PROCEDURAL MUSIC GENERATION AND ADAPTATION BASED ON GAME STATE

A Thesis

presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Timothey Adam

June 2014

© 2014
Timothey Adam
ALL RIGHTS RESERVED

COMMITTEE MEMBERSHIP

TITLE:	Procedural Music Generation and Adaptation Based on Game State
AUTHOR:	Timothey Adam
DATE SUBMITTED:	June 2014
COMMITTEE CHAIR:	Michael Haungs, Ph.D. Associate Professor of Computer Science
COMMITTEE MEMBER:	Foaad Khosmood, Ph.D. Assistant Professor of Computer Science
COMMITTEE MEMBER:	John Clements, Ph.D. Associate Professor of Computer Science

Abstract

Procedural Music Generation and Adaptation Based on Game State

Timothey Adam

Video game developers attempt to convey moods to emphasize their game's narrative. Events that occur within the game usually convey success or failure in some way meaningful to the story's progress. Ideally, when these events occur, the intended change in mood should be perceivable to the player. One way of doing so is to change the music. This requires musical tracks to represent many possible moods, states and game events. This can be very taxing on composers, and encoding the control flow (when to transition) of the tracks can prove to be tricky as well.

This thesis presents AUD.js, a system developed for procedural music generation for JavaScript-based web games. By taking input from game events, the system can create music corresponding to various Western perceptions of music mood. The system was trained with classic video game music. Game development students rated the mood of 80 pieces, after which statistical representations of those pieces were extracted and added into AUD.js. AUD.js can adapt its generated music to new sets of input parameters, thereby updating the perceived mood of the generated music at runtime.

We conducted A/B tests comparing static music, both composed and computer-generated, to dynamically adapting music. We find that AUD.js provides reasonably effective music for games, but that adaptiveness of the music does not necessarily improve player experience over composed music. By conducting a user study during Global Game Jam 2014, we also find that since AUD.js provides a software solution to music composition, it can be a useful tool for game music integration under time pressure.

Acknowledgments

We thank Dr. Zoë Wood and her CPE 476 students for their assistance in assessing the accuracy of AUD.js' output music mood.

We thank the 93 game development students who helped provide the music mood data for AUD.js' training.

We also thank all the members of the 5Alive, Anellu Moore, Bullet'space, heart4u, Phancy Adventures of Charles the Cat and Tricollide teams for helping evaluate the AUD.js system during the 2014 Global Game Jam.

Finally, we thank the creators of Brothers Chronosoff and Bullet'space for allowing us to use their work to evaluate the effectiveness of AUD.js.

Contents

List of Tables viii

List of Figures ix

1 Introduction 1

2 Background 4

2.1 Procedural Music Generation Techniques 4

2.1.1 Stochastic 4

2.1.2 Fractals 5

2.2 Audio Synthesizers 5

2.3 WebkitAudio 7

2.4 MIDI 7

2.5 Adaptive Music in Games 8

2.6 Music Mood Model 9

3 Usage 10

4 Implementation 15

4.1 Starting the generation process 15

4.2 UI and Instrumentation 16

4.3 Generation Process Control Flow 19

4.4 Percussion 21

4.5 Note Selection 22

4.5.1 Prevalence Arrays 22

4.5.2 Elimination Tables 22

4.6 Data Acquisition 24

4.7 Generation & Adaptation 25

4.8 Heuristics 26

4.8.1 Transitions 26

4.8.2 Pattern forwarding 26

4.8.3 Variation in repeats 26
4.8.4 Voice/Instrument limits 27
4.8.5 Hard cutoffs 27

5 Experiments & Results 28

5.1 Global Game Jam Developer Survey 28
5.1.1 Results 29
5.1.2 Analysis 29
5.2 Global Game Jam Player Survey 31
5.2.1 Results 31
5.2.2 Analysis 34
5.3 A/B Test 34
5.3.1 Results 35
5.3.2 Analysis 37
5.4 Mood Identification 38
5.4.1 Results 39
5.4.2 Analysis 40

6 Related Work 41

6.1 LucasArts' iMUSE 41
6.2 Mass Effect 2's Adaptive Music 41

7 Conclusion 43

8 Future Work 45

Bibliography 47

A Classic Game Music used as Training Set 49

B Heuristics used in AUD.js Implementation 52

List of Tables

3.1 AUD.js API description. 11

5.1 Developer survey results: Global Game Jam participants that used AUD.js 29

5.2 Developer survey results: Global Game Jam participants that used JavaScript, but chose not to use AUD.js. 30

5.3 GGJ player survey results: games with AUD.js. 32

5.4 GGJ player survey results: games without AUD.js. 32

List of Figures

2.1 A simple Markov chain, that shows two states: playing an 'E' or an 'A' note, and probabilities on the transitions indicating how likely it is for the Markov Chain to go to either to the same or a different note. Image from [15]. 5

2.2 A Koch curve, a simple two dimensional fractal. 6

2.3 Thayer Model of Mood, modeling human emotion in terms of stress and energy. 9

3.1 AUD.js software design. 1: The web browser on which AUD.js runs. 2: The audio synthesizer. 3: The AUD module itself, which contains the generation code and API. 10

3.2 A demo of AUD.js - a simple web page to interact with the generation and adaptation of AUD.js, using random seeds and random stress and energy values. 12

3.3 The code required to create the AUD.js demo web page. Very little JavaScript code is required for the generation, adaptation and playback when using AUD.js. 13

4.1 The UI for AUD.js' input, from AUD_ui.js. 16

4.2 AUD.js instrumentation readout, showing which voices are associated with percussion or a specific timbre. 17

4.3 AUD.js uses randomized waveforms with a smoothness factor tied to the stress level, and randomized ASDR envelopes. 18

4.4 The probabilities of snare hits and high hat hits occurring, as a function of the beat number in a measure. 21

4.5 Prevalence Arrays. 1: A prevalence array for low stress. 2: A prevalence array for high stress. 3: A prevalence array for low stress with notes lower than a 5% cutoff removed. 23

4.6 One of the 12 entries in an elimination table, showing the likelihood of an 'A' note being played with any other named note. 24

4.7 Rated stress/energy values of classic video game music. 25

5.1 A comparison of the results of the GGJ Player Survey. 33

5.2 The Brothers Chronosoff. 35
5.3 Bullet'space. 36
5.4 The results of all three versions of Brothers Chronosoff. 37
5.5 The results of all three versions of Bullet'space. 38
5.6 AUD.js mood estimation survey results, showing the input categories to AUD.js in red and the rated stress and energy in blue. 39

Chapter 1

Introduction

Procedurally generated music has been a topic sparsely researched but frequently revisited. Early work, like Iannis Xenakis' Illiac Suite [16], is mostly limited to simple probabilistic methods, such as Markov chains and probabilistic grammars [7]. Other areas of research include exploring topics from mathematics and artificial intelligence, like the use of fractals and genetic algorithms. For example, machine learning has been used to generate jazz music [10]. Some systems even combine multiple techniques, like genetic algorithms and Markov chains [2]. Recent research, like the work of David Cope, seeks to model creativity when generating music in an attempt to have computers generate pieces that sound as if they were composed by a human [4].

Procedural audio has previously been used in games, though not for generating full melodies, but rather to adapt composed music. Game audio needs to meet the burden of being pertinent and impactful towards gameplay, something procedurally generated audio rarely achieves [3]. Instances of procedural music in games are usually only to modify or adapt the composed music to avoid sounding repetitive when listened to many times. A solution for this problem is using complex logic to create the music, but very few projects have the budget to dedicate to engineers for these purposes. Current algorithms for procedural music generation typically also have the issue of being less appealing aesthetically than composed pieces. Unless the chosen system can interact on a raw signal or even a sample level, the music would probably not sound composed. This kind of low-level interaction is costly, both from a real-time performance and a financial point of view [5].

A game jam is a gathering of game developers for the purpose of planning, designing, and creating one or more games within a short span of time, usually ranging between 24 to 48 hours. During game jams, audio integration can be challenging. It is rare for a team attempting to create a game in the span of a few days to have a

dedicated composer. Even if the team does, due to the stressful nature of the event, coordination between the composers, audio designers, and the programming team can be sparse. This can lead to subpar audio if the team cannot find a way to dedicate a significant amount of time to audio development and integration. In addition, having a musician or sound designer on the team doesn't necessarily lead to good integration with the game's tempo, logic, or user input. An example of a solution to this problem is 2013's audiodraft.com "Create Music for Game Jam Team 2" contest, where participants from all over the world could submit music to be used in game jams [1]. Even if a Game Jam team has outstanding audio, that audio is likely static and unchangeable at their game's runtime. Trying to make the static music adaptive would also likely break the flow of the track, thereby detracting from the quality of the music experience. Truly adaptive music, which changes with the mood of the game, would require an exponentially larger commitment from the composers as well as careful planning of the game events. Those both can be problematic in a game jam setting.

This thesis presents AUD.js, a tool for procedural music generation based on an input 'mood', and capable of adapting the music to different moods at runtime. The system provides an API capable of generating 10 million different pieces of music due to seeding AUD.js' random number generator. Since AUD.js provides a software solution to music composition and integration into web application and games, it is capable of accelerating game development: instead of having a composer, a team could be content to call a few functions to provide their game with a soundtrack. This reduces pressure on developers, and the mood-based approach helps the system be applicable to multiple genres and types of games. The runtime adaptability of the AUD.js generated soundtracks means that the game's music can match game events in real-time. When something good or bad happens in-game, the audio can adapt to match in under a second, all the while still sounding like a coherent piece of music.

To verify AUD.js' effectiveness, we run a number of studies, including having developers participating in Global Game Jam use the system and an A/B test where players play either a version of a game with composed music or with AUD.js. From our studies we determine that though AUD.js audio quality is slightly less than composed music, it does serve its purpose when it comes to reducing pressure on developers in

game jam setting. This can be attributed to the simple API with a good range of expressiveness: all developers need to do to get their soundtrack is write two or three lines of code and AUD.js can handle the rest.

Chapter 2

Background

This chapter presents some general information that may be useful for completely understanding the later chapters.

2.1 Procedural Music Generation Techniques

This section briefly presents some techniques that have been successfully used for procedural music generation in the past, and will be relevant to AUD.js. For more details on methods for procedurally generating music, we recommend "Algorithmic Music Composition" by Jarvelainen [7].

2.1.1 Stochastic

A stochastic (random) composition algorithm uses probabilities to make all the necessary decisions, including which notes to play and when to play them. In essence, this is randomly generated music, though usually the programmer will put in hard limits (such as forcing the system to play at least one note every few seconds) and weighting (so the system knows which notes or rhythm patterns are more preferable). For example, a Markov chain-based system is a subset of stochastic algorithms (see figure 2.1). A Markov chain is a probabilistic state machine, meaning it takes into account some state (the previous note played, for instance) in making its probabilistic decisions. Each transition has a random chance of occurring, and when it does, some musical cue is added (like playing a specific note or waiting a certain amount of time) and the current state of the Markov chain is updated.

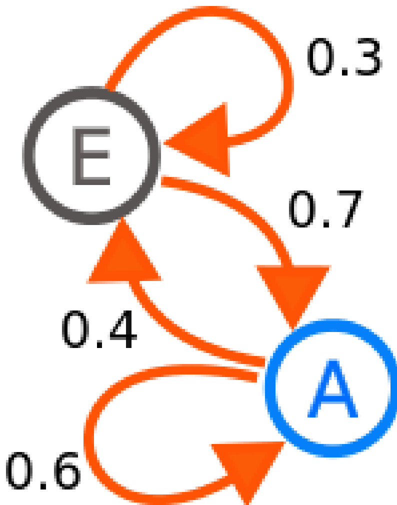

Figure 2.1: A simple Markov chain, that shows two states: playing an 'E' or an 'A' note, and probabilities on the transitions indicating how likely it is for the Markov Chain to go to either to the same or a different note. Image from [15].

