Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Hálózati Rendszerek és Szolgáltatások Tanszék

Csizmadia Dénes

GITÁR-MULTIEFFEKT IOS PLATFORMRA

KONZULENS

Dr. Márki Ferenc

BUDAPEST, 2015

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Csizmadia Dénes**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2015. 12. 10.

...…….

Csizmadia Dénes

Összefoglaló

A technika fejlődésével a zeneipar is lépést tart. Első lépésben az analóg effekteket váltották fel a digitálisak, majd elérkezett az a korszak is, amikor már mindent az okos hordozható eszközökön, például a telefonokon akarunk használni. Ezért felvetődhet a kérdés, hogy ki lehet-e váltani a digitális effekteket is egy ezekre írt alkalmazással. A válasz igen, hiszen ma már a gitár hangját egy iPad eszközbe beépített processzorral is fel lehet dolgozni, és a nagyméretű érintőképernyőjén létrehozható egy egyszerűen használható kezelőfelület is.

Szakdolgozatom keretében el is készítettem egy ilyen multieffekt alkalmazást gitárosoknak, az Apple által fejlesztett iOS operációs rendszert futtató iPad készülékre. Ehhez két fontos feladattal kellett szembenéznem. Elsőként el kellett sajátítanom az iOS fejlesztés alapjait, majd ezt követően az audio jelfeldolgozással ismerkedtem meg. Dolgozatom ennek megfelelően több részre tagolódik: az említettek elméleti összefoglalásával kezdődik, majd ezután térek rá az általam megvalósított program ismertetésére. A dolgozatom végén kitérek arra is, hogy milyen formában juthat el az alkalmazásom a felhasználókig, illetve hogy milyen további feladatok állnak még előttem ezzel kapcsolatban.

Úgy gondolom, az alkalmazásom mostani első verziója jó kiindulópont lehet egy zenészeknek szóló, iOS paltformon futó multieffekt elkészítéséhez, melyet az App Store alkalmazásboltból lehetne elérni a későbbiekben.

Abstract

There is no doubt that the music industry keeps up with the technology. At the early stage of the IT revolution, analogue effects were replaced by the digitals, and after that a new era has arrived. Today we are surrounded by smart systems and we use our smartphones to handle almost everything. The question is whether we could utilize the digital effects in a smart device application too. We already know that the answer is yes. With the help of an iPad processor we are able to generate the sound of a guitar. We can also create the appropriate user interface on a large touch screen.

As a result of my research, I managed to develop a multieffect application for guitarists. It runs on iOS operation system, which is the system of iPad devices, for instance. It was a great challenge for me, since I had to learn from the very beginning how to develop for iOS and how to process audio signals. A brief summary of the theoretical background can be found at the beginning of my thesis. Afterwards I have described how I could create my application. At the end of my thesis I gave a hint about my future plans, including the way of promoting my application. Moreover, I have tried to summarize the tasks which I will have to fulfil in the near future.

I believe that the first edition of my application could be a great base of a multieffect application, which could be downloaded from the App Store. Of course, it would be also an application for musicians.

1 Bevezetés

1.1 A feladat aktualitása

A technika fejlődésével a zeneipar is lépést tart. Első lépésben az analóg effekteket váltották fel a digitálisak, majd elérkezett az a korszak, amelyben mindent az okos hordozható eszközökön, például a telefonokon akarunk használni. Ezért felvetődhet a kérdés, hogy ki lehet-e váltani a digitális effekteket is egy ezekre írt alkalmazással. A válasz igen. El is készítettem egy ilyet az Apple által fejlesztett iOS operációs rendszerre, ami főleg az iPhone-ok és iPadek operációs rendszere. A dolgozatom hátralevő részében erről számolok be részletesen.

Nem véletlen, hogy ezt a platformot választottam, hiszen az Apple már évtizedek óta a zenészek kedvében jár fejlesztéseivel. Ezáltal egy kölcsönös bizalom alakult ki, hiszen a legtöbb stúdióban már Apple számítógépeket és szoftvereket használnak. Igaz, hogy ezeknek az eszközöknek általában magas áruk van, de nagyon megbízhatóak is. Ez nem csak az Apple gépeken futó OSX rendszerről, hanem az iOS-ről is elmondható. Az iOS rendszeren az alkalmazások nem virtuális környezetben futnak, így egy hardver közelibb programozással a hangfeldolgozást minimális késleltetéssel tudtam megvalósítani. Emiatt is előnyös volt számomra ez a platform.

1.2 Motivációk és köszönetnyilvánítás

Kisiskolás korom óta csellózom és emellett szeretek még gitározni és basszus gitározni is. Így elmondható, hogy a zene már gyermekkoromtól kezdve az életem része. Mindig is szerettem volna hangtechnikával foglalkozni, ez motivált már a villamosmérnök képzésre jelentkezésemkor is. Így érthető, hogy a szakdolgozatom témája is ehhez kapcsolódik.

Konzulensemmel a hangtechnikai gyakorlat című választható tárgy keretein belül találkoztam először. Ő és a tárgya is nagy hatással volt rám, ezért is döntöttem úgy, hogy őt kérem fel konzulensnek. Rendhagyó módon itt szeretnék neki köszönetet mondani, amiért végig támogatott elképzelésemben, és bármikor fordulhattam hozzá kérdéseimmel.

Nagy segítség volt számomra, hogy részt vehettem az AUT tanszék által tartott iOS alapú szoftverfejlesztés kurzuson, mivel ez egy teljesen új terület volt nekem. Emellett a győri mobil szoftverfejlesztéssel foglalkozó Attrecto Zrt.-nél töltött gyakorlatom során is sokat fejlődhettem. Köszönöm Dr. Kelényi Imrének, a tárgy előadójának, és a cég vezetőinek és dolgozóinak, hogy segítségemre voltak az iOS fejlesztés elsajátításában.

Végül, de nem utolsó sorban szeretném megköszönni családomnak, barátnőmnek és barátaimnak is, hogy az egyetemi éveim során támogattak.

1.3 Szakdolgozat felépítése

Szakdolgozatom elkészítéséhez két fontos feladattal kellett szembenéznem. Elsőként el kellett sajátítanom az iOS fejlesztés alapjait, majd ezt követően az audio jelfeldolgozással ismerkedtem meg. Dolgozatom ennek megfelelően tagolódik: ezeknek egy elméleti összefoglalásával kezdődik, majd utána térek rá az általam megvalósított program ismertetésére. A dolgozatom végén kitérek arra is, hogy milyen formában juthat el az alkalmazásom a felhasználókig, illetve hogy ehhez milyen további feladatok állnak még előttem.

2 Az iOS szoftverfejlesztés

2.1 Az Apple sikere

Az iOS operációs rendszert és a hozzátartozó eszközöket az Apple fejleszti, akit a világ legsikeresebb vállalatai között tartanak számon. A sokak által példaképnek tartott Steve Jobs 1976-ban alapította meg garázsvállalkozását, amely aztán globális méretűvé nőtte ki magát, köszönhetően az alapító üzleti érzékének. Apple egy újfajta szemléletet vitt a termékfejlesztésbe.

Ábra 2-1 : A siker titka [1]

Fontosnak tartja azt, hogy mindent amit készít, maguk a fejlesztők is akarják maguknak. Ne csak azért jöjjön létre valami új termék, mert a mérnökök képesek arra, hogy létrehozzák, hanem mert úgy érzik hogy anélkül nem tudnának élni. Így nem feltétlenül a technológia áll az első helyen.

Egy másik fontos filozófia, amelyhez Jobs leginkább ragaszkodott, hogy a termékek használata egyszerű legyen. Úgy gondolta, ha valaminek a kezelése nem elég egyszerű, akkor az haszontalan lesz a felhasználóknak. Ők ugyanis egyre több szolgáltatást szeretnének, de nincs kedvük tanulni ehhez, azt azonnal használni szeretnék.

Követni kell az egyszerűség elvét is. Ez nem csak a szoftverekre jellemző, hanem az eszköz kínáltra is. Az emberek nem szeretnek a kiválasztással foglalkozni, fontosabbnak tartják az egyszerű kínálatot. Ezért sokáig minden évben csak egy új eszközzel jött ki az Apple egy termékcsaládon belül. Így mindenki tudta, hogy mit kell megvennie ahhoz, hogy lépést tudjon tartani a technológia fejlődésével. Ez most sincs sokkal másképp, hiszen jelenleg is csak évi két termék közül tudnak választani az

iPhone felhasználók. Kezdetben azonban a két eszköz közüli választás is nehézséget okozott számukra.

A számítástechnika úttörőjeként is emlegetett cég gyakran évekkel előbbre jár a versenytársaknál. Nemcsak a forradalmi újdonságai miatt, hanem amiatt is, hogy a meglévő technológiákat folyamatosan jobbá teszi. Talán ezt a tökéletesség utáni vágyat fejezi ki Jonathan Ive, az Apple dizájnerének jelmondata: “Our goals are very simple — to design and make better products. If we can’t make something that is better, we won’t do it.”. Hozzáteszi azt is, hogy amit nem tudnak jobban, azt meg sem próbálják. [1]

2.2 Az iOS

2.2.1 Fejlődése

Az iPhone OS 2007-es megjelenése egy újabb fejezetet jelentett az Apple Inc. sikertörténetében. Ez volt az első generációs iPhone operációs rendszer, amely még nem számított túl korszerűnek abban az időben, hiszen nem tartalmazott több olyan alapvető funkciót, mint például a másolás vagy beillesztés. Ennek ellenére nagyon sikeres lett ez a modell, amit annak köszönhetett, hogy a felhasználói interakciók fogadását új irányból közelítette meg. Itt már nem a gombok, hanem a kapacitív kijelző érintése lett az elsődleges összeköttetés a felhasználó és a készülék között. Ezáltal a vezérlés egyszerűbb és sokszínűbb lett a különböző érintési módszerek alkalmazásával.

Ábra 2-2 : iOS eszközök [3]

Egy évvel később megjelent az első SDK (Software Development Kit) és az alkalmazásbolt (App Store), amelyek segítségével új fejlesztőket vontak be a programok készítésébe. Ezek a fejlesztők motiváltak voltak, hiszen jó bevételi forrást jelentett számukra az eladott alkalmazások után járó részesedés.

Az újabb változás az iPad megjelenésével következett be. Így már nemcsak az iPhone eszközökön futott iPhone OS rendszer, ezért a jelenleg is használt iOS nevet kapta az operációs rendszer.

A hetedik verzió megjelenésével egy eddigiektől eltérő rendszerrel találkozhatunk. Új design készült, amelynek alapja az egyszerűség és a letisztultság volt. Úgy érzem, hogy ezáltal segített a felületeket könnyebben értelmezni. Jobb átláthatóságot biztosított és a fejlesztőknek is teret nyitott ahhoz, hogy egyedi alkalmazásukkal kitűnjenek az alapnézetek közül.

2015 tavaszán az Apple bemutatta új termékkategóriáját, az Apple Watch-ot, amely egy iPhone eszközzel használható okosóra. A fejlesztők bámelyik alkalmazáshoz írhatnak egy órára kiterjesztett programot, amely egy plusz képernyőként funkcionál az értesítések fogadásához.

A jelenlegi iOS verzió a 9-es, amelyre a programomat készítettem. Ennek a verziónak a kiadásával megnyílt a fejlesztők előtt az Apple TV platform is. Folyamatosan készülnek TV-re alkalmazások, amelyhez az is nagyban hozzájárul, hogy az iOS fejlesztői fiókkal rendelkezők 1 dolláros áron juthattak a készülékhez.

Összefoglalásként elmondható, hogy az iOS rendszer folyamatosan fejlődik és várhatóan a jövőben is képes lesz fejlődni versenyt tartva a konkurenciával. Havonta akár több frissítést is kiadnak tele újdonságokkal és újabb innovatív technológiák támogatásával. [2]

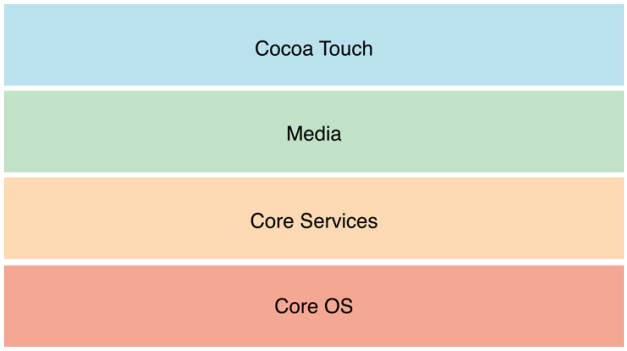
2.2.2 Sajátosságai

Az iOS az OSX-el közös UNIX/BSD alapokra épül, a nagyobb változtatások és frissítések is egyszerre szoktak megjelenni a két operációs rendszeren. Egy lényeges különbség azonban mégis van: az alkalmazások nem kompatibilisek a két platformon. Az iOS-re csak az App Store-ból lehet alkalmazásokat telepíteni, így az Apple teljesen kontroll alatt tudja tartani, hogy milyen alkalmazások érhetők el. Emiatt több kritika is éri, de ennek van jó oldala is, mert szinte minden esetben minőségi és megfelelően működő applikációkkal találkozhatunk.

