Virtual Musical Performance Venue

Eva Leonard

B.A.(Mod.) Computer Science

Final Year Project April 2014
Supervisor: Dr. Mike Brady

School of Computer Science and Statistics

O’Reilly Institute, Trinity College, Dublin 2, Ireland
Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

______________________________  ______________________
Name                        Date
Acknowledgements

I would like to thank my supervisor, Dr. Brady, for being a huge source of assistance and wisdom for the duration of this project.

I would also like to thank my parents and my boyfriend, Gareth, for constantly supporting and encouraging me for the entirety of my degree.
Abstract

This report outlines the processes and technologies used to create a virtual concert experience. The virtual concert experience comprises of a virtual reality perspective into a live musical performance. This experience aims to be as realistic as possible with much attention to detail such as humanoid avatars, audio panning, user manoeuvrability and realistic lighting. The program accepts a MIDI file, splits the file into its separate streams, converts those streams to .wav files, and then plays them in precise coordinates in the 3D environment. Through much research, I believe this to be a rather individual program, as this exact process of splitting MIDIs, converting them to .wavs and then playing them in a 3D environment seems to never have been done before.

The aim of this program is to assist those in a choir or orchestra, to help them practice pieces in their own time in a realistic environment. It can also be used to preview a concert or performance before it is held live.
**Introduction**

This chapter will outline the aims, motivations and core objectives of the project.

This project presents a virtual concert experience. It involves the user moving around a virtual 3D space containing avatars which producing harmonic sound. The program begins by reading an input MIDI file. The MIDI is split into its individual tracks, which are converted to .wav files, which in turn play out of the 3D avatars in the virtual space.

**Aims**

The aim of this project is to create a virtual concert experience. This would involve a 3D environment that the user can navigate. In this 3D environment would be a stage with avatars standing on it. There would be streams of music coming from the avatars. The separate streams would be separate lines of music - separate alto, tenor, soprano and alto lines for example. As the user moves through the scene the music coming from the avatars would hold its coordinates in the 3D environment. As the user turns, the music would move from one speaker to another, giving the illusion of static sound. A real 3D experience.

**Motivations**

Anyone in a choir or orchestra will tell you there is never enough time to practice. By the time everyone arrives, gets there music, stands in place and everything is set up, there is not much time to run over material. It is also difficult to organise a time when everyone is free. Because of this practice time is extremely valuable.
It is possible for a choir or orchestra member to practice alone at home, but it is never the same as practicing as a group. It is much harder to figure out timing, entry points, and dynamics. Being able to practice with a CD is all well and good, but it is very difficult to figure out exactly what your part is doing, be it alto, violin 1, etc. Another problem is positioning. Say for example you are playing a French horn in an orchestra. The violin part may be very loud on the CD, but when you are sitting in formation, the violins are not near you and therefore very quiet. The oboe that you sit beside and coordinate closely with is difficult to hear on the CD and hard to play along with.

A choir or orchestra is not seen to be made up of different people. It is seen as a unit, one being. If a single alto or cello is out of sync everything is out of sync. It imperative that everyone knows what they are doing and when they are doing it. That’s where the virtual musical performance venue comes in.

Another use for the program is concert organisers. The organisation of a concert is a huge process. It is expensive and time consuming. One area for consideration is how the music will sound with the performers set up in a certain formation. To get the performers on stage and moving around is, again, time consuming. Instead, the user can input the song that will be played, and have a preview of how it will sound.

**Research Questions and Objectives**

There were many parts to this project that needed to be thoroughly researched. The main question was whether MIDI could be split up into its individual tracks and played separately. There was also the question of whether the split MIDI streams could then be played from certain coordinates in a 3D environment.
The primary objective of this project was to split the MIDI files, feed them into a 3D coordinate system and then have them play in that 3D space in a way that is similar to reality, i.e. with static audio placement and stereo panning.

**Background**

This chapter is where we discuss the conception, thoughts and an introduction to the technologies behind the virtual musical performance venue.

The idea for this final year project came about when the supervisor and the author brainstormed ideas. The topic of music, specifically choirs, was a particular area of interest. It was very appealing to the author having been very interested in music since a young age. The author has played the violin since aged four and has been in many choirs and orchestras, so it was decided that this project was a good fit.

