Eder Jiménez O'Shanahan

Development of a Musical Rhythm Game for iOS

Helsinki Metropolia University of Applied Sciences

Degree Programme in Information Technology

Thesis

30 May 2014

Helsinki Metropolia University of Applied Sciences

Author(s) Title Number of Pages Date	Eder Jiménez O'Shanahan Development of a Musical Rhythm Game for iOS 40 pages + 4 appendices 30 May 2014
Degree	Media & ICT
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Juha Huhtakallio, Supervisor

The aim of this project was to face the game development process from every perspective, assuming the roles of a game designer, graphic artist and programmer to create a musical game that trains the musical ear while improving the rhythm skills of the player.

The project consisted of the development of a 2-D rhythm game for *iOS*, specifically for the iPhone. Starting with a study of the actual market, the mechanics of other successful musical games were analyzed. Next, an iterative game design process was established, including the concept, story and mechanics of the game, which directly influenced the implementation. Using *Sprite Kit* as the rendering engine and *Core Audio* for the audio synchronization, the game was implemented natively using *Objective-C* as the programming language.

This project resulted in the first stable version of the game. The main objectives were achieved and some flaws were found through user feedback. Although not ready for publication, the result of this project constitutes a robust starting point for further development.

Keywords	iOS, game development, software engineering, Core Audio

Contents

1 Introduction

The idea behind this project was to create a game with a special purpose: to serve as a training tool for the musical ear. In this way listening should be the main skill needed to win and the skill the user has to develop in order to progress. Moreover, the game will focus on rhythm rather than notes, meaning that being able to distinguish between different pitches is not needed. This purpose alongside with a story, game mechanics and appealing graphics have to constitute a game that can entertain and at the same time teach.

For this project one developer assumed the three main roles involved in the game development process: game designer, graphic artist and programmer. This approach has the benefits of providing a full experience of the game development process and allowing the developer to have an overall view of the process at any time. However, it clearly involves a greater workload, and the time restrictions become a very important matter.

The first challenge of this project resides in the game design. The creation of a musical game that succeeds in training the rhythm skills of the player has to be approached with an exhaustive analysis and refinement of the gameplay. It has to gradually build up the player's abilities as the game progresses, and at the same time provide a fun experience. It cannot be too complicated for beginners because it could lead to frustration and would cause them to stop playing. Similarly, it has to offer a challenge for the most experienced players to prevent boredom, which in fact has the same outcome of the players abandoning the game.

However, game design is not the only obstacle to overcome. Succeeding in the implementation is a crucial matter that cannot be overlooked. Apart from the complexity associated with any game, a musical game requires audio synchronization functionalities that can take a significant part of the development time.

Consequently, in order to successfully handle all the roles and overcome the exposed challenges the best option was to create a mobile game. This project consisted of the development of *The Musical Platform Game*, a 2-D rhythm game primarily made for iPhone but also meant to run on iPad in a further development stage.

2 Theoretical background

The first step when approaching the design of a new game is to take a look at the current market. It is important to see what has already been done, what has already been successful and how it could be improved. Since this project is a rhythm game, this market study was done primarily in the music games category of the App Store.

One of the most popular approaches for musical game mechanics is the one used by the *PlayStation* game series “Guitar Hero”. In this game, there are a number of trays that go from the top to bottom of the screen in perspective, ending in indicators where the upcoming notes will be triggered. These notes move through the trays and the user has to trigger them at the exact time in which they reach the indicators. The notes correspond to the notes of the song that is playing, and therefore the user has to trigger the notes to the beat, creating the musical feeling of the game.

The same mechanics are used by a successful game in the App Store, “Tap Tap Revenge”. Unlike “Guitar Hero”, which is played with a guitar controller, this game only presents three trays and is played by tapping the screen. Adding more notes to the song, together with adding simultaneous notes, makes the user tap more often, increasing the difficulty. One of the key elements of “Tap Tap Revenge” is that it uses popular songs for the levels, which engages the user in the gameplay.

“Cytus” is another successful music game from the App Store, presenting some similarities with “Tap Tap Revenge”. It is similar in the sense that the user has to tap notes at the right time, but they do not fall to the bottom of the screen. Instead, they are placed all over the screen. There is a horizontal line that keeps moving up and down at the same rhythm as the playing song. When the line intersects a note it has to be tapped by the user. Therefore, the essence remains, but the approach is different. [1.]

Continuing with this type of mechanics, “Audio Ninja” offers another visual approach. In this case the notes are not explicitly shown on the screen for the user to tap them. The game shows a running ninja that has to defeat the upcoming enemies by tapping the right and left side of the screen to the beat. The enemies are actually representing notes and the trigger indicator is the moment at which they reach the main character. In this sense, “Audio Ninja” presents the same mechanics but offers an alternative way of showing it to the user, creating a different game experience. [2.]

Outside of the App Store, there is a music game worth mentioning because of its game mechanics. “Patapon” is a rhythm game for the PlayStation Portable (PSP), presenting a substantial change in the way gameplay is structured. Rather than making the user play to the beat, that is, at the same time as the events are occurring, it plays a sequence of buttons the user has to repeat after watching it. The rhythm nature of the game comes with the fact that these sequences have to be repeated according to the beat of the song. Therefore the player has to be aware of the music that is being played.

The reviewed games are of interest to this project because of their mechanics. They seem to be a good starting point to achieve the educational purpose of the game. However, there are more musical games that have been successful in the App Store but are not reviewed in this thesis since their mechanics do not contribute to the general purpose of this project.

3 Methods and materials

3.1 Game design

In order to be successful, a game has to provide an added value not already present in the market. This added value is basically achieved in two different ways: either by innovating and creating something entirely new or by taking something existing and making it better than any other game. This was an important fact to take into account during game design, as the game would eventually be released in the market where it has to offer something new.

It was also really important not to lose the aim of the project, which was to provide a tool to have fun and at the same time improve the rhythm skills of the player. The game mechanics have to allow the player to slowly build up his or her skills as the game progresses, while being fun and preventing the user from getting bored.

The problem with games such as “Tap Tap Revenge” is that the user does not necessarily need to feel the underlying music, as it can be played visually. The game can be played ignoring the music, by only focusing on when the notes hit the indicators. This is the reason why in *The Musical Platform Game* there are no visual aids to predict when to tap the note. Instead, the only way for the user to succeed is to rely on the music. Playing without visual aids can be easy for players who play an instrument or have some musical background. However, it could be really hard for an average player, and that is one of the biggest challenges of this project. It should be understandable and fun for someone without any previous musical knowledge.

Apart from the game mechanics, there should be an underlying story and a theme. The game has to be appealing to the player, not only because of its gameplay but also because of its style and visual appearance. All of the characteristics previously discussed were taken into account in the development of *The Musical Platform Game*.

Game design evolution

Since the idea was to provide a rhythm-training tool, the first game concept was a drumming game. In this game a drum set is shown from a top view and the user is an aspiring drummer. During the game the user is shown a series of *drumbeats* of increasing difficulty and he or she has to repeat them correctly in order to continue playing. The *drumbeats* are played and the drums are lighted up in the same order, making it easier for the user to grasp the beat. In this game it would also be allowed to select among different music styles and even have a store to purchase new tracks or customize the drum set.

This concept evolved into another approach that was visually more interesting. The essence of the drumming game remains, but this time the user is not an aspiring drummer and does not have to tap a drum set. This game is a musical platform game where the main character is an adventurer running inside ancient ruins trying to find his loved one. The problem is that it is extremely dark and the adventurer cannot see what is coming next in front of him. The right side of the screen is completely covered by a black fog that does not let him see through it, hiding all the traps and dangers of the ruins. Therefore, the only way for him to survive is to listen to his loved one who will tell him the way in the form of *drumbeats*. These sounds tell him whether he has to jump or bend down on the next beat, in order to avoid the upcoming danger.

The downside of the previous game concept is that half of the screen has to be black, considerably limiting the visual experience. Mobile devices have a limited screen size, which should be used efficiently in games. The musical platform idea remains in the current implementation which shares the same mechanics which will be explained thoroughly in the following sections. However, the underlying story was modified in order to create a much more interesting gaming experience.

Game mechanics

The game is about a rapper-looking kid that walks around the city with a *boom box* listening to his music. However, his bad luck will make it a risky journey, as he will have to face all sorts of dangers: dogs that come out to bite him, falling cans and even rakes placed on the streets will threaten him during the way. Luckily, there is a way to avoid all these dangers, and it is by listening to his music. He cannot predict what is the next danger to come, but his magical *boom box* will show him the way in the form of *drumbeats*. In this way, he will have to carefully listen to the music that is playing and then repeat it as accurately as possible.

Repeating the *drumbeats* will make him perform actions to avoid the upcoming dangers. For instance, if the *drumbeat* plays a bass drum and he repeats it correctly, he will bend down at the exact moment to avoid a dog that is coming out from a fence to bite him. Therefore, he will survive as long as he keeps on listening to the *drumbeats* and repeating them correctly.

The game is divided into levels, each of them composed of a variable number of *drumbeats*. During a level, the game switches between *showing mode*, where the *drumbeat* is played and shown to the user, and *capturing mode*, where the user has to repeat it by tapping the screen. A screenshot of both modes is shown in figure 1.

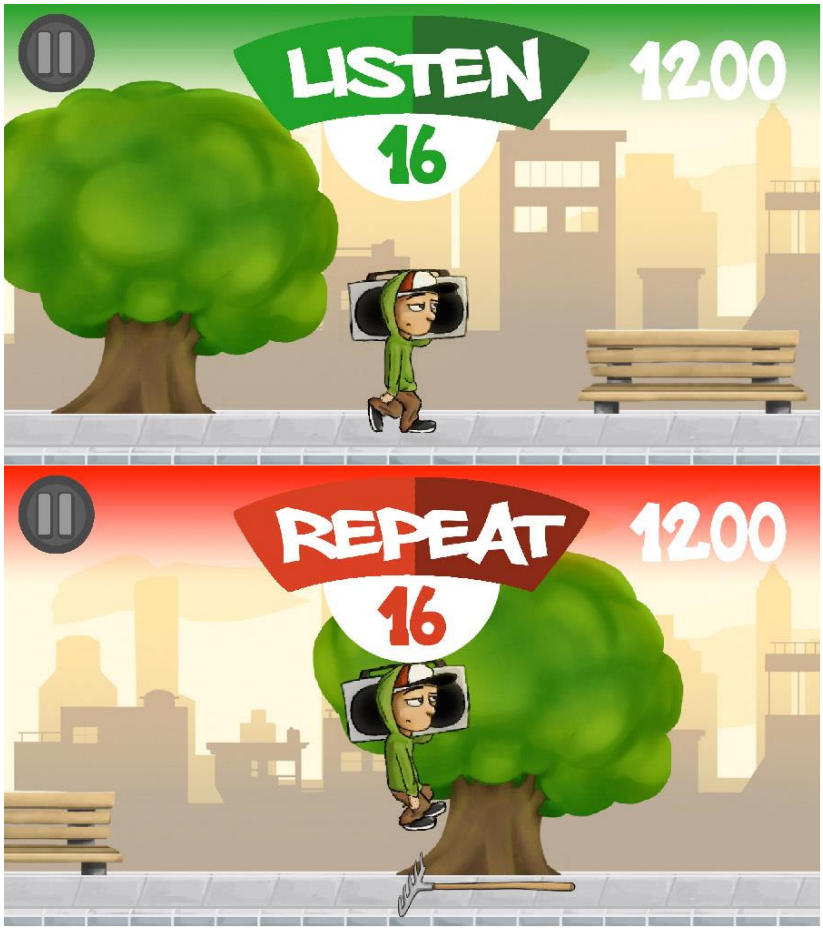


Figure 1. *Showing* and *capturing* modes, respectively

As seen in figure 1, the character is jumping over the rake as the user taps the *drumbeat* correctly. During the *showing mode* the character is walking and the *drumbeat* is played, and the user has to listen and memorize the pattern. During the *capturing mode* the dangers appear in the places where the user is supposed to tap when repeating the *drumbeat*. The countdown and the bar on top of the character indicate when the current mode is about to end. This lets the user know when to start repeating the *drumbeat* that is currently being played.

