QuBits, an Interactive Virtual Reality Project and Compositional Space for Sound and Image

By

Jonathan Kulpa

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Music

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Edmund Campion, Chair
Professor Franck Bedrossian
Professor Myra Melford

Summer 2019

QuBits, an Interactive Virtual Reality Project and Compositional Space for Sound and Image

by Jonathan Kulpa

Abstract

QuBits, an Interactive Virtual Reality Project and Compositional Space
for Sound and Image

by

Jonathan Kulpa

Doctor of Philosophy in Music Composition

University of California, Berkeley

Professor Edmund Campion, Chair

This paper describes the *QuBits* project, a virtual reality (VR) environment created by the composer, offering an expanded medium for musical experience with integrated space and visuals. The *QuBits* VR environment is the essential supporting material for the dissertation and this paper provides a full description of the project.

The environment was designed to explore a musical aesthetic valuing sound mass, spatial sound, evolution, and algorithmically generated sonic structures. Additionally, the user of the VR system plays a key role in shaping these musical elements. The user first must discover what behaviors are possible by exploring and through chance encounters. They can then shape each discovered behavior with nuance if they choose.

In the VR environment, each sound has a corresponding visual component. To achieve this, a system was built with two software platforms, one for digital sound processing and another for 3D graphics and algorithmic event generation. These platforms communicate via Open Sound Control (OSC). The sounds are a mix of real world sampled sound, granular synthesis, and real-time generated synthetic sound.

The *QuBits* VR environment represents the results of this methodology. Pros and cons of the methodology are discussed, as well as implications for future projects.

CONTENTS

CHAPTER 1:
INTRODUCTION TO THE QUBITS PROJECT

A Virtual Reality Sound Space

QuBits is a virtual reality (VR) project that surrounds an individual, referred to as the user, in a simulated world of visuals and sounds, a virtual "sound space". The user navigates this world with headphones and a system like the Vive VR System (Vive, 2019). Such a system includes a wearable headset (with head tracking sensors and a separate image display for each eye), hand controllers, and sensors to map the user's physical space to virtual space (see Figure 1 for example of virtual space). In the virtual space, users can look around freely, rotate while maintaining a vantage towards the floor's center, and move forward and backward. The user encounters many types of "VR characters" (also see Figure 1), or beings with a presence in the virtual space. Each VR character has a type of sound, appearance, movement physics, and ability to affect other characters. The different VR character types will be covered in detail in Chapter 4 (see "VR Characters"). The user is not merely an observer. By their hand controller inputs and movements through the space, they influence the VR characters' sonic and visual behavior and evolution to new behaviors.

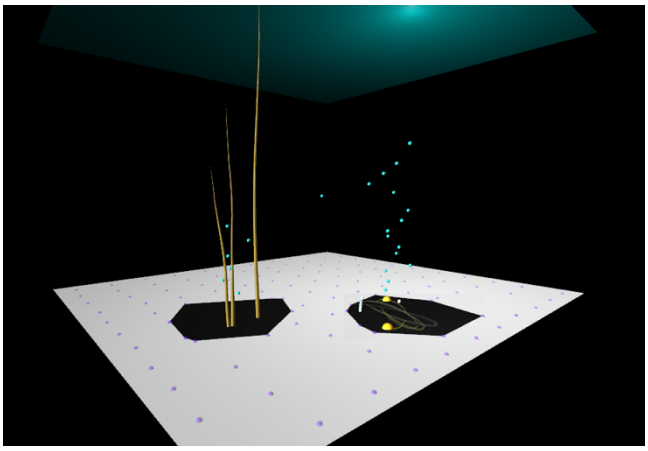


Figure 1: various VR characters in virtual space

VR Characters to Explore the Morphology of Energy

This project explores energy as comprised of four elements, energy as 1) algorithmically-shaped 2) sound and 3) visuals, 4) moving through time and space. The VR characters are the very medium for this exploration.

Every event that occurs in *QuBits*, i.e. all energy in the system, is the result of VR character activity. When each character makes a 2) sound, it has a 3) *corresponding, simultaneous* visual behavior, aimed at being two elements of the same energetic event. As simple examples, a character's speed of movement corresponds to its amplitude (volume) of sound, or if a character is orbiting, speed of orbit corresponds to speed of rhythmic pulse. As a composer, I privilege sound, but for this project, I understand it as being one element of evolving energy. This audiovisual energy is 1) shaped by the computer as it executes coded rules, sometimes allowing user input. Also, VR characters exist at 4) specific locations in virtual space (and time), thus locating the energy. As the VR character acts upon the system, then, it entwines together these four elements. It is not intended the user perceives this makeup for every event. Rather, for a user to have some understanding while perceiving some mystery is itself my compositional goal.

Real-Time Generation of Sound

Many of the audiovisual events in this project are generated in real-time. This term reflects that events are not pre-composed, instead generated by the computer at the time the user witnesses them. In *QuBits*, the shaping of audiovisual behaviors in real-time is often a joint venture between computer and user.

Composing and Coding as One Process

For *Qubits*, it was necessary to compose with rules (algorithms) as well as ranges of values (numbers) that populate the algorithms. I not only conceived of these rules, I implemented them in computer code[1]. In my view, composing and coding are two parts of the same creative process. Consider that the process of orchestration can affect musical ideas, and vice versa, because discoveries and limitations are encountered in each that inform the other. By having to implement a musical idea in a concrete form, ideas emerge. For the same reason, the process of coding affects what algorithms for music making I imagine, and vice versa. I embrace this creative feedback loop.

First Iteration and the Need to Rebuild

Initially, both the visual and audio processing were implemented in Max/MSP, a visual programming language for generating music and other media (Version 7.3.5; Cycling '74, 2016). The visuals were created with Jitter and GenJitter, Max/MSP's 3D vector graphics engine. The algorithms driving the system were encoded in multiple expression languages running inside Max/MSP. These include Max and Jitter data processing objects, the GenExpr expression language, and the odot expression language (Version 1.2-20_beta; Regents of the University of California, 2017). odot operates on Open Sound Control (OSC) (Wright, 2002) data bundles, and translates these bundles to Max/MSP's native data types. After much development, this version resulted in bottlenecks in the processing chain, with a significant degree of visual latency.

As a first step to troubleshoot systematically, an attempt was made to consolidate multiple GenExpr code boxes into one, however, this proved to be extremely challenging. GenExpr must further compile to C code. Many compile errors resulted while consolidating, with no report as to the responsible line numbers in the code. Rob Ramirez, who previously worked on Jitter development for Cycling '74, graciously helped me fix a few of these errors, his expertise allowing him to interpret the limited error feedback. Though a start, I needed tools to find these errors myself wherever they occurred. Even if the task was successful to consolidate code, this was only the first step to troubleshoot a complex web, where visuals drove sound, and vice versa, and involving many expression languages. I felt I had hit a wall as a composer in deep computational territory, exceeding my ability to move forward in this iteration.

Hardware and Software in the Rebuild

Around this time, I met with Björn Hartmann, Professor of Electrical Engineering and Computer Science at the University of California, Berkeley, and his then PhD student, Bala

[1] In early iterations, coding was done with the tutelage of composers (and skilled coders) Ilya Rostovtsev and Rama Gottfried. This is how I learned to code and troubleshoot. Also, computer game scientist Andrew Ajemian provided consultation for questions I had while rebuilding the project in Unity/C#.

Kumaravel, doing research in VR and 3D graphics. They advised me to rebuild the visual and control data system in Unity (Version 2.18f1; Unity Technologies ApS., 2018), which also works in conjunction with C# code. In this rebuild, I have benefited from Unity's and Visual Studio's[2] error reporting and troubleshooting tools, able to solve issues myself when they arise. Equally if not more important, the visuals in this platform run much faster, even when there is a high degree of activity. Max/MSP is retained as the sound engine, affording many more possibilities than what Unity provides specific to digital sound processing.

Unity and Max/MSP communicate by sending OSC data bundles using the User Datagram Protocol (UDP) (Postel, 1980)[3]. OSC bundles cannot be understood by either Max/MSP or C#, so additional software runs inside each platform, translating to native data types. On the Max/MSP end, odot handles the task. On the Unity end, the software package OSC simpl (Version 1.3; Sixth Sensor, 2018) translates to C# data types. A powerful artistic canvass is then made, where visuals and algorithms in Unity can trigger sounds and algorithms in Max/MSP, and vice versa. This is how sounds and visuals are produced simultaneously, central to the four-element exploration of energy discussed above. Figure 2 summarizes the potential flow of data, in either direction between platforms. In my own work, to centralize logic and by preference of scripting language, I utilize C# to drive the system from the Unity side whenever possible. Rarely, data from Max/MSP drives Unity. An additional software component needed is SteamVR (Version 1.4.14; Valve Corporation, 2019) to interpret and map VR hardware input data in C# scripts.

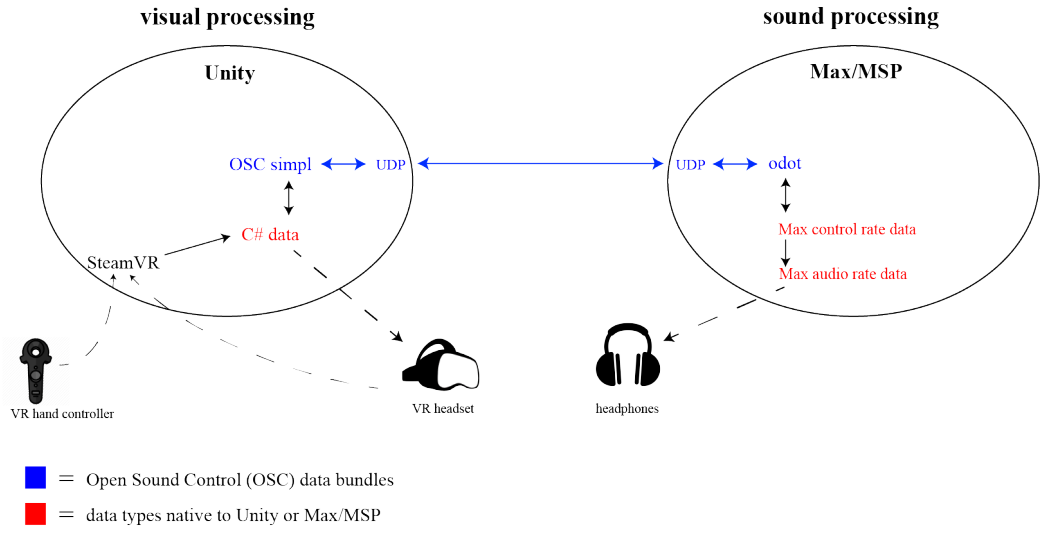


Figure 2: summary of software and hardware communication. Arrows indicate possible direction of data flow.
** Hardware clipart free for personal use as per copyright notices.*

[2] Visual Studio is a C# scripting and editing software (Version 8.0.9; Microsoft Corp., Xamarin Inc., and MonoDevelop contributors, 2019).

[3] On the Unity side, the UDP networking is handled by the OSC simpl software. In Max/MSP, this networking is handled by native objects, udpreceive and udpsend.

CHAPTER 2:
COMPOSITION PRINCIPLES AND AESTHETICS

An Environment of Wonder and Discovery

I have personally felt a sense of wonder in perceiving some inner workings of the laws of physics, while being aware there are enumerable mysteries. I have also felt wonder in being able to affect my surroundings in a small way, while perceiving I am but one small agent embedded in a complex system. These dualisms, knowledge and mystery, agent but embedded, compel me. I similarly aim to place a user in a situation that evokes their awareness of these dualisms. Aimed at knowledge/mystery, behaviors in *QuBits* are not static, evolving and developing relationships over time, hopefully creating a rich environment that requires the user to continually renew their understanding. Aimed at agency/embeddedness, some behaviors allow for user input while others are shaped entirely by the wider system, the underlying algorithms. As agent, the user first has to discover what behaviors they can engage with, through chance encounters and explorative input. They can then shape these behaviors with nuance if they choose.

Algorithmic and Generated Sound

I have always been interested in thinking about music as algorithmic. I am inspired by nature's designs, like waves and clouds. These physical structures are composed of complex patterns deriving from underlying laws of physics. Similarly, sounds can be generated by computer algorithms. The underlying rules generate sonic structures that are constantly varied yet recognizable as a class.

