



# Use of Procedural Audio in Unity

Eetu Tähtinen

BACHELOR'S THESIS  
May 2021

Media and Arts  
Music Production

## **ABSTRACT**

Tampereen ammattikorkeakoulu  
Tampere University of Applied Sciences  
Degree Programme in Media and Arts  
Music Production

TÄHTINEN, EETU:  
Use of Procedural Audio in Unity

Bachelor's thesis 54 pages, appendices 2 pages  
May 2021

---

The objective of this study was to investigate procedural audio as a sound design tool in Unity. Procedural audio, which means real-time sound synthesis, is put into context with other sound design techniques and compared to those by examining its benefits and disadvantages. The practical part consists of a virtual reality gallery project that dives deeper into techniques of creating sounds using procedural audio and ways of creating ever-changing, unique soundscapes in a non-linear environment.

The thesis explores Pure Data, visual programming language for multimedia, that was used to create procedural audio for the virtual reality gallery. Different objects and tools found in Pure Data are thoroughly covered and explained how they can be used for synthesis, waveshaping and alike to achieve complex soundscapes that could be further modified.

In the thesis' practical project, four different rooms created for a virtual reality gallery were taken under inspection on how the procedural audio was created for them and what the steps were to get all the sounds to play in Unity. The thesis provides some general guidelines for creating procedural audio as well as different resources for compiling Pure Data patches to work in different platforms Unity offers.

---

Key words: procedural audio, sound design, Pure Data, Unity

## CONTENTS

1	INTRODUCTION .....	5
2	PROCEDURAL AUDIO AS A SOUND DESIGN TOOL .....	7
	2.1 Procedural audio in the context of other sound techniques.....	7
	2.1.1 Recorded sound .....	7
	2.1.2 Sequenced sound.....	8
	2.1.3 Synthetic sound .....	9
	2.1.4 Generative sound .....	10
	2.2 Real-time creation of sounds.....	11
	2.2.1 Rules and live input .....	11
	2.2.2 Advantages of procedural audio .....	12
	2.2.3 Disadvantages of procedural audio .....	13
3	PROGRAMMING WITH PURE DATA .....	15
	3.1 Introduction to Pure Data .....	15
	3.2 Common Pure Data objects .....	16
	3.3 Building patches.....	16
4	PROCEDURAL AUDIO IN PRACTICE .....	18
	4.1 Project: Kaleidoscopers – First steps .....	18
	4.1.1 Analysing the required sounds .....	18
	4.1.2 Determining the most important sound attributes .....	19
	4.1.3 Devising a strategy to procedurally create the sounds .....	20
	4.2 Project: Kaleidoscopers – Building patches with Pure Data .....	23
	4.2.1 Aramis’s room.....	24
	4.2.2 Misa’s room .....	25
	4.2.3 Minh’s room .....	27
	4.2.4 Anna’s room .....	31
	4.3 Project: Kaleidoscopers – Importing patches into Unity .....	44
	4.3.1 Compiling Pure Data patches into C source code .....	44
	4.3.2 Building Unity plugins .....	46
5	DISCUSSION .....	50
	REFERENCES .....	51
	APPENDICES.....	53
	Appendix 1. VR-Gallery Procedural Audio Compilation .....	53
	Appendix 2. Bubble Sound Comparison .....	54

**GLOSSARY**

ADSR	Stands for Attack, Decay, Sustain and Release. A common type of audio envelope used in music production
BPM	Stands for Beats Per Minute. Tells the number how many beats are played in a minute
Float number	Number with decimal point
Granularity	Very detailed, small part sound consists of
Linear media	Type of multimedia that has distinct beginning and end e.g., a movie
MIDI	Short for Musical Instrument Digital Interface. Standard music technology protocol that connects digital music instruments
Non-linear media	Type of multimedia that allows interaction with the consumer e.g., a videogame
QA	Quality assurance. Tests programs and products to ensure their quality meets the requirements

## 1 INTRODUCTION

Sound has played an important role in games and other non-linear media from the very early days as developers and players have constantly been looking for new ways to provide information to the player through audio to deepen the immersion to games. From the first bleeps and bleeps, sounds in games have gone through massive steps to get to the point where they are now. While there have been various types of audio recording and implementing methods over the years, use of recorded sound has held its place as the industry standard for a long time. Its ease of use, accessibility and gentle learning curve has made it a valid option in nearly every linear and non-linear media. Even though it has its advantages, the ever-growing expectations among consumers require new ways to create content that offers unique, interesting, and interactive soundscapes in the highly contested industry.

To fill the expectations, sound designers may have many different goals when trying to create an experience using sound. These goals could be anything from adding audible feedback such as clicking sound when the user presses a button to creating immersion with a believable, emotionally involving experience where the user can forget about the outside world. There are various kinds of sound design methods that are used to create different types of sounds from music and ambience sounds to sound effects, all of which help to achieve the set goals. All the methods and techniques have their own advantages and while there is no one correct way to create sounds, the flexibility of procedural audio allows to both build sounds from scratch and make use of sound engine inside the digital experience, creating exciting possibilities. (Hillerson 2014, 14).

Instead of using recorded audio material, procedural audio uses real-time sound synthesis which can generate its own sound that is ever-changing. If a sound of many different explosions is needed, one way could be to record tons of stuff, or, instead use complex model of an explosion in sound-generation application that allows to create infinite number of variations. This method could go even further where the model reacts to different parameters controlling the explosion sound, creating even more variation than a library full of explosion samples.

(Hillerson 2014, 14). Creating sounds with this type of procedural approach is based on using the most compact set of rules where large amounts of audio data is created from a few rules. Procedurally creating audio real-time means little data storage which makes extremely small memory footprint, thus making it more performant. Even though procedural audio has its advantages, some might argue that synthetic sound is not as realistic as real-world sounds. (Knox 2019).

## **2 PROCEDURAL AUDIO AS A SOUND DESIGN TOOL**

### **2.1 Procedural audio in the context of other sound techniques**

Instead of using recorded audio material, procedural audio uses real-time sound synthesis which can generate its own sound that is ever-changing. It is often mistaken for procedural sound design which relies entirely on recorded sound and is not capable of generating sound on its own, unlike procedural audio is (Crawford 2018). Both techniques apply to computer generated sound effects and music which have applications in interactive audio systems, particularly in video games, but in other non-linear media as well (Farnell 2007, 1). Procedural audio has been around from the very beginning of game development, in fact it used to be the only way to create sounds as the consoles' hardware were fairly primitive, thus creation of sounds was merely limited to what the hardware was capable of producing (Crawford 2018).

To understand better the concept of procedural audio, it is needed to consider it in the context of other sound techniques and in the terms of recorded, sequenced, interactive, synthetic, generative and AI sound (Farnell 2007, 1). The following section will look at these other sound design techniques and how they compare to procedural audio in how they are used and what makes them different.

#### **2.1.1 Recorded sound**

Traditional audio technology is based on recording where real-world sounds are captured with a microphone and further mixed and processed into a finished form (Farnell 2007, 1). Whether it is a piece of music or sound effect, this type of audio is linear, meaning it has a set duration with fixed beginning and end point. The sound is always the same regarding its content and duration, meaning it has always the same elements in the same order at the same point in time and space (Nil 2019). Even though there are many ways to manipulate the sound such as pitch and playback speed, it does not change the linear

nature of pre-recorded audio material. Procedural audio is different in the way that it is created real-time during the experience. As it does not have a set predefined duration, it is possible to create and manipulate the audio content real-time by changing order and pitch or timbre of its elements however desired, making it non-linear. (Nil 2019.)

Recorded audio consists of data where the values are a time sequence of amplitudes, usually measured about 44,000 times per second. A common technology, sampling, is when these samples are played back from start to finish in the same order and rate in which they were recorded. In recorded sounds, there is always a distinction between the data and the device or program that replays it. Good example would be MP3's which are thought as songs and MP3 players as the devices that turn the data back into sounds, reproducing the recorded sound. Procedural audio, however, may not need to store any data at all, in fact, it can be thought of as just the program which contains the data and means to create audio. (Farnell 2007, 1.)

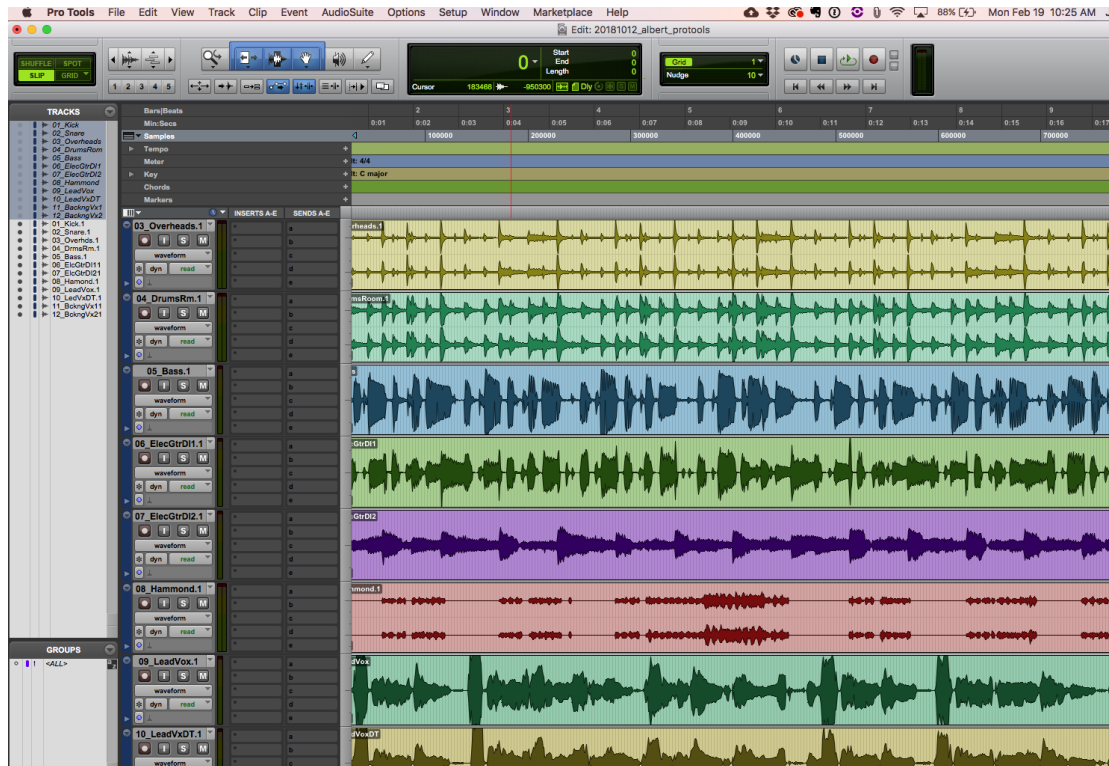
### **2.1.2 Sequenced sound**

Sequenced sound can be thought as of something between recorded sound and interactive sound. Whereas recorded sound is strictly linear, sequenced sound has elements and possibilities for interactivity. This method is widely used in music production today for various genres like hip-hop, rock and pop where the sounds themselves are recorded clips of individual instruments or vocal lines, which are layered together in a sequencer. (Farnell 2007, 2). The sequencer allows to re-arrange the recorded instruments or vocals and play them back in a fixed order, giving it some form of interactivity (Picture 1).

When talking about non-linear media e.g., games, the player can somewhat change the music's or sound's structure through their actions within the game. This has been part of games from the very early days like in Super Mario Bros (Nintendo 1985) where the tempo increases as time runs out on the player. In-game actions can trigger cues that could change the order and timings of the



musical elements and thus has interactive elements to it even though the sounds themselves are fixed. (Collins 2009, 8–9.)



PICTURE 1. Example of a sequenced sound (Loyola n.d.)

### 2.1.3 Synthetic sound

When talking about synthetic sound, it means that sound is created entirely from nothing using equations which convey some functions of time and no other data is needed. Synthesisers, either by software or hardware, produce audio waveforms with dynamic shape, spectrum, and amplitude characteristics which can be used to reproduce real instrument sounds or sound effects, or completely imaginary ones. Combination of sequencers and synthesisers is mainly used in techno and dance music but is also used to create anything from ambient sounds to real-life sounds like rain, wind, or thunder. Through synthesis, it is possible to create just about anything imaginable as long as equations for certain sounds are figured out. (Farnell 2007, 3). This type of synthetic sound is the basis for all the procedural audio.

### 2.1.4 Generative sound

As the name states, generative sound is created through some process that generates the sound instead of a human composing it. The term itself is abstract and includes many others like algorithmic, procedural and AI sound, all of which share the same generative nature and so they are usually talked about as the same thing. Definition for generative sound usually is that it requires no input, or the input is given only initial conditions before execution and with further input, it can alter existing sound or start a new one. If compared to sequenced composition which is laid out in advance and does not change, generative composition happens as the program runs. (Farnell 2007, 3.)

Generative sound can be split into few different approaches: stochastic sound, algorithmic sound, and AI sound. Stochastic sound is based on probabilistic or statistical rules and often uses random or chaotic data, which is subsequently filtered, much like in subtractive synthesis where great amounts of data is present at start which are selectively thrown away to achieve desired sound. Stochastic sound may be generative or interactive since user input can be used to alter the parameters of the generating equation or the filters that operate on the generated data. (Farnell 2007, 4.)

Algorithmic sound refers to a process or system that evolves based on a set of rules. Unlike in stochastic data, rules usually display some sort of order, e.g., based on mathematical functions and numbers that can be used to create melodies for example. In normal computing algorithms are desired to terminate as quickly as possible and with least number of steps to return a value, whereas in algorithmic sound the desired effect is opposite, to keep the algorithm running as long as possible through its steps. With just few lines of code or characters, algorithmic sound can define many hours of evolving sound. Like in synthesis, algorithmic sequencer uses equations that create functions of time, however, unlike waveforms, they are rarely periodic. Where synthesis is usually about sounds produced at waveform level under careful control, algorithmic sound is about the data that is used to control these waveforms in the matter of creating harmonies and melodies. (Farnell 2007, 4-5.)

AI, artificial intelligence, sound is referred to a class of algorithmic methods that are more complex than the mathematical sequences. All algorithms have memory to some degree, to store intermediate variables like few of the last values computed which are usually discarded right away to save memory. The AI system gains intelligence by learning from new inputs and so the synthesis process is modified as new experiences occur. As AI sound has extra knowledge data, some might argue that it breaks the definition of procedural sound being purely a program. (Farnell 2007, 7.)

## **2.2 Real-time creation of sounds**

Procedural audio and sound design expert, Andy Farnell, stated that “Procedural audio is non-linear, often synthetic sound, created in real time according to a set of programmatic rules and live input.” (Farnell 2007, 14). Unlike traditional recorded audio, procedural audio does not have a set duration with fixed ending, meaning it is non-linear and to some degree, time independent. Sound is generated real-time by computers, synthesizers and alike which can be programmed, enabling to split the audio object into small elements which gives new opportunities for interactivity that traditional methods do not. (Nil 2019). Following sections will look through methods of creating sounds real time, as well as the advantages and disadvantages of them.