2.1.2 Fractals

Fractal patterns are self-similar, meaning that if you look at the pattern at a different level, it will look similar (see figure 2.2). Self-similar patterns are very common, which make fractal patterns well suited to music. Fractals allow music to be created recursively, starting at a very high level with a large chunk of music, and then recurring on smaller chunks and refining it. For this project, fractal patterns were used to model a drum beat. Since percussion for music is typically a similar repeated set of drum hits, fractals can represent these similar patterns quite well.

2.2 Audio Synthesizers

A synthesizer is an electronic music instrument that creates sounds by using software to populate digital audio buffers. Audio can be seen as areas of high and low air pressure travelling through space. The frequency at which the air changes from higher to lower pressure is also the frequency of the audio. For example, playing the

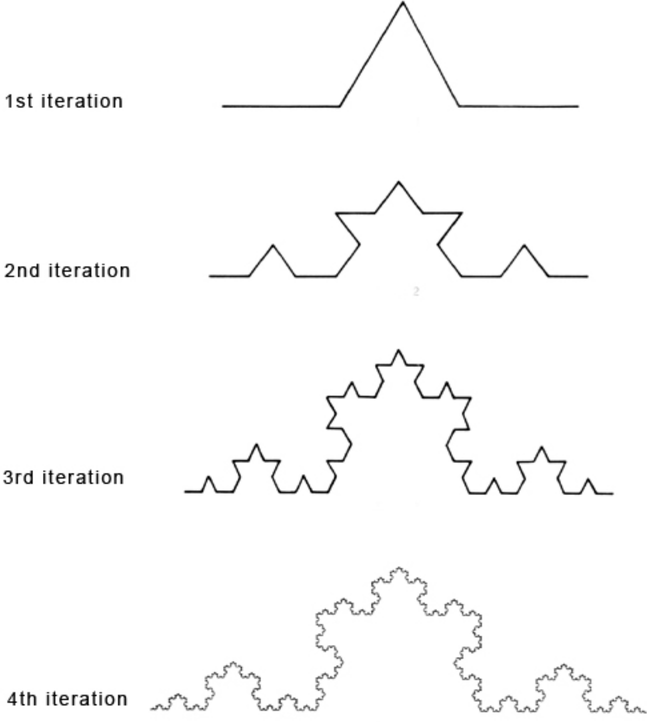

Figure 2.2: A Koch curve, a simple two dimensional fractal.

'A' note from the middle of a keyboard produces sound at 440 Hertz (oscillations per second). Digital audio production and capture works by causing or measuring air pressure differences, and storing those as samples in an audio buffer.

Another important aspect audio synthesizers need to take into account is timbre or sound texture. All sound sources have a timbre, and it's how humans can tell different voices or instruments apart - the same note played on a piano or a guitar or sung by a human still sound very different. The cause of this phenomenon is additional frequencies produced, even if only one frequency was requested: when you press a key on a keyboard, the corresponding frequency is not the only one you hear - you hear a number of high frequencies (overtones) as well. The difference between instruments' timbres lies in which overtones they produce. As such, when designing an audio synthesizer, you need to take into account which overtones you do and don't want to hear.

AUD.js has a custom audio synthesizer built on top of WebkitAudio, the implementation of which is discussed in chapter 4.

2.3 WebkitAudio

WebkitAudio (or Web Audio) is an application programming interface (API) for processing and creating audio in a browser using JavaScript [11]. Before WebkitAudio, browsers could only play sound through a plug-in like Flash or Quicktime. WebkitAudio and HTML5's <audio> tag remove that requirement. The programming interface has been designed in such a way to support many use cases, letting the programmer interact with audio data at many levels of abstraction. Now programmers can load in .mp3 files or .wav files directly into a browser and interact with them using JavaScript, for example playing them back, filtering them, or using the audio data to drive some other process (such as an audio visualizer). Alternatively, WebkitAudio allows programmers to generate music at low levels. Computer audio is comprised of samples: numbers representing the offset of speaker cones (the parts of speakers that move to create differences in air pressure, which we perceive as audio). Most music is recorded and played back at 44100 samples per second. Though that may seem like a lot, since each individual sample is small, the whole process doesn't take much computation time or memory. AUD.js uses WebkitAudio for the sample-per-sample generation capability. WebkitAudio will call the function .onaudioprocess whenever an audio buffer needs to be populated with samples, at which point AUD.js' synthesizer will add the appropriate samples to play back the generated music (see chapter 4 for more details).

2.4 MIDI

MIDI stands for Musical Instrument Digital Interface [6]. Developed in the 1980s, its primary purpose is to represent performance data. It logs events such as note-on's (key presses in the case of a keyboard), note-off's (key releases) or instrument changes. MIDI itself does not produce sound. As an example usage of MIDI, a human performer could use an instrument that logs their performance in the MIDI file format, so the exact key presses (if the human was using a keyboard) can be played back later. Another example usage of MIDI would be to hook up a list of notes representing a song to an audio replay system where the user could choose the instruments for the song. For the purposes of this thesis, MIDI files are used for training AUD.js to have

a sense of what classical video game music sounds like (see chapter 4 for more detail). Since MIDI files log which notes are being played when, it's straightforward to write a parser to read a MIDI file and determine how often each note is played and which notes are commonly played together. When humans listened to the MIDI files used for training, the pieces were played back (programmatically) with the appropriate piano sounds for all notes.

2.5 Adaptive Music in Games

Adaptive music in games comes in two major categories: re-sequencing and re-orchestration.

Sequencing is putting together sounds or groups of sounds into a larger chunk of sound (like a piece of music). Dynamic re-sequencing means adjusting when certain chunks of sounds start or stop. By adjusting when audio clips and loops are played or stopped, a composed track can start or stop when game events trigger.

As the name implies, orchestration is act of writing a piece of music for an orchestra. This includes choosing which instruments play which notes, and how each performer should play those notes. With respect to games, re-orchestration can mean adding or removing certain instruments from the mix (or removing entire layers). It could also mean emphasizing certain notes or instruments more, depending on the game state.

Examples of these techniques can be seen in many games. As an example, Mass Effect 2 (Bioware) uses both major techniques. It has different layers of audio, which turn on and off as game events occur, such as entering or exiting action sequences. Since the music exists in different layers, those layers can be put together in many different ways, like musical building blocks. The system also dynamically filters[1] the audio when the player's health is low or if there is ongoing dialog. When key events occur, additional short audio clips are sequenced [14].

[1] Audio filtering is amplifying or reducing specific parts of a music track while it's playing.

2.6 Music Mood Model

Previous work has examined the mood of music by analysis on several spectra, including loudness of the music, tempo, mode, harmonic complexity and variety of timbre [8]. Based on these qualities, the music can be placed in an "emotion space": a visual representation of human emotion. One such representation is the Russell mood model [12], which categorizes emotion in terms of two axes: arousal & valence. Nearly all mood models from psychological research we reviewed mapped onto two dimensions. Other terms that have been used for these axes include activity & pleasure or stress & energy in the Thayer model of mood [13]. The Thayer model of mood was chosen for this system as stress and energy are two easily understood concepts, and can be expressed in simply musical terms as harmony/dissonance (how well notes go together) and volume & speed (how short the interval between notes is and how loud each note is).

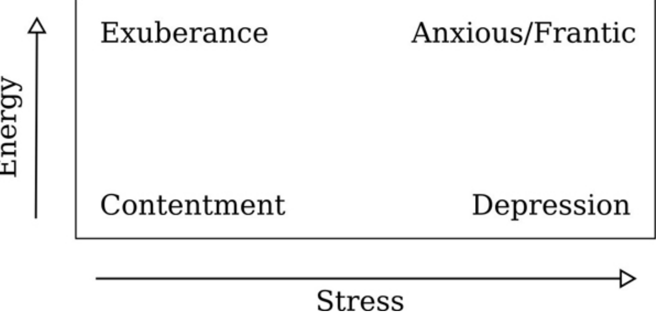

Figure 2.3: Thayer Model of Mood, modeling human emotion in terms of stress and energy.

Chapter 3

Usage

AUD.js is a browser-based framework built on WebkitAudio (see figure 3.1). This allows AUD.js's synthesizer (AUD_interface) to populate the audio buffers used by the browser to play sound, and manipulate that raw audio, allowing for real-time generation and adaptation.

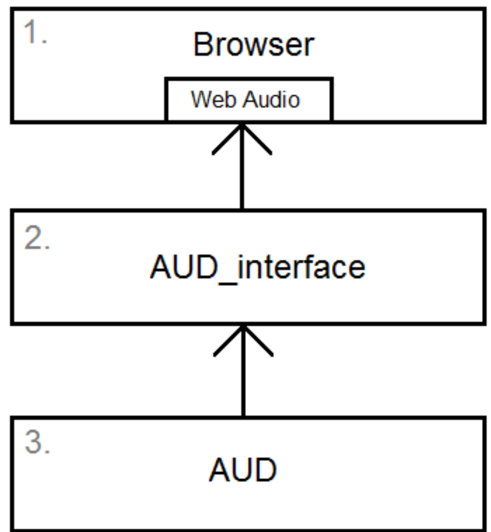

Figure 3.1: AUD.js software design. 1: The web browser on which AUD.js runs. 2: The audio synthesizer. 3: The AUD module itself, which contains the generation code and API.

AUD.js provides an API for developers to include the functionality of this system in their own applications. For a list of available function calls, please refer to table 3.1.

As an example, figure 3.2 shows a small web page that can be created with a very small amount of code, shown in figure 3.3. When the user clicks the "Generate" button, a random seed, stress and energy values are chosen, and a piece is generated.

generatePattern (stress, energy, numpatterns, patternrepeats, seed)	Starts the generation process from the beginning: all the parameters are reset and the piece thats playing gets overwritten and reset to the beginning of the new piece.
adaptPattern (stress, energy)	Adapts the current piece to a new mood by regenerating the entire piece, but not changing the length or the seed. The random number generator is reset, thereby ensure the same piece is regenerated with the new given mood. The net effect is that the piece keeps playing, except it will now have a perceivably different mood (provided the new stress and energy are sufficiently different to warrant a perceivable difference).
togglePause()	Pauses the song if its playing or resumes it from where it was paused.
togglePlay()	Resets the song to the beginning and either pauses it or resumes is depending on whether it was playing.
isPlaying()	Returns whether the piece is playing as a boolean value.
setVolume (newvolume)	Sets the volume of the systems outputted music to the new given value (it should be between 0 and 1, where 0 is muted and 1 is the maximum volume possible).

Table 3.1: AUD.js API description.

When the user clicks the "Adapt" button, the current piece is adapted to a different random stress and energy. If the variables hadn't been declared, but rather their values been directly inserted into the AUD method calls, the page would only need 8 lines of JavaScript code, excluding white space.

To integrate AUD.js into another web application, follow these steps:

1. Ensure that your html file includes any version of jquery and AUD.js as JavaScript source files before trying to call one of the API functions (see lines 3 and 4 in the example (fig 3.3)). AUD.js is a singleton object which initializes when you load it as a source file, meaning you don't need to do any additional initialization.