- **Randomness and Weighted Probability**

To create self-similar but always varying sound, my algorithms often employ constrained randomness (value selection bound within a range) and weighted probability. Constrained randomness allows sound to be constantly varied while guaranteeing each variation conforms to a desired range of behavior. Within a desired range, weighted probability further makes certain values more likely than others. These functions are used for both synthesis and samples. In synthesis, the functions select values fed to parameters that generate ever-varied sound in real-time. For sample libraries, the functions select successive samples in infinite combination and with varied transposition.

Random and probabilistic functions also affect large sonic structures, entire layers of activity. There are many *types* of VR characters, each individual of that type behaving similarly, giving that type a collective identity as a layer of sound. Each layer is governed by these functions, determining parameters such as volume, density, timbre, and harmonic color.

- **A Conceptual Model of Algorithmic Music**

In composing this system, I have come to think about three main elements that comprise algorithmic music. This section is a conceptual one, and the terminology introduced here is necessary to discuss music and interactivity in *QuBits*.

The three elements in this conceptual model are 1) the rules, lying in wait to be activated or not, 2) the values (numbers) that seed the rules, and 3) the timings of these values, causing the rules to output and generate a sonic event. This paper puts an emphasis on the discussion of

rules, values, and timings precisely because these elements shape music in this project. To illustrate, take a conditional statement pertaining to timbre/color of sound:

If an applied force > 10, then make the timbre brighter, else make the timbre darker.

The written statement is the *rule*. Sending a *value*, a strength of applied force, at a particular moment in *time* produces a sonic event, making the sound brighter or darker. By sending successive values, sound is shaped over time. Say there is a list of individual force events, 11, 6, 5, 12, and 13, and each successive value is sent at a particular moment in time, a rhythm of values. This then shapes the sound as brighter, darker, darker, brighter, brighter, and with the same rhythm as the input values. This project engages with this kind of sound-making, though with an entire web of algorithms working together to create a rich system.

An additional component in the conceptual model is the notion of the creator. The creator is who or what determines each of the three elements in the model. Any combination of creators (human, computer, or nature[4]) can be responsible for any of the three elements of the conceptual model (rules, values, or timing). As a composer, I could be creator of every element, perhaps making a fixed piece of electronic music that repeats the same way every time. This, however, is not my goal.

- ***QuBits* in the Conceptual Model of Algorithmic Music**

In *QuBits*, I am creator of the rules[5]. I am also the creator of ranges of values, constraints, that limit the range of possibility for the computer and user. However, in real-time, the computer and user determine the actual values and timings, selected within the defined constraints, to shape the unfolding sound.

As composer of rules (and value ranges), I begin by trying to imagine the behavior of sound I am interested in, and then in service of this, I build up webs of conditional statements in code. I then test the result by becoming a user of the system. Again, this is an emergent musical process. By using, listening, being critical, and imagining some more, the value ranges and algorithms are adjusted until compelling musical behaviors take shape, sometimes reflecting the original idea and sometimes evolving in a new direction. As will be discussed in Chapter 5, to compose and tweak value ranges, I made a script containing every important numerical range, project-wide. This is similar to a written aleatoric musical score, expressing a range of possibility, here bounding possibility for computer and user at project runtime.

In this project, there are three possibilities, or modes, of interaction between computer and user for the creation of values and timing. 1) The computation entirely determines the timing/triggering of an event and which values shape the event over time. 2) The user triggers the beginnings of events, but thereafter values are selected by the computation. 3) The user triggers the beginnings of events, and thereafter user and computer jointly shape the values over time, *a duet of influence*. In this mode, when the user inputs to the system, they influence the

[4] e.g. the fluid dynamics of wind, water, and fire, electrical current, magnetism, chemical reactions, weather conditions, tidal forces, many other cosmological phenomena. These can all be measured with a multitude of appropriate instruments and used to drive one of the conceptual elements.

[5] Because I use sound sampled from the real world, nature is also creator of rules, the laws of physics making these sounds possible.

morphology of a sound, and when they no longer input, the computation completes the sound. By continually inputting and letting go more rapidly, the user can "surf" back and forth between their contribution and that of the computer. In Chapter 4, for each VR character, I will discuss the mode of interaction. The user and computer do not make the rules that govern the system, again this being my task as composer. However, the user affects which of many predefined rule sets is currently active, i.e. they affect the "global evolution" introduced next.

Activating New Rules

- **Independent Rules**

Embedded in the environment are rules that remain dormant until the user discovers how to activate them, using their hand controllers and by moving through virtual space. The new rules introduce new VR characters or change the behavior of those already present, providing change in the overall activity of the sound space. Many of these discoverable rules are independent, not part of a longer-term plan or trajectory.

- **A Composed Trajectory of 5 Rule Sets**

I also composed, and coded, a trajectory of five rule sets, or states, that tell a story of change in a specific order of evolution, most likely unfolding over a longer duration. Throughout this paper, I refer to this trajectory as the global evolution. The order of states is linear, i.e. a user cannot skip over a state; however, a user can experience an evolution forward along this trajectory or devolution back towards the initial state. Details of the global evolution will be discussed in Chapter 4 (see "The Global Evolution"). When a user encounters any change in the system, it is not intended that they are aware whether it is an independent change or part of this longer-term plan. Rather, both are part of my compositional process in designing an environment ripe with new behaviors that may be discovered and shaped.

Several principles guided the design of the global evolution. First, the user causes each change in state (again, see Chapter 4). Second, it is desired that the user witness some, or all, of this trajectory in a single visit of tens of minutes to an hour. Third, the sonic arc of the global evolution progresses from maximum noise/least pitch to maximum pitch/least noise. Finally, as the trajectory progresses, VR characters evolve to become more connected to each other.

Sound Masses and Particulate Sound

Another principle centrally important to *QuBits* is the exploration of particulate sound in which many grains of activity make up the larger sound mass. Usually, each particle is not audible as an individual, rather serving to contribute to the identity of the mass. Nature again provides inspiration, as in the sound of each particle in the ocean contributing to larger sonic structures, waves rushing in, crashing, and receding.

In this project, masses of VR character types allow for the exploration of masses of sound. Not every layer of sonic activity is a mass, but many are. There are 121 individual "qubit" characters, each with its own dedicated audio engine, contributing one particle of sampled real world sound to the greater mass of all qubits. There are also 36 "ceiling light" characters that similarly each run their own audio engine, contributing to a synthesized mass. Another mass becomes ever-denser as the user creates more "void" characters. All of these layers are described in full in Chapter 4. Also, in Chapter 5, a centralized script design is

discussed, constraining possibility in each mass and evolving it locally to different sonic qualities.

Spatial Sound

In *QuBits*, there are a couple ways in which space, here virtual space, is important for the production of sound. The first has already been stated in Chapter 1: VR characters are located in virtual space, producing sonic behaviors at those locations. Energy is broadly explored as algorithmically-shaped sound and visuals, moving through time and space. Second, the user depends on virtual space for the production of interactive sound. A user's two hand controllers allow them to place certain input gestures at points in virtual space, then affecting the sound of VR characters positioned near those locations. Also, audio and visual effect engines are activated and shaped by the user's own movement through virtual space and by travelling to certain locations.

CHAPTER 3:
SOUND MATERIALS

The sounds in this project are a mixture of sampled real-world sounds, phrases of granular synthesis (based on the samples), and synthesis (generated sound without samples).

Sampling of Real-World Sounds

In *Qubits*, the source sounds are integral to the compositional model. I had the desire for each sonic particle to be a quiet, delicate sound from the real world, recorded in great detail. These sounds are essential to the desired aesthetic of the work. The detailing of these sounds was made possible by recording with multiple microphones placed close to the sound source in a room with a low noise floor, a semi-anechoic chamber. Each microphone track was additionally given a gentle pass of de-noising with the software iZotope RX (Version 6; iZotope, Inc., 2017). All tracks were then mixed together per recorded event.

- **Short Samples**

These short samples are my sonic fundamental building blocks, each a particle, or as described later, each a grain. A large collection of objects with various material properties was obtained for making an extensive sample library. The objects include an assortment of metal (e.g., washers, air conditioning ventilation parts, large tubes, elbow adaptors, and plates), pieces of laminated wood, the surface of an audio equipment case (plastic laminate on plywood), bricks, and a large collection of pins. The pins are used to excite the resonances of the other objects, but in so doing, also resonate themselves. With synthesized sound, variation is created by seeding algorithms with ever-varied values. To achieve variety with recorded sound, a large library of self-similar but varied samples was made for each object.

For each metal and wood object, each sample consists of striking the object once with a pin, at a unique node and with a variant amount of force, then letting the object resonate (I will refer to these samples as “pin strikes”). Between 15 to 80 samples were made per object. A hyper-cardioid, an omni, and a large diaphragm microphone captured each event.

For the bricks, contact microphones were applied to capture vibrations traveling inside the material. Here, variation between pins became more important. In each sample, one of the pins, each with a unique size and shape, was dropped from above, first bouncing onto the bricks (rhythms that accelerate) followed by the sound of the pin rolling around (I will refer to these samples as “pin ricochets”). 100 samples were made to represent the collection of pins.

Contact microphones were also used on the audio equipment case. A pin was laid on the material and lightly thwacked, causing it to roll around while recording the vibrations (I will refer to these samples as “roly-poly samples”). 80 samples were made.

- **Longer Samples**

In other samples involving the bricks, a pin was dragged around in figure-8 and other swirling gestures while recording with contact microphones (I will refer to these samples as “pin swirls”). A few samples were made, each a long duration of this activity.

Other materials were obtained to make samples not involving pins as excitation, including abacá paper and baskets made of wire or various fibers (dried grasses, wicker, and bamboo). The same air microphones used for the pin strikes were again used here. In each

sample, one of the materials is placed under tension by squeezing by hand, then relaxing the tension, and repeating this over and over. These are longer samples of crackling and rustling sounds.

Phrases of Granular Synthesis

I wanted to explore the generation of longer musical structures, phrases, by chaining together the short samples described above. Granular synthesis, driven by code, is the main technique employed. The particular granular synthesis engine used is o.granubuf~ (Gottfried, 2016), which runs in Max/MSP. Granular synthesis typically involves splitting a longer sample into its constituent shorter durations, its grains, each perhaps 1 to 2000 ms. The collection of these grains assembled in order recreates the original sound. However, the allure of the technique is rearranging grains in any order, overlapping them, and playing them with any speed, rhythm, and transposition.

- **Principles in Granulation**

Each sample as grain. Had I recorded one long sample, striking an object over and over with a pin, and then split this into evenly-spaced grains, there would be an element of chance as to the morphology of each grain. Some would contain more of the pin attack, some would contain very little, and some would be in between. I wanted each grain to have a predictable attack and decay. This can be accomplished instead by recording individual sounds that have a desired trajectory. My short samples were made with this design in mind. Each entire sample is a short grain with a guaranteed attack and decay.

Recorded granulation. Unfortunately, some granular phrases and effects require a lot of processing power due to a high number of overlapping grains, especially for the time stretching and freezing techniques described shortly. Furthermore, my intentions are to explore an entire mass composed of these granular phrases, in this project as dense as 121 simultaneous voices. The CPU power available to me does not permit 121 instances of this granular engine running at the same time[6]. Thus, the strategy was to record each granulated phrase and to build up a library of phrases. Though not generated in real-time, these phrases are generated algorithmically, with Max/MSP's random functions and odot code guiding the behavior of o.granubuf~.

In sum, I recorded source libraries, activating real-world objects with a pin over and over, and phrase libraries, running a phrase-generating engine over and over. Next, I describe each phrase engine, recorded and collected into a phrase library.

- **Phrase Types**

Feather-beam rhythms. Here, the code engine and o.granubuf~ generate accelerating and decelerating pulses of grains, each grain being a pin strike sample. The code alternates between selecting a fast or slow grain rate target and then interpolates to that target, in a randomly selected duration of time. The default is set to interpolate over a medium-length duration, so that the acceleration or deceleration is relatively gradual. However, the code occasionally generates up to three shorter interpolation events, resulting in what sounds like

[6] o.granubuf~ is written as a gen~ code box. For granular time stretching and freezing, I need at least 75 overlapping grains. This is handled in the gen~ by a for-loop of 75 iterations, calculated 44100 times per second, thus 3,307,500 calculations per second. If there were then to be 121 of these engines running simultaneously, it would be 400,207,500 calculations per second.

quicker waves of pin strikes, providing more surprise and interest. I will refer to these samples as "feather-beam samples", after the written musical notation that denotes accelerating and decelerating rhythms.