Az iOS-re jellemző, hogy az alkalmazások egyszerű másolással kerülnek az eszközökre és elkülönülten futnak. Mindegyik alkalmazásnak megvan a saját környezete („sandbox”), és nem látja a teljes fájlrendszert, csak a publikus

könyvtárakhoz fér hozzá. Ezáltal az alkalmazások közötti kommunikáció erősen korlátozott, viszont biztonságos és stabil rendszert eredményez.

Az iOS architektúrája réteges szerkezetű, amely rétegeken a programozás különböző szinteken lehetséges. Ezek a rétegek láthatók a 2-3 ábrán. Az Apple ajánlása az, hogy amikor csak lehetséges, válasszuk a magasabb szintet az alacsonyabb szintű framework-ökkel szemben, mert a magasabb rétegek szolgáltatásai objektum orientált absztrakciót tesznek lehetővé az alacsonyabb szintek felé. Ezáltal kevesebb kódra lesz szükség az ugyanolyan komplexitású feladatok megoldására és a programozás leegyszerűsödik. Az alacsonyabb szintek használata is megengedett és sokszor szükséges is. Előfordulhat ugyanis, hogy a kívánt funkciót nem tudjuk elérni a felsőbb szintek elfedése miatt. [4]



Ábra 2-3 Az iOS rétegei[4]

Az iOS-re fejlesztéskor figyelembe kell venni azt, hogy az erőforrások sokszor limitáltak, ezért a memóriát a legoptimálisabban kell kezelni. Menedzselése fordítási időben történik, mert az alkalmazás futása közben a rendszer már nem ellenőrzi a memóriahasználatot (nincs „garbage collection”). Az automatikus referencia számlálás (ARC) bevezetése óta a programozók feladata egyszerűbbé vált. Ennek lényege az, hogy a fordító elemzi a kódot és elhelyezi benne a memória felszabadítást végző műveleteket. Ez azt jelenti, hogy a programozó számára egy láthatatlan memória kezelő kód generálódik.

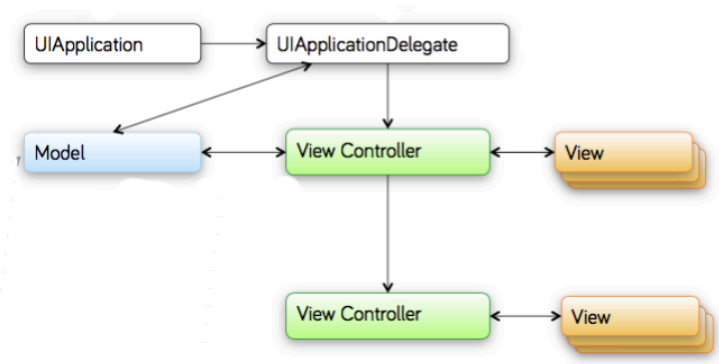
A referenciák tartják életben az objektumokat, így azok akkor törlődnek, ha nem mutat rájuk több erős hivatkozás. Előfordulhat, hogy az objektumok körkörösen egymásra hivatkoznak és ezáltal életben tartják egymást úgy, hogy már nincs is szükség rájuk. Ez a körkörös referenciák problémája, amely memória szivárgáshoz vezethet. Elkerülésére lehetőség van gyenge referenciák létrehozására is, amit az ARC nem vesz figyelembe az objektumra mutató referenciák számolásakor.

Az alkalmazások fejlesztésekor az Apple az MVC (Model-View-Controller) architekturális tervezési minta használatát javasolja.

A View minden, ami a képernyőre rajzol és érzékeli a felhasználó interakcióit, majd továbbítja a Controllernek. Az alapnézet osztályból (UIView) egyszerre több is létrehozható egy képernyőn és egymásba is ágyazhatóak. A Model objektumokkal nem állnak közvetlen kapcsolatban.

A Controller összeköttetést teremt a View és Model objektumok között. Alaposztálya a UIViewController, amely egy összetartozó nézet hierarchiát felügyel, létrehozza őket és feldolgozza a beérkező eseményeket. Maga a Controller sosem rajzol a képernyőre, mert az mindig a nézet feladata.

A Model objektumok tartalmazzák az adatokat, amivel az alkalmazás dolgozik, és az üzleti logikát is. Nincs külön Model osztály, a programozóra van bízva a megvalósítása.



Ábra 2-4 : MVC architektura [5]

A 2-4 ábrán látható, hogy a rendszer minden alkalmazáshoz létrehoz egy UIApplication példányt, majd futtatja az alkalmazás eseménykezelő ciklusát. A UIApplicationDelegate a programozó által megvalósított interfész. Feladata, hogy reagál az alkalmazás életciklus eseményeire (ilyen például az, ha az alkalmazás a háttérbe kerül) és kezeli a beérkező külső eseményeket (értesítéseket). [5]

2.3 Az Xcode fejlesztő környezet

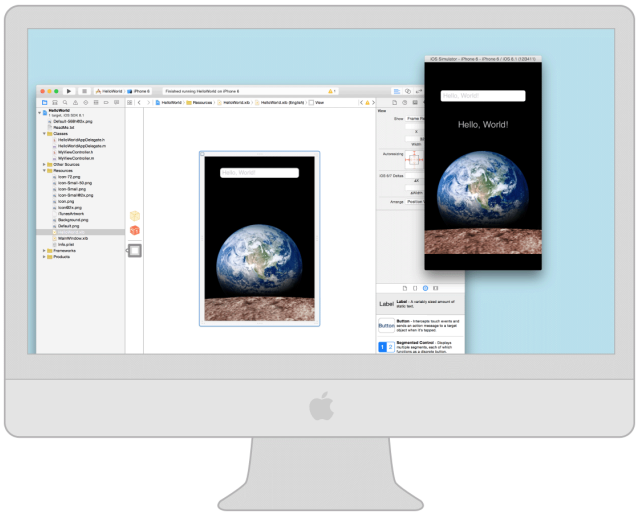
A natív iOS programok fejlesztése OSX rendszeren lehetséges. Az Xcode fejlesztő környezetet használtam a programom készítése során. Beszerzése egyszerű, mert ingyenesen letölthető az alkalmazásboltból. Ez az Apple által fejlesztett integrált

fejlesztői környezet, amelynek telepítése után már létre is lehet hozni az első projektet. Tartalmazza az összes fejlesztéshez szükséges és az azt megkönnyítő funkciókat.

Az Xcode része az iOS SDK, a dubugger és egy LLVM (Low Level Virtual Machine) fordító is C, C++, Objektív-C és Swift programozási nyelvek fordításához. Így egy projekt készítésénél nemcsak az Apple által támogatott nyelveken, hanem C és C++ nyelven írt fájlokat is kapcsolhatunk a projekthez.

Ezeken kívül tartalmaz egy segédprogramot a UI (User Interface) tervezésére is. Ez az Interface Builder, amelynek segítségével grafikus módon, azaz programozás nélkül hozhatunk létre nézeteket. Ez egy jól használható szerkesztő felület, amelyben egyszerűen hozhatunk létre és szabhatunk testre nézeteket, majd az elemeket összeköthetjük a kóddal.

Az Xcode-ba beépített Intruments eszköz segítségével a programok futását tudjuk nyomon követni. A processzor vagy a memóriahasználat figyelésével ki tudjuk szűrni a teljesítmény csökkenéséhez és az alkalmazás leállásához vezető hibákat.



Ábra 2-5 A fejlesztő környezet és a szimulátor [6]

Az Xcode fejlesztő környezetben található egy szimulátor is, amely iPhone, iPad, Apple Watch vagy Apple TV eszközt szimulál OS X alatt. Ez megkönnyíti a fejlesztést, mert az alkalmazások kipróbálhatók célhardver nélkül is.[5]

2.4 Objective-C és a Swift

Az iOS alkalmazás készíthető webes formában, amikor az alkalmazás egy böngésző nézetben fut, vagy natívan is. Natív alkalmazás készítésékor az Objective-C

vagy Swift (esetleg C vagy C++) nyelven megírt program gépi kódra fordul. Natív esetben az iOS funkciói könnyebben elérhetőek és gyorsabban futó alkalmazást is kapunk.

Xcode 6-ig az Objective-C volt az egyetlen hivatalos programozási nyelve az Apple-nek. Ez egy C nyelvre épülő, teljesen objektum-orientált programozási nyelv, amelyben az objektumok közti kommunikáció üzenetküldésen alapul. Sokan idegenkedtek tőle, mert szintaxisa nem hasonlít a mai modern nyelvekéhez és az átlagosnál több kódra van szükség a használatakor. Szükség volt egy nyelvre, amely nemcsak az új programozók, hanem a „veterán" Objective-C –sek számára is vonzó lehet. Így 2014 júniusától egy modernebb, szintén az Apple által fejlesztett programozási nyelv, a Swift is támogatott lett.

A Swift egy script nyelv szerű kompakt szintaxisú nyelv. Nem változtatta meg az iOS fejlesztést teljes mértékben, mert ugyanúgy a framework-ök megtanulása maradt a programozók fő feladata. Az Objective-C –hez képest ez egy tömörebb és jobban átlátható nyelv, amely gördülékenyebb kódolást tesz lehetővé. Szigorúbb szabályainak köszönhetően a hibák nagyrésze már fordítási időben kiderül.

A Swift legfontosabb tulajdonságai közé tartozik, hogy erősen típusos, tehát nincs automatikus típus konverzió. Ha különböző típusok között szeretnénk aritmetikai műveleteket végezni, akkor minden konverziót jelölni kell. (ún. „Castolással") Emellett statikusan típusos is, mert minden változónak vagy konstansnak a deklarálástól kezdve konkrét típusa van.

A nyelvre jellemző még, hogy kötelező a kezdeti érték adás. Viszont előfordulhat, hogy a változónak nem akarunk értéket adni. Erre a célra létrehozhatunk ún. „optional" adattípust, amit minden típusnál egy kérdőjel hozzáadásával tehetünk meg. Ekkor az adott „optional" változó vagy az eredeti típust, vagy az üres (nil) értéket veheti fel.[5]

A két nyelv egyszerre egy projektben is könnyedén használható. Ehhez csak egy „Objective-C bridging header" fájl létrehozására van szükség. Ebbe importálni kell minden olyan Objective-C headert, amit a Swift kódhoz kiterjeszteni szeretnénk. Ha egy Swift osztályt vagy változót szeretnénk Objective-C –ben meghívni, akkor még ennél is egyszerűbb dolgunk van, hiszen csak a projekt modul nevét kell importálni az aktuális Objective-C fájlba a következő módon:

```
#import "ProductModuleName-Swift.h"
```

A feladatomban mindkét programozási nyelvet használtam, ezért is tartottam lényegesnek a legfontosabb tulajdonságaik és a köztük lévő kapcsolatok bemutatását ebben a részben.[7]

3 A digitális hang

„A zene az hangzó matematika” (Kodály Zoltán)

3.1 A hang és tulajdonságai

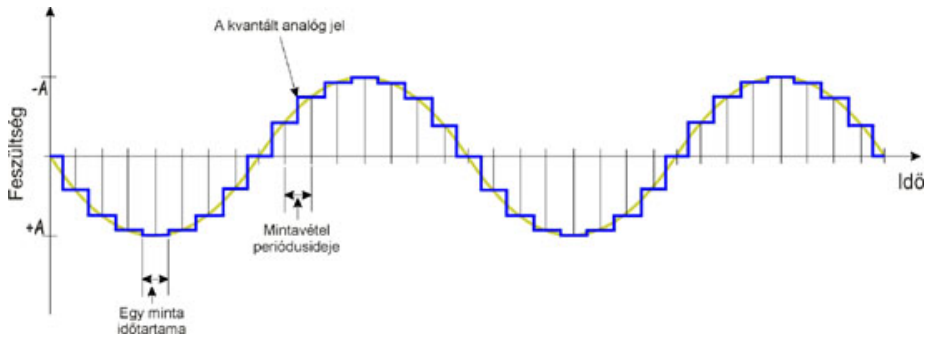
Egy zenei hangot a hangmagasság, az időartam, a hangintenzitás és a hangszín határoz meg egyértelműen. A hang akusztikai jellemzőinek leírásához a legfontosabb fizikai mennyiség a hangnyomás, amely a nyugalmi légnyomásra szuperponálódó légnyomás változás. A hangforrás a közegben (legtöbbször levegőben) mechanikai rezgéseket, ezáltal a fülünk által is érzékelhető hangnyomásingadozást okoz.

A hangfeldolgozás során ezeket a mechanikai rezgéseket alakítjuk át elektromos jellé azért, hogy a hangot alakítani lehessen. Ez az átalakítás kezdetben analóg módon, áramkörök segítségével történt. Majd megjelent a digitális jelfeldolgozás. [8]

3.2 Digitális jelfeldolgozás

Másnéven DSP-nek is szokták nevezni, ami az angol „Digital Singal Processing” kifejezésből ered. Ez egy olyan folyamat, amellyel a csak analóg formában létező fizikai dolgokat valamilyen módon jellemezzük és digitális formába hozzuk. A cél az, hogy számítógéppel is feldolgozható adatokat kapjunk. A digitális jelfeldolgozás előkészítése általában két részből: mintavételezésből és kvantálásból áll.[8]

Mintavételezéskor a folytonos idejű jelből diszkrét idejű jel, majd ezt követően a folytonos amplitúdókból diszkrét amplitúdók keletkeznek a kvantálás során. Az eredeti és feldolgozás során keletkezett jelet mutatja a lenti ábra. [9]



Ábra 3-1 : DSP konverzió [9]

Ahhoz, hogy a jel a mintáiból torzításmentesen visszaállítható legyen, helyesen kell megválasztani a mintavételezési frekvenciát. Ehhez a Nyquist-Shannon mintavételi elvnek kell teljesülnie, ami azt mondja ki, hogy a mintavételi frekvenciának legalább kétszeresének kell lennie a jelben található maximális frekvenciájú komponensnél. A hangfeldolgozás során ezt a frekvenciát az ember hallási képességének megfelelően választjuk meg. A hallási tartományunk 20 Hz és 20 kHz közé esik. A Nyquist-Shannon tételből következik, hogy minimum 40 kHz –es mintavételi frekvenciát célszerű alkalmazni.[10]

3.3 Effektezés

A bejövő hang vagy zene digitális formába alakítása után már lehetőség van arra, hogy azt feldolgozzuk, megmásítsuk, kellemesebbé vagy élvezetesebbé tegyük, azaz effektezzük. Természetesen az effektezés már a digitalizálás előtt is létezett, és sok mai zenész még mindig az analóg rendszereket részesíti előnyben. A digitális rendszerekben azonban matematikai eszközöket használunk, így a lehetőségek tárháza kibővül.

