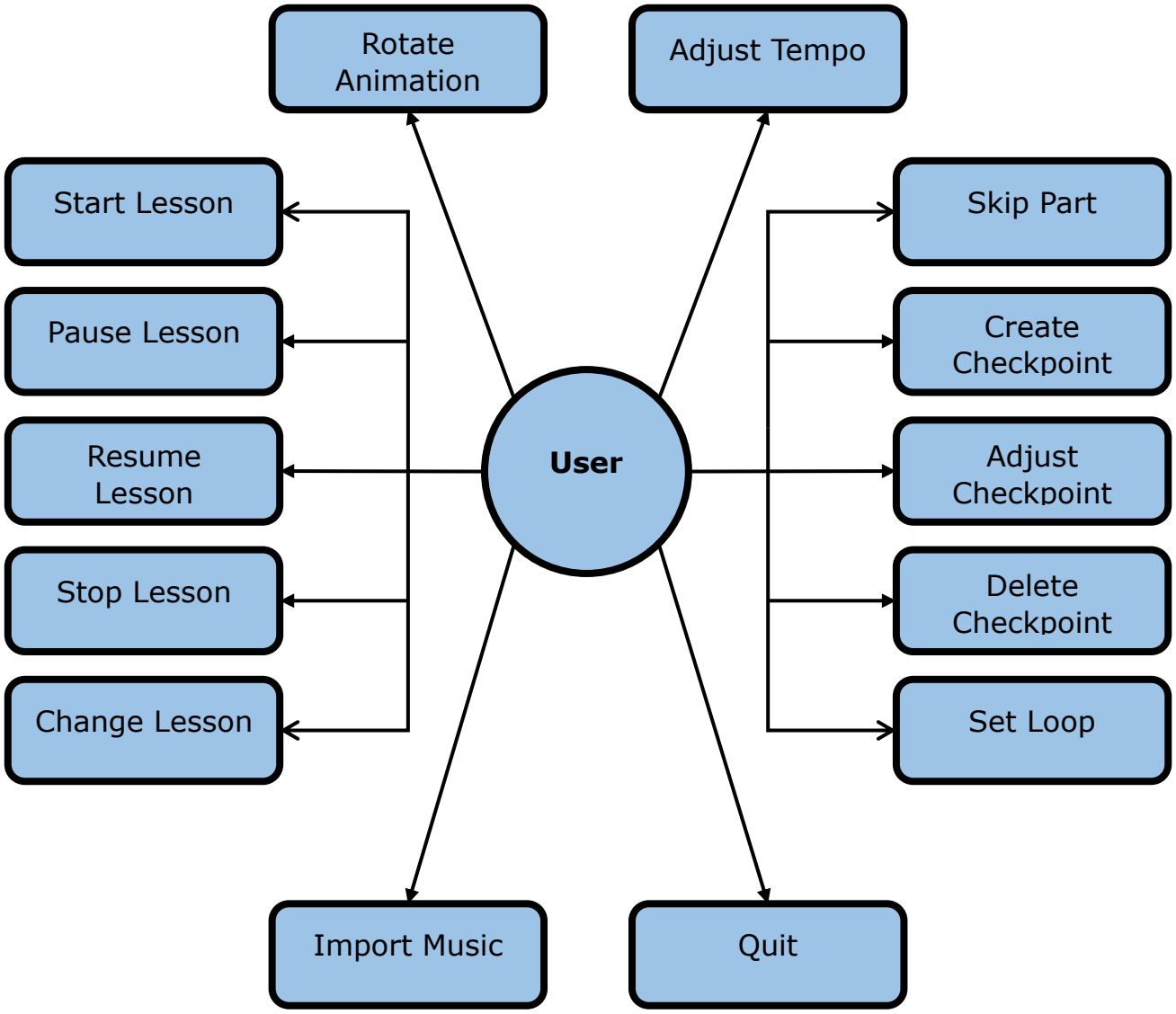

Figure 3.1: Example diagram of Use Cases

IV: Design And Implementation

This chapter details the design and implementation of the project, based upon the planning detailed in Chapter III, as well as explaining the reasoning behind the choices and decisions made, any problems that were encountered during the implementation, and what actions were taken to resolve them.

IV.1) System Overview

IV.1a) Unity

The Unity game engine [12] is a free and openly-accessible video game development tool that allows its users to create both 2D and 3D programs. As the Unity editor is easy to understand and quick to learn from, and as Clone Hero was constructed using it, it was deemed suitable for use within this project to create the 3D virtual guitar tutor. The version used for this project is Unity 2019.2.6f1, the most up-to-date version at the start of the project.

The primary programming language of Unity is C#, a multi-paradigm, object-oriented programming language that is an extension of the common C language. As alternative programming languages such as JavaScript had their compatibility dropped prior to the version used, C# was the only choice available for this project as a result. A built-in copy of Microsoft Visual Studio 2019 to edit the C# code comes with Unity.

IV.1b) System Layout

Figure 4.1: All of the folders stored within the Assets folder.

As a Unity project, all of the key files are stored within the Assets folder. Each file is organised into separate files for ease of access, including imported files from downloaded prefabs and assets obtained from the Unity Asset Store. The key folders relevant to the project are as follows:

- Animations, which contains all of the recorded motion captured hand animations.
- Prefabs, which contains all of the complex game objects and entities used in the Guitar Tutor scene pre-compiled to speed up the loading process.

- Scenes, which contains all of the program scenes for the user to interact with.
- Scripts, which contains all of the C# code files for processing the entities.
- Songs, which contains all of the song files to be loaded by the Guitar Tutor scene.

The program is split into four key scenes, each stored within the Scenes folder, which render everything necessary for the user to interact with their environment and to learn how to play the guitar based upon the recorded animations.

- MainMenu.unity, a simple introductory scene that gives the user the instructions for how to use the program, as well as a guitar model for them to freely rotate and move around to get used to the controls. The user can either move onto the Song List to select what song they wish to learn from, or quit the application entirely.
- SongList.unity, a scene which loads all of available song files in the Songs folder, where the user can select which song they would like to learn from in the tutor. They can also select whether they want to learn with the left hand or right hand. The user can go from here to either the Main Menu or the Loading Screen.
- LoadingScreen.unity, a simple loading screen meant as a buffer between Song List and Guitar Tutor. This automatically sends the user to the Guitar Tutor when it loads.
- GuitarTutor.unity, the main scene where the animations and song player are stored. Here the user can manipulate the song files in order to learn how to play the song to their comfort, such as slowing it down or creating checkpoint loops for practising specific parts of the song. The user can also exit back to Song List if they are finished with the current song and wish to learn a new song.

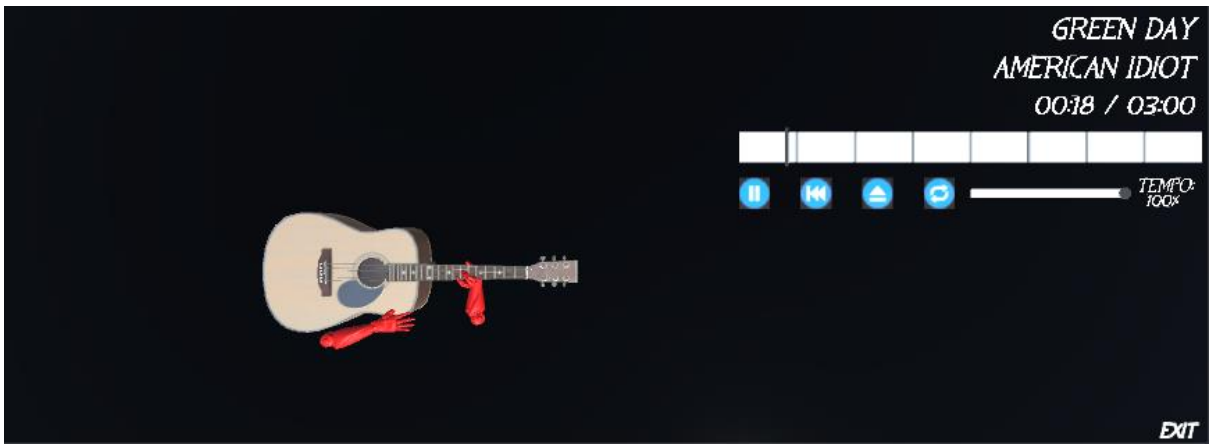

Figure 4.2: Example screenshot of the GuitarTutor scene, with the song and animations playing.

The Scripts folder contains all of the C# code that runs the project. The most important scripts are covered in more detail in the Code Listing section of Appendix A.G, but a brief overview of the code is as follows:

- AnimateHand.cs – controls the hand animations when playing.
- ButtonClick.cs – controls the music controller buttons.
- Checkpoint.cs - controls a checkpoint when a new one is instantiated.
- CheckpointButtonHandler.cs - controls the UI button that relates to a checkpoint.
- CloseCanvas.cs - closes the current UI canvas.
- LayerButtonHandler.cs - controls the song layers to allow for enabling/disabling of specific pieces of the song.
- LoadLevel.cs - loads the selected scene.
- LoadTutor.cs - loads the Guitar Tutor scene (only used for the scene Loading Scene, and for debugging purposes).
- MasterController.cs – a static variable holder used to carry the file path and which hand to use between scenes.
- MidiReader.cs - reads the song's midi file from the song's file.
- MusicLayers.cs - loads the separate song files from the song's file.
- PlaySong.cs - controls the song's play progress.
- ProgressBar.cs - allows the user to manipulate the song's progress.
- RotateGuitar.cs - allows the user to manipulate the guitar model's position, as well as the hands to animate.
- RotateGuitarMenu.cs - identical to RotateGuitar.cs, but without the hands, to play around with in the Main Menu.
- SongListMenu.cs - loads all of the valid song folders for the user to select one in the Song List scene.
- SongPreview.cs - controls the song's path, album and preview sound file.
- SongTracker.cs – a simple storage for the progress bar’s song tracker’s coordinates, to visualise the song’s progress.
- Strum.cs - controls the strumming hand (as no strumming animations could be recorded in time due to the COVID-19 pandemic and university lockdown).
- Tempo.cs - allows the user to manipulate the song's tempo.

The Songs folder contains all of the songs that can potentially by used by the Guitar Tutor to play animations based on their MIDI file. The bare minimum required for a song file to be accepted is a song.ini file, a guitar.ogg file and a notes.mid file with a track labelled ‘GUITAR’. This follows a similar architecture design to how Clone Hero stores its song files; a decision that was made after examining its efficiency, allowing for potential compatibility between the two programs.

This would allow the potential end user to copy over any song that is compatible with Clone Hero and - theoretically - be able to produce hand

animations from the same MIDI file. Consequently, it becomes easier for the user to learn from the songs they want to use, as they can simply find the desired songs from the Clone Hero community database [13] and download it into this project's folders, rather than having to try and convert the song to a compatible format themselves.

IV.2) Obtaining The Motion Capture

IV.2a) Manus VR Gloves

Although there are many existing motion captured hand animations available on the internet, such as those published as part of the BigHand2.2M Benchmark [14], I decided with my supervisor that it would be easier if I personally recorded my own hand movements for the project, rather than using an existing data set or recording the movements of someone else. This saved us having to fill out an ethics form or risk assessment form and removed the need to get the consent of a third-party outside of UCL.

This led us to organising a discussion with the Virtual Reality department at UCL, with whom I had previously undertaken a module titled Virtual Environments in Term 1 of this year, to discuss what technology the university already had available for recording motion capture. After a meeting between myself, my supervisor Yuzuko Nakamura and Sebastian Friston of the VR Department, Sebastian recommended that we use the newly-acquired Manus VR Gloves [15] to track and record the hand movements. The Manus Gloves can allow for real-time hand movement tracking and can connect directly to software such as Unity and MotionBuilder for usage in virtual reality applications, letting the wearer interact directly with the environment with their hands instead of a controller.

After experimenting with the Manus Gloves and getting acclimatised to using them, I set about recording the animations.

IV.2b) Recording The Animations

Originally, the plan was to connect the Manus Gloves to MotionBuilder [16], a 3D animation editing software commonly used for capturing motion captured animations for realistic movements in cinematography, in order to record the hand movements and then port them to Unity. However, following several technical issues with trying to get the movements to record and save, an alternative in Unity itself was found. Two importable asset extensions, the Scene Motion Capture [17] and Animation Baker [18], allow the user to record and save any movements of game objects in the

Unity engine as animation files. This allowed the Manus Gloves to be instantiated in Unity and capture the hand motions whilst the Animation Baker converted their movements to an animation file.

Unfortunately, due to the COVID-19 pandemic causing a national lockdown, from 16th March UCL was closed and, as a result, the Manus Gloves and the technology required to record their movements in the Virtual Reality department were no longer accessible. Consequently, the project only has a limited number of motion captured animations to work with, each of varying quality and usefulness, rather than a large list of high quality animations to choose from. With no ability to record better quality animations, the rest of the project has continued on with this limitation in mind, working with what was available.

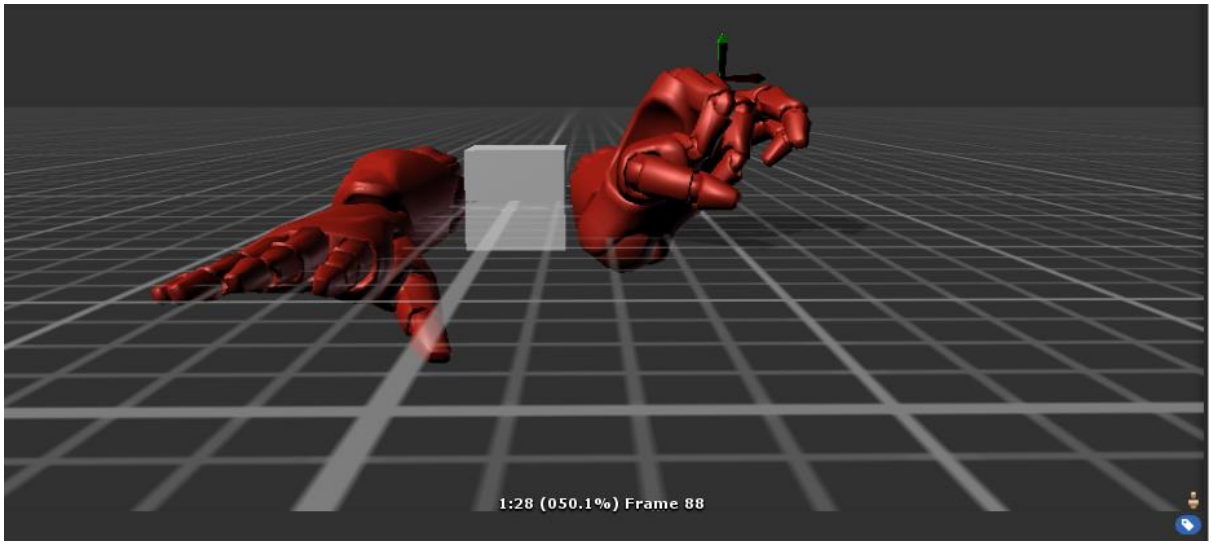

Figure 4.3: Example screenshot from Unity of a motion captured animation recorded by Animation Baker using the Manus VR Gloves.

IV.2c) Editing The Hand Models

As one of the requirements for the project is to teach for both left-handed and right-handed play, a method to swap hands was included. In the SongList scene, the user has the option to toggle which hand to use. When GuitarTutor loads, the correct hand model is enabled. Rather than spending time creating two separate models for both left hands and right hands, Unity uses a depth mask to disable the rendering of the unwanted hand, allowing the same mesh to be used.

However, as a consequence of Animation Baker capturing the hand motions for both Manus Gloves when only one hand (the left hand) was in motion, in order to ensure that the animations would play for the right hand instead of the left, the model was mirrored in the y-axis. This decision was taken to save a lot of time and headaches in remapping the mesh in a tool such as MotionBuilder or Blender.

Although no proper animations were recorded for strumming the guitar, the Strum.cs script controls the movements of the opposite hand so that it strums up and down the guitar whilst the song is playing.

IV.3) Animation Handling

IV.3a) Parsing The MIDI File

Following the decision for this project to share architecture layout with Clone Hero, the choice was made that the controller for the hand animations should base its actions upon the song's MIDI file within the folder, instead of reading from a guitar tablature sheet. The notes.mid file in the song's folder was loaded into the project using functions imported from the DryWetMIDI C# library [19], and the code for handling it is stored in MidiReader.cs file.

The program loads the MIDI file at the start of the GuitarTutor scene, then individually checks the name of each track stored within to find the track relating to the guitar notes. Once the guitar track has been found, it then loads all of the chords in the track (any collection of notes that occur at the same time interval) to prepare for playing.

IV.3b) Mapping Animations To MIDI Events

If the song is playing, the Midi Reader searches through all of the chords until it finds the most recent chord to play. If a new chord is found, MidiReader.cs sends the chord's information to AnimateHand.cs, which handles the individual animations.

By default, AnimateHand.cs assumes that the guitar used is a six-stringed, 24-fret guitar tuned to the default E-A-D-G-B-E tuning; this gives the thicker open-6th String a lower E2 note and the thinner open-1st String a higher E4 note. Figure 4.4 below shows the note played by each fret on each string.

