Project 152

'Real-Time Audio and MIDI Control iPad Application with Backing Drum Generation

Student No: 1026943
Student Name: Steffan Walters
Supervisor: Prof. A.D.Marshall
Moderator: Dr K.Sidorov
Module Code: CM0343 - 40 credits

5/3/2013

Acknowledgments

I would like to express my personal thanks to my supervisor professor A.D.Marshall for giving up his time generously and providing help and guidance throughout my project from the planning stage up to the reporting stage.

I would also like to thank Regaljay Ltd. for providing me with the equipment necessary to complete this project.

Finally, I would like to express my thanks to Mr Matthew R Jones for taking part in the evaluation stage of my report and helping test the application.

Table of Contents

Table of Figures

Introduction

The initial description of my project was to create a tactile musical application for Apple's iPad device. My application emulates some of the features found on the Korg Kaoss pad as well as other similar applications such as Soundgrid and Beatwave to give real-time audio effect control. The application also allows for MIDI control and the creation of various drum tracks, using an easy to use interface. These features add a new level of uniqueness by allowing a user to input their own musical data into the application while also playing a drum track that they created. As usage of touch screen devices such as the iPad have increased, there is a good market for applications such as these, there is however, strong competition from other developers.

The interim report for this project contained all the details regarding my research on how to create the application and the research conducted into similar devices. Most of the applications that I had researched did either the adding of effects well or the drum creation aspect well and not many applications combined the two. This research helped me to begin the development of my application, which forms the basis for this report.

During the report, I will briefly discuss the design of the application, explain the main methods of my implementation and produce my findings after conducting an evaluation. There will also be small sections on future work and reflections of my own learning.

There have not been any major changes to the project from the initial description, the main change to the project is that the title has changed to 'Real-Time Audio and MIDI Control iPad Application with Backing Drum Generation' instead of 'Kaoss Pad Type iPad Application for Real-time Audio and MIDI Control' as was initially stated in the project description. The title has changed, as I have focused more on the backing drums and audio effects rather than trying to include some of the advanced features found on the Korg Kaoss Pad. There are however some features in my application similar to the Korg device. As discussed in the future work section, this project can be developed further to include other features included in the Korg device and other similar applications.

Below is a summary of the features included on my finished application:

- MIDI input via a MIDI keyboard
- MIDI control for the iPad
- Real-time audio controllable effects using different methods:
    - Buttons and sliders - Chord, Arpeggio, Diminish, Volume Envelope, Modulation and sound type effects
    - Gesture input such as 'two finger press' - Modulation effect
- A drum generator
    - Select a pattern

    - Change the tempo of a track
    - Change the volume of the drums
    - Save the track
- MIDI input with backing drum accompaniment

Design

Background Research Update/ Technologies Used

After conducting the research for the interim report, I had most of the information required to complete my application. Here, I have included the other research that I have conducted after the interim report.

Objective-C

To create the applications, I have used many of the aspects of the Objective-C programming language. The Core MIDI and AVFoundation frameworks have been used to allow MIDI input and the playing of drums respectively. I have also used a number of different Cocoa Touch objects, these include UIButtons, UILabels, UITableViews, UITextViews, UICollectionView, UISegmentedControls and UISliders to create the interfaces of my application.

Pure Data

To control the audio processing side of my application I have used the libpd version of Pure Data. This libpd version runs vanilla-PD and not the full extended-PD. The main Objective-C application communicates with libpd by sending parameters to receivers within the Pure Data patch. Each Pure Data patch can contain a number of sub-patches to further divide the processing of audio. For my application, I have designed three different synthesisers, these will be discussed in detail in the implementation section.

MIDI Input

As discussed in the interim report, there is a way of inputting MIDI data to the iPad using Wi-Fi, however, I did not have time to try and implement this feature and therefore I had to keep with the input of MIDI data through a connected keyboard.

Included Features

MIDI

In the MIDI section, I included several simple effects which include:

- Diminish
- Simple Arpeggiation
- Chord
- Sound Type

I have included these as most other synthesisers have these basic effects. Some were simple to implement but others (sound type/ arpeggiation) were more complex. I have also included a feature to allow the user to modulate their sound but using a touch pad, I thought this was a enjoyable way for the users to change the sound produced. Finally, as most other applications use knobs and sliders to control parameters, I decided this was the best way to change the parameters in my application as it is easy for a user to understand how to change them by using sliders.

To allow the user more flexibility, I decided upon not creating a built-in keyboard I find that it is much harder to play a keyboard that is on the iPad compared to a physical keyboard. Instead I allow them to connect a keyboard to the application.

Drum

I have implemented the ability to change the tempo of the track and volume of the drums within the application. During research, I didn't find these to be common amongst drum generators, but I think it allows for more customisability and could attract users to the application.

Design of Interface

As stated in the interim report, my application is built from two main sections which are the drum and MIDI sections. The design of the interface has changed greatly from the one proposed in the interim report due to the number of interface objects I had not thought about. I have tried to keep the interfaces as simplistic and easy to use as possible but I have not focused on making the application look good graphically. To allow navigation between the two sections of the application, I decided upon using a tab bar to separate each section and allow easy navigation between them. The saved drum tracks appear on the left hand side of each section with the same background colour so that a user can easily identify the features as being the same.

MIDI Section

The complete interface for the MIDI section is shown in figure 1 below . I have made use of an intuitive interface with the layout of it being split into four sections:

- Top Left - Sound type choice, keyboard set-up, record and playback
- Bottom Left - Drum track area (list of tracks, play, stop), debug text view
- Central - Effects area
- Right - Advanced Effect

Figure 1 - MIDI Interface

I split up the interface into the four sections so that a user will be able to learn how to use the interface quicker. To emulate certain aspects of the applications I had researched, I have used sliders to control the parameters of the effects and buttons to control whether an effect is on or not. The best aspects of the applications that I had researched into, was the way that they allowed parameters to be changed via different touch gestures (tap, swipe, pinch). I decided upon using a separate UIView to allow the users to use different touch gestures to control the parameters of the modulation effect, this way of changing the parameters could also in theory be used for many other effects.

Drum Section

Similarly to the MIDI interface, I have grouped different aspects of the drum section together. The drum interface is displayed in figure 2 below. The interface is split into three sections:

- Top Left - Set BPM, play, stop
- Bottom Left - Save, load, delete, reset interface, list of tracks

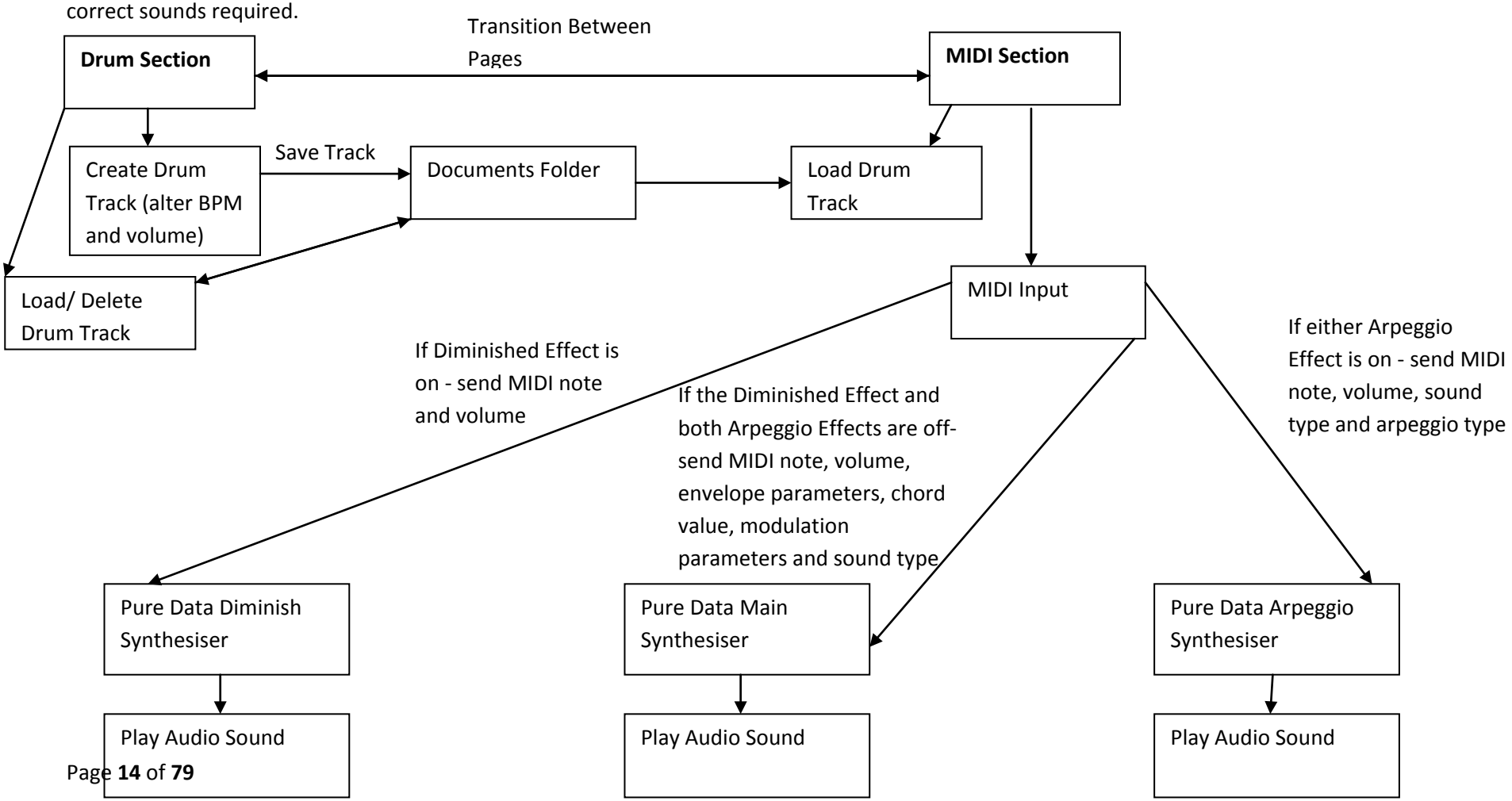
- Central - Drum track customisation (volume, test the drum, change status of drum at specific beats)

Figure 2 - Drum Interface

After conducting the background research, the design I found to be most common with beat generator applications was an x,y grid to allow the user to chose which beat is on at which time. For this reason, I decided on implementing a similar interface within my drum section, the complete right hand side of the interface is made up of an x,y grid implementation, when the application is running, a user will see the grid and be able to change the status of the drums at each beat. This grid also makes it easy for a user to see which drum is on at a specific beat, as all of the views on a specific x axis belong to one drum. To allow the user to know if a drum is on at a specific beat, I have made use of a simple change in background colour to inform them if the drum is on or not, this is a pretty simple way for the user to tell if the selected beat is on or not.

Overall System Diagram

Below is a representation of the flow of data through my application. When a user saves their drum track, it is stored on the iPad in the documents folder, this can then be accessed by both sections to populate the drum track table. When a user is inputting MIDI data to the application and changing the effects, the correct values are sent to the Pure Data patch so that it can produce the correct sounds required.



Implementation

Approach to Solving the Problem

After conducting the background research for this project, it was decided that along with xCode (which is where most of the coding for the project would take place), Pure Data was to be used to manipulate the input into the application and produce different kinds of audio. Therefore, the first stage of development required me to obtain all of the tools required to program my application. This meant, obtaining an 'Apple Developer's Licence' which the Cardiff school of Computer Science provided me with, so that I could create an application and test it on my iPad (not the in-built simulator of xCode), and to download the newest version of Pure Data. To obtain the developer's licence, I had to register my iPad with the department and the newest version of Pure Data was available to download off GitHub[1]. As I had decided upon using a keyboard to input MIDI data, a camera connection kit had to be acquired, this acted as an adapter between the iPad and keyboard.

As discussed in the design section, there are two main sections to the iPad application, these are the drum section and MIDI section (which contains the Pure Data patch). I decided to implement the basic features of the MIDI section first, then move on to the drum section and complete the implementation by adding other advanced features to the MIDI aspect of the application.

In my implementation, I have two main sections; each section has a header and an implementation file. These sections are the ViewController(where the MIDI section is) and the DrumGeneratorViewController.

Overall Sequence of the Application Steps

As there are numerous methods in my implementation, here is a brief description of the sequence in which the methods will be called for each part.

MIDI Section

The MIDI section of the application revolves around the input of MIDI data from a keyboard and therefore, the methods to connect the keyboard are key to the functionality of the application. Then the user may then chose to add different effects to the input and add the drum track. Here is an ordered list of which methods the user will most likely use.

1. Setup the connection to the keyboard. There are three main methods which achieve this connection between the two devices. I followed the steps outlined in the 'Learning Core Audio: A Hands-On Guide to Audio Programming for Mac and iOS'[2] to setup the connection. There are also two smaller methods which I created to ensure that a user wanted to connect a keyboard.
2. Play notes on a keyboard. To read MIDI messages from the attached keyboard, I have adapted a method used in the 'Learning Core Audio: A Hands-On Guide to Audio Programming for Mac and iOS'[2] book. I have used the code from the book to read the MIDI message, note number and velocity and to check the status of the message. From this, I have included my own code to send the required parameters to the Pure Data patch.

3. Record / Playback sequence of notes. There are two methods used to achieve this, the record and playback methods.
4. Add effects to the input. The final methods used on the users input are used to add certain effects to it. There are in total six different effects that a user can apply to the MIDI input. Within these six methods, some set Boolean variables that are used when the application is reading the MIDI notes from a keyboard and others set parameters for the Pure Data patch. The order in which the effects are added isn't important, but, the diminish effect can only be used on its own and the arpeggiator can only be used with the sound type effect.
5. Playing/Stopping of drums. There are two methods used, one to play the drums and the other to stop them.

Along with these main methods that a user may use, there are other key methods which the user may not directly interact with:

6. Populating the table with the correct saved tracks. The four mandatory UITableView methods are used for this. The table is populated from the files stored in an array which is set up in one method on its own.
7. Methods to manage the text view. There are two of these, one to clear the view and one to append text to the view.
8. Finally the view lifecycle methods.

Drum Section

Within the drum section, the main methods that are used are: the method to change which beat is on at a specific beat, the method to play the drums and the methods used to save/load/delete a saved track. Here is a list of methods used in the drum section:

1. The method to change which drum is playing at which beat. To allow the changing of the drums at a specific beat, there are six different methods, two of these reset the interface the its original state and the other four are the mandatory methods of the UICollectionView object.
2. Save / Load / Delete a saved track. For the file management side of the drum section there are, two methods used to save the track, one method to load a track and three methods used to delete a track.
3. Changing the volume of the drums and changing the tempo of the track. There is one method to change the volume of the drums and one to change the tempo of the track.

Along with these methods, there are other important methods a user will not directly use. These are:

4. Populating the table with the correct saved tracks. The four mandatory UITableView methods are used for this. The table is populated from the files stored in an array which is set up in the viewDidLoad method.
5. The view lifecycle methods.

Pure Data

In my application, there is one Pure Data patch. Within this section, I will detail:

1. How to create a Pure Data patch
2. The main synthesiser
3. The arpeggio synthesiser
4. The diminish synthesiser

The synthesisers all receive messages/parameters from the Objective-C code and then use these to create the required sounds.

1. MIDI Section

The MIDI section of my application is focused around the input of MIDI notes and velocities to the iPad by a keyboard, these sounds are then produced and manipulated using a Pure Data patch.

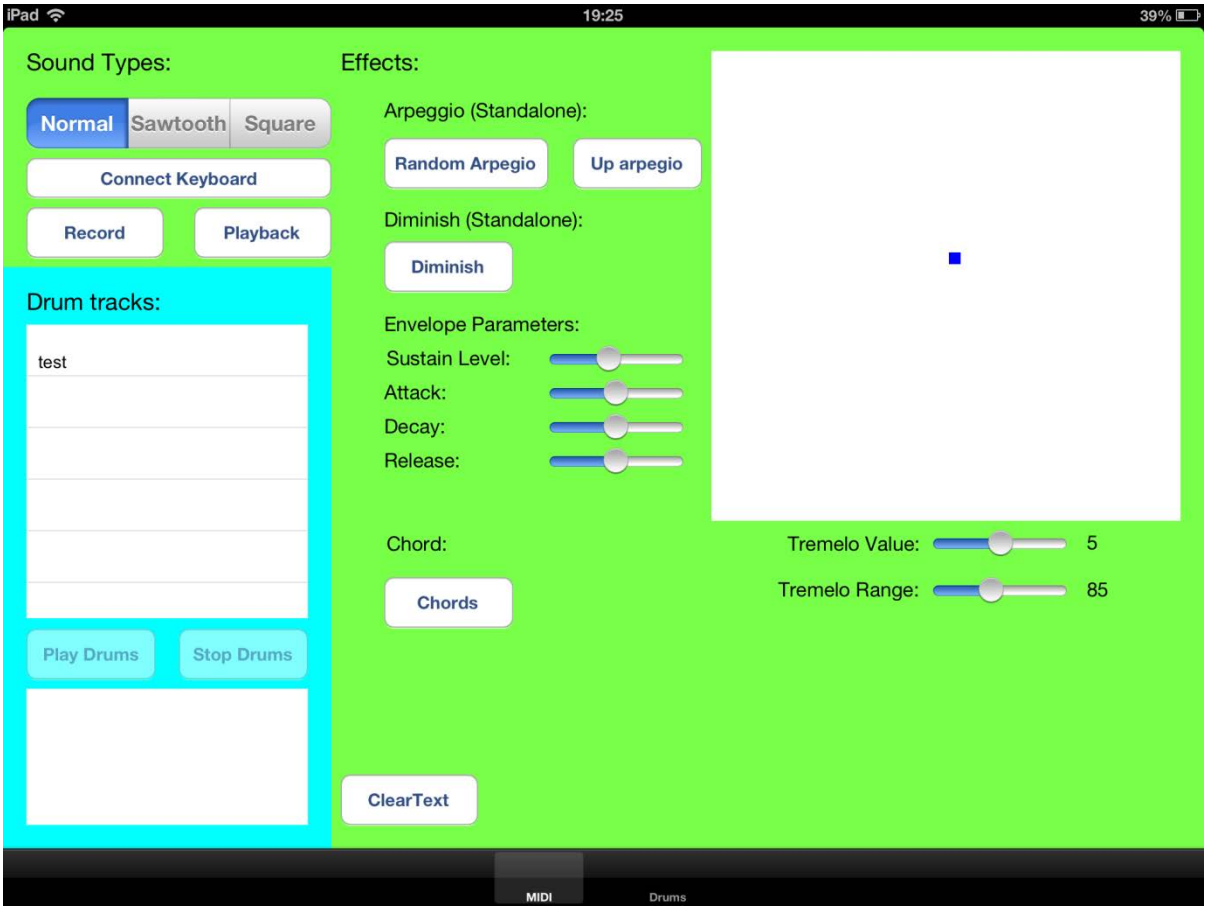


Figure 3 - MIDI interface

1.1 Connect Keyboard

1.1.1 Keyboard Discovery

If the user would like to use a connected keyboard within my application, they must click the button provided (Connect Keyboard). This button, once clicked, displays a UIAlertView on the screen asking the user for confirmation to setup the keyboard. The alert view also informs the user that, if they set up the same keyboard twice, there will be an error (when they play one note it will be read as two - this is currently a problem within my application).

```
#pragma mark - Setup Keyboard Input
- (IBAction)actionConnectKeyboard:(id)sender {
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Set-up Keyboard" message:@"Please note that clicking this
        button twice will cause a failure" delegate:self cancelButtonTitle:@"Cancel" otherButtonTitles:@"Set-up
        Keyboard", nil];

    [alert show];   //create and display an alert asking if the user wants to set up the keyboard
}
```

Figure 4 - Connect Keyboard

Once a button on the UIView is clicked, then the clickedButtonAtIndex method of the UIAlertView is called. This does a simple check to see, whether the title of the UIAlertView is "Set-up Keyboard"(in case there is more than one UIAlertView) and whether the button index is 1 (Set-up Keyboard

clicked). If they are both true, then the [self setupMidi] is called to set up the keyboard and a message is appended to the text view to inform the user that the keyboard has been connected. If this isn't the case, a message appears on the text view saying the connection has been cancelled. As there is only one UIAlertView used in the MIDI section, I can use this method to display the 'Connection Cancelled' message, however, if I were to implement another UIAlertView I would have to remove this, as the message would display when a button on the new UIAlertView is clicked.

```
#pragma mark - alert view method call
//called after button on alert view is pressed
-(void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:(NSInteger)buttonIndex{
    if ((([alertView title] isEqual:@"Set-up Keyboard"]) && (buttonIndex == 1)) {   //if it is the setup keyboard alert view and they chose to set
        up the keyboard
        [self setupMIDI];   //call the method to set up the connection
        [self appendToTextView: [NSString stringWithFormat:@"Connection Established(If keyboard connected)"]];  //print to the screen that a
            connection was established
    }else{
        [self appendToTextView: [NSString stringWithFormat:@"Connection Cancelled"]];   //if there was more than one UIAlertView this message
            would need to be removed as it would display for the other UIAlertView when a button is clicked
    }
}
```

Figure 5 - Alert View Method

1.1.2 MIDI setup

To set-up a MIDI keyboard, I followed the steps outlined in the 'Learning Core Audio: A Hands-On Guide to Audio Programming for Mac and iOS'[2]. There are three methods required to allow my application to read in MIDI data from a keyboard. These are: setupMIDI, MyMIDINotifyProc and CheckError.