Sound masses. Sound masses with various densities were created using algorithms and granulation, recorded to long samples. For this sound, the singular grain was a pin ricochet sample. Given a rapid succession of these grains, the sound is much like dropping an entire collection (or a mass) of pins onto bricks, and never running out of pins. As only contact microphones were used for each pin ricochet, this mass is percussive, without distinct pitch. A code engine controls the duration between grains, affecting how quickly new sounds enter the overall mass, producing a desired density. For different densities (sparse, denser, and densest), the engine was run for a minute and a half while recording. I will refer to these samples as "pin ricochet masses". It will later be described how each of 121 qubits plays a portion of these samples, making a larger sound mass out of these smaller masses!

Time stretching. The pin strike samples involving metal tubes were subjected to granular time stretching, recorded to phrases. With this effect, any sound can be stretched out or condensed to any duration. This is not the same as playing the sample slower or faster, which would change the original pitch. Granulation can stretch or squish a sound's duration without changing its pitch.

It is worth discussing how this effect is accomplished. Grains are launched at a very fast rate (one every ~4 ms), each being very short in duration (~50 ms). At this speed of production, grains are not perceptible as a rhythmic phenomenon, instead contributing to a unified sound. Fast overlapping grains are used to reconstruct the original sound but then stretched to a new duration, as follows. To indicate where in each sample to play a grain, a floating-point value is sent to o.granubuf~ , with 0 being the beginning of the sample, .5 being the middle, and 1 being the end. A phasor waveform, conveniently, is a floating-point interpolation from 0 to 1 over a duration of time, then used to request grains in order from beginning to end. If the phasor's ramp from 0 to 1 takes the same duration as the original sample, the original sound is reconstructed from fast overlapping grains. To time stretch the sound, the phasor's ramp instead takes a longer duration. If this was the entire design, many of the same grains would be repeated consecutively, producing a comb filter artifact (an extra buzzy pitch not present in the original sound). To remedy this, grains are randomly chosen within a range around the phasor's indicated play position.

Each metal tube pin strike sample was time stretched *in reverse*[7], designed to progress from darker to brighter in timbre. The reason for this is discussed in the section on VR characters (see "Qubits Revisited – Type 6"). To begin with a darker timbre, the initial play position was around 20% into each sample, well after the moment the pin strikes the tube, with most higher frequencies decayed and mid to low frequencies still sounding. From this position, a recording was made, stretching in reverse to the beginning of the sample where higher frequencies are present, a brighter timbre.

Time freezing. The pin strike samples involving wood pieces and the metal tubes were subjected to granular freezing, recorded to 15 second samples. This is a similar technique to time stretching, only grains are randomly selected around a fixed position. The effect is that time

[7] The phasor runs in reverse, from 1 to 0 and at a time-stretched duration.

has slowed to a stasis and one can listen to a single temporal slice for as long as desired. Freezing a wood pin strike sample sounds similar to filtered generated noise. Freezing the beginning of these samples yields brighter noise (higher frequencies) and freezing the end yields darker noise (fewer high frequencies). Freezing a metal tube pin strike sample results in powerful low-pitched resonances.

These types of recorded phrases are my main uses of granular synthesis, though additional uses will be mentioned briefly in Chapter 4.

Synthesis (Without Samples)

Granular synthesis is somewhat of a misnomer. Though these phrases are generated with a computer, the sonic atoms are samples, not synthetic. The term synthesis, proper, refers to sounds entirely generated, including the sonic atoms. For instance, synthesized sound can be made by summing sine tones, each sine being generated by the computer. Here I discuss my use of synthesis, with generated atoms.

- **Overview: Synthesis of Spinning Washers**

The main use of synthesis in this project is an engine that recreates the type of sound I encountered while using a pin to strike a suspended washer[8]. I initially suspended a washer by looping a string through its hole, tying the string off. Then I struck the washer with a pin, causing it to rotate and produce spinning pulsations. I had the desire to better control these sounds, timbrally and rhythmically. This led to my first foray into synthesis, attempting to generate these sounds from a model fed by controlled parameters. It is important to note that a real washer could be made to resonate without the spinning pulsations. Likewise, in synthesizing these sounds, the washer model and the spinning pulsations are produced by separate algorithms.

- **The Synthesis Engine**

The synthesis engine used is resonators~ (Regents of the University of California, 1999), an engine running in Max/MSP. Instead of summing sine tones, resonators~ sums resonant filters, where each filter resonates at a specific frequency and amplitude (tone and volume). Additionally, each filter has an independent decay rate, how quickly that tone's volume diminishes over time. This is powerful for modeling sounds, lending an independent trajectory of volume for each constituent tone. After configuring the model with a list of frequencies, amplitudes, and decay rates, an input signal such as an audio click excites the filters, similar to how a pin head excites the resonant frequencies of a washer.

- **Washer Model**

To control the timbre/color of a washer model, I explore degrees of harmonicity/inharmonicity. Given a collection of tones, the lowest tone is called the fundamental, or the lowest partial. Tones higher than the fundamental are called higher partials. The harmonic series is produced when higher partials are whole number multiples of the fundamental's frequency, e.g. for fundamental 440 Hz, the higher partials are 880, 1320, 1760, etc. This series of tones sounding simultaneously results in a consonant timbral quality, without sonic roughness. When higher partials are floating-point multiples of the fundamental, the series

[8] A thin, disk-shaped plate with a hole in the middle, usually used to distribute the weight of a bolt or nut.

of frequencies is inharmonic, less consonant and having various qualities of roughness. Cella (2013) proposed the generalized series, an algorithm to derive a collection of frequencies whose timbral quality is derived from three parameters, one of these being harmonicity/inharmonicity. The generalized series is written as:

$$G = \sum_{n=1}^{\text{num partials}} f\left(n^{\text{harmonicity}} * \text{deformation}^{n} + \text{size}\right)$$

With values harmonicity 1, deformation 1, and size 0, the harmonic series is produced. Values that deviate from these produce an inharmonic series. *n* is the partial number and *f* is the fundamental frequency. One proceeds by building a list of frequencies in an iterative manner: apply the formula for $n = 1$ and add the resulting value to a list; do the same for $n = 2$, etc, until reaching the maximum *n*, the desired number of partials. In my implementation, I hold deformation constant as 1 and size constant as 0, reducing the formula simply to:

$$G = \sum_{n=1}^{\text{num partials}} f * n^{\text{harmonicity}}$$

Because one floating point value is then responsible for harmonicity, this parameter is placed on a continuum from most to least harmonic, or from least to most rough. Harmonicities close to 0 become murky with tones being very close together. The algorithm is implemented in C# code, running in real-time in Unity.

This is how to produce a desired harmonicity within one washer model, but for variety, I generate 50 models at a time, many which could be sounding simultaneously. This additionally requires a way to consider collective harmonicity of all models. Two scripts work together to achieve this. A "score script" defines a progression of conditions. Each condition defines 1) a desired harmonicity and frequency range for *partials within* an individual model and 2) a desired harmonicity and frequency range for *fundamentals across* all models. 1) and 2) may be independent from each other. There is no parameter to control upper partials across all models, though these are given a lower amplitude value, and are thus less salient. An additional condition 3) indicates the probability to include or omit a given partial, affecting a model's timbral complexity. Another script, the "model-making script", references the score's current constraints, first developing a pool of fundamentals as a series of tones in its own right, reflecting the desired harmonicity using parameter 2) above. The script then generates a batch of 50 models: for each model, the generalized series is run with *f* being randomly selected from the fundamentals pool and higher partials produced using parameter 1) above. Periodically, individual VR characters, "ceiling lights", randomly select one of the 50 models. Audio engines generate the sound of models in real-time.

- **Spinning Effect**

To make the washer model have a rhythmic pulse, the spinning sound, amplitude modulation is used. By introducing an additional frequency near the model's fundamental, beatings occur as the two waveforms interfere. At relatively low rates of beating, individual rhythmic pulsations are heard. I first calculate the critical band around the fundamental. At the

critical band distance, the beatings are very fast, not perceptible as individual pulsations, instead contributing to a very rough timbre. The critical band merely gives me a reference point, and is approximated by the equivalent rectangular bandwidth (ERB) (Moore & Glasberg, 1983):

$$\mathrm{ERB}(f) = 24.7 \cdot (4.37 \cdot f + 1)$$

Values between .5% to 8.5% of the ERB result in perceptible rhythmic pulsations and the speeds I desire. For a decelerating spin sound, the extra tone begins as a higher percentage of the ERB (faster pulsations), and then interpolates lower towards .5% of the ERB (slower pulsations).

Additionally, when the resonator model is excited with an audio click, the engine occasionally plays the actual recorded attack from a pin strike sample, ~10 ms of sound. Mixing in this sampled sound with the synthesis provides an embellished richness.

This is my main use of synthesis, though other uses will be mentioned very briefly in Chapter 4.

CHAPTER 4:
THE QUBITS PROJECT AT RUNTIME

VR Characters

Again, each VR character is a being with a presence in virtual space, having a type of sound, appearance, physics, and ability to affect other characters. Upon defining some basic terminology of Unity, C#, and Max/MSP, a VR character's full anatomy will be revealed. I will illustrate all of these terms by referring to the qubit type of character. A qubit begins as a sphere in virtual space, and there are 121 qubits (see Figure 3).

- **Terminology, and the Anatomy of Each VR Character**

Gameobjects. In Unity, gameobjects are all the fundamental objects placed in virtual space, like characters, props, and scenery. Each qubit character is a gameobject.

Prefabs and instances of prefabs. A prefab is an abstracted gameobject, or the blueprint to then make multiple individuals that behave similarly. Each individual that is instantiated, based on the prefab's blueprint, is called an instance. Each of 121 qubits is an instance of the qubit prefab.

Class, script. Class and script are used synonymously herein, a collection of variables, values, and algorithms. Each qubit instance runs the qubit class/script, acting as its brain.

Method. A method is a reusable chunk of code within a class. It performs a specific task and returns a value, executed simply by calling its name, e.g. ChangeTransposition(). In Unity, there are also native methods dealing with time and scheduling. Code placed inside the Start() method executes only once, when the project is first run. Code placed inside the Update() method repeats, executing once per visual frame.

Monobehavior class. A Monobehavior is the class capable of running the Start() and Update() methods. By executing once per frame, the Monobehavior's Update() method enables each qubit instance to live out an independent life, animating its rules and values over time, affected by the wider system and the input of the user.

Poly~ sound engine and instances. Each instance of a VR character extends its anatomy to Max/MSP, communicating with its own sound engine. Similar to prefabs in Unity, Max/MSP's poly~ allows for the abstraction of a sound engine. Any number of independent but similar instances of the engine can be created. Because there are 121 instances of the qubit prefab, there are likewise 121 instances of the qubit poly~ engine.

Full anatomy of a VR character. Each VR character, e.g. each qubit, is an instance of a prefab running an instance of a C# Monobehavior class and communicating via OSC with its own instance of poly~ audio engine in Max/MSP[9]. There are several types of VR characters, every individual per type behaving similarly by running an instance of the same Monobehavior and communicating with an instance of the same poly~. Next, I will describe the character types.

- **Qubits**

121 qubits are located on the floor, initially appearing as small purple spheres, collectively arranged in an evenly spaced grid formation. The computer and user can affect

[9] One can review Figure 2 in this light.

individual qubits in various ways, then causing those instances to change behavior, or type. There are six types.

Type 1. As the sound space begins, all qubits default to type 1. These are completely stationary qubits, unaffected by user or computer. The alpha value of its color is less than 1 so it appears translucent. Though not moving, its script is thinking, weighted probability determining whether it should become a type 2. The computer alone triggers each type 2 and shapes type 2 behavior thereafter.