Each time the user succeeds in repeating a note, the character will avoid the danger and the score will be increased. On the other hand, each time the user fails in repeating a note, a danger will hit the character, decreasing the score. If the rake from figure 1 hits the character, it would look like the screenshot in figure 2:

Figure 2. Character being hit by a rake

As seen in figure 2, the character stepped on the rake and the stick hit him, turning him red and decreasing the score. This can happen in two scenarios: when the user taps the wrong note or when the note is tapped out of beat. There are two notes that compose every *drumbeat*: the bass drum and the snare drum. These notes are repeated by tapping the left and right sides of the screen. The left side corresponds to the snare drum and the right side to the bass drum, as shown in figure 3:

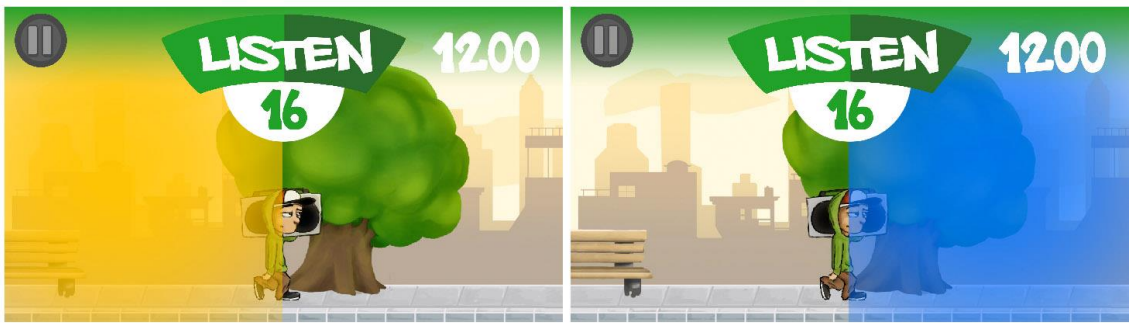


Figure 3. Screen areas depending on the note

During the *showing mode* the *drumbeat* is played and at the same time the corresponding screen areas are lighted up, as seen in figure 3. This helps indicating the user the side of the screen that corresponds to each note.

There are three different types of dangers the player will encounter in his or her way. It can be a falling can, a rake or a dog hiding behind a fence ready to bite him or her. These dangers are shown in figure 4.

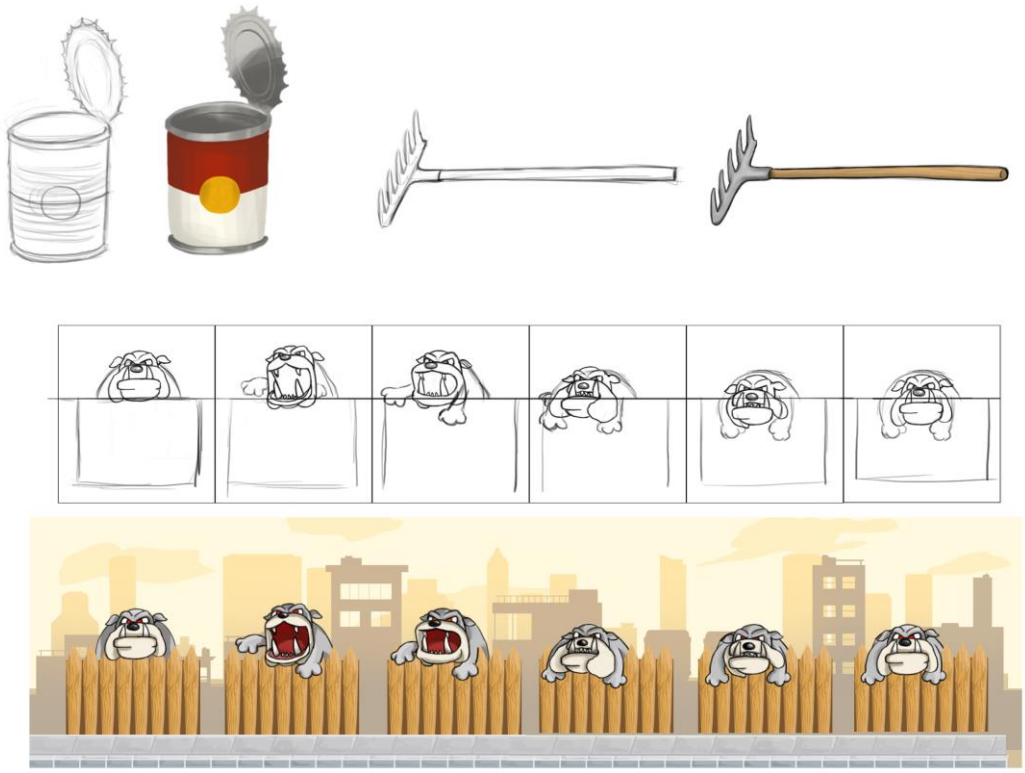


Figure 4. The different dangers in *The Musical Platform Game*

From the three dangers shown in figure 4, there are two that best fit the purpose of the game of being completely musically driven: the falling can and the dog. The reason is that the moment in which the user has to tap to avoid them cannot be visually determined. The user has to rely completely on the music to surpass them. On the other hand, the rake is placed on the floor and the user sees it coming. This means that the user can somehow predict the moment to tap by visually calculating when the rake is about to hit the player.

3.2 The audio engine

A fundamental matter to solve for a musical game is audio synchronization. It is crucial to carefully analyse the game mechanics and choose the simplest and most efficient method that would fulfil the needs of the game. The audio engine has to be simple because there is no need in investing a huge amount of time implementing a complete audio library when the game is not going to use it. Also, it has to be efficient because it needs to work in conjunction with the graphics engine and the rest of the game logic. Therefore the device's resources should be used sparingly as they are shared among the different modules of the game.

In addition, the audio engine should be robust, reliable and constantly synchronized. The platform is a mobile device, which means that interruptions such as phone calls and text messages will occur. Also, depending on the number of applications and processes the phone is running at a given moment, a drop in the Frames Per Second (FPS) can happen. Consequently, the audio engine should be independent from the graphics engine and be able to handle interruptions. The best way to approach the development of the audio engine is to take a look at the mechanics of *The Musical Platform Game*. The following paragraphs will focus on establishing the requirements and extracting the functionalities that should be implemented.

A level of *The Musical Platform Game* alternates between two modes or states: the showing mode and the capturing mode. During the showing mode the drumbeat is played for the user to memorize it. Immediately after, the capturing mode takes its turn; the user has to repeat the drumbeat as accurately as possible by tapping the screen. Therefore, two main issues have to be solved for the mechanics of the game: how to play the drumbeats and how to recognize if the user repeats them correctly.

Apart from the game mechanics, each level has its own soundtrack, the instrumental base on which the drumbeats are played. This means that the music has to be playing in both showing and capturing modes without interruptions. Also, the audio engine should allow the instrumental track to loop and at the same time be synchronized with the drumbeats.

The synchronization does not only concern the audible part of the game. As it is a platform game, the graphics have to be synchronized with the audio. For instance, if a biting dog appears at the next bass drum and the character bends down to avoid it, this action has to happen exactly when the bass drum plays. As a result, the audio engine has to provide a neat and efficient way of letting the graphics engine know when to trigger the various graphic events.

To sum up, the requirements of the audio engine are the following:

- Playing and looping a sound track
- Playing drumbeats
- Synchronization between the sound track and the drumbeats
- Synchronization between the graphics and the audio
- Tapping recognition of the drumbeats.

The graphics engine is to be implemented in native code, with the Sprite Kit framework. This has to be taken into account as well while developing the audio engine in order to meet the requirements previously discussed.

3.2.1 First implementation

There are a number of ways in which audio can be played on an iOS device. The general rule is that the simpler the method is to implement, the less control the developer has on its behaviour. As discussed in section 3.2, the audio engine should meet the requirements of the game trying not to surpass them greatly, finding the most efficient solution for the particular problem. This led to the process carried out in this project: going through the simple methods first before considering the low-level libraries.

The simplest way of playing sounds in iOS is by using *System Sound Services* from the *AudioToolbox* framework. It can play sounds by only using 4 lines of code. It is sufficient for playing sound effects in regular iOS applications but lacks many key features for this project. The maximum length of the sounds is 30 seconds and it does not support the playback of multiple sounds at the same time [3,3]. The simultaneous playback was a very important requirement for this project, since the backing track and

the drumbeat should play at the same time. Thus, *System Sound Services* was not a viable choice.

A more complete solution for audio playback is the *AVAudioPlayer* class, from the *AVFoundation* Framework. It has no restrictions for the maximum length of the sound files, it supports simultaneous playback and allows playing in loop mode, among other features [4,4]. It matched the requirements, and that was the reason why the first implementation of the audio engine was done with *AVAudioPlayer*.

The usage of the *AVAudioPlayer* class is simple. The method for creating an object and loading and playing an audio file is shown in listing 1.

```
AVAudioPlayer *audioPlayer;
NSURL *url = [NSURL fileURLWithPath:[NSString
stringWithFormat:@"%@/audiofile.mp3", [[NSBundle mainBundle]
resourcePath]]];

NSError *error;
audioPlayer = [[AVAudioPlayer alloc]
initWithContentsOfURL:url error:&error];
audioPlayer.numberOfLoops = -1;

if (audioPlayer == nil)
      NSLog([error description]);
else
   [audioPlayer play];
```

Listing 1. Loading and playing an audio file with AVAudioPlayer. Adapted from Grisby (2009) [5].

As seen in listing 1, the problem of playing the instrumental track of the level is solved. Furthermore, the looping can be easily configured with the *numberOfLoops* property. In this example it is set to -1, which means that the audio will loop forever.

Once the instrumental track playback had been achieved, the next matter to solve was the drumbeat playback. The first approach was to create another *AVAudioPlayer* object, resulting in a total of two: one for the instrumental track and another one for the drumbeat part. That is, two audio files of the same length had to be created for each level: the first one containing the whole instrumental part and the second one a series of drumbeats separated by silences of the same length. Therefore, if a drumbeat inside the drumbeats track lasts for 15 seconds, it has to be followed by 15 seconds of

silence. The first 15 seconds correspond to the *showing mode* and the following 15 seconds of silence correspond to the *capturing mode*. In order to clarify this concept, a sample level is presented in figure 5:

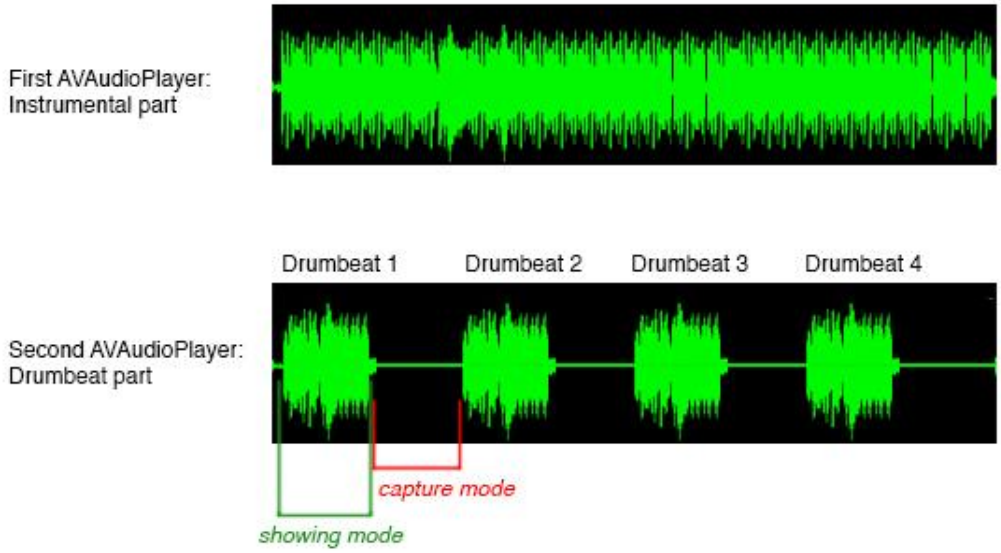


Figure 5. Sample of the first implementation of the audio engine

As seen in figure 5, the synchronization is easily achieved. It is only a matter of starting both *AVAudioPlayer* objects at the same time. Therefore, after successfully implementing the audio playback the next step was to find a way of recognizing the user input during the *capture mode*.