It was decided that the project would be a virtual concert experience. The user would be able to manoeuvre around a 3D environment. There would be several avatars placed around this 3D environment. The avatars would emit sound. The sound coming from the avatars would keep their position in the 3D space as the user moves around. This is a concept called stereo panning. The basic idea of stereo panning is that a sound being played changes volume and intensity between speakers on a listening device such as a laptop or a stereo system. This change gives the illusion to the user that the sound is moving around them. It is a common practice used by many musicians.
It was then decided that the sound that should be emitted by the avatars be separate tracks of a MIDI file. This would give the experience a more "choir-like" feel. For example, if the avatars were to emit the separate tracks of "Messiah" by Handel, the separate streams of sound would include the soprano, alto, tenor and bass lines of a choir. In this way, the experience would feel more like a live concert performance.
Technologies Involved

**MIDI Files**

MIDI stands for “Musical Instrument Digital Interface”. MIDIs are based on Local Area Networks, in a loose sense, based on there being multiple local devices communicating with one another and that they are easy to set up and run, while remaining relatively cheap (Swift, n.d.). MIDI files are quite unusual, in that although they play “music”, they are not actually comprised of musical sound bites. They are instead a series of messages or instructions (amandaghassaei, n.d.). These instructions include “note on”, “note off” and other instructions that control pitch, time signature and dynamics. MIDI is very useful for people who do not own instruments but want to mimic their sounds and use them in compositions. As setting up a MIDI home studio is fairly simple and inexpensive, they are very popular. MIDI is what kick-started the era of modern dance.

A MIDI is generally created by using a special piece of hardware on an instrument. Most of the time the instrument used is an electric keyboard. The keys pressed are changed into MIDI signals and are then sent to an appropriate playback device.
There are some incorrect preconceptions about MIDIs. They do not communicate audio, nor can they recreate sound. It is a digital language that instructs a device or program to playback and control audio. An example of this would be if I asked you to sing a C#. I am not creating the sound, I am merely communicating with one which can create this sound.

When a MIDI “note on” message is created, one of two things are represented. First is the note. This is represented by a value between 0 and 127. This value corresponds to a musical note. For example, middle C is represented by the value 60. The second thing represented is the velocity. This is again represented by a number between 0 and 127. The velocity is a measure of the volume of the message. Therefore the velocity is MIDIs way of representing dynamics. For example, a velocity of 30 would be described as “piano” in musical dynamics terms.
MIDI can be comprised of many tracks of instructions being sent to the playback device concurrently. Therefore we may have an audio stream which represents a guitar and one that represents a singer playing in harmony. The data in the MIDI file is comprised of a header chunk. This is the first chunk of information. This contains information about the entire song, including the key, number of tracks, and time signature (Sonic Spot, n.d.).

Next are the track chunks. These contain information about the individual tracks, including the instrument name, and the musical events such as ornamentation and dynamics (Sonic Spot, n.d.). These individual MIDI tracks, of which there may be many, were what the application processed and split into individual MIDI sources.

The MIDI track events are used to illustrate the different changes in the MIDI tracks. These include an “aftertouch event” which describes when a musician has changed the amount of pressure placed on an instrument’s keys or strings, and a “pan event” which indicates that the track induces audio panning - where the instrument’s sounds to be moving around the listener.

**OpenGL**

Based on the author’s past experience with the technology, it was decided that the OpenGL API would be used to create the visual 3D environment.

OpenGL, created in 1992, is the most popular and widely used graphics API in the graphics programming industry (OpenGL.org, n.d.) It is commonly used for entertainment, medical imaging and virtual reality. Though bindings exist for other languages, the main language used when writing OpenGL code is C++, which is also the language used in this project.
**Core Audio**

All of the audio played and used on the Mac or iPhone OS is powered by Core Audio. The API is exposed via the C programming language, meaning that it is available in C++ as well as Objective-C. The project supervisor had experience with using Core Audio and felt that it would be a strong choice for the audio processing and playback aspects of the project. An important area that Core Audio provides is the OpenAL library.

