HTW Berlin (University of Applied Sciences) WS 2010/2011
Media and Computing (Master)
Faculty of Business Sciences II

Paper

Analysis of audio synthesis possibilities on mobile devices using the Apple iPhone and iPad

Markus Konrad

March 26, 2011
Supervising tutor:
Prof. Dr. Klaus Jung

Contents

1. Introduction **2**
- 1.1. Fundamentals . . . 2
- 1.2. Overview about available audio synthesis libraries . . . 2
- 1.3. Methodology . . . 3

2. Analysed Software-Libriaries **4**
- 2.1. Core Audio . . . 4
  - 2.1.1. Core Audio concepts . . . 4
  - 2.1.2. Core Audio in practice . . . 5
  - 2.1.3. Results . . . 10
- 2.2. LibPd . . . 11
  - 2.2.1. LibPd concepts . . . 12
  - 2.2.2. LibPd in practice . . . 14
  - 2.2.3. Results . . . 20
- 2.3. Other libraries . . . 22
  - 2.3.1. MoMu / Synthesis ToolKit . . . 22
  - 2.3.2. CocosDenshion . . . 23

3. Conclusion **24**

4. Appendix **25**

A. References **25**

B. Example projects **27**
- B.1. iPhoneAudioSimple.zip . . . 27
- B.2. iPhoneAudioBending.zip . . . 27
- B.3. iPhoneAudioBendingMultiwaves.zip . . . 27
- B.4. PdAudioSimple.zip . . . 27
- B.5. PdAudioBending.zip . . . 27
- B.6. PdSoundEngine.zip . . . 27
- B.7. MoMuTest.zip . . . 28

1. Introduction

This paper covers the topic of real-time audio synthesis on mobile Apple devices like the iPhone or the iPad (*iOS devices*). Such mobile multi-touch devices introduced an interesting new kind of user interaction and user experience. Applying them to the music context, can create an exciting new world of musical instruments.

Unfortunately, only very little information exists until now (March 2011) about audio synthesis on iOS devices. There is only one book that will cover this topic in a few chapters and this book will not be released before August 2011 [KC11]. Therefore I hope that this paper brings some light into the darkness. It analyses existing techniques and libraries for these devices and describes the advantages and disadvantages of the chosen libraries. Short examples of how to use them will be given. Two of the introduced libraries, namely *LibPd* and *Core Audio*, will be covered in more detail.

The paper concludes with an overall estimation about the audio synthesis possibilities of iOS devices. The appendix consists of some example applications that implement the mentioned techniques.

1.1. Fundamentals

This paper assumes, that the reader is familiar with topics like programming and digital signal processing (*DSP*) and knows basic terms and definitions of both fields. An understanding of programming languages like C, C++ and Objective-C is recommended but not necessary, because the source code excerpts, where given, are always documented and described in the paragraphs next to them. For the section about *LibPd*, a basic understanding of visual data flow languages like *Pure Data* or *Max/MSP* can be helpful, but again is not necessary. Books like *Designing Sound* [Far10] or *loadbang* [Kre09a] provide both an overview about DSP and also give practical examples using Pure Data, whereas for example *The Computer Music Tutorial* [Roa96] is more theoretical but offers profound information about DSP in general.

1.2. Overview about available audio synthesis libraries

As already stated in the introduction of this document, there is few information about audio synthesis on iOS devices and this does also apply to the amount of available software libraries for this field of application. Nevertheless a few interesting libraries could be found for this topic and table 1 gives an overview about them.

As already said, Core Audio and LibPd will be covered in more detail. The first has been chosen because it is Apple's standard interface for everything concerning audio on

Library	**Approach/Features**
Core Audio	Low-level API. Allows to generate audio by writing samples into an audio buffer.
LibPd	Uses *PureData patches* that describe control and signal data flow for audio synthesis. Writes audio samples Core Audio's buffers.
MoMu/STK	Provides a collection of audio synthesis algorithms using the *Synthesis ToolKit*. Writes audio samples into Core Audio's buffers.
CocosDenshion	No real audio synthesis functionality, but provides easy audio file sampling. Pitch and gain of the samples can be controlled.

Table 1: Available audio synthesis libraries on the iOS platform

their mobile and desktop devices. Although it is not very well documented and does not provide many features, it is important because it is *the* entry point for audio synthesis on iOS devices. Every other library uses Core Audio's API for audio rendering and so does LibPd, which is a very interesting library because it allows to embed so called "patches" made with the visual programming language *PureData* into iOS applications. This is a more high-level approach in comparison to Core Audio, because it allows the developer to design control and signal data flow with a visual tool. Furthermore, a lot of signal processing algorithms are provided in PureData's extensive library of functions or "objects".

Whereas the first two libraries have been compared in more detail, including example applications and CPU/memory load profiling, for the other two libraries, *CocosDenshion* and *MoMu/STK*, only basic research has been done in section 2.3. It turned out that CocosDenshion does not provide audio synthesis functionality but still is useful for audio sample playback. MoMu/STK is a new and interesting mobile variant of the widely known *Synthesis ToolKit* from Stanford University, but unfortunately appeared a bit too late in the research process of this paper to cover it in more detail.

1.3. Methodology

To compare existing software libraries for audio synthesis on iOS devices, each of them has been analyzed in terms of features, documentation, community support and other available information resources. Furthermore, the performance of Core Audio and LibPd has been tested using the profiling tool *Instruments* on an iPad 1. For these test runs, comparable applications have been created using available techniques from the two libraries. Conclusively, a step sequencer application has been created as some kind of "field test" for LibPd.

2. Analysed Software-Libriaries

2.1. Core Audio

As stated in Apple's Developer Documentation, "Core Audio capabilities include recording, playback, sound effects, positioning, format conversion, and file stream parsing" [AI11d]. Audio synthesis and other advanced functions are reserved to Mac OSX versions of the library. This means it provides only low-level access to audio rendering in case of iOS applications. Core Audio (*CA*) sits on top of the Hardware Abstraction Layer (*HAL*) and thus receives/sends signals from/to the hardware through HAL [AI11e], so there is no need for the developer to deal with hardware and drivers.

CA provides very sophisticated but complex APIs and the first step to get even simple audio synthesis working, already requires a lot of knowledge. Besides the official Apple documentation, there are fortunately a few articles on the internet that cover this topic and provide a clearer overview than the official documentation (see for example [Bol10] or [Ada09]).

A word about *Core MIDI*, which has been introduced in November 2010 for iOS 4.2: This framework enables iOS devices to send and receive MIDI data, for example via Apple's Camera Connection Kit. It does not include virtual instruments, i.e. its purpose is clearly not the interpretation and audio rendering of MIDI data, but only the transportation. Therefore this library will not be further analyzed in this paper.

2.1.1. Core Audio concepts

Core Audio uses a very sophisticated concept of *Audio Nodes*, that allows developers to create chains of digital signal processors (DSPs), so called *Audio Units.* Audio Units can be low-level things like oscillators or filters, but can also be whole virtual instruments or effects. As already mentioned, under Mac OSX a lot of such Audio Units already exist. iOS, unfortunately, does not have such a luxurious extra equipment and therefore one has to develop DSP units him- or herself.

The first step for using audio in- and output on an iOS device is always to set up an *Audio Session.* Audio Sessions are an iOS specific concept: They let the developer configure the "audio behavior at the application, interapplication, and device levels" [AI11a], which means that one can define what should happen for example when there is a phone call coming in or when iPod audio playback runs in the background. It also provides interrupt handling for these cases and "audio route change handling" which means that the developer can define behaviors for cases like plugging in or unplugging headphones.