DryWetMIDI stores each note as a seven-bit number, with the middle C (C4) stored at 60. AnimateHand takes the seven-bit number for each open-string note ('Fret 0'), then calculates the seven-bit number for each fret and string for reference – for example, the note numbers for each open string are 40 (E2, 6th String) 45 (A2, 5th String), 50 (D3, 4th String), 55 (G3, 3rd String), 59 (B3, 2nd String) and 64 (E4, 1st String).

Guitar Notes In Relation To Piano Octaves

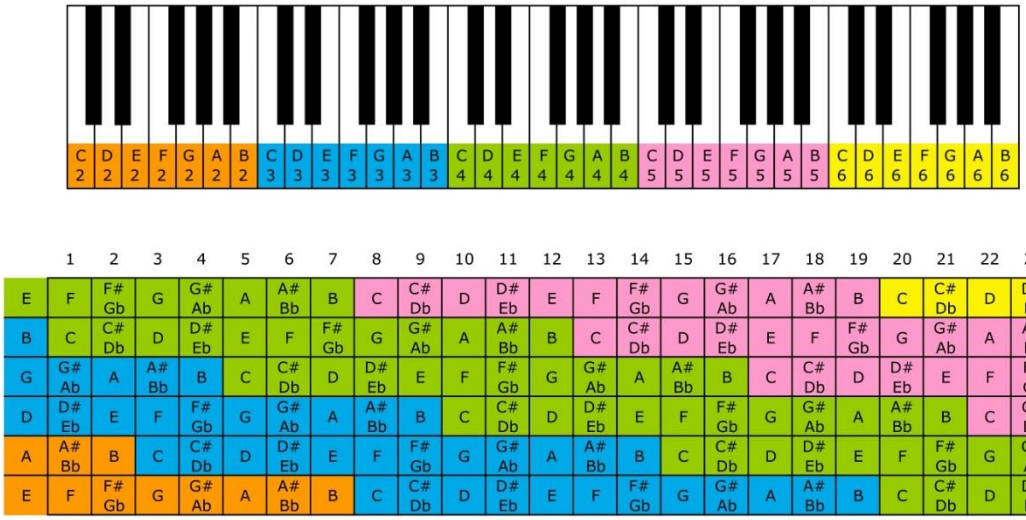

Figure 4.4: Relationship between the octaves on a guitar fretboard to piano keys for the E-A-D-G-B-E tuning. [20]

When a new chord is received from the MIDI file, AnimateHand first compares whether the base note of the chord falls within the fretboard of the guitar, as sometimes the MIDI file includes notes as low as C0 or as high as B8. For the sake of this project, it also clamps the maximum range searching to 17th Fret 1st String (A5), as realistically almost no guitar chord meant for a six-string guitar will have its base note that high.

If the note can be played on the fretboard, it then cycles along each string to find what would be the most comfortable fret to play the chord on – it would be more efficient to play a B2 note as 7th Fret 6th String than try for a 2nd Fret 5th String if the previous notes were also played on the 6th String. Once the optimal fret has been decided, the hand moves to that fret, using Vector3.Lerp to move smoothly from the old position to the new position.

To calculate the distance the hand must move to play the correct fret for the new note, the positions for each fret must be found first. The following formula calculates the distance from the nuts of the guitar strings to each fret.

$$Length_0 = 0\text{mm}$$

$$Length_i = Length_{i-1} + \left(\frac{Scale\ Length - Length_{i-1}}{17.817}\right), 1 \leq i \leq 24$$

The scale length is the length of the guitar strings. This approximately puts the middle 12th fret of the fretboard at half of the scale length [21]. In the program, the scale length is assumed to be the distance between the hand model's world coordinates and the guitar model's world coordinates.

Once it has determined the fret it needs to move to, AnimateHand instructs the hand model to play out a specific animation relating to the notes it has read. It reads off the base note value and then determines which chord hand position would most accurately recreate that position.

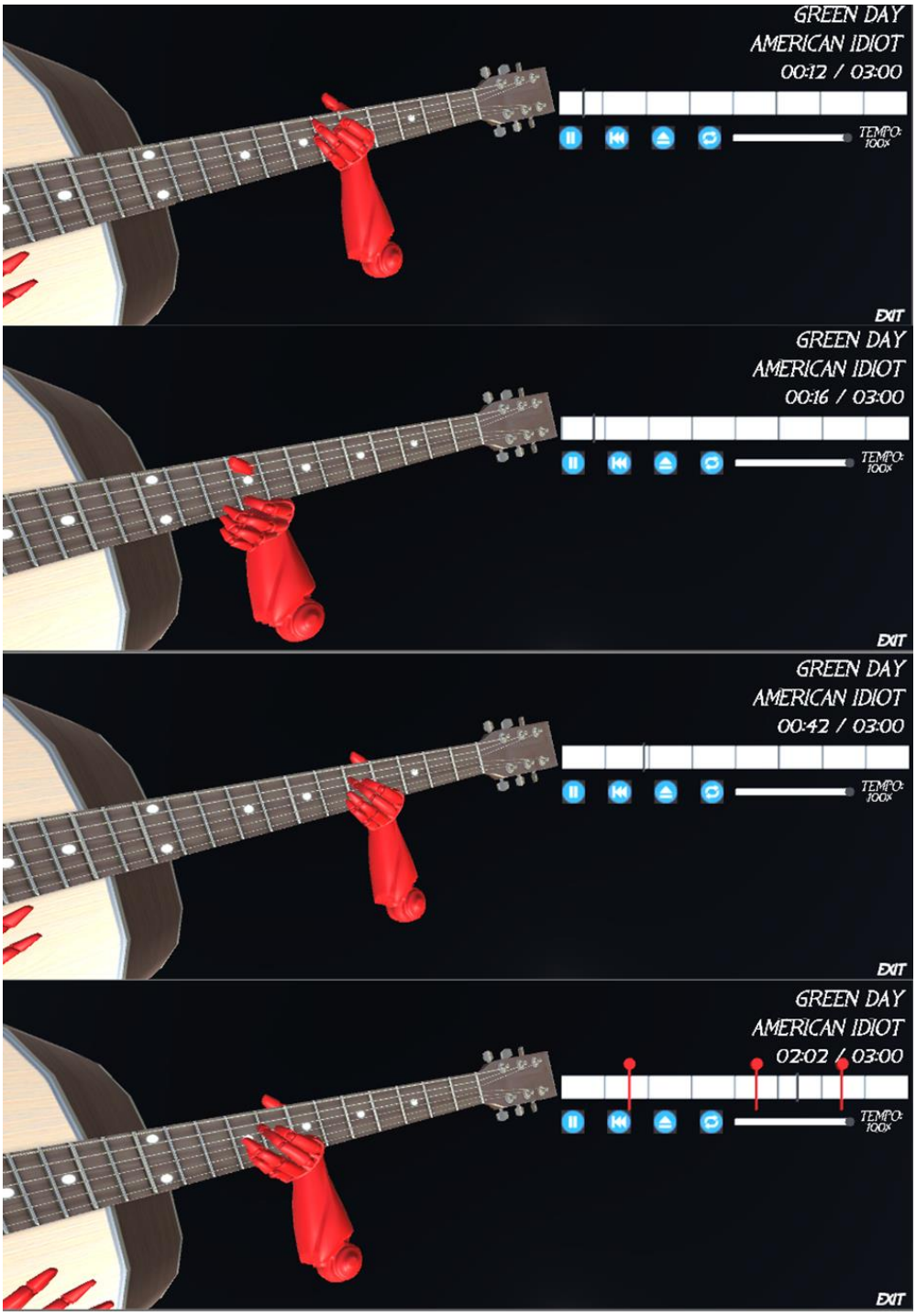

Figure 4.5: Example screenshots of the animation handling in process.

This process of deciding what animation to play has been scaled down as a result of the pandemic limiting what animations are available to use in its database; ideally, the program would try to accurately recreate the individual finger positions for each note in the chord, but as such a feat is

impossible with what is available, it instead only plays a sample chord position based upon what the base note is.

IV.4) Song Player Features

IV.4a) Loading The Song Files

Much like in Clone Hero, the program can allow the user to select a specific song that they have installed in the Songs folder. In the SongFile scene, a scrollable list is created with each song folder saved in ./Songs, as long as the folder has a notes.mid file, a guitar.ogg file and song.ini file. When a song title is clicked on, the load path is set to that folder, so when the Play button is clicked, the GuitarTutor scene loads and reads from that folder. If the folder also has an album.png file and/or a preview.ogg file, they are both displayed in this scene to give the user a visual or auditory cue as to what song is selected. Here, the user can also select which hand they would like to play with.

Figure 4.6: Example screenshot of the Song Selection UI list.

Once the user has selected which song they would like to learn from, the program makes a file path to the chosen folder and examines its contents. The collection of the sound files is handled in the code file MusicLayers.cs, and the general handling of the song files and initialisation is covered in PlaySong.cs, both of which are attached to the SongPlayer game object. This collection occurs at the same time as MidiReader.cs loads the MIDI file from the same folder.

The code checks the selected folder and loads all .ogg sound files found, as this is the sound format used by Clone Hero and thus would allow the user to directly import song folders they might use for that program. It disregards the 'preview.ogg' (which is used only for selecting the song in the menu) and 'crowd.ogg' (which is added only for gameplay purposes in

Clone Hero) sound files and instantiates new Audio Source objects as children of SongPlayer, set to play an individual sound layer file.

While the .ogg files are being loaded, MusicLayers.cs also checks for the existence of a song.ini file. If one is found, it retrieves the song's name and the song artist's name and displays them in the music player.

Once all of the sound files are assigned to a unique Audio Source object, PlaySong.cs checks all of the SongPlayer object's children for the guitar layer file, and sets it as the primary Audio Source object. This is the object that SongPlayer obtains its information from, such as the song length and current progress, and was chosen because every song folder used in Clone Hero contains a guitar.ogg file, so any song folder copied to use in this program will automatically be compatible. The song can then be played by clicking on the Play button.

IV.4b) Enabling/Disabling Song Layers

By clicking on the Layers button, the Song Layer User Interface will be displayed. This lists all of the layers that compose the song and allows the user to individually mute or unmute each layer, or mute/unmute all of the layers at once.

As the sound files are loaded concurrently and thus do not guarantee a set order, and to ensure that the layer buttons are listed in some sort of order, the SongPlayer organises each Audio Source child object alphabetically before instantiating a button in the UI. The buttons will tell the user if their layer is muted or not.

Figure 4.7: Example screenshot of the Song Layer UI list with various layers muted.

Being able to toggle specific layers in a song is a premium feature used in Songsterr, and as Clone Hero already separates the song into individual layers, this feature was also included.

IV.4c) Progress Bar

The Progress Bar is a visual indication of the song's current playing progress. Although the text on the UI displays the song's length and the song's timer, this allows the user to skip forwards or backwards in the song by left-clicking on the bar. ProgressBar.cs is the code file used.

Two visual trackers are used to help guide the user – the song tracker moves constantly along from left to right as the song plays to show the song's progress, and the mouse tracker follows the user's mouse movements around to show them where the song tracker would be moved to if the user were to left-click on the Progress Bar.

IV.4d) Checkpoints And Looping

If the user instead middle-clicks on the Progress Bar, they can create a checkpoint. Checkpoints act as a visual reminder for the user of a specific part of the song that is of interest, such as the start of a particularly difficult guitar solo, and store the song's timer when they are instantiated. Each Checkpoint's individual data is stored in Checkpoint.cs, while ProgressBar.cs handles the creation and management of all checkpoints.

Left-clicking on a checkpoint sets the song's timer to the checkpoint's timer, skipping the song back to that point as if the user clicked on the Progress bar. Middle-clicking on a checkpoint instead destroys the checkpoint. By default, only 8 checkpoints can exist at once, and trying to create any more yields an error message.

As with the Song Layers, clicking on the Checkpoint button underneath the Progress Bar will yield the Checkpoint List UI. Here, the user can choose to save the current checkpoints in a .txt file (which stores a list of all of the checkpoints' times), load the saved .txt files (spawning new checkpoints as a result), or enable/disable song looping.

Song looping is a feature that takes two selected checkpoints and causes the song to loop continuously throughout between them – if the song's timer is registered as going beyond the time stored by the second checkpoint, it skips back to the start of the first checkpoint. By default, if no checkpoints are selected, the song will simply loop from the very start of the song and the very end of the song. This can allow the user to set specific training points for them to work with and improve upon – for example, the beginning and the end of a guitar solo.

Being able to set loops in the song is a Songsterr premium feature, hence its inclusion in this project.

IV.4e) Tempo Controller

The Tempo Controller is a feature that allows the user to manually adjust the playback speed of the loaded song and of the hand controller. This can allow them to slow down the song in order to better practise part of the song. The code that controls the tempo is located in Tempo.cs.

In Unity, an Audio Source object has a Pitch value which, misleadingly, increases the speed of the sound (both the pitch and the tempo) loaded, rather than increasing the pitch while leaving the tempo alone. In order to change the tempo but not the pitch, an Audio Mixer must be linked to the Audio Source object. Then, the program takes the Mixer's Pitch value (which controls both pitch and tempo) and Pitch Shifter value (which only controls the pitch), and adjusts them in accordance with the Tempo Controller's new tempo value. For example, to decrease the tempo from 100% to 75%, the Pitch is decreased from 100% to 75%, while the Pitch Shifter multiplier is changed from x1.00 to x1.33 (its reciprocal value); this has an overall effect of slowing the sound layers whilst keeping them at the same pitch level, as well as slowing the speed of the hand animations for playing.

Originally, it was desired to slow down the tempo of the song as low as 25%, however one limitation of the Unity game engine is that the Pitch Shifter value can only range between x0.5 and x2.0. As a result, the tempo range the program has to offer is 50% to 100%.

Much like sound layer control and setting loops, being able to control the song's tempo and playback speed is a feature that Songsterr has available for its paid subscription, so it was decided to be an included feature here as well.

IV.4f) Guitar Movement

The guitar model used can be freely rotated and moved around in the GuitarTutor scene so that the user can get a closer look at the hand positions needed. Left-clicking and dragging the mouse over the model will spin it around, while the arrow keys or WASD keys move it up, down, left or right in accordance with the camera. The model can move closer towards the camera with the Q or comma key, and move it further away with the E or full stop key. Right-clicking anywhere in the scene resets the guitar to its original position.

V: Testing

This chapter details the testing strategy used throughout programming the software platform, and the testing and debugging results are specified.

V.1) Testing Strategy

In order to test the virtual guitar tutor's efficiency, effectiveness and functionality, a two-part testing strategy was devised. The first phase would be testing the code during its core development, ensuring that it can at least function correctly, using a single constant file input. The second phase would be testing the code after its implementation with a variety of different file inputs, to check that its outputs are as expected.

Throughout the program's construction process, unit tests were implemented to ensure that each component of the song player functioned as intended. This would ensure that the individual prefabs of the guitar tutor scene would function correctly in isolation, so that the functionality tests performed near the end of the program's development could be carried out with minimal interruption; for these tests, the same song was set for the file path (Green Day's American Idiot) to ensure consistency whilst debugging.

Once the program was fully operational, functionality tests were carried out to ensure that the software was robust and performed as intended. This was primarily covered by implementing a vast array of testing songs downloaded from the Clone Hero community database [13] and inputting them as the file path. This debugging and testing was to make sure that the input songs would be handled the same way as the sample song used in the unit tests.

A few key example tests and their results have been included below in this chapter, however Appendix A.D contains a full list of the tests carried out.

V.2) Manual Unit Testing

The unit tests were carried out manually on each part of the guitar tutor as they were being coded, and as such a lot of the debugging process occurred alongside the development.

Unit Test 1	Test Play Button plays the song when paused, and pauses the song when playing.
Input	Sample Song set to Green Day – American Idiot. Click on the Play Button.
Expected Result	PlaySong pauses the song if it is playing, and plays the song if it is paused.
Actual Result	PlaySong pauses the song if it is playing, and plays the song if it is paused.
Reason	Everything working as intended.
Changes Made	No change necessary.
Unit Test 2	Test Reset Button restarts the song.
Input	Sample Song set to Green Day – American Idiot. Click on the Reset Button.
Expected Result	PlaySong resets the song back to the start. All of the animations stop and return to their original position. The guitar returns to its original position and rotation.
Actual Result	PlaySong resets the song back to the start. All of the animations stop and return to their original position. The guitar returns to its original position and rotation.
Reason	Everything working as intended.
Changes Made	No change necessary.

Unit Test 4	Test Layers UI can enable/disable individual song layers.
Input	Sample Song set to Green Day – American Idiot. Click on the Layers Button to bring up the UI. Click on an individual layer.
Expected Result	Layer disables if enabled and text updates. Layer enables if disabled and text updates.
Actual Result	Layer disables if enabled and text updates. Layer enables if disabled and text updates.
Reason	Everything working as intended.
Changes Made	No change necessary.