### **2.2.1 Rules and live input**

Procedural audio is based on a system which takes input and maps it to some output following certain, set rules. When talking about games, input could be regarded as the initial state that can change based on the player actions. It could represent a physical quantity like velocity or a game state like proximity to some object or actor (Farnell 2007, 12). The input is passed through a set of programmatic rules, an algorithm, that is sort of a recipe to define the properties of the audio object that is created. The properties and behaviours of this created object can be varied through live input from the user or through an autonomous system that does not need user input to change. (Nil 2019). After the input is

passed through a set of rules, it is mapped to an output which could be either an audio signal or a trigger for other audio signals (Farnell 2007, 12).

The followed rules can be split into two parts: control structure and synthesis structure, where some set of control data plays the synthesis structure. One could think of a piano and its player, where the player is control structure that basically controls what is played, whereas the piano is the synthesis structure, creating the sounds (Picture 2). This type of control and synthesis structure allows for a very high compression ratio where large amounts of audio data is created from a few rules. This aim to create sound using the most compact set of rules makes the key difference between purely synthetic and procedural audio. (Knox 2019.)



PICTURE 2. Visual representation of control structure and synthesis structure (pngarea n.d.)

### **2.2.2 Advantages of procedural audio**

There are many benefits when it comes to using procedural audio as a part of sound design in games or other non-linear media. Much of it comes to computer performance as creating audio on-the-fly means little data storage, saving vast amount of computer memory (Crawford 2018, Fournel 2012). By using procedural audio, the need to prepare and record multiple separate sound files is gone. In traditional sound design methods, the sound files always have the same audio characteristics when triggered, which makes them repetitive unless

multiple iterations of the same sound are made (Nil 2019). Procedural audio, however, can provide almost infinite number of variations to sound effects with a minimal memory usage (Fournel 2012.)

Asset management becomes easier when there are not thousands of files that need to be tagged, edited, or normalised which means faster development after creating few algorithms to your toolbox. Because of the nature of procedural sounds, they do not really have beginning and ending like linear audio files have, which makes them easy to play on a loop. In a practical study by Berrak Nil, it was found that the procedural approach gave more flexibility when designing simple air conditioner system with three different speed options for the AC. By first creating an algorithm, parameters and necessary audio components, procedural approach gave the opportunity to try different values for different speeds easily without having to record completely new audio files each time a new speed option was added. (Nil 2019.)

As procedural audio is artificially generated, it can provide a great number of possibilities for interactivity. Audio objects can be split into elements it consists of, then use their granularity to control various aspects of a sound. Synthesizing sounds from scratch on the fly gives opportunities for interactivity traditional methods do not, whether based on generative algorithm or user interaction. (Nil 2019.)

### **2.2.3 Disadvantages of procedural audio**

Like many things, procedural audio has its downsides. As procedural audio saves vast amount of memory, creating sound real time has its toll on the CPU, especially when playing several instances concurrently. It is possible to reduce the CPU cost by using procedural audio only on certain sounds, the ones closer to the listener or the ones needing more variation. (Fournel 2012). As audio is created using synthesis, the outcome might not be as realistic when compared to sounds in reality. Besides lack of realism in certain sounds, the learning curve with procedural audio is quite steep and coming up with algorithms might be quite time consuming. (Nil 2019, Fournel 2010).

Implementing a procedural audio system can confront a lot of problems with optimizing and cooperating with other subsystems, such as animation, physics etc., requiring more interaction between sound designers, game designers, programmers, and QA's. Bug testing is a lot simpler when using audio samples as the number of possible bugs is far less as they do not include such great amount of input parameters that could cause problems as in procedural approach. (Fournel 2012.)

There is still a lot of technical and creative issues with procedural audio as it is still lacking standardised development platform and skilled individuals in the big picture. Even though there are ways to use procedural audio in games, there is still no user-friendly middleware or tools to use them with game engines. (Fournel 2012). This might also be due to fear factor where both sound designers and composers worry about procedural audio replacing them as it can generate its own sound without requiring any input (Fournel, 2010.)

### 3 PROGRAMMING WITH PURE DATA

#### 3.1 Introduction to Pure Data

Pure Data is a real-time graphical programming environment for audio, video and graphical processing that is used for composition, audio analysis and sound effect creation for example. In Pure Data patches are created by connecting boxes, most basic units of functionality, together into diagrams that are used to represent the flow of data as well as perform the operations made in the diagram. As the program itself is always running, there is no separation between writing the program and running the program. (Flossmanuals n.d.).





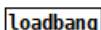
















While traditional text-based programming can be powerful, it is often found non-intuitive among many sound- or visual artists. Being a graphical programming environment, Pure Data can do what the lines of code do by using visual objects that can be manipulated on-screen. Much like analogue synthesizers could use patch cables to produce sounds, Pure Data can create new patches by placing different visual objects on the screen and combining them (Picture 3). These objects can change by connecting them together or sending them messages to do certain functions. This allows for real-time synthesis meaning that changes can be made even when the program is running and so being able to see or hear the results immediately. (Flossmanuals n.d.).



PICTURE 3. Analogue synthesizer patching as equivalent for Pure Data patches (Moog n.d.)

### 3.2 Common Pure Data objects

Pure Data offers a great variety of visual audio objects that can be used to manipulate audio. These include different kinds of messages and numbers that are used for controlling the objects. The practical part will go through a lot of different objects, but for making everything easier to understand when going forward, this section will list some of the most important and widely used visual audio objects (Picture 4).

	Bang - The bang button sends out messages and actions forward
	Toggle - Toggle is used to toggle objects on or off
	Message box - Holds one or more messages. Can be sent out to control objects
	Number box - Can be used to pass number values into objects
	loadbang - Does the same thing as bang but happens when loading up the patch
	metro - Metronome is used to send series of bangs at regular intervals
	random - Generates random numbers from 0 N-1 where N is the creation argument
	delay - Sends a bang to its outlet after a set amount of delay
	osc~ - Sinewave oscillator
	phasor~ - Sawtooth oscillator
	noise~ - White noise generator
	line~ - Audio ramp generator. Can be used to create volume- or filter envelopes
	*~ - Multiplier object. Mostly used to combine signals or to set volume levels
	dac~ - Digital-to-analog converter. Used for outputting the final sound
	sel - Selector object. Takes input value and if it matches argument, outputs a bang
	mtof - Midi to frequency. Changes MIDI note value to frequency
	lop~ - Low-pass filter
	hip~ - High-pass filter
	vcf~ Resonant band-pass and low-pass filter
	inlet - Can be used in subpatch to receive information from the main patch
	outlet - Can be used in subpatch to send information to the main patch

PICTURE 4. Commonly used visual audio objects in Pure Data

### 3.3 Building patches

As mentioned, Pure Data is used to manipulate visual objects on screen, connecting them into a system that produces some desired effect. The visual objects could be referred to as atoms, all of which have their particular job,



whether it is sending messages to other atoms, holding numbers or configuring and controlling objects. Pure Data file is called a patch and it is a textual data file containing information how these atoms are connected and configured. When opening a patch in Pure Data, graphical representation is displayed in its own window. (Hillerson 2014, 15). When going further with the patches, it is good to keep in mind that objects with tilde (~) sign in the end mean that they deal with signals, creating sounds. If the object does not have a tilde sign, it is only used to pass information or numerical values.

In the patch, atoms can be moved around the screen, connected to each other, values adjusted and so on. The visual design allows to easily reproduce patches and understand flow of audio through the system which makes it easier to explain the different functions happening in the patch than if there was only code to look at. Learning Pure Data is a great way to learn how to build sounds in general as the visual programming environment makes it easy to see and understand what is going on. (Hillerson 2014, 15). To successfully build patches, Pure Data mostly uses 3 different atoms: Objects, Messages and Numbers.

Objects are the most important building blocks for everything happening in the patch. On their own, they can already generate sound unlike the other 2. Even though messages and numbers are used to control these objects, they are not necessary for the patch to work as objects themselves can include arguments such as numbers. Messages are used to pass information to the objects, most of the time being number information, like setting a frequency for an oscillator or telling an object to start or stop. Even though messages can be widely used to pass number information to objects, number boxes are better for changing values on the fly. Messages have predefined numeric values whereas number boxes can change their value as frequently as desired. Even though there are more to Pure Data than these 3 atoms, they form the base for everything happening in the patches.

## **4 PROCEDURAL AUDIO IN PRACTICE**

### **4.1 Project: Kaleidoscopers – First steps**

This practical part will demonstrate how I made sounds to virtual reality gallery using procedural audio in Pure Data. The gallery was made as a degree show for fine art students' final tasks. All of them had their ideas on how the rooms would look like and some ideas what kind of sounds or music they would like. I was one of two sound designers creating sounds for this gallery that consists of 12 rooms in total, all of which look different and have corresponding soundscapes. The following sections will present 4 different rooms where the soundscapes varied from ambient sounds to dance music (Appendix 1).

The virtual gallery was built in Unity so first and foremost I had to find a way to implement these sounds into the game engine as they could not be played as audio files due to procedural audio's generative nature. After trying out couple different approaches, I found out that the combination of Unity and Pure Data required least amount of coding to get the sounds working and so I chose Pure Data to create the sounds. Later, I will explain thoroughly how to combine these two programs to easily make procedural system work in Unity.

The thesis will go through all the steps for making the sounds and how everything was planned out on each individual room, however, the following steps will give better understanding how to approach the creation of procedural audio in general.

#### **4.1.1 Analysing the required sounds**

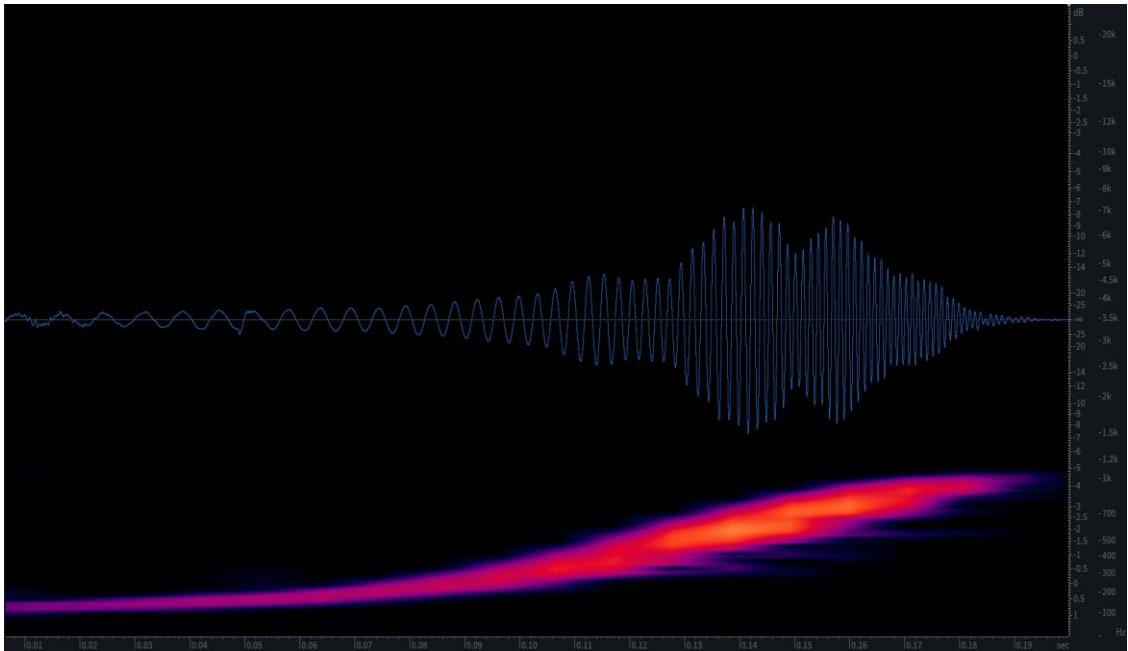
When trying to recreate sounds, whether the sounds are nature-like e.g., wind, animals etc. or something more musical like drums, the ideas of sounds must be broken down to pieces. I will explain more about the individual planning of each room later and so this section will give more of an overview on how to use different kind of listening modes to achieve desired sounds.

According to Michel Chion, theorist and composer, there are 3 modes of listening: causal, semantic, and reduced (Chion 1994, 25). To critically analyse the required sound, one must use a combination of something called causal and reduced listening. Causal is the most common mode of listening, where the point is to gather information about the cause or origin of a sound. If source is not visible this type of listening helps to categorise the source e.g., human, animal, machine, but also form opinions about personality or character for example. If the source is visible, it can provide additional information about the source object e.g., whether a metal can is filled with liquid or if it is empty. (Knox 2019.)

Reduced listening on the other hand focuses on the acoustical traits of a sound, independent of cause or meaning (Chion 1994, 29). By combining causal and reduced listening modes, it is possible to identify the source and evaluate acoustical content and technical quality by focusing on relative balance, frequency content of elements, timing, etc. One example could be a storm sound where causal listening allows to break the sound into originating sources – wind, rain, thunder etc. whereas reduced listening allows to analyse the traits of the sounds: their loudness, dynamic amplitude variations, noisiness, harshness and so on. (Knox 2019). This type of critical listening helps to create the desired sounds, especially if trying to replicate something from real world.

#### **4.1.2 Determining the most important sound attributes**

After critically analysing the required sound using causal and reduced listening, next step is to determine the most important sound attributes or traits in terms of what contributes most to the characteristics of that sound. I will take bubble bursting sound for an example, which happens when a surfacing bubble's surface breaks. When looking at a frequency content of a single bubble bursting, the sound consists of low frequency beginning which ramps up to a higher frequency whereas amplitude slowly rises until the bubble bursts and it decreases again (Picture 5).



PICTURE 5. Visual presentation of amplitude and frequency of a bubble sound

Looking at Picture 5 above, blue colour shows the waveform of the bubble bursting sound whereas purple colour represents the spectrogram of the same sound, showing frequency over time instead of amplitude. As seen from the right side's frequency scale, the bubble sound starts from around 125Hz and exponentially increases pitch up to around 1kHz as the bubble bursts. Time scale on the bottom of the picture shows that everything here happens quite fast as the whole sound only lasts 0.2 seconds.

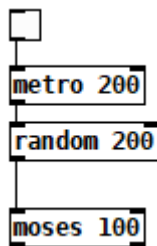
This is only one example and so pitch and speed of the sound of course depends on the size of the bubble and other variables such as liquid elasticity, pressure, height in water etc. When creating multiple bubbles, many concurrent low- and high-pitched sounds for big and small bubbles are played in random timing. (Farnell 2010, 422-423). Using this information allows to proceed to the last part which is for the actual methods for creating sounds.