2. Make a call to *generatePattern*. You'll need to pass the stress and energy correlating to the starting mood to the system, along with a seed (which determines what 'song' will be playing) and a number of patterns and repeats of each pat-

AUD.js Minimalist Random Demo

Generate & play random AUD.js piece

Adapt to random stress/energy

Toggle Pause

Figure 3.2: A demo of AUD.js - a simple web page to interact with the generation and adaptation of AUD.js, using random seeds and random stress and energy values.

tern (see line 18 in the example). The number of patterns is the number of unique motives you'll hear in the generated piece whereas the repeats are how many times they'll be played before transitioning to the next pattern generated. For example, if you choose to have 3 unique patterns with 3 repeats each, the system will generate 9 32-tick chunks of music. Note that the music will not automatically start playing when you call *generatePattern*. You'll need to call *togglePlay* to start the music (see line 19 in the example).

3. If you need to pause the music (for example, if the game is paused), call *togglePause* in your code where you need the music to pause and unpause (see line 32 in the example).

4. If you need the music to adapt to a new mood (for example, if you entered combat or just defeated a boss), call *adaptPattern* with the new stress and energy values (see line 28 in the example).

5. If you need a completely different track (for example, if you enter a different area or game state), you'll need to make another call to *generatePattern*. Be sure to use a different seed so you get a unique melody: using the same seed guarantees that for similar stress and energy values, you will get the same melodies. As such, if you need to go back to an earlier piece of music, just call *generatePattern* with the seed used previously.

6. If you need to change the volume of the music at all (for example, if there's

```
<html>
    <head>
        <script type="text/javascript" src="jquery.min.js"> </script>
        <script type="text/javascript" src="aud.js"> </script>
        <script type="text/javascript">
            // Generates
            function GenerateRandom()
            {
                // numbers between zero and one
                var stress = Math.random();
                var energy = Math.random();

                var numpatterns = 20;
                var numrepeats = 2;
                // number between zero and one million
                var seed = Math.floor(Math.random() * 1000000);

                aud.generatePattern(stress, energy, numpatterns, numrepeats, seed);
                aud.togglePlay();
                // reduce the volume a little
                aud.setVolume(0.9);
            }
            function AdaptRandom()
            {
                var stress = Math.random();
                var energy = Math.random();

                aud.adaptPattern(stress, energy);
            }
            function TogglePause()
            {
                aud.togglePause();
            }
        </script>
        <title> AUD.js Demo </title>
    </head>
    <body>
        <h1>AUD.js Minimalist Random Demo</h1>
        <button onClick="GenerateRandom();"> Generate & play random AUD.js piece </button>
        <br />
        <button onClick="AdaptRandom();"> Adapt to random stress/energy </button>
        <br />
        <button onClick="TogglePause();"> Toggle Pause </button>

    </body>
</html>
```

Figure 3.3: The code required to create the AUD.js demo web page. Very little JavaScript code is required for the generation, adaptation and playback when using AUD.js.

dialog), call *setVolume* with the new volume level, which should be between 0 and 1 (see line 21 in the example).

Note that since AUD.js is dependent on WebkitAudio, it will only work on Chrome and Firefox. As such any applications with AUD.js run on other browsers will likely not function correctly.

To try out AUD.js, a demo is available at *http://timotheyadam.com/AUD/*.

Chapter 4

Implementation

AUD.js generates music by using probabilistic methods. Both rhythm (when percussion and notes play) and melody (which notes play) are chosen with a number of percentage chances that vary based on the input, particularly the mood.

The code for AUD.js can be found on github: *https://github.com/Trayus/AUD.*

4.1 Starting the generation process

Since AUD.js seeks to generate music based off of a representation of music mood, the generation process starts with the user entering five parameters (see figure 4.1).

1: Stress is the first component of the model of mood this system uses[1] and typically governs the level of harmony and dissonance of the music.

2: Energy is the second component of the model of mood this system uses[1], and typically governs the rhythm, tempo, and pace of the music.

3: The seed ensures a unique, re-creatable piece is generated as it seeds the random number generator which governs the random aspects of the generation process. Each unique seed will result in a unique sequence of random numbers used to choose notes in the melodies, thereby resulting in a unique piece.

4: The number of patterns indicates the number of unique 4-measure strings of notes that should be generated. Each measure contains up to 8 eighth notes.

5: The repeat of patterns indicates how many times a generated 4-measure string of notes should repeat. Longer repeat values will result in minor variation in the notes of the repeated patterns.

---

[1] See in general: Livingstone & Brown (2005)

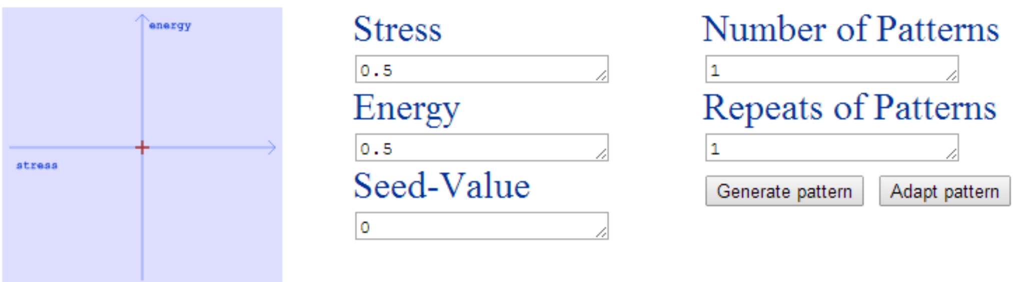

Figure 4.1: The UI for AUD.js' input, from AUD_ui.js.

4.2 UI and Instrumentation

AUD.js is a single JavaScript file, which can be integrated into other projects. However, for convenience a User Interface (UI) was developed with which the user can interact with AUD.js. AUD_ui.js has buttons that let the user call all the API functions, including generating and adapting music. AUD_ui.js also shows the current generated melody (the actual notes). While the system is playing audio, the UI updates to show upcoming notes. For the purposes of this section, screenshots of AUD_ui.js are used to demonstrate the underlying concepts.

The AUD.js UI shows a grid structure (figure 4.2), where vertically different voices are displayed and horizontally different musical time steps (beats)[2] are displayed. AUD.js' output audio has 18 total voices. A musical voice is a single potential source of audio. For instance, a string on a guitar can only produce one note at a time. A guitar has six strings though, so we can say it has six voices. AUD.js groups it's voices into 2 percussive voices representing a snare drum and a high hat, and 4 melodic instruments with 4 voices each. The decision to have 4 melodic instruments and 2 percussive was made because it offers a good balance between the code's performance (having the generation process not take too long) and having enough different instruments to choose from. In other implementations of this algorithm, especially if real-time audio synthesis isn't required, any number of instruments could be used.

For ease of programmatically scheduling when voices should be playing, the four

[2] A beat is a short musical time step; music speed is usually expressed in beats per minute (BPM). AUD.js chooses a random BPM between 90 and 180. The BPM in AUD.js' music varies with the seed, but not with the stress & energy, to allow for smooth adaptation of the music.

	M0 T0	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11
Instr. 1 (Percussion)	C0		C0		C0		C0		C0		C0	
Instr. 2 (Percussion)	C0											
Instr. 3 (Melody 1)	G1								C1			
Instr. 4 (Melody 1)					D2				G2			
Instr. 5 (Melody 1)												
Instr. 6 (Melody 1)												
Instr. 7 (Melody 2)												
Instr. 8 (Melody 2)	C2				D3				C3			
Instr. 9 (Melody 2)	G1		G2		G2		G1		D3		G2	
Instr. 10 (Melody 2)												
Instr. 11 (Melody 3)												
Instr. 12 (Melody 3)												
Instr. 13 (Melody 3)												
Instr. 14 (Melody 3)												
Instr. 15 (Melody 4)												
Instr. 16 (Melody 4)												
Instr. 17 (Melody 4)												
Instr. 18 (Melody 4)												

Figure 4.2: AUD.js instrumentation readout, showing which voices are associated with percussion or a specific timbre.

voices of each melodic instrument are each only capable of one length of note: one produces whole notes (8 beats long), one produces half notes (4 beats long), one produces quarter notes (2 beats long) and the last voice produces eight notes (1 beat long). Higher energy will usually prefer using many short notes, whereas low energy will typically use the longer notes. Not all voices of a melodic instrument will be used at the same time (see section 4.8.4). The decision of having each voice only be capable of one length note was for ease of scheduling when notes could possibly play, thereby simplifying the generation process and the playback. If the implementation had supported tracks that could play any length note, perhaps the number of tracks used per melodic instrument could have been lower.

All the voices of a melodic instrument share the same timbre. Timbre is a sound's texture; it's how humans can distinguish between different instruments or sound sources. At a fixed point in space, sound can be thoughts of as increases and decreases

in air pressure. Timbre is a unique pattern of higher and lower pressure regions associated with an instrument or voice. How fast a sound switches from high to low pressure is called the frequency, or pitch or note. To mimic unique timbres, AUD.js starts with a triangle wave (a simple, linear transition from high to low pressure) and then adds random peaks and valleys of pressure (see figure 4.3). This causes each of AUD.js' melodic instruments sound somewhat unique, while still sounding like chiptunes (classic game music). Depending on stress, these peaks and valleys can be harsher. The higher the stress, the harsher the peaks and valleys will be, and the less smooth or gentle an instrument will sound. Though this randomized approach is not the best, reliable way to get unique sounding instruments, it is computationally fast. A better approach is described in chapter 8 (future work).

Figure 4.3: AUD.js uses randomized waveforms with a smoothness factor tied to the stress level, and randomized ASDR envelopes.

The voices of each melodic instrument also share an envelope. An envelope specifies how loud a note will be after it is turned on until it is told to turn off. Like pressing a key on a piano, the initial hammer strike on the string is loud, and then the note keeps playing while you hold the key down, and fades out quickly once you release it. Specifically, AUD.js uses ASDR (Attack, Decay, Sustain, Release) envelopes. Attack and Decay determine how quickly a note comes on when a note needs to be played (i.e. a key is pressed). Short attack and decay sound very instantaneous, like percussion, whereas long attack times make the note sound gentler. Sustain indicates

how loud the note should be while it needs to stay on (i.e. a key is held). Release specifies how long it should take for the note to fade out (i.e. when the key is let go). The attack, decay and release time are randomly chosen between 3% and 30% of the time of one beat. The max attack volume is always 100%, but the sustain volume is randomly chosen between 30% and 60%.

Percussion instruments on the other hand are represented by noise waves (random audio samples that sound like static) with instantaneous attacks, no sustain and release between 50% and 90% of a beat for a snare and 150% and 200% of a beat for a high hat.

4.3 Generation Process Control Flow

This section documents the process of generating a full piece. A number of data structures are created at the beginning and passed along to the melody- and percussion-generating methods so those methods have an understanding of the general progression of the piece (for example, when which instruments should be playing). After that, musical data for the entire piece is generated on a voice-by-voice basis. In more detail:

1. The process begins by creating a new random number generator (RNG), which starts at the given seed. AUD.js uses a custom RNG, since JavaScript's default cannot be seeded.
2. The process then decides three key factors:
   (a) The base note: this is the lowest note that can ever be played. It is a randomly chosen number between 12 and 24. This number equates to a key number on a piano keyboard, starting from the lowest section of the keyboard. The physical range is 51.9 Hz to 103.8 Hz. These numbers were chosen by trial and error until the generated pieces sounded like they encapsulated a decent note range.
   (b) When and where to forward patterns: this causes certain melodies to be repeated, by indicating when a generated pattern should be moved to a

not-yet-generated part of the track. This is explained in more detail in section 4.8.2.

   (c) What kinds of transitions to use: between unique patterns, there will be an 8 beat transition period to signal to the listener that there will be a melodic change. These transitions are randomly chosen to be either high or low energy (so there will be more or less notes than usual during a transition). This is explained in more detail in section 4.8.1.