Unit Test 6	Test song loops when commanded up Looping UI.
Input	Sample Song set to Green Day – American Idiot. Click on the Loop Button to bring up the UI. Click on Enable Looping from the UI.
Expected Result	Song loops between the checkpoints set if enabled. Song stops looping if disabled. If no checkpoints are set to loop between, song loops between whole song.
Actual Result	Song loops between the checkpoints set if enabled. Song stops looping if disabled. If no checkpoints are set to loop between, song loops between whole song.
Reason	Everything working as intended.
Changes Made	No change necessary.

Unit Test 7	Test Progress Bar updates the Song Tracker as the song plays.
Input	Sample Song set to Green Day – American Idiot. Song is playing.
Expected Result	Song time text is updated as the song plays. Song Tracker game object's position is updated.
Actual Result	Song time text is updated as the song plays. Song Tracker game object remained stationary.
Reason	Accidentally set Song Tracker's original position to be constantly updated, rather than its current position, causing the object to remain inanimate.
Changes Made	Changed what position was checked to update the Song Tracker.

Unit Test 9	Test Tempo Bar updates the song's tempo for all layers when clicked.
Input	Sample Song set to Green Day – American Idiot. Mouse clicks on the Tempo Bar.
Expected Result	The Audio Mixer changes the tempo based upon the position clicked, slowing down/speeding up the song as required.
Actual Result	The Audio Mixer changes the tempo based upon the position clicked, slowing down/speeding up the song as required.
Reason	Everything working as intended.
Changes Made	No change necessary.

Unit Test 10	Test guitar model rotates when clicked on.
Input	Mouse clicks and drags over the guitar model.
Expected Result	Guitar model and the playing hands models rotate as the mouse moves.
Actual Result	Guitar model and the playing hands models rotate as the mouse moves.
Reason	Everything working as intended.
Changes Made	No change necessary.

Unit Test 12	Test hand animations move the hand models to the right fret.
Input	Sample Song set to Green Day – American Idiot. Song is playing.
Expected Result	AnmiationHandler moves the hands to the desired fret.
Actual Result	AnimationHandler moves the hands to less optimal fret.
Reason	AnimationHandler simply chose the smallest fret that gave the correct note available on another string, rather than accepting the fret on the current string if it minimises movement to relating chords.
Changes Made	Changed the code so that it checks whether the current fret/string combination is a sensible solution, favouring the lower, deeper strings with a higher fret value over a higher string with a lower fret value.

Unit Test 13	Test whether a valid song list is displayed in the Song List.
Input	Sample Song set to Green Day – American Idiot.
Expected Result	Song List loads Green Day – American Idiot from the Songs folder and displays it in the list.
Actual Result	Song List loads Green Day – American Idiot from the Songs folder and displays it in the list.
Reason	Everything working as intended as the valid files necessary for the virtual tutor to function were detected.
Changes Made	No change necessary.

Unit Test 14	Test whether the next scenes load when clicked.
Input	Click on the Play/Exit button on each scene to load the next scene.
Expected Result	Play Button on the Main Menu loads the Song List. Play Button on the Song List loads the Loading Scene, which automatically loads the Guitar Tutor. Exit Button on the Guitar Tutor loads the Song List. Exit Button on the Song List loads the Main Menu. Quit Button on the Main Menu produces a prompt, which closes the application.
Actual Result	Play Button on the Main Menu loads the Song List. Play Button on the Song List loads the Loading Scene, which automatically loads the Guitar Tutor. Exit Button on the Guitar Tutor loads the Song List. Exit Button on the Song List loads the Main Menu. Quit Button on the Main Menu produces a prompt, which did not close the application.
Reason	Unity will not close the application if the program has not been fully built – as this was still in the editor, the Quit Button from the Main Menu did not close the program, but worked as intended otherwise.
Changes Made	No change necessary.

V.3) Functionality Testing

The vast majority of the functionality tests were visual-based, due to the nature of the project, particularly with regards to whether the MIDI reader and animation controller were reading the guitar chords correctly. The hand movements would be compared with the notes to play on a corresponding guitar tab sheet to test their accuracy – which, admittedly, is rather low due to the limited number of animations available following the disruption caused by the COVID-19 pandemic.

Below are a few example tests that were carried out.

Functionality Test 1	Test all of the buttons in the entire program can be clicked.
Input	Click every button in the program to ensure they function correctly.
Expected Result	Every button's Raycast hitbox is unobstructed, allowing it to be clicked on.
Actual Result	Every button passed this test, except for the Close button for the Help instructions on the Main Menu.
Reason	The Close button's Raycast hitbox was obstructed by the help text, which still had its Raycast hitbox enabled.
Changes Made	Disabled the help text's Raycast hitbox, allowing the Close button to be clicked on.

Functionality Test 2	Test if the Song List displays only valid files.
Input	Include an empty TestFolder in the Songs folder.
Expected Result	Song List scans TestFolder and finds it lacks the correct files. Song List ignores TestFolder and does not display it.
Actual Result	Song List scans TestFolder and finds it lacks the correct files. Song List ignores TestFolder and does not display it.
Reason	Everything working as intended. TestFolder lacked a song.ini, notes.mid and a guitar.ogg file.
Changes Made	No change necessary.

Functionality Test 3	Test if the Song List displays only valid files.
Input	Include a variety of song files to test in Songs folder.
Expected Result	Song List scans each song file for the correct files. Song List includes the songs if they are all valid.
Actual Result	42 song files inputted, 42 song files displayed in the Song List, all with the valid files present.
Reason	Everything working as intended.
Changes Made	No change necessary.

Functionality Test 4	Load Queen – Bohemian Rhapsody into the Guitar Tutor.
Input	Select Queen – Bohemian Rhapsody from the Song List.
Expected Result	Set artist as 'Queen', set song as 'Bohemian Rhapsody'. Load the song layers to split. Load the hand animations.
Actual Result	Set artist as 'Bohemian Rhapsody', set artist as ' '. Song layers loaded correctly. Hand animations loaded correctly, however upon playing the song the animator would play some of the piano parts as well as the guitar parts.
Reason	The song.ini file had a blank on its first line, shifting all of the values below the expected location. The issue with the animator, after examination, is down to how the MIDI file has been formatted, with no differentiating with piano keys and guitar notes, and thus not a problem with how AnimateHands.cs and MidiReader.cs function.
Changes Made	Updated MusicLayers.cs to scan file for the words 'name' and 'artists' in every line, rather than assuming all .ini files shared the same format as the sample song. No change was deemed necessary for the hand animator.

When testing the song Bohemian Rhapsody, it was found that during some parts of the song, the animation handler would instruct the guitar to play for certain piano parts of the song, when there was no guitar playing at that moment. Initially, this was believed to be a fault with the way the virtual tutor read the MIDI file, in that it was reading the incorrect track.

However, upon investigation, it was discovered that this is simply how the song had been formatted in Clone Hero – the guitar.ogg sound file at certain times plays the piano music if there is no guitar to play, and when the guitar is playing the piano music is shifted instead over to song.ogg. Additionally, there are many piano-central songs that have guitar tablature translations available, of which Bohemian Rhapsody is an example.

Thus, it was determined to not be an issue that needed resolving, as this could be a feature to include when the virtual tutor's own guitar tab display is implemented.

Functionality Test 5	Load The Killers – When You Were Young into the Guitar Tutor
Input	Select The Killers – When You Were Young from the Song List.
Expected Result	Set artist as 'The Killers', set song as 'When You Were Young'. Load the song layers to split. Load the hand animations.
Actual Result	Midi File loaded correctly, but the song layers were not, causing the tutor to halt.
Reason	The initial track for the MIDI file was untitled, so any checks to compare the track's name yielded null, preventing MidiReader.cs finding the guitar layer.
Changes Made	Updated MidiReader.cs to ignore any null tracks until it found the guitar track.

Functionality Test 6	Load Snow Patrol – Hands Open into the Guitar Tutor
Input	Select Snow Patrol – Hands Open from the Song List.
Expected Result	Set artist as 'Snow Patrol', set song as 'Hands Open'. Load the song layers to split. Load the hand animations.
Actual Result	Set artist as 'Error Name', set artist as 'Error Name'. Song layers loaded correctly. Hand animations loaded correctly.
Reason	The sample song's song.ini file was listed as "name = songname", whereas this song's song.ini file was listed as "name=songname". The check to find the correct line for the name and artist ended up skipping over them as a result, displaying the default error message.
Changes Made	Updated MusicLayers.cs with an extra if clause to check for '=' with or without spaces surrounding it.

Functionality Test 7	Load Nirvana - Smells Like Teen Spirit into the Guitar Tutor
Input	Select Nirvana - Smells Like Teen Spirit from the Song List.
Expected Result	Set artist as 'Nirvana', set song as 'Smells Like Teen Spirit'. Load the song layers to split. Load the hand animations.
Actual Result	Set artist as 'Nirvana', set song as 'Smells Like Teen Spirit'. Song layers loaded correctly. Hand animations loaded correctly.
Reason	Everything working as intended.
Changes Made	No change necessary.

VI: Conclusion And Evaluation

This chapter details an evaluation on the overall effectiveness of the finished project by comparing it against aims and objectives set out in Chapter I, and the original criteria and requirements set out in Chapter III. It also details the potential future of the project, as well as the unfortunate impact of COVID-19 and the global lockdown upon the project.

VI.1) Initial Planning Review

VI.1a) Aims Review

- The final product is a software platform that makes use of motion captured hand animations for teaching its users how to play songs by demonstrating the necessary hand movements required to play certain chords in the song loaded.
- The project was an opportunity to learn about motion capture, such as its ability to create smooth, realistic animations that are natural in appearance, and its unfortunate overreliance in specific hardware and software in order to be properly applied.
- Though the overall effectiveness of the teaching provided by the program due to its animations is questionable, at the very least it has cemented the foundations required for a higher quality teaching tool to be created as a potential successor.

VI.1b) Objectives Review

- The finished platform renders each guitar chord animation in fully rotatable 3D space, and plays each animation in order, step-by-step, with numerous customisation options to suit the user's comfort, as initially put forward.
- Due to the global pandemic preventing any more animations from being recorded, the library of animations available is limited in quantity, and thus do not cover nearly as many chord shapes as desired. However, the animations do smoothly transition into one another, as required.
- Again, due to the pandemic, it cannot be truly said that the project is openly accessible for users without full use of both hands. At best it can accommodate for users who have full usage of one hand but not of the other, such as myself, so there is still a degree of accessibility available.

VI.1c) Deliverables Review

- The interactive program allows the user to learn the guitar while tailoring its functions to their comfort, using a small number of available 3D motion captured animations to teach them how to play.

- As the program's core structure shares many similarities with that of Clone Hero, most if not all songs that are compatible with Clone Hero are compatible with this project's program. So long as there is a MIDI file, a song.ini file and a guitar.ogg file, any song folder can be loaded.
- As a result, there is no need to generate any lesson files. The only files that are generated are the checkpoint time .txt files used when saving and loading checkpoints.
- Sadly, due to the COVID-19 pandemic, as most of the development time was spent ensuring the animations available worked well enough, there was not enough time to properly implement a score sheet database to go alongside with the animation database, nor was there enough time to properly create, read and display a guitar tablature sheet in the program.
- An overview of the testing strategies and bug reports are included in Chapter V, with a greater detail of more test reports included within Appendix A.D.

VI.2) Requirements Review

VI.2a) MoSCoW Criteria Review

The table below lists all of the MoSCoW Criteria requirements detailed in Chapter III, and whether that goal is considered to have been achieved or not. If a goal is marked with an asterisk, then it is considered to have been disrupted as a result of the COVID-19 pandemic affecting study at UCL.

VI.2b) Requirements Evaluations

The final program for this project managed to achieve all of the agreed Must Have and Should Have requirements, and although not all of the Could Have requirements were met, many of them were achieved also.

For the system requirements, the finished product is a fully 3D virtual guitar tutor constructed in Unity C# that utilises motion captured animations to teach the user the necessary hand movements – albeit of limited quality due to the pandemic. It can accommodate for both left-handed and right-handed users, however it does not have the full accessibility for any user with upper- or lower-limb deficiencies as anticipated, again due to the pandemic lockdown.

Though the amount of animations available is restricted, they all smoothly transition between one another and can be viewed in fully rotatable three dimensions. As the MIDI file is read at the beginning of the scene once it finishes loading, this ensures that the animations will be identical every single playthrough, ensuring consistency, and the chord positions are examined across the guitar's entire fretboard to try and limit unnecessary hand movements.

ID	Description	Priority	Achieved?
System Overview			
S1	The system will provide a teaching platform for playing the guitar.	Must have	Yes
S2	The system will use lifelike 3D animations to teach the user in real time.	Must have	Yes*
S3	The system will be programmed in C#, using the Unity engine.	Should have	Yes
S4	The system will cater to both left-handed and right-handed users.	Should have	Yes
S5	The system will cater to users with upper- or lower-limb deficiencies.	Could have	No*
S6	The system will have online functionalities.	Won't have	No
S7	The system will be available on iPhone and Android phones.	Won't have	No
S8	The system will provide animations and tutorials for learning other instruments, such as the piano or violin.	Won't have	No
Animations			
A1	The animations will transition fluidly and smoothly between each other.	Must have	Yes
A2	The animations will be completely rotatable in 3D space.	Must have	Yes
A3	The animations will remain constant every time the same song is played.	Must have	Yes
A4	The animations will adapt, depending upon the chord positions, to minimise the required hand movements.	Should have	Yes*
A5	The animations will be compatible for people without full usage of both hands (missing hand, loss of finger mobility, etc.)	Could have	No*
A6	The animations will be compatible for people without full usage of either hands (i.e. there are problems with both hands)	Could have	No
User Interface			
U1	The user interface will be easy to use and understand.	Must have	Yes
U2	The user interface will allow the user to pause, resume, rewind and fast-forward the song.	Should have	Yes
U3	The user interface will allow the user to easily skip between parts of the song.	Should have	Yes
U4	The user interface will allow the user to create certain loops in the song to better practise certain parts.	Should have	Yes
U5	The user interface will be professionally-made.	Could have	No
U6	The user interface will have multiple options for languages.	Won't have	No
Customisation			
C1	The user can adjust the tempo of the music and speed of the animations to practise at a more comfortable level.	Should have	Yes
C2	The user can import music files (.mp3, .wav, etc.) and have the system teach them songs of their own choice, provided there is a matching music sheet uploaded.	Could have	Yes
C3	The user can create their own music sheets for the system to teach them, much like similar free programs (MuseScore, PowerTab, and more primitive software such as Mario Paint Composer).	Could have	No
C4	The user can scan in and import music sheets, which the system will convert into animations.	Could have	No
C5	The user can edit any imported music sheets, in case of errors that might have been created from conversion.	Could have	No
Miscellaneous			
M1	The final version will be free to distribute and download.	Must have	Yes
M2	The final version will have a wide variety of demo songs available to test with.	Could have	Yes
M3	The final version will make you a pro at Guitar Hero.	Won't have	No

For the user interface, the main UI is simple enough to understand what everything achieves, and the user has the option to freely pause, play, restart, skip ahead or skip back the song, as well as enable/disable specific song layers and even set, save and load checkpoints or set loops within the song. However, its simplicity means it looks rather basic compared to some higher market products available, and currently only an English version is available. Additional languages and improvements to the UI would be something to be considered for future developments for the project.

The user can also adjust the tempo of the song and animations to practise them more precisely, as well as import any other song files that are compatible with Clone Hero to work with the program, including a vast array of demo songs included in the final program as examples of its workings. However, as development of the hand animations was prioritised over reading music sheets, there was no time available for implementing anything relating to a guitar tablature displayer, hence the exclusion of any work relating to music sheets. As a result, custom songs can be implemented without the need for a relevant guitar tab sheet.

Also, at the time of writing this report, the project has not been publicly released, but should the time come, it will be freely downloadable and distributed without cost.

VI.3) Future Of The Project

Due to the complications involved with the university being locked down and access to the virtual reality labs being denied as a result, the quality of this project would definitely improve if additional time, effort and resources were devoted to it. Below are a few examples of what could be done to improve the project in an additional six months and why they would be important:

- Record and implement higher-quality motion capture animations, and in greater number.
    - Because of the COVID-19 global pandemic and the subsequent lockdown, the animations in the final build of the project are not as informative as they could be. Further development would allow for more precise finger positionings, as well as ensure that the hand model does not clip through the guitar model, improving the quality of the teaching.
    - This would also allow for more animations to accommodate for users with upper-limb deficiencies or lack of full hand movements, improving upon its accessibility.
- Implement proper separate hand models to eliminate the need for render culling in Unity.
    - Presently, the spare hand is culled by using a depth mask to cover it; however, as the whole arms model uses the same

rendering shader, this means that if the hands are viewed at a specific angle then it can accidentally delete both hands instead of just the unnecessary hand. Having individual models with working animation skeleton rigs would negate this issue, as well as provide more realistic, human-like hand models rather than the default bright red mannequin models used currently.

