Knots in the String of Time: Prototyping Tools for Interactive Music

Nick Venden
Bob Cole Conservatory of Music
California State University, Long Beach
250 N Bellflower Blvd.
Long Beach, CA 90840-7101
nick.venden@gmail.com

ABSTRACT

This paper documents an effort to adapt computer game prototyping tools as controls for interactive digital music. Within that effort the social aspects of interactive music were explored. Is it possible give the game-player-listeners the ability to structure the music interactively and become the composers of their own game experience? Throughout the process these social and aesthetic questions countered and informed the technical questions.

In every media composer's digital studio, the tools of creation and distribution have elided. Work is instantaneously and efficiently distributable on a global scale; many of the aesthetic difficulties specific to art-making in 2014 derive from this fact. This paper describes some of these difficulties strictly within the context of the rising demand for mediated interactive experiences.

The conclusions are decisive. Contemporary game prototyping programming techniques—even those limited to WebGL-enabled browser applications—are highly effective tools for the creation and control of interactive digital music.

CCS Concepts

• Human-centered computing→ Scenario-based design • Applied computing→ Computer games. • Social and professional topics.

Keywords

Unity3D; C# programming for Unity3D; generative music composition; computer game music; digital music; game audio; computer game theory.

Typographical Conventions

In order to indicate the various syntactic components of the programming languages C# and JavaScript, this paper uses the following conventions:

`Monospace` indicates code examples, code snippets, variable names and parameter names.

1. THE COMPOSITIONAL PROCESS

1.1 The Project in its Social Context

The genesis of *Knots in the String of Time* was a comment by Ivica Bukvic, Director, Digital Interactive Sound and Intermedia Studio (DISIS) at Virginia Tech University, who claims that "the [video] Gaming industry has prototyping tools that could be adapted for the development of interactive art installations" [7]. Bukvic's remarks made me wonder whether, indeed, such tools be adapted to create control for interactive digital music. A second point of inspiration comes from the contemporary conceptual artist Joseph Beuys:

> Every human being is endowed with creativity, an inborn, universally distributed faculty which lies fallow in most people. The social task of professional artists is to liberate this repressed creative potential until all human labor deserves to be called artistic. Essential to his belief in creativity is that it is future-oriented: it has the performative structure of a promise. [9]

I am a professional artist; thus, Beuys has charged me and other like me with a 'social task.' Could I build an interactive environment that would allow a non-professional to structure time, to sequence affect, to create cause-and-effect (i.e, musical meaning [19]) through the use of harmonic language, rhythm, and other musical elements, and do what composers do?

Music composition that combines with interactive digital technology typically enforces a reading of these questions strictly within the context of interactivity. But within that context lie other questions about the presentational vs. the participatory, authorship, art as appropriation, and intellectual property, among others. These latter and other social and aesthetic questions and considerations are more important than the technical, not merely because the technology explored in the project that this paper documents (and all other similar projects) will be outdated before the binding on the paper is dry, but because of difficulties specific to art-making in 2014, when this project first began.

CUNY lecturer and art critic Claire Bishop, in her landmark essay on digital art *The Digital Divide* [2], asks: "How many artists who use digital technology actually confront the question of what it means to think, see, and filter affect through the digital? How many thematize this, or reflect deeply on how we experience, and are altered by, the digitization of our existence?" For a composer the most intriguing of Bishop's challenges is to "filter affect through the digital." Taken together, Bukvic and Beuys imply, if not outright suggest, there can be interactive music that is both intrinsic and unique to the digital experience. Although this paper is primarily technical, its proper subject is how all these questions are linked—how each guided the others during the planning and evolution of this project.

To begin my work I needed to define the ideal listener-participant for this project. Interactive media no longer conflates the listener's experience and the music itself into a commodity. Rather, today's listener is conditioned to, and expects to have, an active role in the music itself. Film theory speaks of the omnipresent human

tendency to apprehend meaning by creating narratives out of sensory data.[1] A new form of (narrative) aleatoric composition, driven by the listener's choices, may be one possible solution to Beuy's challenge, but a complex narrative requiring hundreds of 3D models was out of the scope for this project. Nevertheless, there is something subtle and attractive in the notion of simply "creating narratives out of sensory data." This notion also provides method and constraint.

Contemporary Swiss installation artist Thomas Hirshorn states, "I do not want to do an interactive work. I want to do an active work. To me, the most important activity that an art work can provoke is the activity of thinking" [1]. Therefore, in Hirshorn's view, an 'active' music composition would require that listeners first give of themselves. Hirshorn, Pierre Huyghe, Santiago Sierra, Rirkrit Tiravanija, and other artists with a social-political practice activate their viewer, but their works do not use seductive, theatrical, and shallow interactive digital technology [4]. Viewers move through their installations and encounter surprising juxtapositions—new meanings in quotidian elements.

Their installation viewer-listener is akin to Baudelaire's *flâneur* or Flaubert's Frédéric in *L'Éducation sentimentale*, who wandered the arcades of 19th-century Paris looking at and listening to the kaleidoscopic manifestations of life in 1869 [30]. One hundred years later this 'wandering' became a practice of the *Situationist Internationale* movement in Paris in the late 1960s, which was epitomized by Guy DeBord's logic of the *dérive.* The *dérive* is the act of selecting, on the basis of chance, anecdote, and coincidence [17]. This has given way to the 21st-century act of internet surfing: the pursuit of impromptu, subjective connections while navigating the Web. Today the *dérive* is the logic of our dominant social field, the internet—a practice and precondition of my audience.

With all the aforementioned operating as equal parts inspiration and guideposts, I imagined *Knots in the String of Time* as a digital landscape for the listener to explore with the logic of the *dérive.* By moving through the digital space the listener triggers sequences of affect which are both musical and visual. The listener encounters isolated objects and "apprehends meaning by creating narratives out of sensory data." And for Beuys's injunction, hopefully, the listener joyfully structures musical events along *Knots in the String of Time* and becomes the project's composer.

Why do this? For me the attraction is not the 64-billion-dollar game industry. In digital music studios today the tools of creation and distribution have elided; work is instantaneously and efficiently distributable on a global scale. WebGL[2] plugins in the current Chrome and Mozilla Internet browsers—capable of rendering interactive, animated 3D graphics and multi-channel audio in real time—offer ubiquity and an astounding level of access to potential audiences. This should not be interpreted as power, but as potential for personal efficacy in a world where data have overwhelmed meaning; where no contemporary sculptor believes the world needs another object; where cynical and disillusioned MFA students work at dematerialized projects based on their purported 'social practices'; where a graduate composer of one more harmonic change for aesthetic effect has been made to feel ridiculous. A composer in such a world needs a social conscience.

This 'efficacy' demands constraint at the technical level: imagery must render in WebGL data within an Internet browser in real time. The smaller-scale and more intimate computer monitor with its near-field speakers demands a different expressivity. Audio events must be attached to simpler 3D visual elements such as formalist-geometric or figurative abstractions. Pre-rendered video files, displayed on simple 3D surfaces, can replace the more CPU-intensive dynamic surface-shaders. Most importantly, the use of high resolution 3D models must be very limited. I tried applying these constraints. Few of them actually worked, and the reasons had to do with the process of working in new media.

The process of generative design does not begin with formal questions, but with the observation of phenomena [3]. Because it feels symbiotic with digital music, generative art[3] is produced procedurally (mathematically), and it does not require memory-expensive models; thus, it seemed like a good place to begin. In new InterMedia (digital music + 3D gaming environments), as in Generative design, it is nearly impossible to conceptualize before observing what is possible. This was exactly my experience as I began building limited visual objects in Unity3D and writing JavaScript code to enhance Unity's own audio implementations. It was clear that conforming this process to a few vague conceptual statements, thesis questions, or preliminary constraints was going to be very difficult.

---

[1] This is called *The Kuleshov Effect* named after Lev Vladimirovich Kuleshov (Russian 1899-1970), perhaps the very first film theorist.

[2] WebGL (Web Graphics Library) is a JavaScript API for rendering interactive 3D graphics and 2D graphics within any compatible web browser without the use of plug-ins.WebGL is integrated completely into all the web standards of the browser allowing GPU accelerated usage of physics and image processing and effects as part of the web page canvas. WebGL elements can be mixed with other HTML elements and composited with other parts of the page or page background.WebGL programs consist of control code written in JavaScript and shader code that is executed on a computer's Graphics Processing Unit (GPU). WebGL is designed and maintained by the non-profit Khronos Group, http://www.khronos.org/webgl/.

WebGL is a cross-platform, royalty-free web standard for a low-level 3D graphics API based on OpenGL ES 2.0, exposed through the HTML5 Canvas element as Document Object Model interfaces. Developers familiar with OpenGL ES 2.0 will recognize WebGL as a Shader-based API using GLSL, with constructs that are semantically similar to those of the underlying OpenGL ES 2.0 API. It stays very close to the OpenGL ES 2.0 specification, with some concessions made for what developers expect out of memory-managed languages such as JavaScript.

Examples of my target delivery platform can be found at http://www.chromeexperiments.com/webgl/ (accessed April 26, 2014).