**OpenAL**

OpenAL is the audio counterpart to OpenGL. It shares many similarities with OpenGL, such as the fact that it is accessed via a C based API. The API itself closely shares the programming style of that of OpenGL, and it too utilises the concept of a virtual 3D space. Because they are so similar it is very easy to implement both APIs side by side.
**Blender**

Blender is a suite of tools for 3D modelling. In Blender, an user can created 3D objects, shapes and even complex character and environmental models for virtual reality and video games. Blender models can be exported and used in the virtual 3D environments of OpenGL.

![Audio Source Model created in Blender](image)

**MAC Developer OpenAL Example**

The project supervisor found a relevant example of OpenAL on the MAC Developer Centre (Apple Inc., 2012). This example had a similar goal in mind as that of the project. This example involved a 2D virtual space. In this space were several sound sources in the far corners of the screen. The user was represented by a moveable avatar which initially started in the centre of the screen. This avatar was moveable by dragging it with the mouse. When the avatar was moved closer to one sound source, the others would sound softer, and the one it was being dragged towards sounded louder.
As well as this, the user could adjust the sound levels of the sound sources. When the user was being dragged around the screen, the sound sources would also move using stereo panning in accordance to where they were on the screen. For example if the user was dragged from the left to the right of sound source A, the sound that sound source A was emitting would move from the right to the left speaker on the user’s laptop.

This was a very good starting point for the project. It encapsulated the idea of the sounds coming from different sources in a 3D space, and using audio panning in accordance to their and the user’s positions.
Technical Overview

The aim of this project is to take the reader through the steps of the program, explaining what they do and how they work.

**MIDI Splitting**

MIDI files can comprise of a number of separate tracks. These tracks can be compared to different orchestral parts on sheet music, i.e. violin 1, violin 2, piano etc. The first step of this program was to split the single MIDI file into a number of MIDI files comprised of one track each. In other words, the original MIDI file, which would feature a number of tracks, the orchestra file if you will, would be split into its individual parts - the parts of each of the instruments playing in the original track. These tracks needed to be split up so that they could later be played from the individual avatars in the 3D environment.

**MIDI Padding**

These individual MIDI track files had a problem. Say if one instrument didn’t begin to play until the chorus of the song, it’s length would be much shorter than that of an instrument that was playing since the beginning of the song. Because of this, some of the tracks had to be padded with varying amounts of silence by means of adding a ‘silent note’ to the beginning of the track. This would cause the tracks to all begin at the same time, even though some of them would just be playing silence. This kept them all in sync.

**MIDI Conversion**

OpenAL has one major flaw in relation to this project. It is unable to play MIDI files. This is because MIDI files are not technically audio files. They are comprised of instructions which can be interpreted to a playback device which then emits sound. Because of this, the MIDI files had to be converted to an audio file type which could be played by OpenAL. In this case, .wav was chosen.
**3D Environment**
The 3D environment had to have a number of properties. These included a way for the user to move around in it, a way to include avatars, and a way for these converted files to be played into the 3 space.

**Sound in 3D Environment**
The converted files needed to be played in our 3D environment. They couldn’t just be played in any old way, however. They needed to be played from certain 3D coordinates. They also needed to hold their position in this 3D coordinate system as the user moved around in it.
Implementation

In this section, the technologies and methods in which the project was implemented are explained.

The splitting of the MIDI tracks was to be the main and most important part of the project. If the MIDI tracks could not be split, the project would be unfeasible. It was key to have the track split up correctly - with each part, be it singing line or instrument, playing individually from its own sound source. This is what would result in a more realistic 3D experience.

The original desired result for this section of the project would be as follows: the user would input the location of a MIDI file via command line. The project would then fetch this MIDI, read the instructions, and then split the MIDI into multiple MIDIs comprising of its separated tracks. These separated MIDI files would then be passed onto the next stage of the pipeline.

The splitting functionality was achieved through the usage of a third party open source library called JDKSMidi, which is publicly available on Github and released under the GNU General Public Licence.

Once given an input stream representing a MIDI file on disk, JDKSMidi provides a clean API for parsing and manipulating the MIDI data at the track level, as well as at the individual message level. A filename or path to an existing MIDI that the user wishes to use in the simulation is provided on the command line upon program execution. This is
first checked to ensure it points to a valid file on disk and once that has been confirmed, it is loaded and fed into an instance of the **MidiProcessor** class.