Type 2. A type 2 embarks upon a random walk, bounded by a small area near its most recent stationary position, for a randomly selected duration of time. Its alpha value is 1 so that the color becomes more solid in appearance. The sound design of the entire mass of type 2 activity is one of my most important discoveries, a sound mass composed of constituent sound masses. Each type 2 selects a random play position in one of the pin ricochet mass samples, a density of pins dropping on bricks. This fairly loud sound is reduced to no more than a particle by playing at a very low volume. As a community of qubits perform random walks, the larger mass becomes a fluttery sound distinct in nature from the underlying pin ricochet particles. This could be applied more generally as a principle of composition: take a complex, highly energetic sound and crush it to a speck, via amplitude, and then use this as the atom to build a new quality of sound mass. Chapter 5 elaborates on scripting control over the mass density, here sometimes making a flurry of type 2 activity, and at other times very little activity. The change in density is further accentuated by having each qubit play from different pin ricochet mass samples, density conditions recorded as sparse, dense, or densest. Because each qubit's translucence varies between type 1 and 2, the visual representation of the mass corresponds to its sonic density (see Figure 3).

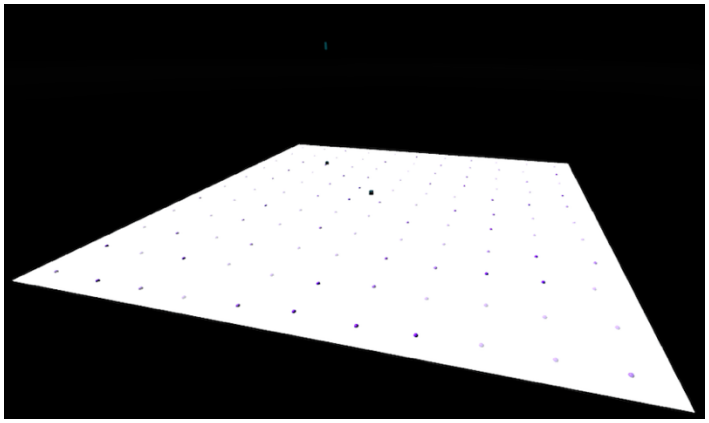


Figure 3: randomly walking qubits have a more solid appearance. Those that are stationary and faded out. This visualizes the density of the sound mass.

Type 3. The computer periodically selects qubits to become type 3. A type 3 splits into two spheres, each sphere expanding away from that qubit's most recent stationary position and then contracting back into a single sphere at this position. This corresponds to a crescendo and decrescendo of recorded synthesis, derived from an engine not yet discussed. This engine is based on a Fast Fourier Transform (FFT)[10], synthetically reproducing the sounds of the wood pin strike samples and then playing grains of that synthesis (Hynninen, 2016). These sounds are noise-based and rhythmic with a sporadic quality. A similar sound could have been achieved with samples and granular synthesis, but this synthetic version was full of dynamism, so I recorded a library of these sounds for this behavior type.

[10] An FFT analyzes all the component frequencies and amplitudes (corresponding energy of these frequencies), to reconstruct the original sound through synthesis.

Type 4. The user initiates type 4 behavior. With the use of their hand controllers, the user can click anywhere on the virtual floor which adds a force there (or click-drag to create a continuous trail of forces). Qubits close enough to the force become type 4, repelling from that location and becoming more solid and larger in appearance. The sound is similar to type 2, play positions randomly selected from a pin ricochet mass sample, however, the amplitude here is louder, more saliently revealing the composite pin ricochet particles.

The speed of a type 4 qubit determines the volume of sound, and this speed is a duet of influence between computer and user. The user determines the strength of force applied to a qubit through proximity of click, the strength then resulting in a speed of travel and corresponding increase in volume. When the user stops inputting, the computer applies a force of friction to slow the qubit, lowering its volume, eventually halting it and returning it to types 0 - 3. As long as the user continues to click and drag, nearby qubits move and sound more loudly. This is the first independent interactive rule the user activates, at first by chance as they attempt to impact the environment. They can thereafter shape this with intent.

I will discuss type 5 and type 6 after introducing the voids, characters that exist in a tightly knit relationship with the qubits.

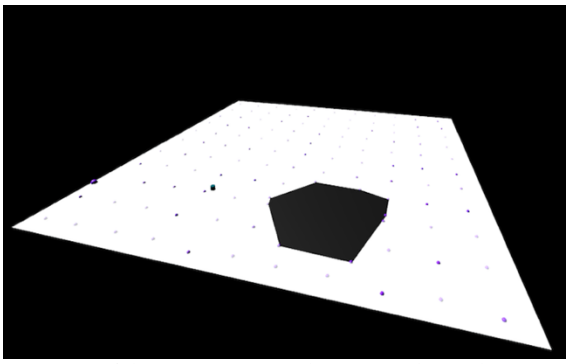


Figure 4: if the negative space between qubits is large enough, a "void" opens up in that space, bounded by the qubits that bound the negative space

- **Voids**

Formation and rules of shaping. The user creates voids. As the user displaces qubits (type 4), they most likely discover another independent rule: if the negative space between a local group of qubits is large enough, a void will open, bounded by those qubits (see Figure 4). The user may then use this rule deliberately, opening more voids wherever they see fit. Voids may occasionally open without intention, but these are still caused/created by the user as they push qubits around the space. At the instant of void formation, the bounding qubits are defined and exclusive; other qubits that come in contact with a void edge bounce away. The void's perimeter is a polygon whose vertices are marked by the positions of the bounding qubits.

There is also an algorithmic force counteracting void formation. If a displaced qubit does not bound a void after a sufficient duration, the computer moves the qubit along a trajectory back to its original grid-formation position. In this way, the qubit mass has a self-healing property.

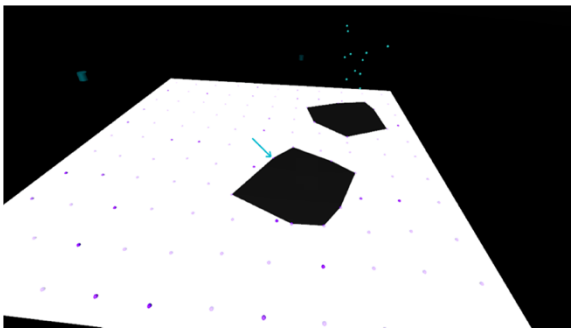


Figure 5: keeping void convex: the qubit by the drawn arrow is not allowed to move any further inward towards the void centroid.

The user and computer jointly create the evolving shape of the void. As the user displaces bounding qubits, the void's polygon is reshaped accordingly. A user can also get rid of a void, by making its area shrink below the threshold. The bounding qubits are also subject to two algorithmic forces to serve the needs of the void. 1) Each void polygon needs to remain convex, all its vertices pointing outward (this condition is necessary for other algorithms in the system). If the user attempts to push a bounding qubit into a position that forms a concave angle, the system instead moves the qubit back outward (see Figure 5). 2) Each void cannot grow too large in area. If the user pushes a bounding qubit too far outward, growing the void's area beyond the threshold, all bounding qubits contract inward until the area is acceptable again (see Figure 6). These forces, applied by user and computer, additionally place qubit and void in an *inter*-relationship: bounding qubits dictate the shape of the void, and the void restricts where bounding qubits can go. The user perhaps becomes aware of this relationship. Additional connections between qubit and void are revealed as the user continues to explore.

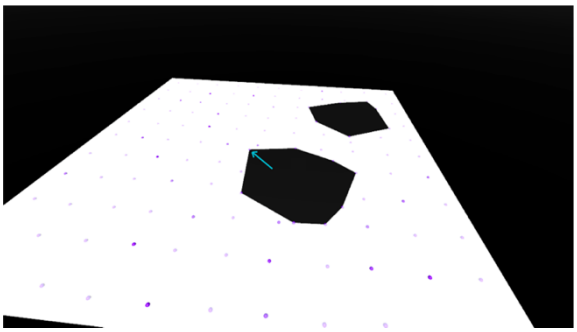


Figure 6: keeping void below area threshold: the qubit by the drawn arrow has moved too far outward; all bounding qubits contract towards the void

Sound. Voids make two types of sound, one being triggered by algorithm and the other triggered by user. All void polygons are rendered as a collection of triangles (see Chapter 5). By weighted probability, the computer periodically makes each void's component triangles flutter up and down ever so slightly (see Figure 7). This corresponds to a sound not yet discussed. While running the project on a particular instance, I granularly time stretched the entire mass of type 2 qubits (randomly walking and playing pin ricochet masses). The sound was recorded to a long sample for variety. This time-stretched sample is layered back into the project as the sound of fluttering voids.

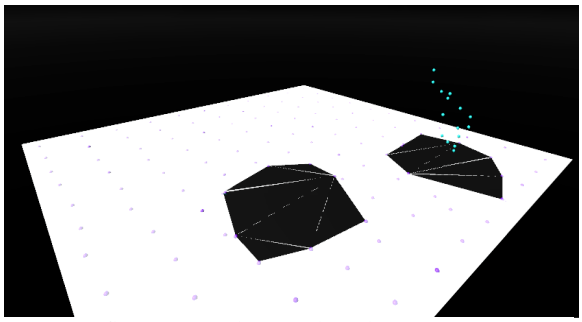


Figure 7: fluttering voids, revealing their composite triangles

For the other void sound, the user determines the beginning and the duration, but the actual sound is an algorithmically determined mix of samples. The user may discover, by chance, that by click-dragging on a void, it becomes displaced a short distance from its centroid; when the user lets go, terminating the click, the void springs back into place. For as long as the user displaces the void and while it springs back, crinkling sounds are produced, a randomly selected mix of the wire basket samples, transposed lower. Void displacement becomes especially important if and when the user discovers they can trap qubits inside voids, discussed shortly.

"Geyser" and "virtual particle" characters emit from a parent void. Though part of each void's system, these will be discussed as character types in their own right.

The voids are central to the design of *QuBits*. They interact with and affect almost every other character type, directly or indirectly, and the more that are open, the denser the overall

sound becomes. Having introduced the voids, I return to type 5 and type 6 qubits, further establishing a relationship between qubit and void.

- **Qubits Revisited**

Type 5. The user initiates type 5 behavior by causing a collision between qubits, the computer determines the end (allowing type 5's to last only a short duration), and in between, user and computer jointly shape the phrasing. If two qubits collide, they each become type 5, usually caused by the user as they push qubits around. This is another independent rule the user can discover and then use with intention. Type 5 qubits immediately become larger in diameter and turquoise in color. Instead of repelling from the user's click location, they attract to it. The sound engine plays a selected frozen sample, the bright noise made from freezing a play position at the beginning of a wood pin strike sample. Similar to type 4, speed corresponds to volume and the same duet of influence ensues, the user increasing volume via click input, and the computer diminishing the volume by applying friction.

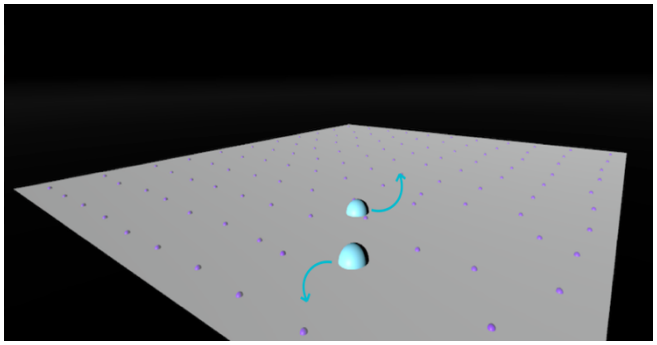


Figure 8: two entangled qubits (type 5). When the user moves one, the other simultaneously moves in mirrored opposite direction (drawn arrows demonstrate example motion vectors)

An additional feature is that both qubits involved in the collision become an "entangled pair": if the user moves one of them, the other also makes a sound and moves in mirrored opposite direction (see Figure 8). The computer only allows the pair to remain entangled for a short duration, each interpolating back to being small and purple, then carrying on as types 0 – 4.

It was previously stated that non-bounding qubits bounce off a void when trying to enter. A member of the entangled pair is the exception, able to pass into a void, though with consequence. At the moment of entry, it becomes a type 6, captured inside for as long as the void exists. This is another independent rule that may be uncovered. It also creates yet another relationship between qubit and void.

Type 6. There are a few behavioral possibilities of type 6 qubits. A type 6 first becomes a new shape, a tentacle with spatial extension (see Figure 9). By default, the tentacle's behavior is algorithmically driven. However, the user can interrupt this by displacing a captive void (again, by click-dragging the void), then beginning a duet of influence over tentacle behavior. A third possibility involves a tentacle morphing back into a sphere, then orbiting inside the void; this behavior is instigated by the user but then is algorithmically driven.