- Refine the algorithm used for determining the necessary fret and chord selection to improve its accuracy.
    - With access to more animations, this would mean that there would be more chances for precise hand selections, rather than having to cut corners due to limited animations and output movements that are not as accurate as they could be.
- Implement a guitar tab reader and projector so that the user has an additional teaching source to instruct them how to play the song.
    - A lot of the development time for this project went into ensuring that the hand animations and fret positions were correct, leaving very little time to implement some of the lower priority requirements such as a guitar tab reader. As tablature is the more traditional method for learning how to play a new song, combining it with animation teaching would greatly enhance the learning quality offered by the program.
- Implement the option to teach the user other types of guitars that are present within the song, such as bass or rhythm.
    - Currently the program only allows for teaching the lead guitar, however Clone Hero has the option to let the user play the notes for a secondary guitar, if one exists. This would allow for the project to accommodate for musicians who use any of those guitar types, instead of just the lead guitar.
- Improve loading speeds between scenes.
    - Although the program works as intended, depending upon the processing power of the computer running it can take several minutes to change scenes between SongList and GuitarTutor. If the program were to have a public release, the time taken to load everything up would need to be drastically decreased.
- Refine the user interface so that it looks more professional.
    - The current UI is simple enough for the user to understand and follow, however it is rather bland in appearance. A public release would require the UI to look more interesting whilst still keeping the clarity – for example, interactive loading screens to show the loading progress.
- Implement a translation for other languages.
    - At the bare minimum, including the option for other core European languages (French, German, Spanish, etc.) would improve the accessibility of the program, however the UI is

simple enough that instructions should be able to transcend language barriers.

- Implement a port for Android/iOS.
    - Not everyone who practises the guitar will be doing so in front of a computer screen, but some might have a phone available instead. This would improve the accessibility of the program, allowing anyone to practise wherever they are. Songsterr also has a mobile app equivalent, so a mobile app for this program would put it on par with its rival.

VI.4) Final Thoughts

This project has had a lot of ups and downs, with many unforeseen difficulties impacting it. Issues such as the availability of the motion capture technology at UCL, and the difficulties in requiring a Unity workaround to record the animations, not to mention everything related to COVID-19, all confounded the project, resulted in the test animations becoming the final animations used.

Nevertheless, despite these shortcomings, I strongly believe that this project was a success, as well as being very enjoyable to work on. The final program is a really useful tool for students learning the guitar with its real-time animation display, and the option to adjust the tempo of the song to slow down the animations, set checkpoints and looping, and enable/disable specific layers of the song all add to its flexibility in teaching.

Additionally, the last-minute compatibility with Clone Hero allows the program to automatically render animations for almost any song that the user inputs into the virtual tutor, giving the program access to a vast library of premade songs already available on the internet. This means that the user will always have at least one song available for them to learn from.

"It's something unpredictable, but in the end it's right.
I hope you had the time of your life."

- Good Riddance (Time Of Your Life), Green Day

Appendix

This chapter contains various miscellaneous items of interest relevant to the project as a whole, such as the bibliographical citations, the code listings and the full lists of use cases and test cases, to save room in their respective chapters.

A.A) Systems Manual

To install the project, download the .zip file from the Google Drive link here:

https://drive.google.com/file/d/1TGzfsoamNSOu1cCR5fcS-ySTDh735HH7/view?usp=sharing

The .zip file contains the project ready for use in Unity. To open it in the editor, select 'Add Project' in the Unity Hub menu and then load it by selecting it from the list.

Everything related to the project is stored within the Assets folder, with each subfolder containing the organised files. The important folders for consideration are Animations, Prefabs, Scenes, Scripts and Songs

As Unity is self-contained, all of the code is automatically compiled when it is saved in Visual Studio, and is executed when the game scene is run, so long as there is a game object present in the scene that holds the script file component.

To add/remove any additional game scenes, click on 'File -> Build Settings…' from the toolbar, and then either drag the desired game scene into the 'Scenes In Build' list to add, or right click a scene in the list and select 'Remove Selection'.

To export the finished project as a .exe file, select Build from the Build Settings list. This builds the project and publishes it as a single file. Alternatively, Build and Run publishes the .exe file and then immediately runs it for testing.

A.B) User Manual

The following instructions are included on the Help option in the main menu, and included as part of a readme.txt file included within the project.

"Welcome to StringStar, a free virtual guitar tutor!

StringStar uses motion captured hand animations to recreate the hand positions required to play guitar notes and chords, visualising the movements needed to learn how to play a song.

To install a song to play, copy the song file into the Songs folder. The song must have a song.ini file, a guitar.ogg file and a notes.mid with a guitar track for the tutor to play it. StringStar is compatible with Clone Hero song files, so any song that can be played in Clone Hero can be played in StringStar.

Controls:

WASD / Arrow keys: moves the guitar up, down, left and right.

Q / comma key: move the guitar closer to the camera.

E / full stop key: Move the guitar further from the camera.

Left Click on the guitar and drag the mouse to rotate it.

Right Click to reset the guitar to its original position.

Play/Pause button: starts/stops the song.

Reset Button: Sets the song back to the start and resets the guitar's position.

Layer Button: Individually enable/disable specific layers of the song.

Loop Button: Enable song looping, select which checkpoints to loop between, and save/load checkpoints.

Left Click on the Progress Bar to skip forwards or backwards within a song.

Middle Click on the Progress Bar to create a checkpoint to easily skip back to that point in the song.

Left Click on the Tempo bar to increase/decrease the tempo of the song, to make it easier to follow the hand animations."

A.C) Use Cases

ID	Use Case
UC1	Start
UC2.1	Pause Song
UC2.2	Resume Song
UC2.3	Skip Part
UC2.4	Create Checkpoint
UC2.5	Delete Checkpoint
UC2.6	Save Checkpoints
UC2.7	Loads Checkpoints
UC2.8	Set Loop
UC2.9	Adjust Tempo
UC2.10	Stop Lesson
UC2.11	Change Lesson
UC2.12	Rotate Animation
UC3.1	Import Song
UC3.2	Delete Song
UC4	Quit

Use Case	Start
ID	UC1
Brief Description	The user begins their guitar lesson with their chosen song and its animations.
Primary Actors	User
Secondary Actors	Song Database
Pre-conditions	The user must be on the main menu; the user must select a valid song when prompted from the song list.
Main Flow	1) The user selects Start from the main menu and is taken to the song list. 2) The user selects a valid song to practise from the song list. 3) On-screen instructions will appear for the user to follow, as the song database loads up the chosen song. 4) The user clicks the play button on the tutor screen.
Post-conditions	The virtual tutor begins, as the song and animations count the player in.

Use Case	Pause Song
ID	UC2.1
Brief Description	The user pauses the song and its animation.
Primary Actors	User
Secondary Actors	None
Pre-conditions	The user must be on the tutor screen; the song must be playing.
Main Flow	1) The user clicks the Pause button.
Post-conditions	The song and animations pause.

Use Case	Resume Song
ID	UC2.2
Brief Description	The user resumes the song and its animation after it had been paused.
Primary Actors	User
Secondary Actors	None
Pre-conditions	The user must be on the tutor screen; the song must have been paused.
Main Flow	1) The user clicks the Pause button.
Post-conditions	The song and animations unpause and resume.

Use Case	Skip Part
ID	UC2.3
Brief Description	The user skips ahead or behind in a song, in order to practise a specific part.
Primary Actors	User
Secondary Actors	None
Pre-conditions	The user must be in the tutor screen.
Main Flow	1) The user clicks on the song progression bar.
Post-conditions	The song skips to the time indicated on the progression bar.

Use Case	Create Checkpoint
ID	UC2.4
Brief Description	The user marks a specific point in the song.
Primary Actors	User
Secondary Actors	None
Pre-conditions	The user must be in the tutor screen.
Main Flow	1) The user middle-clicks on the song progression bar.
Post-conditions	A marked checkpoint appears on the progression bar, allowing the user to easily identify an important part in the song.

Use Case	Delete Checkpoint
ID	UC2.5
Brief Description	The user removes a previously-created checkpoint.
Primary Actors	User
Secondary Actors	Song Database
Pre-conditions	The user must be in the tutor screen; there must be at least one checkpoint created.
Main Flow	1) The user middle-clicks on a checkpoint.
Post-conditions	The marked checkpoint is removed from the progression bar.

Use Case	Save Checkpoints
ID	UC2.6
Brief Description	The user saves all of the current checkpoints.
Primary Actors	User
Secondary Actors	Song Database
Pre-conditions	The user must be in the tutor screen; there must be at least one checkpoint created.
Main Flow	1) The user clicks on the save checkpoint button.
Post-conditions	The checkpoints are saved in the song database.

Use Case	Load Checkpoints
ID	UC2.7
Brief Description	The user loads previously saved checkpoints.
Primary Actors	User
Secondary Actors	Song Database
Pre-conditions	The user must be in the tutor screen; there must be previously-saved checkpoints.
Main Flow	1) The user clicks on the load checkpoint button.
Post-conditions	The saved checkpoints are loaded and displayed.

Use Case	Set Loop
ID	UC2.8
Brief Description	The user sets two checkpoints for the lesson to repeatedly loop between.
Primary Actors	User
Secondary Actors	Song Database
Pre-conditions	The user must be in the tutor screen
Main Flow	1) The user clicks on the Loop button. 2) The user selects two checkpoints for the song to loop between – one as the starting point, one as the ending point (by default, if no checkpoints exist, this will be the start and end of the lesson).
Post-conditions	The song will loop back to start at the first checkpoint once it progresses past the second checkpoint.

Use Case	Adjust Tempo
ID	UC2.9
Brief Description	The user adjusts how fast or slow the song and animations play to practise better.
Primary Actors	User
Secondary Actors	None
Pre-conditions	The user must be on in the tutor screen
Main Flow	1) The user clicks on the tempo slider (by default, the song cannot exceed a tempo of 100% of its original value). 2) The percentage value associated with the tempo slider updates to match.
Post-conditions	The lesson will slow down or speed up to match the user's input value.

Use Case	Stop Lesson
ID	UC2.10
Brief Description	The user stops the lesson and returns to the song selection menu.
Primary Actors	User
Secondary Actors	Song Database
Pre-conditions	The user must be in the tutor screen.
Main Flow	1) The user clicks on the Exit button, and the song will end.
Post-conditions	The song will end, and the user will be taken back to the song selection menu.

Use Case	Change Lesson
ID	UC2.11
Brief Description	The user selects a new song to learn.
Primary Actors	User
Secondary Actors	Lesson Database
Pre-conditions	The user must be in the tutor screen; the user must select a valid song when prompted from the song list.
Main Flow	1) The user clicks on the Exit button, and the song will end. 2) The user is taken back to the song selection menu. 3) The user selects a new song to learn from the tutor and clicks the play button. 4) If any checkpoints were previously created for the lesson, they are also loaded.
Post-conditions	The current song will close, and the user will be taken to the next song.

Use Case	Rotate Animation
ID	UC2.12
Brief Description	The user rotates the animation.
Primary Actors	User
Secondary Actors	None
Pre-conditions	The user must be in the tutor screen.
Main Flow	1) The user clicks and drags the mouse on the guitar model. 2) The user will be able to freely adjust the angle and position of the guitar model to move the animations closer or further away from the camera. 3) The user releases the left mouse button to stop moving the animation.
Post-conditions	The angle and position of the guitar model and hand animations are adjusted.

Use Case	Import Song
ID	UC3.1
Brief Description	The user imports a song file from their desktop into the program's database.
Primary Actors	User
Secondary Actors	Song Database
Pre-conditions	The user must be outside the application to copy the file.
Main Flow	1) The user selects the compatible song file they would like to import. 2) The user copies the song file into the application's Songs folder. 3) The user then goes into the application's main menu and follows the instructions to the song selection list screen.
Post-conditions	The selected song file(s) are stored into the song database and made available to play.

Use Case	Delete Song
ID	UC3.2
Brief Description	The user removes the song file from the song database.
Primary Actors	User
Secondary Actors	Song Database
Pre-conditions	The user must be outside the application.
Main Flow	1) The user goes to the application's Songs folder. 2) The user selects which song file they would like to remove. 3) The user takes the song file out of the Songs folder. 4) When the user returns to the song selection list, the song in question will no longer be available to load.
Post-conditions	The selected song file(s) are deleted from the song database.

Use Case	**Quit**
ID	UC4
Brief Description	The user quits the software.
Primary Actors	User
Secondary Actors	None
Pre-conditions	The user must be on the main menu.
Main Flow	1) The user selects Quit from the main menu. 2) The user is prompted if they want to quit. If they select No, they are returned to the main menu. If they select Yes, the software terminates.
Post-conditions	The software closes.

A.D) Test Cases

Manual Unit Tests

Unit Test 1	Test Play Button plays the song when paused, and pauses the song when playing.
Input	Sample Song set to Green Day – American Idiot. Click on the Play Button.
Expected Result	PlaySong pauses the song if it is playing, and plays the song if it is paused.
Actual Result	PlaySong pauses the song if it is playing, and plays the song if it is paused.
Reason	Everything working as intended.
Changes Made	No change necessary.

Unit Test 2	Test Reset Button restarts the song.
Input	Sample Song set to Green Day – American Idiot. Click on the Reset Button.
Expected Result	PlaySong resets the song back to the start. All of the animations stop and return to their original position. The guitar returns to its original position and rotation.
Actual Result	PlaySong resets the song back to the start. All of the animations stop and return to their original position. The guitar returns to its original position and rotation.
Reason	Everything working as intended.
Changes Made	No change necessary.

Unit Test 3	Test Layers Button brings up the Layers UI.
Input	Sample Song set to Green Day – American Idiot. Click on the Layers Button.
Expected Result	Layers UI displays.
Actual Result	Layers UI displays.
Reason	Everything working as intended.
Changes Made	No change necessary.

Unit Test 4	Test Layers UI can enable/disable individual song layers.
Input	Sample Song set to Green Day – American Idiot. Click on the Layers Button to bring up the UI. Click on an individual layer.
Expected Result	Layer disables if enabled and text updates. Layer enables if disabled and text updates.
Actual Result	Layer disables if enabled and text updates. Layer enables if disabled and text updates.
Reason	Everything working as intended.
Changes Made	No change necessary.

Unit Test 5	Test Loop Button brings up the Looping UI.
Input	Sample Song set to Green Day – American Idiot. Click on the Loop Button to bring up the UI.
Expected Result	Looping UI displays.
Actual Result	Looping UI displays.
Reason	Everything working as intended.
Changes Made	No change necessary.

Unit Test 6	Test song loops when commanded up Looping UI.
Input	Sample Song set to Green Day – American Idiot. Click on the Loop Button to bring up the UI. Click on Enable Looping from the UI.
Expected Result	Song loops between the checkpoints set if enabled. Song stops looping if disabled. If no checkpoints are set to loop between, song loops between whole song.
Actual Result	Song loops between the checkpoints set if enabled. Song stops looping if disabled. If no checkpoints are set to loop between, song loops between whole song.
Reason	Everything working as intended.
Changes Made	No change necessary.

Unit Test 7	Test Progress Bar updates the Song Tracker as the song plays.
Input	Sample Song set to Green Day – American Idiot. Song is playing.
Expected Result	Song time text is updated as the song plays. Song Tracker game object's position is updated.
Actual Result	Song time text is updated as the song plays. Song Tracker game object remained stationary.
Reason	Accidentally set Song Tracker's original position to be constantly updated, rather than its current position, causing the object to remain inanimate.
Changes Made	Changed what position was checked to update the Song Tracker.

Unit Test 8	Test Progress Bar updates the Mouse Tracker as the user moves the cursor across it.
Input	Sample Song set to Green Day – American Idiot. Mouse is moved across the Progress Bar.
Expected Result	Mouse Tracker position updates when the mouse moves over it. Mouse Tracker disappears when the mouse is moved out of it.
Actual Result	Mouse Tracker position updates when the mouse moves over it. Mouse Tracker disappears when the mouse is moved out of it.
Reason	Everything working as intended.
Changes Made	No change necessary.

Unit Test 9	Test Tempo Bar updates the song's tempo for all layers when clicked.
Input	Sample Song set to Green Day – American Idiot. Mouse clicks on the Tempo Bar.
Expected Result	The Audio Mixer changes the tempo based upon the position clicked, slowing down/speeding up the song as required.
Actual Result	The Audio Mixer changes the tempo based upon the position clicked, slowing down/speeding up the song as required.
Reason	Everything working as intended.
Changes Made	No change necessary.

Unit Test 10	Test guitar model rotates when clicked on.
Input	Mouse clicks and drags over the guitar model.
Expected Result	Guitar model and the playing hands models rotate as the mouse moves.
Actual Result	Guitar model and the playing hands models rotate as the mouse moves.
Reason	Everything working as intended.
Changes Made	No change necessary.

Unit Test 11	Test hand models are culled to eliminate the excess hand.
Input	Mouse clicks and drags over the guitar model to rotate it.
Expected Result	Spare hand within the depth mask is not rendered, causing only one hand from the mesh to be displayed.
Actual Result	Spare hand is correctly culled, however depending upon the angle the guitar is rotated both hand models may not be rendered.
Reason	This is a shortcoming caused by being unable to get a better hand model due to the recorded animations following the COVID-19 pandemic lockdown.
Changes Made	No change necessary; if better animations for a model of only one hand were available, this would be changed.