3. The next step is to pick when to use which instruments. At no point in time should too many or too few instruments and notes be playing. This is described in more detail in section 4.8.4. At this stage, the envelopes and waveforms for the instruments are decided based on the stress and energy.

4. Next, a method called *populate_perc* is called twice. It populates a voice with percussion data - it is called once for the snare-like instrument and once for the high-hat-like instrument. The method has different probabilities to use depending on which voice is being generated. This is described in more detail in section 4.4.

5. Next, the probabilistic data structures are created based on the stress and energy. The data structures are described in section 4.5, and their creation based on the input is described in section 4.7.

6. Finally, the *populate_melody* method is called on each voice of the melodic instruments. This is explained in more detail in section 4.5. This method is given all the data structures: the probabilistic ones for note selection and the information regarding pattern forwarding and transitions. The method is also given a base note: the first melodic instrument gets the original base note, the second gets a base note that is 7 half-steps higher, the third that is 12 keys higher than the original and the fourth gets a base note that is 19 keys higher. Each melodic instrument may generate notes from the range of their base note plus 19 keys (an octave and a half). These numbers were chosen through trial and error, until the instruments all contributed sufficiently to giving the system a good expressive range. Seeing as how each melodic instrument has a personalized base note, each instrument captures a slightly different range, but there

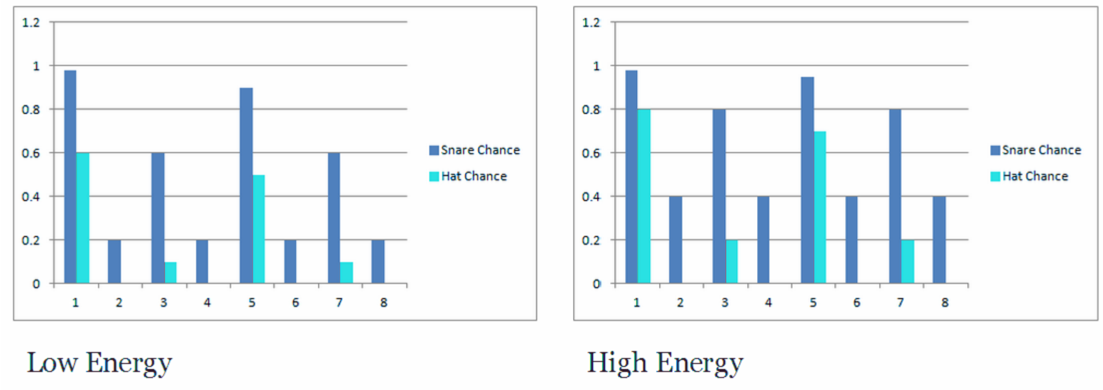

Figure 4.4: The probabilities of snare hits and high hat hits occurring, as a function of the beat number in a measure.

is some overlap. Once each voice has been generated, the generation process is complete.

4.4 Percussion

To determine when a percussive hit (snare or high hat) should occur, a random number is generated. The chance of a percussive hit occurring depends on which beat of the measure is currently being checked. A measure is a grouping of beats, and is usually considered one rhythmic unit, where the notes of first beat are the loudest. Other beat numbers in the measure can have varying levels of loudness, depending on how many beats there are in the measure. The way AUD.js represents this is by weighting the chance of a percussive hit occurring higher on the first beat (98% for snare), and lowest on the even numbered beats (between 20% and 40% depending on energy for snare). AUD.js uses 8-beat measures. See figure 4.4 for more detail. These percentages were acquired by trial-and-error testing of the software during development, reflecting the authors' taste and preferences. During development, if there was too much percussion, or if the track was overpopulated (e.g. at low energy it was playing a snare on every beat), the numbers would be tuned down by hand and vice-versa. This was a subjective process based on some familiarity with classic video game music.

4.5 Note Selection

To determine if a note should be played (or to insert a rest), the same process as percussion generation is performed. The probabilities were also determined by trial and error: if there were too few notes for high energy, the numbers would get tuned up by hand, and vice-versa.

The note selection itself, however, was not done by trial and error. Two probabilistic data structures are used to pick notes when needed: prevalence arrays and elimination tables. The numbers used were obtained from classic game music, as described in section 4.5.

4.5.1 Prevalence Arrays

A prevalence array is a list with 12 entries: one for each named note (figure 4.5). Each entry is a probability of that note being chosen. Before a note is probabilistically selected, any note with a low percentage change is completely eliminated. This is to prevent any note that is too "bad" from being selected, even when the chance a "bad" note is selected is very low. If a "bad" note were to be chosen, especially for long notes, it could throw off a listener and make the piece sound dissonant when that was not the intention. The cutoff percentage for "bad" notes depends on how long the note played will be. Whole notes must be at least 10% prevalent, half notes 5%, quarter notes 2% and eighth notes 1%.

4.5.2 Elimination Tables

When a note needs to be chosen, before the prevalence array should be used, all potential notes need to be checked to see if they would sound decent given the notes that had already been chosen, and would be playing at this point in time. To do so, an elimination table is referenced (figure 4.6). This is a 12 by 12 table, comparing each named note against each other named note. A note may always be played with itself. For all other notes, there's a percentage cutoff which varies between 2% and 5% based on stress.

After eliminating all notes that would not sound good with the already-chosen notes, the prevalence array is referenced as described above.

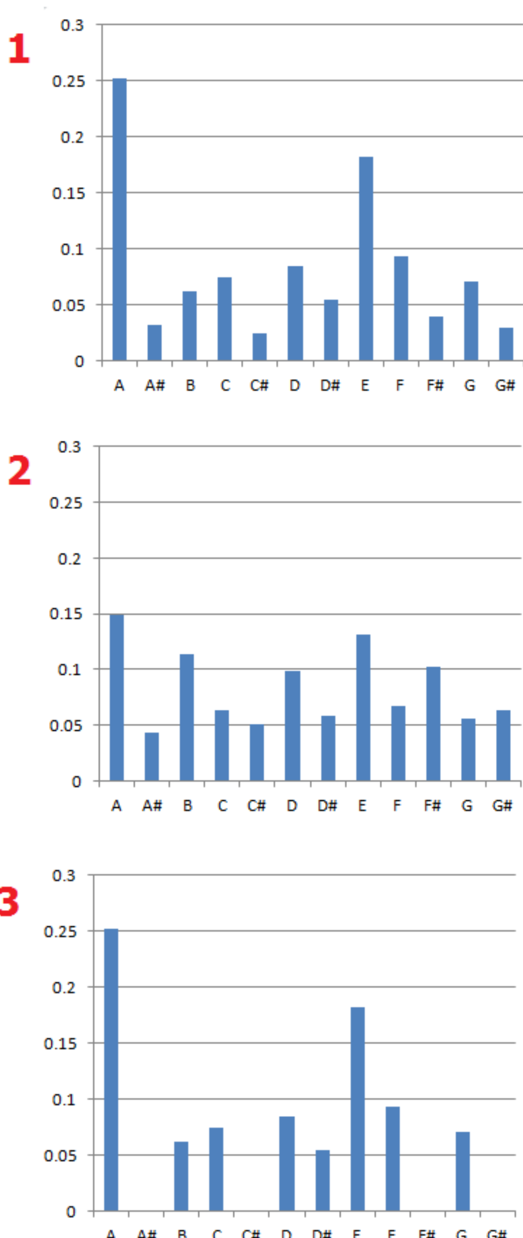

Figure 4.5: Prevalence Arrays. 1: A prevalence array for low stress. 2: A prevalence array for high stress. 3: A prevalence array for low stress with notes lower than a 5% cutoff removed.

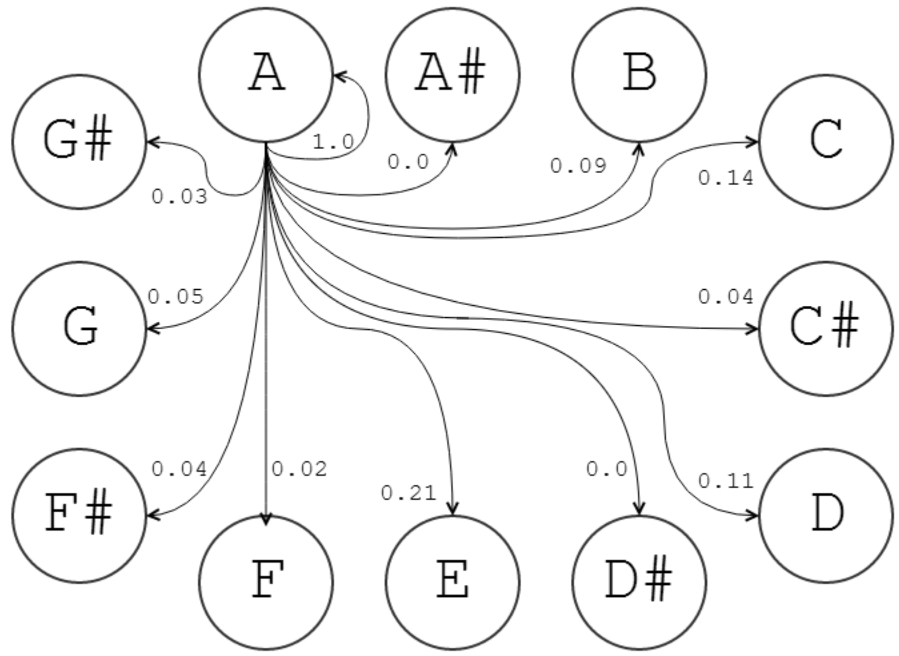

Figure 4.6: One of the 12 entries in an elimination table, showing the likelihood of an 'A' note being played with any other named note.

4.6 Data Acquisition

The prevalence arrays and elimination tables were extracted by parsing classic video game soundtracks. MIDI files of the tracks were programmatically inspected to see how frequently each uniquely named note was played and for how long, as well as how frequently every possible pair of notes occurred together. For a full list of pieces used, see Appendix A.

A total of 80 pieces were rated by first-year game developers (potential end-users of the system) as having either low, moderate or high stress and low, moderate or high energy on a scale from 1 to 5. The average stress is 2.9 / 5 and the average energy is 3.3 / 5, meaning high energy is slightly overrepresented (see figure 4.7). Each piece was rated by at least 30 people, and a total of 93 people participated. These studies were conducted by the authors on December 3, 2013 and February 18, 2014 during Cal Poly's Intro to Game Design and Intro to Interactive Entertainment, respectively.

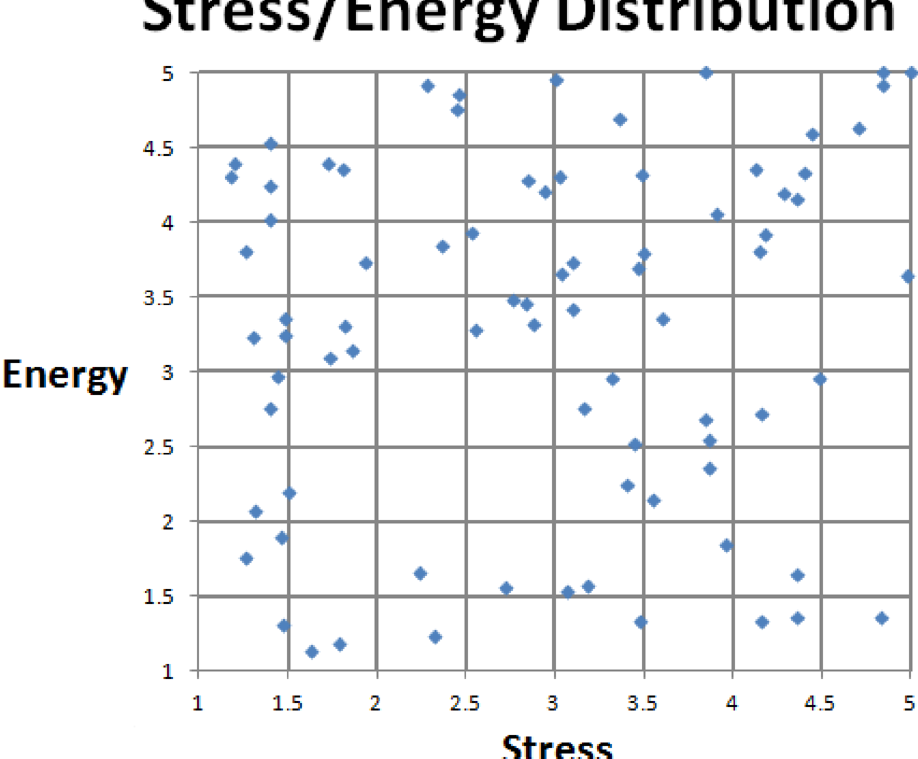

Figure 4.7: Rated stress/energy values of classic video game music.