1.1.2.1 setupMIDI

The setup MIDI method is where the MIDI keyboard is setup. The basic outline of this method is, to create a reference to a session and then create an input port to receive MIDI data using the session reference. We then iterate over the number of sources and set up the connection between the application and the keyboard using the MIDIPortConnectSource method (there are two sources for my application if a keyboard is connected - one is the keyboard itself and the other is a network session). After the keyboard has been setup, MIDI messages can be read by the application.

```
#pragma mark - Objective-C Methods for MIDI input set up
//Code adapted from Learning Core Audio - A hands-On guide to Audio Programming for Mac and iOS - chapter 11 Core MIDI
-(void) setupMIDI{
    MIDIClientRef client = NULL;    //create a session
    CheckError(MIDIClientCreate(CFSTR("Core MIDI to System Sound Demo"), MyMIDINotifyProc, (__bridge void *)(self), &client), "Couldn't create
        MIDI client");

    MIDIPortRef inPort;     //Input port to recieve MIDI data
    CheckError(MIDIInputPortCreate(client, CFSTR("Input port"), MyMIDIReadProc, (__bridge void *)(self), &inPort), "Couldn't create MIDI input
        port");

    unsigned long sourceCount = MIDIGetNumberOfSources();   //Get number of sources
    [self appendToTextView:[NSString stringWithFormat:@"%ld sources\n", sourceCount]];

    for(int i = 0; i < sourceCount; ++i){   //create an ebdpoint reference for each input device
        MIDIEndpointRef src = MIDIGetSource(i);
        CFStringRef endpointName = NULL;

        CheckError(MIDIObjectGetStringProperty(src, kMIDIPropertyName, &endpointName), "Couldn't get endpoint name");   //get the name of the
            device

        char endpointNameC[255];
        CFStringGetCString(endpointName, endpointNameC, 255, kCFStringEncodingUTF8);

        NSString *temp = [[NSString alloc]initWithUTF8String:endpointNameC];
        [self appendToTextView: [NSString stringWithFormat:@"  source %d: %@\n", i, temp]];

        CheckError(MIDIPortConnectSource(inPort, src, NULL), "Couldn't connect MIDI port");     //connect to the source
    }
}
```

Figure 6 - setupMIDI

1.1.2.2 MyMIDINotifyProc

This method simply notifies the user of any messages that have occurred during the setup of the MIDI keyboard.

```
#pragma mark - C-Method Used While Creating Client
//Code adapted from Learning Core Audio - A hands-On guide to Audio Programming for Mac and iOS - chapter 11 Core MIDI
void MyMIDINotifyProc(const MIDINotification *message, void *refCon){
    printf("MIDI Notify, messageId=%ld", message->messageID);
    ViewController *viewController = (__bridge ViewController *) refCon;
    [viewController appendToTextView:[NSString stringWithFormat:@"MIDI Notify, messageId=%ld \n", message->messageID]]; //log any message
}
```

Figure 7 - Log Messages

1.1.2.3 CheckError

This method is used to ascertain there are no errors while setting up the keyboard, this method also allows us to shorten the method calls as we no longer need to use the assert() method if the method is called using this CheckError method.

```
#pragma mark - Check Error C-Method For MIDI Setup
//Code adapted from Learning Core Audio - A hands-On guide to Audio Programming for Mac and iOS - chapter 11 Core MIDI
//assert there is no error
static void CheckError(OSStatus error, const char *operation){

    if(error==noErr){
        return;
    }

    char errorString[20];
    *(UInt32 *)(errorString + 1) = CFSwapInt32HostToBig(error);
    if (isprint(errorString[1]) && isprint(errorString[2]) && isprint(errorString[3]) && isprint(errorString[4])){
        //errorString[0] = errorString[5] = '/';
        //errorString[6] = '/0';
    }else{
        sprintf(errorString, "%d", (int)error);
    }
    exit(1);
}
```

Figure 8 - Check Error

1.2 Keyboard Input

Once a keyboard has been setup, a user is then able to play notes on this keyboard to create sounds via the application. When the MIDI messages are received, the MIDI note number and velocity are passed to the Pure Data patch with the correct effect parameters. The method that reads the data in from the keyboard is MyMIDIReadProc which has been adapted from the code in the 'Learning Core Audio: A Hands-On Guide to Audio Programming for Mac and iOS'[2] book. The code in figure 9 below is the code taken from the Learning Core Audio book.

```
#pragma mark - MIDI Input Recieve C-Method - Send Data to PD Patch
//Code to get MIDI data adapted from Learning Core Audio - A hands-On guide to Audio Programming for Mac and iOS - chapter 11 Core MIDI
static void MyMIDIReadProc(const MIDIPacketList *pktlist, void *refCon, void *connRefCon){

    ViewController *viewController = (__bridge ViewController *) refCon;    //get a reference to the view controller
    MIDIPacket *packet = (MIDIPacket *)pktlist->packet; //get the current packet

    for (int i = 0; i < pktlist->numPackets; i++) { //iterate over packets

        Byte midiStatus = packet->data[0] >> 4;     //get the status byte of the MIDI data
        Byte note = packet->data[1] & 0x7F;         //get the MIDI note number
        Byte velocity = packet->data[2] & 0x7F;     //get the velocity that the note was played
```

Figure 9 - MIDI Input via Keyboard

Once a packet has been assigned to the packet variable, we enter a for loop. Within the for loop there are three byte variables, the first stores the status of the MIDI (note on or note off), the second holds the MIDI note number and the third stores the velocity that the note was played at.

If the midiStatus is equal to 0x09 then this signifies that a note on event has occurred and the application needs to start playing a sound.

```
if(midiStatus == 0x09){ //if a note is played
```

Figure 10 - Note on Event

Within this if statement, we firstly check is the user wishes to record their input by checking if the Boolean variable boolRecord is true or not.

```
//if the user is recording the input
if(boolRecord){
    //find the time since the start of the dateStart Date (to ensure correct timing on playback)
    NSTimeInterval timeInterval = [dateStart timeIntervalSinceNow];

    //allocate space for now item object
    curr = (item *)malloc(sizeof(item));

    //set all parameters of the item object (or struct)
    curr -> note = (int)note;
    curr -> volume = (int)velocity;
    curr -> OnOfftime = timeInterval;
    curr -> onOrOff = true; //specifies note on
    curr -> soundType = intSoundType;
    curr -> diminish = boolDiminished;
    curr -> arpeggio = boolUpArpegio;
    curr -> arpeggioRand = boolRandomArpegio;
    curr -> chord = boolChord;

    //set the next item object to be null
    curr -> next = NULL;

    //if this is the first note recorded, then the head of the linked list is equal to null
    if (!head) {
        head = curr;
        tail = curr;
    }else{
        tail -> next = curr;
        tail = curr;
    }

}
```

Figure 11 - Recording

When a user is recording their input to the application the boolRecord variable has a value of 'true' which allows the application to enter the if statement. To start recording, a new time interval variable is created to ensure that the correct time of the note play is stored. Then a new item object (a struct containing: note, velocity, time played/released, 5 Boolean effect variables and a pointer to the next node in the list) is created, allocated space and the various parameters are then set within the 'curr' object. If the head object is equal to null, then we set the head and tail to be the current item object otherwise, we set the next item in the sequence to be 'curr' and set tail to be 'curr'. Once this sequence of instructions is completed, then the application processes the various (effect) Boolean variables so that it can play the correct sound back to the user.

```
if(boolDiminished){   //diminish play - send correct parameters to diminish synthesiser
    [PdBase sendFloat:note toReceiver:@"diminishedMidinote"];
    [PdBase sendFloat:velocity toReceiver:@"diminishedVolume"];
```

Figure 12 - Diminish Effect

First we test if the diminished effect has been chosen, if it has, the various parameters are sent to the diminished synthesiser in the Pure Data patch. If it hasn't, we test the random arpeggio variable, if it is 'true', then we send the values to the correct receivers in the arpeggio synthesiser, if it isn't on we check the up arpeggio variable and repeat the process.

```
}else if(boolRandomArpegio){    //up arpeggio play - send correct parameters to arpeggio synthesiser
    [PdBase sendFloat:velocity toReceiver:@"arpegioVolume"];
    [PdBase sendFloat:intSoundType toReceiver:@"arpegioSoundType"];
    [PdBase sendFloat:0 toReceiver:@"option"];  //0 specifies random arpeggio to be played
    [PdBase sendFloat:note toReceiver:@"arpegioMidinote"];
}else if(boolUpArpegio){    //random arpeggio play - send correct parameters to arpeggio synthesiser
    [PdBase sendFloat:velocity toReceiver:@"arpegioVolume"];
    [PdBase sendFloat:1 toReceiver:@"option"];  //1 specifies up arpeggio to be played
    [PdBase sendFloat:intSoundType toReceiver:@"arpegioSoundType"];
    [PdBase sendFloat:note toReceiver:@"arpegioMidinote"];
```

Figure 13 - Arpeggio Effects

The three Boolean variables above were check as, if they were true, then the parameters must either be sent to the diminish or arpeggio synthesisers in the Pure Data patch. If none of the above variables were true, then a user is only using effects that are contained within the main synthesiser of the Pure Data patch.

```
}else{  //send correct parameters to the main synthesiser

    [PdBase sendFloat:velocity toReceiver:@"volume"];
    [PdBase sendFloat:intSoundType toReceiver:@"soundType"];    //0 for normal, 1 for sawtooth, 2 for square

    if(boolChord){  //if the chord effect is on
        [PdBase sendFloat:1 toReceiver:@"chord"];   //send a 1 to the PD patch to ensure the chord effect is played
    }else{
        [PdBase sendFloat:0 toReceiver:@"chord"]; //ensure chord effect is not on unless stated
    }

    //send the envelope parameters
    [PdBase sendFloat:intAttack toReceiver:@"attack"];
    [PdBase sendFloat:intDecay toReceiver:@"decay"];
    [PdBase sendFloat:intSustain toReceiver:@"sustain"];
    [PdBase sendFloat:note toReceiver:@"midinote"];

    //send the modulating parameters
    if (intTremeloValue > 0) {
        [PdBase sendFloat:intTremeloValue toReceiver:@"ampMod"];
        [PdBase sendFloat:intTremeloRange toReceiver:@"temp"];

    }

}
```

Figure 14 - Main Synthesiser

Here, the various parameters are sent to the main synthesiser in the Pure Data patch. If the chord effect is on then a '1' is sent to the chord receiver otherwise it is sent a '0'. The final if statement, sends the modulation parameters to the PD patch if the 'value' of the modulation is greater than '0'.

If the midiStatus was not equal to 0x09 we test if it was equal to 0x08, if it is, then we have a note off event taking place. There are only two checks in this part of the method. The first is to test if we are recording at that point, if we are we need to record when the note was released etc. We create a new NSTimeInterval and check the time interval between now and when the NSDate object 'dateStart' was set, this gives us the time that the note should be released in the sequence. As with the MIDI note on event, we have to create and allocate space for a new item object called 'curr'. The various values are then stored in this struct with the value of 'onOrOff' being false to indicate a note off. We then set the next item in the list to be the 'curr' item object and set tail to be 'curr'.

Finally, if the boolDiminished effect isn't on, we stop the playing of the required note(s) by sending the note to the midiOffNote receiver, we then trigger the stop receiver and send the release time of the volume envelope to the release receiver. We also have to send the sound type of the note to the Pure Data patch. In case either arpeggio effect is on, the 'arpegioStop' receiver is triggered in the Pure Data patch.

```
}else if(midiStatus == 0x08){      //if a note is released

    //if the user is recording the input
    if(boolRecord){
        //find the time since the start of the dateStart Date (to ensure correct timing on playback)
        NSTimeInterval timeInterval = [dateStart timeIntervalSinceNow];

        //allocate space for a new item object
        curr = (item *)malloc(sizeof(item));

        //set all parameters of the item object (or struct)
        curr -> note = (int)note;
        curr -> volume = 0;
        curr -> OnOfftime = timeInterval;
        curr->onOrOff = false;  //specifies note off
        curr -> soundType = intSoundType;
        curr -> diminish = boolDiminished;
        curr -> arpeggio = boolUpArpegio;
        curr -> arpeggioRand = boolRandomArpegio;
        curr -> chord = boolChord;
        //set the next object in the linked list to be null
        curr -> next = NULL;

        tail-> next = curr;
        tail = curr;

    }

    if (!boolDiminished) {  //if the diminished effect isn't played (it fades out for the diminished) then stop the playing of the note

        //send the required parameters to the PD patch for the stopping of a note

        //[PdBase sendBangToReceiver:@"stop"];
        [PdBase sendFloat:intRelease toReceiver:@"release"];
        [PdBase sendFloat:intSoundType toReceiver:@"soundType"];
        [PdBase sendFloat:note toReceiver:@"midiOffNote"];
        [PdBase sendBangToReceiver:@"arpegioStop"];
    }
    //debug statement
    [viewController appendToTextView:[NSString stringWithFormat:@"Note Off. Note=%d", note]];
}

packet = MIDIPacketNext(packet);    //move on to the next packet, possibly falling out of the loop
```

Figure 15 - Note Off Event

After checking to see what status the MIDI note was in (on or off), we then get the next packet using MIDIPacketNext which will likely force the program out of the loop.

1.3 Record and Playback

1.3.1 Record

The main record method has been described above, however, when a user wishes to record their input, they must click the record button on the screen. This calls the actionRecord method which simply changes a few variables. A Boolean variable called boolRecord is used so that the application knows whether the user wishes to record or not, if they want to start recording the variable is set to true. When a user wishes to record (boolRecord is false at the time), this method sets the play button to be hidden and sets the boolRecord variable to be true, it also sets the head and tail of the linked list to be null values and sets a new NSDate variable (dateStart) which is used in the recording part of the input method. If the user wishes to stop recording, the play button is shown and boolRecord is set to false.

```
#pragma mark - Record Input and Playback Input
//Record
- (IBAction)actionRecord:(id)sender {
    if (!boolRecord) {  //if not recording - then begin recording
        btnPlayRecording.hidden = YES;  //hide the btnPlayRecording button
        boolRecord = YES;   //set boolRecord to true so that input will be recorded in the MyMIDIReadProc C-Method

        head = NULL;
        tail = NULL;

        dateStart = [NSDate date];  //start the times that will be used to get the time that the notes are played
    }else{  //else recording - then stop recording
        btnPlayRecording.hidden = NO;   //show the play button and set the boolean variable used for recording
        boolRecord = NO;
    }
}
```

Figure 16 - actionRecord

1.3.2 Playback

Unlike most of the key methods in the MIDI section, the actionPlaybackRecording method is a bit more complex. This method, as the method name states, plays the recorded sequence of notes back to the user. The way that this is achieved is by setting a new NSDate variable and then executing a while loop where most of the processing is done. This is all done inside the dispatch_async() which allows concurrent execution of more than one task and allows a user to keep interacting with the application when the while loop in this method is processed.

The 'curr' variable is an item * variable which is a member of the linked list(the linked list is made up of a number of structures pointing to each other). Within the 'curr' variable we have the state of the effects when the note was played, all of the parameters in the 'curr' object are read and stored in seperate variables to test (or send to the PD patch) later in the method.

```
//Playback
- (IBAction)actionPlaybackRecording:(id)sender {

    curr = head;    //set the current element of the list to be the head

    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,(unsigned long)NULL), ^(void) {    //do another task - allows other
        tasks to be performed while the code below is executed
        NSDate *compare = [NSDate date];    //NSDate variable to compare the time to that of the notes stored in the linked list

        while (curr) {  //while there is a node
            int note = curr->note;  //get note
            int velocity = curr -> volume;
            bool boolOnOff = curr->onOrOff; //get it's status (on or off)
            double noteTime= curr->OnOfftime;   //get the time it is meant to be played/ stopped
            int intTypeOfSound = curr -> soundType; //get the sound type
            //Get the state of the effects
            bool arpeggio = curr -> arpeggio;
            bool arpeggioRand = curr -> arpeggioRand;
            bool diminish = curr -> diminish;
            bool chord = curr -> chord;
```

Figure 17 - actionPlaybackRecording

Next we calculate the interval between the start of the method call and the time now (using [compare timeIntervalSinceNow]) and round it to two decimal places, we also round the note time of the note held in the 'curr' item to two decimal places. We do this, as it is almost impossible to match the two times otherwise and I have deemed that being correct to two decimal places is accurate enough. We compare the two above, that is the interval since the method started and the time that the note was played/stopped. If they aren't equal then the process happens again until they are, however if they are equal, the effects in the 'curr' object are checked (using the variables) to see which synthesiser the parameters need to be sent to.

```
NSTimeInterval timeInterval2 = [compare timeIntervalSinceNow];  //compare the start time(compare) with the time now
noteTime = round (noteTime * 100) / 100.0; //round the note time to 2 decimal places
double interval = round (timeInterval2 * 100) / 100.0;  //round the interval time to 2 decimal places

if (interval == noteTime) { //if the interval and the note time are the same then a note is to be played

    if(diminish){   //if the diminished effect is applied, send correct parameters to the diminished synthesiser
        [PdBase sendFloat:note toReceiver:@"diminishedMidinote"];
        [PdBase sendFloat:velocity toReceiver:@"diminishedVolume"];
    }else if(arpeggio){ //if the up arpeggio effect is applied, send correct parameters to the diminished synthesiser
        [PdBase sendFloat:intTypeOfSound toReceiver:@"arpegioSoundType"];
        if (boolOnOff == true) {    //note is to be played
            [PdBase sendFloat:velocity toReceiver:@"arpegioVolume"];
            [PdBase sendFloat:1 toReceiver:@"option"];

            [PdBase sendFloat:note toReceiver:@"arpegioMidinote"];
        }else{  //note is to be stopped
            [PdBase sendBangToReceiver:@"arpegioStop"];
        }
    }else if(arpeggioRand){ //if the random arpeggio effect is applied, send correct parameters to the diminished synthesiser
        if (boolOnOff == true) {    //note is to be played
            [PdBase sendFloat:velocity toReceiver:@"arpegioVolume"];
            [PdBase sendFloat:0 toReceiver:@"option"];
            [PdBase sendFloat:intTypeOfSound toReceiver:@"arpegioSoundType"];
            [PdBase sendFloat:note toReceiver:@"arpegioMidinote"];
        }else{  //note is to be stopped
            [PdBase sendBangToReceiver:@"arpegioStop"];
        }
```

Figure 18 - Check Diminish and Arpeggio Effects

These if statements in figure 18 above, test whether the parameters need to be sent to the arpeggio or diminish synthesisers. If they don't, then the parameters will be passed to the main synthesiser.

```
            }else{  //else send correct parameters to the main synthesizer

                [PdBase sendFloat:intTypeOfSound toReceiver:@"soundType"];   //send the sound type


                if(chord){  //check if the chord effect is applied
                    [PdBase sendFloat:1 toReceiver:@"chord"];   //send if chord effect is applied
                }else{
                    [PdBase sendFloat:0 toReceiver:@"chord"]; //ensure chord effect is not on unless stated
                }

                if (boolOnOff == true) {    //note is to be played
                    [PdBase sendFloat:velocity toReceiver:@"volume"];    //send volume to correct PD reciever
                    [PdBase sendFloat:note toReceiver:@"midinote"]; //send note to correct PD reciever
                }else{  //note is to be stopped
                    [PdBase sendFloat:note toReceiver:@"midiOffNote"];  //send note to correct PD reciever
                }
            }
            curr = curr->next;  //move to next node in the list
        }
    }
});
}
```

Figure 19 - Main Synthesiser

Similar to the arpeggio synthesiser, the type of sound is sent to the main synthesiser (0 for normal, 1 for sawtooth and 2 for square). Unlike the arpeggio synthesiser, for the main synthesiser, the value of the chord effect is sent to the patch (0 for no chord effect, 1 for chord effect). Finally, we find out if the note was played or released using the boolOnOff variable. If boolOnOff is true it means the

note is played, so the correct volume and MIDI note number is passed to the Pure Data patch, if boolOnOff is false, then the note is released so the correct MIDI note number is passed to the Pure Data to stop the playing of the note. We then move to the next item in the list using curr = curr -> next.