**The MidiProcessor Class**

This class provides all of the logic for the processing of our MIDI files, namely track splitting via the `splitTracks()` function, and MIDI to WAV conversion of these tracks via the `convertTracks()` function. All that is required is to provide a valid filename or path for a MIDI file to the constructor, then calling both of those functions in order. Any errors occurring along the way are exposed via C++ exception handling, and the functions will throw a `std::runtime_exception` containing a message explaining what went wrong, accessible via the `what()` function.

Internally, a **MidiProcessor** instance wraps a JDKSMidi `MIDIFileReadStreamFile` object which it constructs from the input file path fed to its constructor. It provides a sanity check via the `isValid()` function, delegating to the function of the same name of its `MIDIFileReadStreamFile`. This function should be called after construction to ensure that the MIDI file you are wrapping both exists and that the program has access to it.

**Track Splitting**

Once the **MidiProcessor** instance has been initialised, the next step is to perform the splitting of the input MIDI file into individual MIDI files each containing a single track from the original. This is where the functionality of the JDKSMidi library comes into play.
All track splitting logic is contained in the aptly-named `splitTracks()` function. After performing an initial sanity check via a call to `isValid()`, the function then proceeds to create several important objects in preparation for parsing and splitting the input file. The most important of these is the `MidiFileRead` class, which is the top-level object used for reading and parsing the actual MIDI data. The number of tracks from the input is first read in order to prepare the objects which will contain the actual track data, and then the input file is parsed with a call to `MidiFileRead::Parse()`.

A reference to the first track in the input is stored, as this track is special in that it contains the tempo and timing signature information for the entire MIDI sequence. When creating the individual track files, this information needs to be copied into each of them to ensure they are played back in the correct way and accurately reflect their source.

The first step in writing the individual track files is to generate an appropriate filename for the new MIDI. The `MidiProcessor::getFilenameForTrack()` function is a simple helper function to construct a filename of the form “$INPUT_FILENAME” + “track” + tracknumber + “.mid”. This is used to construct a `MidiFileWriteStreamFileName` object, which in turn is wrapped in a `MidiFileWrite` object that performs the actual writing of MIDI messages to the output file. Once these have been instantiated the MIDI file header is written, which specifies the MIDI format, number of tracks in the file, and the time division used in the MIDI. All tracks output by this program are “format 1”, contain a single track and use the same time division as the input. The individual track header is then written which specifies the duration of the track.
Once both headers have been written, the process of looping through each MIDI message in the source track begins. For the most part this is a relatively straightforward process of iteration followed by writing a copy of the message to the output file, however there are two notable pieces of logic worth mentioning. The first is the logic that handles writing the tempo and time signature messages found in the first track of the source file. An index into this first track is maintained, and whenever the next message in the current track is reached that occurs at a time later than the message currently pointed to in the first track, that first track message is written to the output file at the current timestamp. In this manner, the original order and location in the track for the timing messages is maintained.

The second notable piece of logic in the message writing iteration is that which deals with creating a ‘silent note’ at the beginning of each track. This is necessary in order to get around the issue of a MIDI file beginning to play at the first instance of a note being played in the sequence, ignoring any delta time between the actual start of the track and the first note being played. This is a quirk in the MIDI specification that causes a problem when splitting the tracks into individual files, as one would need to keep track of timings for when each track should start playing rather than being able to play each track at the same time, and simply having a leading silence for tracks whose notes don’t occur until some time delta into the sequence.

The ‘silent note’ is the solution presented in this project to get around the aforementioned behaviour. Immediately before reaching a MIDI message with a non-zero time delta, if we haven’t seen a note message before this point, we write a ‘note on’ followed immediately by an equivalent ‘note off’ in order to trick the MIDI playback into containing the appropriate leading silence.
This results in each instrument entering the piece at the correct time during playback as individual tracks. Of course, this ‘silent note’ isn’t necessary if the track contains a note at a zero time-delta, so we skip it if this is the case.

Once all MIDI messages have been written to the output stream, we write the end of track message and rewrite the track length in the track header now that we know exactly how long it is. The output filename is written to a `std::vector` of `std::strings` so we can keep track of them for the next stage in the pipeline, MIDI to WAV conversion.