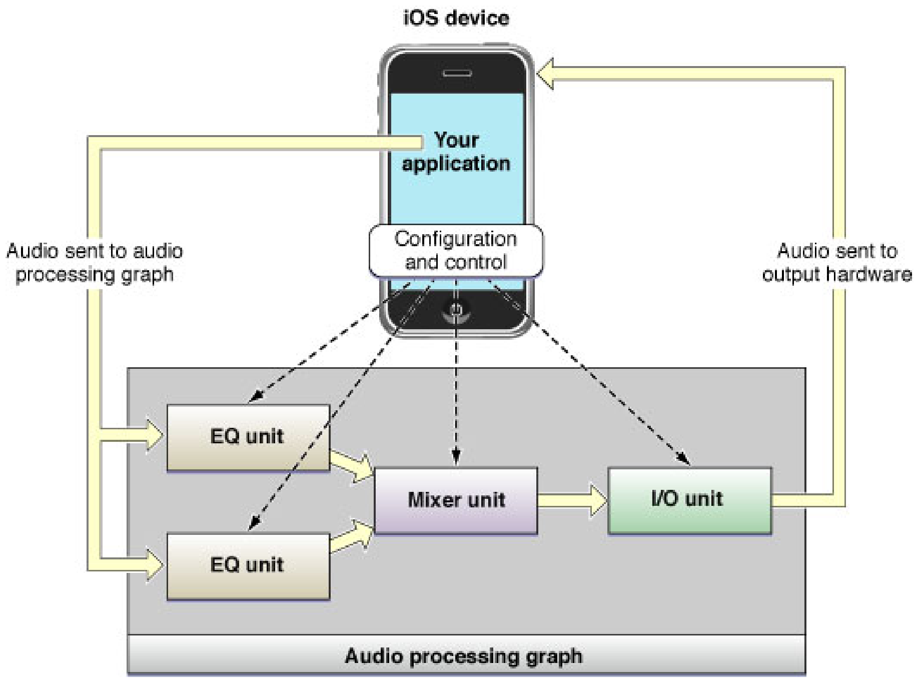
iOS device
Your application
Configuration and control
Audio sent to audio processing graph
Audio sent to output hardware
EQ unit
EQ unit
Mixer unit
I/O unit
Audio processing graph

Figure 1: Example of an Audio Unit graph [AI11c]

The next step is to create the already mentioned chain or graph of DSPs made of Audio Units as for example seen in figure 1. Some basic Audio Units for input/output, mixing, equalizing and voice processing already exist [AI11c], but for audio synthesis they are not really needed. So the approach for audio synthesis via Core Audio on an iOS device is to set up a very basic Audio Unit graph, that can not really be called a "graph", since it only consists of one Audio Unit: The *remote I/O unit*. This Audio Unit receives an input audio buffer and overwrites it with new data for the output for each single audio sample. Each Audio Unit has several properties for configuring its behavior. One of it is the *render callback* property that allows to set a specify callback function that is called within this Audio Unit. As seen in figure 2, it sits between in the in- and the output channel and therefore can be used to analyze the audio buffer from the input and to fill it with newly generated samples for the output.

The next section will describe how to use Audio Sessions, Audio Units and a render callback function in practice.

2.1.2. Core Audio in practice

Creating a simple application

In terms of audio synthesis programming, creating an application that generates a 440Hz sine wave tone is like creating a *Hello World* application. Appendix B.1 provides the

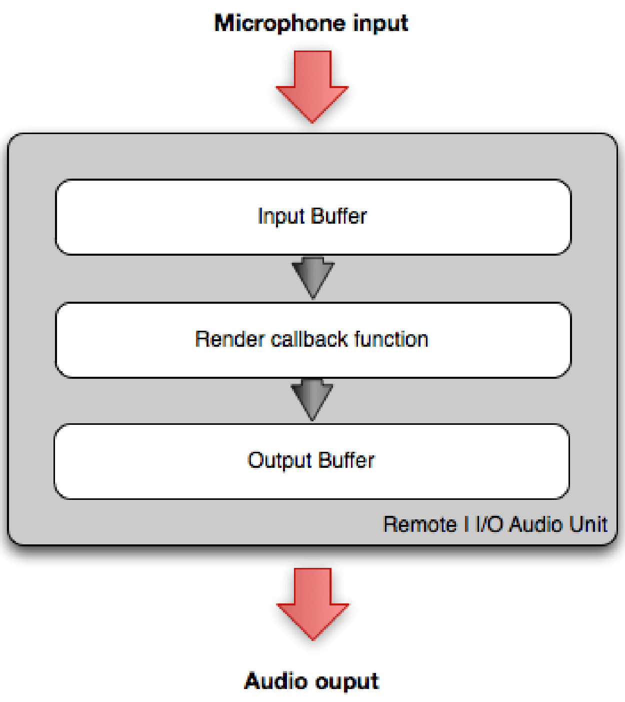

Figure 2: Simple Audio Unit graph for audio synthesis

XCode project example named *iPhoneAudioSimple*. All important implementations concerning audio synthesis are in a class `AudioController`. This section will go through the necessary steps of initializing an Audio Session, setting up and configuring an Audio Unit and writing the audio buffer in a render callback function. It will focus on the most important lines of code, since setting up Audio Sessions and Units already takes up about 200 lines of code. Most of the code comes from the *LibPd* project, described in section 2.2 and has been modified where necessary.

The first step is to create an Audio Session and set an *Audio Session Category*. Such a category expresses the audio role of your application and therefore the main behavior. A table in the Apple documentation [AI11b] gives an overview about these categories. Since audio playback is of primary importance for an audio synthesis application, the categories `AVAudioSessionCategoryPlayback` (for output only) and `-PlayAndRecord` (for in- and output) are the ones to choose:

```
// initialize and set session interrupt listener method
AudioSessionInitialize(NULL, NULL, audioSessionInterruptListener, self);
// set audio session category
AudioSessionSetProperty(kAudioSessionProperty_AudioCategory, sizeof(
    kAudioSessionCategory_PlayAndRecord),
    kAudioSessionCategory_PlayAndRecord);
```

After that, further attributes of the audio session will be requested. Note that they really only will be *requested* and not directly *set*, since Core Audio itself will set the best possible values for the requested ones. The next attribute to be requested is the sample rate. The usual rate is 44.1kHz so that the full spectrum of audible frequencies can be generated:

```
Float64 sampleRate = 44100.0;
AudioSessionSetProperty(kAudioSessionProperty_PreferredHardwareSampleRate,
     sizeof(sampleRate), &sampleRate);
```

The next configuration step is to define the size of the audio buffer. This is a very important step, since it also defines the latency. The smaller the buffer, the smaller the latency but the higher CPU usage because the audio render function is called more often, which creates more overhead. The latency or buffer duration can be calculated in milliseconds as $duration = size/sampleRate * 1000$. This also done for setting the buffer duration property for the Audio Session:

```
Float32 bufferSize = 4096;
Float32 bufferDuration = (bufferSize + 0.5) / kSampleRate;
AudioSessionSetProperty(
    kAudioSessionProperty_PreferredHardwareIOBufferDuration, sizeof(
    bufferDuration), bufferDuration);
```

The last step is to set the session active with `AudioSessionSetActive(true)`. Now all requested properties will be evaluated and the best possible values will be taken.