[3] Generative art refers to art created with autonomous systems, but it is not strictly computer generated. The fugues of J.S. Bach, algorithmic compositions of Iannis Xenakis, and the graphite pencil wall drawings of Sol LeWitt are all considered generative art. Contemporary generative design is primarily created with software tools such as Processing, Quartz Composer, and Jitter.

Before moving to remarks about the survey I conducted and details of my derived process, a few words about Artificial Intelligence (the study and design of intelligent machines) and its relationship to interactivity are in order.

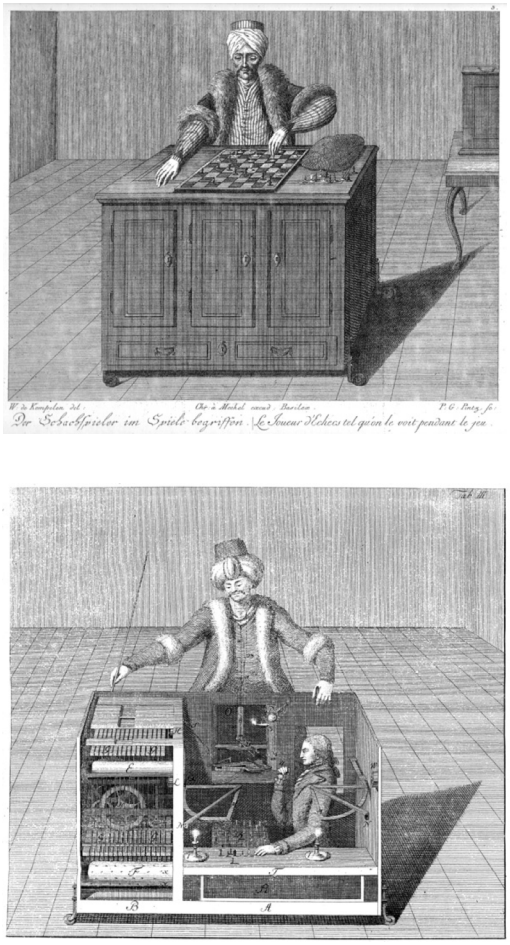


Figure 1. The chess playing Mechanical Turk, designed by Austrian Wolfgang von Kempelenin, 1770 audience and cutaway views.

The Mechanical Turk (Fig. 1 above) both fascinated and deceived audiences for nearly eighty-four years until it was destroyed in a fire in 1854. It survives as a metaphor for Industrial Era human-machine symbiosis, twentieth-century cybernetics, and today's fascination for the intelligent 'other' embedded in contemporary game play.

Criteria for judging the entertainment value of a computer game is not complex. Players often judge a game as 'fun' or 'boring' on one simple factor: Has the game made the player feel empowered through gaining knowledge of its system—knowledge of the 'other' which is present inside the game? Because of this I felt that any purely formalist approach—for example, one where an audio process would mimic or track a Generative Art process—would not provide a sufficient sense of the embedded 'other,' the hidden *persona*.

It has become a convention of game authors to describe their game objects in literary terms. For example, a game player in a first-person game assumes the role of a first-person shooter, or FPS. A game's long term success is often equal to the strength of its narrative structure, the logic of its cause-and-effect relationships, and its lack of circumstantial error. Thanks to the work of music theorists and philosophers such as Leonard Meyer [19], we know that *meaning* in music is derived from these very same concerns.

Human beings tend to create meaning by creating narrative: when we apprehend the unknown or 'the other' in our environment, we tend to explain its presence by creating story-forms. These forms are not merely the province of psychologists. Within these story forms cause-and-effects are fulfilled, new sets of expectations are generated, and we experience the simple joy of just being surprised. Meyer speaks of this at length. The creation of cause-and-effect, antecedent-consequent relationship, and frustration of expectation, are controllable processes for a composer working in linear time. But for the author of music in an interactive visual landscape where the *listener* is in control, there are unique problems.

A computer program is almost completely composed of such cause-and-effect and antecedent-consequent relationships. Look at the jargon: **`if-then-else`**, **`switch-case`**, (the ternary statement) **`message = health > 0 ? "Player is Alive" : "Player is Dead"`**; and my personal favorite: **`if "Player is Dead" = true, print ("You are Dead! Continue?")`**. The logic for creating a complex interactive narrative was available, but all I needed was a sparse visual structure on which to hang musical events: a game-space matrix of objects which could imply a subtle narrative, collect meaning to themselves in a poetic and metaphoric way, and provide the missing affect. To repeat Claire Bishop: "How many artists who use digital technology actually confront the question of what it means to …filter affect through the digital?" I also wanted to keep Bishop's challenge in mind.

So, in the manner of the *dérive,* I searched online sources such as the Unity 3D web site, which provides links to free and paid textures, models, packages, etc., hoping to find thematic content [28]. There were many comic possibilities including animated sperm which could search my game space, futurist buildings, and hundreds of Medieval environments with hack-n-slash action figures. Then I discovered a three-dimensional neuron complete with nucleus, soma, dendrites, axon, and Myelin sheaths. After purchase ($60) I had to adapt the interior of the dendrites to be passageways into the soma or strings for pearls in a net.

For years I have been fascinated by the Hindu concept called The Net of Prince Indra or the Pearls of Indra in which each surface reflects all the other surfaces so that any change in one is reflected on all the others.

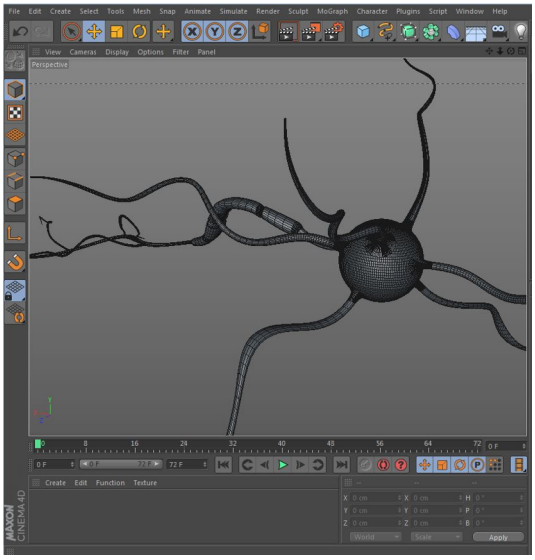


Figure 2. My 3D mesh model of a neuron in the Unity Editor.

I saw my 3D model neuron as a single node in Indra's jeweled net. I could multiply that single node into a matrix—a visual realization of an ancient philosophic concept but also a metaphor for our overly-mediated and networked contemporary lives. I also saw the potential for non-tempered-tuned ethnic orchestration. But what would float inside the jewel-nuclei in the matrix?

It seemed obvious: our comically failed expressions of the Divine: a descending *Deus Ex Machina* in an ancient Greek theater; Bentham's Panopticon filled with hundreds of disembodied eyeballs as an expression of omniscience (and a perfect metaphor for the encroaching NSA)[4]; a quietly meditating Shiva with a potentially lethal constrictive snake around his neck; a spherical space with inner surfaces showing nothing but self-image—those places we find 'God.' Importantly, there was a potential for humor.

Now that I had a potentially beautiful visual world linked to a compelling concept, I thought the real work of music composition could begin. I altered the mesh model in a companion program called Cinema4D [18] to create portals at the end of the dendrites. The listener could now travel up through the dendrites and into the nucleus to encounter the miniature narrative scene inside.

For an object to travel and to have convincing physical behavior, it must accelerate correctly and be affected by collisions, gravity, wind, and other forces. Unity3D, a prototyping system widely used by independent game developers, has a built-in physics 'engine'[5] that provides for this behavior. I knew that I would be using Unity3D's physics to start, stop, and scale musical processes, but I had severely underestimated the difficulty of programming techniques for navigation, pathfinding, and behavior—the very things which create the sense of an intelligent 'other' embedded in the game. This 'other' is my listener's only antagonist.

Navigation controls rhythmic structure. Behavior such as flocking-following and attraction-avoidance can also be used to control musical elements. In Unity3D's Pathfinding, an object is assigned a target transform (a position in 3D space) and is given the logic to move to it. Combining Unity3D's Navigation, Behavior, and Pathfinding, with triggers, collisions, and external forces, creates a rich set of tools for controlling music.

But using these tools on a navigable 2D surface proved insufficient (and boring) so I decided to design a gravity-free 3D world which could be navigated as in Blizzard Entertainment's *World of Warcraft*.[6] Then, however, things became too free. With no constraining corridors of floor-walls-ceiling, the movement was too improvisatory and structurally flaccid. I needed to take some control away from the listener and move the listener through the game automatically, while providing composed and synchronized music; but in doing so I would be inserting authorship and destroying the *dérive*.

At these junctures in my process the thesis challenge had to be re-considered. Ericka Fischer-Lichte [11] in referring to John Cage's '*untitled event*' at Black Mountain College explained that:

The spectators did not need to search for *given meanings* or struggle to decipher possible messages formulated in the performance...Thus looking on was redefined as an activity, a doing, according to their *particular patterns of perception*, their associations and memories as well as on the discourses in which they participated. (italics mine)

By removing some control from the listener I would be 'giving meaning' and disallowing the listener's 'particular patterns of perception.' However, this would provide balance—as in the 'limited aleatorism' of Witold Lutosławski's late music [26]—and control of the musical architecture. Fancy words for computer game music? No. Film-goers and game-players routinely listen to, and are affected by, complex contemporary music they would otherwise never tolerate.