A digitális effektezésnek vannak azonban hátrányai. A feldolgozás során (kvantálás) minden lépésben a kerekítések miatt adatvesztség történik. Emellett az átalakításból adódó időkésleltetés is sok esetben zavaró jelenség. Úgy érzem, hogy ezek a hátrányok a technika fejlődésével kezdenek eltűnni. Egyre fontosabb lesz az, amit már más eszközök esetén is megfigyelhetünk, hogy valami kompakt méretű legyen, ennek ellenére mégis számos funkciót legyen képes megvalósítani. Ma már a gitár hangját egy iPad eszközbe beépített processzorral is fel lehet dolgozni, és nagyméretű érintőképernyőjén egy egyszerűen használható kezelőfelületet is létre lehet hozni. Emiatt nem biztos, hogy szükségünk van nagyméretű és költséges, több analóg effektpedálokból összerakott kombinációkra.

Ábra 3-2 : Effektpedálok [11]

Kihasználva a modern technika előnyeit, létrehoztam egy saját iOS programot a gitár hangjának effektezésére. Ebben az alkalmazásban az elektromos gitárosok számára ismert effektekből valósítottam meg összesen hét darabot, az eredeti formájukat és kezelésüket virtuális környezetbe ültetve. A következő fejezetekben ennek az alkalmazásnak a tervezéséről és megvalósításáról lesz szó.

4 A program szerkezete

4.1 Előkészületek

Az iOS fejlesztéshez OSX rendszerre van szükség, ezért egy MacBook Pro számítógépet használtam a fejlesztéshez. Emellett egy iPad eszközre is szükségem volt a teszteléshez. Ahhoz, hogy a gitár jelét az iPadembe tudjam vezetni, majd a feldolgozott jelet egy hangszóróra tudjam kötni, szükségem volt még egy eszközre. Ehhez az Apple által forgalmazott és zenészeknek szánt kiegészítő családnak, az iRignek egy alap eszközét használtam, amely egy 6.3 mm-es bemenettel és egy 3.5 mm-es sztereó Jack kimenettel rendelkezik. A lenti képen is látható, hogy ezt az eszközt az iPadem felső audio ki- és bemenetébe lehet csatlakoztatni.

Már csak egy elektromos gitárra és egy hangfalra volt szükségem ahhoz, hogy az elkészített effekteket ki tudjam próbálni, és megfelelően tudjam paraméterezni.

Ábra 4-1 : iRig [12]

Az eszközök beszerzése után szükséges a megfelelő szoftverek telepítése is. Az iPad eszközömet a legújabb 9.1 –es iOS verzióra frissítettem, majd az ehhez tartozó 7.1-es verziójú Xcode-ot is letöltöttem az alkalmazásboltból. Ezzel minden készen áll a fejlesztéshez, így a „new Single View Application” kiválasztásával létre is hoztam a kezdő projektemet.

Az iOS 9 egyik újdonsága az, hogy fejlesztői profil vásárlása nélkül is lehet fizikai eszközön az Xcode-ból alkalmazást futtatni. Ennek ellenére mégis szükségem volt egy ilyen fiókra, mert olyan funkciókat is megvalósítottam, amelyekhez ez

elengedhetetlen volt. Ennek a 99 dolláros fióknak a megvásárlása után már fel lehet tölteni az App Store-ba alkalmazást és regisztrálni tudom az eszközöket a teljes értékű teszteléshez.

Korábban az önálló laborom keretében már elkezdtem egy iPhone alkalmazást készíteni, ami fel tudta dolgozni a bejövő audio jeleket. Ezt a programot használtam fel most is kiindulásként.

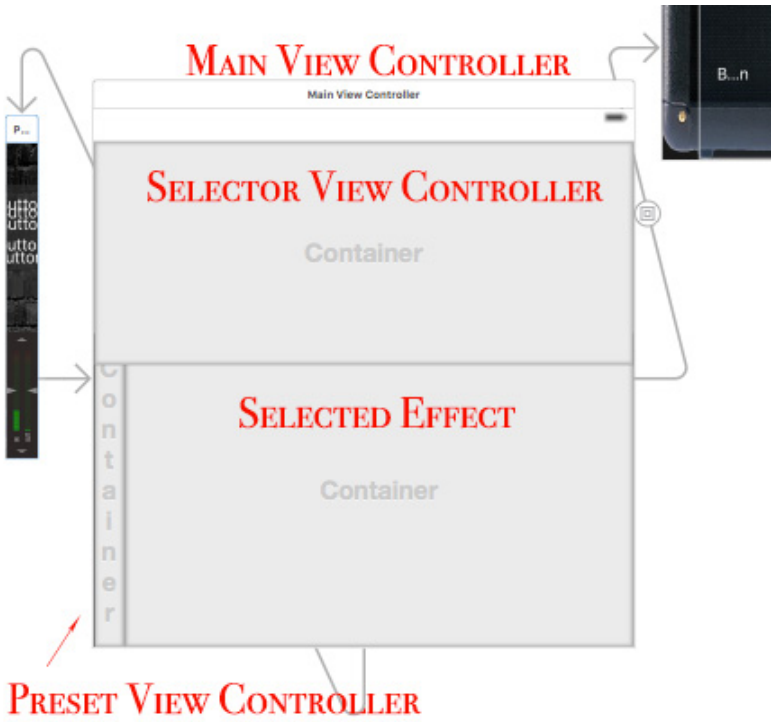
Előre specifikáltam az alkalmazásban megvalósításra kerülő funkciókat azért, hogy a fejlesztés során meghatározott koncepciót követhessek, mivel utólagos módosítások és átalakítások már sokkal több energiát igényelnek és több hibát eredményezhetnek.

4.2 Felhasználói felület

A specifikáció elkészítése után elkezdtem a felhasználói felület tervezését. Fontosnak tartottam, hogy a programom átlátható és könnyen kezelhető legyen, ezért az összes vezérléshez szükséges menü már az alkalmazás nyitóoldalán elérhető. Törekedtem arra, hogy a program esztétikus is legyen. A formai elemeket Photoshop programmal rajzoltam és szerkesztettem.

Az alkalmazáshoz szükséges képernyőket a storyboard szerkesztőben készítettem, így nem kellet kódból létrehoznom az alkalmazás ablakait, mert ezt a keretrendszer elvégezte helyettem. A projekt beállítások között megadtam, hogy a létrehozott storyboard fájlom az alkalmazás indításakor töltődjön be és induljon el.

4.2.1 Main View Controller



Ábra 4-2 : MainViewController

A fenti képen látható egy részlet a storyboard fájlomból, amelyen az alkalmazásom fő vázát alkotó Main View Controller látható. A szerkesztőben az „Is Initial View Controller” kiválasztásával beállítottam, hogy ez legyen az első képernyő az alkalmazás indításakor. Ebbe a View Controllerbe három Containert helyeztem el, amivel gyakorlatilag felosztottam a képernyőt ablakokra. A Containerek segítségével a felhasználói felületből olyan részeket kapok, amelyekhez külön View Controllerek rendelhetők, ezáltal egymástól függetlenül tudom vezérelni őket.

4.2.2 Selector View Controller

A Selector View Controller vezérli a legfelső containerben lévő effekt kiválasztó és ki- bekapcsoló menüt.

UIButton nézettel gombokat helyeztem el a képernyőre. A grafikus szerkesztőben létrehozott UIButtonokat a Selector View Controller osztállyal IBOutletek segítségével kapcsoltam össze. Ezáltal osztály propertyk jönnek létre, így a gombokat már nemcsak a szerkesztőben, hanem kódból is testre lehet szabni. A gombok megnyomásának érzékeléséhez egy IBAction-t kellett létrehoznom, amely a felhasználó

interakciók kezelésére szolgál. A képernyőn lévő gomb megérintésével meghívhatjuk a a Selector View Controller osztálynak a gombhoz tartozó metódusát.

A Selector View Controllerem nem tudja megjeleníteni a kiválasztott effekthez tartozó View Controllert, mert a képernyő azon részéért nem ő a felelős. A kiválasztást tehát továbbítanom kellett a Main View Controllernek, hogy ő töltse be a kiválasztott nézetet a Selected Effect nevű ablakba. Ehhez az NSNotificationCentert használtam, amelyben egy speciális kulccsal értesítéseket adhatunk fel a projekten belül. Az értesítés userInfo részében pedig egy Dictionary típusú érték is továbbítható, amelyben jelezni lehet, hogy melyik effektet választotta ki a felhasználó. A feladás a következőképpen néz ki:

```
NSNotificationCenter.defaultCenter().postNotificationName(mySpec
ialNotificationKey, object: nil, userInfo: ["VC" : 7])
```

Az értesítés fogadásához a fogadó View Controller ViewDidLoad metódusába kell elhelyezni egy figyelőt, aminek a selector részénél megadható az értesítés észlelésekor lefuttatandó metódus azonosítója.

```
NSNotificationCenter.defaultCenter().addObserver(self, selector:
"switchEffect:", name:mySpecialNotificationKey, object: nil)
```

Így a Main View Controller switchEffect metódusában be tudtam tölteni az értesítés userInfojához tartozó ViewControllert a Selected Effect mezőbe.

A menüsorban több a választási lehetőség, mint amennyi a képernyőre kiférne. Emiatt szükség volt a UIScrollView használatára. Ez egy olyan nézet, amely a képernyő húzásával lehetővé teszi a görgetést függőleges és vízszintes irányban is. Használatához egyértelműen meg kell adni a görgetni kívánt tartalom méretét, amit az Auto Layout segítségével tettem meg.

Auto Layoutot használtam az egész projektem során, hogy több képernyőméretet is támogasson a programom. Az Auto Layout egy deklaratív módja a nézetek méretének és pozíciójának meghatározásához. A felület kinézetére vonatkozó elvárásokat, kényszereket lehet meghatározni, ami alapján a rendszer futási időben kiszámolja a pozíciókat és méreteket. A kényszerek, azaz a relációk, a nézetek attribútumai között egy lineáris egyenletrendszert alkotnak, amelynek a megoldása az Auto Layout feladata.[5]

4.2.3 Preset View Controller

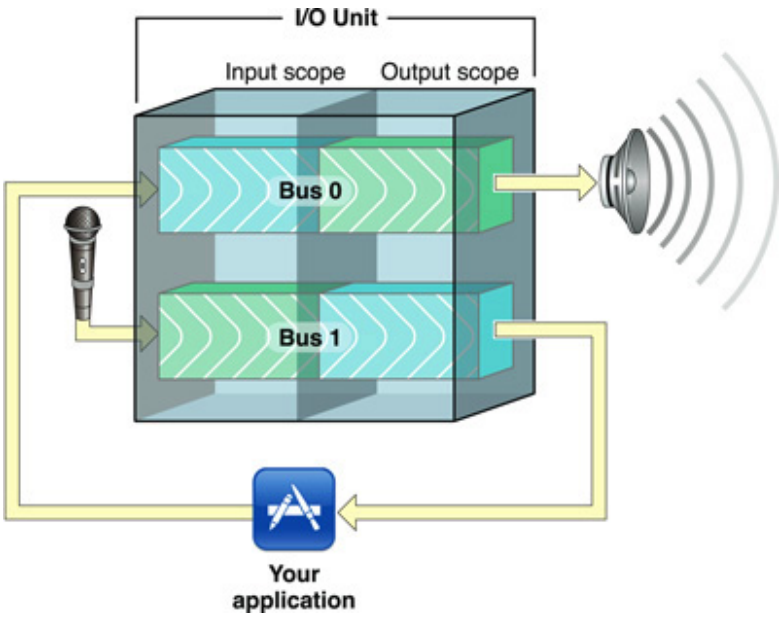
A Preset View Controller osztály felel a bal oldali container vezérléséért. Három effekt beállítási kombinációt lehet eltárolni a programban, amely az összes effektre együttesen vonatkozik. Ebben a nézetben lehet váltogatni a három eltárolt változat, azaz presetek között.