Initializing the Audio Unit even means setting more parameters. It basically follows the same principles of requesting parameters to be set, for example the number of in- and output channels, the audio format (bits and sample rate per channel) and so on. It is best to have a look at the method `initAudioUnit` in `AudioController`, which includes very good documented code by the LibPd project. The most important lines are the ones where the audio render callback function is set:

```
// register the render callback. This is the function that the audio unit
    calls when it needs audio
AURenderCallbackStruct renderCallbackStruct;
renderCallbackStruct.inputProc = renderCallback;
renderCallbackStruct.inputProcRefCon = self; // pass the AudioController
// set the property
AudioUnitSetProperty(audioUnit, kAudioUnitProperty_SetRenderCallback,
    kAudioUnitScope_Input, outputBus, &renderCallbackStruct, sizeof(
    renderCallbackStruct));
```

All this hard work of setting up the Audio Session and Unit finally leads to the heart of the audio synthesis process: The audio render callback function, which is called every time when new audio data is requested from the hardware. The function header is defined as follows:

```
static OSStatus renderCallback(void *inRefCon, AudioUnitRenderActionFlags
    *ioActionFlags, const AudioTimeStamp *inTimeStamp, UInt32 inBusNumber,
    UInt32 inNumberFrames, AudioBufferList *ioData);
```

The most important parameters are the following:

inRefCon Is a pointer provided in the `renderCallbackStruct` above. In this case it is the AudioController object. It allows access to the public members of AudioController. The variable `audioCtrl` is later used in code snippets for that.

inNumberFrames Contains the size of the buffer for *one* channel.

ioData Contains a pointer to the audio buffer with interleaved audio samples.

Generating a sine wave and writing the data into the buffer is now quite simple. A loop goes through every sample in the `short`-buffer (*short* type because the audio format was set to 16 bit) and generates new values as sinus value multiplied with a maximum amplitude and then converted from `float` to `short` by multiplying the maximum value for a `short`, which is 32767. A new phase for the sine wave is calculated at the end of each loop:

```
// get the buffer
short *shortBuffer = (short *) ioData->mBuffers[0].mData;
// calculate buffer size
int bufLength = inNumberFrames * kNumOuputChannels;

// Loop through the callback buffer, generating samples
for (UInt32 i = 0; i < bufLength; i++) {
  // write a new sample
  shortBuffer[i] = (short)(sin(audioCtrl->sinPhase) * audioCtrl->
      masterVolume * 32767.0f);
  // calculate the phase for the next sample
  audioCtrl->sinPhase += M_PI * audioCtrl->sinFreq / kSampleRate;

  // Reset the phase value to prevent the float from overflowing
  if (audioCtrl->sinPhase >  2.0f * M_PI)
    audioCtrl->sinPhase -= 2.0f * M_PI;
}
```

Creating a more complex application

The second test application, *iPhoneAudioBending* (appendix B.2), consists of an iPhone App that allows to hit a note that will be played with a simple synthesizer with amplitude modulation. By tilting the device, the sound will change: Tilting it to the left and right will change the stereo position of the sound, tilting it up and down will change the frequency of the amplitude modulation.

The most important thing is the modified render callback function. First of all, a simple envelope is generated to increase or decrease the sound of a note when it is hit or respectively released[1]. This is made using timestamps for the "note on" and "note off" times:

```
// Ramp up
if (audioCtrl->noteOnTs > 0) {
  audioCtrl->volEnvelopeAmp = (now-audioCtrl->noteOnTs)/audioCtrl->rampUp;
          // rampUp is a value in milliseconds
  if (audioCtrl->volEnvelopeAmp >= 1.0f) { // full amplitude
    audioCtrl->volEnvelopeAmp = 1.0f;
    audioCtrl->noteOnTs = 0; // stop ramp up now
  }
}
```

The new audio buffer loop is also modified. At first, the volume for the current channel is set for the stereo effect. The buffer is interleaved, which means that every even sample index belongs to the left stereo channel, every odd sample index to the right channel. The new sample is calculated by multiplying the sine wave with another sine wave for amplitude modulation and further multiplying it with amplitudes for channel volume, envelope and master volume:

```
// calculate the phase increase for the next sample
float sinPhaseIncr = M_PI * audioCtrl->sinFreq / kSampleRate;
float ampModPhaseIncr = M_PI * audioCtrl->ampModFreq / kSampleRate;
// calculate volume and float to short conversion outside of the loop
float volumeMultiplications = 32767.0f * audioCtrl->volEnvelopeAmp *
   audioCtrl->masterVolume;
for (UInt32 i = 0; i < bufLength; i++) {
  channelVolume = (i % 2 == 0) ? audioCtrl->volumeLeft : audioCtrl->
     volumeRight;  // choose the volume for the channel
  // write a new sample
  shortBuffer[i] = (short)(sin(audioCtrl->sinPhase) * sin(audioCtrl->
     ampModPhase) * volumeMultiplications * channelVolume);

  // calculate the phase for the next sample
  audioCtrl->sinPhase += sinPhaseIncr;
  audioCtrl->ampModPhase += ampModPhaseIncr;

  // Reset the phase value to prevent the float from overflowing
  if (audioCtrl->sinPhase >  2.0f * M_PI)
    audioCtrl->sinPhase -= 2.0f * M_PI;
  if (audioCtrl->ampModPhase >  2.0f * M_PI)
    audioCtrl->ampModPhase -= 2.0f * M_PI;
}
```

It is clear, that different signal processing algorithms like envelopes or modulation should

[1]For sake of simplicity, no interpolation has been implemented here.

be moved to separate classes or functions, otherwise the `renderCallback()` method will become confusing.

2.1.3. Results

Performance tests have been run on the given two applications using Apple's *Instruments* profiler and an iPad 1. The average memory usage of *iPhoneAudioSimple* was 3.4 MB, whereas the average CPU usage was 11%. *iPhoneAudioBending* took 4.2 MB memory in average and caused 27% CPU load. Of course not all of the CPU load is caused by audio synthesis, since a simple GUI and accelerometer methods are also implemented, but tests showed, that this caused only about 3% CPU usage.

Another small test has been made, where *iPhoneAudioBending* was modified so that multiple detuned sinoidal waves were generated and summed (additive synthesis), which had a heavy impact on the CPU load. It was observed, that there is a linear increase of CPU load by about 4% for each sine wave. The amount of 5 waves caused about 24% CPU load, whereas 20 waves caused 82%. With 22 waves the application began to react very slowly and audible clicks and flaws occurred. 25 waves caused the audio system to give up and not produce any sound. An optimized version of the render callback loop was created that used a sine wave table with 256 precalculated sine values, instead of calling the `sin()` function in each loop and calculating the sine values directly.[2] This application, included in appendix B.3, used less than the half of the CPU load that the unoptimized version needed. Therefore even with 50 sine waves being rendered, the CPU load was at about 75%.

As stated at the beginning of this paper, not only performance is important for using a specific library in a software project, but also documentation, features and community support. It was already shown that there is only basic documentation about Core Audio from Apple and not much information from other web resources. This is basically because in terms of audio synthesis, Core Audio does not really implement any needed helping features for developers besides giving the programmer the opportunity to implement own audio synthesis algorithms in the render callback loop. There are no already implemented algorithms or Audio Units for audio synthesis from Apple for the iOS platform.

Luckily, there is an open-source project *mobilesynth*[3] that includes a lot of often needed algorithms and patterns. Of course it is difficult to get into the not often good documented source-code, but it can be a good starting point for learning to do audio synthesis with Core Audio. The later in section 2.3.1 discussed library *MoMu/STK* also offers such features and is better documented.

[2]Note that this is only a very basic implementation without interpolation or the like.

[3]See project homepage `http://code.google.com/p/mobilesynth/`

2.2. LibPd

One of the most interesting library for audio synthesis on mobile devices so far is *LibPd*, which "wraps Miller Puckette's PureData and turns it into a signal processing library" [Lpm11a].

PureData itself is a "real-time graphical programming environment for audio, video, and graphical processing" [HPH+11]. A graphical editor allows the developer to design data flow processes that can generate audio or video output. These small programs are then called "patches", because the data flow is created by linking or "patching" together functions or "objects". A patch is basically a simple text file (with *.pd* as file name extension) that contains only the information about which objects are connected with each other. See figure 3 as an example of a simple patch.