User input is handled with the *touchesBegan* method from the *UIResponder* class, which is called each time the screen is tapped. Therefore, determining if the tapping is correct is only a matter of registering the time at which the *touchesBegan* method is invoked. If it matches with the time the user was expected to tap, it will be considered a success. Furthermore, the times at which the user should play are registered in a list, created beforehand, and during runtime this list is checked and the results are evaluated. However, the times registered in the list are relative to the playing time of the sound track, not to the system time. Hence the evaluation has to be done in context with the audio track that is being played.

The *AVAudioPlayer* class offers a way to access the playing time of the audio file: the *currentTime* property. This property gives the playback point, in seconds, of the audio file [4,8]. Therefore, the first implementation of the audio engine was based on this

property, which in general gave good results. However, some problems started to arise as the development went further. For drumbeats that required a bit more precision and in situations where the device was overloaded, the synchronization started to fail. In general, the *currentTime* property is not accurate enough. It is measured in seconds, and is actually meant to perform simple tasks such as rewinding an audio track but not to provide real-time synchronization. Therefore, a precise audio engine was needed, and the only way to achieve it was to go deep into the *Core Audio* libraries.

3.2.2 Actual implementation

The Audio Units API

The audio engine of *The Musical Platform Game* was implemented with the lowest level sound library available in iOS: the Audio Units API. It gives the most control over the sound processing with the downside of an added complexity, as it deals with raw audio data. Audio units are modules that perform a certain task in the audio processing, such as obtaining audio from a source, applying an effect or mixing different audio streams into one and sending the result to the output hardware, among others. These tasks are performed with buffers of audio samples, meaning that audio units give sample-level accuracy in the audio processing.

Audio Units are connected through audio streams in the same way real audio equipment is connected with wires. Each audio unit has a variable number of elements which can have input and/or output scopes. The number of elements and its scopes will depend on the nature of the particular unit and how it processes audio. Also, depending on the task they perform, audio units are split into various categories, such as effect, mixing and I/O units, which are the most commonly used in iOS. An effect audio unit can, for instance, apply a reverb effect to the audio source or process the audio data in some way. An I/O unit can obtain sound data from the microphone and/or direct the audio stream to the speaker. Mixer units work in the same way as real audio mixers, combining audio inputs and sending the result to various outputs. [6,16; 7,53.]

Typically, audio units are arranged in an *audio processing graph*, which makes the audio data flow from the input source to the output hardware after passing through the audio units [7,53]. The *audio processing graph* works by "pulling" the data. That is, the audio units request new blocks of samples when the buffers are empty, and this occurs in the opposite direction of the audio flow:

> With a pull architecture, the framework tells the programmer,"Don't call us—we'll call you." If you have a chain of audio units, the last one (which is probably the I/O unit that will send audio to the speakers or headphones) pulls from the units connected to its input elements, saying, in effect,"Give me something to play."These units call the units upstream from them, and so on. [7,55.]

In order to clarify this concept, figure 6 shows the structure of a simple *audio processing graph* with two audio units:

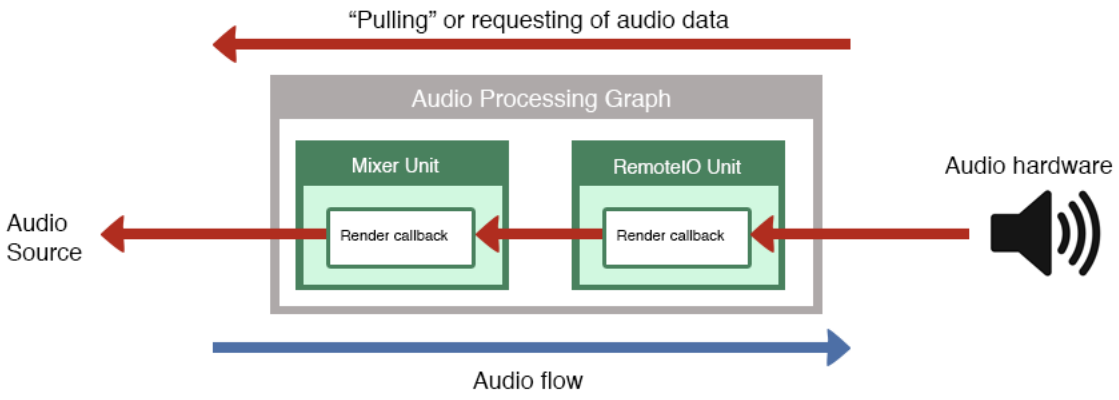


Figure 6. Sample audio processing graph (Modified from Audio Unit Hosting Guide for iOS [6,24])

As seen in figure 6, the audio flow goes from the audio source to the output hardware, passing through the audio units and connections of the *audio processing graph*. However, the data requests occur in the opposite direction: starting at the output hardware that asks the *RemoteIO* unit for the next set of samples, and ending at the *Mixer* unit's *render callback* function, which obtains the audio data from the audio source. In fact, it is in the implementation of the *render callback* where the programmer has access to each individual sample of the audio source, and where most of the synchronization of the game is handled.

The *render callback* function is executed in a real-time priority thread, which means that the code it contains should be efficient and consume the less possible amount of time. The *render callback* function is called hundreds of times per second, so if the function takes too much time processing the audio samples and a new call arrives in the middle, the sound can present gaps. Therefore, the *render callback* function is the best place to handle audio synchronization. The real-time priority thread where it runs provides the most accurate possible timing available in iOS. [6,25.]

The *MPGAudioManager* class

The *MPGAudioManager* class handles all the audio functionality of the game. It manages the *audio processing graph*, is responsible for loading the audio files and handles the audio interruptions, among others. This class is only instantiated once during the life cycle of the game. All the tasks are performed by the same instance. For a clearer explanation, some methods of the *MPGAudioManager* class are presented in appendix 2. The first one to be explained is the *init* method, which handles the initialization in three sequential steps:

- Setting up of the audio session
- Creation of the *audio processing graph*
- Creation of the drum sampler.

The audio session is where the connection to the device's audio system occurs. It is where the application requests access to the recording or playback hardware, and where parameters such as *sample rate* and *buffer duration* are set.

In digital audio, the analog sound is represented numerically by approximating the sound waves. The sound is measured at regular intervals and a value is assigned to each measurement. The number of values measured per second is called the *sample rate*. For instance, the *sample rate* used in CD-quality audio is 44.1 kHz, or 44100 samples per second. The higher the *sample rate*, the more accurately the sound wave is represented, therefore the higher the quality of the sound. [7,27.]

The *buffer duration* indicates the length of the buffer for the input and output cycles. Each buffer handles a specific number of frames, which are a set of samples occurring at the same time. For instance, mono frames consist of one sample whereas stereo frames consist of two, one for each channel. The *buffer duration* is specified in seconds, but the duration in frames can be easily obtained by contrasting it with the *sample rate*, that is, with a *sample rate* of 44.1 kHz and a *buffer duration* of 5 milliseconds the application would be using 256 frames. [8,32-33.]

The audio session is initialized in the *setupAudioSession* method, beginning by setting the audio session's category, that is, how the application intends to use the audio. The different categories include *AVAudioSessionCategoryPlayback, AVAudioSessionCategoryRecord* and *AVAudioSessionCategoryPlayAndRecord*. However, for this application the *AVAudioSessionCategoryPlayback* category is sufficient. The different parameters are set to the *sharedInstance* property of the *AVAudioSession* class, using an error handler for every method call. Listing 2 shows the first lines of the *setupAudioSession* method to clarify this concept.

```
NSError *audioSessionError = nil;
AVAudioSession *sessionInstance = [AVAudioSession
sharedInstance];
[sessionInstance setCategory:AVAudioSessionCategoryPlayback
error:&audioSessionError];
if (audioSessionError != nil) {NSLog (@"Error setting audio
session category."); return NO;}
```

Listing 2. Configuring the audio session

The *setCategory* method makes the audio session adopt the desired category, as shown in listing 2. However, when setting the *sample rate* and the *buffer duration,* a preferred value will be given instead. The audio system tries to set the desired value, but depending on the device another value can be set. That is the reason why after the call to *setPreferredSampleRate,* the actual *sample rate* will be asked and stored in *MPGAudioManager*'s *sampleRate* property.

The *audio processing graph*

With the audio session set, the next step is to create the *audio processing graph*. It is the core of the *MPGAudioManager* class, as it contains all the audio units that handle all the audio from the game. An overview of its implementation is shown in figure 7:

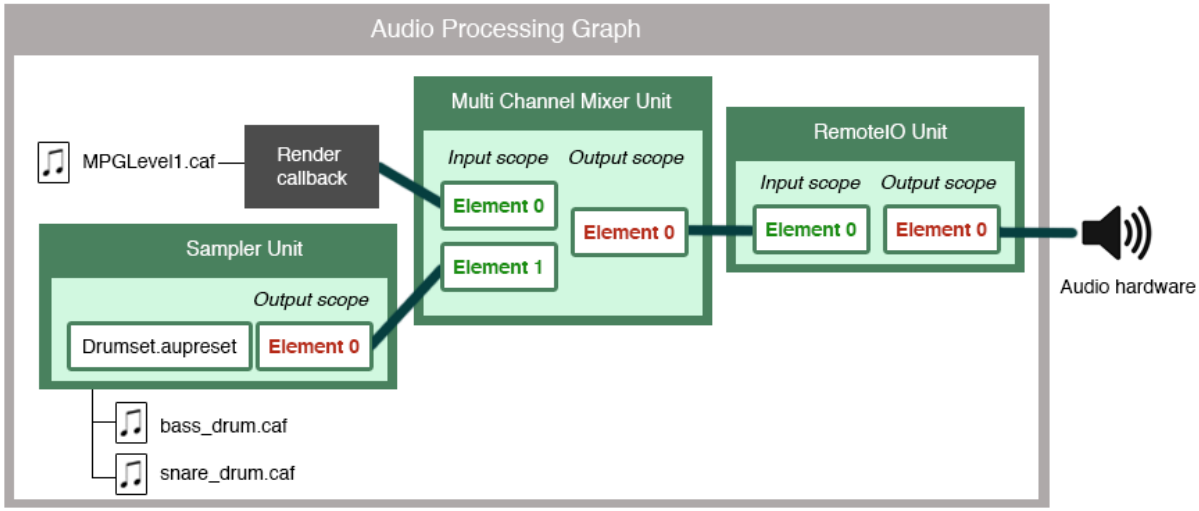


Figure 7. Audio processing graph of *The Musical Platform Game*

As seen in figure 7, the *audio processing graph* is composed of three audio units: the *RemoteIO* unit, the *Multi Channel Mixer* unit and the *Sampler* unit.

The I/O units are the interface for accessing the input and output hardware of the device, and every *audio processing graph* has only one, regardless of the nature of the particular application [6,22]. This implementation uses the *RemoteIO* unit that provides low-latency access to the audio hardware and allows even simultaneous input and output [6,46]. However, since it is not necessary to access the device's microphone, only the interface to the output hardware is used in this case. The role of this unit is to forward the audio coming from the *Multi Channel Mixer* to the audio output hardware, which can be the speaker or the headphones, if connected. No audio processing is performed in this unit.