#### 4.1.3 Devising a strategy to procedurally create the sounds

Last step is to devise a strategy to create the desired sounds algorithmically using software like MaxMSP or Pure Data. I will proceed with the bubble example as the sound attributes that create the sound are covered. The Pure

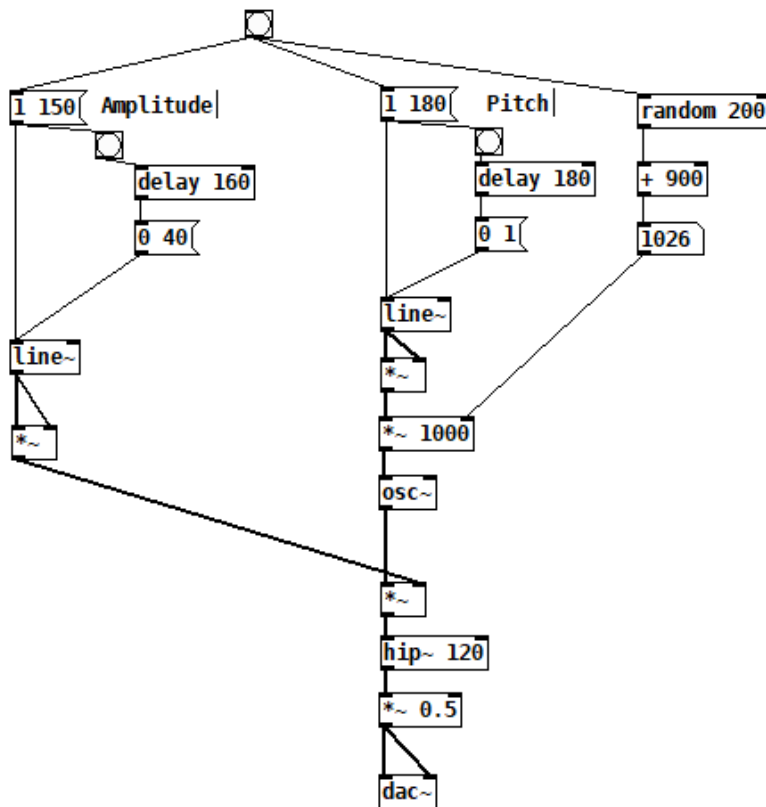
Data patch will consist of forementioned control structure for creating the bubbles on random intervals as well as synthesis structure to create the bubble sounds. The model itself tries to replicate the sound file that was examined before (Appendix 2.)

First thing to look at is the control structure of this patch as it is fairly simple (Picture 6). Starting from the top there is a toggle-object that can either start or stop the whole patch. When turned on, it starts a metronome. The metronome generates a random number from 0 to 199, every 200 milliseconds. The random number is then passed to an object called moses which outputs only numbers that are below 100. As the random-object generates numbers from 0-199, the moses lets through only every second number on average, creating randomness for the intervals between bubbles.



PICTURE 6. Control structure of bubble sound generator

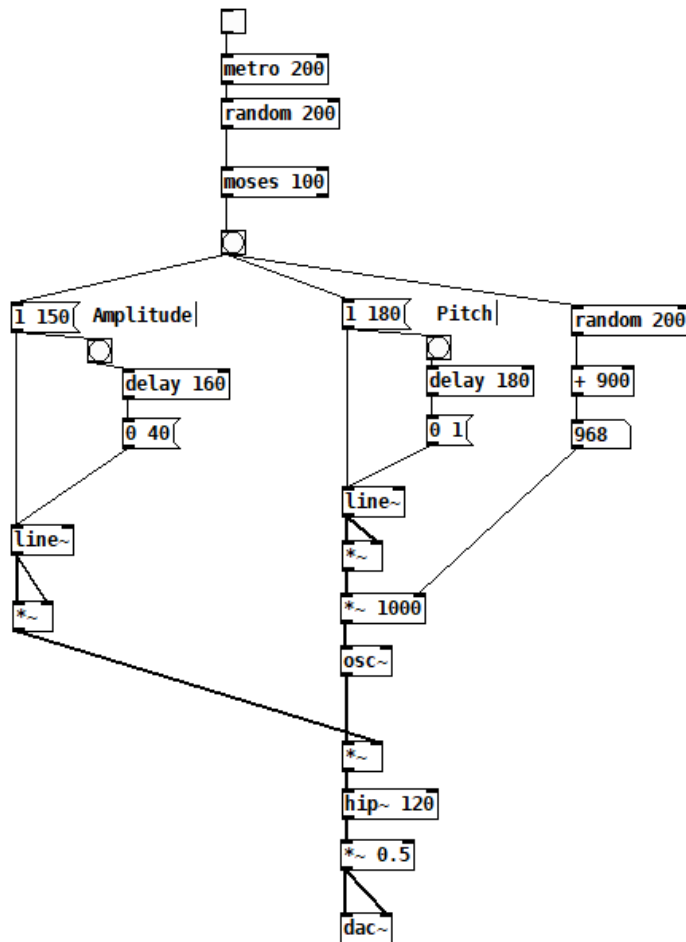
Synthesis structure is a bit more complicated (Picture 7). Starting from the top again, there is a bang object which is pretty much the basis for everything happening in Pure Data. Every time a bang happens, it sends out messages and actions forward. After a bang there is an amplitude envelope on the left side of the picture that tries to replicate the waveform that was examined before. It has a line~ and \*~ object which together create an exponential ramp for the amplitude. The messages line~ receives, make the amplitude go to value of 1, meaning full amplitude, over a time of 150 milliseconds. It once again creates a bang with a delay of 160 milliseconds after which the value goes back to 0 over a time of 40 milliseconds. If thinking about basic ADSR envelope, the attack is 150 milliseconds, after which the sound sustains at the same level for 10 milliseconds and then decays down to 0 over 40 milliseconds. This makes the whole length of the sound 200 milliseconds or 0.2 seconds, the same as in the first spectrogram example.



PICTURE 7. Synthesis structure of bubble sound generator

Now looking at the middle section, there is the same idea happening, instead exponential envelope is created for the pitch. It goes to value of 1, meaning highest pitch over a time of 180 milliseconds. Right after it reaches full pitch the value goes back to zero so that every time new sound is triggered, it starts from a low frequency. Once again it goes through line~ and \*~ objects to osc~ object which creates the actual sound. This object generates a sine wave which gets its frequency from the \*~ 1000 object. The section on the right once again generates a random number between 0-199 and adds +900 to it which is sent out to the \*~ 1000 object, creating some randomness for the frequency of the bubble sound.

The signal from amplitude envelope is multiplied with pitch envelope by using \*~ object which is needed every time when combining 2 signals. The combined signal then goes to hip~ object which creates a high-pass filter, cutting off anything below 120Hz. Finally, the whole sound is multiplied by 0.5, halving the amplitude and then fed to dac~, which outputs the signal. The control structure and synthesis structure are then combined, creating the bubble generator (Picture 8).



PICTURE 8. The whole structure of bubble sound generator

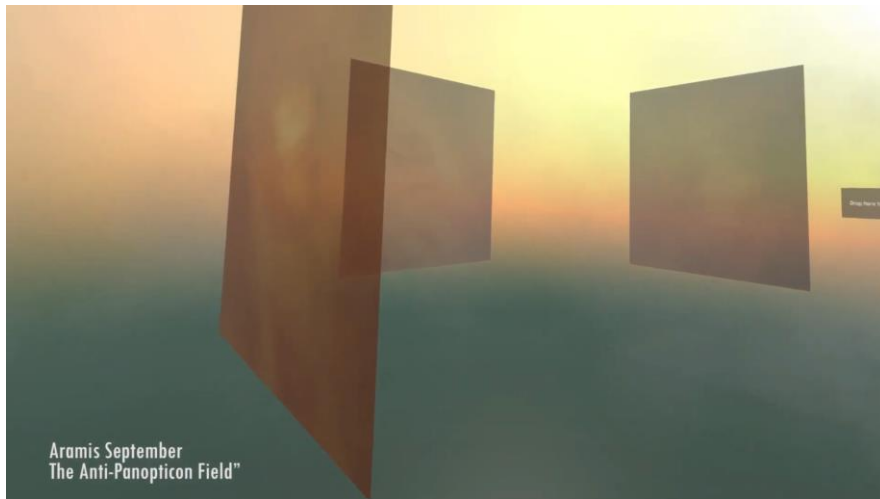
While this may seem complicated at first, using the forementioned objects can already create very advanced soundscapes with procedural nature. I will go through some more objects later on, but these will cover the basics most of the time.

## 4.2 Project: Kaleidoscopers – Building patches with Pure Data

This section will go through all the Pure Data patches I made for the VR gallery with explanations of why I did what I did. I will start off with more simpler patches and progress towards the advanced patches to make it easier to follow what is happening.

### 4.2.1 Aramis's room

For Aramis's room, he wanted something soft, gentle, and ambient for the music, so it would not be anything too attention drawing (Picture 9).

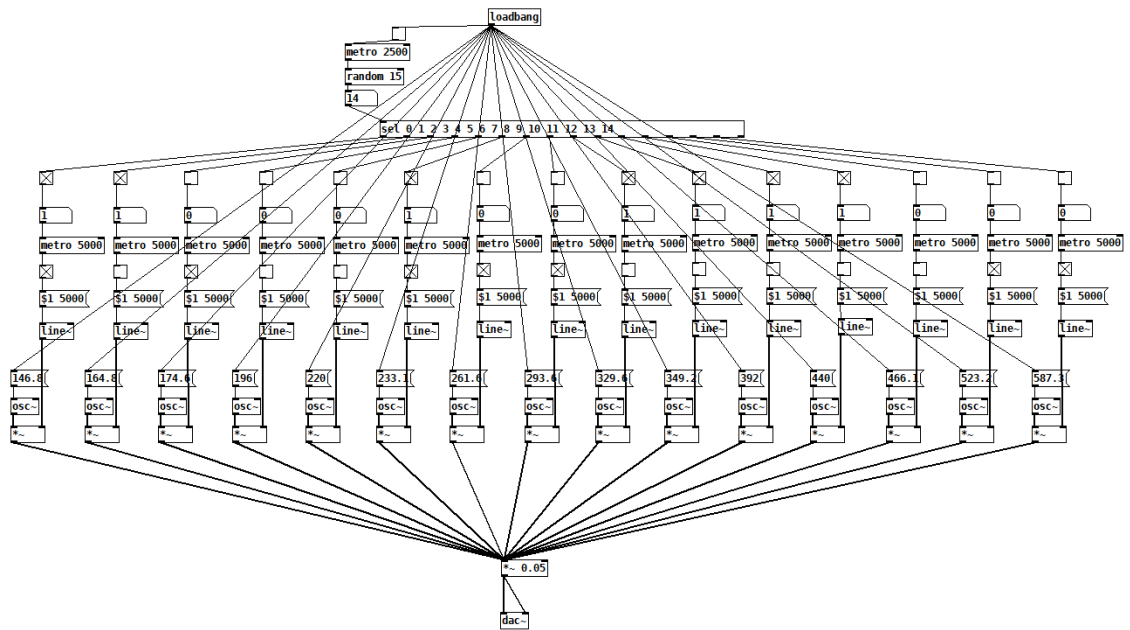


PICTURE 9. Screenshot of Aramis's room

Starting from the top, loadbang object is used to output a 'bang' message when the patch is opened to start everything up. It starts a metronome that generates a random number from 0 to 15 every 2,5 seconds. This number is passed on to select object that toggles on a metronome based on which number is generated. If the generated random number is 0, it starts the leftmost metronome and if the generated random number is 15 it starts the rightmost metronome and so on. The second time the same number is randomly generated, it toggles the metronome off, so all the sounds are not playing at the same time. This metronome together with a line~ object controls the volume of a sinewave oscillator that goes up and down on a 5 second span, creating soft sounds with long attacks and decays.

All the sinewave oscillators have the same idea on how they work, the difference being they all have different frequencies. Starting from the left, frequencies rise so that they create notes on D, E, F, G, A, B $\flat$  and C. These combined create a D minor scale that is played in 2 octaves. All of the oscillators are combined with a \*~ object that multiplies the volume with 0.05, making the overall level sensible (Picture 10).





PICTURE 10. Pure Data patch for Aramis's room

#### 4.2.2 Misa's room

Misa did not have any suggestions regarding the sound, but the main idea was a colourful room with toys that have changeable body parts and so I wanted to make something creepy and oppressive (Picture 11). The patch is split into two parts: distorted drone and warm drone. The idea of distorted drone is to have a lot of randomness in sounds and volume levels whereas warm drone is used to fill up the empty spaces and give warmth to the sound.

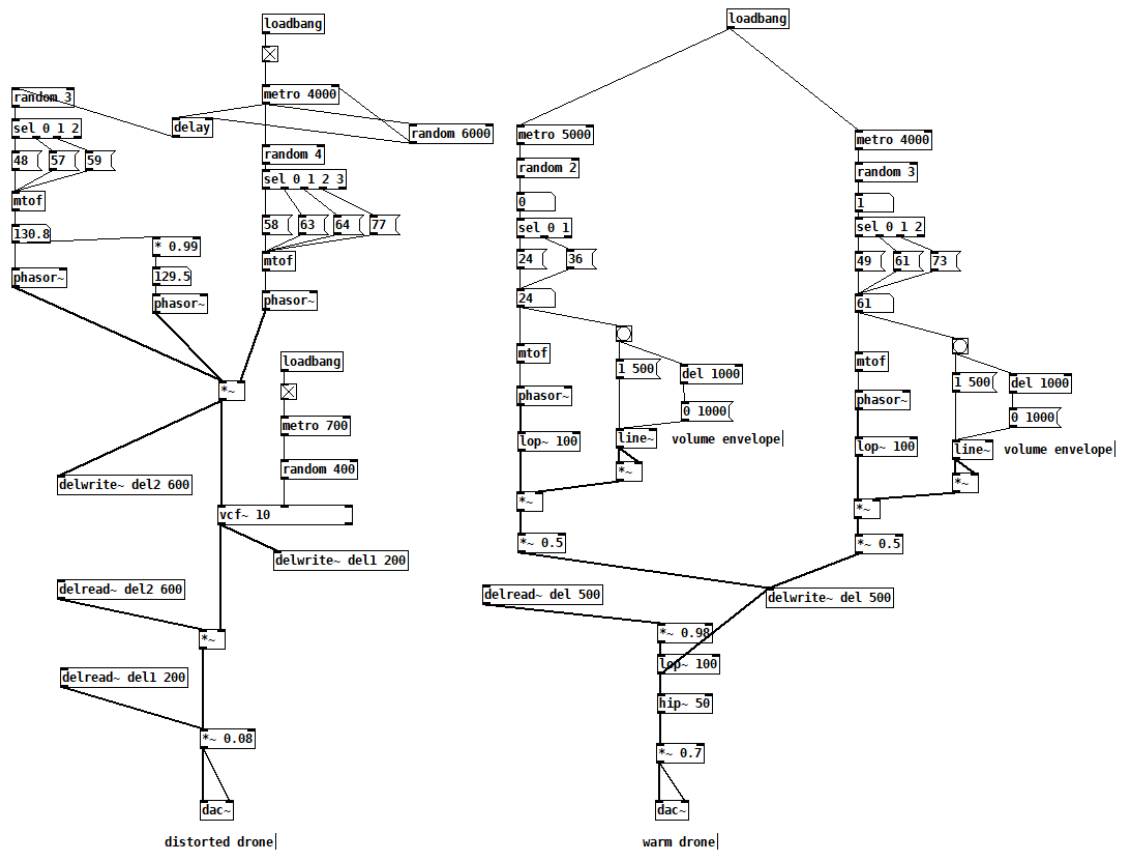


PICTURE 11. Screenshot of Misa's room

Distorted drone once again starts with loadbang that turns on a metronome. This time the metronome is controlled by a random object that changes the metronome interval. The metronome connects to a random object that plays 1 of 4 notes. It goes to mtof object which is an easy way to change midi values to frequencies. The frequencies are then passed on to a phasor~ object, which works much like osc~ object that was covered before, however, instead of creating a sinewave, it creates a sawtooth wave. This phasor~ is combined with another set of phasor~ objects that are played after a randomized delay. The phasors are pretty much alike, however, the other one has a tiny difference in the frequency making them sound slightly detuned.