1.4 Effects

1.4.1 Diminish

When a user wishes to add the diminished effect, the actionDiminished method is called and it is a simple method that sets the Boolean variable boolDiminished to be true (if they want the effect) or false (if they want to stop the effect). The actual sound is generated in the MyMIDIReadProc method that is explained in detail above. As, when the user wants to use the diminish effect, the diminish synthesiser is used, the buttons for the other effects are disabled as they can't be used within the diminish synthesiser.

```
#pragma mark - Diminished Effect
- (IBAction)actionDiminished:(id)sender {
    if (boolDiminished) {   //if the diminished effect is on
        boolDiminished = NO;    //disable the effect

        //Enable other interface objects
        btnChords.enabled = true;
        btnChords.alpha = 1;
        btnUp.enabled = true;
        btnUp.alpha = 1;
        btnRandom.enabled = true;
        btnRandom.alpha = 1;
        sgSoundType.enabled = true;
    } else {    //diminished effect not on
        boolDiminished = YES;   //enable diminished effect

        //Disable other interface objects (those which can't be used at the same time as this effect)
        btnChords.enabled = false;
        btnChords.alpha = 0.5;
        btnUp.enabled = false;
        btnUp.alpha = 0.5;
        btnRandom.enabled = false;
        btnRandom.alpha = 0.5;
        sgSoundType.enabled = false;
    }
}
```

Figure 20 - actionDiminish

1.4.2 Arpeggiator

There are two arpeggio effects, one plays an up arpeggio (actionUpArpegio) and one a random arpeggio (actionRandomArpegio), as with most other effects, when the two arpeggio methods are called, they set a Boolean variable to be true or false depending on whether the effect is to be on or off. If the effect is to be turned off, the methods send a stop trigger to the arpeggio patch. When an arpeggio effect is used, the parameters are sent to the arpeggio synthesiser, so any effect which can't be used within this, is disabled (the buttons to set them).

```
#pragma mark - Arpegio Effects
//Random Arpegio
- (IBAction)actionRandomArpegio:(id)sender {
    if (boolRandomArpegio) {    //if the random effect is on
        boolRandomArpegio = NO; //disable effect

         //Enable other interface objects
        btnDiminish.enabled = true;
        btnDiminish.alpha = 1;
        btnUp.enabled = true;
        btnUp.alpha = 1;
        btnChords.enabled = true;
        btnChords.alpha = 1;

        [PdBase sendBangToReceiver:@"arpegioStop"]; //send stop to PD reciever (in case it is still playing)
    }else{      //random effect not on
        boolRandomArpegio = YES;    //enable effect

        //Disable other interface objects (those which can't be used at the same time as this effect)
        btnDiminish.enabled = false;
        btnDiminish.alpha = 0.5;
        btnUp.enabled = false;
        btnUp.alpha = 0.5;
        btnChords.enabled = false;
        btnChords.alpha = 0.5;
    }
}
```

Figure 21 - actionRandomArpeggio

1.4.3 Chords

The basic outline of the chord method is exactly the same as the diminished method. The chord method is called actionChord and it sets the variable boolChord to be true or false depending on whether the user wants to use the chord effect or not. Similar to the arpeggiator and diminish methods, the chord method disables the effects that can't be used within the main synthesiser.

```
#pragma mark - Chord Effect
- (IBAction)actionChord:(id)sender {
    if (boolChord) {   //if the chord effect is on
        boolChord = NO;    //disable the effect

         //Enable other interface objects
        btnDiminish.enabled = true;
        btnDiminish.alpha = 1;
        btnUp.enabled = true;
        btnUp.alpha = 1;
        btnRandom.enabled = true;
        btnRandom.alpha = 1;

    } else {    //chord effect not on
        boolChord = YES;   //enable diminished effect

        //Disable other interface objects (those which can't be used at the same time as this effect)
        btnDiminish.enabled = false;
        btnDiminish.alpha = 0.5;
        btnUp.enabled = false;
        btnUp.alpha = 0.5;
        btnRandom.enabled = false;
        btnRandom.alpha = 0.5;

    }
}
```

Figure 22 - actionChord

1.4.4 Sound Type

To allow users of the application to chose between the three basic sound types available, I have implemented a segmented control. This segmented control has three states it can be in, the user can chose normal, sawtooth or square which in turn sets the intSoundType variable to be the index number of the button chosen on the segmented control. To make sure that no other sound is carried over (e.g. if the user is still holding down a note when the change occurs), the stopAll receiver is triggered in the Pure Data patch when the sound type is changed.

```
#pragma mark - Sound Type Changed
- (IBAction)sgSoundTypeChange:(id)sender {
    intSoundType = [sender selectedSegmentIndex];   //get index of segment control - normal 0 - sawtooth 1 - square 2
    [PdBase sendBangToReceiver:@"stopAll"]; //send a stop signal to the PD patch to ensure that nothing playes after
        the change of sound type
}
```

Figure 23 - sgSoundTypeChange

1.4.5 Envelope

Four sliders have been used in the application to allow the control of the implemented envelope feature. These four sliders represent the four different phases; attack, sustain, decay and release. Each phase is associated with its own integer variable (e.g intAttack), when the sliders are changed, we created a temporary variable to hold the value of the slider and then set the variables (intAttack etc) to be this temporary variable. As stated in most other methods, the actual creation of sound is done in the MyMIDIReadProc.

```
#pragma mark - Envelope parameters changed actions
- (IBAction)actionSustainChange:(id)sender {
    int temp = (int)[sliSustain value]; //get value of sliSustain slider
    intSustain = temp;  //store value in intSustain
}

- (IBAction)actionAttackChange:(id)sender { //see actionSustainChange for comments
    int temp = (int)[sliAattack value];
    intAttack = temp;
}

- (IBAction)actionDecayChange:(id)sender {  //see actionSustainChange for comments
    int temp = (int)[sliDecay value];
    intDecay = temp;
}

- (IBAction)actionReleaseChange:(id)sender {    //see actionSustainChange for comments
    int temp = (int)[sliRelease value];
    intRelease = temp;
}
#pragma mark - End Envelope
```

Figure 24 - Envelope Actions

1.4.6 Amplitude Modifier

Applying the amplitude modifier effect is a bit more complex than the other effects. There are two ways to change the input parameters of the Pure Data patch. One is to use the on-screen sliders and the other is to tap or swipe the effect view (effectViewSuper) of the interface. Both ways of changing the input parameters change the variables intTremeloValue and intTremeloRange which are passed to the Pure Data patch to achieve the effect.

1.4.6.1 EffectViewSuper

The most complex way to change the input parameters of the modulation is to use the 'viewEffectSuper'which is a UIView placed on the interface. It allows a user to tap or swipe with their finger to change the modulation. The position of the tap or swipe is then used to set the two input parameters.

1.4.6.2 Effect View

When a user swipes or touches the 'viewEffectSuper', a blue square is placed at the position of the tap or swipe. This is done by adding another UIView called 'effectView' onto the 'viewEffectSuper'. This process also takes place when a user changes the values of the sliders.

1.4.6.3 Tap and Swipe Gesture

The tap and swipe gestures contain the same code but they have to be different methods as they are different actions. The position of where the user touched the view is retrieved and stored in a CGPoint variable. This point is then split into its y and x components and stored in yPos and xPos respectively. As the current position of the 'effectView' (blue square) is now out of date, it is removed from its super view. Before placing the 'effectView' onto the screen in its new position, simple checks are made to ensure the swipe of the user hasn't breached the boundaries of the 'viewEffectSuper', if they have, the 'xPos' and 'yPos' are set to the min or max values depending on how the 'viewEffectSuper' has been breached. Once this has been done, the new position of the 'effectView' can be calculated using 'xPos' and 'yPos'. The 'effectView' is then allocated using the new position (or frame) and added to the 'viewEffectSuper'.

The final thing to do, is to set the two containing variables for the modulation parameters, this is done by dividing the 'xPos' by 40 and setting 'intTremeloValue' to be the result, the same process happens to determine the 'intTremeloRange' from the 'yPos', but the 'yPos' is only divided by 2. The sliders are then updated with the new values and the values are printed to the text view for debugging purposes. The two modulation parameters are then sent to the correct receiver in the Pure Data patch ('intTremeloValue' to 'ampMod' and 'intTremeloRange' to temp).

```
#pragma mark - Gestures - Tremelo
- (IBAction)actionTapGesture:(id)sender {
    CGPoint touchPoint = [sender locationInView:viewEffectSuper];   //Retrieve the point of the tap
    float xPos = touchPoint.x;  //store x value of point
    float yPos = touchPoint.y;  //store y value of point
    tvLabelOutput.text=[NSString stringWithFormat:@"%@ touched at: %@", tvLabelOutput.text, NSStringFromCGPoint
        (touchPoint)];   //output point

    [self.viewEffect removeFromSuperview];  //remove any subview from the view (remove previous point graphics)

    //For tremelo
    //ensure that the tap is within the boundary
    if (xPos <0) {
        xPos =0;
    }else if(xPos > 400){
        xPos = 400;
    }

    if (yPos <0) {
        yPos =0;
    }else if (yPos > 400){
        yPos = 400;
    }

    CGRect rect = CGRectMake(xPos, yPos, 10, 10);   //create the graphics point
    viewEffect = [[effectView alloc] initWithFrame:rect];   //allocate and initialise the subview with the frame of
        rect
    [self.viewEffectSuper addSubview:viewEffect];   //add viewEffect onto viewEffectSuper as a subview

    xPos = xPos / 40;   //divide by 40 to get the value that will be sent to the PD patch
    intTremeloValue = xPos; //set the container for the tremelo valuy
    sliTremelo.value = intTremeloValue; //update tremelo slider
    lblTremeloValue.text = [NSString stringWithFormat:@"%i", intTremeloValue];  //set the value of the label

    yPos = yPos / 2;    //divide by 40 to get the value that will be sent to the PD patch
    intTremeloRange = yPos; //set the container for the tremelo range
    sliTremeloRange.value = intTremeloRange; //update range slider
    lblTremeloRange.text = [NSString stringWithFormat:@"%i", intTremeloRange];  //set the value of the label

    [PdBase sendFloat:intTremeloValue toReceiver:@"ampMod"];    //send tremeloValue to correct PD reciever
    [PdBase sendFloat:intTremeloRange toReceiver:@"temp"];      //send tremeloRange to correct PD reciever
}
```

Figure 25 - Modulation - actionTapGesture

1.4.6.4 Sliders

The simpler of the two methods to change the parameters of the modulation effect, from a coding perspective, is the changing of the parameters using the two sliders. There are two sliders that can be manipulated and the method for each is practically the same, with one changing the value of the manipulation and the other changing the range of the manipulation. The important part of the method is when the variable to store the value/range of the manipulation is set to be the value from their respective sliders; the value is also added to the text view for debugging purposes.

The middle part of the method, then updates the 'viewEffectSuper'. It removes the 'effectView' (blue spot) from the 'viewEffectSuper', calculates it's new position using the 'intTremeloValue' and 'intTremeloRange' variables, allocates the 'effectView' with the new position and then adds the 'effectView' to the 'viewEffectSuper'. The 'intTremeloValue' is used as a basis of calculating the x value on the 'viewEffectSuper' of where the 'effectView' should be placed, it multiplies the value stored in the variable by a factor of 40. The 'intTremeloRange' is the same for the y value, but it is only multiplied by a factor of 2.

Finally, it sends the new value/range of the manipulation to the Pure Data patch.

```
#pragma mark - Sliders - Tremelo
- (IBAction)actionsliTremeloChange:(id)sender {
    intTremeloValue = (int)[sliTremelo value];  //retrieve value from the sliTremelo slider
    lblTremeloValue.text = [NSString stringWithFormat:@"%i", intTremeloValue];  //set the value of the
        label

    //update the effect view
    [self.viewEffect removeFromSuperview];
    CGRect rect = CGRectMake(intTremeloValue*40, intTremeloRange*2, 10, 10);   //create the graphics
        point
    viewEffect = [[effectView alloc] initWithFrame:rect];   //allocate and initialise the subview with
        the frame of rect
    [self.viewEffectSuper addSubview:viewEffect];   //add viewEffect onto viewEffectSuper as a subview

    [PdBase sendFloat:intTremeloValue toReceiver:@"ampMod"];    //send update to PD patch
}

- (IBAction)actionsliTremeloRangeChange:(id)sender {
    intTremeloRange = (int)[sliTremeloRange value]; //retrieve the value from the rsliTremeloRange
        slider
    lblTremeloRange.text = [NSString stringWithFormat:@"%i", intTremeloRange];  //set the value of the
        label

    //update the effect view
    [self.viewEffect removeFromSuperview];
    CGRect rect = CGRectMake(intTremeloValue*40, intTremeloRange*2, 10, 10);   //create the graphics
        point
    viewEffect = [[effectView alloc] initWithFrame:rect];   //allocate and initialise the subview with
        the frame of rect
    [self.viewEffectSuper addSubview:viewEffect];   //add viewEffect onto viewEffectSuper as a subview

    [PdBase sendFloat:intTremeloRange toReceiver:@"temp"];  //send update to PD patch

}
```

Figure 26 - Modulation - slider actions

1.5 Play / Stop drums

The playing and stopping of the drum tracks in the MIDI section is very similar to the corresponding methods in the drum section. With regards to the stop method, they are the same apart from the fact that some of the variable names have been changed. The play method in the MIDI section is a mixture of the load method and play method in the drum section. As both sections have the same methods, I will describe the playing and stopping of the drums in more detail in the drum section below.

1.6 Populating the table of saved drum tracks

1.6.1 fileSetup method

To ensure that a user is able to see their saved drum tracks in the drum table, we need to fill the 'array of files' variable (arrOfFiles) with all of the files that a user has saved (the drum table is populated from this array of files). The basic operation of this method is to find the path to the files, and then fill the array using the contentsOfDirectoryAtPath method of the NSFileManager. To make sure the user can't play a file if there are no files, a check is made, if arrOfFiles is empty (count == 0) then the stop and play drum buttons are disabled.

```
#pragma mark - Saved File Setup
-(void)fileSetup{
    //Find files in the directory folder of the app and put these contents in arrayoffiles
    NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES);
    NSFileManager *filemgr = [NSFileManager defaultManager];
    arrOfFiles = [filemgr contentsOfDirectoryAtPath: [paths objectAtIndex:0] error: nil];

    if ([arrOfFiles count] == 0) {  //if there are no files then disable the play and stop buttons
        btnPlayDrums.enabled = false;
        btnPlayDrums.alpha = 0.5;
        btnStopDrums.enabled = false;
        btnStopDrums.alpha = 0.5;
    }
}
```

Figure 27 - fileSetup

1.6.2 Drum table

To allow the user to view which drum tracks they have created, I make use of a single UITableView to display the files. As with the play and stop drum methods, the required UITableView methods for the drum table in the MIDI section are exactly the same as the ones in the drum section; therefore I have explained the code fully in the drum section of this report.

1.7 Text View Methods

To allow myself to debug the application, I had to create a text view which printed out the debugging statements as, when I needed to debug the MIDI section, I couldn't connect it to the computer as I had to connect a keyboard to the iPad. There are two simple methods which relate to the text view, one is actionClearText which sets the text of the text view(tvLabelOutput) to be empty and the other is appendToTextView which adds text to the text view. When this method is called, it adds the text using the NSString method stringWithFormat which allows you to concatenate two strings, in this case, it concatenates the previous text on the view and the text to be added which is derived from the moreText parameter. Finally, the text view is scrolled so that I could see the most recent data to be printed to the screen.

```
#pragma mark - Text View Methods - Addd and Clear
//add to Text View - allows C methods to add to text view
-(void) appendToTextView: (NSString*) moreText {
    dispatch_async(dispatch_get_main_queue(), ^{
        tvLabelOutput.text = [NSString stringWithFormat:@"%@%@\n",
                              self.tvLabelOutput.text, moreText];   //add text to the text view
        [tvLabelOutput scrollRangeToVisible:NSMakeRange(tvLabelOutput.text.length-1, 1)];
    });
}

//Clear Text View
- (IBAction)actionClearText:(id)sender {
    tvLabelOutput.text = @"";   //clears the text view
}
```

Figure 28 - Text View Actions

1.8 View Lifecycle Methods

There are four methods for the view lifecycle, these are viewDidLoad which is the main method, it is called as soon as the view is loaded for the first time, viewDidAppear which is called when a view is transitioned to from another view, viewDidUnload and shouldAutorotateToInterfaceOrientation.

As said, the main method for my application here is the viewDidLoad. In this method, we set the head and tail of the linked list used for recording to be null, set up a dispatcher and load the Pure

Data patch 'simple_synth' into the application (tutorial from 'Making Musical Apps' followed to set up PureData patch).

A '0' is sent to both the ampMod and temp recievers in the main Pure Data synthesiser to ensure that no ampitude modulation is taking place before a user decides they want the effect. To play the drums, we make use of the AVAudioPlayer and these are set up in the viewDidLoad method, the entire description is available in the viewDidLoad method for the drum section below. Finally, the fileSetup method is called.