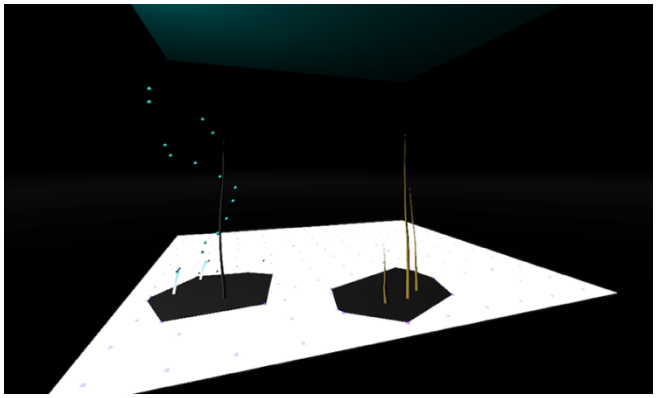


Figure 9: the gray tentacle on the left is algorithmically controlled and contributes a sustained noise sound. The yellow tentacles are user-shaped and contribute low pitched resonances and harmonies.

Behavior 1. The default type 6 behavior is an algorithmically driven gray-colored tentacle. The tentacle begins as very short in length, elongates towards the

ceiling, and shrinks back into the void, occasionally repeating this behavior. This corresponds to swells and fades in volume of a selected frozen sample, the dark noise made from freezing a play position at the end of a wood pin strike sample. The computer also randomly selects target tentacle height, corresponding to peak volume of sound. If the user captures multiple qubits in voids, there is an algorithmic counterpoint between tentacles, each an independent phrase of dark noise.

Behavior 2. By displacing a void, the user begins a duet of influence over the appearance, timbre, and harmony of tentacles in that void. At the beginning of this duet, the engine switches from playing dark noise to playing a randomly selected sample from the library of time stretched metal tubes, very low and resonant pitched sounds. Again, these samples have a timbral morphology from darker to brighter.

Timbral brightness is jointly determined by user and computer. Under the hood, this is because the sample is played either forward or in reverse, and direction of play is jointly determined. While a user displaces the void, its tentacles grow and shrink along a trajectory, interpolate from gray to yellow in color (see Figure 9), and grow timbrally brighter (forward play). The user seemingly causes this brightness entirely, though it is the result of sample design and input mapping. If the user lets go of the void, the tentacles reverse their grow-and-shrink trajectory, interpolate to gray, and become timbrally darker (reverse play). It is even possible for a user to repeatedly displace and let go of a void, with any rhythm of influence, surfing timbre back and forth with nuance.

Also, harmonies are formed between the tones of multiple tentacles; a harmonic shift occurs when each tentacle moves to a new tone, and this process is a jointly determined endeavor. The user determines the timing of each harmonic shift. By continually displacing a void, the user can keep the tentacles elongated, prolonging the duration of all active pitches. Tones only change when the user lets go and the tentacles have completely shrunk into the void. The next time the void is displaced, the algorithm solely determines which tentacles change pitch and to which pitch, via sample selection and transposition. This algorithm ensures some tones do not change, creating common tones between harmonies, a gradual progression.

Behavior 3. If the user displaces a captive void long enough for tentacles to become their brightest and yellowest, an additional transformation takes place: each qubit morphs into an orbiting sphere, tracing ellipses inside the void (see Figure 10). Each orbiting qubit switches from its time stretched sample (of a particular tube object) to a phrase of feather-beam granulation that features the same tube. This design was planned as a poetic device. It is as if the user slowly stretches time in reverse to the moment a pin strikes an object, and at that moment, the sound morphs back into its source, a series of unstretched pin strikes.

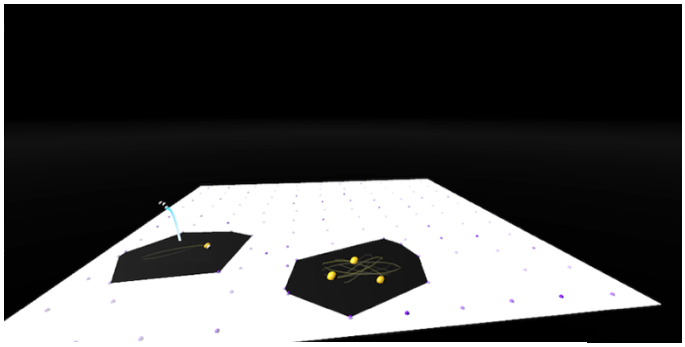


Figure 10: orbiting qubits, playing feather-beamed rhythms

The user triggers the start of orbiting, but they do not influence the trajectory of orbits, each a pre-recorded phrase sample. The speed of the rhythm corresponds precisely to the visual speed of orbiting. This is possible because when the feather-beam samples were recorded, the grain rate was also recorded to a data file with this design in mind. The data is played back real-

time in Max/MSP to drive the orbits in Unity. After playing its phrase sample, each qubit morphs back into a tentacle, algorithmically shifting from yellow to gray, playing its time stretched tube sample in reverse, and shrinking into the void. Without the user displacing the void again, each tentacle resumes its entirely algorithmic life, behavior 1 described above.

- **Ambient Light**

A pervasive light in the distance illuminates the horizon or fades away, its timings and values determined by the computer. This is the one character that is not a prefab/poly~ in design as there is only one ambient light. The ambient light's sound is a mixture of lowpass filtered generated noise and frozen metal tube samples transposed very low. Brightness of light corresponds to loudness of sound. In real-time, the frozen samples are transposed up and down by low frequency oscillators (LFOs). Occasionally by weighted probability, the engine selects an event with more energy, a brighter light and louder crescendo that temporarily overwhelms the rest of the sounds in the space.

- **Virtual Particles**

Algorithmically driven, each void randomly selects moments in time to send out virtual particles, each a tiny white particle moving along a catapulted path. Each particle corresponds to a single wood pin strike sample. Sometimes the virtual particle has a visual tail, marking its trajectory in space, and corresponding to the sample being fed into one of many delay chains with feedback, for various rhythmic effects. The more voids that are open in the space, the denser the overall presence of virtual particle sound. By weighted probability, each void either remains in a sparse state of emission or briefly emits a denser flurry. A third state is caused by the void igniters, described next.

- **Void Igniters**

Void igniters are cylinders that occasionally fall from the ceiling, algorithmically driven. Sometimes they disappear while falling, but occasionally they reach the floor, then undergoing curvilinear random walks. While falling, void igniters play moments from a pin swirl sample (swirling gestures on bricks) starting from a randomly selected play position. While randomly walking on the floor, igniters play the same sample, transposed lower, and a series of roly-poly samples (pins rolling on the audio equipment case). Making a behavioral connection between igniter and void, if one of these characters collides with a void, the void sends out an even denser emission of virtual particles for a brief duration. The sound of this event is a selected phrase of feather-beam rhythms constructed with wood pin strike samples.

- **Geysers**

A geyser is an algorithmically driven particle system that periodically emits from the center of a void and travels upwards (see aqua colored particles in Figure 11). Geysers play a randomly determined mix of up to five voices, each a different crinkling sound of one of the fiber baskets. The sounds are also run through delay chains for additional presence. Because each void has its own independent geyser, the more voids that are open, the denser this activity of sound. In the beginning of the global evolution, the geyser particles disappear before reaching the ceiling. Later, however, streams occasionally travel higher, colliding with and activating ceiling lights.

- **Ceiling Lights**

By affecting the global evolution, the user triggers when ceiling lights first glow and sound (see "The Global Evolution" below); thereafter, this system is governed by algorithms, random values, and chance collisions between geysers and lights[11]. 36 lights are located on the ceiling. When a geyser strikes a particular ceiling light, it glows, starting dim, growing in brightness, and then dimming again (see Figure 11). This corresponds to crescendos and decrescendos of synthesized washer models, not yet with the additional spinning effect. When multiple lights are struck by a geyser or geysers, the models collectively create pitched harmonies, with higher frequencies than those provided by the harmonies of the yellow tentacles. This design invites joint participation between computer and user over the frequency range of harmonies, scene-wide. Algorithms play the highs, synthesis located on the ceiling, and the user chooses when to contribute lows, time stretched samples emanating from tentacles on the floor.

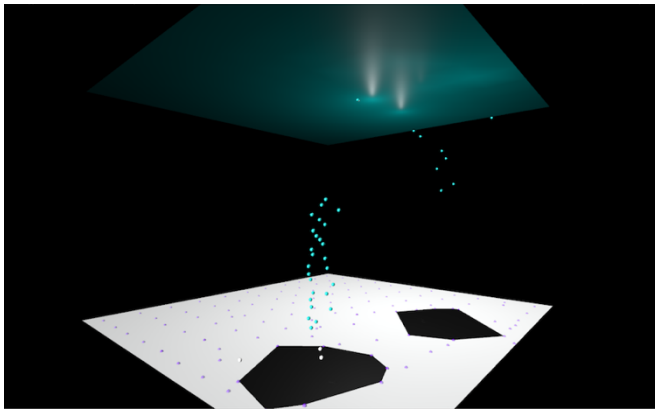


Figure 11: ceiling lights are triggered by collisions with the geyser particles. These lights provide mid to high pitched harmonies.

Each ceiling light's script randomly selects one of the 50 models made by the model-making script. When a ceiling light finishes playing its previous model, it randomly selects another one. Thus, if the score script has not yet moved to the next set of conditions and the same light is activated again, there is sonic variety. The score progresses to the next set of conditions the moment all 36 lights become collectively silent. The score repeats when it gets to the last set of conditions.

Later in the global evolution, tentacle qubits reach high enough to collide with lights, causing the lights to spin, adding the spinning sound effect to the washer model.

The Global Evolution

Again, the global evolution is a composed trajectory of five states, different rule sets. The total number of qubits captured in all voids (number of type 6) corresponds to a specific state in the global evolution. Because the user's actions capture qubits in voids, intentional or not, they are responsible for instigating each change in state. As a forward trajectory, the user cannot skip states, capturing qubits one by one. A user can dissolve a void (freeing its captured qubits), returning the evolution to a previous state; if that void contained multiple qubits, the trajectory returns to a state several steps back. I now will discuss the most relevant details of each state, highlighting the sonic evolution and an increasing connectedness between characters.

In the beginning state, all sounds are noise-based except for the ambient light sound, sometimes only generated noise but sometimes mixing in frozen metal tube sounds, transposed very low. When a user opens a void, or voids, geysers disappear before reaching the ceiling, unable to yet play the ceiling lights. The user is not yet aware of the lights.

[11] The user has a small influence over which ceiling lights are struck. As a user reshapes a void, its centroid changes, moving the geyser location.

The second state begins when the user has trapped a total of two qubits in voids. There is then an algorithmically controlled counterpoint of dark noise, gray tentacles growing and shrinking. Meanwhile, this state introduces the geysers reaching the ceiling, revealing ceiling lights and the first mid to high pitched sounds/harmonies. This perhaps intermingles with the user-shaped yellow tentacles, lower harmonies.

The third state is mostly a visual one. The tentacle qubits become taller (also playing louder) but not yet reaching the ceiling. This prepares the behavior revealed in the next state.

In the fourth state, tentacles sometimes reach the ceiling, colliding with and causing lights to spin (see Figure 12). This introduces the spinning sound effect. Gray tentacles trigger spinning lights at a relaxed rate, as per the underlying grow-and-shrink algorithm. By displacing a void and influencing the captive tentacles, here the user influences when lights spin, potentially creating a higher rate of spinning events. In another joint venture, upon displacing a void, the user determines when the tentacles *could* reach the ceiling, and the computer determines whether it actually happens on that particular displacement. In this state, there is a connectedness between all the major VR characters, qubits creating voids, voids creating tentacle qubits, and both geysers and tentacles reaching high enough to play the ceiling lights.