**MIDI to WAV format conversion with CoreAudio**

Once the input file has been split into its constituent tracks, the next step in the process is to convert them from the MIDI format into a format that OpenAL can process. OpenAL lacks the ability to play MIDIs since they don’t contain raw audio data as typical audio files do, such as MP3 or WAV. To convert them to a compatible format we make use of several CoreAudio APIs. The majority of the code for performing this processing was adapted from an example program on the Apple Developer site, and many provided utility classes were also used. These utility classes exist inside the `PublicUtility` subfolder.

The conversion process begins when the `convertTracks()` function is called, which requires you to have previously called `splitTracks()` in order to have prepared the individual track files for processing. This is explicitly checked for by checking whether or not we have stored any track filenames, and if not, a `std::runtime_error` is thrown. If there are existing filenames to process, we simply iterate through the vector and call the `convertTrack()` function for each one. This is where the actual conversion process takes place.
The first step in the audio conversion process is to create and initialize an instance of the `MusicSequence` class. This class represents one or more music tracks, and can be played using a music player. A simple helper function, `MidiProcessor::LoadMusicSequence()` was adapted from the Apple sample source code for the purpose of instantiating the `MusicSequence` and loading the target input MIDI into it, achieved via calls to the CoreAudio APIs `NewMusicSequence` and `MusicSequenceFileLoad`, respectively. The call to `LoadMusicSequence` is wrapped in a pre-processor macro, `FailIf`, which is used extensively throughout the CoreAudio-related code. It is defined in `PublicUtility/CADebugMacros.h` and is used as a guard for automatically checking the return codes of the CoreAudio APIs. In the case of a failure code being returned, the `FailIf` macro jumps to the supplied label in the code and returns an error string. Using the `goto` statement in C++ is generally frowned upon in modern coding conventions, however as it was given as an example in the official Apple SDK documentation I felt it pertinent to use it as demonstrated.

Once the `MusicSequence` object has been prepared, the next step is to get the `AUGraph` associated with it. An `AUGraph` is one of several key types in the CoreAudio architecture, representing an Audio Processing Graph. The graphs are comprised of several `AUNodes`, representing individual Audio Units or sub-graphs. An Audio Unit can be thought of as a "black box" of sorts, that takes an input audio signal and manipulates it in some way. The audio stream passes through the various nodes and sub-graphs in our main `AUGraph` and ends up rendered to a file on-disk.
The **AUGraph** is extracted and prepared from the **MusicSequence** via the **MusicSequenceGetAUGraph** and **AUGraphOpen** APIs. It is then necessary to extract a particular type of **AudioUnit** from the graph, the one which is responsible for synthesising the MIDI events to actual audio. In order to do this, another helper function **MidiProcessor::GetSynthFromGraph()** is used, which iterates through the **AUNodes** in the input graph and creates an **AudioComponentDescription** object for each one. This **AudioComponentDescription** contains a **componentType** field which we examine, looking for an **AUNode** of type **kAudioUnitType_MusicDevice**. Once we have found this node, we assign it to a reference passed in to the **GetSynthFromGraph** function and return a success code.

Now that we have found the **MusicDevice** node, synth, we need to set some properties on it so that it is in the correct state for converting our MIDI and knows it will be rendering to a file rather than to an audio output device such as speakers or headphones. We first set the **kAudioUnitProperty_CPULoad** property to ensure the synth doesn't consume too much CPU power while rendering, giving it a limit of 80% CPU usage. In reality this level of usage may never occur but it is good practice to have the safeguard in place. The **kAudioUnitProperty_OfflineRender** property is then set to 1 on the synth, which tells it that it will be rendering to a file. This allows it to render at a greater than 100% playback rate, otherwise the user would have to wait for the duration of their input MIDI before conversion would be complete.
After setting up the synth node, it is necessary to make some preparations to the AUGraph itself in advance of playing back our MIDI. Again, a helper function MidiProcessor::SetUpGraph() was used to perform these configuration actions. The configuration required of the AUGraph comprises of two actions:

1. Locate the AudioUnit of type kAudioUnitType_Output, set its SubType to kAudioUnitSubType_GenericOutput, and connect all other non-MusicDevice nodes to this output node.