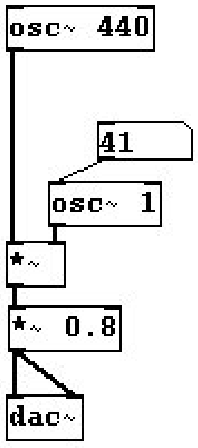

Figure 3: PureData example patch for amplitude modulation

In contrast to the program PureData, which can be used to create and execute such patches, LibPd is a library that can be integrated into iOS and Android applications to load these patches and use them as a digital signal processor for real-time audio rendering. Therefore it is not necessary anymore to implement functions to calculate every audio sample like in Core Audio, because in PureData such functions already exist. They are called "PureData objects" (typically written as `[object]`, e.g. `[osc~]` which is a simple sine wave oscillator that is also used in figure 3) and provide basic signal processing methods. There are sine-wave and saw-tooth oscillators as well as band filters and much more useful functions for audio synthesis. It is also possible to use the microphone input of the device within a PureData patch via LibPd. Apps using LibPd run well on devices using armv7-CPUs (iPad, iPhone 3GS and 4, iPod 3rd and 4th generation) and with some limitations (no microphone input, less performance) also on all armv6-devices [Lpm11b].

One of the most popular examples of LibPd in action are *RjDj*[4] and *Inception The App*[5], which both use the same underlying techniques: LibPd with special "externals", self-written extensions for PureData.[6] The app interactively generates a "personal soundtrack" for the user, while he or she walks with an iPhone through the surroundings. Depending on the environment the sound changes. In nature, for example, the music that is generated live is very calm whereas in the streets of a big town in becomes more hectic. This is done by analyzing the sounds of the environment via microphone input, as well as other sensor data like from the accelerometer of the iPhone. After that, new sounds are generated or existing sound is alienated, depending on this input data and on the "scenes". There are lots of different "scenes", which means different generative pieces of music, some of them done by famous musicians like, for example, AIR [RjD11].

Just like PureData, LibPd is released under BSD license. Some externals like `[expr~]` are under GPL, but do need to be compiled to run most PureData patches. The documentation for LibPd does exist as a Wiki at [Lpm11b], that describes the usage of the only two classes that are needed to set up a project for an iOS device with LibPd. An active online community exists at *noisepages.com*.[7] There is extensive information and a lot of tutorials for PureData on the web (see [HPH+11, Kre09b, Puc06]) and in literature (see [Kre09a, Far10]). Community support and further information can be found on `http://puredata.info`.

2.2.1. LibPd concepts

As visible in figure 4, LibPd sits on top of CoreAudio and generates samples which are used in the `renderCallback`-function (as described in section 2.1.2). It does this by loading a PureData patch, analyzing its contents and calculating the resulting data flow "inside" the patch. There is also a bidirectional communication possible between an application using LibPd and a PureData patch: Messages can be sent to patches to control the behavior of a patch (and thereby, for example, change the audio output) and can also be received from patches for control, feedback and debugging purposes. This will be described in more detail in section 2.2.2.

A key concept of LibPd is that every patch, given that it uses standard PureData-objects, that runs in the PureData program, should also run successfully with LibPd on one of the supported devices without modifications. This means that "input and output of samples occur via `[adc~]` and `[dac~]`, as usual"[8] [Lpm11b]. There are no special LibPd objects to use in the PureData patches.

[4]See `http://rjdj.me/`

[5]See `http://inceptiontheapp.com/`

[6]The externals for RjDj are open-source and can be found on `https://github.com/rjdj/rjlib`

[7]See `http://noisepages.com/groups/pd-everywhere/`

[8]The objects "adc" and "dac" are analog-to-digital, respectively digital-to-analog converters in PureData, which means the first is the microphone input, the second the audio output.

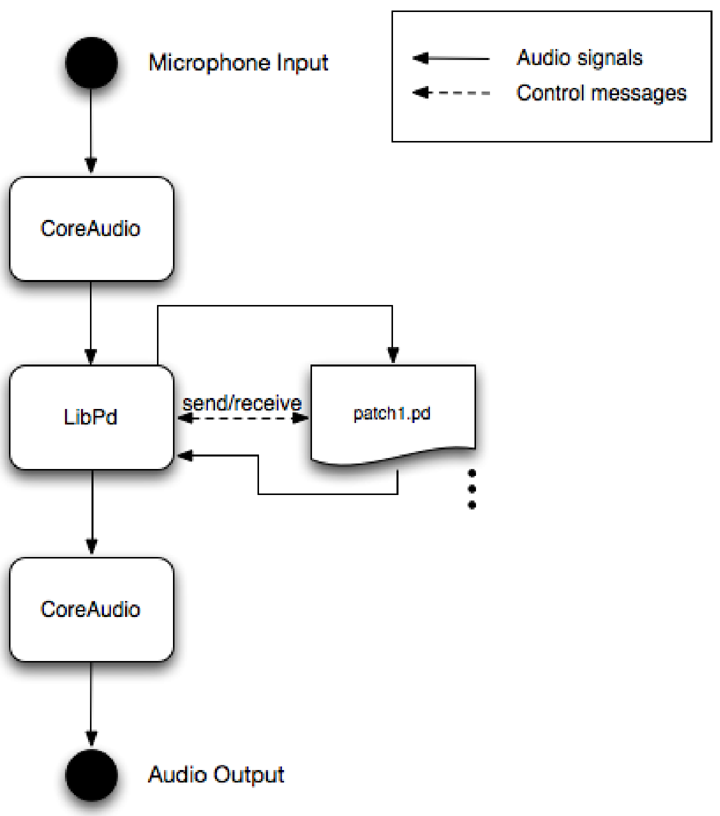

Figure 4: LibPd integration into an iOS app

LibPd consists of a set of functions written in C. Fortunately, there is an Objective-C interface for LibPd, which is very compact and straight forward: There are only two Objective-C classes and one protocol. The `PdAudio` class implements the "glue code" between Core Audio and LibPd, whereas `PdBase` is a class with static functions that are basically wrappers for the `libpd_*`-C-functions. `PdBase` allows to send control messages to PureData-patches. It is also possible to set a delegate object that implements the `PdReceiverDelegate` protocol. This delegate can then receive control messages that are sent by a PureData-patch.

Although PureData comes with a large set of objects that can be combined in patches to complex data flow sceneries, it is sometimes necessary to write self-made primitives, so called "externals" and therefore extend PureData with new objects. Such externals can be written in C, using the *PureData externals API*.[9] Writing externals is not an easy task, but enables the developer to implement functions that are very hard to actualize in an efficient way with a PureData patch without externals. With LibPd it is possible to use this compiled C-language externals on iOS devices as a statically linked library. Loading externals dynamically is only possible on Android

[9]See `http://pdstatic.iem.at/externals-HOWTO/` for a comprehensive overview about the API.

devices [Lpm11b].

2.2.2. LibPd in practice

Setting up LibPd

Setting up LibPd to compile is not as straight-forward as using it: After cloning it from the Git repository[10], one has to copy the folder "libpd" to the project directory and include it in the XCode-project. LibPd comes with support for Android platforms and Python. A lot of these files are not needed and even prevent from compiling it successfully. Therefore it is necessary to remove the following folders and files from the folder "libpd" in XCode:

- pure-data/extra
- python
- samples
- libpd_wrapper/z_jni.h & z_jni.c

Furthermore, the *AudioToolbox*-Framework needs to be added to XCode and the following compiler macros must be defined either in the project settings or in the prefix header file:

```
#define PD
#define USA_API_DUMMY
#define HAVE_LIBDL
#define HAVE_UNISTD_H
```

Creating a simple application

A very simple application could just load a PureData-patch and start it. As a small demonstration, appendix B.4 (*PdAudioSimple*) contains such a simple XCode-project with an iPhone App that produces a sine wave at 440Hz. The master volume can be controlled with a slider. All necessary functions are implemented in the view controller of the application, `PdAudioSimpleViewController`. The method `viewDidLoad` contains the initialization process for LibPd. At first, a `PdAudio` object is created with the necessary parameters:

[10]The Git repository of LibPd is located at `git://gitorious.org/pdlib/pd-for-ios.git`.

```
- (void)viewDidLoad {
  // ...
  pdAudio = [[PdAudio alloc] initWithSampleRate:44100.0
                                  andTicksPerBuffer:64
                            andNumberOfInputChannels:0
                           andNumberOfOutputChannels:2];
  // ...
}
```

This means that LibPd will produce sound for 2 output channels (i.e. stereo) with the default sample rate of 44.1kHz. Input channels, i.e. microphone input, will not be used. The `ticksPerBuffer` parameter is important regarding audio rendering performance and latency: The parameter defines how often a “Pd tick” is called per audio rendering call [Lpm11b]. One “Pd tick” produces always 64 audio samples per channel.[11] Setting the `ticksPerBuffer` parameter to a very small number therefore leads to much smaller audio latency, but also increases the number of “Pd tick” calls, which leads to more function-call overhead and eventually to higher CPU and memory usage.