The *Multi Channel Mixer* unit has two input elements, one with a *render callback* attached to it and another one connected with the *Sampler* unit. Its role is to combine the audio coming from these elements and send the resulting mix to its only output element. The audio coming from the first input corresponds to the instrumental part of

the level, whereas the second input will generate the drumbeats. The *Multi Channel Mixer* can also be configured to level the volume of the inputs to provide an even result. As seen in figure 7, the resulting mix is sent to the *RemoteIO* unit and eventually to the output hardware.

The *audio processing graph* is created in the *initializeAUGraph* method from the *MPGAudioManager* class. It is performed in the following steps:

1. Create the *AUGraph*
2. Create the *AUNodes*, which represent the audio units, and add them to the graph
3. Open the *AUGraph*
4. Make the connections between the nodes
5. Obtain the references to the nodes
6. Configure the *render callback* for the *Multi Channel Mixer* audio unit
7. Initialize the *AUGraph.*

In order to create an audio unit two elements are required: a description for the audio unit and an *AUNode* that keeps a reference to it. The audio units are described with the *CAComponentDescription* class, by specifying its type, subtype and manufacturer. The *AUNode* and the description are passed to the *AUGraphAddNode* method to create the audio unit. This process is shown in listing 3:

```
AUNode mixerNode;
CAStreamBasicDescription desc;
CAComponentDescription
mixer_desc(kAudioUnitType_Mixer,
kAudioUnitSubType_MultiChannelMixer,
kAudioUnitManufacturer_Apple);
CAShowComponentDescription(&mixer_desc);
result = AUGraphAddNode(_processingGraph, &mixer_desc,
&mixerNode );
```

Listing 3. Creation of the *Multi Channel Mixer* audio unit

The code from listing 3 is modified from the *initializeAUGraph* method. It shows the creation of the *Multi Channel Mixer* unit, but the same process is applied to all the audio units. Once created, they are connected with the *AUGraphConnectNodeInput* method, which takes as arguments the two *AUNode* references and the output and

input elements that want to be connected. For instance, the *Multi Channel Mixer* unit and the *RemoteIO* unit are connected as shown in listing 4:

```
result = AUGraphConnectNodeInput(_processingGraph,
mixerNode, 0, ioNode, 0);
```

Listing 4. Connecting the *Multi Channel Mixer* unit and the *RemoteIO* unit

As seen in listing 4 and in figure 7, the output element 0 of the *Multi Channel Mixer* unit is connected to the input element 0 of the *RemoteIO* unit. After making all the connections, the *AUNode* references are updated by calling the *AUGraphNodeInfo* method. The next step then is to configure the *render callback* function for the *Multi Channel Mixer* audio unit. The complete implementation of the *render callback* function is presented in appendix 2, alongside with the other methods from the *MPGAudioManager* class.

The *render callback*

The *render callback* function is called every time the audio unit needs audio data to play. It works by fetching the frames from a source buffer and transferring them to a destination buffer in the audio unit. In this implementation, the source buffer will contain the instrumental track, and the audio will be reproduced as the *render callback* transfers the frames to the *Multi Channel Mixer* audio unit's input buffer. Therefore, the first step is to define the structure of the buffer. Listing 5 illustrates the adopted structure:

```
typedef struct {
    AudioStreamBasicDescription asbd;
    AudioUnitSampleType *data;
    UInt32 numFrames;
    UInt32 sampleNum;
} SoundBuffer, *SoundBufferPtr;
```

Listing 5. The audio buffer structure

As seen in listing 5, the buffer has four elements:

- *asbd*: the structure that stores the audio format information, such as channel count and *sample rate*.
- *data:* a pointer to the buffer that contains the audio frames.

- *numFrames:* the number of frames contained in the buffer.
- *sampleNum:* the current sample that is being processed inside the buffer.

There is one *SoundBuffer* that stores the instrumental track that is playing: *MPGAudioManager*'s *mSoundBuffer* instance variable. When a new level is loaded, the *loadSong* method will be called with the path to the audio file as a parameter. This method retrieves the audio file and updates the four fields of the *mSoundBuffer*. First, it stores the audio format information in the *asbd* field and loads the complete audio file into memory, keeping a reference to it in the *data* field. After that, it updates the *numFrames field* with the number of frames that were loaded into the memory, and finally it initializes the *sampleNum* field to zero.

With the buffer structure defined, the *render callback* can be correctly explained. First of all, the header has to conform to the *AURenderCallback* prototype [6,25]. The *render callback*'s header from the *MPGAudioManager* class is presented in listing 6:

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

Listing 6. The *render callback* header

Note that the *render callback* from listing 6 is implemented in C, as most of the functionalities in the *Audio Unit API*. The function accepts 6 parameters:

- *inRefCon*: points to the reference given when attaching the *render callback* to the audio unit.
- *ioActionFlags*: flags to modify the render process. The one that is used in this implementation is the *kAudioUnitRenderAction_OutputIsSilence*, used when there is no audio to process.
- *inTimeStamp*: a timestamp for the time at which the function is being called.
- *inBusNumber*: the bus in the audio unit the *render callback* is attached to.
- *inNumberFrames*: the number of frames to be processed in the call.
- *ioData*: the destination buffers that have to be filled with audio frames. It is a list of buffers, as the audio can be split into multiple channels. [6,26-27.]

When attaching the *render callback* function to the *Multi Channel Mixer* unit, two elements have to be specified: the function's name and the reference for the *inRefCon* parameter. Every time the *render callback* is called, the *inRefCon* parameter will point to the structure specified here, which should be where the audio data is. As seen in listing 7, it is set to *mSoundBuffer*, which will provide the audio frames of the instrumental track:

```
AURenderCallbackStruct rcbs;
rcbs.inputProc = &renderInput;
rcbs.inputProcRefCon = mSoundBuffer;

result = AUGraphSetNodeInputCallback(_processingGraph,
mixerNode, 0, &rcbs);
```

Listing 7. Attaching the *render callback* function to the *Multi Channel Mixer* unit

As shown in listing 7, the *renderInput* callback is set for the input element zero of the *Multi Channel Mixer* unit. Now, the functioning of the *render callback* can be explained in depth.

There are two *static* variables in the *MPGAudioManager* implementation that are used inside the *render callback*. This is indicated with the "RC" prefix and they are the following: *RCcurrentSong* and *RCgameIsOn*. The first one is a reference to a *MPGSong* object (this class will be discussed in depth later on in this section). The second one is a Boolean variable used to handle the game pauses inside the *render callback*. They have to be *static* because the *render callback* is *static* and therefore cannot get access to *MPGAudioManager* instance variables. The following explanation of the *render callback* refers to the implementation presented in appendix 2.

First of all, the *ioData* parameter is split into the buffers *outA* and *outB*. The samples that are copied into these buffers are the ones that are going to be played by the *audio processing graph*. They represent the left and right audio channels, respectively. Therefore, it is logical that in order not to produce any sound, these buffers have to be filled with zeroes. In addition, the *kAudioUnitRenderAction_OutputIsSilence* flag has to be turned on in the *ioActionFlags* parameter [6,26]. As seen in the *render callback* implementation, if the *RCgameIsOn* is set to *NO,* the buffers will be filled with zeroes and all of the functionality of the *render callback* will be omitted.

A previous implementation of the *render callback* did not have the *RCgameIsOn* variable. The game pauses and interruptions were handled by stopping the *audio processing graph*. However, when the graph is stopped, the whole audio engine is stopped. It is a better solution to keep the *audio processing graph* running with the game pauses. This allows, for instance, a backing track to play during the pause menu, or sound effects to be played when the buttons are pressed. It is important to note that it is still necessary to stop the *audio processing graph* in some special cases. If the graph is not stopped when the application is going to the background or when a phone call arrives, the whole game would crash.

If the *RCgameIsOn* variable is set to *YES,* the *render callback* will continue and prepare to fill the buffers *outA* and *outB* with the frames from the instrumental audio track. As previously discussed in this section, the *inRefCon* keeps a reference to the *mSoundBuffer*. Therefore its fields are split into the *sample, bufSamples* and *in* instance variables, the latter one being the buffer with the audio frames. The buffer is filled as shown in listing 8:

```
for (UInt32 i = 0; i < inNumberFrames; ++i) {
    [RCcurrentSong processCurrentSample];

    // Play the instrumental part
    outA[i] = in[sample++];
    outB[i] = in[sample];

    // Loop the instrumental part
    if (sample > bufSamples) {
        sample = 0;
    }

    RCcurrentSong.currentSample++;
}
sndbuf[inBusNumber].sampleNum = sample;
```

Listing 8. Filling of the *outA* and *outB* buffers

Listing 8 begins with a *for* loop that runs *inNumberFrames* times, which is the number of frames that are to be processed in this render call. The frames are taken from the *in* buffer starting at the position *sample*, which is the last frame that was processed in the previous render call. The looping of the instrumental track is achieved by setting the *sample* variable back to zero when it surpasses the limits of the *in* buffer. In this way, the instrumental track will be looped without any noticeable gap as long as the *RCgameIsOn* variable is set to *YES* and the level is still playing.

The most important statement of the function is the following: “RCcurrentSong.currentSample++”. The implications of this line will be discussed later on in this section, but all the audio synchronization mechanics are based on this *currentSample* property. As previously exposed, the *render callback* lives in a real time priority thread, which means that there will be nearly zero delays in its execution. That makes the *currentSample* property an extremely accurate indicator of the overall progress of the level.

Musical background of the game

Each level of *The Musical Platform Game* consists of two differentiated parts: the instrumental part and the drum part. The instrumental part is the actual song, the audio that gets looped over and over during the execution of the level. The drum part constitutes the series of drumbeats that are programmatically generated in real time. It creates the dynamics of the game by playing drumbeats and then making the user repeat them.

First approach: *drum and moment* list

The instrumental part is simply an audio clip that plays in the loop mode. However, the audio engine generates the drumbeats, and therefore a way of specifying them has to be defined. The first step is to split the problem into small pieces, identifying the minimum piece of information needed. In essence, a drumbeat can be reduced into two elements: drum and moment, that is, which drum should be played and at which moment in time. Therefore, the first approach is to specify these two elements in a list, as shown in table 1.

Table 1. First approach at specifying a drumbeat with absolute timing

Drum	**Moment**
Bass Drum	0 sec
Snare Drum	0.75 sec
Bass Drum	1.5 sec
Snare Drum	2.25 sec

The drumbeat from table 1 is specified in a timeline fashion with absolute timing. Hence, the total length of the drumbeat is 2.25 seconds. If this was to be interpreted by the game, it indicates that a bass drum sound has to be played at the beginning, followed by a snare drum after 0.75 seconds, a bass drum after 1.5 seconds and a snare drum after 2.25 seconds. Note that all of the times are measured from the beginning of the drumbeat.

Since the aim of the game is not to play short drumbeats but complete levels (sequences of drumbeats played one after the other), the absolute timing specification becomes quite challenging. Every moment of every drum has to be calculated beforehand, taking into account all the previous elements in the list. In fact, this constitutes a big handicap, since a level can last from two to three minutes.

An improved version of the *drum and moment* approach is to use relative timing, that is, specify the moments relative to the previous element in the list. In this way, the problem of specifying long levels is partially solved. In table 2 the drumbeat from table 1 is rewritten using relative timing.

Table 2. Drum and Moment approach with relative timing

Drum	**Moment**
Bass Drum	0 sec
Snare Drum	0.75 sec
Bass Drum	0.75 sec
Snare Drum	0.75 sec

In this way, a drumbeat that goes at regular intervals can be specified without considering the previous elements in the list, like table 2 shows. A long drumbeat can be created easily by adding new rows to the table with the same moment value. Although using relative timing solves some of the issues, it is still not sufficient to describe more complex beats. If the drumbeat contains elements that do not follow regular intervals, the same problem as with absolute timing will arise. The past elements have to be taken into consideration.

Therefore, there is a need for a system in which drumbeats of any length can be easily specified without taking into account the previous elements. The best option then is to take a look at the system musicians have been using for centuries to describe their musical pieces: the musical notation.