2. Set both the kAudioUnitProperty_SampleRate as well as the kAudioUnitProperty_MaximumFramesPerSlice properties on each node in the graph. The former dictates the sample rate used when rendering the MIDI audio, which is set to 16000Hz for all MIDI files and tracks, and the latter dictates the maximum number of audio frames an AudioUnit will be required to render per render call. This is set to 512 for each node in this project.

The next step once the AUGraph has been configured is to create a MusicPlayer object. This is the object that handles the playing of the tracks in the MusicSequence. Initialization of the MusicPlayer is done via the NewMusicPlayer() function, followed by setting the sequence the MusicPlayer will use via the MusicPlayerSetSequence() function. Provided these operations succeed, a reference to the MIDI track represented by a MusicTrack object is obtained using the MusicSequenceGetIndTrack() function. The sequence's length is queried from the MusicTrack using MusicTrackGetProperty() while specifying the kMusicTrackProperty_TrackLength enum, which will be used shortly when writing the output file. Some preparation of the MusicSequence is initiated via calls to MusicPlayerSetTime() to set the time to the beginning of the sequence, and MusicPlayerPreroll() to perform some pre-playing setup.
Not calling this function results in some minor latency before playback occurs after being initiated. Finally, a call to `MusicPlayerStart()` tells the `MusicPlayer` to begin playing.

The last step in the conversion process is to actually write out the converted audio to a file on disk. An output filename is constructed by replacing the input MIDI track filename's `.mid` extension with `.wav`, and one final helper function performs the heavy lifting of writing the output, `MidiProcessor::WriteConvertedOutputFile()`. An important thing to note is that this function is passed an `OSType` variable specifying the output data format. For this project, the LPCM format is specified, which is a standard audio modulation format for `.WAV` files. To begin with, various output format parameters are specified by creating an instance of a `CAStreamBasicDescription` class and setting the various required fields therein. Information such as the number of channels per frame, the sample rate, the data format and the bits per frame are all specified here.

Next, the output file is created on disk with a call to `ExtAudioFileCreateWithURL`, passing a `CFURLRef` created from the output file path, along with the `CAStreamBasicDescription`. Once that is done, the `AUNodes` in the `AUGraph` are once more iterated over to find the `kAudioUnitType_Output` node. The output file is informed of the source stream's data format by retrieving the `kAudioUnitProperty_StreamFormat` from the output audio unit and using its value to set the `kExtAudioFileProperty_ClientDataFormat` property. This ensures that the output file is correctly prepared to receive the audio data in a format it expects.
An instance of the utility class AUOutputBL is created to serve as the buffer for receiving data rendered from the output AudioUnit and to serve the same data to the output file. It essentially wraps an AudioBufferList and transparently allocates and deallocates the memory needed for using it. This AudioBufferList is accessed with a call to AUOutputBL::ABL(). Once this has been instantiated, the actual rendering and output loop begins:

1. The AUOutputBL::Prepare() function is called to ensure the buffer has space ready to receive the rendered audio data.

2. AudioUnitRender() is called to get the rendered audio data and store it in the buffer.

3. A timestamp is updated to keep track of how far in the sequence the rendering process has progressed. This is fed to the AudioUnitRender() call in the next loop iteration.

4. The current iteration's data is written to the output file via ExtAudioFileWrite().

5. The loop is terminated once the music player reports that the current time in the sequence is greater than the sequence length.

At this point, the conversion is complete, and all that is left for the WriteConvertedOutputFile() function to do is to dispose the output file handle with a call to ExtAudioFileDispose(). Once control returns to convertTrack(), it too disposes of the various resources it has created, such as the MusicPlayer and MusicSequence. It then pushes the output filename to another std::vector of std::strings, as convenience for the OpenAL initialization later in the program.
**3D Environment**

The 3D environment was rendered using OpenGL. OpenGL is the popular graphics API discussed earlier. The code for rendering the scene was based on and extended from an OpenGL project written as part of a 3D graphics module for beginners earlier in the year.

The two helper libraries GLFW and GLEW were used to assist with ensuring the OpenGL functions were loaded and ready to be used, as well as allowing for easier creation of an OpenGL context and a window to render into.