A mentést úgy oldottam meg, hogy mindhárom kombinációhoz létrehoztam egy singleton osztályt, amelyben az effektek View Controlljait példányosítom, hogy preset váltásokkor innen tudjam előhívni őket. Ezt találtam a legegyszerűbb megoldásnak arra, hogy a beállított paramétereket előhívjam, és emellett a potméterek is megfelelő helyzetben legyenek a grafikai megjelenítésnél. Lent látható az 1-es Presethez létrehozott ViewControllerFactory1 osztály egy részlete. Az látható, hogy egy statikus konstansban példányosítom önmagát, hogy singleton osztályként tudjam használni. A következőkben pedig számított propertykkel létrehozom az effektekhez tartozó View Controllerek példányait, biztosítva azt, hogy csak egyszer jöjjenek létre:

```
class ViewControllerFactory1: NSObject {
static let sharedFactory1 = ViewControllerFactory1()

var ef1 : Effect1ViewController?
var effect1ViewController : Effect1ViewController {
    get {if(ef1 == nil)
    {ef1 =
mainStoryboard.instantiateViewControllerWithIdentifier("Effect1"
) as? Effect1ViewController}
    return ef1!}
    }

var ef2 : Effect2ViewController?
     var effect2ViewController : Effect2ViewController {

      …
```

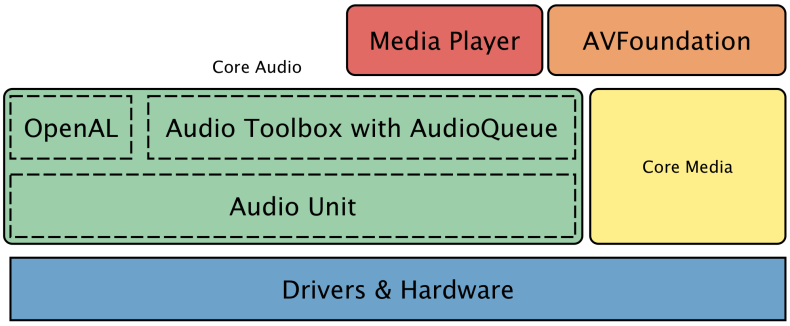
4.3 Audio tartalom feldolgozása

Az Apple többféle megoldást kínál a hangok kezelésére. Ahogy az a 4-3 ábrán látható, a hangfolyamot az eszköz hangkártyáján keresztül lehet fogadni és kiküldeni. A folyamat vezérléséhez egy framework importálása szükséges. A legmagasabb szintű az AVFoundation framework, amely különböző audio formátumok lejátszására és felvételére alkalmas, lehetővé téve a hatékony fájlkezelést.



Ábra 4-3 : I/O Unit

Az AVFoundationnel azonban nem lehet minimális késleltetéssel a kimenetre küldeni a bejövő hangot. A bejövő bufferből sem lehet a hangmintákat elérni. Ezért a hardver közelibb, a Core OS réteget is elérő, AudioToolbox framework Audio Unit részét használtam a programom készítésekor.



Ábra 4-4 [13]

Amint azt a fenti kép is szemlélteti, az Audio Unit alacsonyszintű programozást igényel, ezért használtam Objective-C –t a kezeléséhez. Az Audio Unit ponitereket használ, amit a Swift nyelvben csak nagyon körülményes eszközökkel lehet kezelni. Úgy találtam egyszerűbbnek, ha a két nyelvet vegyítem a projektemben. Ezt a részt Objective-C, amíg a többit Swift nyelven írtam.

Az audio folyamok fogadásához és küldéséhez először az alapvető beállításokat kellett megtennem. Engedélyeztem a ki- és bemeneti buszokat, majd egy 1024 db 16

bites mintából álló buffert hoztam létre úgy, hogy a mintavételi frekvenciát 44100 Hz-re állítottam.

Amikor a bemenetről érkezik bejövő adat, akkor minden esetben a meghívja a rendszer a recordingCallback függvényt. Ebben kezelem a bejövő mintákat és átadom a már Swiftben megírt BufferProc osztálynak, hogy végrehajtsa rajta a feldolgozást és az effektezést. Alul látható az átadás, ahol a BufferProc osztályt azzal az inicializálójával példányosítom, amiben átveszi a feldolgozandó buffert és annak méretét.

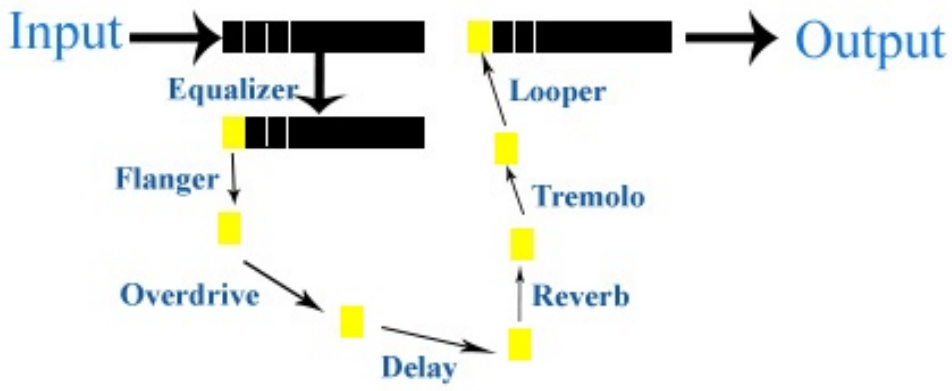
```
[[BufferProc alloc] initWithBuffer:editBuffer size:Buffersize];
```

A minták manipulálása után a playbackCallback függvényt a recordingCallback függvényhez hasonlóan meghívja a rendszer, és a mintákat átadja az kimeneti busznak. Erről részletesen olvashatunk a következő fejezetben.

5 Megvalósított effektek

A gitárosok rendkívül sok fajta effektet használnak. Vannak, akik saját maguknak építenek vagy programoznak, hogy minél egyedibb hangzásvilágot valósítsanak meg. Talán többen vannak, akik a már jól bevált, évtizedek alatt kifejlődött, talán azt is mondhatom, hogy alap effektek közül választják ki a megfelelő kombinációt a játékukhoz.

A programom készítése során az volt a célom, hogy az ilyen alap effektek működését megértve, én is megvalósítsam őket úgy, hogy a megszokott hangzásukat a lehető legjobban közelítsem. Az Overdrive, Delay, Equalizer, Flanger, Tremolo, Reverb és Looper effekteket valósítottam meg úgy, hogy ezek az 5-1 ábrán látható módon egy meghatározott elrendezéssel sorba vannak kötve.



Ábra 5-1 : Az effektek sorrendje

5.1 Overdrive

Ábra 5-2 : OverDrive

5.1.1 Működési elv

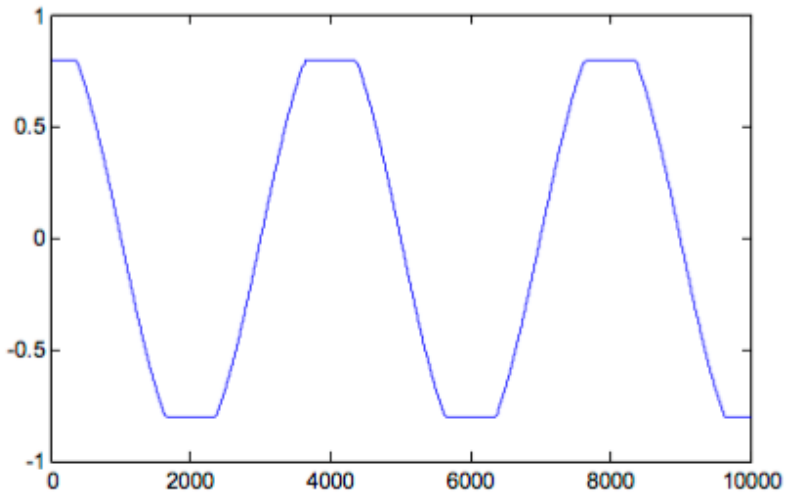
Talán ez a legfontosabb és legelterjedtebb gitáreffekt, amelyet már évtizedek óta alkalmaznak. Eleinte a csöves erősítők túlhajtásával érték el a torzítást, de manapság már számos dedikált eszköz áll rendelkezésre a különböző zenei műfajokhoz.

Közös jellemzőjük, hogy a jelgörbe tetejét valahogyan meg kell törni, és ezáltal kapjuk ezt a különleges hangzást. Ezt a digitális környezetben jelszint korlátozással értem el. [15]

5.1.2 Megvalósítás

Az Overdrivelevel változó az egyetlen paramétere az effektemnek, amely a korlátozást befolyásolja. A következő két soros kód végzi a műveletet a programban. Először az Overdrivelevel változótól függő értékkel szorzom a Sample hangmintámat, majd az –OverdriveLevel alatt és +OverdriveLevel felett levágásra kerül.

```
Sample *= (Double(1 - Overdrivelevel) * 10)

Sample = (Sample < -Double(Overdrivelevel)) ? -
Double(Overdrivelevel) : (Sample > Double(Overdrivelevel)) ?
Double(Overdrivelevel) : Sample
```



Ábra 5-3 : Jelszint korlátozás [15]

5.1.3 Grafikai felület és vezérlés

A potméter és így a paraméter állítása két ujjas forgatással történik, olyan hatást keltve, mintha megfognánk és tekernénk a szabályzót. Ehhez egy üres és átlátszó nézetet hoztam létre arra a felületre, ahol a forgatást érzékelni szeretném. Ezt rotateViewnak neveztem el, és ehhez adtam hozzá a UIRotateGestrureRecognizert a következő módon.

```
let rotationGestrure = UIRotationGestureRecognizer()
rotationGestrure.addTarget(self, action: "handleRotation:")
self.rotateView.addGestureRecognizer(rotationGestrure)
```

Innentől már csak annyi volt a feladatom, hogy a handleRotation függvényben kezeljem az elforgatást. A rotateViewt, amely a potmétert ábrázoló képet tartalmazza, a forgatás szögével forgattam el, és a paramétert is a szöggel arányosan változtattam meg.

```
func handleRotation
(rotationGesture:UIRotationGestureRecognizer) {

self.rotateView.transform =
CGAffineTransformRotate(self.rotateView.transform,
rotationGesture.rotation)

self.overdrivelevel += Float(rotationGesture.rotation)
Overdrivelevel = self.overdrivelevel
rotationGesture.rotation = 0.0
}
```

Ezen kívül még elhelyeztem egy felhasználói interakciót fogadó nézetet a képernyőre. Ezt a tapView névvel ellátott nézetet pont a gomb fölé tettem, és egy

UITapGestureRecognizert adtam hozzá. Ha a felhasználó megérinti a felületet, akkor ez érzékeli, és meghívja a handleTap függvényt.

```
let tapGestrure = UITapGestureRecognizer()
tapGestrure.addTarget(self, action: "handleTap")
self.tapView.addGestureRecognizer(tapGestrure)
```

A handleTap függvényben kapcsolom be vagy ki az effektet, és küldök egy értesítést a SelectorViewControllernek az effekt állapotáról. Már csak a fent lévő kis piros visszajelző lámpát ábrázoló képet cseréltem le a megfelelőre.

```
func handleTap() {

NSNotificationCenter.defaultCenter().postNotificationName("setBy
Pass1", object: nil)

self.isON = !isON
if(isON){self.redLed.image = UIImage(named: "redLightOn.png")}
else {self.redLed.image = UIImage(named: "lightOff.png")}
OverdriveIsOn = self.isON
}
```

A többi effektnél is hasonló grafikai vezérlési megoldást alkalmaztam, ezért a továbbiakban már térek ki rá ilyen részletesen.

5.2 Delay

Ábra 5-4 : Delay

5.2.1 Leírás

A visszhang jellegű effektek valamilyen formában tárolják a beléjük érkező hangot, majd azt adott időközönként, adott számban visszajátsszák, megismétlik. Ezt kezdetben szalagos magnókkal állították elő, amelyek egy körbe-körbe játszott szalagra rögzítették a hangot. Az olvasófejek segítségével késleltették, majd visszaolvasva az eredeti hanghoz adták hozzá. A mai modern visszhangok zöme digitális, ahol már a hangot memóriában tárolhatjuk és onnan játszhatjuk vissza egy beállított idő után.

Az olyan effektet, amely csak egyetlen másolatot képez az eredeti hangról, echonak nevezzük. Sokszor azonban nem elég egyetlen ismétlés, ezért kitalálták, hogy az effektből kijövő jelet visszacsatolják az elejére. Így már megvan a lehetőség arra, hogy újra és újra megismételjük a hangot. Az így létrehozott effektet hívják többnyire delay-nek. [16]

5.2.2 Megvalósítás

Ahogy az 5-5 ábrán látható, a Delay megvalósításához szükségem volt egy bufferre, amihez egy 50000 elemből álló 0.0 értékkel kitöltött tömböt hoztam létre. Ebben a tömbben a helpIndex változó segítségével fogok írni és olvasni.