Unit Test 12	Test hand animations move the hand models to the right fret.
Input	Sample Song set to Green Day – American Idiot. Song is playing.
Expected Result	AnmiationHandler moves the hands to the desired fret.
Actual Result	AnimationHandler moves the hands to less optimal fret.
Reason	AnimationHandler simply chose the smallest fret that gave the correct note available on another string, rather than accepting the fret on the current string if it minimises movement to relating chords.
Changes Made	Changed the code so that it checks whether the current fret/string combination is a sensible solution, favouring the lower, deeper strings with a higher fret value over a higher string with a lower fret value.

Unit Test 13	Test whether a valid song list is displayed in the Song List.
Input	Sample Song set to Green Day – American Idiot.
Expected Result	Song List loads Green Day – American Idiot from the Songs folder and displays it in the list.
Actual Result	Song List loads Green Day – American Idiot from the Songs folder and displays it in the list.
Reason	Everything working as intended as the valid files necessary for the virtual tutor to function were detected.
Changes Made	No change necessary.

Unit Test 14	Test whether the next scenes load when clicked.
Input	Click on the Play/Exit button on each scene to load the next scene.
Expected Result	Play Button on the Main Menu loads the Song List. Play Button on the Song List loads the Loading Scene, which automatically loads the Guitar Tutor. Exit Button on the Guitar Tutor loads the Song List. Exit Button on the Song List loads the Main Menu. Quit Button on the Main Menu produces a prompt, which closes the application.
Actual Result	Play Button on the Main Menu loads the Song List. Play Button on the Song List loads the Loading Scene, which automatically loads the Guitar Tutor. Exit Button on the Guitar Tutor loads the Song List. Exit Button on the Song List loads the Main Menu. Quit Button on the Main Menu produces a prompt, which did not close the application.
Reason	Unity will not close the application if the program has not been fully built – as this was still in the editor, the Quit Button from the Main Menu did not close the program, but worked as intended otherwise.
Changes Made	No change necessary.

Functionality Tests

Functionality Test 1	Test all of the buttons in the entire program can be clicked.
Input	Click every button in the program to ensure they function correctly.
Expected Result	Every button's Raycast hitbox is unobstructed, allowing it to be clicked on.
Actual Result	Every button passed this test, except for the Close button for the Help instructions on the Main Menu.
Reason	The Close button's Raycast hitbox was obstructed by the help text, which still had its Raycast hitbox enabled.
Changes Made	Disabled the help text's Raycast hitbox, allowing the Close button to be clicked on.

Functionality Test 2	Test if the Song List displays only valid files.
Input	Include an empty TestFolder in the Songs folder.
Expected Result	Song List scans TestFolder and finds it lacks the correct files. Song List ignores TestFolder and does not display it.
Actual Result	Song List scans TestFolder and finds it lacks the correct files. Song List ignores TestFolder and does not display it.
Reason	Everything working as intended. TestFolder lacked a song.ini, notes.mid and a guitar.ogg file.
Changes Made	No change necessary.

Functionality Test 3	Test if the Song List displays only valid files.
Input	Include a variety of song files to test in Songs folder.
Expected Result	Song List scans each song file for the correct files. Song List includes the songs if they are all valid.
Actual Result	42 song files inputted, 42 song files displayed in the Song List, all with the valid files present.
Reason	Everything working as intended.
Changes Made	No change necessary.

Functionality Test 4	Load Queen – Bohemian Rhapsody into the Guitar Tutor.
Input	Select Queen – Bohemian Rhapsody from the Song List.
Expected Result	Set artist as 'Queen', set song as 'Bohemian Rhapsody'. Load the song layers to split. Load the hand animations.
Actual Result	Set artist as 'Bohemian Rhapsody', set artist as ' '. Song layers loaded correctly. Hand animations loaded correctly, however upon playing the song the animator would play some of the piano parts as well as the guitar parts.
Reason	The song.ini file had a blank on its first line, shifting all of the values below the expected location. The issue with the animator, after examination, is down to how the MIDI file has been formatted, with no differentiating with piano keys and guitar notes, and thus not a problem with how AnimateHands.cs and MidiReader.cs function.
Changes Made	Updated MusicLayers.cs to scan file for the words 'name' and 'artists' in every line, rather than assuming all .ini files shared the same format as the sample song. No change was deemed necessary for the hand animator.

Functionality Test 5	Load The Killers – When You Were Young into the Guitar Tutor.
Input	Select The Killers – When You Were Young from the Song List.
Expected Result	Set artist as 'The Killers', set song as 'When You Were Young'. Load the song layers to split. Load the hand animations.
Actual Result	Midi File loaded correctly, but the song layers were not, causing the tutor to halt.
Reason	The initial track for the MIDI file was untitled, so any checks to compare the track's name yielded null, preventing MidiReader.cs finding the guitar layer.
Changes Made	Updated MidiReader.cs to ignore any null tracks until it found the guitar track.

Functionality Test 6	Load Snow Patrol – Hands Open into the Guitar Tutor.
Input	Select Snow Patrol – Hands Open from the Song List.
Expected Result	Set artist as 'Snow Patrol', set song as 'Hands Open'. Load the song layers to split. Load the hand animations.
Actual Result	Set artist as 'Error Name', set artist as 'Error Name'. Song layers loaded correctly. Hand animations loaded correctly.
Reason	The sample song's song.ini file was listed as "name = songname", whereas this song's song.ini file was listed as "name=songname". The check to find the correct line for the name and artist ended up skipping over them as a result, displaying the default error message.
Changes Made	Updated MusicLayers.cs with an extra if clause to check for '=' with or without spaces surrounding it.

Functionality Test 7	Load Nirvana - Smells Like Teen Spirit into the Guitar Tutor.
Input	Select Nirvana - Smells Like Teen Spirit from the Song List.
Expected Result	Set artist as 'Nirvana', set song as 'Smells Like Teen Spirit'. Load the song layers to split. Load the hand animations.
Actual Result	Set artist as 'Nirvana', set song as 'Smells Like Teen Spirit'. Song layers loaded correctly. Hand animations loaded correctly.
Reason	Everything working as intended.
Changes Made	No change necessary.

Functionality Test 8	Load Green Day - Know Your Enemy into the Guitar Tutor.
Input	Select Green Day - Know Your Enemy from the Song List.
Expected Result	Set artist as 'Green Day', set song as 'Know Your Enemy'. Load the song layers to split. Load the hand animations.
Actual Result	Set artist as 'Green Day', set song as 'Know Your Enemy'. Song layers loaded correctly. Hand animations loaded correctly.
Reason	Everything working as intended.
Changes Made	No change necessary.

Functionality Test 9	Load Green Day – Boulevard Of Broken Dreams into the Guitar Tutor.
Input	Select Green Day – Boulevard Of Broken Dreams from the Song List.
Expected Result	Set artist as 'Green Day', set song as 'Boulevard Of Broken Dreams'. Load the song layers to split. Load the hand animations.
Actual Result	Set artist as 'Green Day', set song as 'Boulevard Of Broken Dreams'. Song layers loaded correctly. Hand animations loaded correctly.
Reason	Everything working as intended.
Changes Made	No change necessary.

Functionality Test 10	Load Muse - Knights of Cydonia into the Guitar Tutor.
Input	Select Muse - Knights of Cydonia from the Song List.
Expected Result	Set artist as 'Muse', set song as 'Knights of Cydonia'. Load the song layers to split. Load the hand animations.
Actual Result	Set artist as 'Muse', set song as 'Knights of Cydonia'. Song layers loaded correctly. Hand animations loaded correctly.
Reason	Everything working as intended.
Changes Made	No change necessary.

Functionality Test 11	Load Alice Cooper - School's Out into the Guitar Tutor.
Input	Select Alice Cooper - School's Out from the Song List.
Expected Result	Set artist as 'Alice Cooper', set song as 'School's Out'. Load the song layers to split. Load the hand animations.
Actual Result	Set artist as 'Alice Cooper', set song as 'School's Out'. Song layers loaded correctly. Hand animations loaded correctly.
Reason	Everything working as intended.
Changes Made	No change necessary.

Functionality Test 12	Load Deep Purple – Smoke On The Water into the Guitar Tutor.
Input	Select Deep Purple – Smoke On The Water from the Song List.
Expected Result	Set artist as 'Deep Purple', set song as 'Smoke On The Water'. Load the song layers to split. Load the hand animations.
Actual Result	Set artist as 'Deep Purple', set song as 'Smoke On The Water'. Song layers loaded correctly. Hand animations loaded correctly.
Reason	Everything working as intended.
Changes Made	No change necessary.

Functionality Test 13	Load Kaiser Chiefs – Ruby into the Guitar Tutor.
Input	Select Kaiser Chiefs – Ruby from the Song List.
Expected Result	Set artist as 'Kaiser Chiefs', set song as 'Ruby'. Load the song layers to split. Load the hand animations.
Actual Result	Set artist as 'Kaiser Chiefs', set song as 'Ruby'. Song layers loaded correctly. Hand animations loaded correctly.
Reason	Everything working as intended.
Changes Made	No change necessary.

Functionality Test 14	Load Green Day – American Idiot into the Guitar Tutor.
Input	Select Green Day – American Idiot from the Song List.
Expected Result	Set artist as 'Green Day', set song as 'American Idiot'. Load the song layers to split. Load the hand animations.
Actual Result	Set artist as 'Green Day', set song as 'American Idiot'. Song layers loaded correctly. Hand animations loaded correctly.
Reason	Everything working as intended.
Changes Made	No change necessary (especially as this was the sample song).

A.E) Project Plan

The Project Plan was originally submitted Tuesday 12th November 2019, and has been included in this report for completion's sake.

Generating 3D Hand Animations For Virtual Guitar Lessons

Toby Best

MEng Mathematical Computation

Supervisor:

Yuzuko Nakamura

Aims:

- To investigate and learn more about motion capture and its potential applications, particularly in a learning environment.
    - For the most part as of recent, motion capture has predominantly been used for entertainment purposes, such as within video games or computer-generated imagery in films; I would like to explore its potential in a more day-to-day case.
- Gain a better understanding about how to generate 3D animations, specifically those obtained from motion capture.
    - Ideally, I would like to enter the video game industry after I have graduated university, so I believe that this project would be a great opportunity for me to learn about animation as a stepping stone on my career path.
- Create a tool that can allow people to learn how to play a musical instrument (in this case, the guitar) from the comfort of their own home, without the need of going out and hiring a tutor to teach them.
    - Having learnt the guitar for almost 8 years before coming to university, my lessons with my regular tutor stopped due to the travel distance becoming too great, and that I would not be able to have many lessons when I came home on breaks. Providing a 'virtual tutor' would allow me to continue to learn whilst overcoming the problems detailed, as well as others in a similar situation to myself.

Objectives:

- Construct a platform for a working software animation that demonstrates, step-by-step, the necessary hand positions required for guitar chords in a piece of music, within fully rotatable 3D space. This virtual tutor would guide the user through the musical piece in real-time and can be easily adjusted to suit the user's comfort (e.g. speed of the animation/tempo of the song, repeat specific parts of the song, etc.).
- Create a wide library of animations for the 3D animations required, that can smoothly transition from one to another, and can be adapted to cover almost all chord shapes and note patterns in comfortable, easy-to-follow lesson.

- Provide an easily-accessible method of teaching for musicians who, like myself, might lack full use of both hands (e.g. upper- or lower-limb deficiencies, loss of finger mobility, etc.). Accessibility for the users is a key component that I would like to keep in mind for this project.

Deliverables:

- An interactive program that acts as a virtual tutor to teach the user the hand movements required to play a song on guitar.
- A database of 3D motion-captured animations used within the program to animate the virtual tutor.
- A database of lesson files, generated from the 3D animations, that will teach the user how to play a specific song.
- A database of music files containing songs, that can have lessons generated by pairing them with a musical score file (time and resources permitting).
- A database of musical score sheets, that can be used by the animation database to generate a lesson when paired with the corresponding music file (time and resources permitting).
- Test and bug reports of the strategy used for testing the program, any errors that were discovered, and how they were subsequently resolved.

Work Plan:

- **Project Start -> End of October** (up to Reading Week):
  - Literature research and review – analyse previous papers on related topics for inspiration on what to include for project
  - Research appropriate software to use when implementing project
  - MoSCoW – overview of what the project aims to include, what it might include, and what it would like to include given additional time/resources
  - Use Cases – simplified view of how the user interacts with the finished project
- **Start of November -> Mid-December** (up to End of Term 1):
  - Project Plan – finalise and submit by end of Reading Week
  - Begin work on simple prototypes to gain a better understanding of the coding required
  - Acquire motion capture files to be translated to animations for the implemented system (*must be done before end of term!*)
- **Mid-December -> Mid-January** (over Christmas Break):
  - Implement the system over the Christmas holidays – prioritise getting the animations correct and properly synchronised over any other proposed features (e.g. custom songs, reading music sheets, etc.)
  - Test and debug system as the implementation goes underway
- **Mid-January -> Mid-February** (up to Reading Week):
  - Test and evaluate the near-finished system
  - Confer with supervisor over what was proposed from the MoSCoW criteria has been met
  - Implement anything not met from the MoSCoW criteria that both the supervisor and I have agreed I will have enough time to do so and is worth including
  - Interim Report – finalise and submit by Start of February

- **Mid-February -> Mid-March** (up to End of Term 2):
    - Fine tune and finalise the system
- **Mid-March -> End of April** (over Easter Break):
    - Project Report – finalise and prepare to submit
- **End of April -> Start of May** (up to Deadline):
    - Discuss with supervisor any last-minute checks required
    - Review Project Report, checking particularly for spelling, grammar and layout problems that might crop up
    - Submit Project Review
    - Prepare for and sit summer exams
    - Relax!

A.F) Interim Report

The Interim Report was originally submitted Tuesday 4th February 2020, and has been included in this report for completion's sake.

Generating 3D Hand Animations For Virtual Guitar Lessons

Interim Report

Toby Best

MEng Mathematical Computation

Supervisor:

Yuzuko Nakamura

Progress:

The project is a little bit behind where I would have hoped to have achieved by this point, namely as a result of obtaining the hand animations, which are vital for this project to work. After deciding against completing an ethics form for one of the pre-made hand animations that we are initially set on using, after contacting the university's VR department I was informed that they had obtained a set of Manus VR gloves, a new technology for capturing and recording 3D hand movements, at the end of Term 1. By using the Manus gloves to record my animations, I would not need to fulfil an ethics form as everything would be sourced from myself, so my supervisor and I agreed this would be the best choice to move forward.

However, when I went to test the Manus gloves (Wednesday 18th December), although the gloves could capture my hand's movements, the software used for recording (MotionBuilder) was not set up properly at the time. I discussed this with Sebastian Friston and David Swapp, who had been in contact with my supervisor previously to discuss my project, in order to find a workaround. Sebastian obtained access to the plugin necessary for the Manus gloves to record the data in MotionBuilder, but even after testing the plugin I have been unable to record the animations.

As such, we agreed to try recording the animations directly in Unity with a different plugin, which had greater success. I currently have collected a few sample animations, with the intention of recording all of the animations required for the final project before the start of reading week.

Though the animations are behind, the rest of the project is coming along at an expected pace. I now have a functional user interface for interacting with the song, including playing/pausing the song, skipping ahead or behind to a particular part of the song, checkpoint management and switching between the main song and an instrumental version (if one exists).

My next step is to implement the song database, so that the user can select a song from a list to practise with, rather than using the same sample song every time. Once that is done and I have the animations finalised, I will then need to work on getting the animations to match up with a respective guitar tab sheet that reads what animations is required next.

Remaining Work:

In order of importance:

- Record the remaining animations (highest priority)
- Implement the animation handling
- Implement the guitar tab reader and animation player
- Implement the ability to change the tempo and speed of the song
- Implement the song database and its respective UI screen
- Tidy up the UI so it looks cleaner/'more professional', time permitting
- Implement the custom music (lowest priority)

A.G) Code Listings

Although the submitted project includes numerous imported code scripts from assets that were experimented with, such as alternative ways to clip and cull the model mesh of the hands, or the DryWetMIDI library [19], only the most important code relevant to the project is included here. A full overview of all of the relevant code is included in the System Layout in Chapter IV.

Much of the code could not be completed without the example coding offered by the official Unity Scripting Reference documentation [22], or the help provided by the helpful questions on the Unity answers forums [23].