The combined signal is passed on to vcf~ object that creates a resonant band-pass and low-pass filter. The filter's center frequency is changed to a random value between 0 and 399 every 0.7 seconds to give lot of variation to the sound. Few delwrite~ objects are used to create delays to the signal line which are later read in the signal by the delread~ objects. Finally, the signal is multiplied by 0.08 to reduce the volume before outputting the sound.

The 'warm drone' part of the patch has a loadbang that starts 2 metronomes, one with 5 second intervals and the other with 4 second intervals, creating varying rhythm for the sounds. They both choose randomly from few different note values that are once again passed through mtof object to phasor~ object that creates sawtooth wave with changing note. It goes through a lop~ object that creates a low-pass filter at 100Hz after which the signal is fed onto \*~ object and combined with a volume envelope. The volume envelope has the same idea as in the bubble sound generator example; every time the note changes, the volume goes to full value over a span of 0.5 seconds. After another 0.5 seconds of sustain, it decays down to zero in 1 second. The signals are then multiplied by 0.5 and passed on to delwrite~ object that once again creates a delay line. Through a couple of low- and high-pass filters, the signal is multiplied to its final volume and fed to output (Picture 12).



PICTURE 12. Pure Data patch for Misa's room

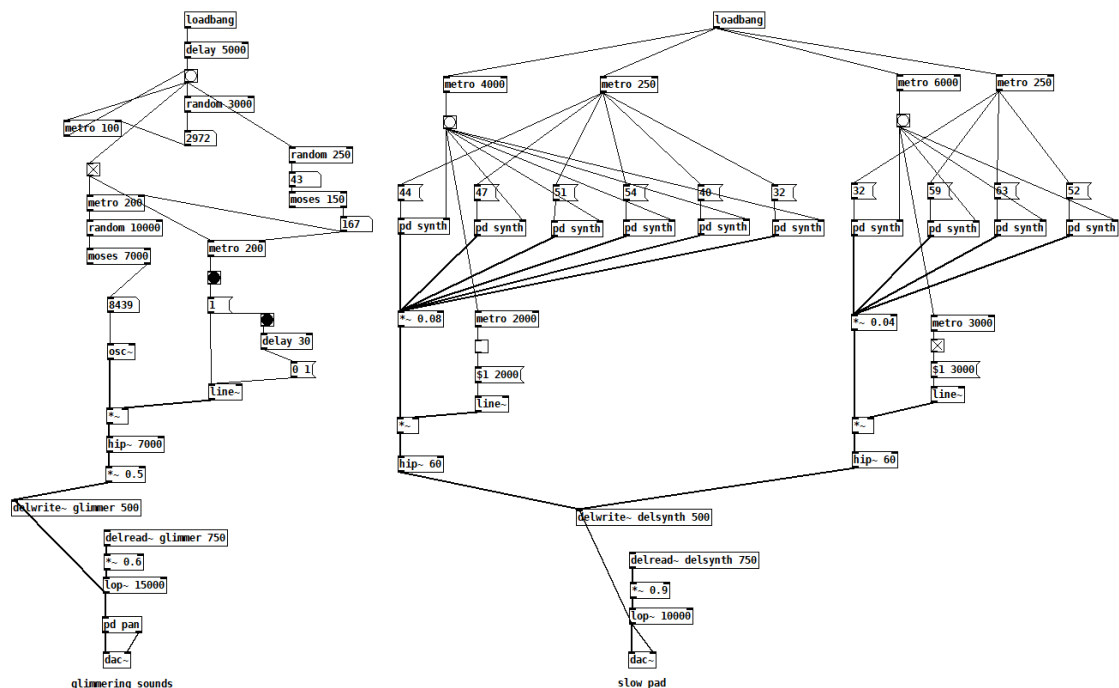
### 4.2.3 Minh's room

For Minh's room, he wanted something slow and atmospheric to create a comfortable and relaxing environment (Picture 13). The main patch is split into two parts; slow pad that has synths with long attacks and decays and a generator for glimmering sounds.



PICTURE 13. Screenshot of Minh's room

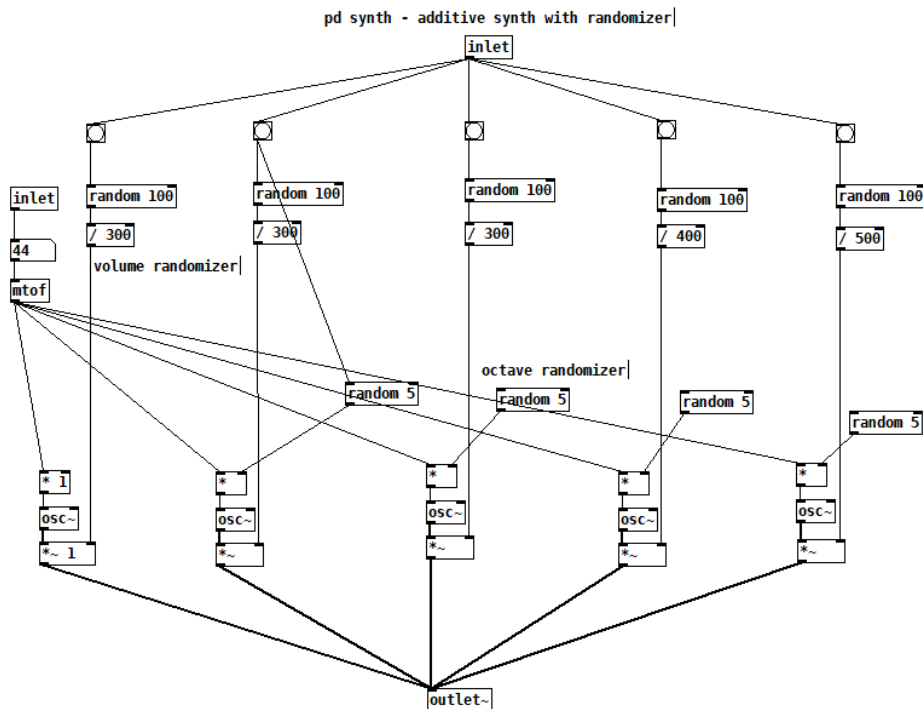
Slow pad starts with loadbang that turns on multiple metronomes. On the left-hand side, the other metronome bangs a note value every 250 milliseconds and the other one creates a bang every 4 seconds that goes to a subpatch called 'pd synth'. The right-hand side of slow pad is pretty much the same, only difference being in the note values of the synths and metronomes that are used to provide more variation to the sound. The 'pd synth' subpatch is used to keep the main patch nice and tidy by having all the necessary objects inside the subpatch. In this case the synthesis structure for the slow pad is held inside the 'pd synth' as their content are all identical to each other and so they would take massive amount of space in the main patch (Picture 14).



PICTURE 14. Pure Data patch for Minh's room

The 'pd synth' subpatch starts with an inlet that receives the bang generated every 4 seconds in the main patch. The patch acts as an additive synth as there are 5 sinewave oscillators that are similar to each other. Firstly, the middle inlet creates a bang which goes into a random object with an argument of 100. The randomized value is divided by 300, 400 or 500, depending on the oscillator, so that the value is sensible enough to be passed to the volume multiplier. The left inlet is used to get the midi note value into the 'pd synth' from the main patch. The note value goes to a \* object that gets multiplied with randomized value from 0 to 4, creating randomized octaves for the oscillators. The signal

with randomized octaves and volumes are then passed on to outlet~ so that the signal is accessible in the main patch (Picture 15).



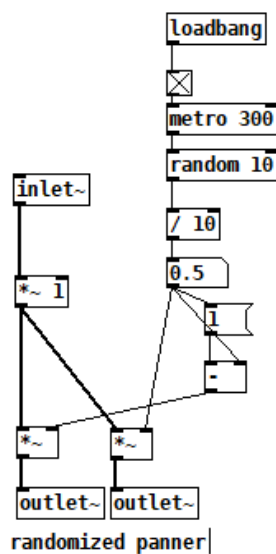
PICTURE 15. The 'pd synth' subpatch

The outlet signals of the subpatches are multiplied with a small value to reduce the overall volume, after which the signal is combined with a line~ object that acts as the volume envelope for the pads. The volume goes from 0 to 1 on a 2 second span and back to 0 on another 2 second span. The right-hand side of the 'slow pad' has a bit different values so that the time for the volume to go up takes 3 seconds and another 3 seconds to go back to zero. The signal is then passed on to hip~ object that creates a high-pass filter to cut out some unwanted low frequencies and finally, after a delay line, the signal is sent to output (Picture 14).

Besides 'slow pad' there is a section for glimmering, magical sound in the patch. It starts with a loadbang after a 5 second delay so that the slow pad gets to start first before bringing in the glimmering sound. The delay goes to random object that generates a number from 0 to 2999, that is used as the metronome's rate in milliseconds. The created bang goes to two objects: metro 2500 and random 250. The metronome is used to create random number from 0 to 9999 that is

used as the sinewave oscillator's frequency. The generated number goes through an object called 'moses' that has a value set to 7000. It works as sort of a floodgate so that any value under 7000 will be disregarded and so only numbers from 7000 to 9999 are passed on to the oscillator.

Now, as mentioned, the created bang goes to a random 250 object as well. It has the same idea that only numbers from 150 to 250 gets passed through moses which will then be used as the following metronome's argument. Using this kind of combination of random- and moses objects allows great amount of randomization for the sound. Like covered before, the metronome goes to a set of objects that are connected to the line~ object that acts as the volume envelope for the glimmering, making the sounds only 30 milliseconds long. The glimmering signal is then passed on to hip~ object to cut any unwanted sounds below 7000Hz. After a volume multiplier, delay line and low-pass filter, the signal is sent to another subpatch called 'pd pan'. It works as a randomized panner for the glimmering sound to go from far left to far right (Picture 16).

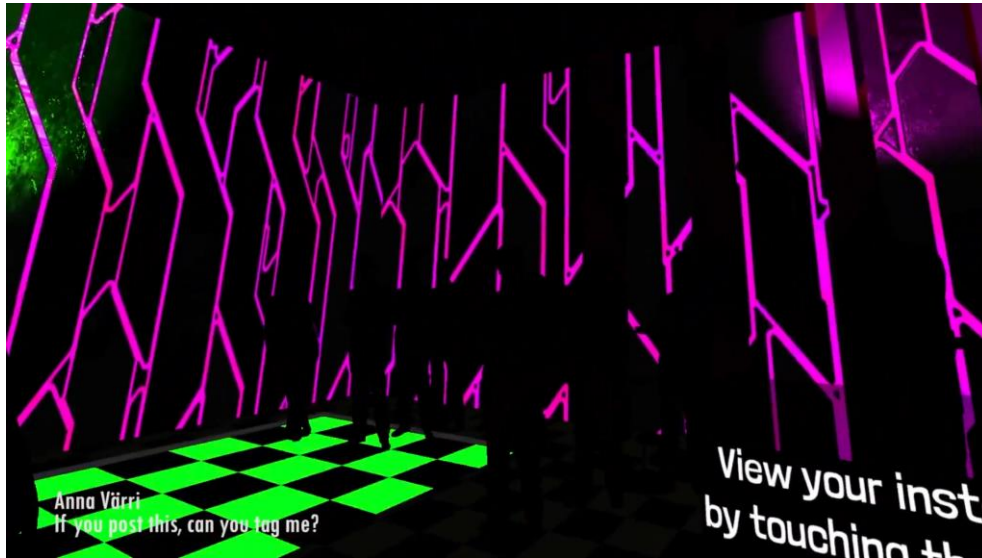


PICTURE 16. Randomized panner for glimmering sounds

The panner creates a random number from 0.0 to 0.9 every 300 milliseconds. The value defines the amount of panning so if the number is 0, the pan is hard left and on 0.9 the pan is on the right, meaning value of 0.4 would be in the middle. Basically, the value defines how much the volume is multiplied by on the right side and the left side is subtracted from that. The sounds are passed through outlets back to the main patch where they are sent to the output.

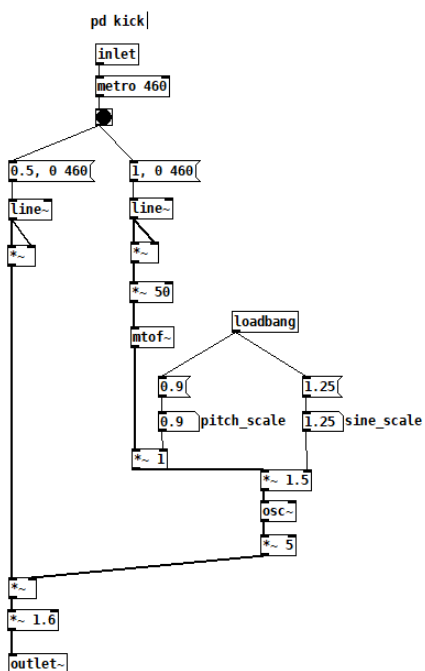
#### 4.2.4 Anna's room

For her room, Anna wanted a dark nightclub feeling with funky dance music playing in the background which was by far the biggest challenge of all the patches due to so many different elements (Picture 17). This patch will be broken down into many smaller pieces that will together form the overall soundscape, starting from the different subpatches that can be found in the main patch.



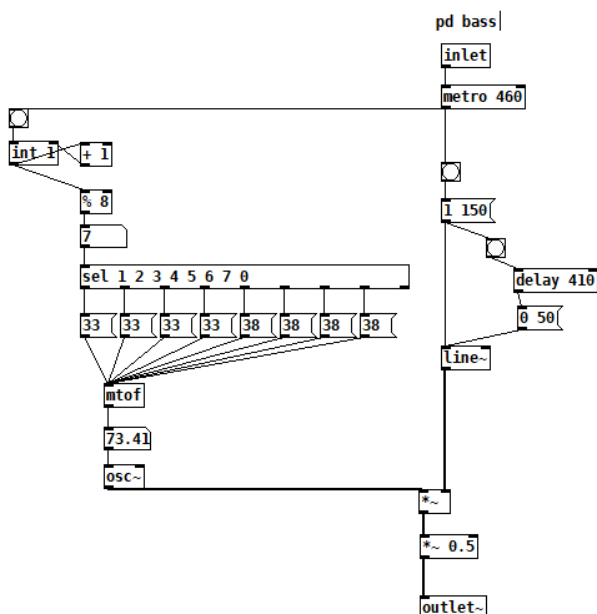
PICTURE 17. Screenshot of Anna's room

The subpatch 'pd kick' starts with a 460 millisecond, i.e., 130 bpm metronome that builds the base for all the rhythmic sounds that are played in Anna's room. The metronome then goes to 2 different directions. The left branch is used to control the volume of the kick as it uses `line~` object to go from 0.5 to 0 over 460 milliseconds. The right branch is used to control pitch over time as it has the same idea with the `line~` object. The value goes to `mtof~` object after which it is multiplied with 0.9 to pitch it down a bit. The pitch scale is combined with `sine scale` and the signal is passed on to `osc~` object to create the actual kick sound. The left branch and right branch are combined and scaled for a reasonable output level before sending them to `outlet~` (Picture 18).



PICTURE 18. The 'pd kick' subpatch

Next up is the bass patch called 'pd bass'. This one is rather simple: once again it starts with metro 460 object that goes into 2 ways. Straight down from the metro is a basic volume envelope where it takes 150 milliseconds for the bass to reach value of 1 to give some more room to the kick. On the left side int 1 and +1 objects are used as a counter that resets every 8 beats. The bass plays a simple 2-note melody created with osc~ object. The signals are then combined before going to the outlet (Picture 19).