Second approach: musical notation

Musical notation defines a way in which any musical piece can be represented, and this includes drumbeats. The system that will be used is the *modern staff notation*, in which notes are arranged on a staff, as shown in figure 8.

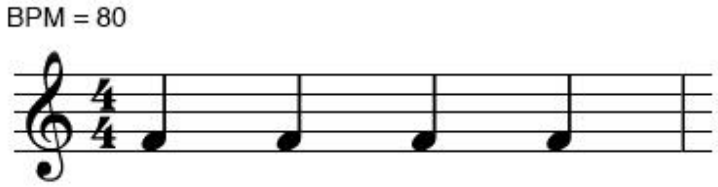


Figure 8. Musical notation approach

The drumbeat from figure 8 matches the one represented in table 1 and table 2, but by using the *modern staff notation*. It is an easy and neat way to specify drumbeats, scalable to any size. However, in order to understand musical notation and establish a way to translate musical scores into a representation understandable by a computer, some terms must be defined.

A note on the staff represents a drum that has to be played, and its position determines which drum it is. Figure 9 shows the convention for drum scores, that is, a note placed on the first space of the staff from the bottom represents a bass drum. On the other hand, a note placed on the second space from the top represents a snare drum.

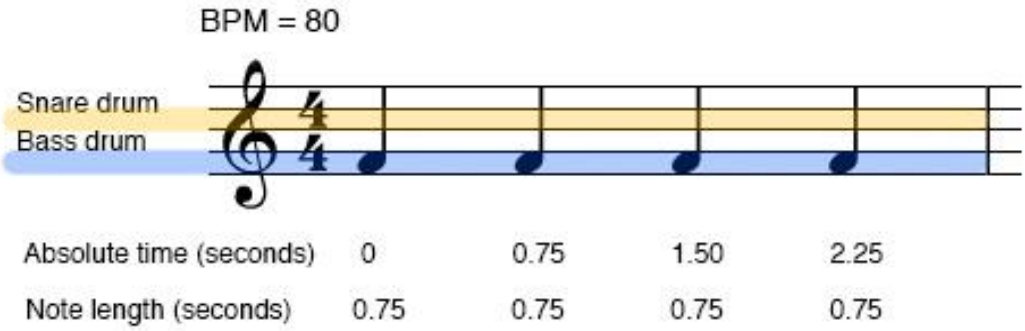


Figure 9. Convention for drum scores

Musical scores are read from left to right, meaning that it is a kind of timeline where the notes are played sequentially. In figure 9, the drumbeat starts with the first bass drum on the left side and ends with the fourth one located at the right end of the staff. Establishing an analogy with the *Drum and Moment* representation discussed in this section, the position of the notes on the score represents the *drum*. The way the *moment* value is represented is to be discussed in the following lines.

All the notes from figure 9 are *quarter notes*, which means that they have the length of one beat. The length of a beat will vary according to the speed of the song, which is given by the Beats Per Minute (*BPM*) value. As its name shows, the *BPM* indicates how many beats take place in a minute, thus calculating the length of a beat is straightforward:

Beat length = 60 / BPM

If this formula is applied to the drumbeat from figure 9, it will give the following result: 60 / 80 = 0.75 seconds. This means that every *quarter note* in a drumbeat that goes at 80 *BPM* has the length of 0.75 seconds.

Apart from *quarter notes*, there are *half notes, eighth notes* and *sixteenth notes*, among others. They have different lengths and combined together are able to specify any drumbeat. For this project a convention was adopted and a numeric value was assigned to each type of note. Table 3 shows this convention:

Table 3. Note values for *The Musical Platform Game*

Note	Name	Value (by convention)	Length in a 80 BPM song (in seconds)
	Whole note	1	3
	Half note	2	1.5
	Quarter note	4	0.75
	Eighth note	8	0.375
	Sixteenth note	16	0.1875

With the convention shown in table 3 it is possible to specify a drumbeat by a series of numeric values. For instance, the drumbeat from figure 9 can be written with this notation as a series of "drum and note" pairs: "(*bassDrum*, 4), (*bassDrum*, 4), (*bassDrum*, 4), (*bassDrum*, 4)". This is exactly the model adopted in the *Musical Platform Game* to specify drumbeats. By using the relative numeric values rather than seconds, the task of creating a drumbeat from a musical score is simpler and the drumbeat can be played at any speed, since this specification is independent from the *BPM*. The way the drumbeats are created and interpreted by the audio engine will be discussed alongside with the convention to specify complete levels: with the *MPGSong.plist* structure.

The *MPGSong.plist* structure

A level of *The Musical Platform Game* is fully specified in a property list, which is simply a collection of key-value pairs. Property lists are mostly used to store user settings or other types of data in an efficient and structured way. The major advantage to them is that they can be converted directly into *NSDictionary* objects, as seen in listing 9:

```
NSString *path = [[NSBundle mainBundle]
pathForResource:filename ofType:@"plist"];
NSDictionary *root = [NSDictionary
dictionaryWithContentsOfFile:path];
```

Listing 9. Conversion of a Property List into a NSDictionary.

All the information from the levels is stored and managed by *MPGSong* objects. However, at any given time there can only be one *MPGSong* instance. This instance manages the level that is being played and it is stored by the *currentSong* property of the *MPGGameManager* class. When a new level is going to start, the property list that contains the level specification will be loaded and a new *MPGSong* object will be created. The structure of the property lists will be explained with the level shown in figure 10:

Figure 10. MPGSampleLevel.plist

The structure of the levels is invariable, as figure 10 shows. Every property list should have three main parts:

- General information
- Instrumental part
- Drum part.

The general information part consists of two fields: the name of the level and the *BPM*. The name is necessary to uniquely identify each level and thus be able to distinguish them during the execution of the game. The *BPM* is crucial for the subsequent calculations and to perform the audio synchronization. It corresponds to the speed of the instrumental part, which will be explained next. Note that the *BPM* could also be calculated by analysing the audio file, but that would require the implementation of a *BPM* calculation algorithm, which is beyond the scope of this project. Therefore, the *BPM* has to be known beforehand and specified in the property list.

The instrumental part specifies all the information the audio engine needs in order to load and play the instrumental audio track. The *filename* and *extension* correspond to the audio file, which is stored in the application bundle. If the file name were "*Song1.caf",* then these fields would be, respectively, "*Song1"* and "*caf"*. The remaining field, *lengthInBars*, is used for further calculations. It is a measure of the length of the track in *bars*. A *bar* is made up of four *beats*, which in this project correspond to four *quarter notes*.

The third and last part of the *MPGSong.plist* structure is the drum part where all the drumbeats are specified in a sequence. It is an array of blocks, which can be either *idle* blocks or *show* blocks. Idle blocks do not represent drumbeats. Instead, they are used for parts of the song where the instrumental track plays alone. In this version of the project it was used to introduce the levels, but in further implementations it could help to make tutorial levels, show statistics during the game, and so on. The only parameter needed to specify an idle block is the number of beats that it will last. This is stored in the *beats* parameter.

On the other hand, *show blocks* specify the actual drumbeats that will constitute the level. This kind of block is composed of the same elements as the *idle blocks*, that is, *type* and *beats*, plus an array called *drumPattern*. In this array a drumbeat is specified with the convention previously explained: as a series of *drum* and *length* pairs. In figure 10 only four items are shown, but the following four are a copy of the previous ones, making a total of eight items of length 2. Figure 11 shows the representation of the drumbeat from figure 10 in a staff:

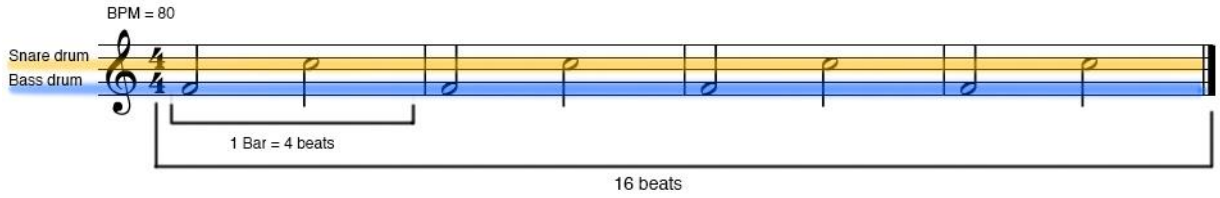


Figure 11. Drumbeat in staff notation

It is important to note that the total length of the drumbeat has to match the length specified in the *beats* field. As seen in figure 11, the total length of the drumbeat is 16 beats, composed of eight *half notes* (the length of a *half note* is twice that of a *quarter note*, that is, two beats).

With the *MPGSong.plist* structure the levels are easily created and they are a direct representation of musical scores. Creating new drumbeats is just a matter of adding or removing elements from the array and adjusting the lengths of the notes. The next step for the game is to translate this specification into a structure understandable by the device. The conversion is done in the *loadSong* method from the *MPGSong* class, shown in appendix 3. This method converts the level into a timeline, which will be explained in the following lines.

The timeline

The convention adopted to internally represent the levels of the *Musical Platform Game* is the timeline. It is a list of different events organized chronologically, which are interpreted by the *render callback* function previously discussed in this section. Rather than using seconds as the time indicator of the timeline, a more accurate way is used: samples. Therefore, a level begins at sample zero and as the song progresses and the *currentSample* property of the *MPGSong* object is increased, the various timeline events will be triggered. The *TimelineEvent struct* type, as shown in listing 10, represents each timeline event:

```
typedef NS_ENUM(NSUInteger, TimelineEventType) {
    TimelineEventType_NULL,
    TimelineEventType_PlayNote,
    TimelineEventType_IdleStart,
    TimelineEventType_ShowStart,
    TimelineEventType_CaptureStart
};


typedef NS_OPTIONS(NSUInteger, DrumsToBePlayed) {
    DrumsToBePlayed_NULL = 1 << 0,
    DrumsToBePlayedBassDrum = 1 << 1,
    DrumsToBePlayedSnareDrum = 1 << 2,
};


typedef struct {
    // For every event
    TimelineEventType type;
    UInt32 sample; // Event in time when it occurs

    // For IdleStart, ShowStart and CaptureStart events
    UInt32 length; // In samples
    UInt32 lengthInBeats;

    // For PlayNote events
    PlayerActionType playerAction;
    MPGDangerType danger;
    DrumsToBePlayed drums;

    } TimelineEvent;
```

Listing 10. The timeline events

Regardless of the type, all of the events share the *sample* field, which specifies the moment in time when it occurs. Depending on the particular type of the event, as listing 10 shows. the other fields may or may not be used The reason behind not using inheritance or another reuse mechanism for the non-shared fields is to improve the performance of the *render callback* function. Objective-C objects add a subtle overhead in accessing and managing them, and since the *render callback* is executed in a real-time thread, efficiency is crucial. Therefore, the timeline is a static C array composed of *TimelineEvent* elements.

There are four kinds of events, represented by the *TimelineEventType* enumeration type. Three specify the transition to another mode, that is, to the *idle*, *showing* or *capture mode*, and one is in charge of playing and recognizing drumbeats. The latter kind is called *PlayNote*, and it represents a drum associated with a specific moment of time. Depending on the mode, it will represent a drum that has to be played back during *show mode* or a drum that has to be tapped by the user during the *capture mode*. An important feature of *The Musical Platform Game* is that it is prepared for simultaneous drum playing and recognition, handled by the *drums* field.

Rather than using an array, the *drums* field uses a bitmask to store multiple values. Its type, *DrumsToBePlayed*, defines two possible values: *DrumsToBePlayedBassDrum* and *DrumsToBePlayedSnareDrum*. The encoding is performed with the bitwise *OR* (|) operator and the decoding with the bitwise *AND* (&). An example of this procedure is shown in listing 11:

```
DrumsToBePlayed drums = 0; // Initialization
drums |= DrumsToBePlayedBassDrum;

if (drums & DrumsToBePlayedBassDrum) { // Returns true
    NSLog(@"The bass drum bit is set!");
}
if (drums & DrumsToBePlayedSnareDrum) { // Returns false
    NSLog(@"The snare drum bit is set!");
}
```

Listing 11. Sample of bitwise encoding/decoding

Listing 11 shows how to set the *drums* variable to store a bass drum. Likewise, the procedure for storing also a snare drum would be to perform the bitwise *OR* operation with the *DrumsToBePlayedSnareDrum* value as well. With this approach, multiple drums can be stored and accessed efficiently in each *PlayNote* event. This also contributes to the efficiency of the *render callback* function, which is of great importance, as previously discussed in this section.