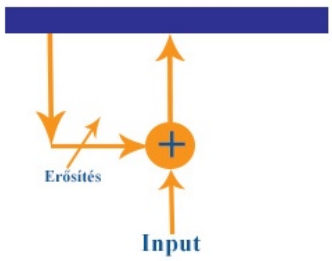
```
var helpBufferCount = 50000
var helpBuffer=[Double](count:helpBufferCount,repeatedValue:0.0)
var helpIndex : Int = 1
```

Ezt az effektemet két paraméteresre terveztem. A DelayLevel potméter állítja, hogy hány mintányi késleltetést szeretnénk. A delayIndex változóba számolom ki, hogy a helpBuffer tömbből hányadik elemet kell kivennem. A túlindexelést a kimásolt kódrészletem második sorában kezelem. A második FeedbackLevel paraméterrel állítható az, hogy a késleltetett mintát mekkora hangerővel adjuk hozzá az aktuális hangmintához.

```
var delayIndex = helpIndex-Int(DelayLevel)
delayIndex = (delayIndex<0) ? delayIndex + helpBufferCount :
delayIndex
let DelaySample : Double = helpBuffer[delayIndex]
Sample += DelaySample * Double(FeedbackLevel)
```

Miután az összes effekt rákerült a hangmintámra, eltárolom azt.

```
helpBuffer[helpIndex] = Sample
helpIndex++
```



Ábra 5-5 : Delay blokkvázlat

Azáltal, hogy nem a direktet, hanem a már módosított hangmintát mentem el, valóban létrehozom a Delay hangzást. Az ismétlést addig fogjuk hallani, amíg az nem hallható hangosságúra csillapodik.

5.3 Equalizer

Ábra 5-6 : Equalizer

5.3.1 Elméleti háttér

Ezt az effektet azért találták ki, hogy a gitárerősítő szokásos mély, közép és magas hangszín szabályzóján túlmenően részletesebben is bele lehessen nyúlni a hangszínbe. A hangszínszabályozást nem csak a gitárosok és zenészek használják, hanem a zenehallgatók is. A hifi lejátszókon vagy akár a telefonok, számítógépek média lejátszó programjában is lehetőségünk van bizonyos frekvenciájú komponenseket kiemelni vagy elnyomni. Ezáltal a hangot ízlésünk szerint alakíthatjuk, vagy a berendezés és a lejátszó helyiség hiányosságait kompenzálhatjuk. [16]

Ennek a megvalósításához a szokásos időtartománybeli szűrés helyett azt a megoldást választottam, hogy az időtartományban lévő hangmintákat diszkrét

frekvencia értékekké alakítom át. Ez a folyamat a Diszkrét Fourier Transzformáció (DFT), amelyet a következőképpen definiálhatunk.

$$C_m = \sum_{n=0}^{N-1} C_n e^{\left[\frac{-i2\pi}{N}\right]^{nm}}$$

Ábra 5-7 DFT [18]

Ahol Cn az időtartománybeli, míg Cm a frekvenciatartománybeli komplex értékek sorozata, és N a minták száma. Kezdetben ennek az algoritmusnak a leprogramozásával próbálkoztam, de az elvégzendő műveletek száma $N^{2,}$, amely az élő feldolgozáshoz túl nagy műveletigényű.

Mivel 1024, azaz kettő hatványú mintát tárolok a bufferben a feldolgozáshoz, használhatom a rekurzív működésű Gyors Fourier-Transzformációt (FFT - Fast Fourier Transform) is. Ennek a műveletigénye már csak N*logN. [17]

Miután a frekvencia mintákat megkaptam, különböző frekvenciájú komponenseket tudok kiemelni vagy elnyomni azok nagyságának állításával. A változtatásom érvényre jutásához az inverz transzformáció elvégzésére is szükséges azért, hogy újra időtartománybeli értékeket kapjak.

5.3.2 Megvalósítás

A megvalósításhoz az Accelerate framework vDSP szolgálatását használtam, amely Apple által fejlesztett, digitális jelfeldolgozásra optimalizált függvényeket tartalmaz. A „v” azt jelenti a nevében, hogy amikor csak lehet, vektor műveleteket végez a minél gyorsabb működés eléréséért.

Az FFT végrehajtásához először létre kell hoznom egy struktúrát az értékek tárolásához. A splitComplex változó egy valós és egy képzetes értékeket tároló tömbből áll. A valós részébe az 1024 darab időbeli mintából álló tömböt töltöm be.

```
var real = [Double](input)
var imaginary = [Double](count: input.count, repeatedValue: 0.0)
var splitComplex = DSPDoubleSplitComplex(realp: &real, imagp:
&imaginary)
```

Ezután a weights változóba elmentem az FFT beállításaimat, azaz a minták számának kettő alapú logaritmusát, és megadom a végzendő műveletek pontosságát az FFT előkészítéséhez.

```
let length = vDSP_Length(floor(log2(Float(input.count))))
let weights=vDSP_create_fftsetupD(length, FFTRadix(kFFTRadix2))
```

Ezek után már meg tudom hívni a vDSP_fft_zipD függvényt a lent látható paraméterekkel. A függvény elvégzi az FFT műveletet, és felülírva a splitComplex változót, eltárolja az értékeket.

```
vDSP_fft_zipD(weights, &splitComplex, 1, length, FFTDirection(FFT_FORWARD))
```

A splitComplex változó 0. eleme most tartalmazza a DC offszet értékét, amely a 0 Hz-es komponenshez tartozó valós érték. Majd a többi eleme a mintavételi frekvencia és a mintaszám hányadosának értékével növekvő frekvenciákhoz tartozó komplex értékeket tartalmazza. Így nálam az 1. elem a 44100 / 1024, azaz a közel 43 Hz-hez tartozó érték, a második elem ennek a kétszereséhez, és így tovább.

Ha megvannak a frekvenciakomponensek, akkor utána megvalósítom a szűrést, amihez a complexFilter komplex változót hozom létre egy külön függvényben, a csúszkák állásának megfelelően. A két komplex vektor szorzását a vDSP_zvmulD függvénnyel hajtom végre a következő módon.

```
vDSP_zvmulD(&splitComplex, 1, &complexFilter, 1, &splitComplex, 1, vDSP_Length(input.count), 1)
```

Ezzel módosítottam a frekvenciákat a beállított értékeknek megfelelően, de ahhoz, hogy újra időtartománybeli értékeket kapjak, el kell végeznem az inverz FFT műveletet ugyanúgy a vDSP_fft_zipD függvénnyel, de ekkor már az irány paramétert megváltoztatva.

```
vDSP_fft_zipD(weights, &splitComplex, 1, length, FFTDirection(FFT_INVERSE))
```

Ezt követően a splitComplex változó valós részében már az átalakított értékeim szerepelnek, amelyeket már használhatok a további feldolgozáshoz. [19]

5.3.3 A grafikus vezérlés és a szűrő létrehozása

Ennél az effektemnél a felhasználói felület különbözik a többitől, mert nem potméterekkel, hanem csúszkákkal állíthatók a paraméterek. Ehhez az interface builderben a képernyőre UISlidereket helyeztem el, beállítva a minimum, maximum és az alapértelmezett értékeket.

Mindegyik UISlider elemhez létrehoztam a képernyőhöz tartozó View Controllerben egy outlet propertyt.

```
@IBOutlet weak var Slider1: UISlider!
```

Majd mindegyikhez létrehoztam akció metódusokat is, amelyeket a rendszer az egyes Sliderek értékének megváltozásakor hív meg. Az EQ1 egy globális változó a projektben.

```
@IBAction func Slider1(sender: AnyObject) {
    self.eQ1 = Slider1.value
    EQ1 = self.eQ1}
```

A hat darab EQ érték segítségével hoztam létre a szűrő formáját. Az értékeket 0 és 11025 Hz között logaritmikusan helyeztem, biztosítva köztük az átmenetet.

5.4 Flanger

Ábra 5-8 : Flanger

5.4.1 Működési elv

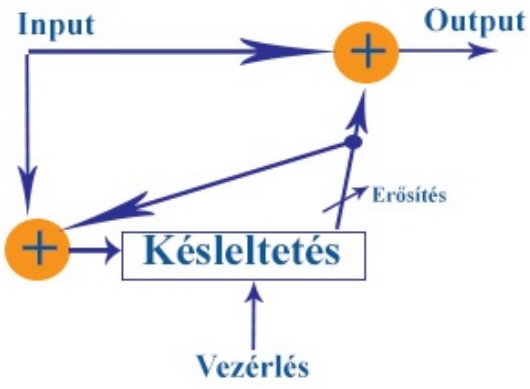
Az első Flanger-t úgy hozták létre, hogy a zenei anyagot egy időben rögzítették két darab szalagos magnetofonra, majd lejátszáskor a két jelet összegezve elküldték a hangszórókra.

A jellegzetes lebegő hangzás a két magnetofon kismértékű sebesség-eltéréseinek hatására szólalt meg. Az effekt hatásának növelése érdekében időnként mechanikusan (fékezéssel) csökkentették az egyik magnó lejátszási sebességét, majd amikor elengedték, akkor újra visszagyorsult a magnó. Így valósították meg a folyamatosan változó, markáns lebegő hangzást.

Ha a fenti módszert jelfeldolgozás szempontjából vizsgáljuk, akkor a következőket állapíthatjuk meg: ha az eredeti jelhez az egész rövid idővel késleltetett változatát hozzáadjuk, akkor az nem más, mintha a jelet egy fésűszűrőn eresztenénk át. Ebben az esetben bizonyos frekvenciatartományok erősítik, míg mások kioltják egymást.

Ha a késleltető időparaméterét kis mértékben, folyamatosan, de lassan változtatjuk, akkor máris elkészítettük a Flanger-nek nevezett effektet. Ezt a jelenséget sweeping comb filteringnek is szokták nevezni, ami azért találó, mert olyan érzésünk van, mintha valami oda-vissza söpörne az egész frekvenciatartományon.[15][20]

A megvalósítás sematikus blokkvázlata lent található. A visszacsatolás alkalmazása még egyedibbé teszi a hangzást.



Ábra 5-9 Flanger blokkvázlat

5.4.2 Megvalósítás

Ezt az effektet két paraméteresre készítettem el. A flangerValue megadja, hogy melyik legyen az az eltárolt minta, amely körül -10 és +10 intervallumban változtatjuk a késleltetést. A flangerDelay az a folyamatosan változó érték, amely befolyásolja azt, hogy az eltárolt értékek közül melyiket adjuk hozzá az élő hangmintához. A flangerIndicator egy Bool típusú, az intervallum bejárását segítő segédváltozó.

```
if(flangerIndicator){
      flangerDelay += 0.1
      if (flangerDelay > flangerValue + 10){
                        flangerIndicator = false
}                         }
else{
       flangerDelay -= 0.1
       if (flangerDelay < flangerValue - 10){
                         flangerIndicator = true
}                         }
```

Ezután már a flangerSample-t a Delay effektnél a delaySample kiszámításához hasonlóan tettem meg és a lent látható módon a másik, flangerFeedback paraméter által beállított erőséggel az aktuális mintához adtam.

```
Sample += flangerSample * Double(flangerFeedback)
```

Végül ezt a feldolgozott mintát mentettem el a memória bufferbe. Ezzel a visszacsatolással a hangzás még jellegzetesebb lett.

5.5 Tremolo

Ábra 5-10 : Tremolo

5.5.1 Leírás

Ez az effekt talán a legegyszerűbben megvalósítható. Pontosan olyan hatást kelt, mintha tekergetnénk a hangerő potméterét a gitáron, ugyanis a gitár hangerejének modulálását valósítja meg. Mérnökileg egyszerű amplitúdó-modulációról beszélhetünk. [15]

5.5.2 Megvalósítás

A 5-6 ábrán látható, hogy ennek az effektnek csak egy paramétere van, amelyet a Tremolo változó rögzít a programban. A potméter állításával szabályozható, hogy a hangerő változtatás mekkora sebességű legyen. A TremoloIndicator Bool típusú változó, amely segíti a TremoloValue változónak a 0 és 1 közötti intervallumot bejárni. Ezáltal a modulációs mélység 100 %.

```
if(TremoloIndicator){
  TremoloValue += Tremolo
  if (TremoloValue > 1.0){
     TremoloValue = 1.0
     TremoloIndicator = false
           }            }
 else{
   TremoloValue -= Tremolo
   if (TremoloValue < 0.0){
      TremoloValue = 0.0
       TremoloIndicator = true
          }            }
```

A TremoloValue változó lesz az, amellyel a hangmintát minden esetben összeszorzom a következő módon.

```
Sample += Double(TremoloValue)
```

Ez az effekt valóban nem igényelt bonyolult programozást és struktúrákat, hiszen pár sorban meg tudtam írni. Bekapcsolásakor létrejön a jellegzetes remegő hangzás.

5.6 Reverb

Ábra 5-11 : Reverb

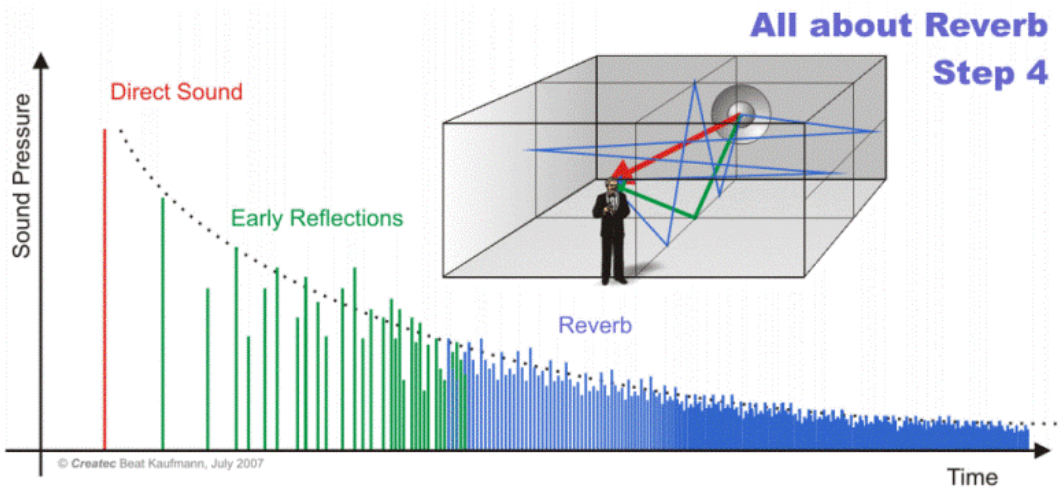
5.6.1 Működési elv

A zenében valószínűleg a leggyakrabban használt effekt a Reverb, azaz a zengető. Amikor valaki a „Reverb” szót hallja, egyből valamilyen kispedál, effekt processzor jut eszébe. A legtöbb ember bele sem gondol abba, milyen fontos a zengetés, és hogy mindennap találkozunk ezzel a jelenséggel mindenféle processzor nélkül is.