The next step in terms of initialization is to set the PdReceiverDelegate for `PdBase` as described in section 2.2.1. The view controller `PdAudioSimpleViewController` implements the method `receivePrint:` of the `PdReceiverDelegate` protocol and therefore will be informed when a `[print]`-message is sent from within the PureData-patch.

```
- (void)viewDidLoad {
  // ...
  [PdBase setDelegate:self];
  // ...
}
// ...
- (void)receivePrint:(NSString *)message {
    NSLog(@"Print from Pd: %@", message);
}
```

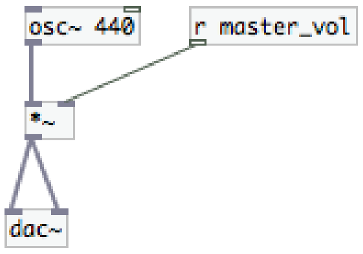

Figure 5: PureData patch “audio_out.pd” for PdAudioSimple

[11]Do not confuse the fixed number of audio samples per “Pd tick” (64 samples) with the ticksPerBuffer parameter, which is in this example also 64.

The last step within `viewDidLoad` is to load the patch and start up the audio rendering. The patch "audio_out.pd" will be loaded, which only contains a simple sine-wave oscillator and a receiver `[r master_vol]`[12] as shown in figure 5.

```
- (void)viewDidLoad {
  // ...
  // open patch located in app bundle
  [PdBase openPatch:[[[NSBundle mainBundle] bundlePath]
      stringByAppendingPathComponent:@"audio_out.pd"]];

  // start audio rendering
  [PdBase computeAudio:YES];
  [pdAudio play];
}
```

Sending messages to the PureData-patch is very straight-forward, as demonstrated by the example of sending the "master volume slider"-value to the receiver `master_vol` in the patch:

```
- (IBAction)masterVolSliderChanged:(id)sender {
    UISlider *slider = (UISlider *)sender;
    [PdBase sendFloat:slider.value toReceiver:@"master_vol"];
}
```

Creating a more complex application

The second test application, which is called *PdAudioBending* and is included in appendix B.5, has the same features as *iPhoneAudioBending* in section 2.1.2 and has been created for comparison.

The PureData-patch that has been created is slightly more complex as shown in figures 6 and 7. The patch can now receive "note"-messages, a simple pair of pitch and velocity arguments just like MIDI-notes. There are also "bend" messages with a bending-value (used when tilting the device). A note with a velocity of 0 acts as a *note off* message and mutes the note. An ADSR-envelope[13] is used to define the loudness of the note over time.

On the Objective-C side, not much has changed: Now a note will be constructed with pitch and velocity values and will be sent to the patch upon `playNote:`-action. There is also a `stopNote:`-action sending a note with a velocity of 0.

[12] "r" is a shortcut for "receive".

[13] ADSR stands for attack, decay, sustain, release that is used to create characteristic amplitudes over time.

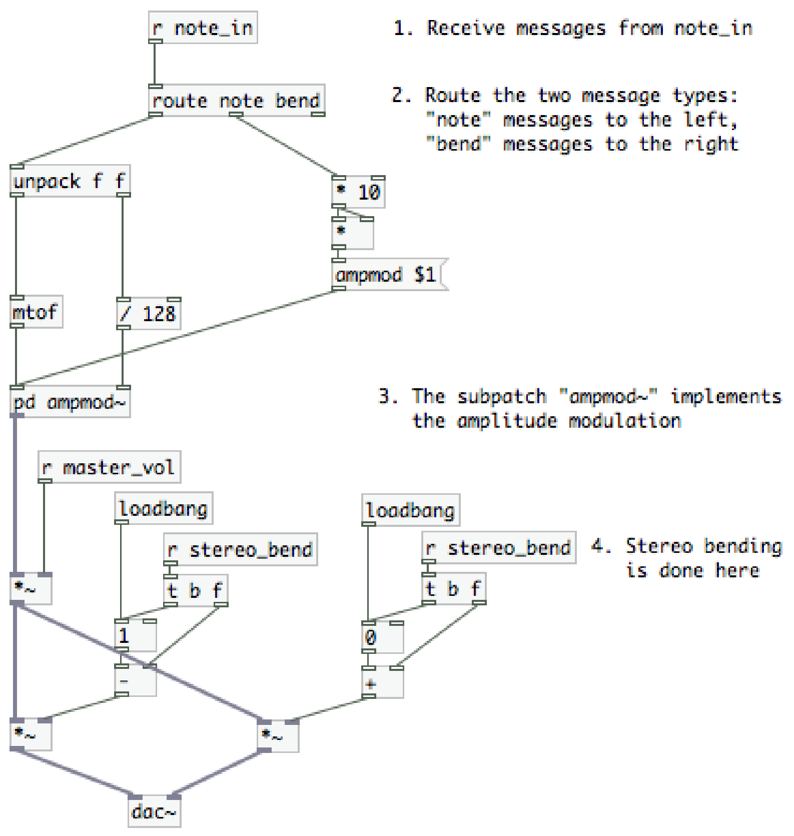
r note_in
1. Receive messages from note_in
route note bend
2. Route the two message types: "note" messages to the left, "bend" messages to the right
unpack f f
* 10
*
ampmod $1
mtof
/ 128
pd ampmod~
3. The subpatch "ampmod~" implements the amplitude modulation
r master_vol
loadbang
loadbang
r stereo_bend
r stereo_bend
4. Stereo bending is done here
t b f
t b f
*~
1
0
-
+
*~
*~
dac~

Figure 6: PureData patch "audio_out.pd" for PdAudioBending

```
- (IBAction)playNote:(id)sender {
    // ...
    // create note arguments: MIDI-note with pitch 64 and velocity of 90
    NSArray *noteData = [NSArray arrayWithObjects:[NSNumber
        numberWithFloat:64.0f], [NSNumber numberWithFloat:90.0f], nil];
    [PdBase sendMessage:@"note" // send a "note" message to "note_in"
            withArguments:noteData
                toReceiver:@"note_in"];
}
```

Sending the bend-values to change the sound upon tilting the device follows the same principles. Apple's `UIAccelerometerDelegate` is used to get informed about how the device is tilted and then the accelerator values are sent to the correspondending receivers in the patch. It would be wise to implement a threshold for value changes, so that only values that have really changed by a specific magnitude are sent to the patch.

```
- (void)accelerometer:(UIAccelerometer *)accelerometer didAccelerate:(
   UIAcceleration *)acceleration {
    // ...
    if (notePlaying) {
        [PdBase sendMessage:@"bend" // send synth bending messages
              withArguments:[NSArray arrayWithObject:[NSNumber
                 numberWithFloat:accelY]]
                 toReceiver:@"note_in"];
    }
    // send stereo bending values
    [PdBase sendFloat:accelX toReceiver:@"stereo_bend"];
}
```

The code is straight-forward and the separation of tasks is clear: The created Objective-C code takes care about user interaction and, if needed, sends messages to the PureData patch which is responsible for audio synthesis.