AnimateHand.cs

The main code for controlling the hand movements. It receives the next chord reading from MidiReader.cs, then determines which fret to move to, and what animation fits the chord best.

```
using System.Collections;
using System.Collections.Generic;
using Melanchall.DryWetMidi.Core;
using Melanchall.DryWetMidi.Common;
using Melanchall.DryWetMidi.Interaction;
using UnityEngine;

public class AnimateHand : MonoBehaviour
{
    public Chord chord;
    public float chordtime;
    public bool newchord = false;

    // At their current size, the hands start at Fret 0 at 0.625 and end at Fret 24 at 0.175
    public float difference = 0.425f;

    Vector3 oldposition;
    Vector3 newposition;
    float toMove = 0f;
    bool moving = false;
    int counter = 0;

    public SevenBitNumber[,] fret;

    // Gets the distance to move for each fret
    // Frets are measured with a constant divider 17.817
    float scalelength;
    public float[] fretlength;
    float divider = 17.817f;

    public Tempo tempo;
    public float tempoPercent = 1f;
    public Animation animator;
    string noteToPlayName;
    public string animationToPlay = "CChord";

    public PlaySong playsong;
    public Strum strum;

    // Number of frames for the animations to smoothly transition between
    int lerpMax = 5;

    // Start is called before the first frame update
    void Start()
    {
        tempo = GameObject.Find("TempoBar").GetComponent<Tempo>();
```

```
        animator = gameObject.transform.Find("Hand").GetComponent<Animation>();

        playsong = GameObject.Find("SongPlayer").GetComponent<PlaySong>();

        fret = new SevenBitNumber[6,24];

        // Distance from the first nuts of the guitar to the end of the strings
        // With the model used, the end of the strings are at 0.00
        // So we assume that the hand's starting position is at the nuts
        scalelength = transform.localPosition.y;

        // 60 is Middle C, C4
        // Sets up frets based on standard guitar tuning, E2 A2 D3 G3 B3 E4 on an open string
        // E2 = 40, A2 = 45, D3 = 50, G3 = 55, B3 = 59, E4 = 64

        fret[0,0] = (SevenBitNumber)40;
        fret[1,0] = (SevenBitNumber)45;
        fret[2,0] = (SevenBitNumber)50;
        fret[3,0] = (SevenBitNumber)55;
        fret[4,0] = (SevenBitNumber)59;
        fret[5,0] = (SevenBitNumber)64;

        for (int i = 0; i < 6; i++)
        {
            for (int j = 1; j < 24; j++)
            {
                fret[i,j] = (SevenBitNumber)(fret[i,0] + j);
            }
        }

        fretlength = new float[24];
        fretlength[0] = scalelength;

        // In theory, the 12th fret should be about halfway along the scalelength
        for (int i = 1; i < 24; i++)
        {
            fretlength[i] = fretlength[i - 1] - (fretlength[i - 1] / divider);

            // Rounds to 3 dp for neatness
            fretlength[i] = Mathf.Round(fretlength[i] * 1000f) / 1000f;
        }
    }

    void GetFret()
    {
        int noteToPlay = 0;

        int myFret = 0;

        foreach (Note myNote in chord.Notes)
        {
            // Checks if the root note actually appears in the guitar's tuning
                (i.e. is an E2 or higher note)
            // Sometimes the MIDI file will have a baseline of C#0,
                for example, which isn't actually played
            // If not, it checks the next note to act as the root

            // For most songs written for 6-stringed guitars, most songs do not
                go beyond 1st String 17th fret (A5)
            // Hence why it has been hard-capped as the maximum here
            // This stops the animation player from reading a MIDI note that's beyond the guitar's
                range and trying to adjust for it
            if((myNote.NoteNumber >= (SevenBitNumber)fret[0,0]) &&
                (myNote.NoteNumber <= (SevenBitNumber)fret[5, 16]) && (noteToPlay == 0))
            {
                noteToPlay = (int)myNote.NoteNumber;

                noteToPlayName = myNote.NoteName.ToString();

                for (int i = 0; i < 6; i++)
                {
                    // Tries to prevent unnecessary movement, e.g. trying to play a B as 5th String
                        2nd fret when it's more efficient to play it as 6th String 7th fret
```

```
                    compared to the surrounding notes
                if (noteToPlay < fret[i, 0] + 5)
                {
                    break;
                }

                myFret = noteToPlay - fret[i, 0];
            }

            Debug.Log("Fret is: " + myFret.ToString() + ", Note Number is: "
            + noteToPlay.ToString() + ", Note is: " + myNote.NoteName.ToString());

            toMove = fretlength[myFret];

            oldposition = transform.localPosition;
            newposition = new Vector3(transform.localPosition.x, toMove,
                                      transform.localPosition.z);

            moving = true;

            counter = 0;

            break;
        }

    }

    return;
}

void Animate()
{
    // Eliminates how along the scale the note is, i.e. C3 -> C, A#2 -> A#
    string rawnote = noteToPlayName.Substring(0, noteToPlayName.Length - 1);

    animator.Stop();

    switch (rawnote)
    {
        // Technically speaking, the Sharp notes are not representative of a minor chord
        // C# is not always the basis of a C Minor chord -
            in fact, usually C is still the root note
        // However, considering the current global pandemic limiting
            what animations I have available, I am having to make do with what I've got
        // Thus, this is more of a demonstration of what the code would do if
            I had everything necessary, rather than what it will do

        case("C"):
            animationToPlay = "CChord";
            break;
        case ("C#"):
            animationToPlay = "CMinorChord";
            break;
        case ("D"):
            animationToPlay = "DChord";
            break;
        case ("D#"):
            animationToPlay = "DMinorChord";
            break;
        case ("E"):
            animationToPlay = "EChord";
            break;
        case ("F"):
            animationToPlay = "FChord";
            break;
        case ("F#"):
            animationToPlay = "FMinorChord";
            break;
        case ("G"):
            animationToPlay = "GChord";
            break;
        case ("G#"):
            animationToPlay = "GMinorChord";
            break;
```

```
            case ("A"):
                animationToPlay = "AChord";
                break;

            case ("A#"):
                animationToPlay = "AMinorChord";
                break;
            case ("B"):
                animationToPlay = "BChord";
                break;
            default:
                animationToPlay = "CChord";
                break;
        }

        // For the purpose of this project demonstration, the speed is increased
              so that the animation actually plays in time
        // With sufficient time and access to higher-quality animations, this would not be necessary
        animator[animationToPlay].speed = tempoPercent * 5f;
        animator.Play(animationToPlay);
    }

    // Update is called once per frame
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Z))
        {
            foreach (Note note in chord.Notes)
            {
                Debug.Log(note.ToString() + ", " + note.NoteNumber);
            }
        }

        if (newchord)
        {
            tempoPercent = tempo.tempoPercent;
            strum.playing = true;
            newchord = false;
            //Debug.Log(chord.ToString() + ", " + chordtime.ToString());

            GetFret();
            Animate();
        }

        if (moving)
        {
            counter++;

            if (counter >= lerpMax)
            {
                moving = false;
            }

            float lerp = counter / (float)lerpMax;

            transform.localPosition = Vector3.Lerp(oldposition, newposition, lerp);
        }

        if (playsong.playing == false)
        {
            animator.Stop();
        }
    }
}
```

MidiReader.cs

The main code for extracting the data from the MIDI file that is needed to play the song. If the song is playing, it checks what chord should be played and feeds it to AnimateHand.cs.

```
using UnityEngine;
using System.Collections;
using Melanchall.DryWetMidi.Core;
using Melanchall.DryWetMidi.Interaction;
using System.IO;
using System.Linq;

public class MidiReader : MonoBehaviour
{
    public GameObject songplayer;
    public PlaySong playsong;
    public string filepath;

    public float songtimer;
    public float oldtime = 0f;

    MidiFile midiFile;
    TempoMap tempoMap;
    TrackChunk guitarTrack;
    ChordsCollection myChords;

    public AnimateHand animateHand;
    public void Setup()
    {
        songplayer = GameObject.Find("SongPlayer");
        playsong = songplayer.GetComponent<PlaySong>();
        filepath = MasterController.absolutePath;

        string midipath = filepath + @"\notes.mid";

        if (File.Exists(midipath))
        {
            Debug.Log("Midi File is: " + midipath);
        }

        else
        {
            Debug.Log("Error with midi file: " + midipath);
        }

        midiFile = MidiFile.Read(midipath);

        if (midiFile != null)
        {
            Debug.Log("Got midi file!");
        }

        else
        {
            Debug.Log("Error with loading midi file.");
        }

        foreach (TrackChunk track in midiFile.GetTrackChunks())
        {
            var trackName = track.Events
                          .OfType<SequenceTrackNameEvent>()
                          .FirstOrDefault()
                          ?.Text;

            // Skips over unnamed tracks
            if (trackName != null)
            {
                Debug.Log(trackName);
                // Ensures we're only reading the Guitar track
                if ((trackName.ToUpper()).Contains("GUITAR"))
                {
                    guitarTrack = track;
                    Debug.Log("Got guitar track!");
                    break;
                }
            }
        }
```

```
        tempoMap = midiFile.GetTempoMap();

        // Manage chords of the guitar track
        using (var chordsManager = new ChordsManager(guitarTrack.Events))
        {
            myChords = chordsManager.Chords;
            Debug.Log("Obtained chords!");
        }
    }

    void Update()
    {
        // Rounds to 3 dp so it's the same format as the chords' times.
        songtimer = Mathf.Round(playsong.songtimer * 1000f) / 1000f;

        if (playsong.playing)
        {
            Chord tempchord = null;
            float temptime = 0f;

            foreach (Chord chord in myChords)
            {
                MetricTimeSpan chordTime = chord.TimeAs<MetricTimeSpan>(tempoMap);
                float realtime = (chordTime.Minutes * 60) + chordTime.Seconds
                        + ((float)chordTime.Milliseconds / 1000);

                if(songtimer > realtime)
                {
                    tempchord = chord;
                    temptime = realtime;
                }

                if((songtimer < realtime) && (temptime > oldtime))
                {
                    //Debug.Log(tempchord.ToString() + ", " + temptime.ToString());
                    animateHand.chord = tempchord;
                    animateHand.chordtime = temptime;
                    animateHand.newchord = true;
                    oldtime = temptime;
                    break;
                }
            }
        }

        else
        {
            oldtime = 0f;
        }
    }
}
```

MusicLayers.cs

The main code for extracting the individual .ogg sound files used by Clone Hero in each song folder. It instantiates a new empty game object with an audio clip set to one layer for every sound layer found, then sets them up for PlaySong.cs to control.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Networking;
using UnityEngine.Audio;
using System.IO;

public class MusicLayers : MonoBehaviour
{
    public int layerCount = 0;
```

```
AudioSource source;
public AudioMixerGroup tempoMixer;

// Relative path to where the app is running
public string absolutePath;

// Ensures we only check files of this type
string fileType = ".ogg";

FileInfo[] files;

void Start()
{
    absolutePath = MasterController.absolutePath;

    if (File.Exists(absolutePath + ".meta"))
    {
        Debug.Log("File path is: " + absolutePath);
    }

    else
    {
        Debug.Log("Error with file path: " + absolutePath
            + "; symptoms unknown.");
    }

    GameObject.Find("MidiHandler").GetComponent<MidiReader>().Setup();

    loadSongNames();

    loadSounds();
}

void loadSongNames()
{
    string path = absolutePath + @"\song.ini";

    PlaySong playSong = gameObject.GetComponent<PlaySong>();

    // Checks if file exists, if not, sets as an error then immediately returns
    if (File.Exists(path))
    {
        Debug.Log("Found song ini file.");
    }

    else
    {
        playSong.songName.text = "Error Name";
        playSong.artistName.text = "Error Name";
        Debug.Log("Error finding ini file.");
        return;
    }

    StreamReader loadfile = new StreamReader(path);

    if(loadfile != null)
    {
        Debug.Log("Loaded song ini file.");
    }

    else
    {
        Debug.Log("Error loading ini file.");
    }

    string contents = loadfile.ReadToEnd();
    loadfile.Close();

    string[] lines = contents.Split("\n"[0]);

    string songname = "Error Name";
    string artistname = "Error Name";
```

```
        // In song.idi, the song name and artist name are stored as
        'name = song name' and 'artist = artist name'

        foreach (string line in lines)
        {
            int idx = line.IndexOf('=');
            if (idx != -1)
            {
                // Gets the substring before ' = ', e.g. 'name = ...' -> 'name'
                string temp = line.Substring(0, idx - 1);

                if(temp == "name")
                {
                    songname = line.Substring(idx + 2);
                }
                // Sometimes the names are stored without a space between =, e.g. 'name=song name'
                if (temp == "nam")
                {
                    songname = line.Substring(idx + 1);
                }

                if (temp == "artist")
                {
                    artistname = line.Substring(idx + 2);
                }

                if (temp == "artis")
                {
                    artistname = line.Substring(idx + 1);
                }

                if (!(songname == "Error Name" || artistname == "Error Name"))
                {
                    Debug.Log(songname + " by " + artistname);
                    break;
                }
            }
        }
        playSong.songName.text = songname;
        playSong.artistName.text = artistname;

        return;


    }
    void loadSounds()
    {
        DirectoryInfo info = new DirectoryInfo(absolutePath);
        files = info.GetFiles();

        //check if the file is valid and load it
        foreach (FileInfo f in files)
        {
            // Checks that the file is a sound file.
            // We need all sound files except the preview here, hence its exclusion.
            // Some song files also have a backing crowd, so we need to delete those too.
            if (validFileType(f.Name) && (f.Name != "preview.ogg"
                        && f.Name != "crowd.ogg"))
            {
                //Debug.Log("Start loading " + f.FullName);

                // Counts the number of sound layers
                layerCount++;

                StartCoroutine(LoadFile(f.FullName));
            }
        }
    }

    bool validFileType(string filename)
    {
        /// Checks if the extension of the file is correct (.ogg)
        if (filename.Substring(filename.Length - 4, 4) == fileType) return true;
```

```
        else return false;
    }

    IEnumerator LoadFile(string path)
    {
        string url = string.Format("file://{0}", path);
        using (UnityWebRequest www =
                UnityWebRequestMultimedia.GetAudioClip(url, AudioType.OGGVORBIS))
        {
            yield return www.SendWebRequest();

            if (www.isNetworkError)
            {
                Debug.Log(www.error);
            }
            else
            {
                GameObject newlayer = new GameObject();
                newlayer.transform.parent = gameObject.transform;

                int idx = path.LastIndexOf(@"\");
                string newname = path.Substring(idx + 1, path.Length - idx - 1);

                // Sets the layer's name to the sound file without the extension
                newlayer.name = newname.Substring(0, newname.Length - 4);
                AudioClip myClip = DownloadHandlerAudioClip.GetContent(www);

                source = newlayer.AddComponent<AudioSource>();
                source.outputAudioMixerGroup = tempoMixer;
                source.clip = myClip;
                source.playOnAwake = false;

                //source.Play();