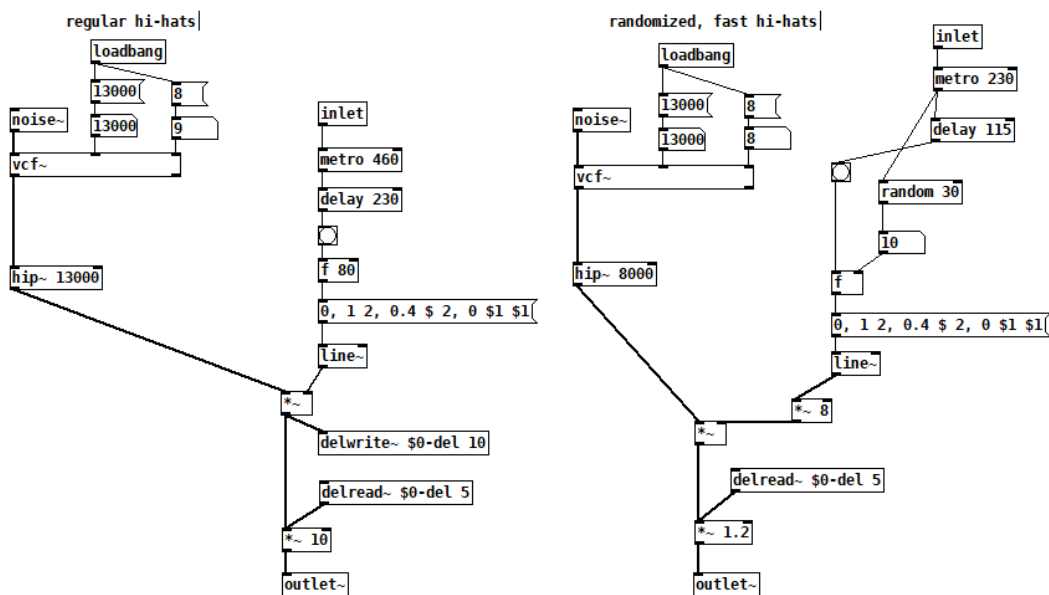


PICTURE 19. The 'pd bass' subpatch



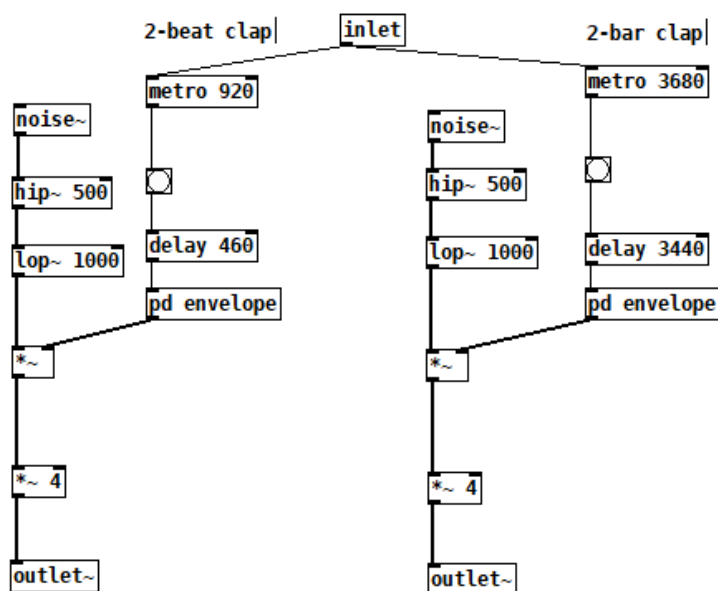
Next one is a subpatch called 'pd hats'. The patch is divided into regular hi-hats and randomized, fast hi-hats. The structure is nearly the same in both, where loadbang is used to give values to vcf~ object that creates a resonant band-pass filter to white noise generator. They are then passed on through high-pass filter to \*~ object which is once again connected to line~ that is used to control volume over time. The regular hi-hats follow the same 130bpm tempo, though it is delayed by half the amount, so it plays in off-beat. The randomized hi-hats have double the tempo instead, to help fill the soundscape. After a bang, float object can be found in the patch that is used to pass certain values to the message line~ receives.

There are \$ signs in the message that receive whatever value there is in the f object and can so change some parts of the message in real-time. The numbers line~ objects receive mean that the value starts at 0. It then goes to value of 1 in 2 ms. Then after 2ms, it goes to 0.4 in X ms. Then back to zero over X amount of time. The X means whatever number the float object has. For the regular hi-hats it is always at 80, whereas in randomized the float value is randomized, creating different length hi-hats. After a delay-line, the hi-hats are passed on to the outlet (Picture 20).



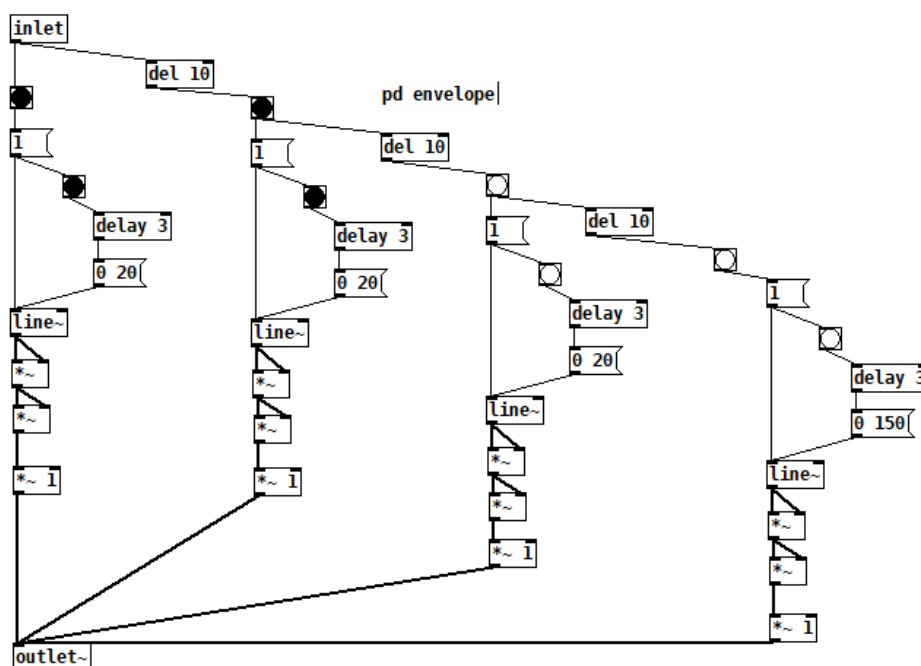
PICTURE 20. The 'pd hats' subpatch

For the last rhythmical element, there is a subpatch called 'pd claps'. It is divided into 2-beat clap and 2-bar clap, which are pretty much identical, main difference being the normal clap happens every 2 beats and the other one every 2 bars to work as sort of a double clap. The claps are generated from white noise that is both high-pass filtered, and low-pass filtered after which they go to \*~ object. This object is connected to 'pd envelope' which acts as the volume envelope for the clap. The 'pd envelope' is triggered in 920ms intervals on the normal clap and 3680ms interval on the double clap after which they are both delayed so that they do not play on the first beat. After some volume multipliers, they are passed on to the outlet~ (Picture 21).



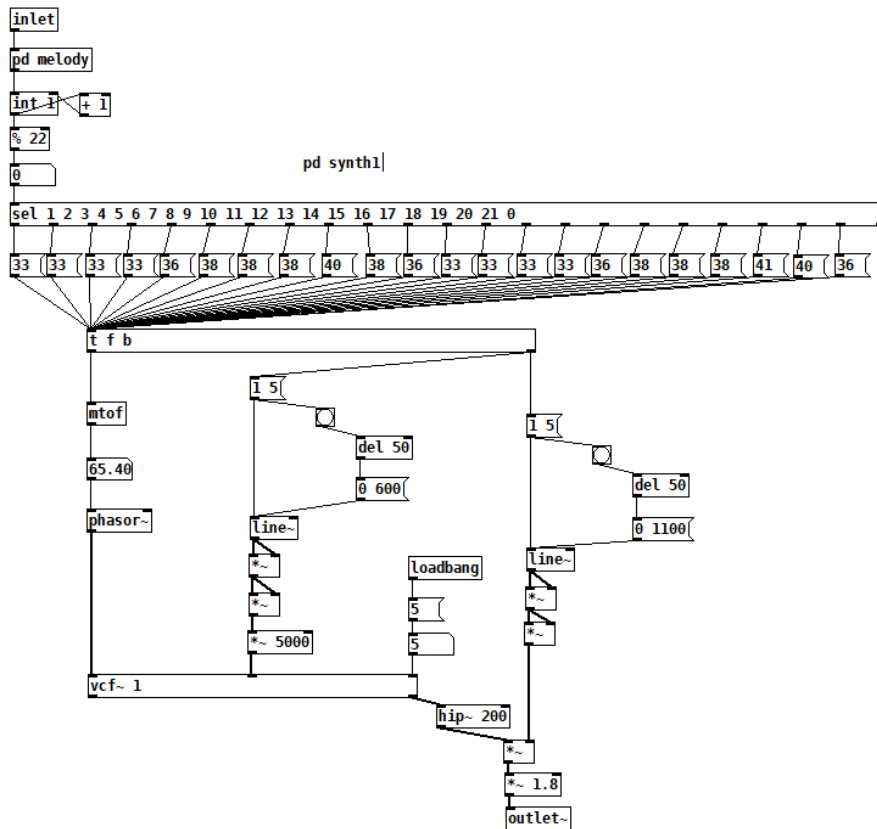
PICTURE 21. The 'pd claps' subpatch

The 'pd envelope' itself is rather simple as well but was made as its own subpatch to save some space. It has 4 nearly identical volume envelopes, all of which happen after a short delay from one another. The first 3 line~ objects start with value of 1 and after 3ms of delay they drop down to 0 in a span of 20ms. These sounds are very short and so they create snappiness to the clap, whereas the last line~ object is a bit different as the decay time is 150ms, giving the clap its body. All the line~ objects go to subsequent \*~ objects that together create a smoother amplitude curve to the sound, before going to outlet~ (Picture 22).



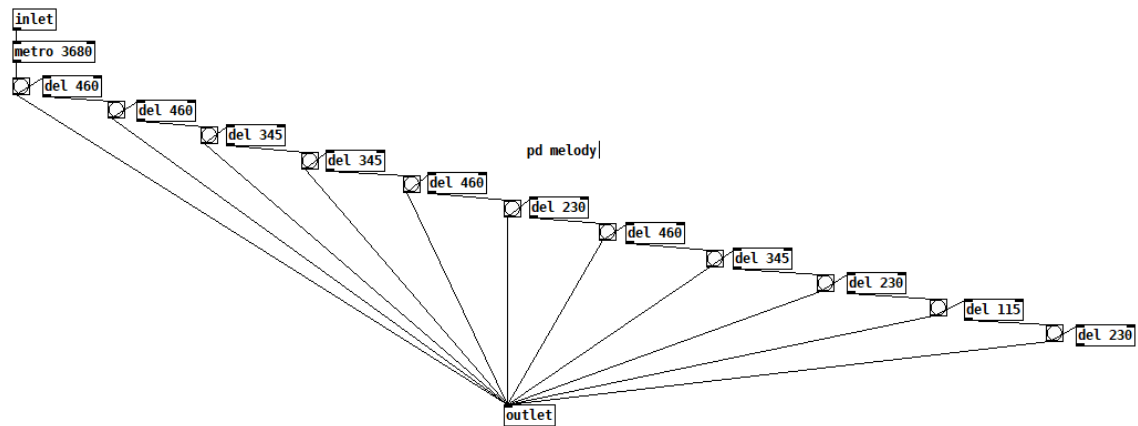
PICTURE 22. The 'pd envelope' subpatch

Now that all the rhythmical elements are covered, the disco needs some synths. There are quite a few different synths playing in Anna's room, all of which follow the same logic with melodies but the sound envelopes or sounds themselves vary. Starting off with 'pd synth1' there is right away subpatch called 'pd melody' that is used in most of the synths. It goes to an int object that once again works as a counter. The counter goes up to 22, after which it resets. Every time a bang is played inside the 'pd melody', it adds a number to the counter. The number then goes to sel object that plays a corresponding midi note depending on which number the counter is currently at (Picture 23).



PICTURE 23. The 'pd synth1' subpatch

Before going further, it is worth to look at the 'pd melody' subpatch to understand how the melody is played. There is a 3680-millisecond metronome, same one as the double clap, that plays the notes on a span of 2 bars (Picture 24). While the patch may look complicated, the idea is very simple. It plays a series of bangs on varying delays. If the delay between bangs is 460 milliseconds it equals one beat, i.e., quarter note. If the delay is 230 milliseconds it equals half beat, i.e., eighth note and so on. All these consecutive notes are played during those 2 bars after which they go to the outlet and so back to the main patch. The bangs happening in this subpatch are summed in the counter and passed on to the sel object where the actual midi notes are played, meaning that this subpatch is only responsible for the timings and lengths of the notes.



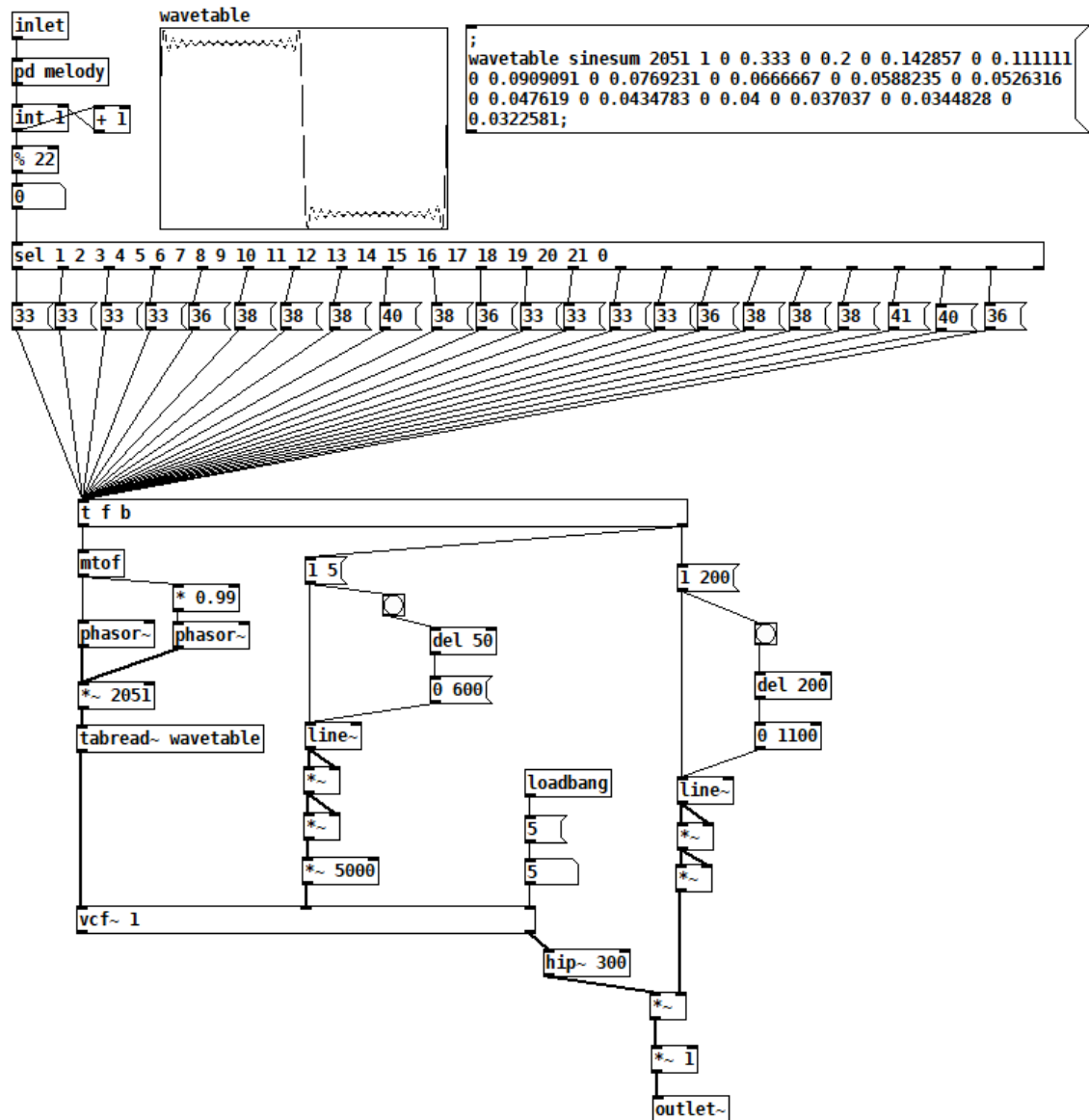
PICTURE 24. The 'pd melody' subpatch that plays a 2-bar melody

Now, going back to 'pd synth1' the midi values go to 't f b' object, which stands for 'trigger float bang'. This object is used to send both float values as well as bangs. The left outlet of the object sends a float number that comes from the sel object, whereas right outlet is used to send a bang forward. The float number is changed to frequency that is passed on to phasor~ object for a sawtooth generator. The signal then goes to previously covered vcf~ object where the center frequency is controlled over time by using the line~ object which allows the use of filter envelope, instead of using envelope just for the volume. The signal then goes to a high-pass filter that is once again connected with line~ object to control the volume envelope. After the volume is set to good level, it is then passed on to outlet~ (Picture 23).