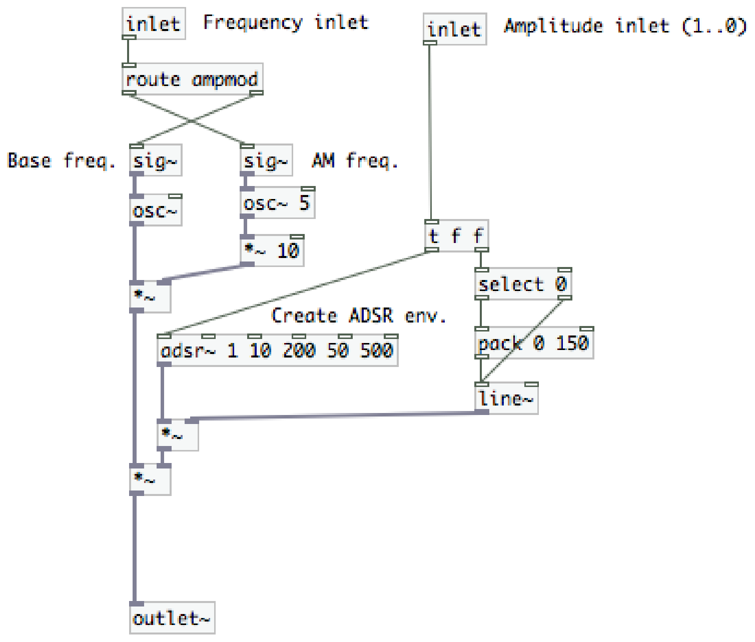

Figure 7: Subpatch "ampmod" for PdAudioBending

Creating a step sequencer application

To test the performance of LibPd, a step-sequencer application has been created and is included as appendix B.6. The GUI of the application allows to place notes on a matrix that can be seen in figure 8. The notes are played from left to right. A pentatonic scale is used to create the pitch of a note on the Y-axis of the matrix. It is possible to create notes on different layers with two different instruments. The ">" and "<" buttons on the top of the screen allow to switch between the layers, the button in the lower left lets the user change the instrument. Of course such an application could have been implemented using audio sample playback, but as already said, this application has only the purpose of testing the performance for real-time audio synthesis with LibPd.

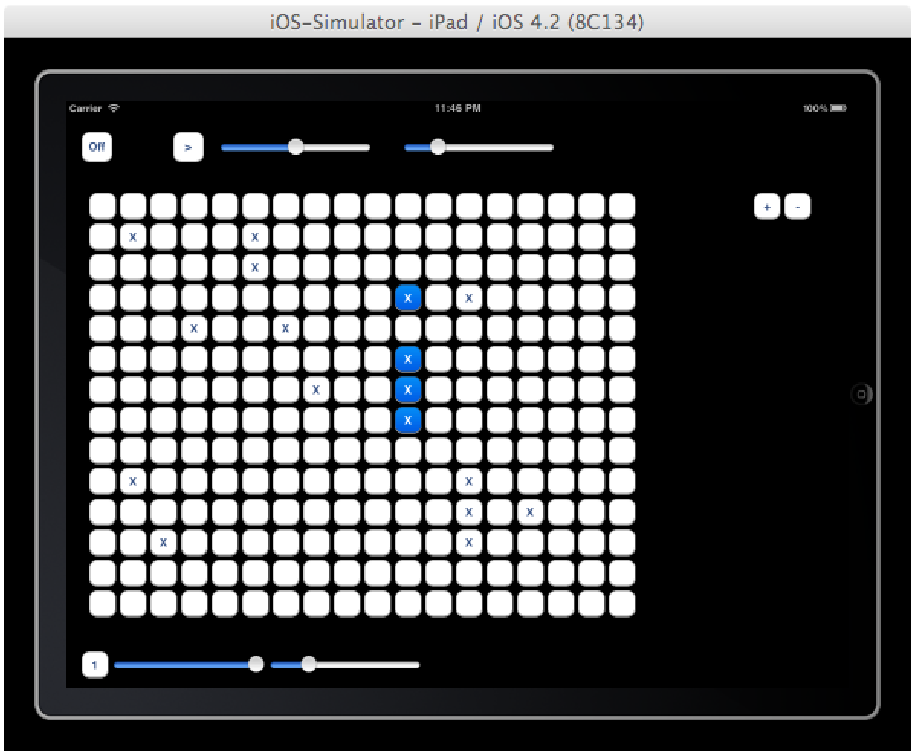

Figure 8: Step-Sequencer GUI for PdSoundEngine

Once again, the Objective-C classes take care about the *Controller*-side of the application, meaning that on this side the user interaction and the sequencer timing functions are implemented. Although it would also be possible to implement the sequencer timing functions in the PureData patch, it was more reasonable to implement them in Objective-C, as this code gets compiled whereas the PureData patch is interpreted at run-time, making it less efficient. As already introduced in the *PdAudioBending* application, MIDI notes consisting of *pitch* and *velocity* are used for communication between the Objective-C code in the iOS application and the PureData patch. This makes it easier to switch from LibPd to another audio rendering backend or even send the MIDI notes via *CoreMIDI* to a real hardware synthesizer.

For this paper it is not so interesting how the Objective-C classes for the sequencer and the user interaction are implemented, but how the PureData patches are constructed. One of the main problems was *polyphony*, meaning that more than one note can be played at one time. There is an object `[poly]` that enables polyphony for PureData, but it is not very easy to use. Fortunately there is an abstraction[14] `[polypoly]` that makes

[14] "Abstractions" are nothing else but extra PureData patches that can be used like a normal object (just like "externals" but not written in C).

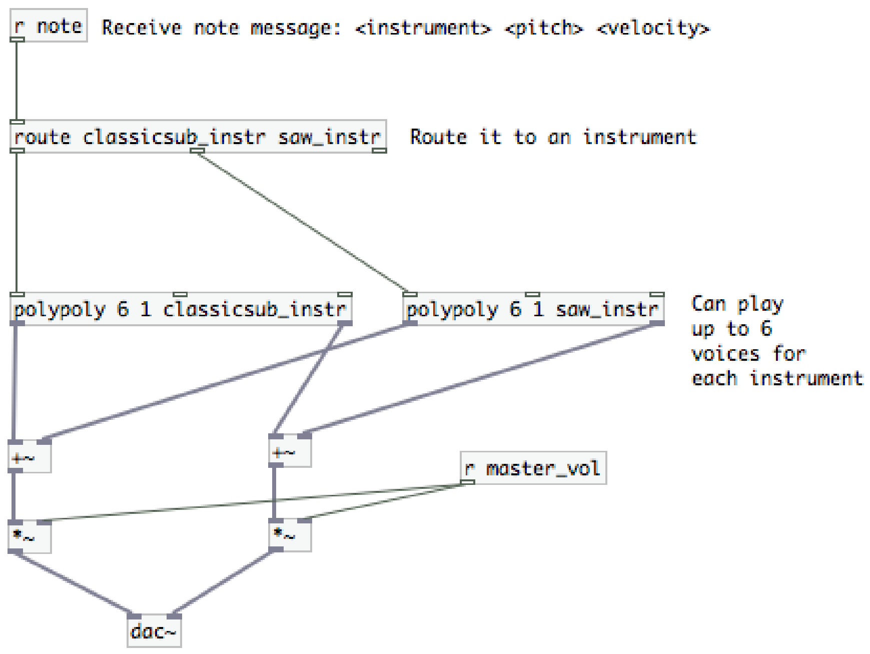
r note
Receive note message: <instrument> <pitch> <velocity>
route classicsub_instr saw_instr
Route it to an instrument
polypoly 6 1 classicsub_instr
polypoly 6 1 saw_instr
Can play up to 6 voices for each instrument
+~
+~
r master_vol
*~
*~
dac~

Figure 9: Patch "poly_seq.pd" for polyphonic sequencer

polyphony much easier to use than with the default [poly] object. Figure 9 shows how [polypoly] is used for two instruments that are implemented in other patches, respectively "classicsub_instr.pd"[15] and "saw_instr.pd". These instruments receive MIDI notes from [polypoly] and produce the audio signals.