One of the challenges and excitements of programming *Knots* was wanting to create an ability to swap out entire (intolerable) musical genres without having to alter the programmatic structure. In other words, if a sound-set proved too esoteric, could it be replaced with something commercial? But generating new visual elements procedurally with each new runtime was beyond my capabilities. My long-term aspirations for the project, however, include linking procedurally generated visual-audio sets in future iterations of the work.

1.2 Technical Problems Begin

My problems soon became quite technical. How to remove control from the listener programmatically, constrain it to a path, then restore control so the listener was free to explore again? The solution lies in using SplinePath Controllers. SplinePath Controllers are computationally cheap and can constrain a game object to a pre-determined path—a type of roller coaster ride with programmable pause-stop-continue nodes along the journey. Some spline paths are bi-directional and looping. They range from simple startPoint-endPoint movement to lengthy controllers which give the illusion of complex behavior such as 'pausing to look

[4] Michel Foucault in his essay *Discipline and Punish* (1975) used the Panopticon as a metaphor for modern "disciplinary" societies.

[5] An engine is a collection (or 'library') of separate modules of code.

[6] Blizzard Entertainment first published *World of Warcraft* on September 2, 2001.

around.' Spline paths are not bound to 2D surfaces such as a floors or terrain meshes, they can be sequenced to create long variable-speed control paths, and they are responsive because a grid does not have to be pre-calculated for heuristics, a consideration that will be discussed a bit later on in this paper. The familiar 'fly through' is an example of 3D spline control. If used extensively, a spline-controlled game is called an 'On-Rails' game.

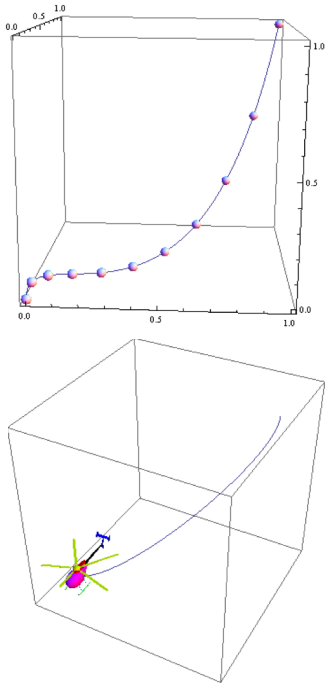


Figure 3. Single splines in 3D space.

Because the programmer can predetermine the duration of the 'ride,' and make the movement through the nodes metronomic, an accurately constructed spline can create rhythmically accurate musical events. In other words, if equidistantly distributed along a path, audio sources or triggers can create regular pulses whose tempo is dependent on the programmable duration of the ride. `Time.deltaTime`, a Unity 3D library API call, is one such method for smoothing physics movements in animation. It creates interpolations which are independent of the often-erratic frame rate that one sees in real-time rendering [26]. Additionally,

Figure 4. A Voronoi 3D spline.

variations in the proximity to the nodes can control dynamics, timbre, or various other musical parameters.

Imagine the nodes seen in Fig. 5 (below) as equidistantly placed triggers for a five-voiced musical event that recedes from the listener while panning to the right. In the graph seen here it is helpful to think of the x (horizontal) axis as time, which increases as one moves toward the right; the y (vertical) axis as the down beat, and to imagine the presence of z axis that mimics the

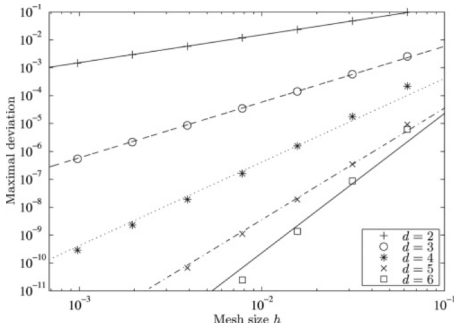


Figure 5. A five-voiced musical event in 3D space.

perspective of the solid, dashed, and dotted lines seen in the plotted space.

These types of musical-visual (spatial-acoustic) structures are easy to build in Unity3D, and can be exploited for their musical effects within Unity's multi-channel audio. If used subtlety these techniques can be musically *expressive*, not merely companions for cinematic experiences.

SplinePath Controllers seemed a perfect solution. Unity has a built-in spline component but it was too simple for my purposes. Through Unity's User Forum I found and installed a freeware Hermite Spline controller that would function better for my purposes.[7]

I created and positioned all of the spline nodes and their associated musical functionality in my single 3D neuron. I was eager to multiply my single 3D neuron into a visual matrix. But first I had to learn how to create a Unity Prefab—a reusable collection of game objects that shared functionality—or live with the consequences of having to add code to individual objects scattered throughout the entire game world that I would be creating.

One beauty of Object Oriented Programming[8] is the ability to create a reusable class from a complex object—in this case a reusable Unity Prefab from my 3D neuron mesh, that included its surface qualities, as well as all of the positions, rotations, behaviors, musical triggers, and other properties that each instance would need in order to perform properly. My modest skills with JavaScript were insufficient. I was forced to learn C# programming syntax as I worked through these problems.

Unfortunately, the free Hermite controller had no ability to vary the speed (tempo) between nodes, pause for a variable time on a node (caesura), 'ease out' (accelerando), brake or 'ease' when

[7] Hermite here means an orthogonal polynomial sequence. It takes its name from the mathematician Charles Hermite (1864).

[8] Any discussion of the vast topic of OOP is beyond the scope of this paper.

approaching a node (ritard), or 'look at' a given transform point in 3D space (dynamic L/R panning). With enough time, perhaps, I could have programed this functionality, but I suspected that someone had already done it for me.

Also, based on splines, camera path animators are generally used to create Cut-Scenes or demonstration movies of games. User's themselves often create their own lengthy Cut-Scenes within interactive narrative games and then post them online as personalized solutions to storylines. A camera path animator would also have the potential to record a (musical) journey through my project. The only commercial plugin for Unity that had all the functionality I needed (variable tempo, caesuras, ritards, accelerandos, and panning) was version 3.0 of *CameraPath Animator* written by Jasper Stocker [25]. In combination with proximity sensing[9] and Unity's physics engine, this plugin had real musical potential.

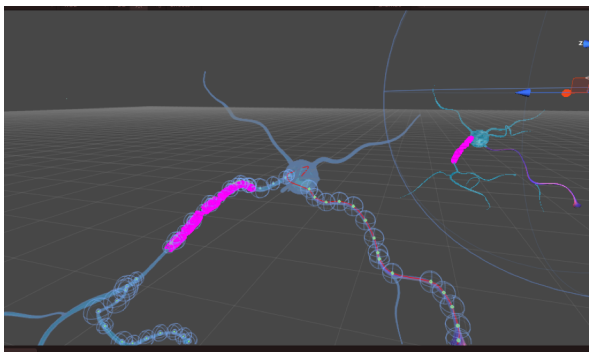


Figure 6. Two instances of a complex prefab with my many spline nodes.

But the required version (3.0) of Stocker's *CameraPath Animator* plugin required that I upgrade the host Unity from 4.2.1 to 4.3.4. I expected everything to blow up, and it did. The other existing plugins, including the principle MasterAudio, had to be updated and reinstalled. In addition, the spline nodes that I had meticulously created and positioned could not be 'dropped onto' my new CameraPath Animator. Thus, I was forced to recreate them. Once that was finished I moved to the problems of Navigation PathFinding.

1.3 Navigation Pathfinding

Pathfinding (or WayFinding) is fundamental to interactivity. At its most sophisticated it resembles Artificial Intelligence (the study and design of intelligent machines) and at its simplest it resembles decisions about movement on a 2D grid such as the Mechanical Turk's chess board. It is no use to develop complex systems for high-level decision-making if an agent cannot find its way around a set of obstacles to implement that decision. On the other hand, if an artificially intelligent agent can understand how to move around obstacles in the virtual world, even simple decision-making structures can look impressive—giving a sense of the embedded 'other,' the listener's antagonist.

1.3.1 Heuristics: The Game World as a 2D Grid

Pathfinding uses a small subset of mathematical game theory called heuristics. In computer science, heuristics is a method of trading optimality, completeness, and precision for speed. In its more developed form, the method involves self-educating processes.

Dijkstra's Algorithm for Single-Source Shortest Paths, conceived by Dutch computer scientist Edsger Dijkstra in 1956 and published in 1959, is a graph-search algorithm that solves the single-source shortest path problem for a graph with non-negative edge path costs, producing a shortest path tree. The Bellman-Ford and Floyd-Warshall methods from the late 1950s and early 1960s, respectively, are other weighted graph algorithms. More recent pathfinding algorithms include A* (pronounced "A star"). As Harika Reddy explains:

> As A* traverses a graph, it follows a path of the lowest known cost, keeping a sorted priority queue of alternate path segments along the way. If, at any point, a segment of the path being traversed has a higher cost than another encountered path segment, it abandons the higher-cost path segment and traverses the lower-cost path segment instead. This process continues until the goal is reached [23].