The scene makes use of a single vertex and fragment shader, neither of which perform any special calculations. The only thing of note is that the vertex shader takes a three-dimensional vector as a uniform input to specify the colour in which it should render a given model. This is used to render each ‘audio source’ in the scene in either red, green, blue or yellow.

The GLFW library, as well as making it easy to create a native window, also provides a nice abstraction on top of common input methods. The program tracks the inputs of various keys in order to control the motion of the camera throughout the scene.

The avatars in the 3D environment were made in Blender. It was decided that Blender was a good program to use as it works well with OpenGL. The avatars were given a human-like appearance to make the virtual experience seem more authentic. The models were made out of simple shapes and then manipulated. Their size, angle, positioning and proportions were manipulated using Blender. They were then exported as .blend files. These files were then loaded into OpenGL and given the specific coordinates of the sound being emitted by OpenAL. This gave the appearance of the avatars emitting the sounds.
Figure 2 - Scene featuring multiple audio sources constructed from individual MIDI tracks
Conclusions
This section will outline the author’s evaluation of the end result of the project, the difficulties encountered throughout, and any possible future work.

Evaluations
The original aim for this project was to create a virtual simulated concert experience. It would involve taking an input MIDI and splitting it into its individual tracks. These tracks would be then played in surround sound around the user. The main proof of concept was to see if MIDIs could be split into their individual tracks and then played into a 3D environment.

Initially, this seemed like a very daunting task. There were a lot of features and many places that could later be discovered to be unviable. I was worried that there would be little support and information regarding MIDIs as they are now fairly dated, nor had I heard of any tools that used MIDIs in the way that this project required. Upon researching the subject, I found there to be multiple sources on MIDIs but a lot of them were quite antiquated and I was not sure if they were still applicable to modern techniques.

I was pleased to find that OpenGL and its audio counterpart OpenAL were both very well suited for use in this project, as I had used OpenGL before and knew OpenAL was very similar.

I feel as though the end goal and proof of concept were achieved. The project does prove that MIDIs can be split into their streams and played in a 3D environment. It also features these sound sources displaying realistic surround sound qualities.
**Difficulties Faced**

The biggest difficulty I faced was programming on a Mac. I had never used a Mac before. I found it difficult to use XCode as a development environment and it took me quite a while to get used to it. I also had difficulties implementing OpenAL and OpenGL on a Mac. The hardest part was including the libraries. They were very similar to those on a Windows machine, but subtly different – enough to trip me up every now and then and leave me stumped. Having said this, the Core Audio libraries, which are only available on Mac, were invaluable and I don’t think that the project could have developed as far as it has in the time it has without them.

When I began the project, I didn’t realise that OpenAL could not handle MIDIs. This threw a bit of a spanner in the works as I had not anticipated this in my research and planning. This created what possibly turned out to be the biggest work item in the project – the conversion of MIDI to .wav format, since there is no viable alternative to OpenAL which could also handle MIDIs. This demonstrates the problem of scope creep.

When I first tried to implement the MIDI to .wav conversion, I did so using a set of APIs that, though worked for a wide variety of audio formats, turned out not to work for MIDI. This is because MIDI is not a classic waveform audio file like mp3 or wav.

Another problem faced during the early stages of this implementation was finding an external command line tool that was able to perform the MIDI to .wav conversion. When discussed with my supervisor, however, we decided that this was not the best course of action, and that it would be far superior if the entire pipeline could be performed internal to the project.
**Future Work**

This section details several potential options for future development of this project that were either deemed out of scope or were conceptualised during the implementation stage.

The original plan for this project was that it would be best used by someone in a band, orchestra or choir. To expand on this, a good development in this area would be the manipulation of the avatars and the sound sources. For those wanting to practice a piece involving other group members, it would be very useful to be able to move them into certain formations. Orchestras and choirs can have varying setups, so it would be beneficial to coordinate the group to the user’s liking. This would also help choirmaster and conductors to decide where certain instruments may be situated for a given performance.

Adding on to this, the idea for changing a sound source’s volume was conceived. This would enable users to practice along with different streams of audio with every run of the program. This feature would also be useful for those who wish to hear their own music played in this fashion. They could analyse a piece and turn up or down parts to their liking, to find the best volumes for each instrument or voice line.