2.2.3. Results

For simple applications like the first two, the results are quite positive. The CPU load for *PdAudioSimple* is interestingly a bit lower than in the comparable application that uses Core Audio: only 6%. But another process that seems to be connected with the audio system, "mediaserverd", uses about 5% more than in *iPhoneAudioSimple* and therefore the summed up CPU load is about the same in both applications. The memory usage in PdAudioSimple is with 3.8 MB a bit higher, compared to 3.4 MB using Core Audio without LibPd. The same was experienced with application number two: Here the memory usage was again a bit higher, in particular 4.5 compared to 4.2 MB. The CPU usage was again similar when taking the higher CPU usage of "mediaserverd" in account.

Especially the last application, *PdSoundEngine*, shows the limits of audio synthesis on

[15]The "classicsub" instrument is part of the "pd-starter-kit" package from `http://gitorious.org/pdlib/pd-starter-kit`.

an iOS device: The PureData patch all the time consumes about 60% CPU usage[16] because of the complex synthesizer algorithms used in the patches. In addition to that, audible clicks and flaws occur with many notes activated. This is probably because of the "voice-stealing" that `[polypoly]` does, when there are more instruments playing at one time than voices are available. The high CPU consumption slows down all other operations, for example for drawing the GUI, too. So this limits the complexity of audio synthesis operations in the patches very much and this is also the reason, why only two instruments have been implemented. There were more instruments available, but including them in a patch would increase the CPU usage so much, that the application would be unusable. In addition to that, the "classicsub" instrument patch was modified to reduce its CPU consumption, because in its original version it did not work on the device at all. It was using 16x-resampling, which caused the application to freeze upon startup for about five seconds and eventually did not produce any sound. A quick look with *Instruments* showed that the CPU usage was heavily going up at this moment before eventually it gave up and the audio system was shut down. What is very frustrating, is that there is no way to identify such problems in a patch, which makes debugging very difficult. In PureData, a patch will work perfectly, on the device the audio system will crash and the only way to find out what is going wrong, is to delete more and more objects to reduce its complexity until it works on the device.
In general, debugging a LibPd application is rather hard: If something does not work correctly inside the PureData-patch, it is very hard to determine the problem, because there is no information what went wrong and where. Of course, it is not possible to use debugger-breakpoints inside the patch. Two things can help to narrow down such problems: First and foremost, the patch should be tested stand-alone using the PureData program with the same control messages that might also be sent by the LibPd-App. Secondly, a delegate that implements the `PdReceiverDelegate` protocol should be created. This delegate can implement the `receivePrint:` method for printing debug messages to the console, which come from `[print]`-objects in the PureData-patch.

Another problem that occurred in the last example application is that sometimes breaks in the timing of the notes can be experienced, especially when there are other complex computations going on, for example when a layer is being switched and the 3D flip animation is performed. This will cause the sequencer scheduler to get out of step and send new notes too late. Maybe this problem could be solved using a different scheduling method than using the `NSObject` method `performSelector:afterDelay:`.

Nevertheless LibPd seems to be a good alternative for using Core Audio, because it makes it much easier to create synthesizer instruments and effects, using a wide variety of already available synthesis components. It has some disadvantages, though: Especially the debugging process with switching between PureData and XCode all the time is frustrating. Sometimes patches would work in PureData, whereas on the device they would not produce any sound. The performance is surprisingly good, compared to pure

[16]This can be very clearly seen in Instruments when toggling the *Audio On/Off* button.

Core Audio applications, although a small overhead exists because of sending and receiving messages, interpreting patches and so on. Realtime polyphonic audio rendering with multiple complex synthesizer instruments is hardly possible because the high CPU usage will slow down the whole application. But as seen with tests using *iPhoneAudioBendingMultiwaves* in section 2.1.3, it is possible to reduce CPU load heavily by optimizing calculation algorithms and using, for example, precalculated wave tables instead of calculating sine values in each render loop. If the resolution of such a wave table is high enough, no difference is audible. PureData also provides reading such wave tables using the `[tabosc~]` object.

2.3. Other libraries

2.3.1. MoMu / Synthesis ToolKit

MoMu stands for *Mobile Music Toolkit* and is a "new open-source software development toolkit focusing on musical interaction design for mobile phones". It has been presented on the NIME conference in summer 2010 and was developed in the Center for Computer Research in Music and Acoustics in Stanford University. It is published under a BSD-like license. It does not only provide an API for audio synthesis, but "a unified access to onboard sensors along with utilities for common tasks found in mobile music development" [BHJO10]. These include:

- Full-duplex audio input and output
- Sound synthesis
- Handling of sensor input (accelerometer, location, compass)
- Handling of multitouch
- Networking (via OSC[17])
- OpenGL graphics
- Filters (for audio synthesis, graphics and other)
- Fast Fourier Transform

The authors of the library did not implement all these features anew, but combined several other libraries and developed a unified C++-API for all of them. The most interesting parts for this paper are the sound synthesis features of MoMu, for which the authors added a port of the *Synthesis ToolKit (STK)* for iOS. The STK has been developed in the same department of Stanford University as MoMu and was already released in 1995 but is still developed until now. It offers "an array of unit generators for

[17]OSC stands for Open Sound Control, a content format for messaging between computers, electronic instruments and other multimedia devices [FS09]

filtering, input/output, etc., as well as examples of new and classic synthesis and effects algorithms for research, teaching, performance, and composition purposes" [CS99]. The STK has been slightly modified by the MoMu team to cooperate with Core Audio on iOS devices.

In terms of documentation and community support, the STK library makes a good impression: A comprehensive tutorial covers the basic features. Several demo projects that are included in the source download provide well documented examples. An API documentation offers an overview about the available classes and their methods. Although the library is already more than 15 years old, the mailinglist is still active and a comprehensive archive exists. All of that can be found at the STK homepage at `https://ccrma.stanford.edu/software/stk/`.

With help of the mentioned information resources, it was quite easy to integrate the STK library into a Core Audio project, based on the *iPhoneAudioSimple* project that has been introduced in section 2.1.2. The application is provided in appendix B.7. Profiling tests did not show any noticeable difference in performance compared to LibPd, but more advanced tests should be done.

All in all this library seems to be very promising. It provides a very comprehensive set of features, is well documented and with 15 years of age a very matured library with efficient algorithms. Still, just like LibPd, it cannot work wonders with the limited resources of mobile devices and therefore virtual instruments that are developed using the STK, should of course use as little resources as possible.

2.3.2. CocosDenshion

A very popular framework for games, 2D graphics and interaction is *Cocos2D*[18]. Since games also need music and other audio output, Cocos2D also includes an API for playing sounds: *CocosDenshion*. Since the API only supports sound playback but not synthesis, only a small overview about its features will be given. Nevertheless, CocosDenshion is a very helpful library, because audio sample playback can be much easier implemented, than for example with Core Audio. The library supports sample playback, filters, panning and pitching. By using the pitching capabilities, it is possible to play different notes of an instrument from one audio sample file. Of course, raising the pitch implies shortening the tone and vice versa, because there is no time-stretching mechanism implemented.[19] The CocosDenshion API documentation states that up to 24 samples playing in parallel are supported.

[18]See `http://www.cocos2d-iphone.org/`

[19]See [Ber99] for a comprehensive overview about time stretching and pitch shifting.