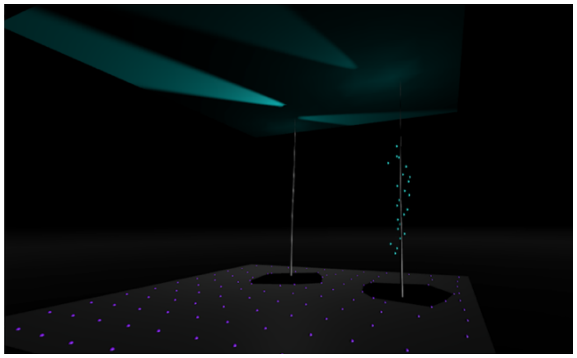


Figure 13: lights additionally spin when the tentacles reach high enough to collide with them. These correspond to the synthesized sounds of spinning washers.

In the final state, tentacles still spin the ceiling lights, but geysers are no longer responsible for the lights glowing. Each light, now bluer in color, glows and fades according to its own algorithmic timer, creating a much more active collective (see Figure 13). This mass must be kept in check, as running 36 resonators~ engines simultaneously is too CPU-intensive, given an otherwise active scene. A C# script manages the activation of engines so that a maximum of 10 engines run simultaneously. The harmony switches to a new progression of 18 conditions. This progression begins as more harmonic and higher in frequency range and gradually gets more inharmonic and lower. Upon reaching the most inharmonic series, the progression then repeats, sonically opening up again by returning to the higher range and maximum harmonicity.

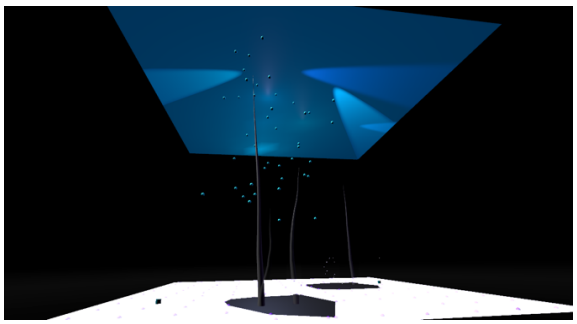


Figure 12: the final state in the global evolution features very active ceiling lights, which glow algorithmically by themselves. Tentacles still make them spin.

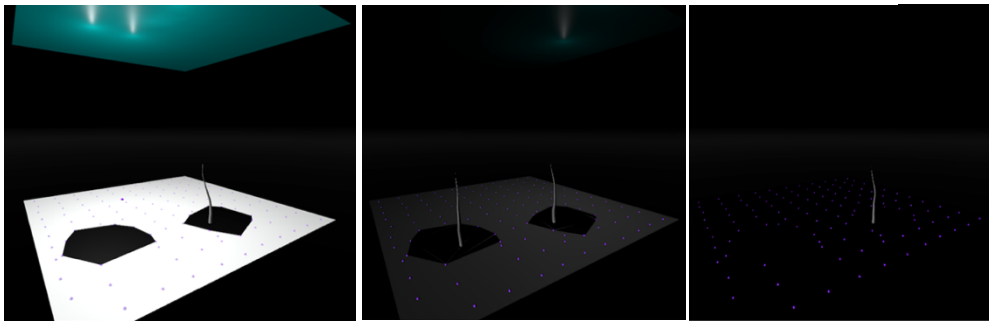
Global Filtering and Effects

There are two engines that apply filtering and effects to every sound in the space, one being entirely algorithmically driven and the other being activated and shaped by the users' location and movement in relation to the voids.

- **Directional Lighting**

An algorithmically-controlled directional light either illuminates the scene or rotates away from it, creating various degrees of visual darkness. A fully-illuminated scene corresponds to all sounds being unfiltered, and a fully-darkened scene corresponds to all sounds being filtered. The light's rotation angle is mapped to a crossfade value, creating a continuum of sound between unfiltered and filtered (see Figure 14). When the crossfade is close to unfiltered, the filter type is randomized, either lowpass or highpass, so that the darkened space has a varied timbre and feeling.

Figure 14: a directional light changes the filtering of all sounds, scene-wide.



a) unfiltered *b) mix of filtered and unfiltered* c) *entirely filtered, either lowpass or highpass*

- **The Rightside-up and the Upside-down**

Introduction. The user can hear and see the sound space through two different lenses, or perhaps visit two parallel universes. These are the rightside-up and the upside-down. The rightside-up is visually colorful, as seen in all previous Figures, and sounding as previously described. The upside-down, on the other hand, is visually warped and in grayscale (see Figure 15). In the upside-down, all the sounds of the rightside-up are processed with a chain of effects, including a lowpass filter, transposition, flange, reverb, and the Max/MSP object pitch-stutter~ (Regents of the University of California, 2007), which employs a lightweight granular processing engine with delay chains.

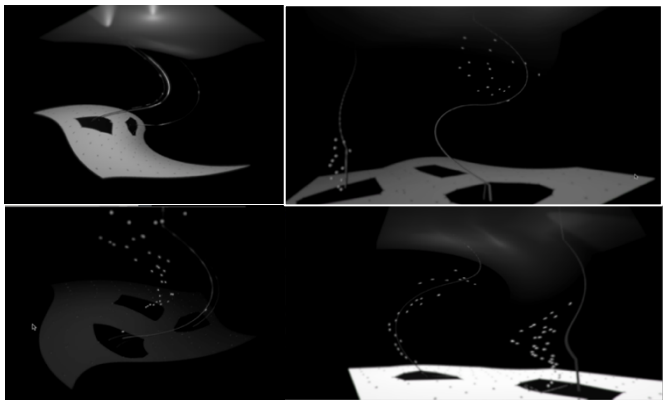


Figure 15: various vantages in the upside-down, where all visuals and sounds are additionally processed, a different "lens" on the sound space.

Warping in the rightside-up. The user begins their experience in the rightside-up. If they happen to move close enough to a void, they activate another independent rule: space becomes visually warped and blurred, like a vortex centered on the void (see Figure 16), and pitch-stutter and reverb effects are enabled. The user's distance to a void increases or decreases the power of these effects. The user can reverse away from the void, returning to an un-warped audiovisual point of view. Upon crossing a certain threshold distance, there is another duet of influence. The void algorithmically pulls the user towards it, increasing the power of the audiovisual effects, but the user can reverse away, reducing this power. The user can surf the power of these effects back and forth through a combination of reversing or allowing the computer's inward pull. The user is transported to the upside-down by passing through a void.

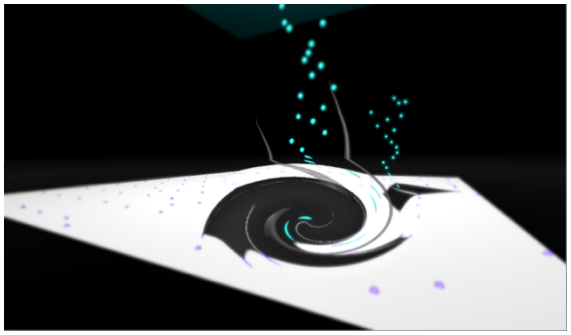


Figure 16: in the rightside-up, if the user approaches a void, space is warped and sound is additionally processed

The upside-down. The original design of the upside-down sent all audio of the rightside-up into a time stretching o.granubuf~, in real-time. Though this version was very compelling, it was computationally too heavy. I decided to build a lighter weight engine that still provides an alternate lens on the sound, the chain of effects described previously. A user's movements in the upside-down shapes these effect engines in various ways.

While the user rotates around the floor centroid counterclockwise, the sounds transpose up little by little and while they rotate clockwise, sounds transpose down. Rotation in any direction distorts the sound, by downsampling at various rates.

A threshold distance to the floor's center is a switch between two collections of values, sent to the effect engines. When the user is more distant than this threshold, reverb is less present, the lowpass filter cuts out higher frequencies, and pitch-stutter creates a rhythmically steady pulse of granulation. When the user crosses the threshold, now closer to the floor, reverb is more present, the lowpass filter lets through higher frequencies, and pitch-stutter creates sporadic, choppy rhythms. This is analogous to how a physical surface might seem smooth from a distance, but zooming to ever-smaller scales, it is revealed to be rougher. The user can zoom in and out repeatedly to examine and contrast each set of conditions.

The way back to the rightside-up is again through any void. Here, if a user comes close to a void, instead of pulling them in, the void algorithmically pushes them away. This push corresponds to temporarily explosive feedback induced from a delay chain, which immediately calms down as the user is distanced from the void. The user can surf back and forth between zooming in on a void and allowing the algorithmic push away, flirting with dangerous feedback (though it is a controlled danger). If the user manages to pass through a void, they are transported back to the rightside-up.

CHAPTER 5:
CODE UTILIZED OR DEVELOPED

This chapter explores some of the code used for *QuBits*, some of it written by me for this project, and some of it being the work of other authors, tailored to my needs.

Centralized Scripts

It is useful to have a centralized script that performs a task for an entire community of VR characters. By not having every instance do the work for itself, processing is saved, the system's design is easier to understand and edit, and as shall be described, this lends itself perfectly for controlling the sonic quality of a mass. Two of the most important centralized scripts are described below. For each, the design principle is the same: every instance of a VR character class references a centralized script to receive current values, then seeding its own algorithms with these values.

- **Define All Important Values, Project-Wide**

A centralized script, the "mixer script", defines every important value/range, project-wide, making the system more readily compose-able and tweakable. Having never built a system of this nature until *QuBits*, my naïve first approach was to define values wherever they were used, in numerous scripts and in Max/MSP. This made tracking down and editing values very cumbersome, potentially in completely different software platforms. A better design was eventually implemented with the mixer script. When a value changes depending on the global evolution state, these values per state are also defined in the mixer. This script is not a Monobehavior class but rather a generic one, as it contains constants, values that do not change at runtime.

Each instance of a character class sets its own like-named variables to the mixer's values in its Start() method. When starting the project, appropriate values are also sent to Max/MSP, telling specific Max objects how to behave and what kinds of sounds to produce when receiving input[12].

- **Govern Sound Masses**

A centralized script, the "mass script", is responsible for controlling possibility within a sound mass. Each character class instance randomly selects a specific value from the range currently allowed by the mass script. The mass, then, is built from the bottom up, from individually determined random values, but these values are shaped from the top down, by the will of the mass script. Some examples of parameters the mass script controls are 1) the probability of each qubit becoming a type 2, affecting the density of the larger pin ricochet mass, 2) the transposition range of this pin ricochet mass, 3) when the voids collectively flutter, and 4) when the voids are allowed to emit virtual particles (creating periodic breaks in this activity)[13].

---

[12] These values are sent into Max object inlets to configure the arguments of the objects, thus tailoring the behavior of the operations when they are utilized.

[13] Additionally, though not the mass script itself, the score script for the ceiling lights, described previously, is another script that controls a sound mass, constraining the collective harmony of the ceiling lights.

For each of these parameters, the mass script runs an independent random timer in its Update() method to periodically change ranges of values, then locally evolving the mass.

Centralizing Global Evolution Changes Within Each Script

Within each character type's class I created a centralized method, the EvolutionParams() method, whose task is to manage every value and rule that changes by global evolution state. By consolidating all of these alterations, I can easily reason about and compose the evolution for each character type.

- **Changing Values**

All *values* contingent on the global evolution belong in the EvolutionParams(). This example from the geyser class demonstrates the pattern, then discussed below:

```
1   void Start()
2   {
3       EvolutionParams();
4
5       dissolveB4Ceiling = Random.Range( 0f, 1f ) <= dissolveProbability;
6       speed             = Random.Range( speedRange[0], speedRange[1] );
7   }
8
9   void EvolutionParams()
10  {
11      if( globalEvolutionState == GlobalEvolution.GlobalEvolutionState.begin )
12      {
13          dissolveProbability = 1;
14          speedRange          = new Vector2( 2, 10 );
15      }
16      else if( globalEvolutionState == GlobalEvolution.GlobalEvolutionState.beginCeiling )
17      {
18          dissolveProbability = .4f;
19          speedRange          = new Vector2( 5, 11 );
20      }
21  }
```

Lines 9 – 21 define the EvolutionParams() method. Here, for different states in the global evolution, two parameters are defined, "dissolveProbability" (the probability the geyser disappears before reaching the ceiling) and "speedRange" (a range of acceptable geyser speeds)[14]. EvolutionParams() always executes first[15], line 3, setting every value that varies by global evolution state. This then configures the behavior of algorithms that follow and make use of those values, lines 5 and 6. [16]

- **Changing Rules**

It is possible to change rules by evolution state, also utilizing the centralized design of the EvolutionParams() method. It was previously described that at first the geysers cause ceiling lights to glow, however, in the final evolution state, the lights cause themselves to glow according to their own algorithmic timer. This is not just a change of value! Rather, this involves different C# methods, each a series of expressions to check for the correct trigger that causes glowing. Other coding patterns are possible, but to consolidate all logic to

[14] These values are in reality defined in the Mixer Script. Values are shown just to make the example clear.