4.7 Generation & Adaptation

When the generation process is started, a custom prevalence array and elimination table are created based on the stress and energy. This is done by averaging the nearest 4 elimination tables and prevalence arrays based on how close they are to the desired stress and energy. The cutoffs (e.g. for percussive hits) are also calculated at this stage.

When adapting, AUD.js completely regenerates the entire piece. A new prevalence array, elimination table and cutoffs are calculated from the new stress and energy. The random number seed gets reset at the beginning of the process to ensure the newly generated piece sounds similar to the old piece. The AUD_interface (audio synthesizer) then switches between the old and new piece by linear interpolation: gradually changing which one is played by fading one out and fading the other in over the course of approximately one second.

4.8 Heuristics

A number of augmentations have been applied to the music to make generated pieces sound more coherent, less repetitive and more unique. For a full list of the heuristics used and their numeric values in the code, please refer to appendix B.

4.8.1 Transitions

Between unique melodies (patterns), a transition is introduced to signal to the listener that a change in melody is coming. This helps emphasize the uniqueness of different patterns by separating them with a 1-measure-long (8 beats) transition. Transitions can be either high or low energy, and the transition being high or low energy is chosen before any audio is generated. The percussion and note selection process occurs just as described earlier, but uses either a higher or lower energy value when generating. For high energy transitions, the energy value is raised by 0.5 (energy is on a 0 to 1 scale) for the 8 beats of the transition. Similarly for low energy transitions, the energy value is reduced by 0.5 temporarily.

4.8.2 Pattern forwarding

In order to make a generated piece sound more coherent, the system will occasionally copy some music to a future, not-yet-generated pattern. An example of this type of phenomenon in composed music is a chorus, where a specific piece of the music is repeated several times. In AUD.js, the choice of where to move which patterns is made before any audio has been generated, since the generation process goes voice-by-voice, rather than pattern-by-pattern. As such, any decision about patterns, pattern progression or transitions needs to be made ahead of time.

4.8.3 Variation in repeats

AUD.js allows for patterns to repeat (play a pattern multiple times in a row). As with pattern forwarding, this may help increase coherence of a piece. However, in order to reduce potential repetitiveness of the generated music, when a piece is copied

as a repeat, slight variation may be introduced. Each note of the copied pattern has a 15% chance to be changed. If it does get changed, the note selection process is performed. It is possible for the old note to get selected as the new note.

4.8.4 Voice/Instrument limits

Good music is rarely completely silent or overwhelming. As such, AUD.js prevents too few or too many instruments from being active at the same time. 1 to 3 instruments will always be playing, depending on energy level. Similarly, an instrument should not be producing too little or too much audio. An instrument can only use 2 or 3 of it's voices at a time, also depending on energy. As a result, in the entire system, not including percussion, there will always be 2 to 9 voices active.

4.8.5 Hard cutoffs

The magic numbers used to eliminate improbable notes and to determine when percussion or notes should be played are also heuristics. The numbers chosen for those probabilities were determined to procude sufficient quality music by trial-and-error testing of AUD.js during development.

Chapter 5

Experiments & Results

AUD.js was developed to programatically provide music of sufficient quality to video games. Since AUD.js is a software framework for music generation, it should also be a way for games to get audio without the developers having to seek out a composer. Since the effort of composition is removed, AUD.js would be able to reduce the pressure associated with music integration into a game when developing on a tight schedule.

As such, the hypothesis we tested were as follows:

- Does AUD.js provide acceptable music for a game?
- Does AUD.js help reduce pressure in a game jam setting?

We ran a number of experiments to assess these hypotheses: two experiments at Cal Poly's Global Game Jam 2014 site (one targeted at developers and one targeted at players), one experiment to assess how adaptive AUD.js music enhances or detracts from gameplay experience, and one experiment to verify that AUD.js' output music mood is audibly equivalent to the desired mood.

5.1 Global Game Jam Developer Survey

During the GGJ, participants had the option to use the AUD.js system to provide them with music. Once the jam was over, the developers that used the system were given a survey asking them to rate different aspects of their experience with AUD.js, such as how easy it was to use and how good the quality of the generated music was (see table 5.1). If participants building a game in JavaScript chose not to use AUD.js, they were instead given a survey to assess why they chose not to (see table 5.2).

Rate how easy you felt the AUD.js API was to use.	4.41 / 5
Rate how easy the music mood model (stress, energy) was to understand.	4.5 / 5
Rate the quality of the generated music.	3.58 / 5
Rate how well the music responded to the input mood you gave.	3.91 / 5
Would you use AUD.js again in the future?	91% yes

Table 5.1: Developer survey results: Global Game Jam participants that used AUD.js

Out of the four teams at Cal Poly's Global Game Jam eligible to use AUD.js, one ultimately decided against keeping it in the final project.

5.1.1 Results

Three teams used AUD.js: Bullet'space (2 person team), Anellu Moore (6 person team) and Phancy Adventures of Charles the Cat (6 person team). All games can be found at *http://globalgamejam.org/2014/jam-sites/cal-poly/games*.

Out of these teams, 12 people filled out the survey For detailed results, see table 5.1. In general, developers found AUD.js easy to use and understand. 91% of participants would use it again, despite the average rated quality of the music being at 3.58 out of 5.

One team that created a game with JavaScript decided against using AUD.js: 5Alive (5 person team). This game can be found at *http://globalgamejam.org/2014/jam-sites/cal-poly/games*. Four members of this team filled out the survey. For detailed results, see table 5.2 (duplicate/highly similar feedback is omitted). The team had a composer available, who made a custom track for their game. The team mentioned on the surveys that AUD.js' quality was not nearly as good the composed track, and that the game did not necessarily require adaptive music.

5.1.2 Analysis

This survey adequately assesses the second hypothesis. Participants responded that it was very easy to use and that they would use it again under similar settings. If the

Why did you elect not to use AUD.js?	"A day into the jam a musician volunteered to do our music." "The song we had fit better, and the game's feel did not depend on the music adapting." "Was cool, but didn't fit the tone of the game well."
How did you get your music for your game?	A day into the jam a musician volunteered to do our music.
Would you consider using a system like AUD.js for a different project?	100% yes
What would need to be improved in AUD.js before you would consider using it in a game jam?	"Not having a musician, as a musician will be able to craft a more coherent melody to match the tone of the game." "We'd need a game that depended more on it and it would be if the audio quality was higher." "Possibility for different styles. No console logs." *(A developer build of AUD.js had been distributed during the GGJ; debug logs had been left in.)* "As hard as it is, the quality of the songs. For people who dont realize the music is procedural, it can be off-putting."

Table 5.2: Developer survey results: Global Game Jam participants that used JavaScript, but chose not to use AUD.js.

team has a composer though, the team has access to higher quality audio. Version 5 of AUD.js was used during the GGJ, whereas version 6 was used in later experiments. The feedback from Global Game Jam helped shape the improvements for version 6, since the developers mentioned the audio quality was not as good as it should be to be used in games.

5.2 Global Game Jam Player Survey

People interested in games developed at Cal Poly's Global Game Jam site played the games that had the AUD.js system integrated, and filled out a survey assessing their opinion of the game and its music. For comparison, survey results were also collected on games that didn't use the AUD.js system.

23 people rated games developed at game jams at Cal Poly. These people were randomly assigned to one of four games, two of which used the AUD.js framework. These people were not told they would be reviewing the music and weren't told that the music may be procedurally generated and adapting. Only the most representative responses of the written responses are included.

5.2.1 Results

15 people rated either Bullet'space [1] or Anellu Moore [2], which both had AUD.js integrated. For detailed results, see table 5.3. Players didn't enjoy the chiptune style music AUD.js generated. AUD.js version 5 was used during GGJ. The feedback was used to create version 6, which was used in the last two experiments (discussed in sections 5.3 and 5.4). Players did notice the music adapting with the game, although not an ideal number (40% of players).

8 people rated either Tricollide [3] or heart4u [4] (games from previous Cal Poly game jams, with comparable gameplay quality as the AUD.js-integrated games [5]). For de-

[1] Bullet'Space was created by Tim Adam and Simon Vurens during GGJ 2014 at Cal Poly

[2] Anellu Moore was created by Daniel Johnson, Daniel Griffith, Isaac James, Melissa James, Robert Del Papa and Sean Slater during GGJ 2014 at Cal Poly

[3] Tricollide was created by Marty Loewenthal, Chris Wallis and Mike Wang during GGJ 2014 a Cal Poly Game Development Club game jam event in Spring 2013

[4] heart4u was created by Jorge Medoza and Patrick Casao during GGJ 2013 at Cal Poly

[5] Other games from Cal Poly's 2014 Global Game Jam either had additional hardware require-

Rate how immersed you felt in the game.	1.93 / 5
Rate how effective you found the music to be.	1.64 / 5
Rate how much you enjoyed the music.	2.21 / 5
Would you listen to the music if it wasn't in a game?	13% yes
Did you feel the music was responsive to game events and user controls?	33% yes
Was there an event in the game, or a specific control that triggered a music change that you noticed?	40% yes
What could be improved about the music?	"Music is too loud/screachy (too high pitched)" "Chiptune isn't my thing" "[The music] didn't sound like it belonged in the game"

Table 5.3: GGJ player survey results: games with AUD.js.

Rate how immersed you felt in the game.	2.25 / 5
Rate how effective you found the music to be.	3.87 / 5
Rate how much you enjoyed the music.	3.75 / 5
Would you listen to the music if it wasn't in a game?	12.5% yes
Did you feel the music was responsive to game events and user controls?	37.5% yes
Was there an event in the game, or a specific control that triggered a music change that you noticed?	12.5% yes
What could be improved about the music?	"More variation" "A bit of reaction to what I was doing"

Table 5.4: GGJ player survey results: games without AUD.js.

tailed results, see table 5.4. As with the AUD.js version, players didn't actually enjoy the music very much. Though the music was composed rather than procedurally generated, it only scored a 3.75 out of 5, whereas AUD.js received a 2.21 out of 5. Though there is a considerable gap, both values are lower than anticipated. Players typically didn't notice changes in the music, which makes sense considering the composed music didn't adapt to game events. Players noted that they would have

ments, were platform-specific or were largely unfinished.

liked to see the music adapting to game events, but that might be because survey participant had been primed to think about game music adaptation when answering the previous questions.