3. Conclusion

Research and example applications clearly showed the limits of audio synthesis on an iOS device. Although interesting libraries exist, it is hard to implement well sounding virtual instruments without causing too much CPU and memory load. Existing virtual instrument implementations that come with LibPd or the STK are often not designed to work well with the limited resources of mobile devices. However, it would be very interesting to know, how much audio synthesis applications can benefit from the new improved hardware of the iPad 2 [Big11].

For the developer that aims on implementing audio synthesis on an iPad 1 or the iPhone, several options exist now:

Using Core Audio without other libraries lets the developer maximum power over how the sound is generated and minimum overhead, but requires to write a lot of additional code since there are no audio synthesis functions provided by Apple.

Using LibPd lets the developer design the signal data flow with a visual tool, offers him or her a wide set of existing signal processing functions but can cause problems on the device which are very hard to debug.

Using MoMu/STK does also offer a lot of signal processing functions directly implemented in C++.

Of course it should be always considered, if real-time audio synthesis is really necessary. If the user should be able to interact with every parameter of the sound that is being generated directly and just in time, then it is hard to bypass audio synthesis. But on the other hand it is often sufficient when the user can only change parameters like pitch, volume, stereo balance and maybe add some additional effects. In this case using audio sampling, envelopes and simple filters are adequate and do not cause such a heavy CPU load as generating each audio sample with complex signal processing algorithms. Since the needed functionality for this scenario is included in LibPd and the STK, advanced audio samplers could be built upon one of this libraries.

4. Appendix

A. References

[Ada09] Chris Adamson. An iphone core audio brain dump. Website, April 2009. Available online at http://www.subfurther.com/blog/2009/04/28/an-iphone-core-audio-brain-dump/; visited on March 18th 2011.

[AI11a] Apple Inc. Audio session programming guide: About configuring audio behavior. Website, 2011. Available online at http://developer.apple.com/library/ios/#DOCUMENTATION/Audio/Conceptual/AudioSessionProgrammingGuide/Introduction/Introduction.html; visited on March 18th 2011.

[AI11b] Apple Inc. Audio session programming guide: Audio session categories. Website, 2011. Available online at http://developer.apple.com/library/ios/#documentation/Audio/Conceptual/AudioSessionProgrammingGuide/AudioSessionCategories/AudioSessionCategories.html#/apple_ref/doc/uid/TP40007875-CH4-SW1; visited on March 19th 2011.

[AI11c] Apple Inc. Audio unit hosting guide for ios: About audio unit hosting. Website, 2011. Available online at http://developer.apple.com/library/ios/#documentation/MusicAudio/Conceptual/AudioUnitHostingGuide_iOS/Introduction/Introduction.html; visited on March 18th 2011.

[AI11d] Apple Inc. Core audio overview: Introduction. Website, 2011. Available online at http://developer.apple.com/library/ios/#DOCUMENTATION/MusicAudio/Conceptual/CoreAudioOverview/Introduction/Introduction.html; visited on March 18th 2011.

[AI11e] Apple Inc. Core audio overview: What is core audio? Website, 2011. Available online at http://developer.apple.com/library/ios/#DOCUMENTATION/MusicAudio/Conceptual/CoreAudioOverview/WhatisCoreAudio/WhatisCoreAudio.html#/apple_ref/doc/uid/TP40003577-CH3-SW1; visited on March 18th 2011.

[Ber99] Stephan Bernsee. Time stretching and pitch shifting of audio signals – an overview. Website, August 1999. Available online at http://www.dspdimension.com/admin/time-pitch-overview/; visited on March 13rd 2011.

[BHJO10] Nicholas J. Bryan, Jorge Herrera, and Ge Wang Jieun Oh. Momu: A mobile music toolkit. In *NIME*, Sidney, Australia, 2010. Online at http://www.educ.dab.uts.edu.au/nime/PROCEEDINGS/papers/PaperH1-H4/P174_Bryan.pdf.

[Big11] John Biggs. Apple announces the ipad 2: A5 processor, front and back cameras, available march 11. Website, March 2011. Available online at `http://www.crunchgear.com/2011/03/02/apple-ipad-2-announcement/`; visited on March 24th 2011.

[Bol10] Tim Bolstad. iphone core audio part 1 – getting started. Website, March 2010. Available online at `http://timbolstad.com/2010/03/14/core-audio-getting-started/l`; visited on March 18th 2011.

[CS99] Perry R. Cook and Gary P. Scavone. The synthesis toolkit (stk). In *ICMC*, Beijing, China, 1999. Online at `https://ccrma.stanford.edu/software/stk/papers/stkicmc99.pdf`.

[Far10] Andy Farnell. *Designing Sound*. MIT Press, September 2010.

[FS09] Adrian Freed and Andy Schmeder. Features and future of open sound control version 1.1 for nime. In *NIME*, 04/06/2009 2009. Online at `http://cnmat.berkeley.edu/node/7002`.

[HPH$^+$11] Holzer, Princic, Hyde, Pais, Baker-Smith, Schebella, Steiner, Davison, and Tahiroglu. *Pure Data*. FLOSS Manuals, March 2011. Online at `http://en.flossmanuals.net/pure-data/_booki/pure-data/pure-data.pdf`.

[KC11] Avila Kevin and Adamson Chris. *Core Audio*. Addison-Wesley Professional, 1 edition, 2011.

[Kre09a] Johannes Kreidler. *Loadbang: Programming Electronic Music in Pure Data*. Wolke Verlagsges. Mbh, March 2009.

[Kre09b] Johannes Kreidler. Programming electronic music in pd. Website, January 2009. Available online at `http://www.pd-tutorial.com/english/index.html`; visited on March 23rd 2011.

[Lpm11a] LibPd project members. Libpd - gitorious. Website, 2011. Available online at `http://gitorious.org/pdlib`; visited on March 3rd 2011.

[Lpm11b] LibPd project members. Libpd project wiki: Embedding pure data as a dsp library. Website, 2011. Available online at `http://gitorious.org/pdlib/pages/Libpd`; visited on March 25th 2011.

[Puc06] Miller S. Puckette. Theory and techniques of electronic music. Website, December 2006. Available online at `http://crca.ucsd.edu/~msp/techniques/latest/book-html/`; visited on March 23rd 2011.

[RjD11] RjDj.me. Artists and labels on rjdj. Website, 2011. Available online at `http://rjdj.me/music/`; visited on March 3rd 2011.

[Roa96] Curtis Roads. *The Computer Music Tutorial*. MIT Press, April 1996.

B. Example projects

All example projects are zipped XCode-projects. Most of the applications will produce bad audio output in the Simulator and therefore it is advised to run them on a device.

B.1. iPhoneAudioSimple.zip

This project contains a simple iPhone-App that produces a 440Hz sine wave using Core Audio. The audio buffer is written in a render callback function. See class `AudioController` for implementation details.

B.2. iPhoneAudioBending.zip

This project contains an iPhone App that allows to hit a note and changing the sound of that note by tilting the device.

B.3. iPhoneAudioBendingMultiwaves.zip

Modified version of the above. Does not implement amplitude modulation but additive synthesis by adding multiple detuned sine waves.

B.4. PdAudioSimple.zip

Just like *iPhoneAudioSimple.zip*, this project produces a 440Hz sine wave. It uses a PureData-patch located under *Resources/*. There is a slider with which one can change the master volume. The value of this slider will be sent to the PureData-patch representing an example on how to send messages to a patch.

B.5. PdAudioBending.zip

This project has the same features as *iPhoneAudioBending.zip* but uses LibPd as DSP library.

B.6. PdSoundEngine.zip

A complete step sequencer application that uses LibPd is contained in this ZIP-file.

B.7. MoMuTest.zip

This project uses MoMu/STK to generate a sine wave on the one, and noise on the other stereo channel. Sliders let the user fade between these channels.