For the second synth, the patch uses something called wavetable synthesis. It is common way of creating synth sounds in DAWs, however, it is a bit more complicated here. Next patch, 'pd wavetable1' has pretty much the same structure than 'pd synth1'. It has the same melody and notes playing, though this time the phasor~ object is affected by tabread~ wavetable.

Pure Data's built-in oscillators such as osc~ and phasor~ are only able to generate sine- and sawtooth waves and so wavetable synthesis is required to create different waveforms. By combining multiple different sine waves, it is possible to generate the waveforms of a saw, triangle, or square wave. When written to an array object, these waveforms can be played back as an oscillator (Flossmanuals n.d.). Next to the wavetable, there is a message saying wavetable sinesum 2051. It basically adds up a bunch of sinewaves that together generate

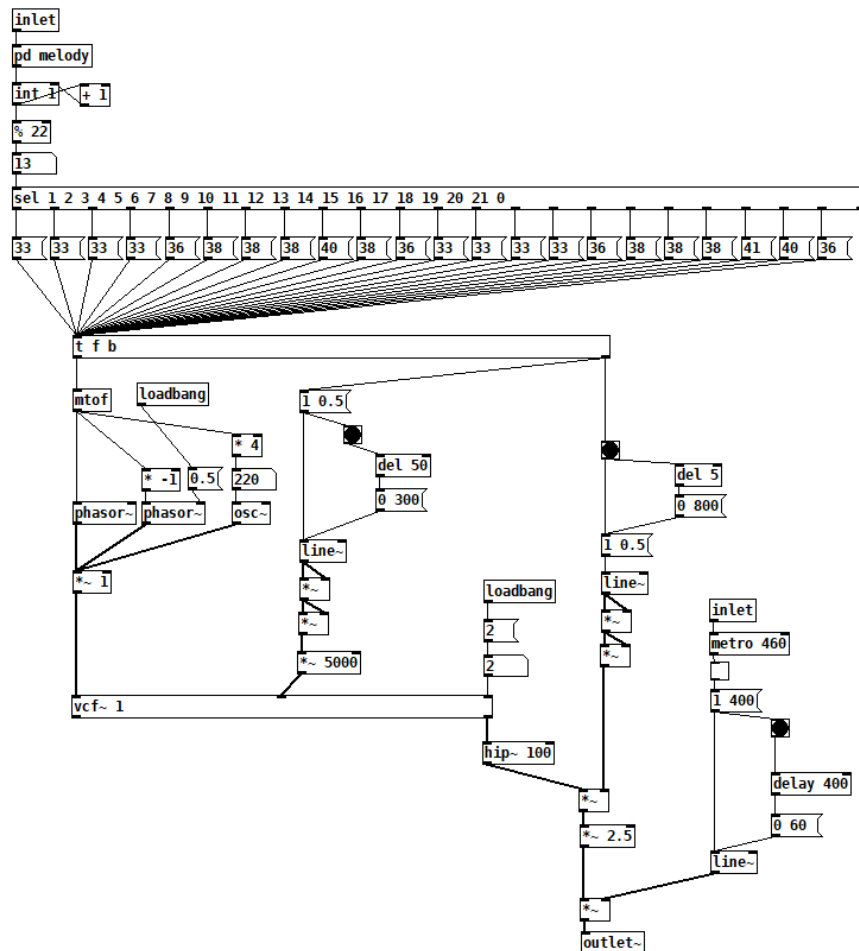
the square wave. This type of waveform is created by only using odd-numbered harmonics such as 1,3,5,7 etc. while even harmonics are set to zero. Without going too deep into the mathematics, the list of numbers starts from 1, then 0, then 0.333, then 0, then 0.2 and so on. The more harmonies are added to the waveform, the more it looks like a square. Once the wavetable is created, the rest of the patch is pretty much identical with 'pd synth1', only with some different values in some places (Picture 25).



PICTURE 25. The 'pd wavetable1' subpatch

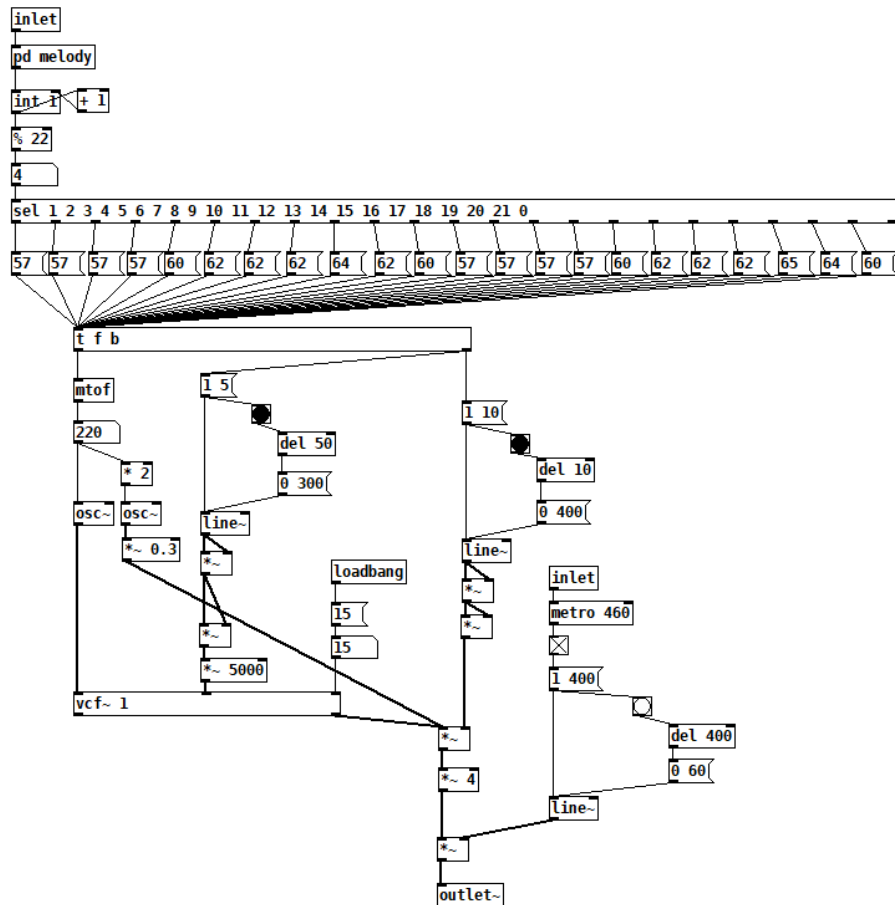
Next synth, 'pd synth2' is nearly identical to the first one. This time there are 2 phasors instead of one, with the other one slightly detuned. Besides phasors there is an osc~ object as well to give more pluckiness to the sound. The filter envelope values are bit different again and there is one extra volume envelope

just before the outlet. The first volume envelope is used for each individual note as the bang that triggers it happens every time the counter changes its number. The extra volume envelope is not dependent of when the notes changed as it always follows the 130bpm metronome. This is used for sidechain type of effect with slow attack and fast decay. This signal then goes to the outlet (Picture 26).



PICTURE 26. The 'pd synth2' subpatch

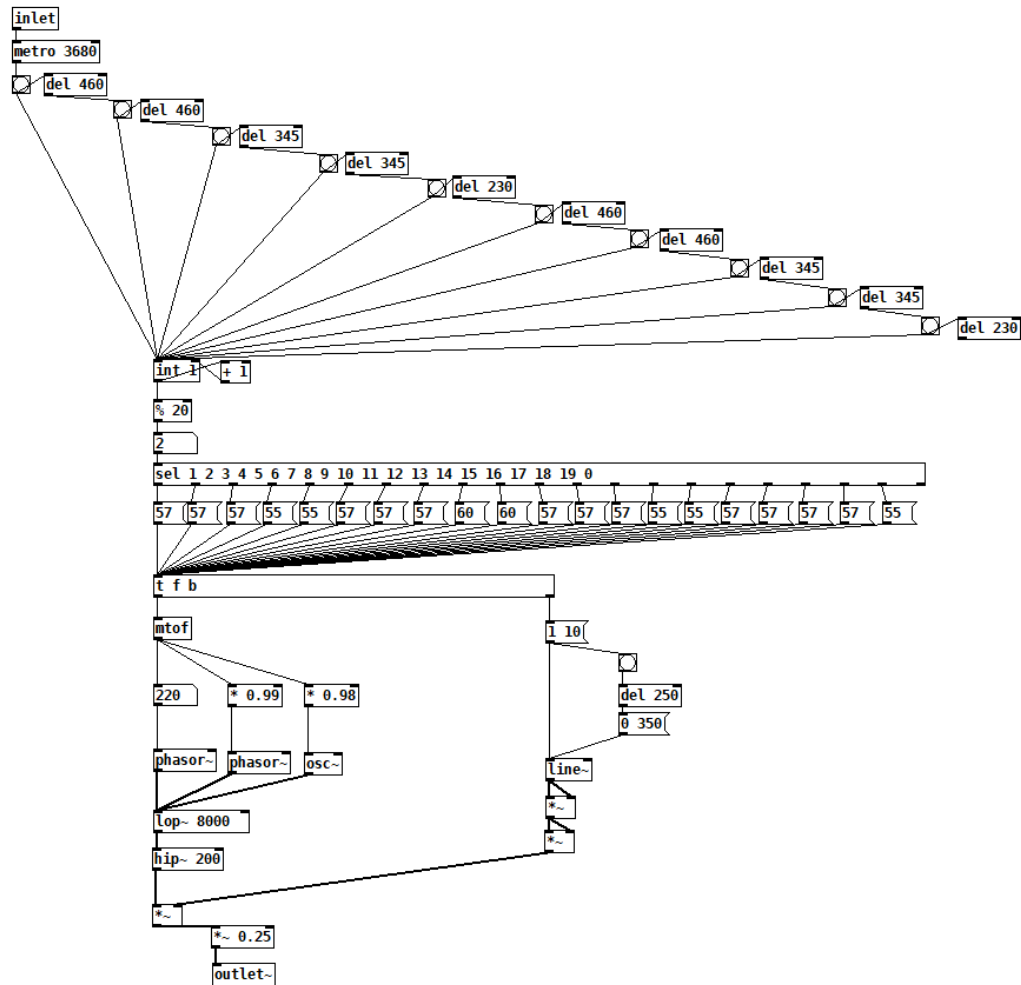
Next, there is a plucky sinewave synth 'pd synth3'. It is nearly identical with 'pd synth2' with extra volume envelope to achieve sidechain type of effect. Instead of phasors it uses 2 osc~ objects with the other one having double the frequency. The first oscillator uses the same path as before, going through the vcf~ filter, however, the double-frequency oscillator goes straight to the multiplier object without going through the filter to keep the overall sound brighter. After multiplying, the signals are passed on to the outlet~ (Picture 27).



PICTURE 27. The 'pd synth3' subpatch

With 2 synths left, 'pd synth4' is a little bit different now. It follows the same idea than the previous ones, however, the melody is a bit different. There are 10 steps in the melody instead of the previous 11 ones and this patch only plays 3 different notes instead of the usual 5. This synth is more used as a background, rhythm synth. The overall structure is quite the same, 2 detuned sawtooths and 1 extra sinewave oscillator with a bit different frequency. After going through low-pass- and high-pass filters, they go to `*~` object that once again connects to `line~` object to create the volume envelope for the notes. After that it is passed on to outlet (Picture 28).



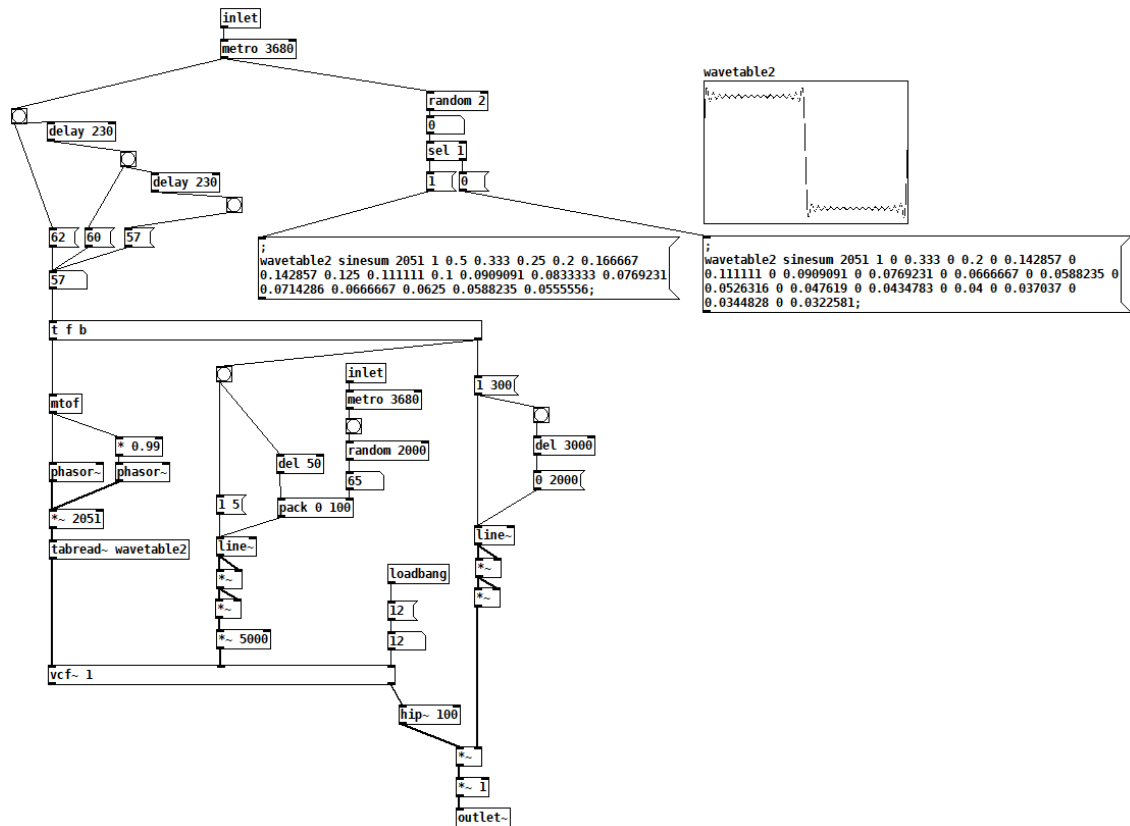


PICTURE 28. The 'pd synth4' subpatch

Now for the last piece of Anna's room. It is a second wavetable synth that plays a 3-note melody where the sound is changing every time it plays. The melody is played every 3680 milliseconds, or 2 bars as usual. From the metronome the line is split into two ways. The right path is where the wavetable functioning is: it chooses randomly whether it plays a square wave or a sawtooth wave. If the generated number is 1, it creates a sawtooth wave and if the number is 0, it generates a square wave. This way the sound has some variation to it every time it plays.