There is an important difference in how the drumbeats are handled during the *showing mode* and *capture* mode. In the first one, the drumbeat is played back for the player to memorize it. During the *capture mode*, the user has to repeat it as accurately as possible, and some visual events are triggered according to the drumbeat. First of all, each note has a danger associated with it, whether it is a biting dog, a rake or a falling can, if the user fails in playing the note the danger will hit the character. Moreover, if the user succeeds in playing the note, the character will avoid the danger by either jumping or bending down. As a consequence, each *PlayNote* event has a *danger* and a *playerAction* field that indicates the graphics engine which visual events to trigger.

When a new song is loaded, the *loadFromDictionary* method from the *MPGSong* class will translate the property list file into the timeline (refer to appendix 3 for the implementation of this method). Throughout the translation, a simple error checking mechanism ensures that the format of the property list is correct. Therefore, if there is a missing field or the values provided are not reasonable, the method will stop immediately. This is the process followed for the reading of the *general* and *instrumental* parts of the property list. However, the parsing of the *drum part* involves more calculations.

The first point to note is that although there are three kinds of blocks: *idle, show* and *capture*, a *drum part* cannot include *capture* blocks. The reason is that every *capture* block relates to a *show* block, in a way that it is an exact copy. The difference is that during the *showing mode* the notes are played and during the *capture mode* the same notes are recognized. Therefore there is no need of specifying *capture* blocks in the property list file. Instead, the *loadFromDictionary* method automatically creates the *capture* blocks from the *show* blocks.

The timeline is stored in the *events* private property, which is a static C-array of *TimelineEvent* elements. Unlike Objective-C mutable arrays, the size of static arrays has to be known prior to initialization and the memory explicitly allocated. That is the first part of the *drum part* creation. The *numEvents* property is calculated and the memory for the *events* property is allocated. In order to understand the process of translating the property list into the timeline, the equivalent timeline for the property list file from figure 10 is shown in figure 12:

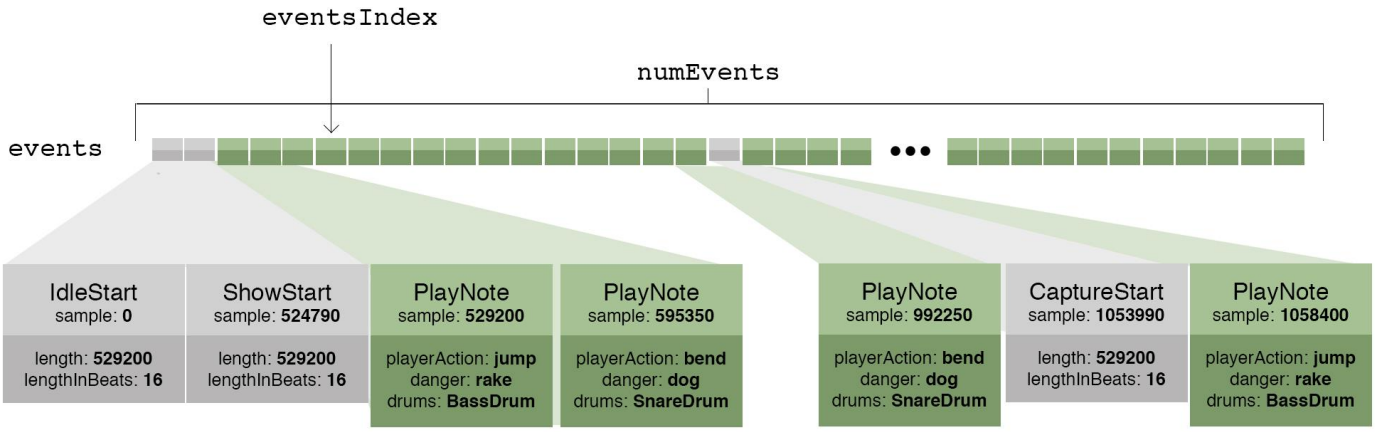


Figure 12. Graphic representation of the timeline

Figure 12 shows three properties of the *MPGSong* class. The *events* property is the pointer to the *TimelineEvent* array and *numEvents* simply specifies its size. The *eventsIndex* property points to the next event that will be processed.

Once the memory for the array is allocated, the *loadFromDictionary* method starts parsing the *drum part*. As it goes through the items, the *TimelineEvent* elements are created and stored in the array. This process is self-explanatory, and can be found in appendix 3. However, the way of calculating the *sample* value of the events requires some explanation.

The *sample* value is expressed with absolute timing, whereas the notes from the property list use relative values, following the number convention from table 3. In order to perform this conversion two crucial values are used: the *BPM* and the *sample rate*. The first step is to calculate the length in seconds of a *quarter note*, which is intimately related to the *BPM.* The way of calculating it has already been presented, and it is obtained with the following formula: 60 / *BPM*.

3.3 The graphics engine

Sprite Kit is a graphics framework for 2-D games announced in 2013 at The Apple Worldwide Developers Conference (WWDC). This shows the big impact the game industry is having in the App Store, since it was the first time Apple released a framework made specifically for games. The release of Sprite Kit was a giant leap for the growing indie developer community, which did not have native support to make their games and had to use third-party libraries. Now, games are not only for big companies producing triple A's (Triple A is a term used for games with a big budget and impact in the industry, involving many developers and usually strong advertising campaigns, examples of AAA games are *"Grand Theft Auto V"*, *"Assassin's Creed"* and *"Battlefield 3"*).

The main advantage of Sprite Kit and an important reason why it is the graphics engine behind *The Musical Platform Game* is that it runs natively on Apple devices. This implies a high efficiency in the graphics processing and ensures compatibility in further iOS versions. With other third-party libraries such as Cocos2D, the release of a new version of the operating system could make a game crash in thousands of devices with no apparent reason. However, as Apple supports Sprite Kit, it is Apple itself who has to fix the incompatibilities and bugs that can arise.

The nature of this project, a musical game that requires an extremely high accuracy in the audio processing, made also Sprite Kit the best choice. The low-level sound APIs required for this game, as previously discussed in section 3.2.2, are native and are written in Objective-C. Sprite Kit is a native library as well, so the coupling of both could be done with ease.

The visual content is structured in scenes which render their content to an *SKView* object. However, only one scene can be presented at a given time. The scenes are instances of the *SKScene* class, and all of its elements are organized in a hierarchical way. Every visual element inherits from the *SKNode* class; hence this organization is a tree of *SKNodes*. [9,10.]

The Musical Platform Game has three different kinds of scenes, each of them implemented as an *SKScene* subclass:

- *MPGMenuScene:* The main menu scene from where the levels are loaded.
- *MPGSongScene*: The scene that handles the actual gameplay. When a level is loaded, an *MPGSongScene* will be created and all of the resources loaded.
- *MPGLoadingScene*: The scene is used as a transition when loading the resources for the next *MPGSongScene* to show.

Inside the *MPGSongScene*, the content is organized logically in layers. That is, an initial *SKNode* tree is created and each node will hold graphical elements according to its depth from the camera. This structure is shown in figure 13:

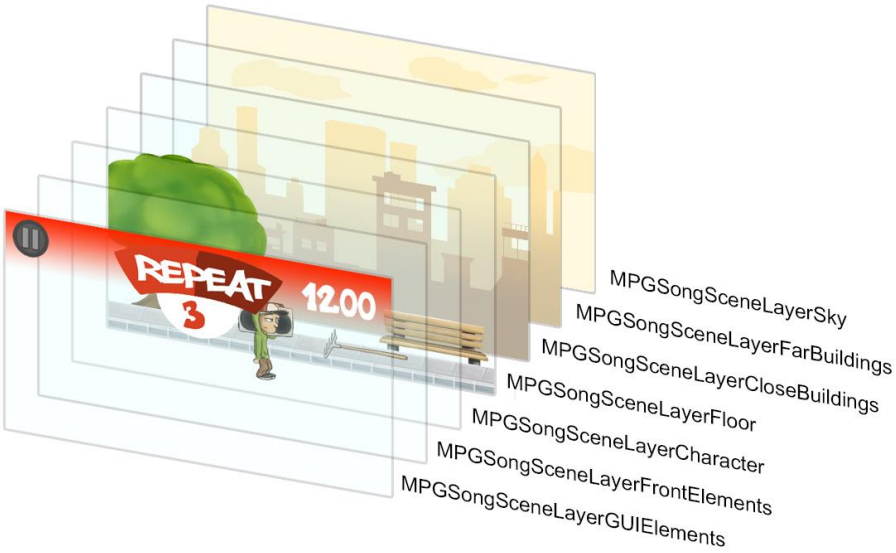


Figure 13. The layers of the *MPGSongScene* class

As seen in figure 13, the *MPGSongSceneLayerSky* layer is the farthest from the camera and the *MPGSongSceneLayerGUIElements* layer is the closest. The latter one contains the GUI, because it has to be on top of everything else. With this approach, placing a new graphical element in the game can be easily done.

4 Results

After stating the purpose and goals of this project, developed a game design process and finally proceeded with the implementation, the first stable version was created. The main achievement of this version is that it constitutes a neat and stable core for the game mechanics and a reliable starting point for further development of this project. The audio synchronization system, which took a significant part of the overall development, was tested and proven to work properly in numerous situations. It is reliable when phone interruptions occur and it is completely independent from the graphics engine. A drop in the FPS or a graphics overload will not affect it.

However, user feedback has shown some leaks that have to be improved in further versions. There was a big gap in the gaming experience between players with a musical background and with the players without it. The players that were in the first case could quickly grasp the game mechanics, and the feedback was generally positive. On the other hand, the players without musical skills found it really hard to play and felt frustrated with the game too quickly. A better calibration of the difficulty will have to be done, in order to target a bigger number of potential players.

In order to take this project a step further and proceed with publication, there are some improvements that can be done. First of all, there has to be a much stronger visual feedback to the user whenever he or she succeeds or fails. This includes showing the score panel changing and even making the main character react to the different events, such as looking at the user to recriminate him.

Moreover, it is important to improve the rewarding system of the game. Only increasing and decreasing the score panel is not enough to keep the player engaged and motivated with the game. A good approach would be to add a counter of the total dangers he or she has to avoid in the particular level and show how many of them he has surpassed so far. This will increase the motivation of the player, as he or she will see the percentage of dangers he avoided and will be more rewarded as he improves the ratio. Another method to improve the rewarding system would be to create a “rage mode” which gets activated when the user has successfully surpassed a certain number of dangers without failing. During this mode, the score would be increase at a greater rate and the mode would stay as long as the winning streak continues.

5 Conclusions

The approach of assuming all the roles in the game development process had some difficulties and some benefits, which should be taken into consideration for a further development of this project. By centralizing these roles, the transitions between the different phases, such as going from the requirements analysis to implementation, was straightforward. There was no need for communication and understanding between different developers. Furthermore, changes in the initial specification or at any development stage could be carried out immediately. On the other hand, this approach involved a greater workload and some important benefits from teamwork were also missing, such as the contrasting of procedures and ideas.

In conclusion, the main goal of developing a video game from every role was achieved. Nonetheless, the game is still not ready to be published, as the game experience can still be improved. The initial purpose of this project was to provide a learning tool to improve the rhythm skills of the player, and it was achieved for a segment of potential players. However, the game has to be adapted in order to reach a wider segment of the population. Taking all of this into account, the current version can be used as a robust starting point for the development of the final product, which could eventually be published in the market.