The drawback of A* is the lag time at the start as A* calculates the entire grid. Adaptive A* and D* are classes of techniques developed to overcome this [15]. The game's author assigns areas of the game a 'weight' or a 'cost.' In simpler game-play terms, a 'weighted' area of a grid could be an elevated terrain, a soggy bog, or a dark forest. The algorithm then calculates the least costly Heuristic path.

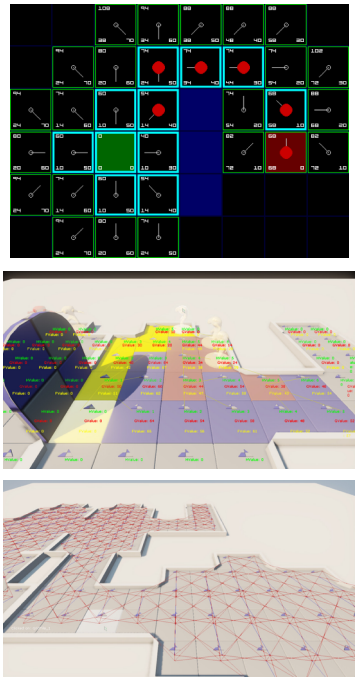


Figure 7. Two value-weighted grid tilings (top and middle) and Grid Nodes for A* (bottom).

As game author, I could link high-cost areas of the grid to the more dissonant or irrational musical elements; I could 'map' a hierarchy of navigation 'costs' onto a hierarchy of harmonic dissonance; thus, concrete visual elements could be mapped onto areas of musical, human affect. In a sense, these 'mapping'

[9] Unity implements a type of proximity sensing in their DopplerShift component on AudioSources.

techniques satisfy Claire Bishop's challenge to "filter affect through the digital."

1.3.2 The Game World as a Dynamic Grid: 2D to 3D Coordinate Space

Computer scientists have researched the problems of a dynamically changing grid for 20 years. For simple recalculations, there is D*, which dates from 1994; Focused D*, which dates from 1995; and Anytime D*, which dates from 2005. Techniques for non-grid pathfinding at any angle include Field D*, which dates from 2007; Theta*, which dates from 2007; and Incremental Phi, which dates from 2009. Today's advanced Adaptive A*, for targeting points that move, is based on research from 2008-2011 [22]. The research continues to evolve. For my purposes, a PathFinding *system* is an interrelated set of scripts which use the arrays of nodes created by these various generators to calculate paths to a variable target position in 3D space.

The Unity3D Asset store has offered several versions of A* and Adaptive A* pathfinding systems for years; however, in August 2013 Aron Granberg published his A* Pathfinding editor-plugin for game authoring. The Unity3D version is now at v3.4. On his blog Granberg [14] discusses esoterica such as Core Graphs, Funnel Corridors, Tiled Recast Graphs, methods for Navmesh Cutting, and Delauney Criteria, which are all built into his software products. Any independent game author with $100 now has access to advanced computer game theory navigation techniques.

Although I had decided that I wanted my users to primarily navigate in a gravity-free world, I wanted small areas inside the nucleus to contain a gravity-bound 2D planar surface. The objects and their behaviors on that surface—the objects which would 'metaphorically imply a subtle narrative'—could then take on a quotidian, real quality. To provide that sort of functionality I needed to implement 2D graph system navigation.

1.4 Types of 2D Graphs in Unity

A graph system such as Granberg's is considered a separate system from the Unity engine system itself. It shares assumptions about how the game is set up, definitions such as Vector3 transforms and other basic structures in the game, but it programmatically (and visually) resembles an overlay. The two systems are linked through a game object referred to as an AI. Scripts describing 'intelligent' behavior are attached to an AI. The most common AI is a character controller protagonist—to keep with my narrative interpretation—but an AI can be any non-static game object. Programmable 2D path finding requires placing the AI agent on a graph. The three simple graph types are: PointGraphs, Grid Graph, and NavMesh Graph, which is short for 'navigation mesh.' A fourth and more complex type, a RecastGraph, is a grid generator based on voxelization—a type of rasterization which generates cubes resembling pixels in a 3D space [13, 15].

1.4.1 PointGraph Graphs (Way Point Navigation)

Similar to simple spline controllers discussed earlier, a PointGraph consists of lists of interconnected points in space, waypoints, or nodes. It can be viewed as intersecting lists of nodes. The diagram below (Fig. 8) is the best explanation.

The musical implications are clear: if the listener is forced to reverse direction in order to reach a tangential path (one of the intersecting lists of nodes), then the musical events assigned to the sequences of nodes must all sound logical in retrograde. In the illustration, the node which is shared by two lists could contain musical material common to both pitch sets. As a 'hinge' it could effect a key change: As in a film score by Danny Elfman the listener could chase along in E minor, reach the 'hinge' node/pitch (a unison B natural), then charge off in another direction (in G sharp minor); or in a clichéd electronic music genre, one granulation technique could then be swapped for another.

This is clever but what about compositional issues such as phrases that are meant to complement each other or create a dialogue

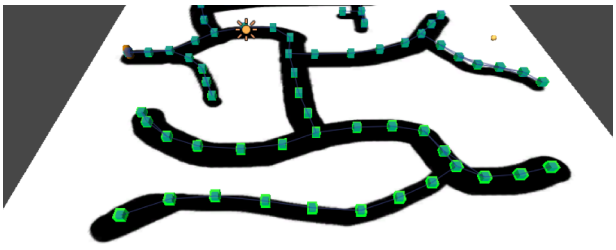


Figure 8. PointGraph—a more complex SplineController.

effect? Imagine the player as a conductor moving through the space and cueing musical events—two phrases: antecedent and consequent:

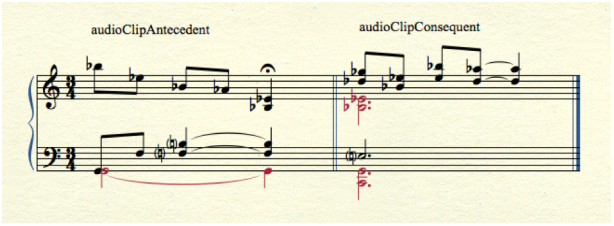


Figure 9. Two audio clips as antecedent-consequent.

The player-conductor 'cues' the antecedent or consequent music by colliding with an object containing either of the following two scripts:

```
public AudioClip audioClipAntecedent;
    void OnCollisionEnter() {
      audio.PlayOneShot(audioClipAntecedent, 0.7F);
    }
```

Figure 10. Code snippet for antecedent.

```
public AudioClip audioClipConsequent;
    void OnCollisionEnter() {
      audio.PlayOneShot(audioClipConsequent, 0.7F);
    }[10]
```

Figure 11. Code snippet for consequent.

Referring to Figures 10 and 11, the argument to **`PlayOneShot`** is the name of the actual **`AudioClip`** which will play; the value 0.7F, which follows the argument, scales the object's **`AudioSource`** volume by 70%.

Placing simplistic antecedent/consequent musical events along a unidirectional spline resembles composing on a traditional, variable-tempo timeline. But how could I effect retrograde inversions or other valuable compositional methods? Spline

[10] You can see that Unity's naming conventions in C# make function(al) blocks of code like this extremely legible.

controls offered automation but not solutions for complex musical problems.

1.4.2 GridGraph Graphs

A GridGraph consists of a collection of nodes arranged in a grid-like pattern. Given a 2D plane (floor) populated with obstacles (cubes), a GridGraph generator casts a ray down from above and returns a floor grid which excludes the obstacles. Height testing during the raycast determines how high an obstacle must be before it is excluded from the GridGraph. GridGraphs are efficient and can change dynamically—for instance if an obstacle is moved or destroyed. An obstacle can be any game object not just a geometric structure, and for my purposes that object can be a sound source.

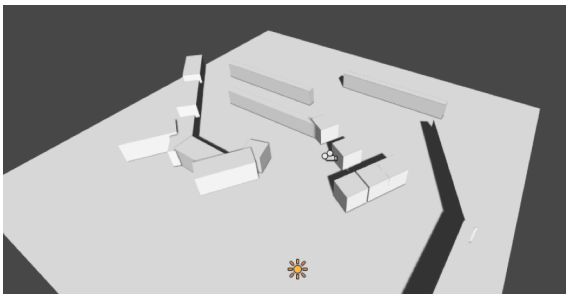


Figure 12. A simple plane with obstacles.