                //Debug.Log("Sound file implemented! " + myClip.name + ", length: "
        + myClip.length);
            }
        }
    }
}
```

PlaySong.cs

The main code for controlling the song, determining whether it is playing or is stopped. PlaySong.cs waits for MusicLayers.cs to hand it the guitar.ogg sound fill before it does anything important.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.UI;

public class PlaySong : MonoBehaviour
{
    public AudioClip song;

    private AudioSource audioSource;

    public bool playing = false;
    public bool swapping = false;
    public bool skipping = false;

    public float songtimer = 0f;

    public TextMesh songName;
    public TextMesh artistName;
    public TextMesh songDuration;
```

```
    public string songlength;

    public ButtonClick playbutton;
    public ButtonClick resetbutton;
    public ButtonClick swapbutton;
    public ButtonClick loopbutton;

    public bool initialised = false;

    public float loopstart = 0f;
    public float loopfinish = 5000f;
    public bool looping = false;
    public bool allmute = false;

    public GameObject layerslist;
    public GameObject checkpointlist;

    MusicLayers musicLayers;

    void Start()
    {
        songName = GameObject.Find("SongName").GetComponent<TextMesh>();
        artistName = GameObject.Find("ArtistName").GetComponent<TextMesh>();
        songDuration = GameObject.Find("SongDuration").GetComponent<TextMesh>();

        musicLayers = gameObject.GetComponent<MusicLayers>();

        songlength = string.Format("{0}:{1}", Mathf.Floor(song.length / 60).ToString("00"),
        Mathf.Floor(song.length % 60).ToString("00"));

        songDuration.text = "00:00 / " + songlength;

        audioSource = GameObject.Find("SourceMain").GetComponent<AudioSource>();
        audioSource.clip = song;

        Debug.Log("Song length: " + songlength);

        //Debug.Log("Press the I key to start Song 1 (the main song).");
        //Debug.Log("Press the O key to start Song 2 (the instrumental).");
        //Debug.Log("Press the P key to switch between Song 1 and Song 2.");
        Debug.Log("Press the R key to restart the song.");

        playbutton = GameObject.Find("PlayButton").GetComponent<ButtonClick>();
        resetbutton = GameObject.Find("ResetButton").GetComponent<ButtonClick>();
        swapbutton = GameObject.Find("SwapButton").GetComponent<ButtonClick>();
        loopbutton = GameObject.Find("LoopButton").GetComponent<ButtonClick>();
    }

    void Initialise(GameObject guitarLayer)
    {
        audioSource = guitarLayer.GetComponent<AudioSource>();
        song = audioSource.clip;
        songlength = string.Format("{0}:{1}", Mathf.Floor(song.length / 60).ToString("00"),
        Mathf.Floor(song.length % 60).ToString("00"));
        songDuration.text = "00:00 / " + songlength;

        loopfinish = song.length;

        // Ensures that we don't have to loop through this more than once
        initialised = true;

        return;
    }

    public string GetLayerName(string layerName)
    {
        char[] temp = (layerName.Replace("_", " ")).ToCharArray();
        temp[0] = char.ToUpper(temp[0]);

        return new string(temp);
    }

    public void HideImage(Image image)
    {
```

```
        // Toggles visibility
        // If we want to disable it, we make it invisible
        // If we want to enable it, we make it visible again

        Color tempcolor = image.color;

        if (tempcolor.a > 0)
        {
            tempcolor.a = 0f;
            image.color = tempcolor;

            GameObject temp1 = GameObject.Find("CheckpointElement");

            foreach (Transform child in temp1.transform)
            {
                Button childbutton = child.GetComponent<Button>();
                childbutton.interactable = false;
            }

        }

        else
        {
            tempcolor.a = 1f;
            image.color = tempcolor;

            GameObject temp1 = GameObject.Find("CheckpointElement");

            foreach (Transform child in temp1.transform)
            {
                Button childbutton = child.GetComponent<Button>();
                childbutton.interactable = true;
            }
        }
    }

    public bool MuteLayer(GameObject layer)
    {
        audioSource = layer.GetComponent<AudioSource>();

        audioSource.mute = !audioSource.mute;

        allmute = true;

        foreach (Transform child in transform)
        {
            AudioSource layerSource = child.gameObject.GetComponent<AudioSource>();
            if (layerSource.mute == false)
            {
                allmute = false;
            }
        }

        return audioSource.mute;
    }

    public void MuteAll()
    {
        allmute = true;
        foreach (Transform child in transform)
        {
            AudioSource layer = child.gameObject.GetComponent<AudioSource>();
            if (layer.mute == false)
            {
                allmute = false;
            }
        }

        foreach (Transform child in transform)
        {
            AudioSource layer = child.gameObject.GetComponent<AudioSource>();
            if (allmute == false)
            {
                layer.mute = true;
```

```
            }
            else
            {
                layer.mute = false;
            }
        }

        allmute = !allmute;

        GameObject temp1 = GameObject.Find("LayerElement");

        foreach (Transform child in temp1.transform)
        {
            if (child.gameObject.name == "LayerButton")
            {
                LayerButtonHandler temp2 = child.gameObject.GetComponent<LayerButtonHandler>();
                temp2.muteall = allmute;
                temp2.mute = allmute;
                temp2.SetText();
            }
        }
    }

    void Reset()
    {
        resetbutton.enabled = false;
        playing = false;
        songtimer = 0f;
        swapping = false;
        GameObject.Find("MidiHandler").GetComponent<MidiReader>().oldtime = 0f;

        foreach (Transform child in transform)
        {
            AudioSource layer = child.gameObject.GetComponent<AudioSource>();
            layer.Pause();
            layer.time = songtimer;
        }

        RotateGuitar guitar = GameObject.Find("Guitar").GetComponent<RotateGuitar>();
        guitar.resetting = true;
        guitar.isRotating = false;

        GameObject.Find("TempoBar").GetComponent<Tempo>().Reset();

        songDuration.text = "00:00 / " + songlength;

        playbutton.GetComponent<Renderer>().material = playbutton.material1;

        Debug.Log("Resetting...");

        return;
    }

    void SortChildren()
    {
        // Simple Bubble Sort to organise the layer GameObject

        int childCount = transform.childCount;

        bool swaps = true;

        while(swaps)
        {
            swaps = false;

            for (int i = 0; i < childCount - 1; i++)
            {
                Transform child1 = transform.GetChild(i);
                Transform child2 = transform.GetChild(i + 1);

                if (string.Compare(child1.name, child2.name) > 0)
                {
                    swaps = true;
                    int temp = child1.GetSiblingIndex();
```

```
                child1.SetSiblingIndex(child2.GetSiblingIndex());
                child2.SetSiblingIndex(temp);
            }
        }
    }

    Debug.Log("Finished sorting children!");
}

void Update()
{
    if (!initialised && transform.childCount == musicLayers.layerCount)
    {
        GameObject guitarLayer = gameObject.transform.Find("guitar").gameObject;

        if (guitarLayer != null)
        {
            Initialise(guitarLayer);

            SortChildren();

            GameObject button = GameObject.Find("LayerButton");
            GameObject layerElement = GameObject.Find("LayerElement");

            foreach (Transform child in transform)
            {
                GameObject newbutton = Instantiate(button,
                    new Vector3(button.transform.position.x, button.transform.position.y,
                    button.transform.position.z), Quaternion.identity);
                newbutton.transform.parent = layerElement.transform;
                newbutton.name = "LayerButton";

                LayerButtonHandler buttonhandler = newbutton.GetComponent<LayerButtonHandler>();
                buttonhandler.myLayer = child.gameObject;
            }

            Destroy(button);

            layerslist.SetActive(false);

            Image temp = GameObject.Find("CheckpointList").GetComponent<Image>();
            HideImage(temp);

            GameObject.Find("LoadingScreen").SetActive(false);

        }
    }

    else

    {
        string songtime;

        if (playing)
        {
            songtime = string.Format("{0}:{1}", Mathf.Floor(audioSource.time / 60).ToString("00"),
            Mathf.Floor(audioSource.time % 60).ToString("00"));

            songDuration.text = songtime + " / " + songlength;

            if (!skipping)
            {
                songtimer = audioSource.time;
            }

            // If the song has reached the end, has stopped, and is not looping
            if (songtimer == 0 && !audioSource.isPlaying && !looping)
            {
                playbutton.GetComponent<Renderer>().material = playbutton.material1;
                playing = false;
            }
        }

        if (Input.GetKeyDown(KeyCode.P) || playbutton.enabled)
```

```
{
    playbutton.enabled = false;

    //
    playing = !playing;

    if (playing)
    {
        foreach (Transform child in transform)
        {
            AudioSource layer = child.gameObject.GetComponent<AudioSource>();
            layer.Play(0);
            layer.time = songtimer;
        }

        playbutton.GetComponent<Renderer>().material = playbutton.material2;

        Debug.Log("Playing song...");
    }

    else
    {
        foreach (Transform child in transform)
        {
            AudioSource layer = child.gameObject.GetComponent<AudioSource>();
            layer.Pause();
            songtimer = layer.time;
        }

        playbutton.GetComponent<Renderer>().material = playbutton.material1;

        Debug.Log("Pausing song at " + songtimer.ToString());
    }
}

if(Input.GetKeyDown(KeyCode.O) || loopbutton.enabled)
{
    loopbutton.enabled = false;
    Image temp = GameObject.Find("CheckpointList").GetComponent<Image>();
    HideImage(temp);
}

if (looping)
{
    // If the song plays past the second checkpoint
    if(songtimer >= loopfinish)
    {
        songtimer = loopstart;

        foreach (Transform child in transform)
        {
            AudioSource layer = child.gameObject.GetComponent<AudioSource>();
            layer.time = songtimer;
        }

        // Sets the old value to just before the current song timer,
        // so that we can get the current animation to show
        GameObject.Find("MidiHandler").GetComponent<MidiReader>().oldtime
             = Mathf.Max(songtimer - 0.1f, 0f);
    }

    // If the song reaches the end and stops playing, ensures that it keeps playing
    if (songtimer == 0 && !audioSource.isPlaying && playing)
    {
        songtimer = loopstart;

        foreach (Transform child in transform)
        {
            AudioSource layer = child.gameObject.GetComponent<AudioSource>();
            layer.Play(0);
            layer.time = songtimer;

        }
```

```
                // Sets the old value to just before the current song timer,
                // so that we can get the current animation to show
                GameObject.Find("MidiHandler").GetComponent<MidiReader>().oldtime = 0f;
            }
        }

        if (Input.GetKeyDown(KeyCode.I) || swapbutton.enabled)
        {
            swapbutton.enabled = false;
            bool temp = layerslist.activeSelf;
            layerslist.SetActive(!temp);
        }

        if (skipping)
        {
            songtime = string.Format("{0}:{1}", Mathf.Floor(songtimer / 60).ToString("00"),
            Mathf.Floor(songtimer % 60).ToString("00"));

            songDuration.text = songtime + " / " + songlength;

            foreach (Transform child in transform)
            {
                AudioSource layer = child.gameObject.GetComponent<AudioSource>();
                layer.time = songtimer;
            }

            skipping = !skipping;

            // Sets the old value to just before the current song timer,
            // so that we can get the current animation to show
            GameObject.Find("MidiHandler").GetComponent<MidiReader>().oldtime
            = Mathf.Max(songtimer - 0.1f, 0f);

            Debug.Log("Skipping...");
        }

        if (Input.GetKeyDown(KeyCode.R) || resetbutton.enabled)
        {
            Reset();
        }
    }
}
}
```

ProgressBar.cs

The main code for controlling whether the song is skipping ahead or backwards, as well as when to instantiate a new checkpoint.

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.IO;
using UnityEngine;

public class ProgressBar : MonoBehaviour
{
    bool canclick;
    Renderer rend;

    public float songPercent;
    public float songtimer = 0f;
    public float songlength = 0f;

    public int checkpointCount = 0;
    int maxCheckPointCount = 8;

    public float[] checkpointTimes;

    GameObject tracker;
    SongTracker songtracker;
```

```
    GameObject cursortracker;

    GameObject songplayer;
    PlaySong playsong;

    public GameObject checkpoint;

    public bool loopcheck = false;

    // Default value means that, if looping is called, they will always loop from the start to the end
    public int loopID1 = -1;
    public int loopID2 = -1;

    float trackerYPos;
    float trackerZPos;

    GameObject button;

    void Start()
    {
        rend = GetComponent<Renderer>();
        canclick = false;

        tracker = GameObject.Find("ProgressTracker");
        songtracker = tracker.GetComponent<SongTracker>();

        songtracker.start = rend.bounds.min.x;
        songtracker.finish = rend.bounds.max.x;

        cursortracker = GameObject.Find("MouseTracker");

        checkpoint = GameObject.Find("Checkpoint");

        songplayer = GameObject.Find("SongPlayer");
        playsong = songplayer.GetComponent<PlaySong>();

        trackerYPos = tracker.transform.position.y;
        trackerZPos = tracker.transform.position.z;

        button = GameObject.Find("CheckpointButtonExample");
        button.SetActive(false);

    }

    void OnMouseEnter()
    {
        canclick = true;
        cursortracker.transform.position = new Vector3(cursortracker.transform.position.x, 0f,
        cursortracker.transform.position.z);
    }

    void OnMouseExit()
    {
        canclick = false;
        cursortracker.transform.position = new Vector3(cursortracker.transform.position.x, -1f,
        cursortracker.transform.position.z);
    }

    public void SaveCheckpoints(float[] checkpoints)
    {
        // Path of the file
        string path = songplayer.GetComponent<MusicLayers>().absolutePath;
        path = path + @"\checkpoints.txt";

        // Checks if file exists, if not creates it
        if (!File.Exists(path))
        {
            File.WriteAllText(path, "Checkpoint List\n");
        }

        // If no checkpoints are made, returns
        if (checkpointCount == 0)
        {
            Debug.Log("Error - no checkpoints to save!");
```

```
            return;
        }

        // Else it saves the current times recorded for each checkpoint
        else
        {
            string value = checkpoints[0].ToString();
            File.WriteAllText(path, value);

            for (int i = 1; i < checkpointCount; i++)
            {
                value = checkpoints[i].ToString();
                File.AppendAllText(path, "\n" + value);
            }
        }

        Debug.Log("Checkpoints saved!");
        return;
    }

    public void LoadCheckpoints()
    {
        // Path of the file
        string path = songplayer.GetComponent<MusicLayers>().absolutePath;
        path = path + @"\checkpoints.txt";

        // Checks if file exists, if not creates it, then immediately returns
        if (!File.Exists(path))
        {
            File.WriteAllText(path, "Checkpoint List\n");
            return;
        }

        StreamReader loadfile = new StreamReader(path);
        string contents = loadfile.ReadToEnd();
        loadfile.Close();

        string[] lines = contents.Split("\n"[0]);

        // If no checkpoints are saved...
        if (lines[0] == "Checkpoint List\n")
        {
            return;
        }

        else
        {
            // Deletes any existing checkpoints
            foreach (Transform child in transform)
            {
                if (child.gameObject.name == "checkpoint")
                {
                    child.gameObject.name = "ToDestroy";
                    Destroy(child.gameObject);
                }
            }

            checkpointCount = 0;

            // Then rebuilds the saved checkpoints
            foreach (string line in lines)
            {
                float checkpointtimer = float.Parse(line);

                songPercent = 100f * checkpointtimer / songlength;

                float xPos = (songPercent / 100f) * (songtracker.finish - songtracker.start)
                + songtracker.start;

                CreateCheckpoint(xPos, songPercent);

            }

            SpawnCheckPointButton();
```

```
            Debug.Log("Checkpoints loaded!");
            return;
        }
    }

    public void CheckPointSort()
    {
        // Creates an array of each checkpoint, and then organises them based on their registered time

        if (checkpointCount == 0)
        {
            return;
        }

        checkpointTimes = new float[checkpointCount];

        int i = 0;

        foreach (Transform child in transform)
        {
            if (child.gameObject.name == "checkpoint")
            {
                Debug.Log("i: " + i);
                checkpointTimes[i] = child.GetComponent<Checkpoint>().songtimer;
                i++;
            }
        }

        // Sorts the array in ascending order
        Array.Sort(checkpointTimes);

        for (i = 0; i < checkpointCount; i++)
        {
            foreach (Transform child in transform)
            {
                if ((child.gameObject.name == "checkpoint")
                && child.GetComponent<Checkpoint>().songtimer == checkpointTimes[i]))
                {
                    child.GetComponent<Checkpoint>().identifier = i;
                    break;
                }
            }
        }

        bool swaps = true;

        if (checkpointCount > 0)
        {
            while (swaps)
            {
                swaps = false;

                for (i = 0; i < transform.childCount - 1; i++)
                {
                    Transform child1 = transform.GetChild(i);
                    Transform child2 = transform.GetChild(i + 1);

                    Checkpoint child1ID = child1.GetComponent<Checkpoint>();
                    Checkpoint child2ID = child2.GetComponent<Checkpoint>();

                    if ((child1.gameObject.name == "checkpoint") && (child2.gameObject.name
                         == "checkpoint") && (child1ID.identifier > child2ID.identifier))
                    {
                        swaps = true;
                        int temp = child1.GetSiblingIndex();
                        child1.SetSiblingIndex(child2.GetSiblingIndex());
                        child2.SetSiblingIndex(temp);
                    }
                }
            }
        }

        return;
```

```
    }

    void CreateCheckpoint(float xPos, float songPercent)
    {
        // Creates a new checkpoint, but only if there is enough room for a new checkpoint
        if (checkpointCount < maxCheckPointCount)
        {
            GameObject newcheckpoint = (GameObject)Instantiate(checkpoint, new Vector3(xPos,
        gameObject.transform.position.y, gameObject.transform.position.z), Quaternion.identity);
            newcheckpoint.name = "checkpoint";
            newcheckpoint.transform.parent = gameObject.transform;
            Checkpoint mycheckpoint = newcheckpoint.GetComponent<Checkpoint>();

            songtimer = songlength * songPercent / 100f;
            mycheckpoint.songtimer = songtimer;
            checkpointCount++;
            Debug.Log("Spawning checkpoint...");
        }

        else
        {
            Debug.Log("Too many checkpoints spawned!");
        }

        CheckPointSort();

        return;
    }

    public void SpawnCheckPointButton()
    {
        GameObject temp = GameObject.Find("CheckpointElement");

        foreach (Transform child in temp.transform)
        {
            if (child.name == "CheckpointButton")
            {
                Destroy(child.gameObject);
            }
        }

        GameObject.Find("LoopingButton").GetComponent<CheckpointButtonHandler>().clickcount = 0;

        foreach (Transform child in transform)
        {
            if(child.name == "checkpoint")
            {
                GameObject newbutton = Instantiate(button, new Vector3(button.transform.position.x,
                 button.transform.position.y, button.transform.position.z), Quaternion.identity);
                newbutton.transform.parent = temp.transform;
                newbutton.name = "CheckpointButton";
                newbutton.SetActive(true);

                CheckpointButtonHandler buttonhandler
                 = newbutton.GetComponent<CheckpointButtonHandler>();
                buttonhandler.id = child.GetComponent<Checkpoint>().identifier;
                buttonhandler.timer = child.GetComponent<Checkpoint>().songtimer;
            }
        }
    }

    void LoopSong(float loopstart, float loopfinish)
    {
        playsong.looping = !playsong.looping;

        if (playsong.looping)
        {
            // Ensures that the start is lower than the finish
            if(loopstart > loopfinish)
            {
                float temp = loopstart;
                loopstart = loopfinish;
                loopfinish = temp;
            }
```

```
            playsong.loopstart = loopstart;
            playsong.loopfinish = loopfinish;
        }
    }

    void Update()
    {
        if(Input.GetKeyDown(KeyCode.K))
        {
            if(checkpointCount != 0)
            {
                SaveCheckpoints(checkpointTimes);
            }

            else
            {
                Debug.Log("Error - no checkpoints to save!");
            }
        }

        if(Input.GetKeyDown(KeyCode.L))
        {
            LoadCheckpoints();
        }