The viewDidAppear method is used to reload the drum table, in case a user has deleted a file in the drum section and then transitioned to the MIDI section. This ensures the files in both sections are kept up to date.

```
#pragma mark - View lifecycle
- (void)viewDidLoad
{
    [super viewDidLoad];

    head = NULL;    //set the head of linked list to be NULL
    tail = NULL;     //set the taile of linked list to be NULL

    //set up dispatcher/patch for normal midi play
    //Tutorial from Making Musical Apps book followed to load a Pure Data patch into the xCode
        application
    normalDispatcher = [[PdDispatcher alloc]init];
    [normalDispatcher addListener:self forSource:@"pitch"];
    [PdBase setDelegate:normalDispatcher];
    normalPatch = [PdBase openFile:@"simple_synth.pd" path:[[NSBundle mainBundle] resourcePath]];

    [PdBase sendFloat:0 toReceiver:@"ampMod"];  //ensure no effect is on at the start
    [PdBase sendFloat:0 toReceiver:@"temp"];

    // Set up avaudioplayers for the drums
    NSString *soundFilePath = [[NSBundle mainBundle] pathForResource:@"drum1" ofType:@"wav"];
    NSURL *soundFileURL = [NSURL fileURLWithPath:soundFilePath];
    player = [[AVAudioPlayer alloc] initWithContentsOfURL:soundFileURL error:nil];
    player.numberOfLoops = 0;
    [player prepareToPlay];

    NSString *soundFilePath2 = [[NSBundle mainBundle] pathForResource:@"drum2" ofType:@"wav"];
    NSURL *soundFileURL2 = [NSURL fileURLWithPath:soundFilePath2];
    player2 = [[AVAudioPlayer alloc] initWithContentsOfURL:soundFileURL2 error:nil];
    player2.numberOfLoops = 0;
    [player2 prepareToPlay];

    [self fileSetup];   //call fileSetup to populate arrOfFiles with the files saved on the iPad
}

-(void)viewDidAppear:(BOOL)animated{    //called when view is transitioned to from another view
    [self fileSetup];   //call fileSetup to populate arrOfFiles with the files saved on the iPad
    [tvFileLoad reloadData];  //reload the data in the file table
}

- (void)viewDidUnload{
    [super viewDidUnload];
    [PdBase closeFile:normalPatch];
    [PdBase setDelegate:nil];
}

- (BOOL)shouldAutorotateToInterfaceOrientation:(UIInterfaceOrientation)interfaceOrientation{
    return (interfaceOrientation != UIInterfaceOrientationPortraitUpsideDown);
}
```

Figure 29 - View Lifecycle

2. Pure Data Patch

General Overview

Pure Data is an essential part of my application; it is used for all of the audio processing and generation. Within my Pure Data patch, there are 3 synthesisers:

- The main synthesiser - effects used in this synthesiser are: volume envelope, chord, sound type and amplitude modulation.
- The diminish synthesiser - used for the diminished effect
- The arpeggio synthesiser - used for the arpeggio effect, can also use the different sound types in this effect

Objective-C uses [PdBase sendFloat:parameter to Reciever@"receiver"] and [PdBase sendBangToReceiver@"receiver"] to send the parameters and triggers to the receivers in the Pure Data patch.

2.1 How to Create a Simple Pure Data Patch

Pure Data uses a graphical user interface to allow developers to place objects onto the screen without having to code their specific locations into the patch. To create an object, you place the object onto the screen (using the put option from the toolbar), if the object required a name then, you would need to specify the name, examples include:

- osc~ - to create a cosine wave
- vline~ - to create a volume ramp
- r temp - to create a receiver called temp

The input to the Pure Data objects are called inlets and the outputs are called outlets. The main element of Pure Data that a developer needs to know is, when a value/trigger is inserted to the left inlet of a Pure Data object, the object is triggered. If a value is inserted to n inlet which isn't the leftmost one, the object won't be triggered; these types of inlets are called cold inlets.

2.1.1 Simple Patch

Figure 30 below shows what objects can be placed into the Pure Data patch. As said, when you want to add something to the patch then you place the required object onto the screen. For my simple patch, I am creating a simple cosine wave to output a single sound to the Macintosh computer.

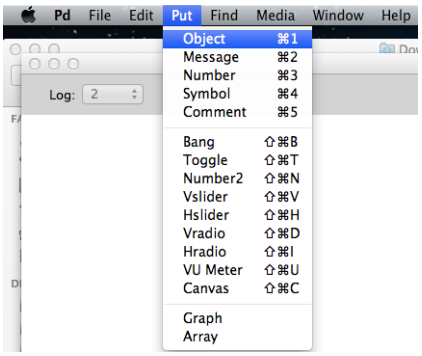


Figure 30 - Objects that can be Placed on Screen

For the patch, I need two objects. I name the first osc~ 220, which creates a simple cosine wave at the default frequency of 220Hz. The other object I need is a dac~ to convert the wave to digital sound. To allow the data to pass from the osc~ object to the dac~ object, I have connected the outlet of the osc~ object with the right and left inlet of the dac~ object. With the left and right inlets of the dac~ connected, a sound is outputted through the speakers. The completed patch is shown in figure 31.

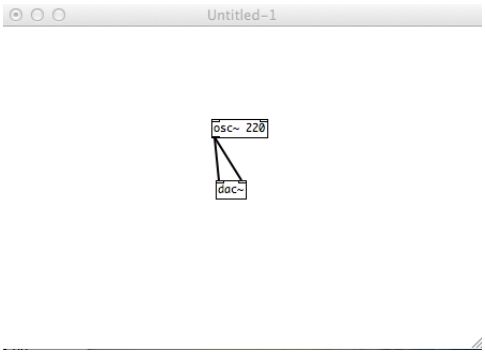


Figure 31 - Simple Patch

After creating a few patches like the one in figure 31, I moved on to more complex patches. There is an extensive list of objects that Pure Data makes use of, but the principle of connecting each object is the same.

2.2 Arpeggio Synthesiser

The arpeggio synthesiser allows a user to play a random note from an arpeggio or an arpeggio going up in scale. There are four receivers; each gets inputted to the upArpeggio sub-patch. The poly, pack and route objects are again used to ensure that four notes can be played at the same time. Unlike the main synthesiser, the values from the list are unpacked after the route and inputted into the sub-patch.

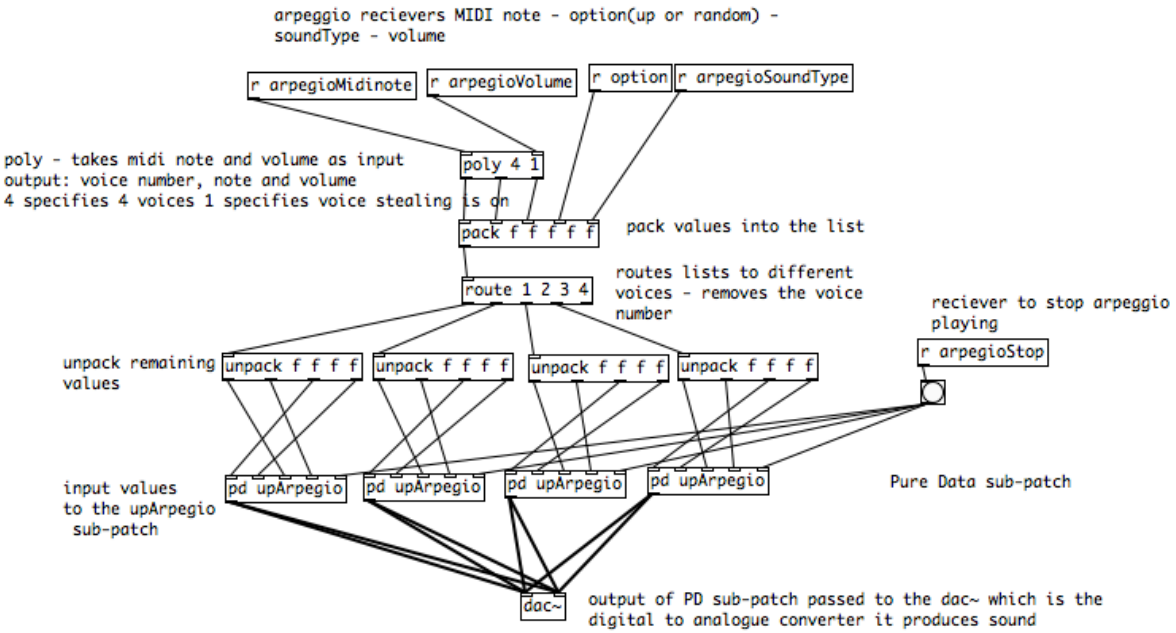


Figure 32 - Arpegigio Synthesiser

Inside the upArpegio sub-patch we find the code required for the up arpeggio and the random arpeggio effects, as we can see, the patch is complicated so I am going to break it down into two stages:

- Finding the arpeggio effect to use and the note
- Adding the sound type effect to the note

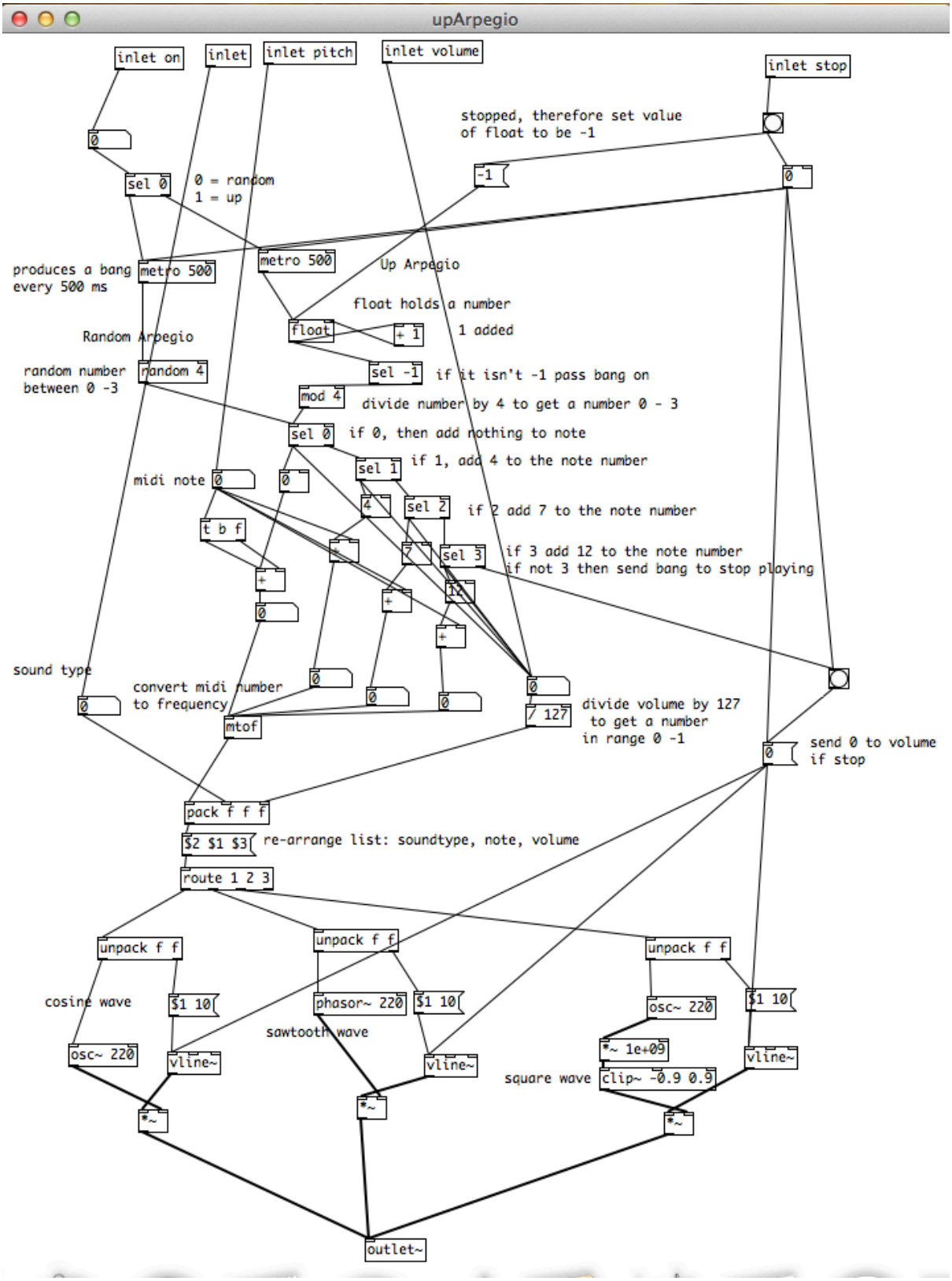


Figure 33 - Up Arpeggio Sub-Patch

(Figure 34) Here we can see the top half of the upArpeggio sub-patch. The arpeggio effect is decided upon using the inlet 'On', if the value of this is 0 then we use the random arpeggio which, every 500ms chooses a random number between 0 and 3. If the up arpeggio effect is to be used, then

every 500 ms, the value stored in float is incremented by 1, this is then passed to a 'sel' object which compares it to the value -1, if it isn't -1 it passes the value to the 'mod' object which does the 'mod' operation on the number to gain a number between 0 and 3.If the arpeggio effect is stopped, then the float value is changed to be -1 and a 0 value is placed in the left inlet of the metro object so that it stops producing a 'bang' every 500ms.

Once we have a value between 0 and 3, the value then passes through a series of 'sel' objects which compares the value to 0, 1, 2 and 3 if it is one of these then it takes the MIDI note number and adds the corresponding value (0 for 0, 4 for 1, 7 for 2 and 12 for 3), if it isn't then it sends a 'bang' to the stop trigger. Once the MIDI number has been obtained it is then passed to the 'mtof' object to convert it to the frequency. This along with the volume and sound type are then packed into a list.

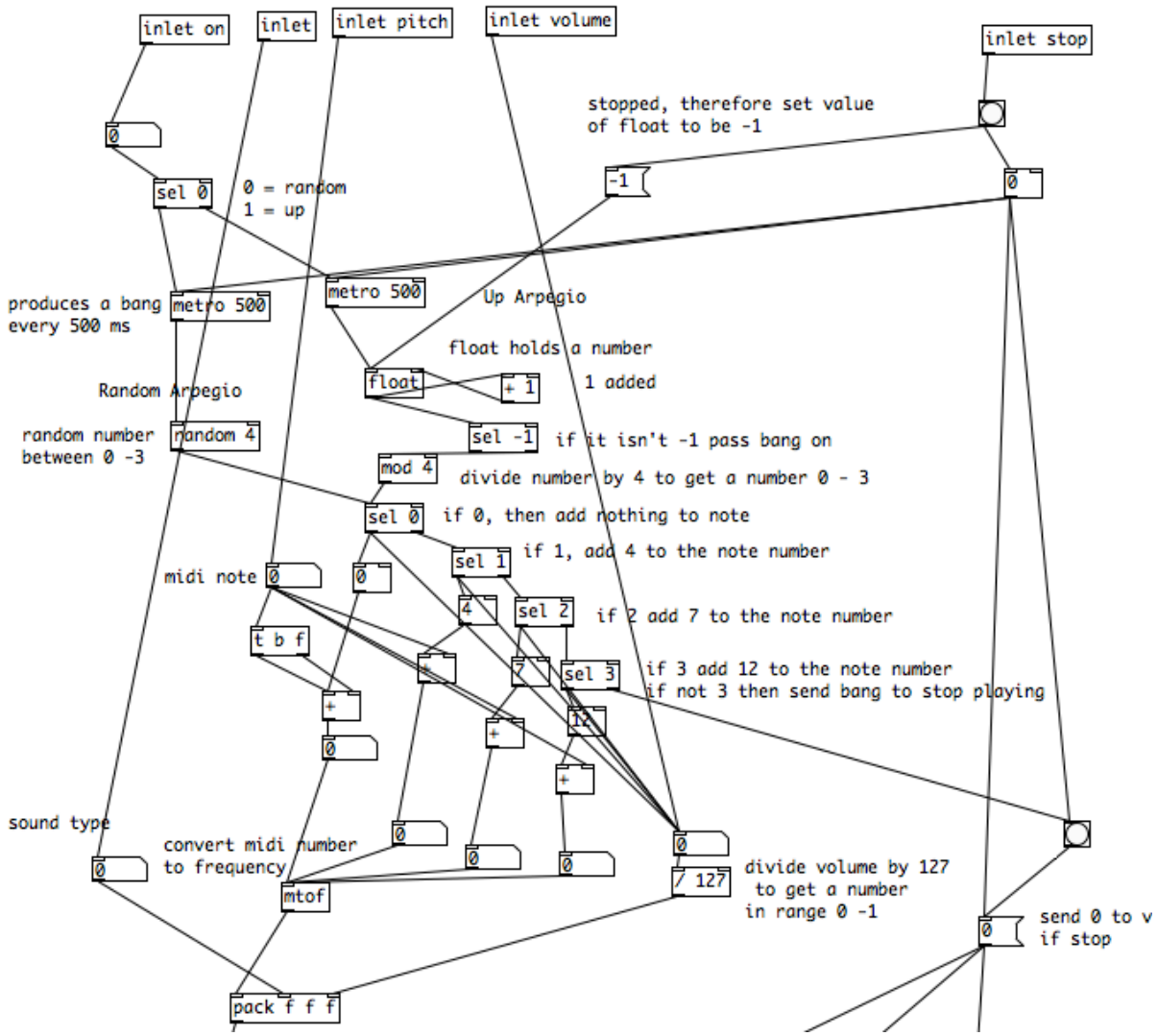


Figure 34 - Select Which Note to Play

The bottom half of the patch first re-orders the list and then plays the MIDI note depending on the sound type as we see in the main synthesiser. It outputs the signal to the signal outlet.

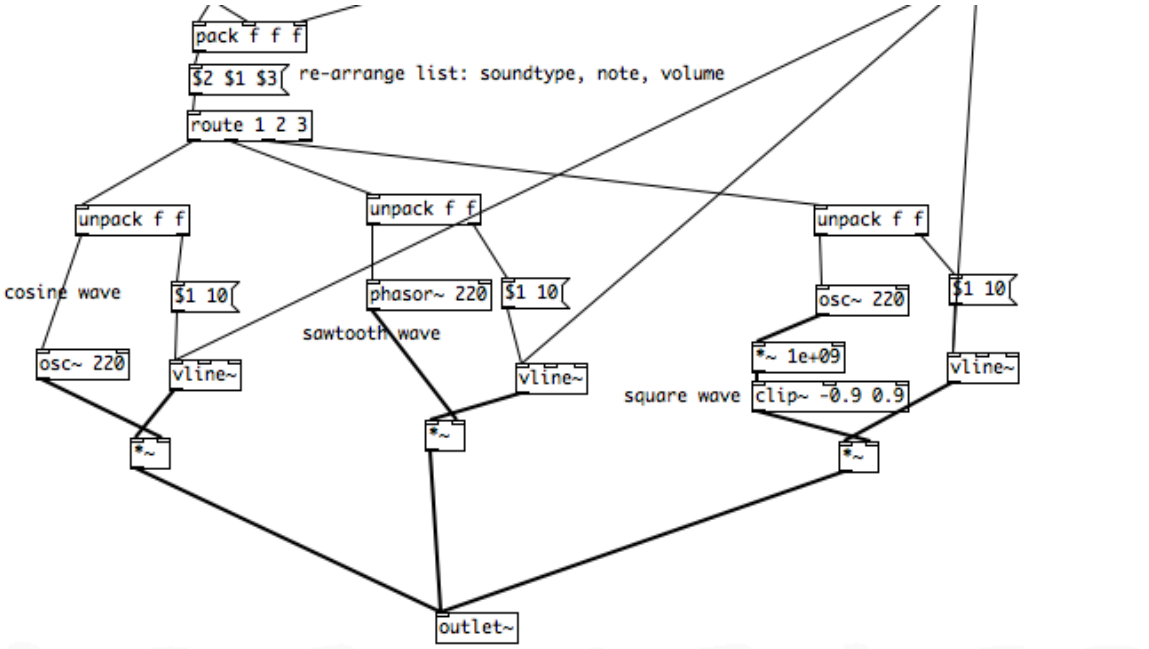


Figure 35 - Select Which Sound Type to Play

2.3 Diminish Synthesiser

The simplest synthesiser I have implemented is the diminish synthesiser. This part of my Pure Data patch is responsible for adding the diminish effect to the users input. Two parameters are used in this synthesiser, the MIDI note and the volume.