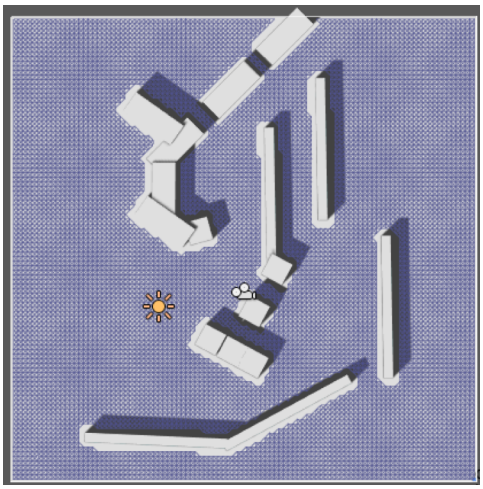


Figure 13. After RayCasting—HeightScanning

Re-scanning the grid each time a dynamic (non-kinematic) obstacle moves is computationally expensive with a potential to slow down frame rate [29]. Variations in frame rate jeopardize musical rhythmic values, so **`MonoBehaviour.FixedUpdate`** and physics timings must be used instead of **`MonoBehavior.Update`** frame rates.[11]

[11] The **`MonoBehaviour.FixedUpdate()`** function is called on every fixed framerate frame, if the MonoBehaviour is enabled. **`FixedUpdate`** should be used instead of **`Update`** when dealing with Rigidbody. For example, when adding a force to a rigidbody, one must apply the force at every fixed frame inside **`FixedUpdate`** instead of every frame inside **`Update`**.

What are the implications of GridGraph pathfinding for interactive music? One example: Maneuvering or automating an object's movement across an evenly spaced 2D GridGraph at a steady speed implies any music with tight rhythmic structures such as Baroque instrumental music, Balinese Gamelan, or minimalist pattern music. Maneuvering several objects across 2D grid nodes which trigger short percussion sounds with interlocking patterns has the potential to create hocketing, poly-rhythmic patterns. Swapping out those sounds for balafon, berimba, inanga, kora, djembe, and sanza has the potential to create an African musical landscape that is 'played' by passing through it. Maneuvering those objects at slightly different speeds (phase) has the potential to create the phasing effects as in Steve Reich's African inspired *Drumming* (1970/71). I will discuss this and the creation of an African landscape in a later section.

1.4.3 NavMesh Graphs

A NavMesh represents the 2D plane (floor) using polygons to accurately describe a surface that an AI object can 'walk' on. These polygons can be modelled by hand or automatically generated. Given a mesh (pre-drawn to exclude obstacles), a NavMesh generator returns a (random looking) array of nodes centered within the triangulated polygons.

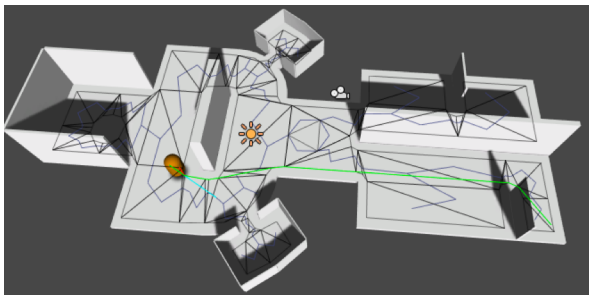


Figure 14. Auto-Generated NavMesh (as a Georges Braque cubist painting).

What are the musical implications? If the polygons on the surfaces in the illustration above are modelled by the game author-composer, then navigable paths and consequent musical events can be more tightly controlled by the author-composer. For instance, in Fig.14 (above), a node which is shared by adjacent polygons could serve as a cadential point with an ambiguous resolution. Placing a chord such as a German-Sixth—which can resolve in multiple directions—on that node would offer 'pathways' to new harmonic areas.

1.4.4 RecastGraphs

A RecastGraph is a type of generator which takes all existing graphs and returns a complex graph which describes things such as walkable height, walkable climb, max slope, edge length (of generated polygons), and boundaries [12]. A RecastGraph offers limitation for audio dynamics, compression-limiters, dissonance, and many other musical parameters.

1.5 Pathfinding System and Music

Adding visual elements to a purely musical experience is risky because the visual elements will often subsume the music; our primary sense, sight, will often conflict with the auditory; the temporal structure of the experience will be dictated by the visual element. Why build an entire 3D world in which the visual experience subsumes the musical experience? Why not distribute triggerable, invisible, sound sources in a blank, non-visual field and create an experience which is more purely musical? I think

the answer derives from an understanding of a composer's efficacy in our 2014 mediated world: the audience expects a rich and mediated (or inter-media) experience. Claire Bishop points to answers to these questions of mediation and efficacy when she asks, "what does it mean to think, see, and filter affect through the digital?" In this context graph systems seemed superficial; they could not hint at an embedded intelligent 'other'—the only (and requisite) antagonist in *Knots*. What was missing was autonomous behavior. To create game decision-making and intelligent looking movement, it was necessary to add behavioral scripts to the AI wayfinding agent.

1.6 Adding Behavior and Other Complexities

If a flock of identical AI objects is given pathfinding scripts with an identical target value, the behavior is called Flocking-Following. If all those same objects have a **`lookAt`** script which forces them to look at the same target, plus a physics script which causes them to react to collisions with other members of their flock, the flock's behavior starts to look intelligent. But what does this mean for interactive music?

I asked this same question precisely at this point in the development process. How can I derive musical uses from Flocking-Following? How can I synchronize music to this visual functionality? The creative work, the work that kept pulling me forward, was finding answers for this type of recurring question. The answer often came after formulating a complex **`if-and-then-else`** statement.

Here's a simple example: **`If`** each object in a Flock is a freely moving agent which implements collision physics, **`and`** each object has a soundSource (dissimilar to all the others) which plays on collision, **`then`** the sequence created by their collisions is in effect a random pitch and rhythm generator. It follows that: altering the speed of the objects' movement, the size of their constraining space, and the relationship of all their pitches, resembles altering distribution and seeding in pseudo-random-number generation. In addition, Unity can map collision-force to pitch-shift. Taken together, all these techniques begin to resemble a playable musical instrument, a digital synthesis engine, embedded in game prototyping software—one of the goals of this project.

This process—prose translated to logical statement translated to actual C# code—is a common method for constructing algorithms. I only qualify as an 'enhanced-cut-and-paste' programmer but the pedagogical value was clear: C# scripts—components of Object Oriented Programming—can quickly and correctly be comprehended as *behavior* in Unity's 3D visual world. This process is also future oriented and proves a common pedagogical maxim: I can participate in the aesthetic, the search for beauty, only if I improve my programming skills. I was eager for more.

1.7 Programming Languages in Unity

Unity3D supports three languages, C#, JavaScript, and Boo. Programmers prefer C# because it is more strictly typed, its variable and method scope [20] must always be declared, it implements namespaces, and (although it is more verbose) it is faster and safer. I began this project with modest JavaScript skills but was forced to learn the C# syntax. Happily the two languages are very similar.

Of the hundreds of Unity3D online tutorials, very few were dedicated to audio: "when your projectile strikes," "enter the forest zone and hear birds", "synchronize your character's footsteps," and others. One of Unity's own website tutorials showed an otherwise brilliant instructor stumbling through a presentation on **`audioSource`** while confessing he didn't understand waveforms. However, with online C# programming tutorials I learned to write scripts as components of audioSource-objects, use physics library collisions, and even write esoteric methods such as **`gameObject[dot]GetComponent (ScriptName)`** which addresses and triggers functions within scripts on any object in the game regardless of its proximity, physics collisions, or triggers.

In the illustration below, the small circles represent nodes on splineControls; each circle represents an audio collision zone. If I were to multiply this single (base) neuron into a matrix—the net of Indra's Pearls—my audio functionality would be distributed chaotically all over the gameSpace and I would never be able to locate and manage the audio programming.

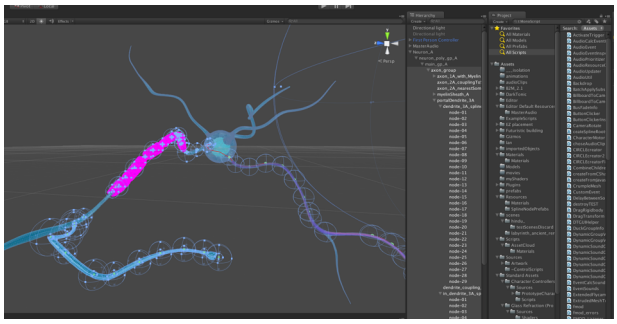


Figure 15. Unity 3D editor. On the left: the Scene View showing spline controls with nodes.

1.8 Choosing an AudioEditorInterface: Master Audio by Brian Hunsaker

From the beginning I was aware I might need an audio editor interface plugin. AudioEditors such as FMod and WWise are used in the industry by sound designer-composers who prepare proprietary interface files for use by other high-level editors. Sound designer-composers usually cede control of their designs and music to these other editors in the later stages of game development. WWise and FMod would not give me access to methods for audio design at the script-component level. Additionally, FMod and WWise did not give me access to Unity's own Echo, Distortion, High/Low Pass, or Reverb filters. Unity's ReverbFilter alone has nearly twenty programmably accessible parameters such as first reflection time. FMod and WWise hid too much behind their often-confusing graphical user interfaces.

I first looked to Ivica Bukvic at DSIS and CMAT (Center for New Music and Audio Technologies at Berkeley) for any work on a Unity 3D audio editor interface. I found toolkits such as Bukvic's MYU for interfacing Max/MSP/Jitter to Unity through TCP network packets and other interesting interfaces but no audio editor interface. Even if I had found one, I could not use it. Using Max/MSP/Jitter or PureData would require users of my project to install and maintain esoteric software that could not be ported to webGL, violating one of my first constraints: accessibility.