        /*if (Input.GetKeyDown(KeyCode.C))
        {
            loopcheck = true;
        }*/

        songlength = playsong.song.length;

        songtimer = playsong.songtimer;

        if (loopcheck)
        {
            loopcheck = false;
            float temp1;
            float temp2;

            // If no loopID is given, sets to loop from the start
            if (loopID1 < 0 || loopID1 >= checkpointCount)
            {
                temp1 = 0f;
            }

            else
            {
                temp1 = checkpointTimes[loopID1];
            }

            // If no loopID is given...
            if (loopID2 < 0 || loopID2 >= checkpointCount)
            {
                // ...and if only one checkpoint is set to loop,
                 it loops from the beginning until the checkpoint...
                if ((temp1 >= 0f) && (temp1 < songlength / 2))
                {
                    temp2 = 0f;
                }

                // ...otherwise it sets to loop until the end
                else
                {
                    temp2 = songlength;
                }
            }
            else
            {
                temp2 = checkpointTimes[loopID2];
            }
            // Failsafe in case they both end up the same
            if (temp1 == temp2)
```

```
            {
                if (temp2 == songlength)
                {
                    temp2 = 0f;
                }
                else
                {
                    temp2 = songlength;
                }
            }

            LoopSong(temp1, temp2);
        }



        float difference = (songtracker.finish - songtracker.start) * (songtimer / songlength);

        tracker.transform.position = new Vector3(songtracker.start + difference, trackerYPos,
trackerZPos);

        if (canclick)
        {

            Vector2 mouse = new Vector2(Input.mousePosition.x, Input.mousePosition.y);
            Ray ray;
            ray = Camera.main.ScreenPointToRay(mouse);
            RaycastHit hit;

            if (Physics.Raycast(ray, out hit, 100))
            {
                cursortracker.transform.position = new Vector3(hit.point.x,
                 cursortracker.transform.position.y, cursortracker.transform.position.z);
                string percent = ((hit.point.x - songtracker.start) /
                (songtracker.finish - songtracker.start) * 100).ToString("F2");

                songPercent = float.Parse(percent);

                if (Input.GetMouseButtonDown(0))
                {
                    songtimer = songlength * songPercent / 100f;
                    Debug.Log(songtimer.ToString("F2"));

                    playsong.songtimer = songtimer;
                    playsong.skipping = true;
                }

                if (Input.GetMouseButtonDown(2))
                {
                    CreateCheckpoint(hit.point.x, songPercent);
                    SpawnCheckPointButton();
                }

            }

        }
    }
}
```

RotateGuitar.cs

The main code for controlling the movement of the guitar model, such as its position and its rotation.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class RotateGuitar : MonoBehaviour
{
    float rotSpeed = 20f;
    public Quaternion originalRotation;
```

```
    public Vector3 originalPosition;

    public bool isRotating = false;
    public bool resetting = false;

    public bool rightHand = false;

    public GameObject hand;
    public GameObject strumhand;

    public Vector3 originalhandposition;
    public Quaternion originalstrumhandrotation;

    public PlaySong playsong;

    // Ensures that the guitar does not go behind the background
    float miny = 3f;

    Renderer rend;

    void Start()
    {
        rightHand = MasterController.rightHand;

        if(rightHand)
        {
            transform.Rotate(0f, 0f, 180f);
            transform.position = new Vector3 (transform.position.x + 2,
                transform.position.y, transform.position.z);
            Destroy(GameObject.Find("LeftHand"));
            Destroy(GameObject.Find("StrummingIfLeft"));
            hand = GameObject.Find("RightHand");
            strumhand = GameObject.Find("StrummingIfRight");
        }

        else
        {
            Destroy(GameObject.Find("RightHand"));
            Destroy(GameObject.Find("StrummingIfRight"));
            hand = GameObject.Find("LeftHand");
            strumhand = GameObject.Find("StrummingIfLeft");
        }

        AnimateHand myHand = hand.GetComponent<AnimateHand>();

        myHand.strum = strumhand.GetComponent<Strum>();

        strumhand.GetComponent<Strum>().animator = myHand;

        originalhandposition = hand.transform.localPosition;
        originalstrumhandrotation = strumhand.transform.rotation;

        GameObject.Find("MidiHandler").GetComponent<MidiReader>().animateHand = myHand;

        originalRotation = transform.rotation;
        originalPosition = transform.position;
        rend = GetComponent<Renderer>();

        playsong = GameObject.Find("SongPlayer").GetComponent<PlaySong>();
    }

    void Update()
    {
        if (Input.GetMouseButtonDown(0))
        {
            Vector2 mouse = new Vector2(Input.mousePosition.x, Input.mousePosition.y);
            Ray ray;
            ray = Camera.main.ScreenPointToRay(mouse);
            RaycastHit hit;

            if (Physics.Raycast(ray, out hit, 100))
            {
                if ((hit.point.x >= rend.bounds.min.x) && (hit.point.x <= rend.bounds.max.x)
                && (hit.point.y >= rend.bounds.min.y) && (hit.point.y <= rend.bounds.max.y))
```

```
        {
            isRotating = true;
            resetting = false;
        }
    }
}

if (isRotating)
{
    float rotX = Input.GetAxis("Mouse X") * rotSpeed;
    float rotY = Input.GetAxis("Mouse Y") * rotSpeed;

    transform.Rotate(Vector3.up, -rotX);
    transform.Rotate(Vector3.right, -rotY);
}

if (Input.GetMouseButtonUp(0))
{
    isRotating = false;
}

if (Input.GetMouseButtonDown(1))
{
    resetting = true;
    isRotating = false;
}

// Moves the guitar to the right
if (Input.GetKey(KeyCode.RightArrow) || Input.GetKey(KeyCode.D))
{
    Vector3 position = this.transform.position;
    position.x = position.x + (Time.deltaTime);
    this.transform.position = position;
}

// Moves the guitar to the left
if (Input.GetKey(KeyCode.LeftArrow) || Input.GetKey(KeyCode.A))
{
    Vector3 position = transform.position;
    position.x = position.x - (Time.deltaTime);
    transform.position = position;
}

// Moves the guitar up
if (Input.GetKey(KeyCode.UpArrow) || Input.GetKey(KeyCode.W))
{
    Vector3 position = this.transform.position;
    position.z = position.z + (Time.deltaTime);
    this.transform.position = position;
}

// Moves the guitar down
if (Input.GetKey(KeyCode.DownArrow) || Input.GetKey(KeyCode.S))
{
    Vector3 position = this.transform.position;
    position.z = position.z - (Time.deltaTime);
    this.transform.position = position;
}

// Moves the guitar closer to the camera
if (Input.GetKey(KeyCode.Q) || Input.GetKey(KeyCode.Comma))
{
    Vector3 position = this.transform.position;
    position.y = position.y + (Time.deltaTime);
    this.transform.position = position;
}

// Moves the guitar away from the camera
if (Input.GetKey(KeyCode.E) || Input.GetKey(KeyCode.Period))
{
    Vector3 position = this.transform.position;
    position.y = position.y - (Time.deltaTime);

    // Stops the guitar model going beyond the background
```

```
            if(position.y < miny)
            {
                position.y = miny;
            }
            this.transform.position = position;
        }

        if (resetting)
        {
            transform.rotation = Quaternion.Slerp(transform.rotation, originalRotation,
                Time.deltaTime * rotSpeed * 0.5f);
            transform.position = Vector3.Lerp(transform.position, originalPosition,
                Time.deltaTime * rotSpeed * 0.5f);

            if ((transform.position == originalPosition) && (playsong.playing == true))
            {
                transform.rotation = originalRotation;
                resetting = false;
            }
            if (playsong.playing == false)
            {
                hand.transform.localPosition = Vector3.Lerp(hand.transform.localPosition,
                originalhandposition, Time.deltaTime * rotSpeed * 0.5f);
                strumhand.transform.rotation = Quaternion.Slerp(strumhand.transform.rotation,
                originalstrumhandrotation, Time.deltaTime * rotSpeed * 0.5f);
                hand.GetComponent<AnimateHand>().animator.Stop();
                strumhand.GetComponent<Strum>().counter = 0;

                if ((hand.transform.localPosition == originalhandposition)
                && (transform.position == originalPosition))
                {
                    transform.rotation = originalRotation;
                    strumhand.transform.rotation = originalstrumhandrotation;
                    resetting = false;
                }
            }
        }
    }
}
```

SongListMenu.cs

The main code for reading what songs are saved in the Songs folder, and determining whether they are of a valid format for the tutor to teach.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.Networking;
using UnityEngine.UI;
using System.IO;

public class SongListMenu : MonoBehaviour
{
    public string absolutePath = "./Songs";
    FileInfo[] folders;
    public GameObject content;
    public GameObject buttonexample;

    public Font font;
    public bool rightHand = false;
    bool first = true;

    // Start is called before the first frame update
    void Start()
    {
        content = GameObject.Find("Content");
        buttonexample = GameObject.Find("SongButton");
        // To test in Unity Editor
        if (Application.isEditor) absolutePath = @"Assets\Songs";
        DirectoryInfo info = new DirectoryInfo(absolutePath);
        folders = info.GetFiles();
```

```
    foreach (FileInfo folder in folders)
    {
        // Eliminates the pesky .meta extension
        string filepath = folder.FullName.Substring(0, folder.FullName.Length - 5);
        string midipath = filepath + @"\notes.mid";
        string guitarpath = filepath + @"\guitar.ogg";
        string namepath = filepath + @"\song.ini";

        // Ensures that we only instantiate the button for the song folder
            if it can be loaded by the Guitar Tutor
        // i.e. the folder has a midi file, a guitar sound layer,
            and a list of the song's name and artist
        if (File.Exists(midipath) && File.Exists(guitarpath) && File.Exists(namepath))
        {
            string foldername = folder.Name.Substring(0, folder.Name.Length - 5);
            Debug.Log(foldername);
            GameObject songitem = Instantiate(buttonexample,
             new Vector3(buttonexample.transform.position.x, buttonexample.transform.position.y,
             buttonexample.transform.position.z), Quaternion.identity);
            songitem.transform.parent = content.transform;

            Text songname = songitem.transform.Find("SongText").gameObject.GetComponent<Text>();
            songname.text = foldername;
            SongPreview songpreview = songitem.GetComponent<SongPreview>();
            songpreview.filepath = @"Assets\Songs\" + foldername;

            // Ensures that the tutor has at least one song loaded
            // If Play is clicked before a song is selected,
            // the MasterController will default to the first song on the list
            if (first)
            {
                first = false;
                if (Application.isEditor)
                {
                    int idx = filepath.IndexOf(@"Assets");
                    if (idx != -1)
                    {
                        filepath = filepath.Substring(idx, filepath.Length - idx);
                    }
                }
                MasterController.absolutePath = filepath;
                MasterController.rightHand = rightHand;
            }
            string albumpath = filepath + @"\album.png";
            if (File.Exists(albumpath))
            {
                Texture2D tex = new Texture2D(200, 200);
                byte[] albumData = File.ReadAllBytes(albumpath);
                tex.LoadImage(albumData);
                songpreview.album = Sprite.Create(tex, new Rect(0.0f, 0.0f,
                    tex.width, tex.height), new Vector2(0.5f, 0.5f), 100.0f);

                Debug.Log("Found Album for " + foldername);
            }
            Debug.Log(albumpath);
            string previewpath = filepath + @"\preview.ogg";
            if (File.Exists(previewpath))
            {
                StartCoroutine(LoadAudioPreview(previewpath, songpreview));
                Debug.Log("Found Preview for " + foldername);
            }
            Debug.Log(previewpath);
        }
    }
    Destroy(buttonexample);
}

IEnumerator LoadAudioPreview(string path, SongPreview preview)
{
    // Similar code used in MusicLayers.cs
    // Used to get the sound file from the folder
    string url = string.Format("file://{0}", path);
    Debug.Log(url);
```

```
        using (UnityWebRequest www = UnityWebRequestMultimedia.GetAudioClip(url, AudioType.OGGVORBIS))
        {
            yield return www.SendWebRequest();

            if (www.isNetworkError)
            {
                Debug.Log(www.error);
            }
            else
            {
                preview.preview = DownloadHandlerAudioClip.GetContent(www);
            }
        }
    }
    public void SwapHands()
    {
        rightHand = !rightHand;
        MasterController.rightHand = rightHand;

        if(rightHand)
        {
            GameObject.Find("HandButtonText").GetComponent<Text>().text = "Right Hand";
        }
        else
        {
            GameObject.Find("HandButtonText").GetComponent<Text>().text = "Left Hand";
        }
    }
    // Update is called once per frame
    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            //Debug.Log(MasterController.absolutePath);
            Debug.Log(MasterController.rightHand);
        }
    }
}
```

A.H) Citations

[1] Lin, John & Wu, Ying & Huang, T.S.. (2000). Modeling the constraints of human hand motion. Proceedings Workshop on Human Motion, Austin, Texas, USA, pp. 121-126 (2000).

[2] Albrecht, Irene & Haber, Jörg & Seidel, Hans-Peter & Breen, David & Lin, Ming. (2003). Construction and Animation of Anatomically Based Human Hand Models. ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SIGGRAPH-SCA-03), ACM, 98-109,368 (2003).

[3] Lin, Chih-Chun & Liu, Damon. (2006). An intelligent virtual piano tutor. Proceedings - VRCIA 2006: ACM International Conference on Virtual Reality Continuum and its Applications. 353-356. 10.1145/1128923.1128986.

[4] Yamamoto, Kazuki & Ueda, Etsuko & Suenaga, Tsuyoshi & Takemura, Kentaro & Takamatsu, Jun & Ogasawara, Tsukasa. (2010). Generating natural hand motion in playing a piano. 3513-3518. 10.1109/IROS.2010.5650193.

[5] Zhu, Yuanfeng & Ramakrishnan, Ajay & Hamann, Bernd & Neff, Michael. (2013). A system for automatic animation of piano performances. Computer Animation and Virtual Worlds. 24. 10.1002/cav.1477.

[6] ElKoura, George & Singh, Karan. (2003). Handrix: Animating the human hand. Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation. 110-119.

[7] Kim, J. & Cordier, F. & Magnenat-Thalmann, N.. (2000). "Neural network-based violinist's hand animation," Proceedings Computer Graphics International 2000, Geneva, Switzerland, pp. 37-41.

[8] Yin, Jun & Wang, Ye & Hsu, David. (2005). Digital violin tutor: an integrated system for beginning violin learners. In Proceedings of the 13th annual ACM international conference on Multimedia (MULTIMEDIA '05). Association for Computing Machinery, New York, NY, USA, 976–985. DOI:https://doi.org/10.1145/1101149.1101353

[9] Songsterr, an online guitar tab player and song database (2008): https://www.songsterr.com/

[10] Clone Hero, a free rhythm-based video game inspired by the Guitar Hero franchise (2017): https://clonehero.net/

[11] A TalkBass community forum discussing the layout of songs taken from Guitar Hero and Rock Band in order to add custom songs (2018): https://www.talkbass.com/threads/getting-multitracks-isolated-bass-recordings-to-create-bass-transcriptions.1372004/

[12] Unity, an open game development engine (2005): https://unity.com/

[13] Clone Hero's community song database, for easy access to thousands of ready-made songs available to play in-game (2017): https://docs.google.com/spreadsheets/d/13B823ukxdVMocowo1s5XnT3tzciOfruhUVePENKc01o/htmlview?pru=AAABcl1yAUk*0cH-QWcp02bDAAiLAaNP2A#gid=0

[14] Yuan, Shanxin & Ye, Qi & Stenger, Bjorn & Jain, Siddhant & Kim, Tae-Kyun. (2017). BigHand2.2M Benchmark: Hand Pose Dataset and State of the Art Analysis. 2605-2613. 10.1109/CVPR.2017.279.

[15] Manus VR, a Dutch company that specialises in motion capture hardware and technology (2016): https://www.manus-vr.com/

[16] MotionBuilder, a 3D animation recording and editing software by AutoDesk (2017): https://www.autodesk.com/education/free-software/motionbuilder

[17] Scene Motion Capture, a free Unity asset and extension to Animation Baker for recording animations within the Unity scene editor (2014): https://assetstore.unity.com/packages/tools/animation/scene-motion-capture-19622

[18] Animation Baker, a free Unity asset for managing animations recorded outside Unity and converting them to compatible formats (2014): https://assetstore.unity.com/packages/tools/animation/animation-baker-18217

[19] DryWetMIDI, a .NET library for C# for reading, writing and editing MIDI files (2017): https://melanchall.github.io/drywetmidi/index.html

[20] A Reddit thread on r/guitarlessons for how to convert the notes played on piano keys to guitar fret positions (2019): https://www.reddit.com/r/guitarlessons/comments/cp7dg5/guitar_fretboard_octavesinrelationtopianooctaves/

[21] A guide for how to measure fret distances when designing a guitar (2000): https://www.buildyourguitar.com/resources/tips/fretdist.htm

[22] Unity scripting API, which contains thousands of example coding structures for the functions featured within the official Unity libraries (2014): https://docs.unity3d.com/ScriptReference/

[23] Unity community questions and answers forums (2017): https://answers.unity.com/index.html