To allow the user to play more than one note at a time, the poly object is used. In my implementation I have used [*poly 4 1*], which allows for four notes to be played at the same time, the 1 indicates that voice stealing is on. Voice stealing is when a fifth note is played, it overwrites the first note. The output of the poly object is passed to the [*pack f f f*] which creates a list of the three values it is sent, this list is of the order: voice number, MIDI note number then volume. The final stage before audio processing begins is the routing of the list to a specific path; the route object is used for this. When the list is passed to the route object, the first item is removed (voice number), it then checks this value and sends the rest of the list on the correct path (i.e. a list with voice number 1 will be passed to the first route etc.).

Once the information has been routed to the correct path, the unpack method is used to retrieve the values from the list. The MIDI note number is passed through the mtof object and osc~ object, this process converts the number to the frequency and creates a cosine wave. The volume on the other hand, is passed to a message object, it is inserted to this object and replaces the $1 holder, this message means that it will ramp the volume to the volume specified in 5ms, then ramp back down to 0 in 5000ms after a 10ms wait. This is then passed to the vline~ which creates the volume ramp. Finally, the cosine wave and volume ramp are multiplied together and passed to the dac~ object which is a digital to analogue converter which converts the digital signals to analogue sounds.

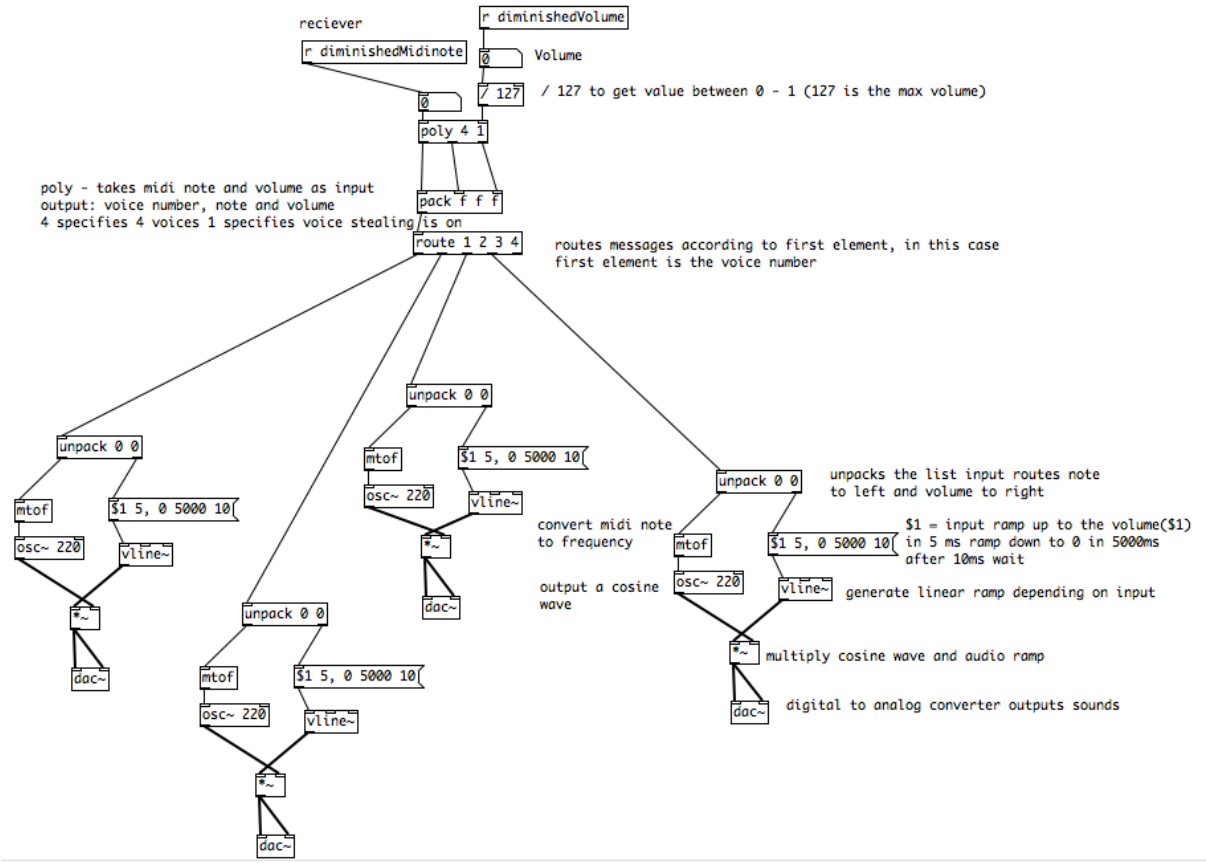


Figure 36 - Diminish Synthesiser

2.4 Main Synthesiser

The main synthesiser for my application is displayed below. This synthesiser includes the ability to change the sound type, the envelope function, the amplitude modulation and the chord effect. The main synthesiser is split into three sections, the overall synthesiser, then the Pure Data sub-patch chooseEffect and inside this sub-patch there are another three sub-patches for each different sound type. The overall synthesiser is pretty basic, there are 10 receiver objects. To stop a note playing, one of these is used, the MIDI note of which to stop is read in from the midiOffNote receiver which is then passed to a message object. This then stops the play of the specific note. To stop all notes, the 'stopAll' receiver is given a 'bang' from the objective-C code which triggers the stop message, which in turn, stops all notes from being played.

To play a note, 8 of the receivers are used. First, the MIDI note and volume are read in by a poly object which outputs the voice number, MIDI note number and volume, the voice number is used for polyphonic output. The output is then packed with the other 6 receivers in the order: voice number, sound type (type of sound to be played, normal, sawtooth, square), attack, decay, sustain, release(envelope parameters) and chord (used to trigger the chord effect or not).

After the values are packed, they become a list which is then routed to different paths using the route object. This route object removes the first element from the list and passes it to the corresponding outlet (if 1 is at the start, the list would be passed to the first sub-patch).

Once the sub-patch has been processed, there are two signal outlets which are passed to the dac~ object which converts the signals to sound.

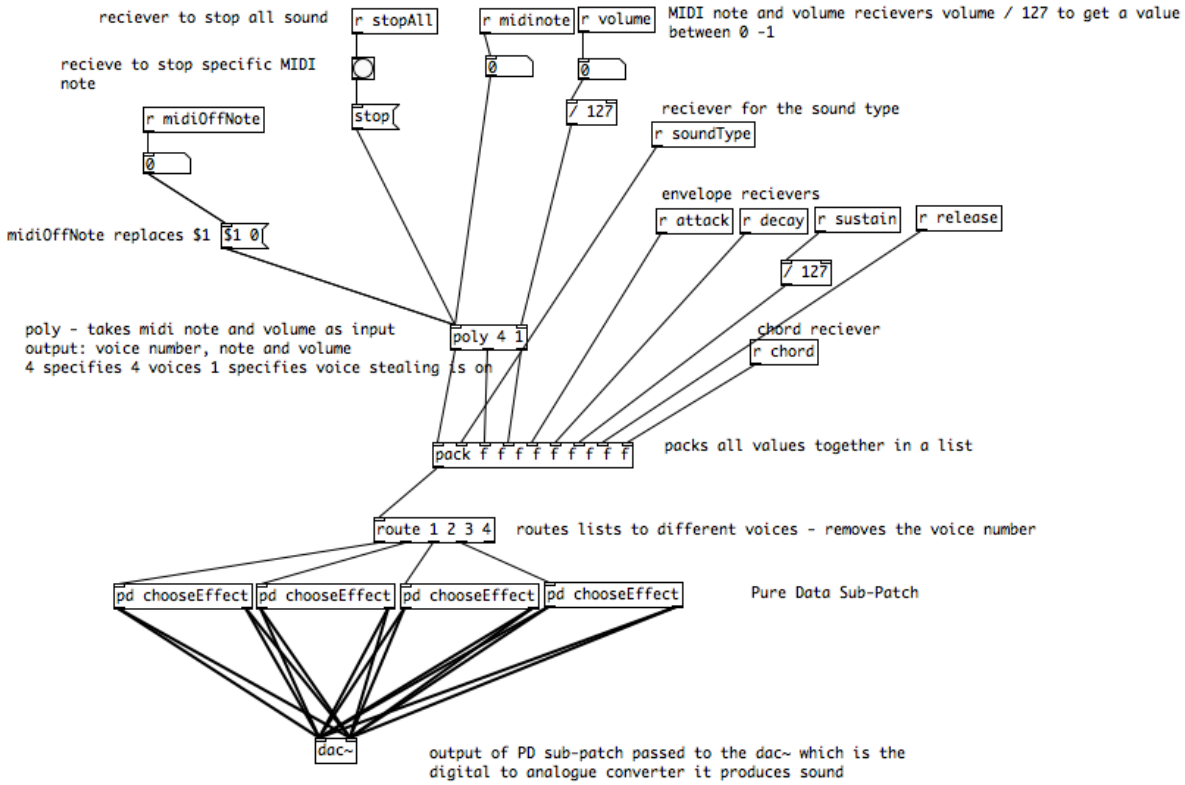


Figure 37 - Main Synthesiser

chooseEffect Sub-Patch

The 'chooseEffect' patch is used to route the list of values to the correct sound type sub-patch. It takes a list as the input with the value for the sound type at the front of the list. The list is passed to the route object, which removes the first value (sound type) and forwards it to the corresponding sub-patch (if sound type = 1 then it would route it to the normal sub-patch, sound type = 2 the phasor sub-patch, sound type = 3 the square sub-patch).

When the sub-patch has completed its processing, it returns two signals, these two signals are then returned to the main synthesiser using two signal outlets.

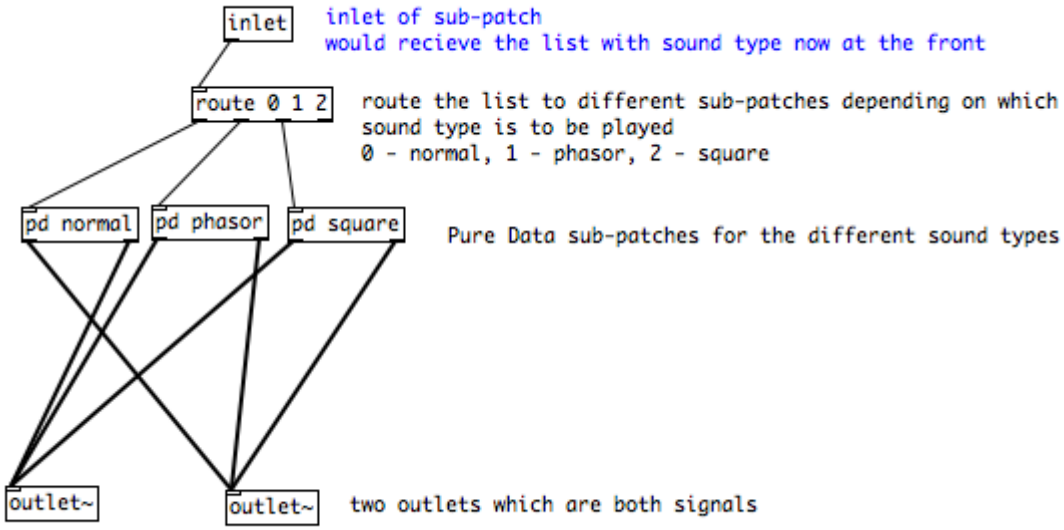


Figure 38 - Choose Effect Sub-Patch

Choose Effect Sub-Patches

The three sub-patches of the 'chooseEffect' patch are almost identical. In these sub-patches, the main signal processing is done to manipulate the input and create an audio signal depending on the input parameters. For simplicity of explanation, I have split the patches into three sections, the modulation area, MIDI note section and volume section. In the modulation area, a cosine wave is created based on the modulation input from the Objective-C application. This is then used to modulate the other cosine waves created using the MIDI note that a user has played.

The volume envelope is created in the volume section. Firstly the volume is inputted into the 'sel 0' object to test whether the volume is above 0 (note played) or 0 (note off), depending on which case is satisfied the pack objects are triggered. To create the envelope, the volume will be above 0 and the [pack f f f f f] object will be triggered. The list created will be the volume, attack, decay and sustain values. These are then re-ordered and placed in a message object. This message object is then passed to the vline~ object to create the volume envelope. If a note off is played, then the volume will be 0 and the [pack f f] object will be triggered. The list will consist of the 0 value and the decay time, these are passed into a message object and then to the vline~ object to create the ramp down to 0.

Finally the MIDI note section. Two parameters are passed to this part, the MIDI note number and the chord number. These two are packed to create a list and re-ordered so the chord number is at the front of the list. I have then made use of the route object again, if the chord number is 0, then only the MIDI note number is used to create a cosine wave, otherwise the MIDI note number and the two notes above it in the arpeggio scale are used to create three cosine waves. After each note number has been translated to its frequency, they are modulated by the modulating signal before the cosine waves are created. These waves are then multiplied by the volume envelopes and passed to the signal outlets.

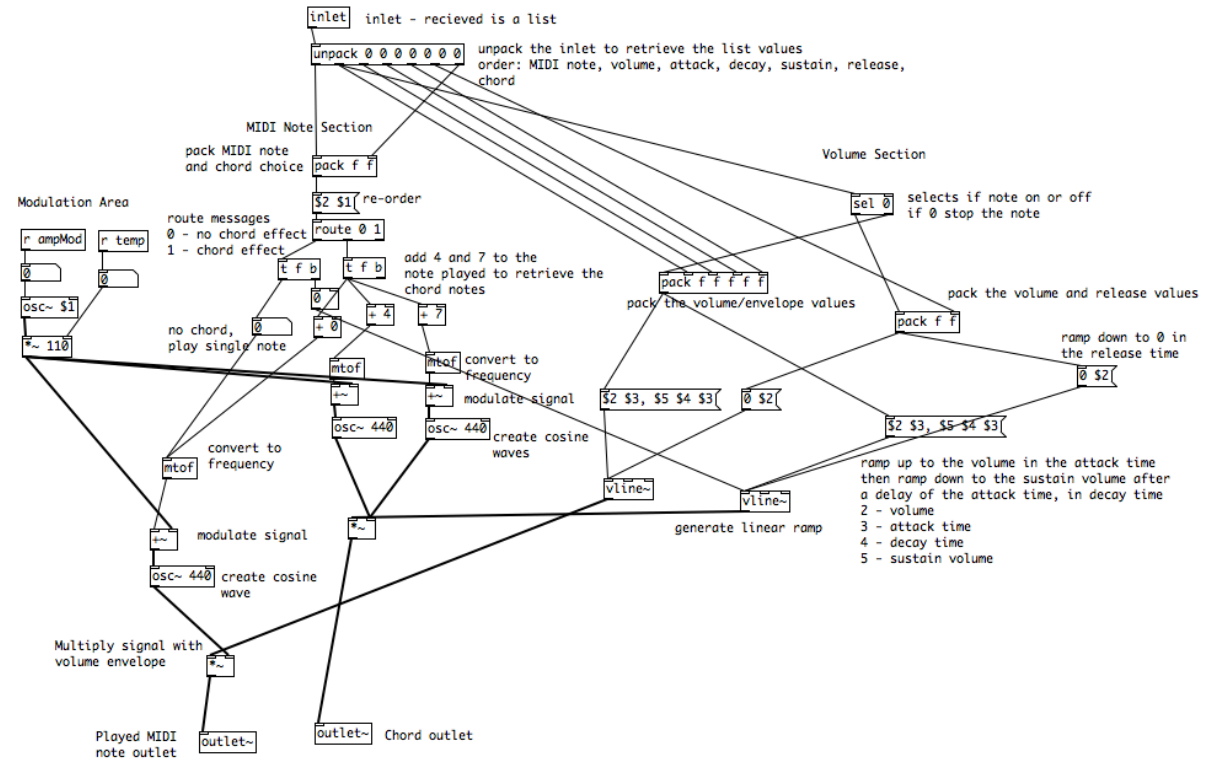


Figure 39 - Normal Sub-Patch

In the phasor sub-patch, the only difference is that a phasor~ object is used to create a sawtooth wave instead of an osc~ object.

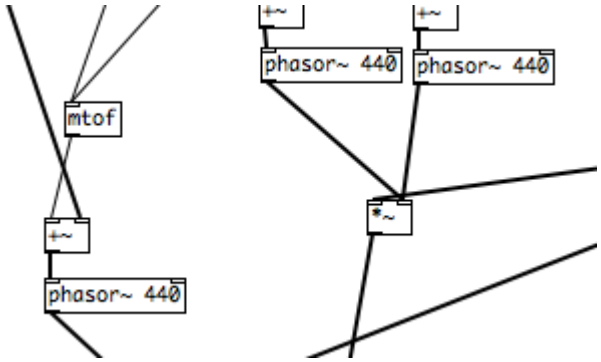


Figure 40 - Phasor Changes

Similarly, the square sub-patch has only minor changes to create a square wave form.

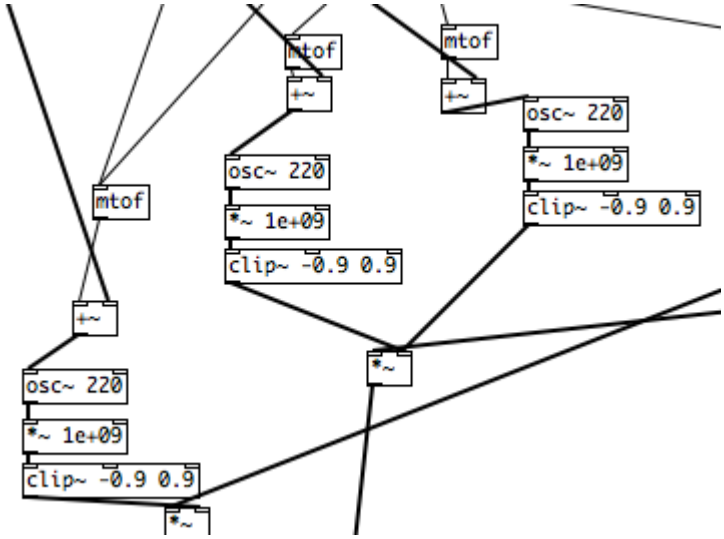


Figure 41 - Square Changes

3. Drum Section

The drum section of the application is where the generation of the drum tracks occur. I have made use of a simplistic interface to allow the user to create drum tracks easily.

Figure 42 - Drum Interface

3.1 Choosing a Drum Pattern

3.1.1 Storing the Status of the Drums

In my application, I decided upon using a drum array for each of the drums to hold the status of the drum at different beats. In each drum array there are 16 objects, each object in the array stores either a 1 (to indicate that the drum is on at that specific beat) or a 0 (to indicate the drum is off). The values in the array are changed when a user touches one of the views in the drum grid (UICollectionView).