[15] Most character classes run EvolutionParams() once per frame, in their Update() method.

[16] The details of these algorithms are not essential to discuss.

EvolutionParams(), C# “delegates” are used. A delegate is a generic method that can be dynamically set to a specific method. This example demonstrates the pattern, then discussed below:

```
1   // **** run once per frame
2   void Update ()
3   {
4       EvolutionParams();
5       CheckGlow();
6   }
7
8   // **** set delegate to specific method
9   void EvolutionParams()
10  {
11      if( globalEvolution == GlobalEvolution.GlobalEvolutionState.beginCeiling )
12      {
13          CheckGlow = CheckGlow1;
14      }
15      else if( globalEvolution == GlobalEvolution.GlobalEvolutionState.final )
16      {
17          CheckGlow = CheckGlow2;
18      }
19  }
20
21  // **** define methods
22  void CheckGlow1()
23  {
24      // code to see if hit by geyser
25  }
26
27  void CheckGlow2()
28  {
29      // code to glow by self (random timer)
30  }
```

Above, CheckGlow() is a delegate, not defined until the EvolutionParams(), lines 9 - 19. Inside EvolutionParams(), the delegate is set specifically to either CheckGlow1(), the check for geyser collision, or CheckGlow2(), a timer responsible for glowing. When CheckGlow() is then executed in the Update() method, line 5, it points to one of the two methods. This powerfully allows for the consolidation of changes in both values and rules, facilitating a much better fluidity in reasoning about and tweaking the evolution.

Script Abstractions to Shape Time

There were a couple behaviors I found myself coding over and over, each pertaining to the shaping of time; I eventually created a class for each that can be reused and customized whenever needed. These classes allow me to spend more time trying out and honing new behaviors for VR characters, quickly giving each instance an independent trajectory.

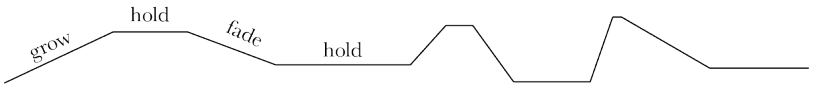
- **Growing and Fading**

The GrowAndFade() class generates a constant trajectory of increasing and decreasing values over durations of time. Target values and durations are randomly selected but conform to desired ranges. This is useful for creating behaviors that shift little by little including crescendos and decrescendos, lights glowing and fading, entangled qubits transitioning back to unentangled, and tentacles growing and shrinking, also corresponding to amplitude of sound. The basic concept for any of these is linear interpolation, used to gradually morph between two values. In

this algorithm, there is an initial value, a target value, and a duration to morph from initial to target. Linear interpolation, over a duration of time, is expressed as:

$$\text{current value} = \frac{\text{duration passed}}{\text{total duration}} * (\text{target value} - \text{initial value}) + \text{initial value}$$

The GrowAndFade() class also allows a value to be held for a duration. This class then might generate a graph of values like:



8 arguments configure the behavior. Two arguments configure what happens when the trajectory begins (hold or not, grow or not). The rest of the parameters are ranges to constrain random selection of target value and duration. The duration ranges are independently customizable for growing, fading, holding while grown, and holding while faded. Once these ranges are defined, GrowAndFade() runs in an Update() method, generating a trajectory as in the above graph *in a single line of code*. Here is an example that could be used to make tentacle height carry on an independent life:

```
public class Qbit : MonoBehaviour
{
    GrowAndFade growAndFade = new GrowAndFade();
    float tentacle_height;

    // values for arguments
    bool    start_hold     = false;
    bool    start_grow     = true;
    Vector2 targetGrow     = new Vector2( .6f, 2.5f );
    Vector2 durToGrow      = new Vector2( 2, 6 );
    Vector2 targetFade     = new Vector2( 0f, .1f );
    Vector2 durToFade      = new Vector2( 3, 4 );
    Vector2 durToHoldGrown = new Vector2( 1, 3 );
    Vector2 durToHoldFaded = new Vector2( 2, 5 );

    // run once per frame
    void Update()
    {
        tentacle_height = growAndFade.ReturnCurrentValue( start_hold, start_grow, targetGrow, durToGrow,
                                                          targetFade, durToFade, durToHoldGrown, durToHoldFaded );
    }
}
```

- **On/Off Durations**

Another class, the OnOffTracker(), is used in cases where a behavior should turn on for a duration, then off for a duration, then repeat this over and over; fine control is possible in specifying longer vs shorter "on" durations, longer vs shorter "off" durations, and the weighted probability of selecting each (long on, short on, long off, short off). This customizable binary switch allows layers in the sound space to float in and out, providing great variety of combination between any script running the class. The OnOffTracker() appears often in the mass script to switch on and off the activity of an entire mass. It is used, for example, to govern the voids collectively being allowed to emit virtual particles and collectively being allowed to flutter. The OnOffTracker() also runs in any Update() method, requiring only a single line of code.

Detecting and Rendering Voids

- **Delaunay Void Detection**

A Delaunay triangulation for a set of points is a triangulation such that no point is inside the circumcircle of any triangle (see Figure 17). Hervías, et al. (2013), used this configuration of triangles to detect cosmological voids, which they define as large empty spaces in maps of the distribution of galaxies. I will refer to their method as Delaunay void detection, described as follows. First, draw the Delaunay triangulation for a set of points. In this configuration, the longest edge spans a void (Figure 18b). The two triangles that share this longest edge are included with the void, which then has four edges (Figure 18c). Next, the triangles sharing an edge with each of the previous step's edges are included with the void (Figure 18e). The void then consists of six triangles (Figure 18f). One could continue expanding outward, including more triangles with shared edges; however, six-triangle voids are a perfect size for this project[17]. Given a set of points, this describes how to find one void, specifically the one containing the longest edge of the set. The full algorithm to find all voids, given a set of all qubit positions, requires further steps.

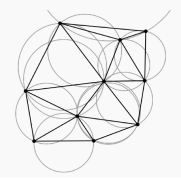


Figure 17: a Delaunay triangulation with circumcircles shown
Image: Nü es (2006)

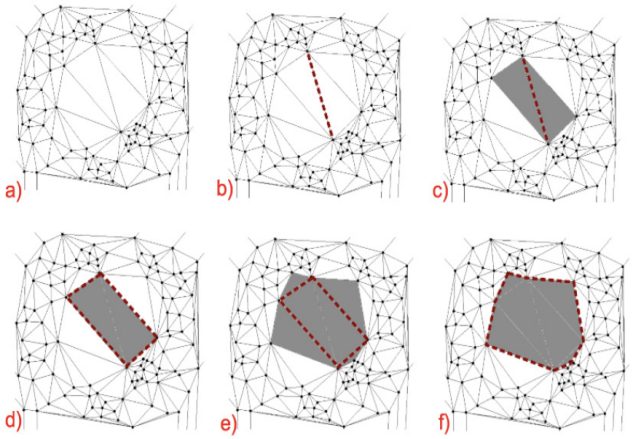


Figure 18: using a Delaunay triangulation to find a six-triangle void.
Image: Hervías, et al. (2013)

- **The Method Only Proposes New Voids**

Before describing the full algorithm, it must be noted Delaunay void detection merely proposes new voids. After a void is created, this detection no longer informs the shape of that void. Instead, each void's vertices are defined by wherever its bounding qubits happen to travel, rendered as a triangulation of those qubits (see Figure 19). Voids are proposed on every visual frame in Unity, but each is only instantiated subject to two conditions. 1) The centroid must be distant enough from the centroid of each void that already exists, i.e. no void already exists in that local region. 2) The area must be above a desired threshold.

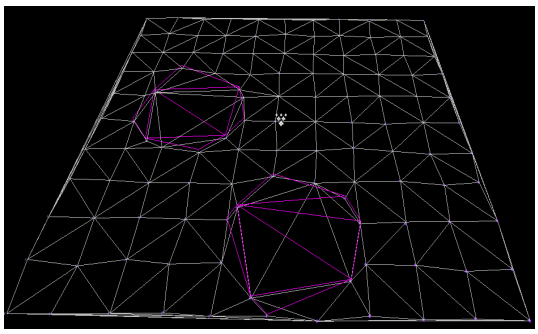


Figure 19: Delaunay analysis and two already-existing voids. The Delaunay analysis is in white. The triangulation of the void's bounding qubits is in magenta. The void does not follow the Delaunay analysis after formation.

[17] My implementation of this algorithm sometimes results in the void consisting of five triangles.

- **Procedure of the Full Algorithm**

These six steps are executed on every visual frame in Unity.

1) Run the Delaunay triangulation on the set of points, the set of qubit positions (Fortune, 1994; Shaw, 2009 – ActionScript3; Ceipek, 2014 – C# translation). Exclude type 5 (those allowed to enter the voids), type 6 (those already inside a void), and those whose positions are on the convex hull, the convex polygon bounding the entire set (see Figure 20).
2) Order all triangles by longest edge, an ordered list. For each triangle in the list, there is a reference to the qubits involved in the triangulation so bounding qubits can later be identified.
3) Inspect the next triangle in the list[18], which has the longest edge of triangles not yet inspected. Assemble a proposed void using the method: find and include the triangle sharing this longest edge. Find and include the four triangles that share edges with the previous two triangles, arriving at six triangles. Also, only include any triangle in this step if it has not yet been tagged as "inspected", step 5. A triangle, then, can only participate in one void.
4) Test the two conditions of void formation. The area of the six triangles must be greater than the area threshold. The centroid of the six triangles must be distant enough from all other void centroids. Only create a new void if both conditions are true.
5) Tag each of six triangles as having been inspected.
6) Iterate steps 3 – 5 for the next triangle in the list, until all triangles are inspected.

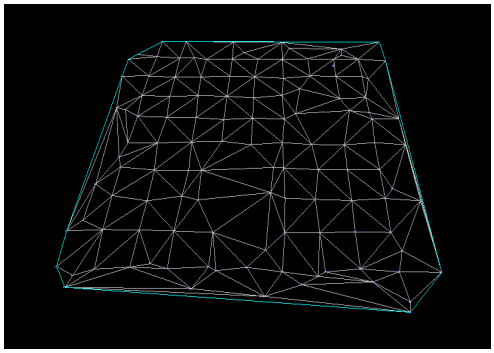


Figure 20: qubits on the convex hull (those touching the aqua colored polygon) do not participate in the formation of voids.

[18] The first time through, this means the first triangle in the list, which has the longest edge of the set.

CHAPTER 6:
FUTURE DIRECTIONS

The following discussion considers directions for future iterations of *QuBits* and entirely new projects, informed by the strengths and weaknesses of principles explored herein. I will discuss the scale-ability of each principle, referring to my own ability to use that principle successfully to grow a project into a more expansive environment. This includes considerations of how time-consuming it is to execute a principle even once, how difficult editing the environment becomes due to that principle, and CPU load as that principle is employed for ever-more voices.

Sound Mass

- **Pros and Cons of Sound Mass with Samples**

Though I find sampled and granular sound to be beautiful, this design poses four challenges for scale-ability, especially in an environment of multiple evolving sound masses. 1) It is very time-consuming to develop each source sample library. 2) Granular synthesis can be computationally heavy, so given a sound mass design, these phrases must be recorded to samples. It is very time-consuming to develop each granular phrase library. 3) Recorded sound is fixed, not generated, posing a problem if some other type of sound is instead needed. Originally, for the sound of the yellow tentacles, I recorded a source library of higher-pitched metal sounds, then time stretching these samples for a phrase library. In the context of the project, I decided the sounds were too similar to the synthesized ceiling lights, and transposing the samples down was not convincing enough. I had to go back to the studio to record another source library of large, low-sounding metal tubes, then again repeating the time stretching process. This makes for a very slow-going workflow. 4) Some ideas cannot be accomplished using samples without a very complex design. For instance, it would have been better if the user was able to shape the speed of orbiting qubits, corresponding to speed of rhythmic pulse. Real-time granulation of a multitude of orbits is too heavy, and though a sample library approach is possible, this design would be unwieldy. To give the user this control, for each metal tube, there would have to be four libraries, phrases broken down into slow pulsations, accelerations, fast pulsations, and decelerations. Adding to the design problem, each accelerating and decelerating sample would have to arrive at the correct tempo in a corresponding slow or fast pulsation sample. Due to sample design, this interactive possibility was avoided in this iteration.