On the left path, 3 descending midi notes are played subsequently that creates the melody for this patch. They go to the usual t f b object and again to the tabread~ object that reads the waveforms generated in the wavetable. The center frequency of the filter has a bit different look than usual as it has a 'pack' object. This works sort of like the \$ sign does, meaning the initial argument can be changed real-time. This pack object works as the filter envelopes decay, where it

is randomized every time the melody is played. With some high-pass filtering and volume envelope, the signal then goes to the outlet (Picture 29).

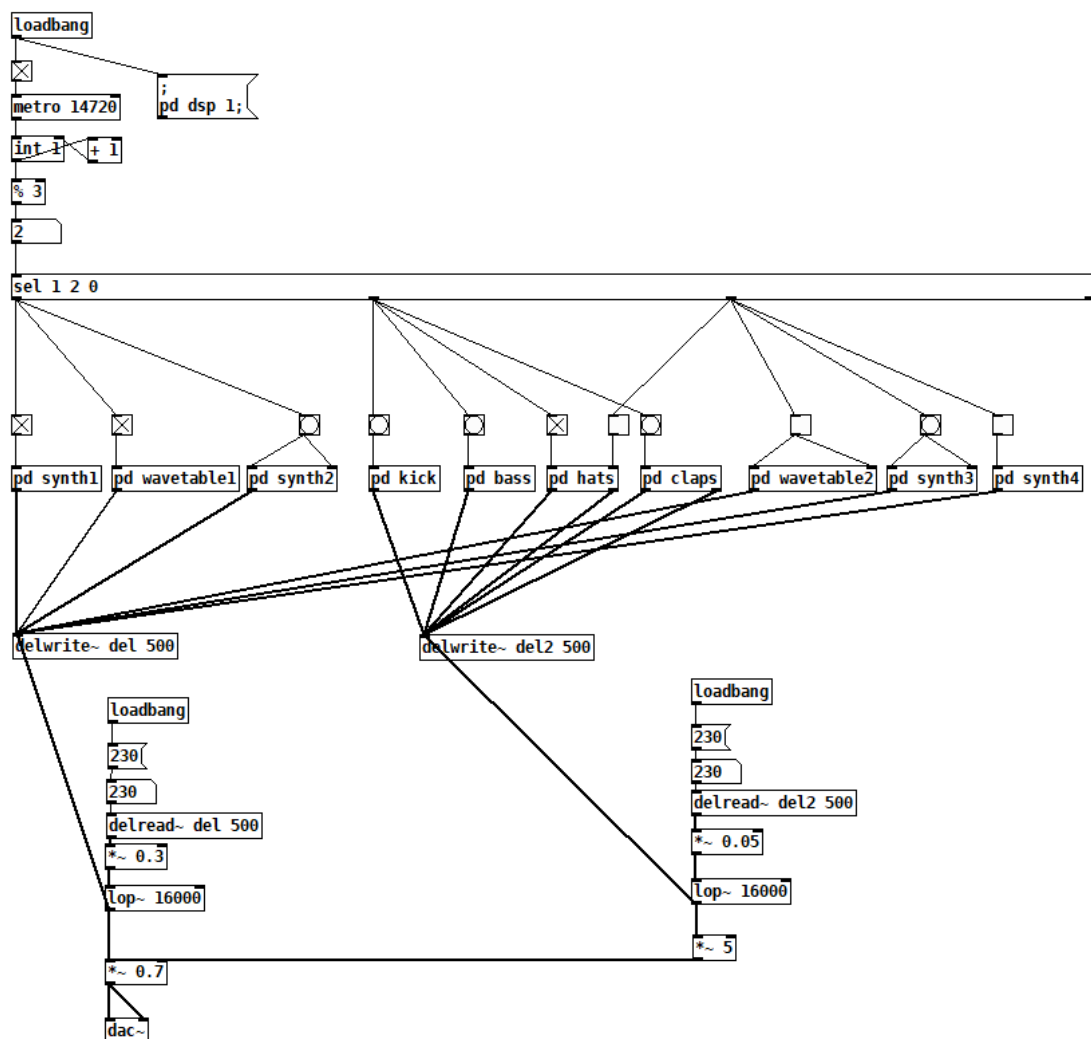


PICTURE 29. The 'pd wavetable2' subpatch

Now, the main patch that sums all the subpatches together is quite simple. It has a metronome of 14720 milliseconds, meaning 8 bars, and a counter that counts from 0 to 2. The counter goes to a selector object which has divided all the individual elements into sections so that everything would not start at the same time and play so forever. The first section has 'pd synth1', 'pd wavetable' and 'pd synth2'. The pd synth1 and pd wavetable are connected to the selector with a toggle object, which starts when the counter hits 0 and stops when it hits 0 again and so on. The pd synth2 has a bang instead so once it starts, it never stops. This is to make sure there is at least one synth playing always.

So once the patch is opened, 8 bars of synths are played alone at first. Then comes in the second section that has all the rhythmical elements; kick, bass, hi-hats, and claps. Here the kick, bass and claps are connected to a bang, meaning once they start, they never stop. The hi-hats are separated so that the normal hi-hats are toggled on in the second section whereas the fast, randomized hi-hats

are toggled on in the third section. This is to keep the elements varying and interesting. For the last section, there is left 'pd wavetable2', 'pd synth3' and 'pd synth4'. Here the pd wavetable2 and pd synth4 are connected to a toggle and pd synth3 is connected to a bang. This variation in different sections and toggles/bangs allows for interesting, varying soundscape so that it is not just an 8-bar loop. All the synths, meaning section 1 and 3 are connected to a long delay that acts almost as a reverb, whereas rhythmical elements are connected to a shorter delay. After the delays everything is set to nice overall level before going to dac~ (Picture 30).



PICTURE 30. The main patch for Anna's room

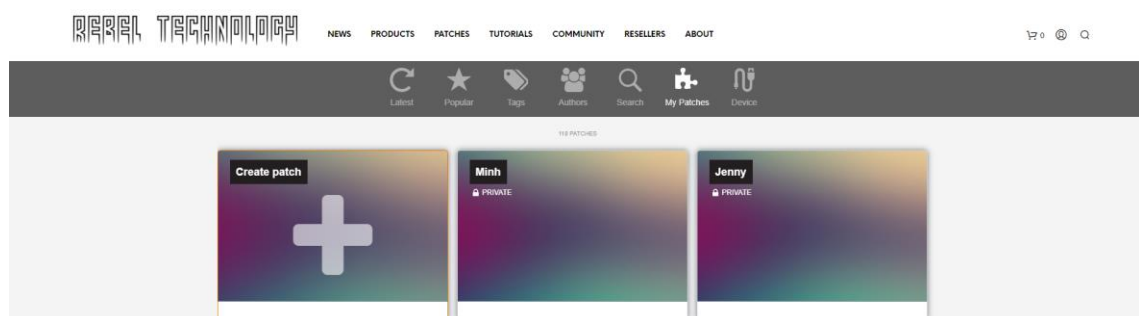
### 4.3 Project: Kaleidoscopers – Importing patches into Unity

Now that all the patches are done in Pure Data, next step is to get them into Unity. In order to use procedural audio in Unity or any other game engine like Unreal Engine, there are few steps that needs to be done for the patches to work. It took quite a lot of trial and error to import the patches into Unity but following these next steps it should be rather simple thing for anyone to do.

#### 4.3.1 Compiling Pure Data patches into C source code

When using procedural audio in Unity, first thing to do is compile the patches into a form it understands. There are couple different ways this can be done but I followed the same routine for all my patches as I found it the easiest way to do that requires least amount of coding. I used an online compiler made by Rebel Technology that is free and open for anyone to use. It turns Pure Data patches into C source code that can be used to build plugins the game engines support (Rebeltech 2018). It is worth noticing that there are some limitations on objects it works on and so some of the patches could have been done in a simpler way at some points but nearly all the objects it does not support, can be created in some other way.

The online compiler offered by Rebel Technology uses Heavy compiler, a python-based dataflow audio programming language that is used to generate C/C++ code (Github 2018). I will go through the steps on how one could compile their own Pure Data patches to work in Unity. First step is to create an account and from there go to 'My Patches' and 'Create patch' (Picture 31).



PICTURE 31. Creating a patch using Rebel Technology's online compiler

After clicking 'Create patch' you need to upload the Pure Data patch that is compiled. At this point, only thing that needs to be made sure is that the Compilation Type is set to heavy (Picture 32).

CREATE PATCH

Patch Name

Name

Compilation Type

Type **heavy**

Add Source Files

Upload files or add files from GitHub.

Upload Files **CHOOSE FILES...**

Supported File Types: c, h, cc, cpp, hpp, s, pd, dsp, maxproj, maxpat, gendsp, soul, soulpatch

GitHub File Url  **ADD**

Source Files

**Minh.pd** Main File **REMOVE**

**SAVE** **SAVE AND COMPILE** **CANCEL**

PICTURE 32. Uploading Pure Data patch and using heavy compilation

Now after clicking 'Save and Compile', you will be taken to patch details page from where you can download the C source if everything is fine with the patches and it does not create any errors (Picture 33).

Latest Popular Tags Authors Search My Patches Device

**Minh**

nacke95

PRIVATE

STAR

DESCRIPTION

INSTRUCTIONS

**A IN** 35 **A**

**B IN** 35 **B**

**C IN** 35 **C**

**D IN** 35 **D**

**E IN** 35 **Exp**

**B1 IN**

**Pushbutton**

**PLAY**

**CONNECT TO DEVICE**

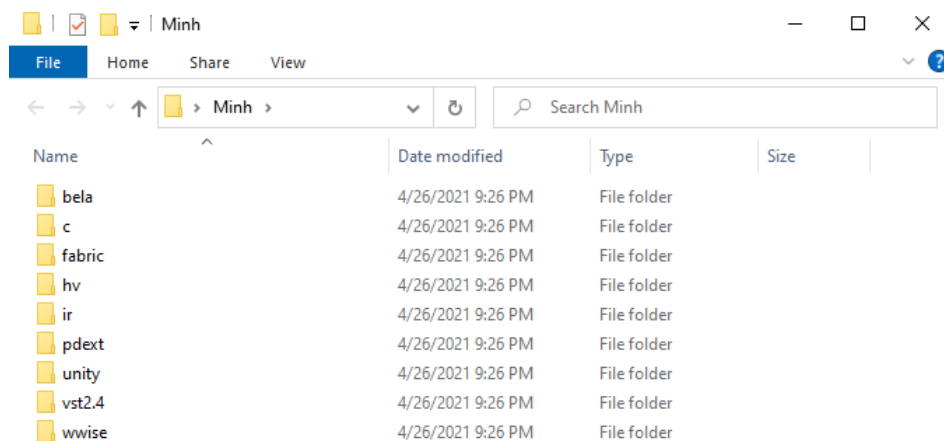
c : [Download](#) (built on 17.3.2021 klo 20.34 26)

PICTURE 33. Created download link for the patch in the bottom left corner

### 4.3.2 Building Unity plugins

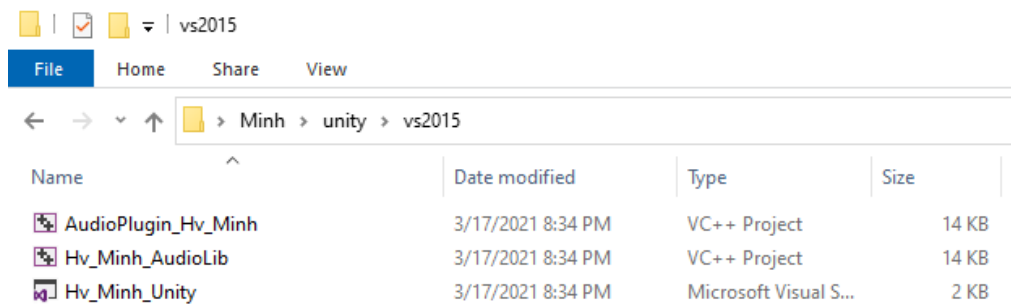
After downloading the C source code, it is time to build the plugins Unity requires. I will only go through the process of creating Unity plugins to Windows, however, there are ways to create plugins for MacOS, Android etc. (Github 2018). There are few things required for the compiling to work; Visual Studio 2015 which is a code editor, and MSBuild 14.0 which is a platform for building applications Visual Studio uses. I will not be going too much into theory with this but only show the required steps to build the plugins.

First step is to download Visual Studio 2015 and MSBuild 14.0. Visual Studio can be installed anywhere on the computer, however, I suggest installing MSbuild in C:/Program Files (x86)/ if you are using a 64-bit version of Windows. After having these installed, it is time to open the folder that was created by using the Heavy compiler. For this example, I will be using the one created for Minh's room, but the procedure is same for every room. The main folder includes all sorts of subfolders, but we are only interested in the unity folder inside of it (Picture 34).