3.1.2 Test drums

To allow the user to hear what the drum sounds like before deciding upon using it, I have implemented a method for each drum that plays the drum for its duration. When a user touches the button to test the drums, the method actionTestDrum1 (where '1' is the drum number) is used; this calls the play method of the AVAudioPlayer for the specified drum.

```
#pragma mark - Drum Sound Checks
//test the sound of the drums
- (IBAction)actionTestDrum1:(id)sender {
    [player play];
}

- (IBAction)actionTestDrum2:(id)sender {
    [player2 play];
}
```

Figure 43 - actionTestDrum

3.1.3 Interface for drum selection[4]

To create the grid of drums, I firstly tried to use a series of switches which would be changed by the user (from on to off depending on their choice). I did get these to work, but unfortunately the switches didn't provide my application with the best of usability and it wasn't aesthetically pleasing. Therefore I decided to think of a different way to get around the problem. I finally decided upon using the UICollectionView object, which as the name suggests, is a collection of views with each view in the collection view being of the type UICollectionViewCell, by default the background colour is white with a green border. Figure 43 below has the four methods that are required for a UICollectionView to work in an application.

```
//UICollectionView methods
#pragma mark - UICollectionView
- (NSInteger)collectionView:(UICollectionView *)view numberOfItemsInSection:(NSInteger)section {
    return 16;  //16 drums per sections
}

- (NSInteger)numberOfSectionsInCollectionView: (UICollectionView *)collectionView {
    return 2;   //2 drums
}

- (UICollectionViewCell *)collectionView:(UICollectionView *)cView cellForItemAtIndexPath:(NSIndexPath *)indexPath {
    drumCell *cell = (drumCell *)[cvDrumGrid dequeueReusableCellWithReuseIdentifier:@"Cell"
        forIndexPath:indexPath];    //find the cell the user clicked
    return cell;    //return the cell the user clicked
}

- (void)collectionView:(UICollectionView *)collectionView didSelectItemAtIndexPath:(NSIndexPath *)indexPath{
    drumCell *cell = (drumCell *)[cvDrumGrid cellForItemAtIndexPath:indexPath];
    int drumSelected = indexPath.section;

    //Change colour of views if the user clicked the specific view
    switch (drumSelected) {
        case 0: //drum 1
            if([[marrDrum1 objectAtIndex:indexPath.row] integerValue] == 0){    //if cell selected at beat i then play
                drum one
                [marrDrum1 replaceObjectAtIndex:indexPath.row withObject:@"1"]; //change value in drum array
                cell.drumView.BackgroundColor = [UIColor redColor];     //change background colour of cell
            }else{ //cell de-selected
                [marrDrum1 replaceObjectAtIndex:indexPath.row withObject:@"0"]; //change value in array
                cell.drumView.BackgroundColor = [UIColor whiteColor];   //change colour back to normal
            }
            break;

        case 1: //drum 2
            if([[marrDrum2 objectAtIndex:indexPath.row] integerValue] == 0){    //if cell selected at beat i then play
                drum one
                [marrDrum2 replaceObjectAtIndex:indexPath.row withObject:@"1"];
                cell.drumView.BackgroundColor = [UIColor redColor];
            }else{
                [marrDrum2 replaceObjectAtIndex:indexPath.row withObject:@"0"];
                cell.drumView.BackgroundColor = [UIColor whiteColor];
            }
            break;

        default:
            break;
    }
}
//End required UICollectionView methods
```

Figure 44 - UICollectionView Methods

As I decided on using 16 beats for each drum, the method numberOfItemsInSection is self-explanatory, it means that each section of the collection has 16 items in it(or views) as there as 16 beats. The numberOfSectionsInCollectionView method defines how many sections you want in the collection, each section is related to one drum sample. As I have struggled to find good drum samples, I have only implemented 2 drums for this application. The final self explanatory method here is the cellForItemAtIndexPath which returns a cell at a specific index path (e.g. section 1, row 1).

The final method required for the UICollectionView is the didSelectItemAtIndexPath. This method is called when a user touches any of the views in the collection table to change the status of the specific drum at that specific beat. Firstly, the cellForItemAtIndexPath method is used to return a reference of the drum cell that the user has touched and then the drum which was touched is discovered by getting the section of the index path. The switch statement is pretty simple, the basic idea is that it checks what drum has been changed and then updates the drum array relating to that drum. Figure 45 shows the if statement used, this checks the value that is contained at the specified beat (if beat 3 had been changed, then indexPath.row would be 3), if the drum is off at that beat (i.e == 0) then the value in the array is changed to be 1 and the background colour of the view is set to be red so that the user notices the change. If the drum is playing on that beat, the reverse of the process above happens, a 0 is placed in the array and the background colour reverts to white.

```
if([[marrDrum1 objectAtIndex:indexPath.row] integerValue] == 0){
```

Figure 45 - Check Value in Drum Array

3.1.4 Play / Stop

After a user has created their drum track (or even during), they may select to play the track to hear what it sounds like. If they select to play the drums the actionPlayDrums method is called. The main part of this method is the infinite loop that is based around the boolInfiniteLoop Boolean variable, while this is true, the drums keep playing. To allow a user to keep doing other things to the drum track while this infinite loop is running, I make use of dispatch_async() which allows concurrent execution of more than one task, this is needed as otherwise, when the loop is running, no other interaction with the application will be possible and eventually the application will crash.

To play the drums, a for loop is used, which at every pass checks what drums are supposed to be on and plays them accordingly. This is achieved by comparing the values in the separate drum arrays to see if the value stored in it is a 1 at that specific beat, if it is, the AVAudioPlayer for that specific drum is played. After checking all of the drums, the [NSThread sleepForTimeInterval:floatBeatsPerMin] method is called, the parameter floatBeatsPerMin(explained in BPM above) is passed to it and the thread sleeps for that amount of time.

The final part of this code is the if condition to see if the stop button has been pressed, if it has there is a 'break;' statement to break out of the loop so that the drums stop playing.

```
#pragma mark - Play Drums
- (IBAction)actionPlayDrums:(id)sender {
    boolInfiniteLoop = true;    //keep playing - condition for the while loop

    btnPlay.alpha = 0.5;    //to ensure that you can't initiate the code below twice, disable the button
    btnPlay.enabled = false;

    btnStop.alpha = 1;    //enable the stop button so you can break the loop below
    btnStop.enabled = true;

    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,(unsigned long)NULL), ^(void) {    //do
        another task - allows other tasks to be performed while the code below is executed
        do {
            for (int i = 0; i < 16; i++) {   //16 beats

                //if the first drum is selected at the current beat, then play it
                if([[marrDrum1 objectAtIndex:i] integerValue] == 1){
                    [player play];
                }

                //if the second drum is selected at the current beat, then play it. etc..
                if([[marrDrum2 objectAtIndex:i] integerValue] == 1){
                    [player2 play];
                }

                if([[marrDrum3 objectAtIndex:i] integerValue] == 1){
                    [player3 play];
                }

                [NSThread sleepForTimeInterval:floatBeatsPerMin];    //sleep for certain period of time to ensure beat
                    (e.g. if beats per minutes is 60 then it would sleep for a second, 120 sleep for 0.5 etc.)
                [player stop];  //if players are still playing, then stop them (in case sample is longer than
                    necessary)
                [player2 stop];

                if (!boolInfiniteLoop) {   //if booldrumloop set to false it will stop playing (stop button pressed) -
                    break out of loop
                    break;
                }
            }
        } while (boolInfiniteLoop); //continue until user clicks the stop button
    });
}
```

Figure 46 - actionPlayDrums

To stop the playing of the drums, the user must click the 'Stop' button which then sets boolInfiniteLoop to be false. By setting this variable to be false, the application breaks out of the infinite while loop in the play method using the 'break;' statement as explained above.

```
#pragma mark - Stop Drum Playback
- (IBAction)actionStopDrums:(id)sender {  //stops the drums by setting boolInfiniteLoop to false which breaks the while loop in the
    actionPlayDrums method
    btnPlay.alpha = 1;    //disable the play button
    btnPlay.enabled = true;

    btnStop.alpha = 0.5;    //enable the stop button
    btnStop.enabled = false;

    boolInfiniteLoop = false;
}
```

Figure 47 - actionStopDrums

3.1.5 Reset Drum Grid

As with any changes made, the user may wish to revert back to the initial state, therefore the application makes use of a reset method to reset the drum arrays and the interface to its default state. Two methods are used in the application to reset the drum generator. Firstly, when a user touches the reset button the actionResetGrid() method is called. Through this all of the variables are reset (volume, bpm and drum arrays) and the interface is reset with the play button being enabled and stop button being disabled. To reset the UICollectionView, the method calls the resetCollectionRow() method with the parameters for the row and section given so that each cell in the UICollectionView can be turned back to having a white background. Resetting the collection view and drum arrays are done inside a for loop, the 'i' variable is used as the index of the current beat to be reset in the drum arrays and it is also used as the index for the row (beat) to reset in the collection view.

```
#pragma mark - reset interface
- (IBAction)actionResetGrid:(id)sender {  //reset drum arrays and the views in the uicollection view
    for (int i = 0; i < 16; i++) {      //do 16 times as there are 16 boxes representing beats
        [marrDrum1 replaceObjectAtIndex:i withObject:@"0"]; //replace 1 with a 0 in the array representing drum 1
        [self resetCollectionRow:i resetCollectionSection:0];   //call method to reset the required view in the
            uicollectionview

        [marrDrum2 replaceObjectAtIndex:i withObject:@"0"];
        [self resetCollectionRow:i resetCollectionSection:1];
    }

    //resets the volumes for each drum and resets the UISliders representing these volumes
    sliDrum1Vol.value = 5;
    floatDrum1Volume = 0.5;

    sliDrum2Vol.value = 5;
    floatDrum2Volume = 0.5;

    //resets the bpm value and the edit box for the bpm
    tfBPM.text = @"60";
    floatBeatsPerMin = 1;

    boolInfiniteLoop = false;

    btnPlay.alpha = 1;
    btnPlay.enabled = true;
    btnStop.alpha = 0.5;
    btnStop.enabled = false;
}
```

Figure 48 - actionResetGrid Method

```
-(void) resetCollectionRow:(int)row resetCollectionSection:(int)section{    //reset view in the uicollectionview
    NSIndexPath *indexPath= [NSIndexPath indexPathForRow:row inSection:section];    //get the index of the view to reset
    drumCell *cell = (drumCell *)[cvDrumGrid cellForItemAtIndexPath:indexPath]; //create a new drumCell for that index
    cell.drumView.BackgroundColor = [UIColor whiteColor];   //set the colour to be white
}
```

Figure 49 - resetCollectionRow Method

3.2 Save / Load / Delete a Track

3.2.1 Save

A key part of the application is to save the drum tracks generated for future use. When a user decides to save a track the actionSaveFile method is called. The code below in figure 50 illustrates the process that is undertaken when a user saves their track, saving the drum track involves saving one array to a specific file. Firstly a 'voumeArray' is created to hold the beats per minute of the track and the volumes of each drum. To add these into the array, three NSString variables have to be created with the values of the different variables to be stored, then the three are added to the 'volumeArray'. Then another array is created called 'array' to store all of the information on the track, the 'volumeArray' is added to it, and then the two drum arrays.

The users chosen file name is retrieved from the 'tfTextSaveFile' text field edit box and stored as a NSString variable and the path of where to save the text file is retrieved using the NSSearchPathForDirectoriesInDomains, getting the object at the first index of the array returned from this method and storing it in an NSString variable.

Finally, the array is written to the specified path using the 'writeToFile' method of the NSMutableArray class, the file name is appended to the end of the path found from the method above. I have included a simple if statement to check whether that track name is already in the table displaying all of the saved files, if it isn't, it is added to the array that populates the table and the table is reloaded, if the name already exists in the table then nothing is done as the track will have just been overwritten into that file (an error message could easily be added to ensure the user knows they are overwriting another track).

```
#pragma mark - File Options
- (IBAction)actionSaveFile:(id)sender {  //Save drum beat into a file, it saves it as and array of arrays

    NSMutableArray *volumeArray = [[NSMutableArray alloc]init]; //allocate a space for the volume array that contains
        the volume of each drum and the bpm of the beat
    //create a string each to hold the bpm and volumes
    NSString *strBeatsPerMin = [[NSString alloc]initWithFormat:@"%i", tfBPM.text.integerValue];
    NSString *drum1StringVolume = [[NSString alloc]initWithFormat:@"%e",floatDrum1Volume];
    NSString *drum2StringVolume = [[NSString alloc]initWithFormat:@"%e",floatDrum2Volume];

    //adds the bpm and volumes to the volume array
    [volumeArray addObject:strBeatsPerMin];
    [volumeArray addObject:drum1StringVolume];
    [volumeArray addObject:drum2StringVolume];

    //create array for saving to the file, it contains the volume array and every drum array
    NSMutableArray *array = [[NSMutableArray alloc]init];

    //add the volume array and drum arrays to the array
    [array addObject:volumeArray];
    [array addObject:marrDrum1];    //array drum1 at position 0 of array array
    [array addObject:marrDrum2];

    //find the path of where to save the file
    NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES);
    NSString *documentsDirectory = [paths objectAtIndex:0];
    //the name of the file
    NSString *fileName = tfTextSaveFile.text;

    [array writeToFile:[documentsDirectory stringByAppendingPathComponent:fileName] atomically:YES]; //save the array
        to a file

    //if file not already named, then add it to the marrOfFiles and reload the table    (//if file already in the
        table, it will overwrite it)
    if (![marrOfFiles containsObject:tfTextSaveFile.text]) {
        [marrOfFiles addObject:tfTextSaveFile.text];
        [tvFiles reloadData];
    }
}
```

Figure 50 - actionSaveFile

3.2.2 Load

To allow the user access to their saved drum tracks, the application has a load feature which loads the drum track to the screen to allow the user to manipulate it further if they wish. When a user wishes to load a drum track, the first method called is the actionResetGrid method to reset the collection view on the screen. The track to be loaded is discovered using a method call on the UITableView that is on the screen, this is explained in detail further on in the report. The 'filePath' of the file is discovered by using the NSSearchPathForDirectoryInDomains method again to get to the place where the files are saved (this is stored in an array called paths), the file is then loaded into a NSString variable 'filePath', after the name of the file to be loaded (strTableSelectedFile) is appended

to the first element from the paths array. From here, the reverse process of the save method is used, where we allocate an array to store the contents of the 'filePath' variable. In this array, there will be three other arrays, the 'loadedVolume' array and two drum arrays. The main drum arrays are then loaded with a mutable copy of the drum arrays found in the file.

From the 'loadedVolume' array, we retrieve the beats per minute(bpm) of the saved track and set the text of the 'tfBPM' text field to be this, we also set the main 'floatBeatsPerMin' variable to be 60 divided by the bpm value.

The two volumes for the different drums are then retrieved from the array and each drum is assigned it's volume. The value of the volume sliders on the screen are also changed so they display the correct value.

The final thing to do is to update the collection view. This is done again in a for loop which loops over each element in the two drum arrays and sets the background colour of a specific cell to be red if the drum is meant to be played at that specific beat.

```
- (IBAction)actionLoadFile:(id)sender {
    [self actionResetGrid:sender];    //calls the actionResetGrid to reset the uicollectionview before setting it up
        with the contents of the loaded file

    NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES);   //path to
        documents directory
    NSString *filePath = [[paths objectAtIndex:0]stringByAppendingPathComponent:strTableSelectedFile];   //path of file

    NSArray *fileArray = [[NSArray alloc]initWithContentsOfFile:filePath];      //the array of the drum beat
    NSMutableArray *loadedVolume = [[NSMutableArray alloc] initWithArray:fileArray[0]]; //array of the bpm and drum
        volumes
    NSMutableArray *loadedDrum1 = [[NSMutableArray alloc] initWithArray:fileArray[1]];  //array of drum 1
    NSMutableArray *loadedDrum2 = [[NSMutableArray alloc] initWithArray:fileArray[2]];  //etc

    marrDrum1 = [loadedDrum1 mutableCopy];  //mutable copy of loaded drum one, drum 1 has the same values as
        loadeddrum1
    marrDrum2 = [loadedDrum2 mutableCopy];  //etc

    NSString *bpm = [loadedVolume objectAtIndex:0];
    tfBPM.text = bpm;
    floatBeatsPerMin = 60 / [bpm floatValue];

    NSString *vol1 = [loadedVolume objectAtIndex:1];
    floatDrum1Volume = [vol1 floatValue];
    sliDrum1Vol.value = (floatDrum1Volume*10);

    NSString *vol2 = [loadedVolume objectAtIndex:2];
    floatDrum2Volume = [vol2 floatValue];
    sliDrum2Vol.value = (floatDrum2Volume*10);

    for(int i = 0; i < 16; i++){    //16 beats
        int intDrum1IsOn = [[loadedDrum1 objectAtIndex:i] integerValue];    //returns 1 or 0 depending on value of
            array (1 is on, 0 is off)
        int intDrum2IsOn = [[loadedDrum2 objectAtIndex:i] integerValue];    //returns 1 or 0 depending on value of
            array (1 is on, 0 is off)

        if (intDrum1IsOn == 1) {    //if drum 1 is played at beat i
            NSIndexPath *indexPath= [NSIndexPath indexPathForRow:i inSection:0];    //get indexPath of the required
                view in the uicollectionview

            //change colour of the view at the required beat to indicate that it is on
            drumCell *cell = (drumCell *)[cvDrumGrid cellForItemAtIndexPath:indexPath];
            cell.drumView.BackgroundColor = [UIColor redColor];
        }

        if (intDrum2IsOn == 1) {    //if drum 2 is played at beat i
            NSIndexPath *indexPath= [NSIndexPath indexPathForRow:i inSection:1];    //get indexPath of the required
                view in the uicollectionview

            //change colour of the view at the required beat to indicate that it is on
            drumCell *cell = (drumCell *)[cvDrumGrid cellForItemAtIndexPath:indexPath];
            cell.drumView.BackgroundColor = [UIColor redColor];
        }
    }
}
```

Figure 51 - actionLoadFile

3.2.3 Delete

The final method that a user can use in the drum section is the delete method to delete a saved track. When a user decides to delete a file, an alert is displayed on the screen asking if they are sure that they want to delete the specified track. Once a user presses either the cancel or delete button on the alert, the [alertView:(UIAlertView *) clickedButtonAtIndex:] method is called, this will be detailed later. If the user decides to delete the file, the delete() method is called.

Again, the path of the file to be deleted is discovered and stored in the NSString variable 'filePath'. To delete a file, a file manager is used and a Boolean variable is setup to allow the application to send out a different response if the deletion was successful or not. This Boolean variable is set as the response to the deletion call, we make use of the removeItemAtPath: method of the file manager to delete the selected file. We also remove the file, from the array of files that is used to display the saved files and update the table.

To finish the delete method, there are two if statements, the first one displays an alert on the screen to inform the user if the file has been deleted. The second statement checks whether there are any more files in the array of files, if there isn't the load and delete buttons are disabled.

```
- (IBAction)actionDeleteFile:(id)sender {   //create an alert asking user if they are sure they want to delete the
    track
    UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Delete Track" message:@"Are you sure that you want to
        delete this track?" delegate:self cancelButtonTitle:@"Cancel" otherButtonTitles:@"Delete", nil];

    [alert show];

}
```

Figure 52 - actionDeleteFile

```
-(void) delete{
    //Get path of file to be deleted
    NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES);
    NSString *filePath = [[paths objectAtIndex:0]stringByAppendingPathComponent:strTableSelectedFile];
    NSFileManager *fileManager = [NSFileManager defaultManager];

    BOOL fileDeleted = [fileManager removeItemAtPath:filePath error:nil];   //removes file at position file path and
        returns yes if the file is deleted or no is the file isn't deleted

    [marrOfFiles removeObjectIdenticalTo:strTableSelectedFile];    //remove file from array (strTableSelectedFile is
        the file selected in the table)

    [tvFiles reloadData];  //reload table after deletion of file

    if (fileDeleted != YES)     //print if the file is deleted
    {
        UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"" message:@"There was a problem - try again"
            delegate:self cancelButtonTitle:@"Okay" otherButtonTitles:nil, nil];
        [alert show];   //create alert and display it saying that the file was deleted
    }else{
        UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"" message:@"Track Deleted" delegate:self
            cancelButtonTitle:@"Okay" otherButtonTitles:nil, nil];
        [alert show];   //create alert and display it saying that the file was deleted
    }

    /*if there aren't any more files in the table, then set the load and delete button to be disabled
     so that the app doesn't try and load/delete a file that isn't there(this would caus a crash)*/
    if ([marrOfFiles count] == 0) {
        btnLoadFile.alpha = 0.5;
        btnLoadFile.enabled = false;
        btnDeleteFile.alpha = 0.5;
        btnDeleteFile.enabled = false;
    }
}
```

Figure 53 - Delete Method

3.2.4 Alert View Method

When a button on a UIAlertView is pressed, the following method is called:

```
-(void)alertView:(UIAlertView *)alertView clickedButtonAtIndex:(NSInteger)buttonIndex{
    //action called after button of a UIAction is clicked
    if ([[alertView title] isEqual: @"Please enter a number in the range of 60 - 180"]) {   //check if it is the bpm alert box
        tfBPM.text = @"120";    //set value of bpm text box to be a valid number
        floatBeatsPerMin = 0.5;
    }else if ((([[alertView title] isEqual:@"Delete Track"]) && (buttonIndex == 1)) {   //else it is the delete alert box - check if it was the
        delete button that was pressed
        [self delete];  //call delete method to delete fiele
    }
}
```

Figure 54 - Alert View Method

The alert view method above is implemented to check the users response to an alert box that has appeared on the screen. In this application I have used two alert boxes. If a user has entered an invalid number into the bpm text field then an alert is shown. The code above shows that to test whether this is the case, we check the title of the UIAlertView that has appeared. If the alert view has been called due to an invalid number being entered, then the value of 'floatBeatsPerMin' is set as 0.5 and the value in the 'tfBPM' is set to 120.