As could be expected in a 64-billion-dollar industry, there are a lot of gaming assets for sale. I learned early on not to trust everything. After purchasing an animated day-night skybox ($5), I found the original resource for free on the Unity User Forum. The publisher, O4karitO, had merely repackaged something a programmer had offered to the community for free. There were

software packages using the buzz word 'procedural audio' that offered the functionality of a vintage Arp2600[12] complete with Max-style patchcord editing, SFX players that offer randomization of pitch and volume [8], and other useful plugins available, but nothing with the complexity I needed.

Brian Hunsaker is a musician-programmer and the Technical Director of Dark Tonic Games Studio. His MasterAudio interface was my solution. It is an audio editor plugin which pools audio resources and their controls into one component in the game hierarchy. Its GUI is clean and intuitive. His Synkro feature offers synchronous multitrack audio with dynamic cross-fading, meaning a complete multi-voiced score could be started, individual tracks could be featured, and then cross-faded to other tracks. This is not unique to Master Audio. It is a common technique in WWise and FMod for footsteps on gravel changing to footsteps on concrete, or the changing angle of a snowboard on hardpack snow. Brian Hunsaker's implementations and the structure of his GUI seemed tailored for music, plus it maintained the availability of all the non-synced techniques.

How does Master Audio pool audio resources and their control into one game component? First an explanation of 2D vs 3D sounds: 2D AudioSources are normally stationary sounds such as the music score; 3D AudioSources emanate from a precise point in the 3D space. Sound effects are generally 3D.

Both 3D AudioSources and their listener can be in motion; the effect of that in two-channel audio is a simple pan; in multi-channel surround (up to 7.1), 3D sounds are always heard relative to their positions in 3D space. A familiar use is a car chase. In a scene with moving vehicles Hunsaker's Master Audio says in effect, "I have the 3D audioClip for the passing vehicle here in an audio bin, I will play it as it passes but I will play it at the Caller Location." The 'caller' is the passing vehicle itself.

Figure 16 below shows five sounds in a SoundGroupMixer. Figure 17 below shows one of Master Audio's EventSounds scripts in Unity's inspector. EventSounds is a long and complex C# script which exposes its public variables in the inspector. Note the Sound Spawn Mode at the top and the list of controls below.

There are options for various types of collisions and triggers including particle generator collisions, plus Mouse Enter and Click. If this EventSounds script is placed on the FPC (listener object moving through the game space), and if the SoundSpawnMode is set to CallerLocation, 3D sounds will always appear to be coming from the caller object even if everything is in motion.

Each spline node, seen in Figure 18 below, can have multiple scripts attached. Some can be for physics behavior and others for musical triggers. Master Audio keeps all audio bins in a single location, and keeps track of their assignments (by audio

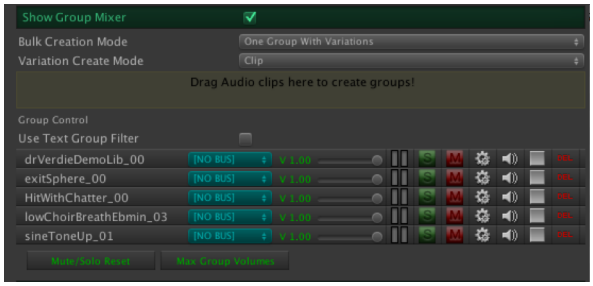


Figure 16. Master Audio's control center, showing five bins of audioClips.

Figure 17. One eventScript in Unity's inspector.

scripts) using the "tag" property—a unique name assignable to every game object. Note the tag field in Figure 17.

When the default options in Brian Hunsaker's exemplary Master Audio were insufficient, I was forced to create custom scripts—or adapt existing code.

2. CASE STUDIES

2.1 Case Study 1 Ice Caves: Meltable Terrain

2.1.1 Synchronizing Music to a Plasma Cutter Laser Beam (in B flat minor)

This is a case study of how to adapt an existing asset: in this case, an entire scene complete with FPC with all of its behaviors, a raycast laser beam, and procedurally generated (and destroyed) wall of ice.

[12] A good introduction to physics in Unity3D can be found at http://unity3d.com/learn/tutorials/modules/beginner/physics. Switching objects to a non-kinematic state relieves the CPU of constantly recalculating its position. Note: the use of the familiar word "kinematic" is a bit confusing here. See Unity3D's website tutorials at http://unity3d.com/learn/tutorials/modules and at their scripting reference, which is located at http://docs.unity3d.com/Documentation/ScriptReference/Rigidbody-isKinematic.html (accessed May 20, 2014).

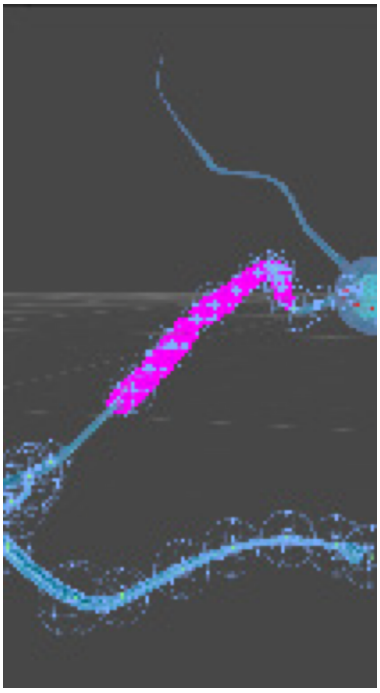


Figure 18. Spline nodes, each with event scripts.

Ice Caves is a small navigable environment without audio written by Dr. Benjamin Day Smith, Professor of Computational Design at Case Western Reserve University. It is bundled with an editor suite of C# scripts called Ruaumoko which I purchased from him through the Unity3D website AssetStore [29]. There are scripts in *Ice Caves* which procedurally generate environments which can be altered or destroyed with other deformation scripts. These deformation scripts are attached to a raycast laser beam or to mouse point-and-click operations. The gameplay requires the player to melt his way out of an ice-enclosure with a raycast laser beam—a fanciful plasma cutter. I wanted to use lines of Smith's C# code to generate interactive musical elements synchronous with the flickering blue light which appears at the impact-end of the laser.

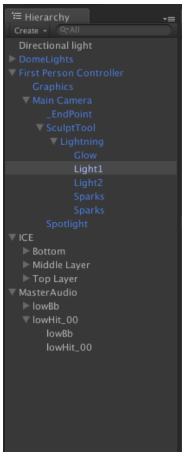


Figure 19. The Heirarchy in Unity's Editor.

2.1.2 The Process—Search for the Variable

The first problem was to find the single dynamically changing value causing the flickering. Since most code describes behavior, and the laser behavior is controlled by the player, it was a matter of 'drilling down' into the hierarchical structure of the scene and looking around. Logically, the first-person controller (FPC) fires the laser. The hierarchy of objects attached to the FPC showed two lights; disabling them confirmed that the flicker script had to be on one of these two lights creating the blue flickering effect: Light1 or Light2 (see Fig. 19, above). Unity 3D's companion Script Editor is MonoDevelop, a cross-platform IDE (integrated development environment) primarily designed for C# and other .NET languages [21]. It is a source code editor, compiler, and debugger. Attached to each of the two lights in *Ice Caves*'s FPC hierarchy was a JavaScript component named "Light_Flicker.js." Double-clicking on a script component in Unity's inspector opens it in MonoDevelop for editing, as can be seen in Figure 20, below.

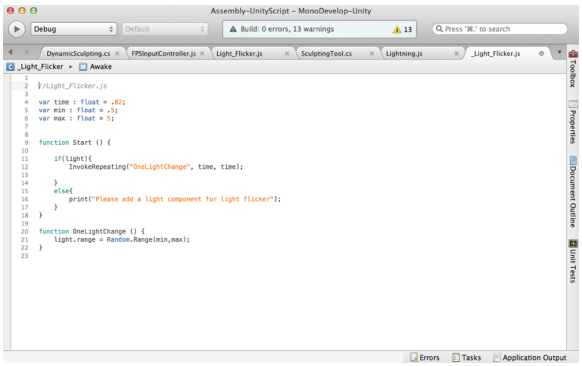


Figure 20. Light-Flicker open in MonoDevelop.

In the play mode this script generates a near-metronomic value that, for a composer, feels like 16th notes at @70 bpm—useful for music.

```
//Light_Flicker.js

var time : float = 2;
var min : float = .02;
var max : float = 1;

function Start () {
  if(light){
    InvokeRepeating("OneLightChange", time, time);
  }
  else{
    print("Please add a light component for light
flicker");
  }
}

function OneLightChange () {
  light.range = Random.Range(min,max);
}

function PlayAudio () {
  audio.PlayOneShot(soundSyncToLight);
}
```

Figure 21. Light_Flicker.js code snippet.