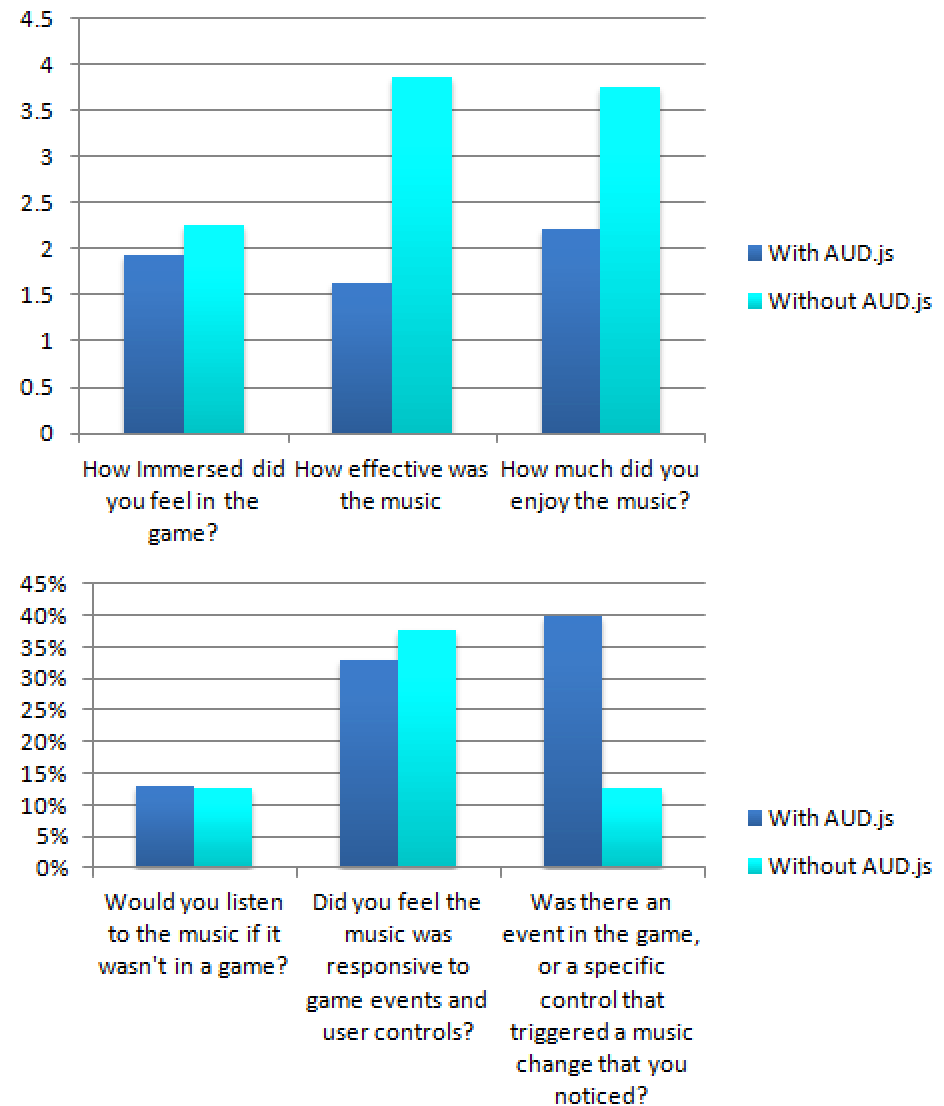

Figure 5.1: A comparison of the results of the GGJ Player Survey.

For a visual comparison of the numeric results, see figure 5.1. In a number of the categories (immersion, responsiveness, listening to the music outside of a game), the results for AUD.js and composed music are almost the same. Considering the difference in effort (writing a few lines of code for AUD.js versus spending hours composing a piece of music) this is good news for AUD.js. As expected, people

enjoyed the composed music more, and found it to be more effective for the game. More people noticed music adapting with AUD.js, which is also to be expected, although not quite as many people noticed it as would be ideal.

5.2.2 Analysis

The first hypothesis set forth in the previous chapter is this: Does AUD.js provide acceptable music for a game? The Global Game Jam Player survey was the first attempt at evaluating AUD.js in this regard. Though the experiment did show that the adaptiveness was perceivable, it also showed that AUD.js music quality was far lower than composed music.

This survey did have experimental issues though. The survey did not control for appropriateness of the music for the game and did not control for the game's quality. As such, a follow-up experiment was conducted (see next section).

5.3 A/B Test

The experiment run at Global Game Jam did not control for game quality or music appropriateness. As such, we set up a second experiment, which took two classic-style games which would work well with AUD.js chiptune. We created three versions of each game: one version with composed music (chosen by the original developers), one version with non-changing AUD.js music, and one version with the actual AUD.js system integrated.

The first game, The Brothers Chronosoff (figure 5.2) [6], was a 2D platformer where the player controls one of the two characters at a time. Each character leaves a trail of afterimages, which the other character can stand on. In the AUD.js version, the music adapts when switching between characters. The blue brother has low stress and the red brother has high stress music. The energy of the piece also increases whenever the player beats a level (there are 3 levels total).

The second game, Bullet'space (figure 5.3) [7], was a scrolling space shooter. The

[6] The Brothers Chronosoff was created by Nick Alereza and Chad Brantley during Dr. Foaad Khosmood's Spring 2012 CSC 478 (Interactive Entertainment) class at Cal Poly.

[7] Bullet'Space was created by Tim Adam and Simon Vurens during GGJ 2014 at Cal Poly

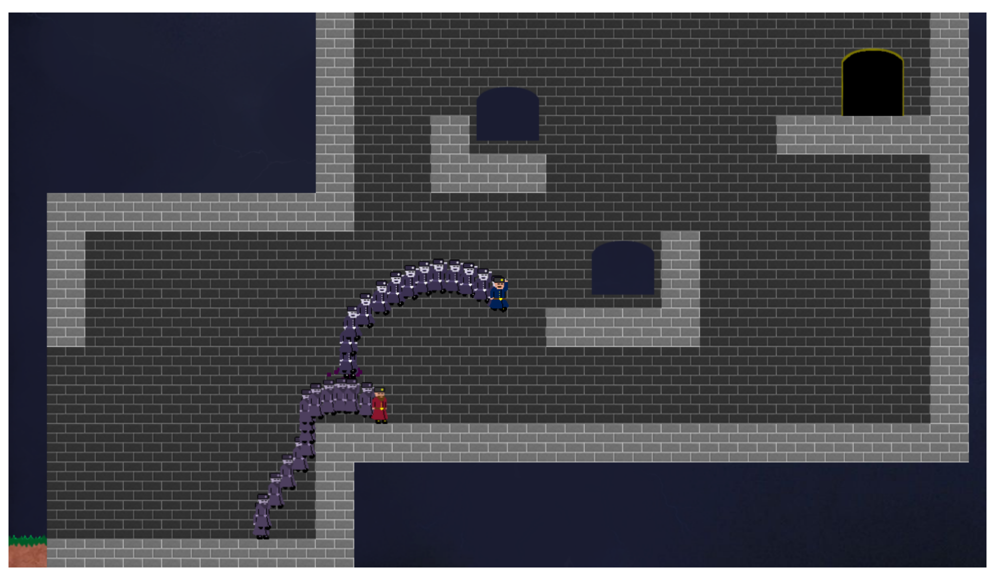

Figure 5.2: The Brothers Chronosoff.

player controls a ship that has three unique configurations: shield, guns and speed. Enemy waves spawn on a timer, but the type of enemy is dependent on the current configuration of the player's ship. In the AUD.js version, the music adapts when the player switches ship configurations. The shield configuration has low stress and energy, the speed configuration has low stress and high energy and the gun configuration has high stress and energy. In all versions of the game, the track changes when the boss spawns; this is not considered an adaptation.

Participants in the study were randomly assigned to one of the three versions of one of the games, and asked to fill out a survey. They were not told about AUD.js or procedural music generation until after they had completed the survey.

5.3.1 Results

30 people total participated, resulting in each version of either game being played exactly 5 times.

For both games on average, the immersiveness of the game, the effectiveness of the music and the enjoyment of the music is roughly equal. Composed music is unsurprisingly enjoyed slightly more and is considered slightly more effective.

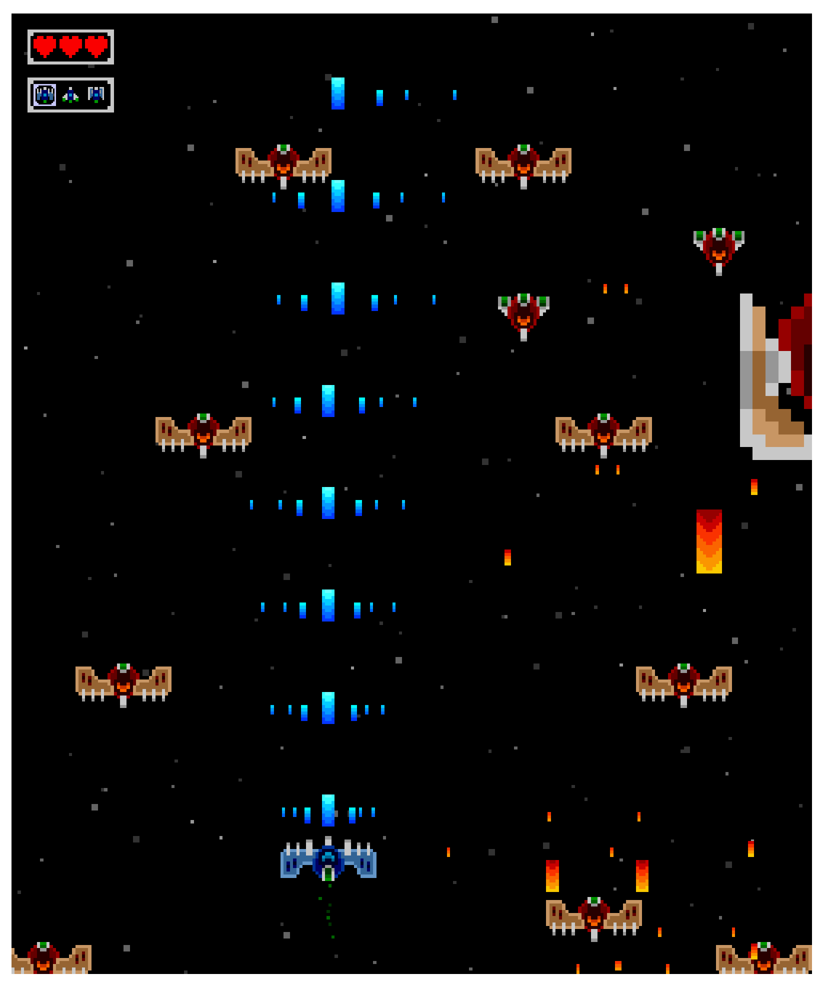

Figure 5.3: Bullet'space.

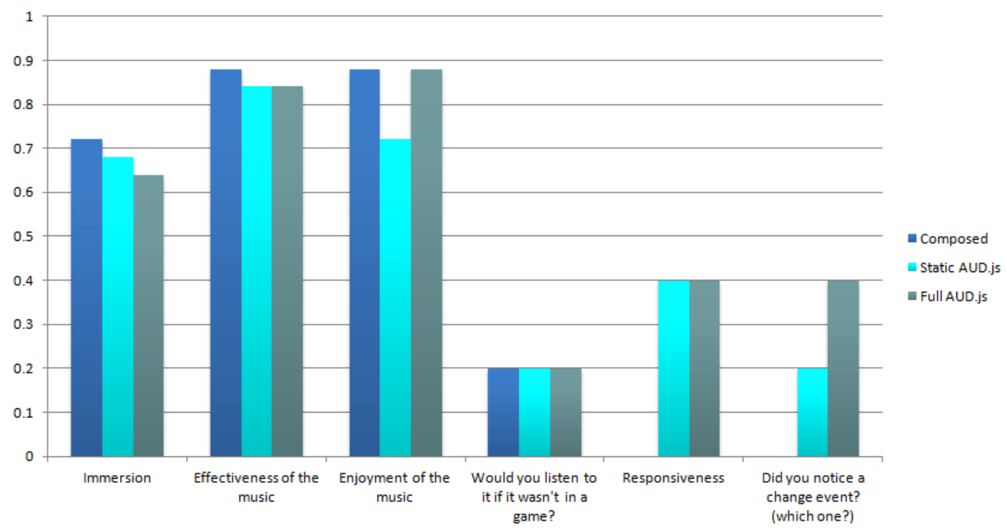

Figure 5.4: The results of all three versions of Brothers Chronosoff.