The second of the alert boxes I have implemented in this application checks if the user intended to delete their file, it is displayed when the user clicks on the 'Delete File' button. If the user selects cancel from this alert box then the file will be kept, however, if the user selects the delete button, then the delete() method will be called to remove the file.

3.3 Drum Parameters changed (volume and tempo)

3.3.1 Volume for drums

Each drum used in the drum generator section has been given its own volume so that a user can chose to customise what drum is played louder etc. As each drum has its own AVAudioPlayer, I have used the volume feature of the AVAudioPlayer to allow the allocation of a volume to the drum. When a user changes either of the volume sliders included in the interface, the corresponding action is called (drum 1 uses actionDrum1Volume etc.). In these methods, a simple call is used to get the current value of the slider; this is divided by 10 and then set as the audio players volume.

```
- (IBAction)actionDrum1Volume:(id)sender {    //set volume for the drum player
    floatDrum1Volume = [sliDrum1Vol value];     //get value from the slider
    floatDrum1Volume = floatDrum1Volume / 10;     // /10 as it needs to be within the range 0 - 1
    player.volume = floatDrum1Volume;        //set player of drum 1 to have specific volume
}

- (IBAction)actionDrum2Volume:(id)sender {    //set volume for the drum player
    floatDrum2Volume = [sliDrum2Vol value];
    floatDrum2Volume = floatDrum2Volume / 10;
    player2.volume = floatDrum2Volume;
}
```

Figure 55 - actionDrum1Volume

3.3.2 BPM

Included in my application, is a feature to allow a user to change the tempo of their created drum track. To change the tempo, a user is required to input a value for the beats per minute of the track , if the value entered is less than 60, greater than 180 or not a number, I have displayed an alert to ask for a more reasonable number. To allow the play method to take account of the BPM, the variable

'floatBeatsPerMin' is used as a holder of the BPM, this is assigned to be 60 (60 seconds in a minute) divided by the value that the user entered into the BPM text box.

```
- (IBAction)actionBPM:(id)sender {  //action to retrieve the value for bpm
    float textOfBox = tfBPM.text.integerValue;
    NSLog(@"%e", textOfBox);
    //if bpm is < 60 and >180 or bpm NAN then display alert asking for better bpm value
    if ((textOfBox < 60) || (textOfBox >180) || (isnan(textOfBox))) {
        NSLog(@"OH NO");
        UIAlertView *alert = [[UIAlertView alloc] initWithTitle:@"Please enter a number in the range of 60 - 180" message:nil delegate:self
            cancelButtonTitle:@"Okay" otherButtonTitles:nil,nil];
        [alert show];
    }else{
        NSLog(@"%e,", textOfBox);
        floatBeatsPerMin = 60 / textOfBox;   //set sleep time for the track as 60 seconds / bpm
    }
}
```

Figure 56 - actionBPM

3.4 Populate the Drum Table

3.4.1 Table View Methods

When a user wants to load or delete a file, they have to select a file from the table on the screen, also when they save a track, the filename is inserted into a table. A NSMutableArray variable 'marrOfFiles' is used to store all of the tracks that have been saved to the iPad, this variable is set up in the viewDidLoad method discussed in section 3.5 below. For the application, I have used a UITableView, there are four mandatory methods when you implement a UITableView. The first two define the number of rows and sections in the table, there is one section in the table and the number of rows contained in the table will be dependant on how many files are saved to the iPad (each file saved will be stored in an array called 'marrOfFiles'). The third method returns the file name of a selected file from a table. This file name is then used in the delete and load methods.

```
//UITableView methods
#pragma mark Table View Data Source Methods
- (NSInteger)tableView:(UITableView*)tableView numberOfRowsInSection:(NSInteger)section{
    return [marrOfFiles count]; //number of rows is equal to the number of items that have been saved
        (in marrOfFiles)
}

- (NSInteger)numberOfSectionsInTableView:(UITableView *)tableView;{
    return 1;   //one section in the table
}

- (void)tableView:(UITableView *)tableView didSelectRowAtIndexPath:(NSIndexPath *)indexPath{    //row
    selected
    strTableSelectedFile = [marrOfFiles objectAtIndex:indexPath.row];  //find what file was selected

    //enable the load and delete buttons when a file has been pressed in the table
    btnLoadFile.enabled = true;
    btnLoadFile.alpha = 1;
    btnDeleteFile.enabled = true;
    btnDeleteFile.alpha = 1;
}

- (UITableViewCell *)tableView:(UITableView *)tableView cellForRowAtIndexPath:(NSIndexPath *)indexPath{
    UITableViewCell *cellTable;
    UILabel *labelCell;
    cellTable = [tableView dequeueReusableCellWithIdentifier:@"Cell"];

    if (cellTable == nil)   //set up the cell in the table
    {
        cellTable = [[UITableViewCell alloc] initWithStyle:UITableViewCellStyleDefault
            reuseIdentifier:@"Cell"];
        labelCell                = [[UILabel alloc] initWithFrame:CGRectZero];
        labelCell.font           = [UIFont fontWithName:@"Arial" size:15];
        labelCell.numberOfLines  = 0;
        labelCell.tag            = 1;
        labelCell.backgroundColor = [UIColor clearColor];

        cellTable.selectionStyle = UITableViewCellSelectionStyleBlue;
        [[cellTable contentView] addSubview:labelCell];
    }

    NSString *strTextCom = [marrOfFiles objectAtIndex:[indexPath row]];
    CGSize constraint = CGSizeMake(184.0, 300.0);
    CGSize size = [strTextCom sizeWithFont:[UIFont fontWithName:@"Arial" size:15] constrainedToSize:
        constraint];

    if (!labelCell){
        labelCell = (UILabel*)[cellTable viewWithTag:1];
    }

    [labelCell setText:strTextCom];
    [labelCell setFrame:CGRectMake(10, 10, 184.0, MAX(size.height, 44.0f))];
    return cellTable;
}
//End required table view methods
```

Figure 57 - Table View Methods

The final method creates the cells for the table, formatting them and placing the correct text into the label on the row.

3.5 - viewDidLoad Method

In the viewDidLoad method, all of the default variables are set or allocated space. Each drum array is filled with all 0's as when the screen loads, no drum is selected at any beat. Two AVAudioPlayers are defined for each drum, the AVAudioPlayer method 'initWithContentsOfURL' is used to find the drum samples in the resource folder of the application. The initial volume of each drum is also set and the 'prepareToPlay' method is called so that the players are ready to play the drum tracks when required. The final thing that happens in this method is to populate the array of files stored by the application. To do this, the path to the files is used alongside a file manager, the contents at the given path is then loaded into the 'marrOfFiles' variable.

```
#pragma mark - View lifecycle
- (void)viewDidLoad{

    [super viewDidLoad];
    boolInfiniteLoop = true;
    floatBeatsPerMin = 1; //initialise bpm

    //Initialise the drum arrays
    marrDrum1 = [[NSMutableArray alloc]init];
    marrDrum2 = [[NSMutableArray alloc]init];

    for(int i = 0; i<16; i++){
        [marrDrum1 addObject:@"0"];
        [marrDrum2 addObject:@"0"];
    }

    //UICollectionView views
    [self.cvDrumGrid registerClass:[drumCell class] forCellWithReuseIdentifier:@"Cell"];

    // Set up avaudioplayers for the drums
    NSString *soundFilePath = [[NSBundle mainBundle] pathForResource:@"drum1" ofType:@"wav"];
    NSURL *soundFileURL = [NSURL fileURLWithPath:soundFilePath];
    player = [[AVAudioPlayer alloc] initWithContentsOfURL:soundFileURL error:nil];
    player.numberOfLoops = 0;
    floatDrum1Volume = 0.5;
    player.volume = 0.5;
    [player prepareToPlay];

    NSString *soundFilePath2 = [[NSBundle mainBundle] pathForResource:@"drum2" ofType:@"wav"];
    NSURL *soundFileURL2 = [NSURL fileURLWithPath:soundFilePath2];
    player2 = [[AVAudioPlayer alloc] initWithContentsOfURL:soundFileURL2 error:nil];
    player2.numberOfLoops = 0;
    floatDrum2Volume = 0.5;
    player2.volume = 0.5;
    [player2 prepareToPlay];

    //Find files in the directory folder of the app and put these contents in arrayoffiles
    NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory, NSUserDomainMask, YES);
    NSFileManager *filemgr = [NSFileManager defaultManager];
    marrOfFiles = [filemgr contentsOfDirectoryAtPath:[paths objectAtIndex:0] error:nil];

    //Display the saved files onto the terminal
    int count = [marrOfFiles count];
    for (int i = 0; i < count; i++){
        NSLog (@"%@", [marrOfFiles objectAtIndex: i]);
    }
}
```

Figure 58 - View Life Cycle

Results and Evaluation

In my opinion, the development of my application has been a success as I have completed all the aims that I had set myself in the initial plan, however, I believe that the overall application is only partially completed as there are so many other features that could be added to the application.

I have split the results and evaluation of my application into five separate sections:

- Unique features of my application
- Advantages/Disadvantages of my application
- Evaluation against the aims and objectives
- Evaluation of the key methods
- External Evaluation

Unique Features

My developed application has a few unique features that other similar application do not have. Most of these features come from the combining the MIDI input aspect and drum track creator into one application. Here is a list of the unique features:

- The ability to change the tempo of the drum track - Not many of the applications that I had researched had this feature, most had a fixed tempo which I thought limited the customisability of the drum track.
- The ability to play the drum track when the user is playing notes from a keyboard.
- The ability to save the drum track - Few applications that I researched had this ability. However, I didn't research the paid for application and therefore this may be a feature included in many of these applications.
- The ability to change an effect by swiping the screen is fairly unique. Most similar applications use sliders and knobs to control the parameters

Advantages/Disadvantages of my Application

Advantages

The main advantages of my application are:

- Simplistic interface - touch to control most of the parameters and/or effects
- Ability to change the tempo of the drum track created
- Ability to save the drum tracks and play them over MIDI input
- Ability to modulate input using a swipe and touch gesture

Disadvantages

The main disadvantages of my application are:

- The fact that the user needs a keyboard and the camera connection kit to use the whole functionality of the application. Other applications get around this problem by implementing an on-screen keyboard, therefore this could be a solution to this specific problem.
- Not a very interesting interface.

Aims and Objectives

Here are the aims and objectives that I set out in my initial plan. After each aim/objective, I have discussed whether I have completed them, if I haven't, then I give a reason as to why I didn't implement it.

- *Create a tactile music application for the iPad*
  - *Music creator*
  - *Backing Drums and MIDI Accompany*
  - *Real-time audio controllable effects*
  - *MIDI control for the iPad*

The main aim of the project was to create a tactile music application for the iPad and I believe have done this. I have implemented the ability to create drum tracks and play these while playing music through a MIDI enabled keyboard. While playing the keyboard, a user can also change the effects applied to the music in real-time.

Detailed Aims and Objectives

- Implement the ability to input MIDI events to the iPad via a keyboard
  - Change the volume using the iPad
  - Add different effects to the MIDI input
  - Record input and play it back

My iPad application, allows a user to connect a MIDI enabled keyboard and play musical data using this keyboard via a connector. Effects can be placed upon the notes in real-time, such effects include: diminish, up arpeggio and random arpeggio. I have implemented a record function that works with most effects. I decided not to implement a way to change the volume as I later implemented a more advanced volume envelope(the volume envelope parameters do not get saved, but the user can then modify the recorded piece using the volume envelope parameters).

- Implement the drum/beat generator
  - Create a drum generator based on an x,y grid
  - Each point x,y would have some option to allow the drum to be played at that point if required
  - Implement a way to have different volumes for each different drum
  - Create a drum track with either 4 or 8 beats with a time signature of 4/4
  - Save the drum beat
  - Play the saved drum beat while recording/inputting MIDI via a keyboard

A user can create a drum track with my application using two drum samples (more can easily be added at a later stage of development). Each drum has its own separate volume. Instead of asking the user to chose between having 4 and 8 beats, I decided to implement 16 beats as I thought it would give the user more choice. I have also decided to implement the ability to allow the user to slow down or quicken the tempo of the track, but it still has a 4 /4 time signature. The drum track can be saved and stored on the iPad, these tracks can then be loaded or deleted by using the table

provided. A user can listen to their drum tracks in the drum section, but after saving their track, they can then play the drum track in the MIDI section whilst also playing music via their connected keyboard.

- Add effects to the MIDI input using different gesture controls e.g. two finger swipe, three finger press

I have implemented the ability to allow the user to modulate the sounds they create using a separate view which they can touch or swipe to control the parameters of the modulating signal. They can also modulate the signal using two sliders. (The modulation parameters do not get saved, a user can however modify the recorded piece using this effect)

- Implement additional features, such as some from the devices I researched during the completion of my interim report.

Additional features that I have included in the application are: a volume envelope, a chord effect, the changing of the track tempo and the ability to chose from three sound types (cosine , sawtooth and square waves).

Evaluate Strategy for Key Methods

MIDI section

MIDI input

Check whether the application allows for MIDI input.

Ways to Test:

- Connect keyboard to the application
- Select 'Connect Keyboard'
- Play a few notes

If a user can hear the notes that they are playing then I would deem this method to be functioning.

Effects

Check whether the user can add different effects to the MIDI input.

Ways to Test:

- Play notes without effect
- Select effect
- Play notes with effect
- Re-test for each effect

If a user can hear that the effects have been added, then I would deem this method to be functioning. The user should also notice some of the effects being disabled when certain effects are applied as not all effects can be used together.

Record

Check whether a user can record a series of notes into the application and be able to play back the recording.

Ways to Test:

- Select the record button and play a series of notes. Then playback the recording to see whether the notes have been recorded in the correct sequence.
- Select the record button and play a series of notes. Change what effects are used while recording. When playing back the recording, listen to whether the effects have been added and changed at the correct time.

If a user can hear that their piece has been recorded and played back correctly (with the effects added) I deem the record and playback methods to be working correctly.

Modulation

Check whether the user can apply the modulating signal to their input.

Ways to Test:

- Play notes using the connected keyboard.
- Change the slider values or tap or swipe the view to change the modulating parameters.
- Listen for the change in the audio sound

If a user can apply the modulation and hear the difference between the original and modulated wave form I deem this method to be working correctly.

Drum section

Set Drums

Check whether a user can change the status of the drums at a specific note.

Ways to Test:

- Select a drum at a specific beat - the view under the beat should change to red
- Play the drum track to see if the drum plays at the specified beat.

If the drum plays on the correct beat, I deem this method to be functioning.

Save/Load/Delete

Check whether a user can save a track, load a track and then delete the track.

Ways to Test:

- Select a pattern for the drums by changing the drums status at different beats
- Type a name for the track in the text box on screen.
- Save the file.
- Re-set the drum track so that all drums are off.
- Load the file.
- Delete the file.

For this method to be deemed successful, a few things have to be tested. Once a user has saved their track, the name that they specified should be placed in the table of tracks on the screen.

To test whether the track was saved, the interface needs to be reset to its original state and a saved track needs to be loaded from the table by using the 'load' button. If the correct pattern of drums has been loaded, then the save and load methods are functioning.

The delete method is tested by deleting the file, if an alert pops up asking if the user wanted to delete the file and they chose yes, then the file should be removed from the track table. If the three methods work, I deem this part of the application to be a success.

External Evaluation

To determine how well the application functions and how useable it is, I have asked another student to conduct a test of my application. Unfortunately, due to time constraints, only one student was able to test my application for me and provide feedback. In general I would've like to have at least ten people test it and give me their views on it but it wasn't possible. The testing method that I have used, is a mixture between a questionnaire and observation. I asked the student to carry out the testing of the application following the questionnaire and I observed how efficiently he managed to complete the tasks. A complete copy of the questionnaire results is attached in the appendix.

When designing the questionnaire, I wanted to ensure that all aspects of the application were tested by the user. Therefore, instead of giving them the application and asking for their thoughts on it, I decided to outline a number of tasks that I would want the user to complete. This way, I made sure the user tested all of the functionality provided by the application. After each task in the questionnaire, I asked him for further comments on the usability of the application for that specific task. I have also asked the tester whether he was happy with the functionality of the device or not, I provided clear questions asking what they like most and if they would change anything.

While the tester used my application, I sat in the room and observed how he interacted with the device to gauge whether other people understood the process that is required to use the application, for example, you need to connect the keyboard through a button because it doesn't find it automatically.

Results

Feedback of the drum section from the participant was of a positive nature. He liked the way in which the drum patterns are selected and generally thought the drum section was easy to use with every task being classified as easy apart from testing the drum. Unfortunately, he had a bit of a problem with testing the drums as he thought it was unclear what test actually meant and it took him awhile to realise that the two drum buttons actually played the two drums. These may need to be reworked in future to make the user aware that they can actually test the drums.

As with the drums, he generally thought every task asked of him in the MIDI section was easy, again he struggled on one task and that was to connect the keyboard to the application. He thought it would be better if the application could automatically detect a connected keyboard which I agree with. He didn't like the arpeggio effects as he thought there wasn't much point to them, this could be rectified by creating a more advanced arpeggiation technique. The overall feedback I received from the participant was positive on this section, the final comments he made were that he liked the fact that buttons are disabled if you can't use them and he was really impressed with the modulation feature.

Functionality

The participant of the testing process did think the functionality of the application was good overall, he particularly enjoyed the touch screen modulation process. He would however, like to be able to add more effects to his input in the MIDI section (e.g. a wah-wah effect) and he also commented that he would perhaps like to be able to add effects to the drum track created. He didn't really see the point in the random arpeggio and he said that it sounded pretty strange. Finally, he would like the ability to not only save the drum track, but to also save the recorded MIDI input which, if

implemented, could give the application an edge as not many other applications that I have researched allows this.

General

The participant thought the application was easy to use overall and enjoyed the experience. He liked the fact that the interface was easy to use but he did have some reservations on the design of it, he thought that the design could be improved graphically which I do agree with. Finally, he didn't report any problems with his experience.

Critical Evaluation of the Application

Design

During the implementation of the project, I intended to keep the design of the application as simplistic and easy to use as possibly to help the users learn the application quicker. I believe that the interface that I have designed is simplistic and easy to use, however, as I focused more on the functionality, the interface isn't very graphically pleasing. A lot of improvement could be done to improve the look of the application but hopefully keeping it simple enough for a novice user.