References

1 Ford E. ‘Cytus’ Review – Beauty In Simplicity [online]; January 2012. URL: http://toucharcade.com/2012/01/18/cytus-review/. Accessed 18 June 2014.

2 Musgrave S. ‘Audio Ninja’ Review – Simple, Short, but Satisfying Enough [online]; September 2013. URL: http://toucharcade.com/2013/09/13/audio-ninja-review/. Accessed 18 June 2014.

3 Apple Inc. System Sound Services Reference [online]. Cupertino, United States: Apple Inc.; August 2013. URL: https://developer.apple.com/library/ios/documentation/AudioToolbox/Reference/SystemSoundServicesReference/SystemSoundServicesReference.pdf. Accessed 17 April 2014.

4 Apple Inc. AVAudioPlayer Class Reference [online]. Cupertino, United States: Apple Inc.; February 2014. URL: https://developer.apple.com/library/ios/documentation/AVFoundation/Reference/AVAudioPlayerClassReference/AVAudioPlayerClassReference.pdf. Accessed 17 April 2014.

5 Grisby D. Tutorial: Easy Audio Playback With AVAudioPlayer [online]; August 2009. URL: http://mobileorchard.com/easy-audio-playback-with-avaudioplayer. Accessed 18 April 2014.

6 Apple Inc. Audio Unit Hosting Guide for iOS [online]. Cupertino, United States: Apple Inc.; September 2010. URL: https://developer.apple.com/library/ios/documentation/MusicAudio/Conceptual/AudioUnitHostingGuide_iOS/AudioUnitHostingGuideForiOS.pdf. Accessed 18 April 2014.

7 Adamson C. Learning Core audio: a hands-on guide to audio programming for Mac and iOS. Indiana, United States: Pearson; 2012.

8 Apple Inc. AVAudioSession Class Reference [online]. Cupertino, United States: Apple Inc.; March 2014. URL: https://developer.apple.com/library/ios/documentation/AVFoundation/Reference/AVAudioSession_ClassReference/AVAudioSession_ClassReference.pdf. Accessed 18 April 2014.

9 Apple Inc. Sprite Kit Programming Guide [online]. Cupertino, United States: Apple Inc.; February 2014. URL: https://developer.apple.com/Library/ios/documentation/GraphicsAnimation/Conceptual/SpriteKit_PG/SpriteKit_PG.pdf. Accessed 1 May 2014.

Appendix 1. Game screenshots


REPEAT
12
10


REPEAT
3
1200

Appendix 2. The *MPGAudioManager* class

```
- (id) init {
    if (self = [super init]) {
        // This is only called once in the app's life cycle
        // clear the mSoundBuffer struct
        memset(&mSoundBuffer, 0, sizeof(mSoundBuffer));

        RCgameIsOn = NO;
        [self setupAudioSession];
        [self initializeAUGraph];
        [self loadDrumsetSampler];

        [self startAUGraph];
    }

    return self;
}


- (BOOL) setupAudioSession {

    NSError *audioSessionError = nil;

    // Configure the audio session
    AVAudioSession *sessionInstance = [AVAudioSession sharedInstance];

    // our default category -- we change this for conversion and
playback appropriately
    [sessionInstance setCategory:AVAudioSessionCategoryPlayback
error:&audioSessionError];
    if (audioSessionError != nil) {NSLog (@"Error setting audio
session category."); return NO;}

    NSTimeInterval bufferDuration = .005;
    [sessionInstance setPreferredIOBufferDuration:bufferDuration
error:&audioSessionError];
    if (audioSessionError != nil) {NSLog (@"Error setting audio
session buffer duration."); return NO;}

    double hwSampleRate = 44100.0;
    [sessionInstance setPreferredSampleRate:hwSampleRate
error:&audioSessionError];
    if (audioSessionError != nil) {NSLog (@"Error setting audio
session category."); return NO;}

    // activate the audio session
    [sessionInstance setActive:YES error:&audioSessionError];
    if (audioSessionError != nil) {NSLog (@"Error activating the audio
session."); return NO;}

    // Obtain the actual hardware sample rate and store it for later
use in the audio processing graph.
    self.graphSampleRate = [sessionInstance sampleRate];
    sampleRate = self.graphSampleRate;

    return YES;
}
```

```
- (void)initializeAUGraph
{
    // *************************************
    // ****** Creating the AUGraph **********
    // *************************************

     OSStatus result = noErr;
    AUNode samplerNode, mixerNode, ioNode;
    CAStreamBasicDescription desc;

    // ***** Create a new AUGraph *****
     result = NewAUGraph(&_processingGraph);
    NSCAssert2 (result == noErr, @"Unable to create an AUGraph object.
Error code: %d '%.4s'", (int) result, (const char *)&result);

    // ***** Create the AudioComponentDescriptions for the AUs we want
in the graph *****

    // sampler unit
    CAComponentDescription sampler_desc(kAudioUnitType_MusicDevice,
kAudioUnitSubType_Sampler, kAudioUnitManufacturer_Apple);
    CAShowComponentDescription(&sampler_desc);

    // output unit
     CAComponentDescription io_desc(kAudioUnitType_Output,
kAudioUnitSubType_RemoteIO, kAudioUnitManufacturer_Apple);
    CAShowComponentDescription(&io_desc);

    // multichannel mixer unit
     CAComponentDescription mixer_desc(kAudioUnitType_Mixer,
kAudioUnitSubType_MultiChannelMixer, kAudioUnitManufacturer_Apple);
    CAShowComponentDescription(&mixer_desc);


    // ***** Create the nodes for the audio units *****

     result = AUGraphAddNode(_processingGraph, &io_desc, &ioNode);
    NSCAssert2 (result == noErr, @"Unable to add the IO unit to the
audio processing graph. Error code: %d '%.4s'", (int) result, (const
char *)&result);

     result = AUGraphAddNode(_processingGraph, &mixer_desc, &mixerNode
);
    NSCAssert2 (result == noErr, @"Unable to add the Mixer unit to the
audio processing graph. Error code: %d '%.4s'", (int) result, (const
char *)&result);

     result = AUGraphAddNode(_processingGraph, &sampler_desc,
&samplerNode );
    NSCAssert2 (result == noErr, @"Unable to add the Sampler unit to
the audio processing graph. Error code: %d '%.4s'", (int) result,
(const char *)&result);

    // ***** Open the graph. AudioUnits are open but not initialized
(no resource allocation occurs here) *****
     result = AUGraphOpen(_processingGraph);
    NSCAssert2 (result == noErr, @"Unable to open the audio processing
graph. Error code: %d '%.4s'", (int) result, (const char *)&result);
```

```
    // ***** Make the connections between the nodes *****

    // Connect the Mixer Unit to the RemoteIO Unit
     result = AUGraphConnectNodeInput(_processingGraph, mixerNode, 0,
ioNode, 0);
    NSCAssert2 (result == noErr, @"Unable to interconnect the nodes in
the audio processing graph. Error code: %d '%.4s'", (int) result,
(const char *)&result);

    // Connect the Sampler unit to the Mixer Unit
     result = AUGraphConnectNodeInput (_processingGraph, samplerNode,
0, mixerNode, 1);
    NSCAssert2 (result == noErr, @"Unable to interconnect the nodes in
the audio processing graph. Error code: %d '%.4s'", (int) result,
(const char *)&result);

    // ***** Obtain the references to the audio units *****
    // Obtain a reference to the Mixer unit from its node
     result = AUGraphNodeInfo(_processingGraph, mixerNode, NULL,
&_mixerUnit);
    NSCAssert2 (result == noErr, @"Unable to obtain a reference to the
Mixer unit. Error code: %d '%.4s'", (int) result, (const char
*)&result);

     // Obtain a reference to the Sampler unit from its node
     result = AUGraphNodeInfo (_processingGraph, samplerNode, 0,
&_samplerUnit);
    NSCAssert2 (result == noErr, @"Unable to obtain a reference to the
Sampler unit. Error code: %d '%.4s'", (int) result, (const char
*)&result);

     // Obtain a reference to the I/O unit from its node
     result = AUGraphNodeInfo (_processingGraph, ioNode, 0, &_ioUnit);
    NSCAssert2 (result == noErr, @"Unable to obtain a reference to the
RemoteIO unit. Error code: %d '%.4s'", (int) result, (const char
*)&result);

    // ************************************
    // *** Configuring the Render Callback **
    // ************************************

    // set bus count
     UInt32 numbuses = 2;
     UInt32 size = sizeof(numbuses);

    result = AudioUnitSetProperty(_mixerUnit,
kAudioUnitProperty_ElementCount, kAudioUnitScope_Input, 0, &numbuses,
sizeof(UInt32));
    NSCAssert2 (result == noErr, @"Unable to set an AudioUnits'
property. Error code: %d '%.4s'", (int) result, (const char
*)&result);


    // Set up render callback struct
    AURenderCallbackStruct rcbs;
    rcbs.inputProc = &renderInput;
    rcbs.inputProcRefCon = mSoundBuffer;

    printf("set kAudioUnitProperty_SetRenderCallback\n");
```

```
    // Set the callback for the mixer node's input 0
    result = AUGraphSetNodeInputCallback(_processingGraph, mixerNode,
0, &rcbs);
    NSCAssert2 (result == noErr, @"Unable to set an AudioUnits' render
callback. Error code: %d '%.4s'", (int) result, (const char
*)&result);

    // set input stream format to what we want
    printf("get kAudioUnitProperty_StreamFormat\n");
    size = sizeof(desc);
    result = AudioUnitGetProperty(_mixerUnit,
kAudioUnitProperty_StreamFormat, kAudioUnitScope_Input, 0, &desc,
&size);
    NSCAssert2 (result == noErr, @"Unable to get an AudioUnits'
property. Error code: %d '%.4s'", (int) result, (const char
*)&result);

    desc.ChangeNumberChannels(2, false);
    desc.mSampleRate = self.graphSampleRate;

    printf("set kAudioUnitProperty_StreamFormat\n");
    result = AudioUnitSetProperty(_mixerUnit,
kAudioUnitProperty_StreamFormat, kAudioUnitScope_Input, 0, &desc,
sizeof(desc));
    NSCAssert2 (result == noErr, @"Unable to set an AudioUnits'
property. Error code: %d '%.4s'", (int) result, (const char
*)&result);


    // set output stream format to what we want
    printf("get kAudioUnitProperty_StreamFormat\n");

    result = AudioUnitGetProperty(_mixerUnit,
kAudioUnitProperty_StreamFormat, kAudioUnitScope_Output, 0, &desc,
&size);
    NSCAssert2 (result == noErr, @"Unable to get an AudioUnits'
property. Error code: %d '%.4s'", (int) result, (const char
*)&result);


    desc.ChangeNumberChannels(2, false);
    desc.mSampleRate = self.graphSampleRate;

    printf("set kAudioUnitProperty_StreamFormat\n");

    result = AudioUnitSetProperty(_mixerUnit,
kAudioUnitProperty_StreamFormat, kAudioUnitScope_Output, 0, &desc,
sizeof(desc));
    NSCAssert2 (result == noErr, @"Unable to set an AudioUnits'
property. Error code: %d '%.4s'", (int) result, (const char
*)&result);

    RCSampler = &_samplerUnit;
    printf("AUGraphInitialize\n");

    // now that we've set everything up we can initialize the graph,
this will also validate the connections
    result = AUGraphInitialize(_processingGraph);
```

```
    NSCAssert2 (result == noErr, @"Unable to initialize the processing
graph. Error code: %d '%.4s'", (int) result, (const char *)&result);


    [self setInputVolume:0 value:1.0];
    [self setInputVolume:1 value:15.0];

    CAShow(_processingGraph);
}



static OSStatus renderInput(void *inRefCon, AudioUnitRenderActionFlags
*ioActionFlags, const AudioTimeStamp *inTimeStamp, UInt32 inBusNumber,
UInt32 inNumberFrames, AudioBufferList *ioData)
{
    AudioUnitSampleType *outA = (AudioUnitSampleType *)ioData-
>mBuffers[0].mData; // output audio buffer for L channel
    AudioUnitSampleType *outB = (AudioUnitSampleType *)ioData-
>mBuffers[1].mData; // output audio buffer for R channel

    if (!RCgameIsOn) {
        *ioActionFlags |= kAudioUnitRenderAction_OutputIsSilence;
        memset(outA, 0, inNumberFrames);
        memset(outB, 0, inNumberFrames);
    }
    else
    {
        // Game is on
        SoundBufferPtr sndbuf = (SoundBufferPtr)inRefCon;
        UInt32 sample = sndbuf[inBusNumber].sampleNum;      // frame
number to start from
        UInt32 bufSamples = sndbuf[inBusNumber].numFrames;  // total
number of frames in the sound buffer
        AudioUnitSampleType *in = sndbuf[inBusNumber].data; // audio
data buffer


        if (RCcurrentSong.currentSample >= RCcurrentSong.totalSamples)
{
            // The song is finished, dont output anything
            RCcurrentSong.songState = SongStateType_EndedPlaying;

            [[MPGGameManager sharedManager] goToMenu];
            *ioActionFlags |= kAudioUnitRenderAction_OutputIsSilence;
            memset(outA, 0, inNumberFrames);
            memset(outB, 0, inNumberFrames);
        } else {
            for (UInt32 i = 0; i < inNumberFrames; ++i) {
                [RCcurrentSong processCurrentSample];

                // Play the instrumental part
                if (0 == inBusNumber) {
                    outA[i] = in[sample++];
                    outB[i] = in[sample];
                } else {
                    outA[i] = in[sample++];
                    outB[i] = in[sample];
                }
```

```
                // Loop the instrumental part
                if (sample > bufSamples) {
                    sample = 0;
                }

                RCcurrentSong.currentSample++;
            }
            sndbuf[inBusNumber].sampleNum = sample;
        }
    }

    return noErr;
}
```