A zengés oka a hang sok-sok visszaverődése, ami egy adott térben keletkezik. Bármilyen hangforrásról is beszélünk, legyen az a hifi berendezésünk hangfala, vagy maga az emberi beszéd, esetleg énekhang, a hang egy része közvetlen úton jut el a fülünkig, más része viszont hosszabb utakat bejárva, a tér különböző felületeiről (falak, plafon, különféle tárgyak) visszaverődve érkezik el hallószervünkhöz. Mivel ez utóbbi esetben a hang hosszabb utat jár be, ezért később ér el a hallgatóhoz és természetesen sokkal gyengébb is lesz, mert a tér különböző felületei elnyelik a hanghullámok egy hányadát. Persze ezek a visszaverődések is tovább csapódhatnak és újabb felületek érintésével érkezhetnek a fülünkhöz, és így tovább. A hanghullámok ilyen késleltetett és gyengített sorozatát nevezzük zengésnek, és ez adja az adott tér, például egy szoba vagy terem „térhatását”.

A 5-8 ábrán egy terem úgynevezett hisztogramját láthatjuk. A zengetés során, a direkt hang (piros csík) érzékelését követően nagyon rövid időn belül egy jól meghatározható, korai visszaverődési halmaz éri el fülünket. Ez a jelenség szorosan

összefügg a terem alakjával és méretével, valamint a hangforrás és a hallgató elhelyezkedésével. A korai visszaverődések (early reflections) után a visszaverődések beérkezési sűrűsége egyre nő, és sokkal véletlenszerűbbé válik. Ezt a jelenséget diffúz zengésnek vagy késői visszaverődésnek szokták nevezni. A diffúz zengés tulajdonképpen az elsődleges tényező egy terem méretének érzékelésében.



Ábra 5-12 : Hisztogram [21]

A zengés idejét elsődlegesen a teremben található felületek és a terem mérete befolyásolja. A terem felületei határozzák meg, hogy mennyi energia vész el az egyes visszaverődések során. A mérete pedig azt, hogy a hanghullámoknak mekkora utat kell megtenniük.

Bár a zengés mindig körülvesz bennünket, mégis szükség van Reverb effekt használatára. Megszoktuk, hogy egy élő produkció többnyire nagy méretű és jó hangzású termekben hangzik el. Ezzel szemben a felvételek lehallgatása többnyire kisméretű helyiségekben történik, amelyek hangzásképe teljesen más. A legnagyobb különbség leginkább úgy fogalmazható meg, hogy a „térhatás otthon kicsi”. Ez műszakilag azt jelenti, hogy a sok elnyelő felület (szoba bútorzata) és a rövid visszaverésmentes hangutak miatt az átlagos lehallgató helyiségek utózengési ideje sokkal kisebb, mint azé a teremé, ahol amúgy az élőprodukció elhangzana.

Tehát azért, hogy a gitár hangzását ne érezzük száraznak egy kisebb teremben sem, a Reverb effekt segítségével többletvisszhangokat adhatunk a felvételhez, vagy az élő játékhoz. [15][16]

5.6.2 Megvalósítás

A Delayhez hasonlóan itt is egy tömböt hoztam létre a hangminták tárolásához. Ezt a tömböt a reverbIndex segítségével fogom bejárni.

```
var reverbBufferCount = 100000
var reverbBuffer = [Double](count: reverbBufferCount,
repeatedValue: 0.0)
var reverbIndex : Int = 1
```

Itt viszont a visszhangok ráadása előtt még eltárolom a mintát, hogy ne jöjjön létre az ismétlődő Delay effektus.

```
reverbBuffer[reverbIndex] = Sample
reverbIndex++
```

Ezután hét darab mintát veszek a reverbBuffer tömbömből a következő kódrészletben látható módon. A ReverbValue paraméterrel lehet állítani a visszhang minták direkt hangtól való távolságát. Ha ezt nagyobb értékre állítjuk, akkor a késleltetés, és ezáltal a térérzetünk is nagyobb lesz. A ReverbFeedback a másik paraméter, amellyel a késleltetett minták intenzitását befolyásolhatjuk.

```
var reverbIndex1 = reverbIndex - (1 * Int(ReverbValue))
reverbIndex1 = (reverbIndex1<0) ? reverbIndex1 +
reverbBufferCount : reverbIndex1
var reverb1 = reverbBuffer[reverbIndex1]
reverb1 *= Double(ReverbFeedback/1.0)

var reverbIndex2 = reverbIndex - (2 * Int(ReverbValue))
reverbIndex2 = (reverbIndex2<0) ? reverbIndex2 +
reverbBufferCount : reverbIndex2
var reverb2 = reverbBuffer[reverbIndex2]
reverb2 *= Double(ReverbFeedback/2.0)

var reverbIndex3 = reverbIndex- (3 * Int(ReverbValue))

…
```

Miután a reverb hangmintákat egy változóban eltároltam, az aktuális hangmintához adtam a következő egyszerű formulával.

```
Sample += reverb1 + reverb2 + reverb3  + reverb4  + reverb5  +
reverb6 + reverb7
```

5.7 Looper

Ábra 5-13 : Looper

5.7.1 Leírás

Ez az effekt valamennyire különbözik az előzőektől, hiszen itt nem az egyes hangok színezése a cél, hanem az, hogy egy felvett zenei részletet folyamatosan ismételve játsszunk vissza. Általában több sávot is rögzíthetünk, így akár egy egész kórus létrehozható egyedüli énekesként is. A gitárosok is szeretik alkalmazni még élő koncerteken is, hiszen a zenei számokban általában sok ismétlődő rész van. Ezeket elég egyszer feljátszani, és utána új szólamokkal lehet színezni a zenét.

A digitális effektezés megjelenésével egyre inkább meghatározó lett a Looperek szerepe. Ma már vannak olyan kis létszámú együttesek, akik looper zenekarnak nevezik magukat, hiszen ennek az effektnek a használata lett a fő profiljuk.

5.7.2 Megvalósítás

A megvalósításhoz egy enumeration típust hoztam létre, amelyben négy állapotot vettem fel.

```
enum looperState {
    case recordingFirstly
    case play
    case record
    case stop
}
```

A currentLooperState változó tárolja az aktuális állapotot. Kezdetben stop állapotból indul a rendszer, ahol nem végez műveletet. Ha a felhasználó megérinti a

középen található gombot, akkor átkerül recordingFirstly állapotba. Ebbe csak egyszer kerülhet, az első felvétel esetén.

Ebben az állapotban a looperIndex futó változó segítségével a looperBuffer tömbbe mentem el az aktuális hangmintákat. Ezt a buffert úgy hoztam létre, hogy 1000000 darab mintát tudjon tárolni, így körülbelül 23 másodpercet tudunk felvenni. (1000000/44100) Ha ez a buffer megtelt, akkor automatikusan record állapotba kerül a rendszer. Az első felvétel határozza meg az ismétlődő szakasz hosszát. A looperValue változóba pedig azt mentem el, hogy éppen hány mintát tárol a buffer.

Ha a felhasználó másodszor is megérinti a gombot, akkor egy zöld led kigyulladása jelzi, hogy play állapotba kerültünk. A play állapotban a looperBufferből looperValue darab mintát adok körkörösen az aktuális mintához. Ez egészen addig ismétlődik, amíg egy újabb érintéssel record állapotba nem kapcsolunk. Ekkor újabb szólamokat tudunk felvenni a meglévőre, úgy, hogy az aktuális mintát mindig az eddig a bufferben tárolt értékhez adjuk.

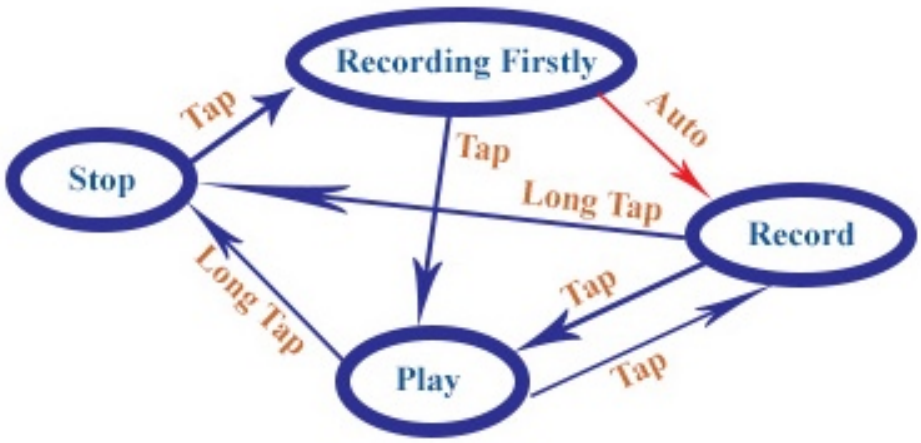
Egy újabb érintéssel play helyzetbe kerülhetünk vissza. A fent leírtakat a következő kóddal valósítottam meg a projektemben.

```
switch currentLooperState {
        case .stop: break

        case .recordingFirstly:
            looperBuffer[looperIndex] = Sample
            looperValue = looperIndex
            looperIndex++
            if looperIndex >= looperBufferCount {
            currentLooperState = .record
            looperIndex = 0}
        case .play:
            Sample = Sample + looperBuffer[looperIndex]
            looperIndex++
            if looperIndex > looperValue {
            looperIndex = 0}
        case .record:
            looperBuffer[looperIndex] += Sample
            Sample = looperBuffer[looperIndex++]
            if looperIndex > looperValue {
            looperIndex = 0}

    }
```

Most már bármennyiszer tudunk felvenni és lejátszani. Talán egy dolog hiányzik még: a leállítás és visszatérés stop helyzetbe. Ehhez nem helyeztem el plusz érintési felületet, hanem egy új felhasználó interakciót fogadó lehetőséget, a hosszan érintést használtam. Ehhez a lent látható módon létrehoztam egy UILongPressGestureRecognizert és a tapView nézetemhez adtam.

```
let longTapGesture = UILongPressGestureRecognizer()
longTapGesture.addTarget(self, action: "handleLongTap")
 self.tapView.addGestureRecognizer(longTapGesture)
```

Innen már csak a handleLongTap függvény megírása volt a feladatom, amelyben az effektemet újra kezdeti állapotba helyeztem. Az állapotok közötti kapcsolatotokat a következő ábra szemlélteti:



Ábra 5-14 : Looper állapotok

6 Üzleti modell

6.1 Lehetőségek

Az App Store alkalmazás áruházban számos kategóriából választhatunk programokat magunknak, amelyek többféle céllal íródtak és ezért különböző üzleti modellt testesítenek meg. Ilyen például a freemium, az üzletitevékenységet közvetlenül támogató vagy az In-App purchasing modell.

6.1.1 Freemium

Ezek közül a „freemium” modell talán az egyik leggyakrabban használt megoldás. Lényege, hogy maga az alkalmazás ingyenesen letölthető formában minél több felhasználó telefonjára rákerüljön, majd a kialakult közösségre építve bevételt generáljon különböző megoldásokon keresztül.

A jelentős mennyiségű felhasználóbázissal rendelkező applikációk esetében működőképes modell lehet a hirdetési felületek értékesítése is. A hirdetések megjelenítésére használhatóak a különböző partnerhálózatok (pl. Google Admob), illetve alkalmazható saját értékesíti hálózat is. Az előbbi esetében számolni kell azzal, hogy nincs ráhatásunk a megjelenő hirdetésekre, míg a saját értékesítési hálózat fenntartása jelentős költséggel és munkaerő igénnyel járhat. A modell veszélye, hogy a közösség a hirdetéseket zavarónak találhatja.

Létrehozható még egy olyan modell, ahol elérhető az alkalmazás ingyenes és fizetős verziója is, a kettő között pedig kizárólag annyi a különbség, hogy megjelennek-e a hirdetések, vagy nem. Így a felhasználó maga dönti el, hogy hajlandó-e megvásárolni a hirdetésektől mentes verziót, vagy továbbra is használja az ingyenes alkalmazást, ahol folyamatosan találkozik a reklámokkal.