To translate the JavaScript code above: when the script is started, presumably by the FPC, the **Start** function calls a cycling function named **InvokeRepeating**. This in turn invokes a co-routine named **OneLightChange** repeatedly during the time the light is on. Unity3D's documentation describes a light's diameter as its range; so here **range** in **light.range** is assigned to a number which varies randomly between min and max. Meaning, the light flickers at the variable **time : float = 2**, and with each flicker the diameter is altered by **Random.Range(min,max)**.

I altered the original script by adding public variables for **PitchMin**, **PitchMax**, **SoundSyncToLight**, and **cutoffFrequency**, all bound up in the co-routines **playAudio** and **pitchShift** shown at the bottom of the script.

```
//Light_Flicker.js

var time : float = 2;
var min : float = .02;
var max : float = 1;
var pitchMin : float = 0.7;
var pitchMax : float = 1.3;
var soundSyncToLight : AudioClip;

function Start () {
  if(light){
     InvokeRepeating("OneLightChange", time, time);
     InvokeRepeating("PitchShift", time, time);
     InvokeRepeating("PlayAudio", time, time);
     }
     else{
       print("Please add a light component for light
flicker");
     }
}

function OneLightChange () {
  light.range = Random.Range(min,max);
}

function PlayAudio () {
  audio.PlayOneShot(soundSyncToLight);
}

function PitchShift () {
  audio.pitch = Random.Range(min, max);
  audio.volume = Random.Range(min, max);
}
```

Figure 22. My adapted script for Light_Flicker.js.

```
function PitchShift () {
  audio.pitch = Random.Range(min, max);
  audio.volume = Random.Range(min, max);
  GetComponent(AudioLowPassFilter).cutoffFrequency =
Random.Range(min * 1000F, max * 1000F);
}
```

Figure 23. An additional PitchShift script for Light_Flicker.js.

Figure 24, seen below, shows my adapted script on the left and Unity's inspector for Light1 on the right. My four new public variables from the top of the script are exposed in the inspector. My adapted script's new user variable **SoundSyncToLight** takes an **AudioClip**., meaning, the sound for the **Light_Flicker** can be changed by drag/dropping any audio clip that has been loaded into the project's asset folder onto the name field in the inspector. Similarly all of the script's (public) variables can be changed this way—or better yet changed dynamically with other scripts during the game play. For instance, the script's **GetComponent(AudioLowPassFilter).cutoffFrequency = Random.Range(min * 1000F, max * 1000F)** is a way of accessing the cutoff frequency of the **LowPassFilter** component attached to this same object.

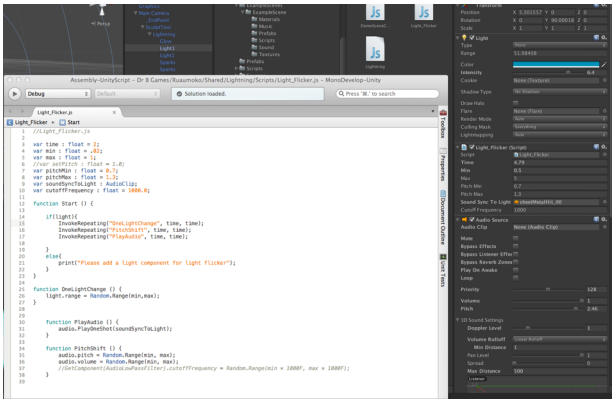
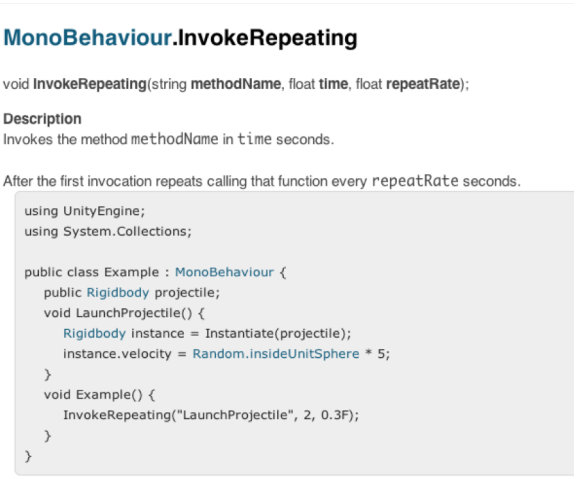


Figure 24. My adapted script for Light_Flicker.js and the corresponding game object in Unity's inspector.

The **time, time** arguments in the **InvokeRepeating("OneLightChange", time, time)** were unfamiliar to me, but I found a description and example code in Unity's application programming interface (API):



MonoBehaviour.InvokeRepeating

void **InvokeRepeating**(string **methodName**, float **time**, float **repeatRate**);

Description

Invokes the method `methodName` in `time` seconds.

After the first invocation repeats calling that function every `repeatRate` seconds.

```
using UnityEngine;
using System.Collections;

public class Example : MonoBehaviour {
   public Rigidbody projectile;
   void LaunchProjectile() {
      Rigidbody instance = Instantiate(projectile);
      instance.velocity = Random.insideUnitSphere * 5;
   }
   void Example() {
      InvokeRepeating("LaunchProjectile", 2, 0.3F);
   }
}
```

Figure 25. Unity's well-documented API with code examples.

Here one can see that the first significant argument, **time**, in which absolute time is passed, is a floating-point number, and that the second significant argument, **repeatRate**, is also a floating-point number.

2.2 Case Study 2 Audio in 3D Space

2.2.1 Using the Unity's API to Create Multi-Channel Effects

Here is a look at the scripting reference for one component (**AudioSource**) and one of its assignable behaviors called the (**PlayClipAtPoint**) method:

The Unity scripting reference in Figure 26 below is very clear: when this script is enabled, a sound emanates from a point five meters to the right, one meter above, and two meters forward of

AudioSource.PlayClipAtPoint

static function **PlayClipAtPoint**(**clip**: AudioClip, **position**: Vector3, **volume**: float = 1.0F): void;

Parameters

clip	Audio data to play.
position	Position in world space from which sound originates.
volume	Playback volume.

Description

Plays an AudioClip at a given position in world space.

This function creates an audio source but automatically disposes of it once the clip has finished playing.

```
@script RequireComponent(AudioSource)

public var clip : AudioClip; //make sure you assign an actual clip here in the inspector

function Start() {
    AudioSource.PlayClipAtPoint(clip, new Vector3 (5, 1, 2));
}
```

Figure 26. Unity's well-documented API.

the center of the game's world. How could this be used? The programmer could create a portal in space that can enable this script. If the player crosses the portal this script is enabled with the (**`Start()`** function), and the player hears a musical phrase (clip) from that specific point in space (if the clip had been designated as 3D audio).

Here is a variation: the programmer can add additional audio sources (clips 1-4) and alter their positions to form a square around the center of the world, as can be seen in Fig. 27:

```
public AudioClip clip1;
public AudioClip clip2;
public AudioClip clip3;
public AudioClip clip4;

void Start(){
  AudioSource.PlayClipAtPoint(clip1, new Vector3(-5,
1, -5));
  AudioSource.PlayClipAtPoint(clip2, new Vector3(5,
1, -5));
  AudioSource.PlayClipAtPoint(clip3, new Vector3(5,
1, 5));
  AudioSource.PlayClipAtPoint(clip4, new Vector3(-5,
1, 5));
}
```

Figure 27. Code snippet for a surround-sound effect.

If the player is at the center of the world (0,1,0) when the script is enabled, the player hears a quadraphonic surround-sound effect. If the audio clips are taken from true multi-channel, surround-sound recordings, the effect could be stunning.

I have always considered multi-channel surround sound to be an *expressive tool* in a composer's kit. It is largely misused. In the 1970's Roger Reynolds from UC San Diego announced exciting new quadrophonic re-mixes of all his previous electronic music. But listeners still sat passively and 'received his authorial presence'—from two additional directions. And what can we make of Kraftwerk's recent surround-sound, stereoptic concerts for Disney Hall's passive "*Society of the Spectacle*" audience? Guy DeBord railed against this dangerous passivity in 1967, saying: "In societies where modern conditions of production prevail, all of life presents itself as an immense accumulation of spectacles. Everything that was directly lived has moved away into a representation" [10].

Intermedia? Interactivity? Interactive Musical Composition in Game Prototyping Software? Can any of this move art praxis toward social practice—or even viable music?

2.3 Return to the Thesis Questions

Let us return to the thesis question—"Could computer game prototyping software tools be adapted to create controls for interactive digital music?"—and to the challenges given by Joseph Beuys and Claire Bishop. What conclusions has my work led me to so far? Throughout the paper I've tried to force the technical issues to interrogate the aesthetic questions; to have them 'speak up to' any social (wide-audience) uses of an interactive musical composition in a popular and free WebGL player; to have the users of *Knots* feel empowered through gaining knowledge of its system—knowledge of the intelligent 'other' which is present inside the game; and to feel they are the composers of their own musical-visual experience.

The horizon kept shifting significantly as I realized the potential of the techniques. I was discovering. It seemed that any parameter of Unity's visual world—any dynamic or static variable, any node in a pathfinding system, any physics behavior or force, any blue flashing light at the end of a laser beam, any change of lacunarity in a procedurally generated surface—could be accessed and used for musical purposes. It would take more programming skill than I possess now, but the challenge and potential is exciting.