Usability

As stated, I have tried to implement the application in such a way, that a novice user could use it and still be able to use all of the functionality of the application. Mostly, I believe this is the case, there is a simple way to navigate between the two sections, and most of the interface is self-explanatory (e.g. chord button adds the chord effect). However, some things do need to change to make the application more usable, for example, it would be better if the application could detect a keyboard automatically and some more helpful labels could be added. Finally, as I have made use of disabling certain objects from the screen, I believe this does enhance the usability of the application, due to the user not being able to use certain objects and therefore the application won't fail due to user error.

Functionality

As I will discuss later in the future work section, I believe that the functionality of this application is only half implemented as there are a lot more effects which could be added to the application. I believe most of the effects that I have implemented are good, however, I would consider removing the random arpeggio and diminish effect in future as they aren't very useful. As already discussed, I would add more drums to the drum section to give the user more of a choice when it came to creating their tracks.

Future Work

Problems Encountered

The main problem that I encountered whilst developing my application was the fact that I could only program the application on my Macintosh computer. This meant that when I was away from this computer I could not work on the project, e.g. using my laptop or other computers (as they are all Windows) as they did not have the required software (Xcode) needed for iPhone/iPad development. This caused significant problems during the implementation phase of the project as I was unable to spend as much time as I would've liked to on developing the application.

Another problem that hindered my progress in developing the application was the fact that I used Pure Data as the means of creating / processing the audio signals. As I did not have any prior experience using Pure Data, I had to learn the language as I was developing the application. At first I managed to use the language very easily and thought that I had created a good synthesiser for the application, but after finishing the implementation of the drum section of the application, I went back and realised that I had barely scratched the surface with regards to my knowledge on Pure Data. It can be a very complex application and there is not much direction with regards to learning how to program well in it. However, there is a strong community of users that post regularly in the forums if you need some guidance. Pure Data is also split into two versions, there is a vanilla version of Pure Data that means there are less objects that you can use and there is an extended version of Pure Data with more objects available. Unfortunately, I use libpd to enable me to use Pure Data on the iPad and this contains the vanilla version of Pure Data. One thing that I didn't realise was that my computer holds the extended version of Pure Data, which meant that I coded some good features such as a guitar effect or flute effect and these worked when I tested them on the computer. However, once I put them onto the iPad they didn't work due to the fact that these patches made use of the extended objects in Pure Data. I didn't realise that this was the case until I had spent a lot of effort trying to figure out why they worked on the computer but didn't work on the iPad.

Finding books about the Pure Data and Core MIDI technologies was difficult to start with, I checked a number of different libraries in Cardiff and none of them really had any relevant books that I could learn from. My supervisor did order a book to the university library for me, but that took awhile to get here. This book was on Core Audio and it did contain useful information about Core MIDI which I ended up using in my application. With regards to Pure Data, I eventually bought the 'Making Musical Apps'[3] book which contained good information on how to use Pure Data in an iPad application but it still lacked information on the more complex ideas and I have yet to find a good book explaining Pure Data.

A minor problem that I had, was the fact that I had updated my iPad to the newest version of iOS without realising, that by doing this, it meant that my iPad was no longer compatible with Xcode. To solve this, I had to update Xcode to the newest version available which luckily did support the new version of the iOS. This wasn't a major problem, but it has made me think about refraining from updating my iPad whilst using it for development in the future, just in case.

General Improvements

I have achieved everything that I set out to implement in the initial plan, however, I believe that there is still scope for improving the project, as there is with any iPad application that is brought out, with general updates brought out to improve the application or to fix certain bugs found. The main way that the application could be improved, would be to implement some of the features that I drifted away from during the early stages of my research. The original idea behind my application, was to emulate some of the features found on the 'KORG Chaos Pad' but as said, I moved away from this slightly, but I believe that my application sets a good foundation for the implementation of the more complex ideas found on the 'KORG Chaos Pad'.

As I focused more on the functionality of the application, the interface design isn't as good as it could be and therefore I believe this could be improved. At the end of my implementation, the interface of the application is very basic using the default views, buttons, sliders, textboxes and labels provided by the Xcode development team. These objects have been customised slightly e.g. by using different colours, and with regards to the UICollectionView I have customised what appears on this screen(drum selection), but I believe that the interface could be improved with the further customisation of the on screen objects to enhance the look and feel of the application.

They key limitation of the application is the fact that it isn't very portable, it can't be used on other operating systems such as Android or Windows 8, therefore, limiting the potential market for the application. Future work could be focused on the development of an Android or Windows 8 version of my iPad application to make use of the fact that there are a lot of tablets using the android / Windows 8 operating system.

One of the key limitations of my application is the need for the camera connection kit to connect a MIDI enabled keyboard to the iPad device. These camera connection kits usually cost £25 and it may put some people off using the application. To improve the usability of the application, future work could be focused on different ways to input musical data to the application. A possible way of doing this would be to use wireless MIDI to input the MIDI signals instead of using a MIDI keyboard (this would get rid of the need for the camera connection kit and would appeal to a wider range of users), or to use audio tracks that are already on the iPad. This data could then be used and manipulated by the users within the application (similar to the original idea of the project).

Midi Section Improvements

Most of the future work of this application would be to improve the MIDI section as this is the main feature of my application. As discussed above, the first thing to improve would be to include some adapted versions of the features in not only the KORG Chaos Pad but also the Yamaha Tenori and other similar applications.

There are two similar effects in my application, these are the diminish effect and the volume envelope (these are both discussed in the implementation section above). I believe that both of these effects could be combined into one due to how similar they are, the only difference is that when a user releases a key, the diminish effect keeps playing until the sound fades out and while

using the envelope the sound cuts out immediately, however, if you hold the key long enough the sound will fade out if you set the correct parameters.

The Final improvements that could be made to the MIDI section would be to include more effects that could be utilised by a user to give them a wider palette of effects to chose from. Some features that could be added are:

- Adding a wah-wah effect to the audio
- Improving the arpeggio effect by splitting the effect in two
    - Keep the ability for users to play an arpeggio.
    - Improve on the arpeggio synthesiser so that it acts like an arpeggiator. Possible inclusions could be the choosing of notes to play as an arpeggio and the ability to keep playing the arpeggio after releasing the note.
- Allowing the user to adjust the speed of a recording
- Allowing the user to use all of the effects while recording (merge the three Pure Data synthesisers into one)
- Adding an echo effect to the audio
- Adding a reverb effect to the audio
- Adding different instruments (this would more than likely require the ability to use the extended objects of Pure Data , currently I do not think this is available)

Drum Section Improvements

Overall, I do not believe that a lot of improvements need to be made to the drum section as I am pleased with how I have implemented it. There are however, two improvements that I would suggest.

Currently, there are only two drum samples that can be used when creating your drum tracks. Obviously this means that it will be very limited with regards to the patterns that can be created using these two drums and therefore, I would like to add more drums to the drum section in the future. I have already set out the space required for a number of other drums to be added into the application, it will just be a matter of finding good drum samples to include as the code required to use these drums will be exactly the same as the code used for the two drums that are currently being used within the application. It may also be a good idea to conduct research into allowing a user to input some of their own drum samples to the application to provide a very customisable experience.

The final improvement that I would like to make to the drum section, would be to improve the interface that is utilised by the users to pick which drum is played at which beat. With regards to all the other elements of the interface, this is also pretty basic and as such I believe it could be improved to make it have a more sleek design.

Pure Data Synthesiser Patch Improvements

The main improvement that could be made to the synthesiser patch would be to combine the three different patches together to create one main Pure Data patch that contained everything. This can be done, however, during my implementation I didn't have enough time to combine the three into one as it would've been time consuming to ensure that every effect has been ported over into the one synthesiser correctly.

Conclusions

Upon completing the implementation phase of my application and evaluating it, I believe that my application meets the needs of the aims and objectives set out in my initial plan. However, I believe that my project has taken a slightly different approach to solving the original problem. The project was initially meant to bean emulation of the Korg KAOSS pad, some features on my application are 'Korg like' but some aren't. I do believe that with my implementation, that there is scope to further improve it by not only adding some of the advanced features found on the Korg Kaoss pad, but some features that are included on other similar applications.

Overall, I am satisfied with how my project has been implemented and with the amount of work that I have done. I believe that I have implemented a good application that could be further improved upon in future. Even though I have completed all of my aims set out in the initial plan, I feel that the application is only half completed and could easily be carried on in future to improve upon it. With regards to the audio effects, I am satisfied with the effects that I have included and I am very fond of the way in which the modulation parameters are changed (using the UIView whilst tapping or swiping).

My methods of development were quite efficient, mostly I worked well and on schedule, however, I did struggle with Pure Data. After finishing the basic implementation of the MIDI section, I felt as if I had nearly finished with the Pure Data patch, however, after completing the drum section I realised that I had only just touched the surface in what Pure Data could actually do. After this realisation, I spent most of my time trying to understand what Pure Data was able to do. I did manage to finish the implementation but I have three Pure Data synthesisers within my Pure Data patch, which I would ideally like to combine sometime in the future. With regards to the time spent on developing the application, I believe that I managed my time efficiently, I set myself deadlines and I managed to stick to these and finish the implementation on time.

Next time, I would do a few things differently. The main thing I would do, would be to research the technologies I need in more detail and make sure I understand exactly what they can or can't do. This would save me time in the long run compared to learning the technologies whilst implementing the application like I tried to do with Pure Data. Perhaps, I would also make more detailed notes as I was implementing the application so that I could refer back to them at a later stage. I wrote a few things down this time, but not enough to understand exactly why I did things in a specific way.

Finally, as I have a background in music, I have enjoyed creating this application, mainly as I was able create different sounding effects from my input and I was able to create some interesting sounding drum tracks.

Reflection on Learning

During the course of this project, the main skill that I have improved upon is my programming skills, as my project was mostly a programming project. I had a basic understanding of Objective-C before starting the project as I had developed a few small test applications in the previous summer. My skills in Objective-C have mainly been improved by developing the application and seeking help from materials such as online forums, books and tutorials when I wasn't able to implement what I wanted to do. After programming in Objective-C for two semesters I believe that my skills using this programming language have improved significantly to the point where I could now create an application from nothing without the need of looking through different sources for help.

Along with Objective-C, I have used Pure Data during the course of this project. I had never used or heard of Pure Data before I began researching into the technologies I could use to develop my application. Unlike Objective-C, I did find using Pure Data hard, this was due to the limited amount of helpful information about Pure Data online or in books. However there is a committed developer community behind Pure Data that are very helpful, this is where I have learned most of my Pure Data knowledge by going through tutorials and explanations on the forum provided by other developers.

Along with my programming skills, my time management skills have improved due to the fact that I have not been working with other people and therefore it has been up to me to get the work done by the deadlines. I mostly managed to keep to the schedule created in the initial plan, sometimes I was a week behind but at other times, I was a week ahead of the schedule and therefore I have been able to complete all of my work by the deadlines that I had set myself.

A skill that links in to time management is organisational skills. In the limitation and future work section, I discussed the fact that I could only develop the application on one computer and therefore, I had to organise when I could work on my application as for example, when I needed to be away from my computer I couldn't do any work. Therefore I decided to set some times every week where I could work on the project. If I needed to be away, I then caught up by setting more time aside the week after to develop the application. My supervisor also helped improve my time management skills as we had weekly meetings to discuss the progress of the project, therefore I tried to complete a different task each week so I could get his opinion on it.

To keep backups of my work or different versions, I used the 'Time Machine' feature of the Macintosh computers, which stored copies of my work at specific intervals and keeps them until the hard drive is full before it starts overwriting the old version (my hard drive isn't full and therefore I didn't need to worry about older versions being overwritten). This feature did help me, as I found myself needing to look back at previous versions of my implementation when I had gone completely wrong.

During the course of the project, I have learned more about audio processing due to the nature of the project. I had little knowledge before but after creating the application and especially the Pure Data patches I understand the process a bit better. This knowledge will be useful as I have taken the

Multimedia module which discusses a bit of audio processing, hopefully when revising for the exam, my extra knowledge will provide useful.

A skill set that I never thought I had were research skills, which I used during the initial stages of my project to gather ideas of what could or couldn't be included in the application. These skills were also used when encountering problems during the implementation phase of the project.

Finally, I believe by completing this project, I have gained more knowledge in the process of completing a project e.g. getting work done by deadlines. Also I have improved my report writing skills due to the three deliverables that have been submitted as part of this project.

Overall, I feel this project has been of benefit to me personally, due to the number of skills that I have improved upon and the new skills that I have learned. These skills will hopefully be of use to me when I leave the university and start a career.

Glossary

Term	Meaning
Tactile	Perceptible to the sense of touch
MIDI	Musical Instrument Digital Interface
XCode	Apple's development environment for Objective-C
Objective C	Object-oriented programming language used to create ipad/iPhone/iPod applications
iOS	Apple's mobile operating system
Patches	A Pure Data program
Tempo	Speed of a piece of music
Arpeggio	Musical technique of playing a sequence of notes in a chord
Synthesiser	An electronic instrument capable of producing sounds
Modulation	The varying of one waveform or signal using another.
Vanilla-PD	Basic implementation of Pure Data.
Extended-PD	Contains more useable objects than the Vanilla-PD
GitHub	Web-based hosting service used by developers to manage their projects
Boolean	A data type with two values, true or false
Debug	Testing the system to reduce the amount of 'bugs' found in it
Voice Stealing	Voice stealing is a function of the poly object in Pure Data. Voice stealing occurs when all the voices are being used and a new sound is played. The first sound to be played would be stopped and the channel given to the new sound
Arpeggiation	The playing of an arpeggio by aninstrument.
libpd	Allows the running of Pure Data on an iPad application

Table of Abbreviations

Term	Meaning
MIDI	Musical Instrument Digital Interface
PD	Pure Data
Wi-Fi	Wireless Fidelity
bpm	Beats per minute

Appendix

S R Walters Evaluation of Project 152

Evaluation Questionnaire - Final Year Project

Project Number: 152

Dear Student,

I would like to ask you to participate in a quick questionnaire to aid me in the evaluation of my final year project. The project that I have created is an iPad application, which allows a user to play musical notes into the application via a keyboard and also allows a user to create their own drum tracks. Different effects can then be added to the musical input and the drum track can be played on top of it.

The following questionnaire should take you approximately 15 minutes to complete. If you chose to take part in this process but would like your participation to remain confidential please DO NOT include your name at the top of the next page. Any information provided by yourself will be used in my final year project to evaluate the application, the information provided will not be used for any other purposes. Please answer all questions honestly and if you would like to omit some questions please feel free to do so.

Thanks in advance for taking the time to read this and I hope you will take part in the questionnaire.

S. R. Walters

S R Walters Evaluation of Project 152

Evaluation of Project 152

Name: Matthew Jones Date of Testing: 12th Apr 13

Drum Track Generator

#	Task	Tick / Cross	Ease of use (1 - difficult, 2 - moderate, 3 -simple)
1	Navigate to the drum section	✓	3
	Any Comments: Would be nice to have an icon, on navigation		
2	Can you test each drum?	✓	2
	Any Comments: Unclear as to what test meant		
3	Can you select a pattern for your drum track?	✓	3
	Any Comments:		
4	Can you play your drum track? Can you stop the drum track?	✓	3
	Any Comments:		
5	Can you change the tempo (bpm) of the track	✓	3
	Any Comments: Good error message if tempo out of range. " real time - change of tempo		
6	Can you reset the grid interface	✓	3
	Any Comments: Clear Button to reset interface		
7	Can you save a drum track	✓	3
	Any Comments:		
8	Can you load a drum track	✓	3
	Any Comments:		
9	Can you delete a drum track	✓	3
	Any Comments:		

S R Walters — Evaluation of Project 152

Additional Comments:

Easy way to provide choice of drum pattern.
Nice to be able to save drum pattern.

MIDI Section

#	Task	Tick / Cross	Ease of use (1 - difficult, 2 - moderate, 3 -simple)
1	Are you able to connect a keyboard?	✓	2
	Any Comments: Perhaps better if keyboard was connected automatically		
2	Are you able to select a drum track to be played? Are you able to stop this drum track?	✓	3
	Any Comments:		
3	Are you able to play notes through the keyboard?	✓	3
	Any Comments:		
4	Can you play different sound types? (normal, sawtooth, square)	✓	3
	Any Comments: Easy to see buttons		
5	Are you able to use the up arpeggio effect?	✓	2
	Any Comments: Message to say hold key down needed		
6	Are you able to use the random arpeggio effect	✓	3
	Any Comments: Nice effect, is there any point to it-		
7	Are you able to use the envelope parameters?	✓	3
	Any Comments: Make it clear that it is time, incl. volume level.		

S R Walters — Evaluation of Project 152

8	Are you able to apply the chord effect? Any Comments:	✓	3
9	Can you change the modulation parameters using the sliders? (tremelo value/range) Any Comments: Nice to see dot move around on screen	✓	3
10	Can you change the modulation parameters using the view provided? Any Comments:	✓	3
11	Can you record your input? Can you playback your recording? Any Comments:	✓	3
12	Are the correct effects applied to the recording? Any Comments:	✓	3

Additional Comments:
Like the fact options are greyed out if they can not be used.
Tremelo sliders change if you move dot on grid

S R Walters

Evaluation of Project 152

Questions on the Functionality of the Application

Please provide extra comments to explain your answers.

MIDI Section

Are you happy with the functionality of the MIDI section?

Yes, good use of different sound types
Nice modulation touch pad

Would feature do you like most? Please Explain

The modulation touch pad because you can alter your recorded MIDI whilst it is playing
Neat way of modulating the sound

Would you like more effects to be added to the MIDI section?

Yes

If the answer to the question above was yes: What additional effects would you like in the application?

Perhaps. Nice to use the touch pad idea to add more effects.
Would like to see something like a Wah-Wah ~~Wout~~

Would you remove any of the effects?

Think about removing the random arpegio effect, I don't see much point in it being there

Additional Comments:

I would like to be able to save my audio track and any effects I had added to it

S R Walters

Evaluation of Project 152

Drum Section

Are you happy with the functionality of the drum section?

Yes, It is good at generating dum noises.

Would feature do you like most? Please Explain

The grid to select when you want your drum to be played.

Would you change anything within the drum section?

Yes

If the answer to the question above was yes: What would you change?

Add a greate range of drums

Look at different effects you could add to the drums

Would you want to add other instruments to the drum generator? Please explain.

Maybe some low range instruments to give a repeating bass line

Additional Comments:

Nice that you can save the drums and use them in the MIDI section.

S R Walters Evaluation of Project 152

General Questions

Please use the rating of 1 = extremely good to 5 = extremely bad to answer the final questions (when required), and please feel free to provide extra comments.

How easy is it to use the application? 2

Simple to use and create effects for music.
Easy to create drum beats

How intuitive is the user interface? 2

Interface is easy to use
Some adjustments needed for design

Any problems with your experience?

N/A

References

[1] - GitHub. libpd / pd-for-ios. [Online]. Available at: https://github.com/libpd/pd-for-ios [Accessed: 18 October 2012]

[2] - Adamson, C and Avila K. 2012. Learning Core Audio: A Hands-On Guide to Audio Programming for Mac and iOS. New Jersey: Pearson Education, Inc.

[3] - Brinkmann, P. 2012. Making Musical Apps. Sebastopol, California: O'Reilly

[4] - Adoption Cureve Dot Net. A Simple UICollectionView Tutorial. [Online] Available at: http://www.adoptioncurve.net/archives/2012/09/a-simple-uicollectionview-tutorial/ [Accessed:12 Feburary 2013]

- Walters, S.R. 2012. Interim Report. Cardiff
- Walters, S.R. 2012. Initial Plan. Cardiff