Természetesen az is előfordulhat, hogy egy alkalmazás úgy ingyenes, hogy nincs mögötte semmilyen modell és a fejlesztők jóindulatából került fel az App Store-ba. Talán mondhatom, hogy az iOS platformra fejlesztés nagyobb költségekkel jár, mint más mobil platformokra. Egyrészt azért, mert Apple eszközök szükségesek hozzá, amelyek általában premium árkategóriásak. Másrészt pedig szükséges a fejlesztői profil megvásárlása is. Ezek után kevés fejlesztő fogja azt mondani, hogy nem tart igényt

semmilyen bevételre a letöltők után. Ennek előnyei is vannak, hiszen a fejlesztés így teljesen az Apple kontrollja alatt történik, és nincs lehetőség működésképtelen vagy hibás alkalmazások megosztására. [22]

6.1.2 Üzleti tevékenységet közvetlenül támogató

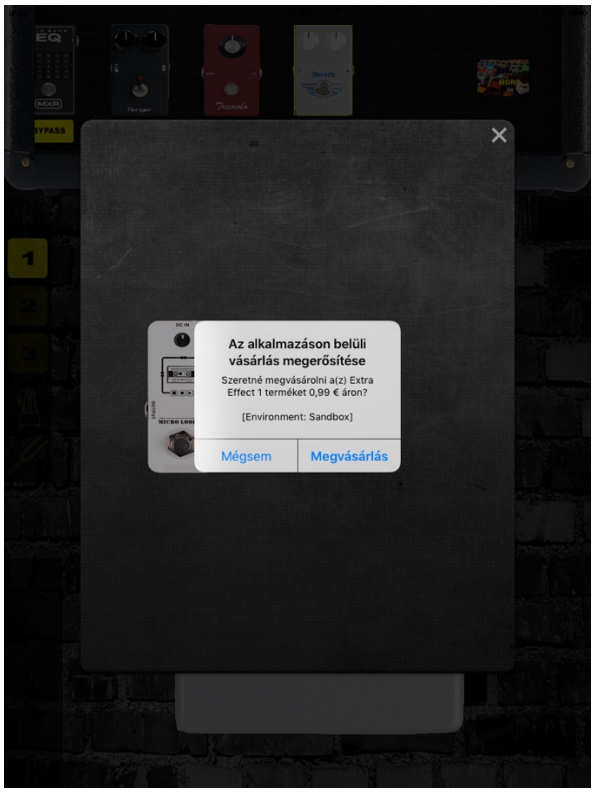
Az ebbe a csoportba tartozó alkalmazásoknál általában egy cég vagy csoport fizet a fejlesztőnek azért, hogy az alkalmazást elkészítse, majd az a felhasználóhoz már ingyenesen jut el.

Ezáltal a mobil alkalmazásokat a marketing stratégiába tudatosan integrálhatjuk, így lehetőség nyílik a meglévő üzleti tevékenység mobilplatformra történő kiterjesztésére, lojalitásfokozó kényelmi szolgáltatások bevezetésére, új potenciális felhasználók elérésére és elkötelezésére, vagy akár a szervezeten belüli belső kommunikáció támogatására. Ilyen például a BKV menetrend vagy a Telekom alkalmazás. [22]

6.1.3 In-App purchasing

Gyakori megoldás, hogy a fejlesztők az applikáción belül plusz tartalmakat pénzért árusítanak. Ez olyan, mintha az App Store-t az alkalmazásba építenénk és egy saját boltot hoznánk létre benne. Ezt a megoldást alkalmaztam én is, így a következőkben ennek a megvalósításáról fogok írni.[22]

6.2 Az alkalmazásom üzleti modellje

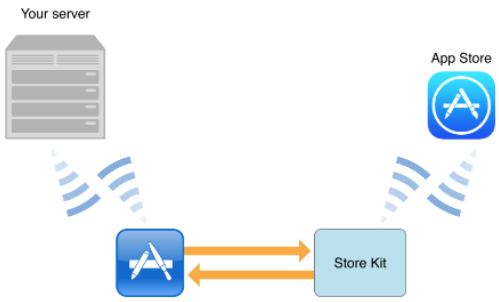


Ábra 6-1 : In-App Purchase

Az alkalmazásom megnyitásakor hat effekt érhető el. Ha a felső menü sávban megnyomjuk a more gombot, akkor egy olyan ablak nyílik meg, amin lehetőségünk van a hetedik effekt megvásárlására. Ehhez a következőkben bemutatásra kerülő beállításokat kellett megtennem.

6.2.1 Előkészületek

Első lépésként engedélyezni kellett a projekt beállítások között az In-App Purchase lehetőséget. Ekkor a fejlesztő környezet automatikusan linkeli a StoreKit framework-öt a projekthez. A StoreKitben találhatóak meg azok a függvények, amelyekkel az alkalmazáson belüli vásárlás lehetséges.

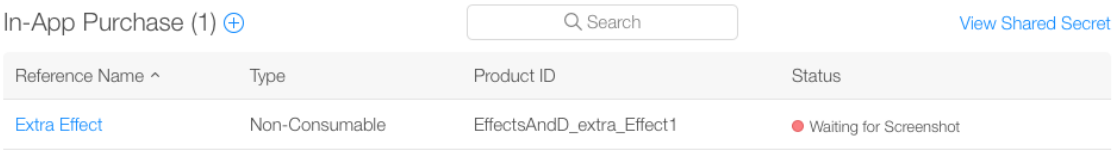


Ábra 6-2 In-App Purchase [23]

A fejlesztői fiókban is több beállítás szükséges. Először is a fizetős alkalmazásokra vonatkozó szerződéseket kell megkötni. Ezután létre kell hozni az alkalmazásnak egy alkalmazás azonosítót (App ID-t) , amiben az In-App Purchase-t engedélyezni kell.

Ezt követően be kell lépni a fejlesztői fiók iTunes Connect részébe, ahol új alkalmazásokat lehet létrehozni, és azokat kezelni is. Itt adhatjuk meg a hozzá tartozó leírásokat és végül ide lehet feltölteni az Xcode-ban megírt alkalmazás binárisát.

Az alkalmazás létrehozása után a „features” menü In-App Purchase részében egy új terméket adtam hozzá a következő adatokkal.

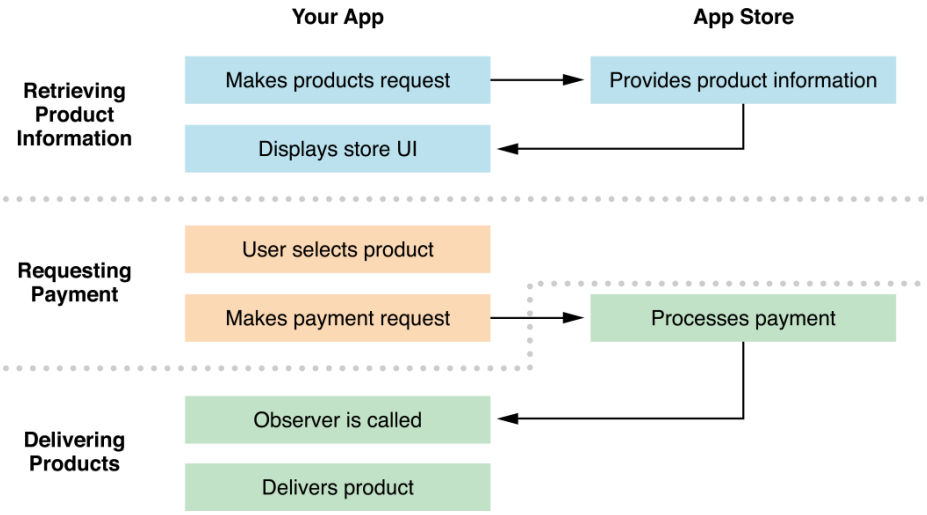


Látható, hogy az Extra Effectem típusa Non-Consumable, ami azt jelenti, hogy a felhasználónak elég egyszer megvásárolni a plusz funkciót, és az mindig elérhető lesz a telepített alkalmazásában. Az árnak egy szintet lehet megadni, ami országonként változó. Én ennek első kategóriát állítottam, ami azt jelenti, hogy Magyarországon 0.99 euróért lehet majd megvásárolni a plusz tartalmat.

Annak érdekében, hogy ne kelljen megvásárolni mindig az extra terméket, a teszteléshez egy nagyon fontos lépés egy Sandbox Tester felhasználó létrehozása a fejlesztői oldalon. Ehhez akár nemlétező email címmel is tudunk egy felhasználónév és jelszó párost regisztrálni, aminek segítségével szimulálhatjuk a vásárlást. [24]

6.2.2 Megvalósítás

Az előre ugró ablakot egy third party library felhasználásával valósítottam meg. Ezt úgy tettem meg, hogy egyszerűen a projektembe húztam azokat a letöltött swift fájlokat, amelyekben az animációk már előre meg voltak írva. Ezeknek az előre megírt függvényeknek a felhasználásával egy esztétikus felületet kaptam. Miután a nézetem megjelenik, a következő műveleteket kell végrehajtanom a vásárláshoz:



Ábra 6-3 : In-App Purchase lépései [25]

6.2.2.1 Termék információk lekérése

A 6-3 ábrán látható, hogy első lépésként egy kérést kell indítani az App Store felé, amitől válaszként a megvásárolható termékek listáját kapjuk vissza. Amíg a válaszra várakozni kell, addig minden esetben jelezni kell ezt a felhasználó felé. Ha ezt nem tenném meg, akkor az alkalmazásom nem kerülhetne fel az App Store-ba. Erre a célra egy Activity Indicator nézetet helyeztem el a képernyő közepére, amely egy forgó animációt jelenít meg addig, amíg az adatok megérkeznek.

6.2.2.2 Termék kiválasztása

Ha az App Store válasza megérkezett, akkor már a felhasználónak lehetősége van a vásárlásra. A kiválasztott effekt gombjának megérintése után a következő kóddal indítható el a vásárlás.

```
@IBAction func buyEffect1Button(sender: AnyObject) {
  let payment = SKPayment(product: productsArray[0])
  SKPaymentQueue.defaultQueue().addPayment(payment)
}
```

Ezután a szokásos dialógus indul el, amiben meg kell adnia a felhasználónak az Apple Id felhasználónevét és a jelszavát, majd meg kell erősíteni a vásárlást.

6.2.2.3 Termék aktiválása

Az App Store ezután küld egy visszaigazolást a vásárlásról. Az én feladatom az, hogy a fizetés után a vett funkciókat megnyissam. Ezt úgy oldottam meg, hogy kezdetben a hetedik effektet kapcsoló gomb el van rejtve, majd a vásárlás után az alábbi módon megjelenítem.

```
self.Effect7.hidden = false
```

7 Tesztelés

7.1 Unit-Teszt

Lehetőségünk van unit-tesztet, vagy más néven egységtesztet futtatni, amely a metódusokat teszteli. A unit-teszt megvizsgálja, hogy a tényleges visszatérési érték megegyezik-e az elvárttal. Ha igen, sikeres a teszt, egyébként sikertelen. Elvárás, hogy magának a unit-tesztnek ne legyen mellékhatása.

Az Xcode projekt létrehozásánál már jó körülmények generálódnak a teszteléshez, hiszen már létrejön egy példa tesztosztály, és egy külön csoport is a tesztfájloknak. A Unit-teszteléshez XCTest framework-öt kell használni, és a tesztosztályoknak, amelyeket létrehozunk, az XCTestCasenből kell leszármazniuk. [26][27]

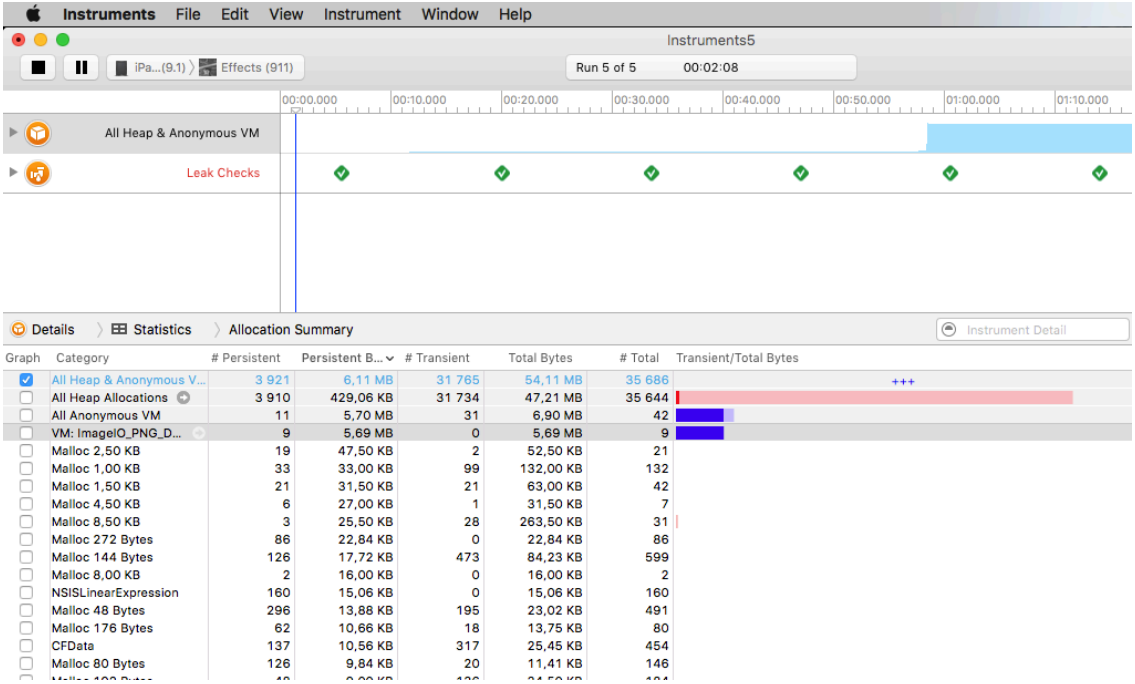
Ezt a tesztelési módszert a programomnál nem tudtam alkalmazni, mert olyan függvényeket használtam, amelyeknek nincs visszatérési értékük.