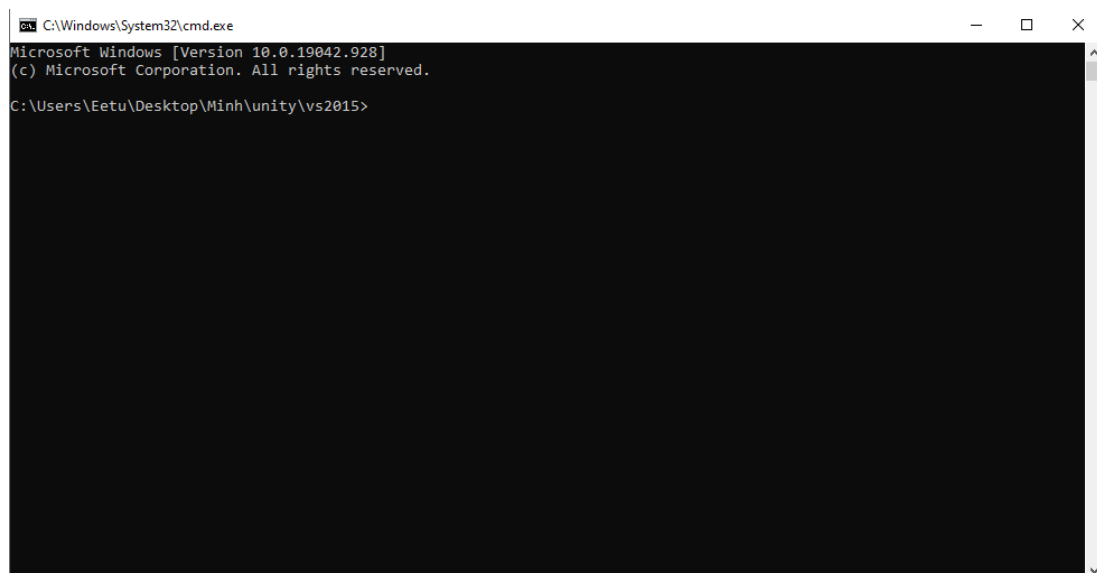
PICTURE 34. The downloaded folder that includes unity subfolder

Inside the unity folder, there is a subfolder called vs2015 that includes 3 objects. AudioPlugin\_Hv\_Minh is used to create Audio Mixer plugin inside Unity that allows to change volume, attenuation and so on for the procedural patch. Hv\_Minh\_AudioLib is used to create the running .dll plugin itself and Hv\_Minh\_Unity creates the C# code that handles all the functions and how the plugins work (Picture 35). These are the necessary objects to create C# code and .dll plugins Unity uses to run procedural audio.



PICTURE 35. Necessary objects to create C# code and .dll plugins

Now, compiling will be done using command prompt. This can be done by clicking the navigation field inside vs2015 folder and typing cmd, that will open up a command prompt with the directory it was opened from (Picture 36).



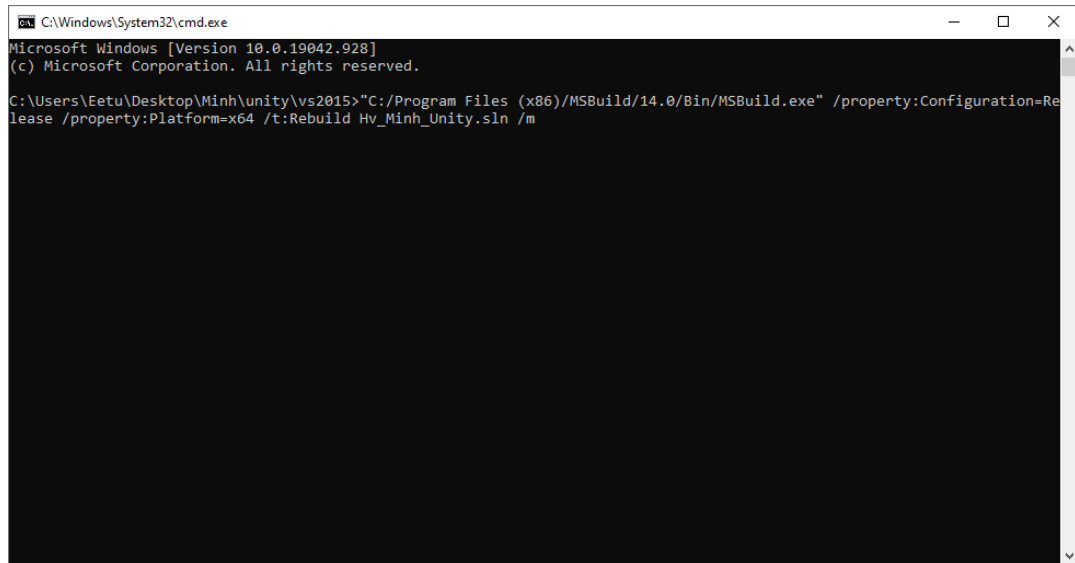
PICTURE 36. Opening a command prompt for the vs2015 folder

Next step is to paste a line of text that basically tells MSBuild to build the needed plugins. Right after the existing path, this line of text can be pasted into the command prompt:

```
"C:/Program Files (x86)/MSBuild/14.0/Bin/MSBuild.exe"
/property:Configuration=Release /property:Platform=x64 /t:Rebuild
Hv_Minh_Unity.sln /m
```

It is worth noticing that if the MSBuild exists in another path, it must be replaced with the correct one. Also, the 'Hv\_Minh\_Unity.sln' must be replaced with

whatever the last object is called in the vs2015 folder. After pasting the text, command prompt should have all the necessary information in order it to compile (Picture 37).



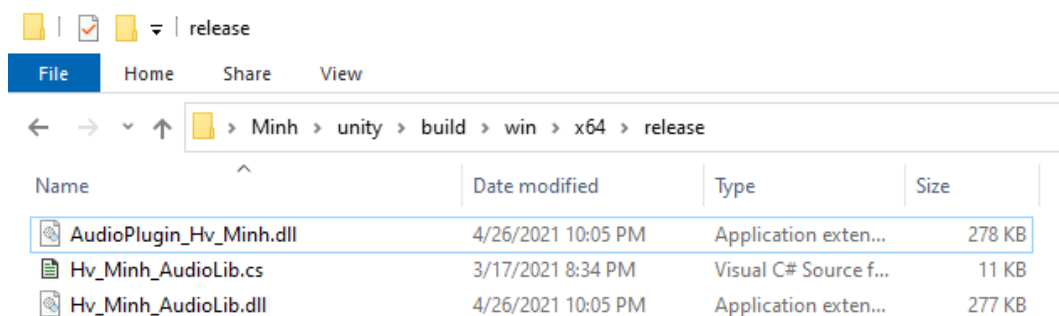
```

C:\Windows\System32\cmd.exe
Microsoft Windows [Version 10.0.19042.928]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Eetu\Desktop\Minh\unity\vs2015>"C:/Program Files (x86)/MSBuild/14.0/Bin/MSBuild.exe" /property:Configuration=Release /property:Platform=x64 /t:Rebuild Hv_Minh_Unity.sln /m
  
```

PICTURE 37. All the information needed in command prompt

Now pressing enter, it should compile the information and if everything goes correctly, after some seconds it tells the build succeeded. The results of the build are placed inside the unity folder, inside a subfolder called 'build'. From there you can now find one script (.cs) object and two plugins (.dll) that Unity needs (Picture 38).

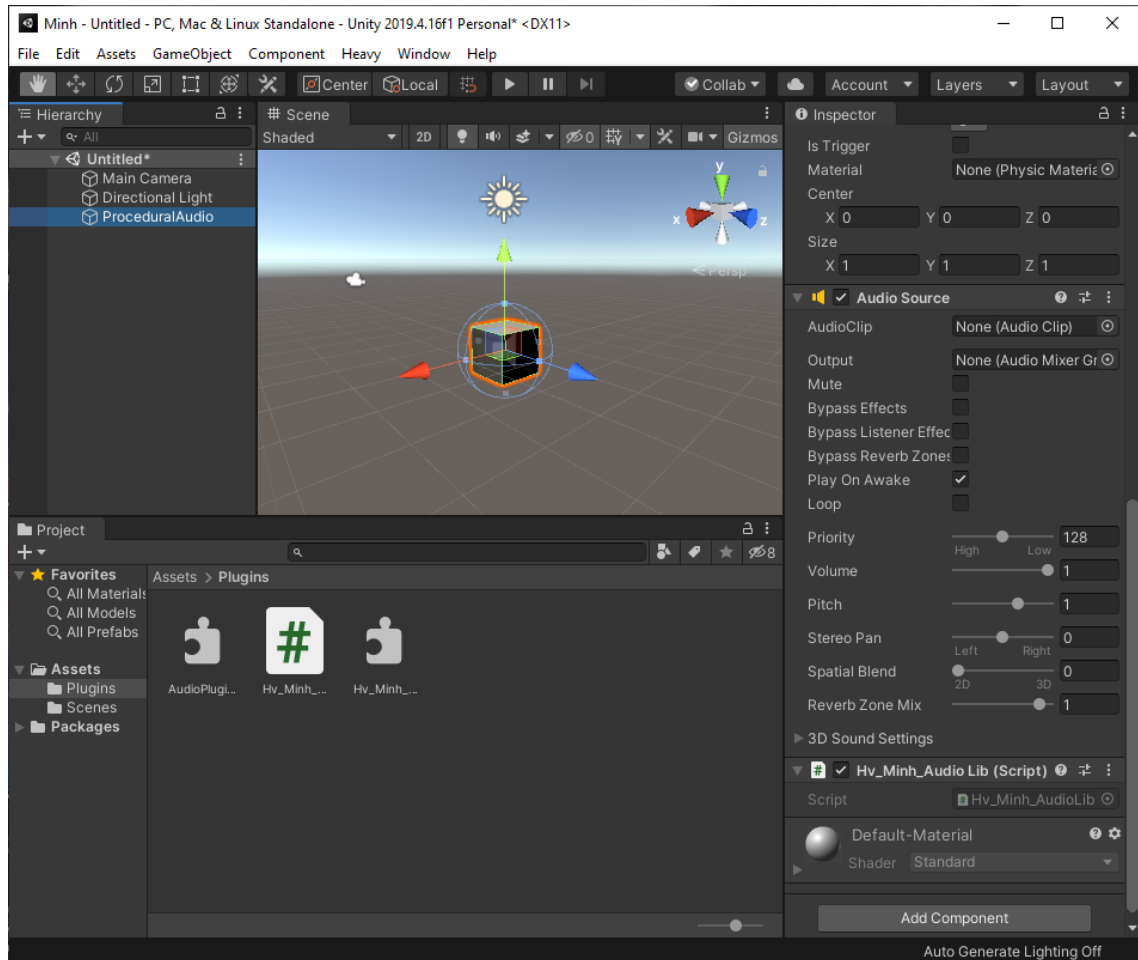


PICTURE 38. Built script and plugins inside the unity folder

These 3 objects are everything needed to run procedural audio in Unity. Inside your Unity project, it is a good practice to create a folder within the Assets directory called Plugins. Inside that folder, put in the 3 objects that were created with the build. It is enough to drag and drop the script to any game object in the



scene and it will automatically create the audio source with volume, pitch, panning options etc. (Picture 39). Now when pressing play, the patch should start playing.



PICTURE 39. Unity project including the 3 required objects

## 5 DISCUSSION

Sound design in games and other non-linear media has gone through major steps over time. As the expectations and attention to detail concerning sound are growing all the time, sound designers are constantly looking for new and creative ways to provide information and immerse the player even better. Real-time creation of sounds, procedural audio, can be used to tackle these demands as it can save time and resources of sound designer. It has been around from the very beginning of game development, in fact it used to be the only way to create sounds. It was a necessary step to include sound in games as there simply was not enough RAM to use sound assets, instead they had to be programmed. Ever since hardware got better, use of sound assets took over and only recently, procedural approach has made steps to become more user-friendly option for sound designers with new advances in popular middleware technology. (Crawford 2018).

In my practical project, procedural audio was used to create different kind of soundscapes to a virtual reality gallery. Using real-time synthesis to create the soundscapes and music allowed for a unique, ever-changing audio even though it created some challenges. For certain, ambient-like sounds, procedural audio worked really well but replicating real-world sounds such as instruments proved to be a challenge. It is safe to say creating a 10-instrument dance track took a lot more time than it would have in a regular way using recorded audio. Unity implementation also showed that while use of procedural audio is possible in game engines, it still requires quite a lot of work, at least the first time. As development platforms for procedural audio are becoming better and more available to sound designers, combining these older and newer techniques could be the key for more cohesive, immersive soundscape in a non-linear environment.

## REFERENCES

- Andersen, A. 2016. Why Procedural Game Sound Design is so useful – demonstrated in the Unreal Engine. Published on 18.1.2016. Read on 7.1.2021. <https://www.asoundeffect.com/tag/procedural-sound-design/>
- Chion, M. 1994. Audio-vision: sound on screen. New York: Columbia University Press.
- Collins, K. 2009. An Introduction to Procedural Music in Video Games. Contemporary Music Review 28, 8–9.
- Crawford, D. 2018. What is Procedural Audio? Published on 25.2.2018. Read on 11.1.2021. <https://daracrawford.com/new-blog-3/what-is-procedural-audio>
- Farnell, A. 2007. An introduction to procedural audio and its application in computer games. Published on 23.9.2007. Read on 28.1.2021. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.531.2707&rep=rep1&type=pdf>
- Farnell, A. 2010. Designing Sound; Procedural Audio for Games and Film. MIT Press.
- Flossmanuals. N.d. Generating Waveforms. Read on 28.4.2021. <http://write.flossmanuals.net/pure-data/generating-waveforms/>
- Flossmanuals. N.d. Graphical Programming. Read on 27.4.2021. <http://write.flossmanuals.net/pure-data/graphical-programming/>
- Flossmanuals. N.d. Pure Data Introduction. Read on 27.4.2021. <http://write.flossmanuals.net/pure-data/introduction2/>
- Fournel, N. 2010. Procedural Audio for Video Games: Are we there yet? Read on 18.2.2021. <https://www.gdcvault.com/play/1012645/Procedural-Audio-for-Video-Games>
- Fournel, N. Sound Designer. 2012. Interview on 4.6.2012. Interviewer Nair, V.
- Github. 2018. The heavy hvcc compiler for Pure Data patches. Updated 21.9.2018. Read on 27.4.2021. <https://github.com/enzienaudio/hvcc>
- Github. 2018. Building Unity plugins. Updated 17.8.2018. Read on 27.4.2021. <https://github.com/enzienaudio/hvcc/blob/master/docs/05.unity.md>
- Hillerson, T. 2014. Programming Sound with Pure Data: Make Your Apps Come Alive with Dynamic Audio. Raleigh: Pragmatic Programmers.

Klang, M. 2018. Compile Pure data patches with free online Heavy compiler. Published on 12.9.2018. Read on 27.4.2021.

<https://www.rebeltech.org/2018/09/12/compile-pure-data-patches-with-free-online-heavy-compiler/>

Knox, D. Senior lecturer. 2019. Games Sound Design: Generative systems, Procedural Audio. Lecture. Glasgow Caledonian University on 22.10.2019. Glasgow.

Loyola University. Avid Pro Tools Bootcamp. Read on 28.4.2021.

<https://pacs.loyno.edu/avid-protools-bootcamp>

Moog. N.d. Product catalogue. Read on 28.4.2021.

<https://www.moogmusic.com/products/matriarch>

Nil, B. 2019. Procedural Audio on the Web: Part One. Published on 17.8.2019. Read on 15.2.2021.

<https://medium.com/@berraknil/procedural-audio-on-the-web-part-one-166462e7be1e>

Pngarea. N.d. Piano player silhouette. Read on 28.4.2021.

[https://www.pngarea.com/view/e848c033\\_piano-keyboard-png-piano-player-silhouette-png-png/](https://www.pngarea.com/view/e848c033_piano-keyboard-png-piano-player-silhouette-png-png/)

## APPENDICES

Appendix 1. VR-Gallery Procedural Audio Compilation

<https://youtu.be/mdDgggW9gRk>

## Appendix 2. Bubble Sound Comparison

<https://youtu.be/72QChAqSzcY>