In its present version *Knots* is a neural network resembling the Hindu concept of the Pearls of Indra—a visual realization of an ancient philosophic concept but also a metaphor for our overly-mediated and networked contemporary lives. It is both a mico-world and a cosmology.

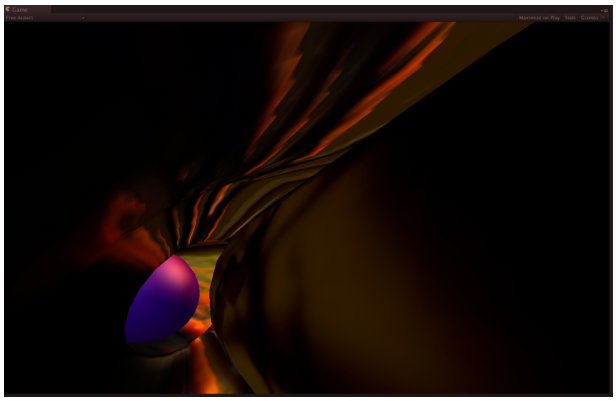


Figure 28. Inside the Myelin Sheath - Navigating to the Soma Core (screenshot)

3. REFERENCES

[1] Bishop, Claire. "Antagonism and Relational Aesthetics." *October* 110 (Fall 2004): 51-80.

[2] Bishop, Claire. "Digital Divide." *Art Forum*, September 2012.

[3] Bohnacker, Hartmut. *Generative Design*. New York: Princeton Architectural Press, 2012.

[4] Buchloh, Benjamin H. D. "An Interview with Thomas Hirschhorn." *October* 113 (Summer 2005): 77-100.

[5] Bukvic, Ivica and Kim Ji-Sun. "μMax-Unity3D interoperability toolkit." In *Proceedings of the 35th International Computer Music Conference*. San Francisco, USA: International Computer Music Association. 375-378.

[6] Bukvic, Ivica and Scott Betz "Using Gaming Engine for Virtual Prototyping and Impact Assessment of Complex

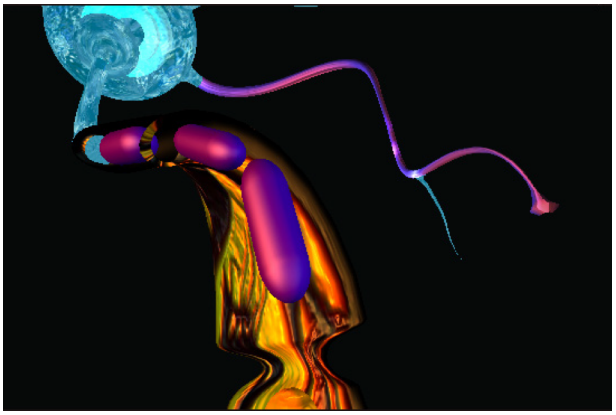


Figure 29. Approaching the Core (screenshot)

Interactive Art Installations." In *Innovation, Interaction, Imagination*. Edited by Monty Adkins and Ben Isaacs. Proceedings of the International Computer Music Association. San Francisco, USA: International Computer Music Association, 2011.

[7] Bukvic, Ivica. "Innovation, Interaction, Imagination." In *Proceedings of the International Computer Music Association*. San Francisco, USA: International Computer Music Association, 2011.

[8] Clockstone Software GmbH. "Audio Toolkit Editor Extension." Unity Asset Store. Accessed April 10, 2014. https://www.assetstore.unity3d.com/en/#!/content/2647.

[9] De Duve, Thierry. "Why Was Modernism Born in France?" Art Forum, January 2014, Vol 52, no 5. p192.

[10] Debord, Guy. "Society of the Spectacle," Black & Red, trans. (1977). 1967. Accessed April 20, 2014. https://www.marxists.org/reference/archive/debord/society.htm.

[11] Fischer-Lichte, Erika. "Performance Art and Ritual: Bodies in Performance" In *Performance: Pt. 1. Identity and the Self*. Edited by Philip Auslander. New York: Routledge, 2003.

[12] Granberg, Aron. "A* Pathfinding Project." Accessed March 14. 2014. http://arongranberg.com/astar/.

[13] Granberg, Aron. "Getting Started, Part 2." A* Pathfinding Project. Accessed April 16. 2014. http://arongranberg.com/astar/docs/getstarted2.php.

[14] Granberg, Aron. Home Page. Accessed March 15, 2014. http://www.arongranberg.com/.

[15] Hart, Peter E, Nils J. Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths." *IEEE Transactions on Systems Science and Cybernetics* 4 no. 2 (1968): 100-107.

[16] Kelsey, John. "Next-Level Spleen." *Art Forum*. September, 2012, 412-415.

[17] Knabb, Ken, ed. *Situationist International Anthology*, Berkley: Bureau of Public Secrets, 1995.

[18] MAXON Computer GmbH. "Overview." Cinema 4D. Accessed April 29, 2014. https://www.maxon.net/en/products/cinema-4d/overview/.

[19] Meyer, Leonard B. *Emotion and Meaning in Music*. Chicago: Chicago University Prerss, 1956.

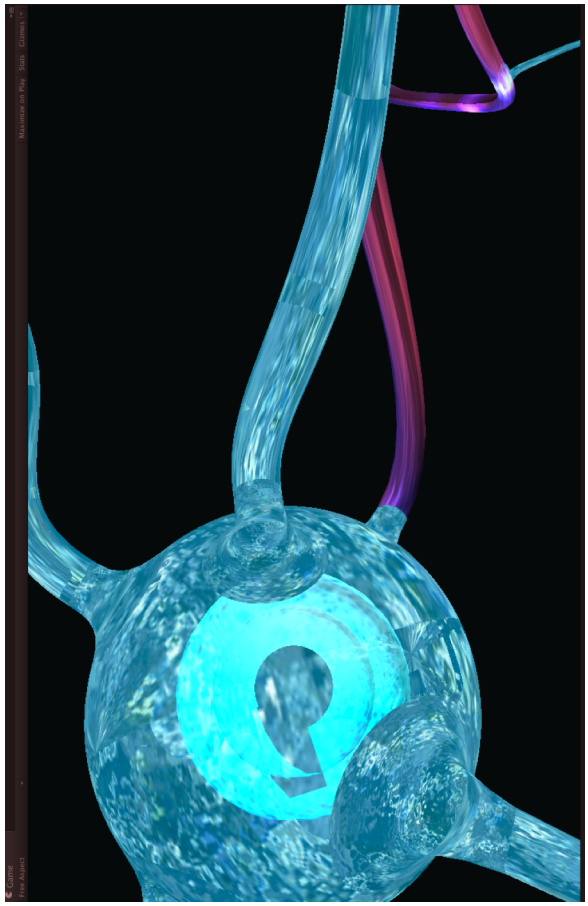


Figure 30. Peering into the Soma Core where Shiva, Choked by a Snake, Sits in Meditation (screenshot)

[20] Microsoft. "Types." C# Programming Guide. Accessed February 12, 2014. http://msdn.microsoft.com/en-us/library/ms173104.aspx.

[21] MonoDevelop Project. "MonoDevelop."Accessed April 27, 2014. http://www.monodevelop.com/

[22] Pflughoeft, Danny. "How Do the State-of-the-art Pathfinding Algorithms for Changing Graphs (D*, D*-Lite, LPA*, etc) Differ?" Computer Theory Stack Exchange. July 13, 2012. Accessed April 19, 2014. http://cstheory.stackexchange.com/questions/11855/how-do-the-state-of-the-art-pathfinding-algorithms-for-changing-graphs-d-d-l.

[23] Reddy, Harika. *PATH FINDING - Dijkstra's and A* Algorithm's*. [Self Published] (December 13, 2013). Accessed March 14, 2014. http://cs.indstate.edu/hgopireddy/algor.pdf.

[24] Smith, Benjamin D. "Ruamoko." Unity Asset Store. Accessed March 10, 2014. https://www.assetstore.unity3d.com/en/#!/content/8176

[25] Stocker, Jasper. "Camera Path Animator 3.0 - Animate in Unity with Ease." Assets and Assets Store Forum. February 10, 2014. Accessed February 23, 2014. https://forum.unity3d.com/threads/camera-path-animator-3-0-animate-in-unity-with-ease.227416/.

[26] Stucky, Steven. *Lutosławski and His Music*. Cambridge: Cambridge University Press, 1981.

[27] Unity Technologies. “Delta Time.” Tutorials. Accessed April 29, 2014. https://unity3d.com/learn/tutorials/topics/scripting/delta-time.

[28] Unity Technologies. “Game Asset Website List.” Answer Forum. Accessed April 28, 2014. http://answers.unity3d.com/questions/16650/game-asset-website-list-free-and-paid-textures-mod.html.

[29] Unity Technologies. “Rigidbody.isKinematic.” Unity (Online) Manual. Accessed March 3. 2014. http://docs.unity3d.com/Documentation/ScriptReference/Rigidbody-isKinematic.html.

[30] Wood, James. *How Fiction Works*. New York: Farrar, Straus and Giroux, 2008.