7.2 Instruments

Ha úgy érezzük, hogy az alkalmazás már elkészült, akkor következhet az Instruments. Ennek az Xcode fejlesztői környezetbe beépített modulnak a segítségével teljesítmény, memória és egyéb más téren vizsgálhatjuk az alkalmazást, hogy az esetleges hibákat kiszűrjük, vagy csak a működést még jobban optimalizáljuk.

A programomban többször alkalmazok alacsony szintű programozást, aminél a memória foglalást és felszabadítást nekem kell elvégeznem. Ez növeli annak az esélyét, hogy a figyelmetlenségekből adódóan a memóriahasználat folyamatosan növekedjen, tehát memóriaszivárgás jelenjen meg a programban. Ha nem kezelem ezt a hibát, akkor előfordulhat, hogy a program magától kilép, mert az iOS operációs rendszer túlzott memória használati szint felett a programot automatikusan leállásra kényszeríti. Emiatt a programom teszteléséhez a leaks eszközt tartottam a legfontosabbnak, amely a memória használatot figyeli és jelez, ha hibát észlel futás közben.

Elindítottam egy tesztet, és közben elkezdtem használni a programomat. A 7-1 ábrán látható eredményt kaptam.



Ábra 7-1 : Instruments

Memóriaszivárgást nem jelzett az eszköz. A legtöbb memóriát a minták tárolására létrehozott bufferek, és a grafikai elemek használnak. Az idődiagramon az ugrást kicsibel 01:00:00 előtt egy olyan effekt bekapcsolása okozta, amely bufferben tárolt mintákkal dolgozik. [28]

7.3 Felhasználói élmény

A felhasználói élmény több komponensből tevődik össze. Annak a mutatója, hogy milyen érzelmeket vált ki az alkalmazás használata az emberekben.

Az egyik, hogy a program könnyen kezelhető legyen. A másik a grafikai megjelenítés, hiszen az is nagyban befolyásolja az alkalmazás sikereséget.

Az én esetemben, ezek mellett lényeges szempont még a gitárosoknak is vonzó hangzás. Több zenész ismerősömnek is megmutattam az alkalmazásomat, és összegyűjtöttem a visszajelzésüket. Ez segít nekem abban, hogy miben tudnék javítani az alkalmazásomon az App Store-ba feltöltés előtt, és majd a következő verziókban.

7.4 Testflight

A programomat a saját iPad mini készülékemen futtattam. Szerettem volna más, nem saját eszközökön is kipróbálni, ezért ehhez az Apple által fejlesztett tesztprogramot, a TestFlight-ot használatam.

Ehhez annak, akinek elküldöm a programot teszteléshez, elsőként telepítenie kell az eszközére a TestFlight programot. Ezt az App Store-ból ingyenesen megteheti.

A küldés úgy történik, hogy először egy fordított archivált verziót kell az Xcode-on keresztül a fejlesztői oldalra feltölteni. Ezután az iTunes Connectbe bejelentkezve, az alkalmazás kezelésénél megjelenik ez a verzió, amit elküldhetek béta review-ra. Ezután körülbelül egy nap alatt átvizsgálják az alkalmazást. Ha ez is sikeresen zárult, akkor már meg is hívhatok bárkit emailben, hogy a feltelepített TestFlight segítségével telepítse, majd 60 napig használhassa az alkalmazásomat a saját eszközén.

Jelenleg a tesztelés fázisában tartok, és várom a visszajelzéseket, hogy tapasztaltak-e leállásokat vagy hibás működést. Ha rendben zajlik minden, akkor már el tudom küldeni a szükséges adatok kitöltése után az alkalmazásomat a fő review-ra, ami már körülbelül egy nyolc napos folyamat. Itt már az Apple egy dolgozója próbálja ki az alkalmazásomat, és ha működése nem összeegyeztethető az App Store szabályaival, akkor jelzi nekem azt.

8 Értékelés és továbbfejlesztési lehetőségek

Az alkalmazásom fejlesztésével sok új ismeretet szereztem, mert több egymástól különböző témába is bele kellett mélyednem. Az iOS fejlesztés számos nehézségével is találkoztam, amelyek eddig még nem kerültek elő a korábbi fejlesztéseim során. Igazi kihívást jelentett számomra a grafikai tervezés és megvalósítás, mint például képek szerkesztése, a felbontások kezelése a különböző eszközök támogatására. A saját üzleti modellem kidolgozásával a mobilalkalmazások piacára is nagyobb rálátást kaptam. Mindazonáltal a legjelentősebb kihívás a hangfeldolgozás alapjaival és a különböző effektek működésével való megismerkedés volt.

A programom ugyan még nem készült el teljesen, de mégis úgy érzem, hogy ez nagyon jó alapja lehet egy későbbi továbbfejlesztett verziónak, amely már a zenészek által is kedvelt multieffekt, iOS platformra. A következő pontokban összefoglaltam azt is, hogy milyen fejlesztési irányok és célok fogalmazódtak még meg bennem.

8.1 További funkciók és eszközök támogatása

Egyrészt szeretném továbbfejleszteni a már meglévő effekteimet. Több paramétert hoznék létre a vezérlésükhöz, ezáltal a kezelésüket és hangzásukat is még jobban közelíthetném az eredeti pedálokhoz. Másrészt a felhasználói felületet is úgy alakítottam ki, hogy lehetőség legyen még további effektek hozzáadására is, illetve hogy két új funkciót, a hangolót és a metronómot is bele tudjam később ágyazni.

Felmerülhet még az a probléma is, hogy ha a gitáros éppen játszik a hangszerén, akkor a keze nem szabad, így nem tudja az alkalmazásomat vezérelni. Egy jó megoldás lehet erre a problémára az iRig által gyártott, bluetooth technológiával kommunikáló pedál, melyet az alkalmazásba lehetne építeni. Ilyen például a következő képen látható Rig BlueBoard eszköz.

Ábra 8-1 : iRig BlueBoard [30]

8.2 Univerzális effektek

Az OSX operációs rendszeren már korábban megjelent az Audio Unit Extension rendszer. Ez lehetővé teszi azt, hogy egy host alkalmazáshoz külső forrásból új zenei eszközöket vagy effekteket hívhassunk meg, azaz plugin-ként az alkalmazásba építhessük őket.

Az operációs rendszer sajátosságaiból adódóan erre eddig még nem volt lehetőség az iOS rendszeren. Ahhoz, hogy ez a modell működhessen, a host alkalmazásnak és a letöltött kiegészítőnek is egy időben kell futnia az eszközön, amit az eddigi iOS verziók nem engedtek. Az iOS 9-cel jött a nagy változás. Ettől a verziótól már multitasking valósulhat meg, tehát nem kell átadni az egymással kommunikáló alkalmazásoknak egymás között a futási jogot, hanem egyszerre párhuzamosan futhatnak.

Ez jelenleg a nagyobb teljesítményű iPadeken lehetséges, ahol ez nem csökkenti a futási sebességet, így nem rontja a felhasználói élményt. Ezeken az eszközökön így elméletileg lehetséges, hogy az App Store-ból külön effekteket töltsünk le egy host alkalmazáshoz. Gyakorlatban ez még nem valósult meg, de a hírek szerint erre már nem kell sokat várni.

Nagy előrelépés lehetne az én programomnál, ha tudna támogatni más fejlesztők által megírt, App Store-ból letölthető effekteket is. Ezáltal a feladatok megoszlanának, és a fejlesztésem felgyorsulhatna, mivel elég lenne kezdetben egy közkedvelt host alkalmazást fejlesztenem.[29]

8.3 Amivel több lehetek

Az alkalmazásom jelenleg nem számít forradalmi újdonságnak a zenészeknek készült iOS alkalmazások között. Már nagy gyártók is készítettek hasonló programokat. Ilyen például az iRig kiegészítőket gyártó IK Multimedia AmpliTube alkalmazása is.

Ábra 8-2 : iRid AmpliTube [31]

Emiatt egy új koncepció megvalósítására is törekszem, amivel több lehetek és kitűnhetek alkalmazásommal az eddig elkészültek közül. Ennek alapja az, hogy a programom egy közösségi alkalmazás is legyen, amely összeköti a zenészeket. Lehetőség lenne a programban sávok vagy szólamok felvételére, és azok továbbküldésére. A címzett újabb sávokat rögzíthet a meglévőkhöz. Így a nem egy térben jelenlévő zenészeknek is lehetősége lenne közös produkció létrehozására.

Irodalomjegyzék

[1] Tim Bajarin: 6 Reasons Apple Is So Successful,
http://techland.time.com/2012/05/07/six-reasons-why-apple-is-successful/
(2015.09)
kép: http://www.hdwallpapers.in/walls/think_different_apple-wide.jpg (2015.09)

[2] iOS fejlődése
https://en.wikipedia.org/wiki/IOS (2015.09)

[3] iOS eszközök képek
http://www.technokrata.hu/uploads/2015/08/apple-tv.jpg (2015.09)
https://devimages.apple.com.edgekey.net/watchos/submit/images/hero.png
(2015.09)
http://cdn.osxdaily.com/wp-content/uploads/2011/11/iphone-and-ipad-ipod.jpg
(2015.09)

[4] iOS rétegek
https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/
iPhoneOSTechOverview/Introduction/Introduction.html (2015.09)

[5] Kelényi Imre: iOS alapú szoftverfejlesztés diasor
https://www.aut.bme.hu/Course/ios (2015.09)

[6] iOS Simulator
https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/iOS_Sim
ulator_Guide/Introduction/Introduction.html (2015.09)

[7] Objective-C és Swift egy projektben:
https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Building
CocoaApps/MixandMatch.html (2015.09)

[8] A hang fizikai jellemzői:
https://hu.scribd.com/doc/96678267/A-hang-tulajdonsagai-es-a-
digitalizalasa#scribd (2015.10)

[9] Huszák Árpád: Médiakommunikáció diasor (2015.10)

[10] Digitális jelfeldolgozá
https://hu.wikipedia.org/wiki/Digitális_jelfeldolgozás (2015.10)

[11] Effekt pedálok:
http://blog.reddogmusic.co.uk/2014/02/28/separate-pedals-or-multi-effects/
(2015.10)

[12] Irig kép
http://www.mannysonlinemusicstore.com.au/wp-
content/uploads/2014/09/ipad_front_irig_amplitube2.5.jpg (2015.10)

[13] Audio feldolgozás képek
http://nagano.monalisa-au.org/wp-content/uploads/2009/08/IO_unit.jpg (2015.10)
http://i.stack.imgur.com/Bnrbk.png (2015.10)

[14] Stefan Popp : Capture iPhone Microphone
http://www.stefanpopp.de/capture-iphone-microphone/ (2015.10)

[15] Márki Ferenc : Digitális hangfeldolgozás
http://acoustics.hit.bme.hu/documents/58digitalis_hangfeldolgozas_v2.0.pdf (2015.10)

[16] Gitáreffekt kézikönyv
http://www.instrumentweb.hu/gitareffekt_kezikonyv (2015.10)

[17] FFT
http://morse.inf.unideb.hu/valseg/gybitt/02/ch08s05.html (2015.10)

[18] Apple: Using Fourier Transform
https://developer.apple.com/library/ios/documentation/Performance/Conceptual/vDSP_Programming_Guide/UsingFourierTransforms/UsingFourierTransforms.html#//apple_ref/doc/uid/TP40005147-CH3-16195 (2015.10)

[19] iOS Fourier Transform
https://mikeash.com/pyblog/friday-qa-2012-10-26-fourier-transforms-and-ffts.html (2015.10)

[20] Néhány gondolat a modulációs effektekről
http://soundhead.hu/portal/irasok/technika/nehany-gondolat-a-modulacios-effektekrol.html (2015.10)

[21] Hisztogram kép
http://karagioza.com/wp-content/uploads/2014/03/bktuastep4allaboutrev2.gif (2015.10)

[22] Üzleti modellek
https://digitalguideblog.wordpress.com/2014/05/27/48/ (2015.10)

[23] About In-App Purchase
https://developer.apple.com/library/ios/documentation/NetworkingInternet/Conceptual/StoreKitGuide/Introduction.html (2015.10)

[24] A Beginner’s Guide to In-App Purchase Programming
http://www.appcoda.com/in-app-purchase-tutorial/ (2015.10)

[25] In-App Purchase steps
http://www.brianjcoleman.com/tutorial-in-app-purchases-iap-in-swift/#prettyPhoto (2015.11)

[26] Unit Test
http://code.tutsplus.com/tutorials/introduction-to-testing-on-ios--cms-22394 (2015.11.)

[27] Egységteszt
http://www.inf.unideb.hu/kmitt/konvkmitt/szoftverteszteles/book.xml.html (2015.11.)

[28] Ray Wenderlich : Instrments Tutorial
http://www.raywenderlich.com/97886/instruments-tutorial-with-swift-getting-started (2015.11.)

[29] Audio Unit Extension
http://www.midiflow.com/blog/audio-unit-extensions.html (2015.11)

[30] iRig BlueBoard
http://cdn.cultofmac.com/wp-content/uploads/2013/10/irig-blueboard-1.jpg (2015.11.)

[31] AmpliTube
http://www.iclarified.com/images/news/10765/37020/37020.jpg (2015.11.)