For Brothers Chronosoff (see figure 5.4), all version of the music would likely not be listened to outside of the game. The composed music is considered less responsive, and more people notice the adaptation of the music with the dynamic AUD.js. One person claimed to have heard adaptation with the static AUD.js, which is likely due to fortuitous timing of the soundtrack with game events. For detailed results, see figure 5.4.

For Bullet'space (see figure 5.5), many people would listen to the composed music outside of the game, which is understandable. Dynamic AUD.js and composed music are considered highly responsive, but most importantly every participant noticed the adaptiveness of the dynamic AUD.js music. No one noticed any adaptation with the other versions of the game. For detailed results, see figure 5.5.

5.3.2 Analysis

This experiment addresses the issues with the Global Game Jam survey, while still addresses the first hypothesis (assessing whether AUD.js provides adequate quality music). Since the same two games were used in different versions, and the games were explicitly chosen for their retro feel, the two issues from the first survey are addressed: the music fits the game (chiptune for retro games) and differences in game quality (by using the same game with different music). In this survey we see that AUD.js'

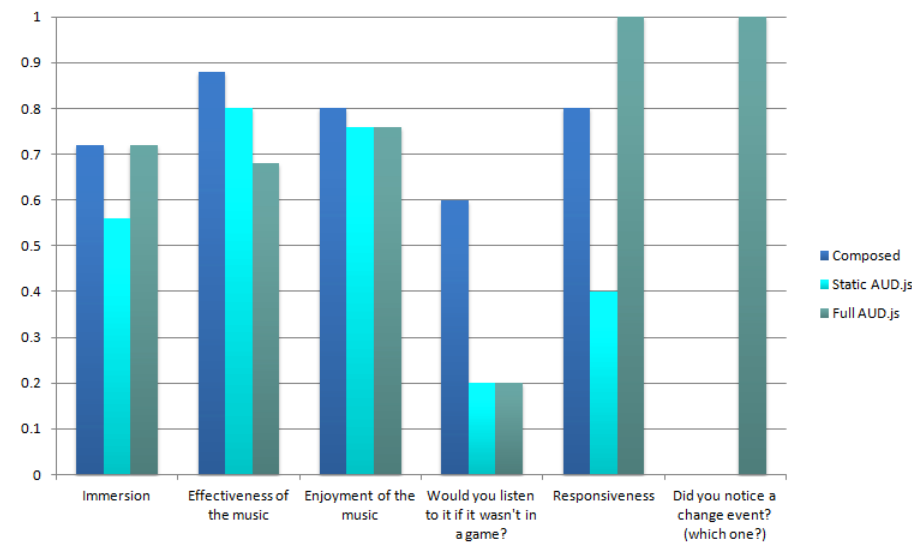

Figure 5.5: The results of all three versions of Bullet'space.

music quality is nearly on par with composed music. AUD.js does not detract or add to game experience though. It would have been neat to see game immersion improve with adaptive music, but unfortunately AUD.js doesn't cut it on that field. Participants in that experiment noticed the adaptations and found AUD.js to be a lot more responsive to the game events. So overall, with this survey we can say that AUD.js does provide sufficient quality audio to a game.

5.4 Mood Identification

As mentioned in section 4.5, AUD.js has been trained with classic video game music that got rated (stress & energy) by game development students. In order to assess whether the perceived mood of AUD.js' output music matches the input stress and energy values, we ran the same experiment, except using AUD.js as the music to be rated. The goal here is not necessarily to address one of the key hypotheses, but rather to test if AUD.js lives up to its main implementation goal: generating and adapting music based on mood.

We created 5 categories of stress and energy: 0%, 25%, 50%, 75% and 100%. We tested one sample output from each of the 25 possible combinations of the stress/energy categories. The participants were informed about AUD.js, but were

not told prior to the survey that there was exactly 1 piece from each combination of categories. The participants would hear a piece, and circle which stress and energy category they thought they heard, both ranging from 1 to 5.

5.4.1 Results

The experiment was conducted on April 18, 2014 in Dr. Zoë Wood's CPE 476 (Real-time 3D Graphics). A total of 31 third and fourth year game development students participated in this study. The results are shown in figure 5.6.

Across all pieces, the average perceived stress was 2.57 and the average perceived energy was 2.46. Since there was an even distribution of the pieces across the categories, the averages should have been 3 each. The standard deviation, however, was 0.76 for perceived stress and 0.48 for perceived energy.

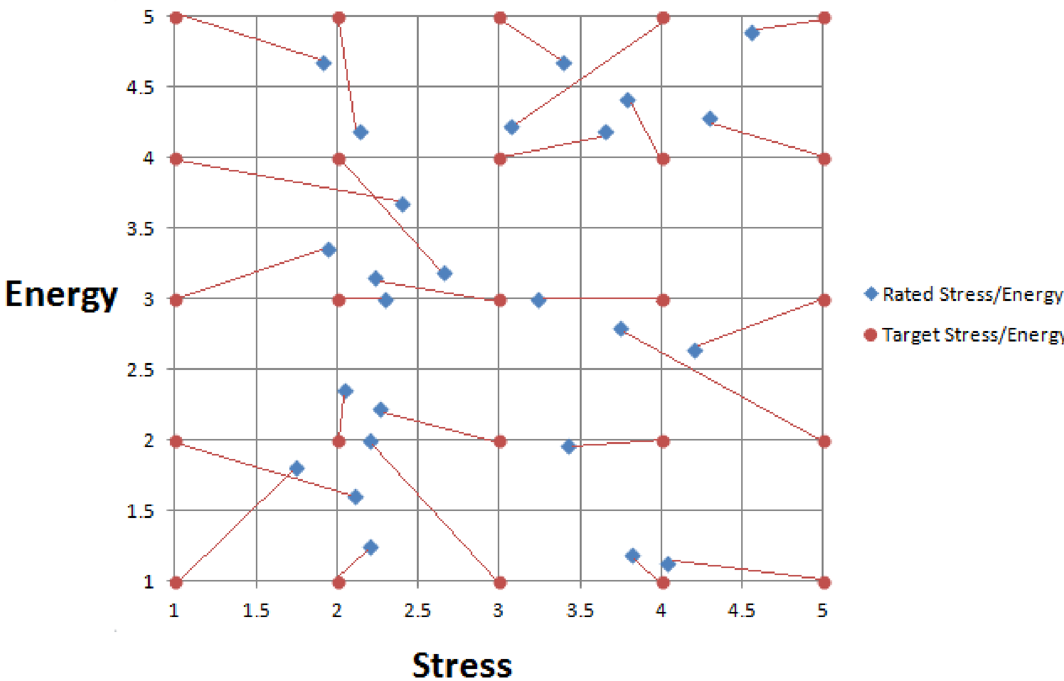

Figure 5.6: AUD.js mood estimation survey results, showing the input categories to AUD.js in red and the rated stress and energy in blue.

Inspecting the individual categories, a trend emerges. Both for high stress and

energy, people tend to underestimate. Both for low stress and energy, people tend to overestimate.

- Category 1 (low) stress pieces on average got estimated at 2.01, more than an entire category higher.
- Category 3 (medium) stress pieces on average got estimated at 2.74, just slightly lower.
- Category 5 (high) stress pieces on average got estimated at 4.16, almost an entire category lower.
- Category 1 (low) energy pieces on average got estimated at 1.47, about half a category higher.
- Category 3 (medium) energy pieces on average got estimated at 3.03, nearly exactly correct.
- Category 5 (high) energy pieces on average got estimated at 4.53, about a half a category lower.

5.4.2 Analysis

This experiment was run to verify that AUD.js performed it's core task adequately: generation and adaptation based on mood. Since the rated stress and energy values trend towards medium stress and energy (i.e. high stress/energy is underestimated, low stress/energy is overestimated) we can deduce that AUD.js could benefit from additional expressiveness to emphasize extreme values of stress and energy. The average stress and energy ratings were close to average overall stress and energy though, from which we can deduce that the system is equally equipped to represent higher and lower values of stress and energy. From the ratings we can also deduce that a change in stress and energy is clearly perceivable, although not as effective as we would hope. From Livingstone & Brown, additional tools for expressiveness that could be added to AUD.js include more closely controlling the tempo, the mode, and instrument timbre and loudness [8].

Chapter 6

Related Work

Other game development studios have created their own frameworks to provide dynamically adaptive music in the past. This chapter covers the most notable examples and how AUD.js differs from those frameworks.

6.1 LucasArts' iMUSE

In 1991, LucarArts' Michael Land and Peter McConnell filed a patent for a dynamic composition system, dubbed iMUSE (interactive music streaming engine). The system composed soundtracks by drawing from a bank of music performance data and instruments and implemented a decision tree structure which allows for real-time adaptations. The patent specifically covers the use of compositional databases, the use of conditional messages and the use of decision trees with real-time input to create real-time adaptive soundtracks [9].

This patent is somewhat old, though iMUSE had been used on several games, including The Secret of Monkey Island. The main difference between iMUSE and AUD.js is that AUD.js does not require any performance data database or instrumentation settings. AUD.js is pre-trained and has its own instrumentation. Though it is considerably less configurable, AUD.js' smaller scope makes it quicker to learn and quicker to use in game.

6.2 Mass Effect 2's Adaptive Music

As mentioned in section 2, Mass Effect 2 used a number of techniques to create dynamic music. These including filtering (amplifying or reducing specific parts of the music), layering (having multiple pieces of the music that can play together to create

more complex tracks) and sequencing short music clips [14].

Again, the main difference between AUD.js and this system is that AUD.js requires no composed music at all. The simplicity of AUD.js interface makes it so that persons with no musical knowledge can still use it. This can save time, but the trade-off naturally is that the music does not sound as though it was composed (and definitely not by a triple-A game development studio).

Chapter 7

Conclusion

In time-critical game development situations like game jams, a programmatic method of including music in a game can be a significant relief. The survey conducted with developers at Cal Poly's Global Game Jam shows that there is a market for easy-to-use programmatic music composition frameworks. The key here is that the API should be easy to include in games, and that the quality of the generated music should be up to video game standards.

In terms of music quality, AUD.js does not perform as well as composed music, but the A/B tests show that AUD.js comes close by providing adequate quality music. The adaptive nature of AUD.js, though noticeable, did not seem to have a strong effect on game experience. Anecdotally, composed adaptive music in other games does contribute to immersion and reinforces the intended mood of the game. The simple rhythmic and melodic adaptation AUD.js performs is likely not sufficient to immerse players further. Most other approaches to adaptive music use composed music as a base, which AUD.js avoids in favor of higher usability: developers don't need a background in music theory and don't need to spend time searching for composed music. That trade-off in quality makes AUD.js uniquely suited to time-critical game development, but more work needs to be done before AUD.js could feasibly be including in full games.