Another way to make this project more user friendly would be to apply realistic textures and lighting to the project. The use of textures would create a better sense of realism, as would the lighting. They would make the virtual environment add to a more believable experience.

Recordings of concerts and full length films based around concerts are very popular. The virtual musical performance venue could be used in an attempt to emulate these experiences. The playlist for a concert could be loaded into the program and each piece could be played one after another to get a more realistic reproduction of a certain performance.
There has been a recent surge in the number of areas using virtual reality. These include home entertainment, the medical industry, the transport industry, the military, and various space programs. There have also been a number of companies creating virtual reality devices which are suitable for home use. The upcoming product most relevant to this project would be the Oculus Rift. It is a virtual reality headset that will enable users to experience a level of immersion never before thought possible (Oculus Rift, n.d.).

The virtual musical performance venue could be integrated with the Oculus Rift. This would be more targeted at those who want to experience a show rather than those who want to practice. The Oculus Rift would enable the user to reach a whole new level of captivation that would involve their eyes as well as their ears.

For those who play larger instruments, such as the drums, the double bass and the tuba, it can be difficult to play along with the virtual performance venue in its current state of development. Their headphones may not reach them as they need a lot of space between them and the computer, they may find it bothersome to keep getting in and out of position to interact with the keyboard, etc. It would be beneficial for these players to have a 3D speaker system set up in the room they are practicing in. This would be similar to the surround sound speakers generally featured in a home cinema. The user could sit in the middle of a set of speakers and move themselves around the room depending on which sound source they wanted to be closer to.

Another area for development would be the ability to change the sound font each sound source uses. As previously mentioned, sound fonts are similar to text fonts in that they slightly change the way certain sound are projected. This would enable those who are practicing or those who wish to tweak their own music to hear the individual sound streams in slightly different ways.

These are just a few potential directions in which this project could be continued. There are doubtless many more viable areas for customisation catering to many different needs.
**Final Thoughts**

I believe that the project was a great success. Everything that I set out to achieve was completed. Although I did run into a few problems along the way, none of them were showstoppers and I managed to overcome all of them.

If I was to spend more time on the project, I would implement the moving and change of dynamics of the avatars and sound sources. I believe that this would really improve the project overall and make it much more attractive and helpful to those wanting to practice playing along with a song.

If I were to restart the project from scratch, knowing what I do now, I would build it with a nice interface from the start, as I believe that it is a bit rough around the edges.

Other than that I am delighted with how the project turned out.
Attached Electronic Sources and Instructions

Included on the attached DVD is the directory and file structure of this project's code only. The XCode project file used during development is also provided. Some configuration may be required in order to get it working on another machine, most likely the include file and lib file paths will need to be updated to reflect the location of the above libraries on your machine. Framework and subproject references will also potentially need updating in a similar manner.

The latest compiled binary is also provided though it is not guaranteed to work out of the box, and again may need the above libraries installed.

The binary was compiled for OSX 10.9 (Mountain Lion).

This project was developed using the following tools and libraries:

- GLEW 1.10 - http://glew.sourceforge.net/
- GLFW 3 - http://www.glfw.org/
- AssImp 3 - http://assimp.sourceforge.net/
- JDKSMidi - https://github.com/jdkoftinoff/jdksmidi
- OpenGL 3.2 - http://www.opengl.org/
- Mac OSX 10.9 (Mountain Lion)
The program should be launched given a single command line argument - a path to an input MIDI file you wish to process and use in the scene.

e.g.

`.OpenGLApp test.mid`

would run the app targeting a 'test.mid' file in the current directory.

Controls:

- W - Move Forward
- A - Turn Left
- S - Move Backwards
- D - Turn Right
- Esc - Exit
References

http://www.ias.uwe.ac.uk/~a-winfie/teach2003/midiseq2.jpg

http://www.sonicspot.com/guide/midifiles.html

amandaghassaei. (n.d.). *What is MIDI?* Retrieved from Instructables.com:
http://www.instructables.com/id/What-is-MIDI/

Apple Inc. (2012). *OpenALEExample*. Retrieved from Mac Developer Library:
n/Intro.html

Oculus Rift. (n.d.). *Oculus Rift*. Retrieved from Kickstarter.com:


http://www.doc.ic.ac.uk/~nd/surprise_97/journal/vol1/aps2/