- **Pros and Cons of Sound Mass with Synthesis (and Physical Modeling)**

Using synthesis as the atomic unit of a sound mass solves some of the cons associated with samples, though not all of them. Given that a sound can be generated from a physical model, the tedious task of creating the source library is eliminated. Variety is then obtained by seeding the model with different values. Editing a model also is most likely faster than re-recording an entire library of sounds. On the other hand, as a replacement for samples, synthesis is not a magic bullet for eliminating CPU overload. In the current iteration, the washer sounds were synthesized, and yet given an otherwise active scene, the mass needed to be limited to ten voices. Also, it is not computationally demanding to play a single sample. Rather the granulation of those samples becomes heavy, iterating through each overlapping grain to write to

an audio buffer. An engine that granulates synthesis, based on FFT analysis or a physical model, might be similarly taxing. Finally, the use of synthesis provides more control and immediate variation, but for me, the sounds are not as desirable in sonic quality[19].

- **Implications for Sound Mass Design in Future Projects**

In no way do I plan on abandoning samples, but as an experiment in scale-ability, I wish to put a greater emphasis on exploring sound mass with synthesis. For me, the most compelling sound in this project is the swarm of type 2 qubits, smaller masses summing to a larger mass with its own distinct identity. I would like to take this idea and scale it to more layers of activity, and synthesis could be useful for this task. *QuBits* could be scaled to more grids/layers of qubit characters, each layer consisting of 121 instances, having its own timbral identity, and having an independent trajectory of density. The environment would then be much more varied, multiple sound mass colors weaving in and out of the scene. Synthesis would not necessarily save processing power, but it would offer great control in experimenting with, and when needing to edit, the timbre of each layer. Each layer could evolve over time through seeding the synthesis algorithms with different sets of values, or perhaps through linearly interpolation of values. Samples could be included as the identity of some layers.

Spatiality and Immersion

- **Future Iterations of QuBits**

The experience of space in *QuBits* is not as immersive and visceral as I wish. The user is constrained to a rather small area of virtual space, a play area around the floor they cannot move beyond. To me, this feels like an interaction with an object in space rather than a strong exploration of space itself. However, rather than solve this issue of spatial scale for virtual space, I am fundamentally more attracted to physical space. I feel a particular quality of immersion and wonder from the inherent properties of physical space (and by moving through it), a sensation I do not experience thus far while interacting with virtual space. The *QuBits* project could be translated to physical space. I will refer to the user here as visitor. Even if the visitor has the same limited scope of movement, say installed in a room, I believe physical space would drive a more immersive experience and would be better suited for demonstrating energy as algorithmically created spatial sound. Visuals could be projected on multiple surfaces, the visitor's movement captured with sensors, speakers placed at multiple locations (perhaps also using open-ear headphones), and some of the virtual physics would need to be remapped. Another issue in the current iteration is that a user's distance to a sound source does not affect the energy (volume) coming from that source. This could be remedied by employing a spatial audio engine, such as SPAT (Jot, 2018), ICST Ambisonics (Schacher & Kocher, 2015), or Atmos (Dolby Laboratories, 2018).

[19] A synthesis engine was created for comparison to granular time stretching and freezing. This synthesis engine first obtains an FFT analysis of a pin strike sample and then time stretches or freezes the analysis through similar methods of randomization around a play position. This engine is modeled after the work of Charles (2011). The synthesized version is perhaps "cleaner" sounding, less airy or noisy; however, for my taste, I prefer the granular version for its airiness and liveliness.

- **Other Future Projects**

A future project could engage with a vast field of space and an entire community of visitors. Here, the motion of multiple visitors could be detected with sensors. Most exciting to me is the possibility of locating different textures and sound masses at different positions in space, emitting from speakers placed at those locations. Visitors could zoom in on particular masses by physically moving close in proximity, exploration dependent on movement across large distances. A visitor would also be able to shape the trajectory of their local sound mass, through similar principles of interactivity explored in *QuBits*. A compelling feature is that instead of a duet of influence, local change would be scaled to an entire "ensemble of influence". This is precisely the kind of work I aim at creating beyond this project. This discussion will be continued in the next section.

Evolution

- **More Inter-Relationships**

In a future iteration of *QuBits*, there could be more discoverable independent rules that place character types in *inter*-relationships, where each affects the other. The most direct inter-relationship in the current iteration is that bounding qubits shape voids and voids shape where bounding qubits can go.

Additional inter-relationships could be explored between qubit and void. A logical next step would be that voids capture qubits *and captured qubits likewise affect voids*. Perhaps if an individual void captures a threshold number of qubits, the void becomes permanent, no longer able to be reshaped, and visually sealing off, its trapped qubits no longer visible.

With permanent voids in the system, further opportunities would arise for inter-relationships between characters. Thus far, void igniters affect voids, but voids do not affect void igniters. Perhaps an igniter that travels close to a permanent void is pulled into orbit just outside the void's perimeter/convex hull, introducing a new type of rhythmic sound. As more igniters went into orbit, this sound would become another mass.

There could be an iteration of *QuBits* where the upside-down can affect the rightside-up, and vice versa. It would be interesting to make the processed sound of the upside-down sometimes "leak" into the rightside-up, perhaps corresponding to some visuals trying to make their way out of a void. Then in the upside-down, perhaps there are occasionally quick interpolations sonically and visually to the conditions of the rightside-up, like flashes of the other universe, but immediately interpolating back to the upside-down. An additional possibility is for the upside-down to play a role in evolving the global state. Perhaps there is something the user can only activate in the upside-down that then affects the global state, then also affecting the right-side up.

- **Better Global Evolution Rules**

The premise that the user influences the global evolution is interesting to me, but the specific rule used in this project deserves reexamination. First, each state change in the current iteration is an instantaneous switch, beginning the moment another qubit is captured in a void. Instead, it might be examined how an accumulation within a state could be the rule to change to the next state. For instance, any twice-repelled qubit could become red in color, more frenetic in its type 2 random walk, and louder in volume. The mass, then, would build sonically and

visually towards a threshold number of red qubits, a threshold that begins the next state. When the next state begins, all qubits return to their initial color and play quieter once again, a release in energy.

Second, while VR characters grow ever more connected, shifting their governing rules and deepening the sense of discovery, the rule that evolves the global state remains fixed. Instead, the rule of evolution could change. Perhaps the transition from the first to the second state involves obtaining a threshold number of twice-repelled qubits, then announcing the entrance of the geysers playing ceiling lights. Later, however, the rule could change, requiring user and computer to obtain a threshold number of ceiling lights struck twice by tentacles. The ceiling lights would grow gradually redder and louder, eventually announcing the beginning of the fifth state, more active blue lights.

- **Other Future Projects**

Much as *QuBits* aims at an increasing interconnectedness of VR characters over time, a similar trajectory could be explored for spatially separated sound masses in a vast physical space. At first, each sound mass could behave disparately from the others, but visitor contributions, the ensemble of influence, could slowly increase similarity and connectedness, eventually arriving at a textural and harmonic unity across the vast space.

CPU load will be a great challenge in an attempt to afford an entire community of visitors interactive control over particulate sound mass. To make such a design computationally viable, at the time of this writing, most likely parallel computing needs to be utilized, recruiting the use of multiple computers and coordinating their work. Nonetheless, this future direction is essential in achieving my core vision, to evoke wonder in others as they explore and discover possibility in a vast, mysterious environment.

REFERENCES

Cella, C. (2013). Generalized series for spectral design. Unpublished manuscript.

Charles, J-F. (2011). Spectral Tutorials [Computer software]. Retrieved from https://cycling74.com/tools/charles-spectral-tutorials

Cycling '74. (2016). Max 7 [Computer software]. Retrieved from https://cycling74.com/downloads

Dolby Laboratories, Inc. (2018). Atmos Production Suite [Computer software]. San Francisco, CA. Retrieved from https://www.avid.com/plugins/dolby-atmos-production-suite#Features

Fortune, S. (1994). Fortune's Algorithm [Computer software]. AT&T Bell Labs. as3delaunay [Computer software] (2009) (A. Shaw, implementer in ActionScript 3). Massachusetts Institute of Technology, Cambridge, MA. Unity-delaunay [Computer software] (2014) (J. Ceipek, Trans. to C#) (2014). Retrieved from https://github.com/jceipek/Unity-delaunay

Gottfried, R. (2016). cv.jit.contours [Computer software]. Center for New Music and Audio Technology (CNMAT). Univsersity of California, Berkeley. Retrieved from https://github.com/CNMAT/Music-and-Computing/tree/master/help/io/cv.jit.contours

Gottfried, R. (2016). o.granubuf~ [Computer software]. Center for New Music and Audio Technology (CNMAT). University of California, Berkeley. Retrieved from https://github.com/CNMAT/CNMAT-odot/releases

Hervías, C., Hitschfeld-Hahler, N., Campusano, L.E., Font, G. (2013). On finding large polygonal voids using delaunay triangulation: the case of planar point sets. In J. Sarrate & M. Staten (Eds.), *Proceedings of the 22nd International Meshing Roundtable* (pp. 275 – 292). Switzerland: Springer International Publishing.

Hynninen, M. (2016). Spectral Granulator [Computer software].

iZotope, Inc. (2017). iZotope RX [Computer software]. Retrieved from https://www.izotope.com/en/products/repair-and-edit/rx.html

Jot, J-M. (2018). Spat [Computer software]. Institute for Research and Coordination in Acoustics/Music (IRCAM). Paris, France. Retrieved from https://forumnet.ircam.fr/product/spat-en/

Microsoft Corp, Xamarin Inc., MonoDevelop contributors. (2019). Visual Studio [Computer software]. Retrieved from https://visualstudio.microsoft.com/downloads/

Moore, B.C.J., Glasberg, B.R. (1983). Suggested formulae for calculating auditory-filter bandwidths and excitation patterns. *Journal of the Acoustical Society of America*, 74, 750-753.

Nü es (User name). (2006). Delaunay circumcircles. In *Wikipedia*. Retrieved June 15, 2019, from https://en.wikipedia.org/wiki/Delaunay_triangulation#/media/File:Delaunay_circumcircles_vectorial.svg

Postel, J. (1980). User Datagram Protocol. Retrieved from https://tools.ietf.org/html/rfc768

Regents of the University of California. (2017). odot [Computer software]. MacCallum, J., Freed, A., Gottfried, R., Rostovstev, I. Center for New Music and Audio Technology (CNMAT). University of California, Berkeley. Retrieved from https://github.com/CNMAT/CNMAT-odot/releases

Regents of the University of California. (2007). pitch-stutter~ [Computer software]. Center for New Music and Audio Technology. University of California, Berkeley. Retrieved from https://github.com/CNMAT/CNMAT-MMJ-Depot/releases

Regents of the University of California. (1999). resonators~ [Computer software]. Freed, A. Center for New Music and Audio Technology. University of California, Berkeley. Retrieved from https://github.com/CNMAT/CNMAT-Externs/releases

Schacher, J., Kocher, P. (2015). ICST Ambisonics [Computer software]. Institute for Computer Music and Sound Technology (ICST). Zurich University of the Arts, Zurich, Switzerland. Retrieved from https://www.zhdk.ch/forschung/icst/software-downloads-5379/downloads-ambisonics-externals-for-maxmsp-5381

Sixth Sensor. (2018). OSC Simpl [Computer software]. Retrieved from https://assetstore.unity.com/packages/tools/input-management/osc-simpl-53710

Unity Technologies ApS. (2018) unity [Computer software]. Retrieved from https://store.unity.com/download-nuo

Valve Corporation. (2019). SteamVR [Computer software]. Retrieved July 21, 2019, from https://store.steampowered.com/app/250820/SteamVR

Vive. (2019). Vive VR System [Computer hardware]. Retrieved July 21, 2019, from https://www.vive.com/us/product/vive-virtual-reality-system

Wright, M. (2002). Open Sound Control 1.0 Specification. Retrieved from http://opensoundcontrol.org/spec-1_0