Appendix 3. The *MPGSong* class

Public interface (MPGSong.h)

```
@interface MPGSong : NSObject
@property NSString *songName;
@property NSUInteger bpm;
@property Float64 sampleRate; // For the callback function
@property SongStateType songState;

@property NSString *instrumentalURL;

@property NSUInteger totalSamples;
@property NSUInteger currentSample;

@property NSInteger score;

// For the triggering of the next event
@property TimelineEvent *runningEvent;
@property BOOL playerSucceededRunningEvent;
//@property BOOL capturedNoteWasSuccesful; // If the running event has
been evaluated, the result is stored here

// Used for calculations in the scene
@property NSUInteger startingSampleForCurrentMode;
@property NSUInteger lengthOfCurrentMode;
@property NSUInteger lengthInBeatsOfCurrentMode;

@property TimelineEvent *nextDangerToSpawn;

@property NSTimeInterval showingOffset;
@property NSTimeInterval capturingOffset;


@property NSTimeInterval quarterNoteTime;

- (id)init;

- (void)nextDanger;

- (void)deleteSong;
- (BOOL) loadSong:(NSString *)filename;
- (void) processCurrentSample;


@end
```

Private interface (MPGSong.m)

```
@interface MPGSong ()

@property NSUInteger instrumentalLengthInBars;
@property NSUInteger numEvents;

@property NSUInteger eventsIndex;
@property TimelineEvent *events;

@property NSUInteger nextDangerIndex;
```

```
@property NSInteger triggerInSample;
@property NSInteger triggerOutSample;

- (void)initializeFields;
@end
```

Method implementations

```
- (BOOL)loadFromDictionary:(NSDictionary *)root {
    Float64 sampleRate = [[MPGGameManager sharedManager]
audioManager].graphSampleRate;

    // ****************************************
    // **** General information part reading *****
    // ****************************************

    _songName = (NSString *)[root objectForKey:@"name"];
    if (_songName == nil) return NO;

    _bpm = [[root objectForKey:@"bpm"] integerValue];
    if (_bpm == 0) return NO;


    // ****************************************
    // ****** Instrumental part reading **********
    // ****************************************

    NSDictionary *instrumentalPart = (NSDictionary *)[root
objectForKey:@"instrumentalPart"];
    if (instrumentalPart == nil) return NO;

    _instrumentalLengthInBars = [[instrumentalPart
objectForKey:@"lenghtInBars"] integerValue];
    if (_instrumentalLengthInBars == 0) return NO;

    NSString *filename = [instrumentalPart objectForKey:@"fileName"];
    if (filename == nil) return NO;
    NSString *extension = [instrumentalPart
objectForKey:@"extension"];
    if (extension == nil) return NO;

    _instrumentalURL = [[NSBundle mainBundle] pathForResource:filename
ofType:extension];
    if (_instrumentalURL == nil) return NO;

    // ****************************************
    // ********** Drum part reading **************
    // ****************************************

    Float64 quarterNoteTime = 60000.0 / _bpm;
    self.quarterNoteTime = quarterNoteTime;
    NSLog(@"SongName: %@", _songName);
    NSLog(@"Bpm: %d", (unsigned int)_bpm);
    NSLog(@"instrumentalLengthInBars: %d", (unsigned
int)_instrumentalLengthInBars);
    NSLog(@"DrumPart: ");
```

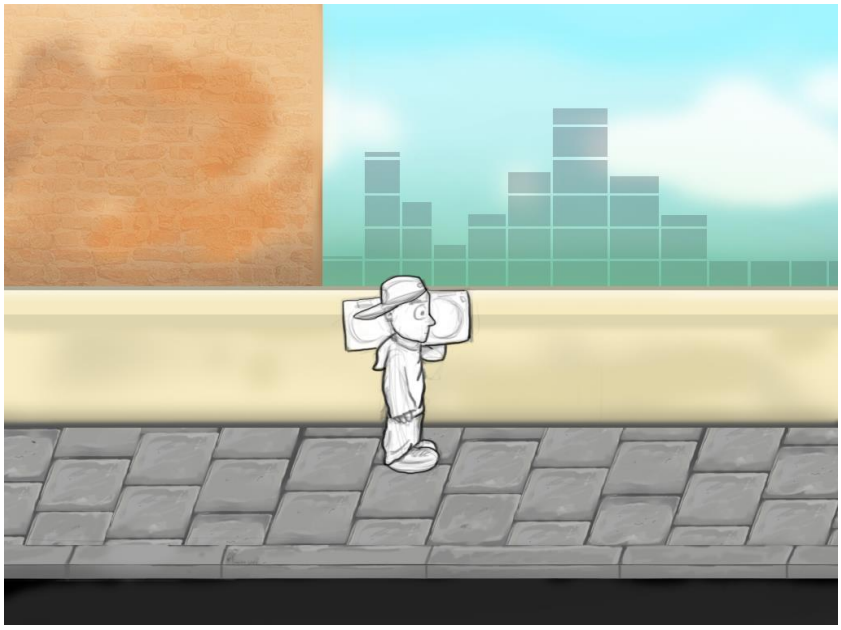
```
    // Count the number of notes that will be created
    NSUInteger eventsCount = 0;
    NSArray *drumPart = (NSArray *)[root objectForKey:@"drumPart"];
    if (drumPart == nil) return NO;
    for (NSDictionary* block in drumPart) {
        NSString *type = (NSString *)[block objectForKey:@"type"];
        if ([type isEqualToString:@"idle"]) {
            eventsCount++;
        } else if ([type isEqualToString:@"show"]) {
            NSArray *drumPattern = [block
objectForKey:@"drumPattern"];
            if (drumPattern == nil) return NO;
            eventsCount += (drumPattern.count + 1) * 2; // Reserves
space for the Capture block that follows
        } else {
            return NO;
        }
    }

    // Allocate space for the events
    _numEvents = eventsCount;
    _events = calloc(eventsCount, sizeof(TimelineEvent));

    NSLog(@"numEvents: %i", (unsigned int)_numEvents);


    // Creation of the timeline
    NSUInteger numPlayNoteEvents = 0;
    UInt32 sampleNum = 0;
    UInt32 eventsIndex = 0;
    for (NSDictionary *block in drumPart) {
        NSString *type = (NSString *)[block objectForKey:@"type"];
        if (type == nil) return NO;
        if ([type isEqualToString:@"idle"]) {
            TimelineEvent event;
            event.type = TimelineEventType_IdleStart;
            event.sample = sampleNum;
            NSUInteger lenghtInBeats = [[block objectForKey:@"beats"]
integerValue];
            if (lenghtInBeats == 0) return NO;
            event.lengthInBeats = lenghtInBeats;
            event.length = lenghtInBeats * quarterNoteTime *
(sampleRate/1000);

            /*
            NSLog(@"Inserting event!");
            NSLog(@"-type: IdleStart");
            NSLog(@"-sample: %d", (unsigned int)event.sample);*/
            _events[eventsIndex] = event;
            eventsIndex++;

            UInt32 beats = [[block objectForKey:@"beats"]
integerValue];
            if (beats == 0) return NO;
            sampleNum += beats * quarterNoteTime * (sampleRate/1000);
        } else if ([type isEqualToString:@"show"]) {
            for (int i = 0; i < 2; i++) {
                // Create the show and capture blocks
                //NSLog(@"Inserting event!");
```

```
                TimelineEvent event;
                if (i == 0) {
                    event.type = TimelineEventType_ShowStart;
                    //NSLog(@"-type: ShowStart");
                } else {
                    event.type = TimelineEventType_CaptureStart;
                    //NSLog(@"-type: CaptureStart");
                }
                event.sample = MAX(0, sampleNum - self.showingOffset *
(sampleRate/1000));


                NSUInteger lenghtInBeats = [[block
objectForKey:@"beats"] integerValue];
                if (lenghtInBeats == 0) return NO;
                // Watch out for errors on this calculation
                event.lengthInBeats = lenghtInBeats;
                event.length = lenghtInBeats * quarterNoteTime *
(sampleRate/1000);

                //NSLog(@"-sample: %d", (unsigned int)event.sample);
                _events[eventsIndex] = event;
                eventsIndex++;

                NSArray *drumPattern = [block
objectForKey:@"drumPattern"];
                for (NSDictionary *drumEvent in drumPattern) {
                    TimelineEvent event;
                    event.type = TimelineEventType_PlayNote;
                    event.sample = sampleNum;
                    event.danger = [MPGSong
dangerTypeForString:(NSString *)[drumEvent objectForKey:@"danger"]];
                    if (event.danger == MPGDangerType_NULL) return NO;

                    event.playerAction = [MPGSong
playerActionTypeForString:(NSString *)[drumEvent
objectForKey:@"playerAction"]];
                    if (event.playerAction == PlayerActionType_NULL)
return NO;

                    event.drums = 0;
                    event.length = 0;

                    NSLog(@"Inserting event!");
                    NSLog(@"-type: PlayNote");
                    NSLog(@"-sample: %d", (unsigned int)event.sample);

                    NSArray *drums = (NSArray *) [drumEvent
objectForKey:@"drums"];
                    if (drums == nil) return NO;
                    for (NSString *drum in drums) {
                        DrumsToBePlayed currentDrum = [MPGSong
drumForString:drum];
                        if (currentDrum == DrumsToBePlayed_NULL)
return NO;
                        event.drums |= currentDrum;
                    }

                    _events[eventsIndex] = event;
```

```
                    eventsIndex++;

                    Float64 length = [[drumEvent
objectForKey:@"length"] floatValue];
                    if (length == 0) return NO;
                    sampleNum += ((4 / length) * quarterNoteTime *
(sampleRate / 1000));
                    numPlayNoteEvents++;
                }
            }
        }
    }
    _totalSamples = sampleNum;

    _eventsIndex = 0;
    _currentSample = 0;
    _nextDangerIndex = 0;
    NSLog(@"About to call nextDanger for the first time!");
    [self nextDanger];

    NSLog(@"Total Samples: %i", (unsigned int)_totalSamples);

    return YES;
}
```

Appendix 4. Early mock-ups and artwork

Background concepts



Character concept evolution