In the future, with augmentation of AUD.js' music quality, it could be a valuable tool for games produced on a small budget. Many independent developers either can't afford a composer or can't find one. Since AUD.js is open-source and free to use, it could be the perfect solution: it would provide games with free music and would require very little of the developers' time. With improvements to the adaptive component of the music, games would get music that can adjust to any scenario in game. Adaptive music can be effective if done right, which to date has only been

done by highly skilled composers and developers. If AUD.js can get to that level in terms of quality, usability and adaptiveness, many games could have free, high-quality immersive soundtracks.

Chapter 8

Future Work

A number of items would be interesting to explore in future versions of AUD.js.

A large limitation of AUD.js in its current state is that it depends on WebkitAudio. This framework is not available on all browsers, so as such AUD.js cannot be run on all browsers. Browser-neutral implementations, as well as non-browser versions of the algorithm, are an important next step.

Currently, only note prevalence data is extracted from the training set. AUD.js uses magic numbers for rhythm. An interesting approach for future versions of the algorithm would be to extract rhythm data in some form from the training set, and use the gathered data in some fashion in AUD.js.

AUD.js currently produces chiptune, which doesn't work for all types of games or applications. Customizable instrumentation would greatly help alleviate this issue. However, in doing so, the algorithm would need to know about the chosen instruments and their usage (for example, so it doesn't use a bass to generate high-pitched melodies). The API for instrumentation should also not be too cumbersome, as the rest of the API has been specifically created to be quick to learn and use and still provides a good range of functionality. Finally, the instrumentation should be quick to load. Loading samples can take quite while on a web browser, which is why a non-sample based approach was chosen for AUD.js. Since AUD.js is meant to be used in games, it should minimally impact their loading times, so as not to irritate players.

AUD.js' method of creating various timbres is very rudimentary. AUD.js' instruments sound similar due to the nature of the waveforms used (mostly triangle waves with random peaks and valleys). A better way of creating diverse, interesting instruments would be to choose specific overtones we'd like to be present in an instrument's

timbre. Then perform an inverse Fourier transform on that frequency spectrum to obtain a waveform to plug into AUD.js' synthesizer. So rather than randomizing the waveform and hoping the resulting overtones sound good, we would be picking (perhaps randomly or probabilistically) which overtones we want, then creating the waveform.

Finally, in part to address the issue of AUD.js' music not fitting every game, customizable training sets could greatly benefit AUD.js. The current MIDI-parser that creates the elimination tables and prevalence arrays is a separate program. If the parser were to be integrated into AUD.js, as with the instrumentation, it would need to be quick and have a simple API. However, once added to AUD.js, it should open the possibility for customized styles of music. Paired with customizable instrumentation, it would provide the option of customized genres as AUD.js' output music.

Bibliography

[1] Challenge: Create game music for game jam team 2. https://www.audiodraft.com/contests/190-Challenge-Create-game-music-for-Game-Jam-Team-2, 2013.

[2] C. Bell. Algorithmic music composition using dynamic markov chains and genetic algorithms. volume 27, pages 99–107, 2011.

[3] K. Collins. An introduction to procedural music in video games. *Contemporary Music Review*, 28(1):5–15, 2009.

[4] D. Cope. *Virtual music: computer synthesis of musical style*. MIT press, 2004.

[5] A. Farnell. An introduction to procedural audio and its application in computer games. In *Audio Mostly Conference*, pages 1–31, 2007.

[6] A. Ghassaei. What is midi, 2013.

[7] H. Jarvelainen. Algorithmic music composition. *Tik-111.080 Seminar on content creation*, 2000.

[8] S. Livingstone and A. Brown. Dynamic response: real-time adaptation for music emotion. In *Proceedings of the second Australasian conference on Interactive entertainment*, pages 105–111, 2005.

[9] P. N. M. Michael Z. Land. Method and apparatus for dynamically composing music and sound effects using a computer entertainment system, 05 1994.

[10] R. Ramirez and A. Hazan. A learning scheme for generating expressive music performances of jazz standards. In *International Joint Conference On Artificial Intelligence*, volume 19, page 1628, 2005.

[11] C. Rogers. Web audio api, 2013.

[12] J. Russell. A circumplex model of affect. *Journal of personality and social psychology*, 39(6):1161, 1980.

[13] R. E. Thayer. *The biopsychology of mood and arousal*. Oxford University Press, 1989.

[14] Wall Of Sound Inc. Mass effect 2 interactive score demo. Accessed: 2014-04-26.

[15] Wikipedia.org. Markov chain. Accessed: 2014-05-31.

[16] I. Xenakis. Formalized music. *Bloomington: Indiana University Press*, 1971.

Appendix A

Classic Game Music used as Training Set

Each piece was reviewed by approximately 30 people. A total of 93 game design students participated in this study.

Franchise	Name	Stress	Energy
CastleVania (Konami)	Bloody Tears	3.48	4.32
	Ending	1.47	1.31
	Heart of Fire	3.5	3.8
	Nocturne of Shadow	3.18	1.57
	Out of Time	3.02	4.31
	Poison Mind	3.9	4.06
	Silence of Daylight	3.46	3.7
	Stalker	3.03	3.66
	Vampire Killer	3.1	3.73
	Walking on the Edge	4.97	3.65
	Wicked Child	4.28	4.2
Donkey Kong (Nintendo)	Candy's Theme	1.46	1.9
	Forest Frenzy	3.32	2.96
	Fungi Forest	1.26	1.76
	Gangplank Galleon	1.2	4.4
	Map	1.48	3.36

Franchise	Name	Stress	Energy
Final Fantasy (Square Enix)	Airship	1.8	4.36
	Clash on the Big Bridge	3.84	5
	Golbez Clad in Darkness	3.96	1.84
	Mana's Mission	2.72	1.56
Fire Emblem (Nintendo)	Fortune Teller	2.83	3.46
	Inescapable Fate	3.44	2.52
	Main Theme	1.73	3.1
	Reminiscence	2.23	1.66
	Together We Ride	3.1	3.42
Legend of Zelda (Nintendo)	Bolero of Fire	3.84	2.68
	Boss Battle	4.15	3.81
	Deku Palace	2.76	3.48
	Death Mountain	3.86	2.55
	Dungeon	4.48	2.96
	Ganondorf Battle	4.44	4.6
	Gerudo Valley	2.55	3.28
	Goron Race	1.39	4.02
	Deku Tree's Last Words	4.83	1.36
	Kakariko Village	1.5	2.2
	Kokiri Forest	1.26	3.81
	Lost Woods	1.72	4.4
	Majora's Theme	4.36	1.36
	Middle Boss Battle	4.7	4.63
	Overworld	2.46	4.86
	Stone Tower Temple	4.36	1.64
	Title Theme (OoT)	1.63	1.13
	Zelda's Theme	1.31	2.07
Megaman (Capcom)	AirMan	1.86	3.15
	Bubbleman	2.53	3.93
	CrashMan	2.36	3.84
	Dr.Wily's Stages (MM 2)	2.28	4.92
	Ending (Megaman X)	4.13	4.36
	GeminiMan	2.94	4.21
	Hardman	2.44	4.76
	MetalMan	3.36	4.7
	Sigma's Stage	3.6	3.36
	SparkMan	2.84	4.28
	Title Screen (MM 3)	2.88	3.32
	We Are The Robots	3	4.96

Franchise	Name	Stress	Energy
Metroid (Nintendo)	Brinstar	1.81	3.31
	Crashed Frigate	2.32	1.24
	Escape Theme	3.55	2.15
	Kraid's Lair	4.16	2.72
	Ridley	4.84	4.92
	Rocky Maridia	3.47	1.34
	Sandy Maridia	4.16	1.33
	Torvus Bog	3.16	2.76
	Tourian	5	5
Pokemon (Nintendo)	Burned Tower	3.4	2.24
	Celadon City	1.4	4.53
	Champion Battle	4.18	3.92
	Elite Four	1.93	3.73
	Lavender Town	3.06	1.53
	Kanto Gym Leader Battle	4.84	5
	Pallet Town	1.4	2.76
	Pokemon League	1.78	1.18
	Radio Tower Takeover	4.4	4.33
	Route 02 Theme	1.48	3.24
	S.S. Anne Theme	1.44	2.97
	Team Rocket Hideout	4.36	4.16
Super Mario (Nintendo)	Castle	3.86	2.36
	Overworld	1.4	4.24
	Overworld (SMB 64)	1.18	4.31
	Underwater	1.3	3.23

Appendix B

Heuristics used in AUD.js Implementation

Though a large portion of AUD.js' generated music depends on the Elimination Tables and Prevalence arrays obtained from classic game music, a number of checks and bounds were added to make generated pieces of music sound 'better'. These heuristics were added to make pieces more coherent, less predictable and less likely to create undesirable music (such as silence, cacophony or bad harmonies). This table lists those heuristics, along with their function and actual values from the source code.

Description	Type	Value(s)
Transitions: every unique patterns ends with an 8 beat transition, which can be either high energy or low energy (50% chance for each). A high energy transition adds 0.5 energy, whereas a low energy transition removes 0.5. Note that energy is typically on a 0 to 1 scale.	Array of booleans (one entry per pattern	True = high energy transition, false = low energy transition

Description	Type	Value(s)
Instrument choices: before any generation is done, the process decides when which instruments will play. For each pattern, one to three instruments may be active.	Two dimensional array of booleans	True = the instrument should play during this pattern, false = the instrument should not play during this pattern. The number of instruments playing is *floor(1.5 + 1.4 * energy + 0.5 * random)*.
Voice limit: once the instruments choices are made, each instrument is given a number of voices to use when generating. An instrument should always be using either two or three voices.	Two dimensional array of integers, one for each pattern/instrument combination	0 if the instrument should not be playing. 2 or 3 otherwise, chosen by *floor(2 + 1.2 * energy + 0.5 * random)*. Which specific voices are used is chosen at generation time, and is randomly chosen to be either 2 or 3 longer note types or 2 or 3 shorter note types.
Pattern Forwarding: in order to create recurring motives, previously generated content may be sent to future, not-yet-generated sections of the piece.	Two dimensional array of integers (one dimension represents all the patterns, the other all the instruments)	Which pattern to copy when generating. When the generation process reaches an instrument/pattern combination where a recurring motive is desired, instead of generating, it will copy the already generated pattern instead. These integers will be a pattern number strictly less than the currently-being-generated pattern, but equal or greater than zero. Besides that, they are chosen at random.
Variation: when the user request patterns repeat a few times, some notes may be changed, rather than just copying the generated pattern N number of times, where N is the requested number of repeats.	Floating Point	For each note in the pattern being copied, if the current pattern being generated is an even multiple (2, 4, 6, etc), each note has a 0.3 chance of being re-chosen, according to the normal generation process. For odd multiples of the pattern number (3, 5, etc), there is only a 0.1 chance.

Description	Type	Value(s)
Percussion values: the odds of a percussive hit being triggered during the generation process is determined by what beat number is being generated.	Floating Point	For average energy, the odds of a snare occurring on beat 1 is 0.98, on beat 5 is 0.9, on beats 3 and 7 is 0.7 and on beats 2, 4, 6 and 8 is 0.3. For the high hat, the second percussive instrument, on beat 1 is 0.7, on beat 5 is 0.6 and on beats 3 and 7 is 0.15 (and 0 for the other beats).
Hard cutoffs: for both Elimination Tables and Prevalence Arrays, entries with very low percentage chances of occurring are eliminated, so that 'bad' options can't ever be selected. The cutoffs for both depend on how long the note would be played for.	Floating Point	For Elimination Tables: for whole notes the cutoff is 0.12, for half notes is 0.05, for quarter notes is 0.03 and for eighth notes is 0.02. For Prevalence Arrays, for whole notes the cutoff is 0.2, for half notes is 0.15, for quarter notes is 0.08, for eighth notes is 0.04.