Metagaming Concepts for Analysing Techniques and Aesthetics in Bytebeat Performance: The Technology Tree, the Tier List, and the Overpowered

Prof. Jeffrey M. Morris, DMA.
Department of Performance Studies, Texas A&M University, College Station, Texas, USA
www.morrismusic.org
e-mail: morris@tamu.edu

1. Introduction

As a composer, I make music for a variety of unique situations, including site-specific work and serious concert music for toy piano, slide whistle, and Sudoku puzzles. For each work, I seek a composition technique that engages and challenges the situation at hand, e.g., turning glitches into featured elements or using data about a place to shape music that will be played there, an approach I call *native composition*. Feedback loops, intermedia translations, and intentional misapplication are common techniques in this approach, and they require an intimate understanding of the situation at hand, including any subject matter and media involved, in order to find ways to make it sing most naturally. In my technology-based performance, I have pursued bytebeat programming, which is such a heavily constrained protocol that the programmer is constantly immersed in — and must wrestle with — the most basic nature of the digital computer. It is so restrictive that it almost seems impossible to make any serious music with it, especially by any familiar and convenient techniques, and many basic achievements feel like clever hacks.

I taught a course on bytebeat programming in spring 2019 at Texas A&M University. As I prepared demonstrations, as the class discussed and explored examples, and we created solo performances and improvised together, we created tutorials, began compiling a knowledge base, and encountered many accidental discoveries about bytebeat programming, its tools, and its aesthetics. This paper is not a tutorial on bytebeat programming techniques, but certain technical concepts are explained in order to facilitate a discussion of aesthetics.

Structures from video games like the technology tree and the tier list emerged in our knowledge base as we continued to fill and organise it. Practical needs for focused assignments, improvisational prompts, and clear and fair grading led us to adopt gaming concepts like *challenge modes* and the *overpowered* in our discussions. In reflecting on the lessons of our bytebeat experiences, the *metagame* emerged as an enlightening framework for discussing the aesthetics of bytebeat and exploratory programming in general, as well as the nature of exploratory research.

2. Background

This discussion brings together concepts that are related in complex or subtle ways, and ones that seem increasingly distant from music and aesthetics. Before bringing them together to discuss techniques and aesthetics in bytebeat performance, this section presents each concept separately.

2.1 Bytebeat

Bytebeat is a computer programming practice for making music from a single line of code, often a single mathematical expression with a highly restricted set of operations. It was introduced by Ville-Matias Heikkilä (known by the screenname viznut) in 2011 [1]. It received a flurry of attention for a couple of years, fell into obscurity as a novelty, and has recently attracted more serious attention by a few scholar-artists [2], [3].

In this protocol, a bytebeat interpreter program allows a user to enter a mathematical expression that applies arithmetic and logical operations to a variable, *t*, which represents time as a constantly rising counter. The interpreter evaluates this mathematical expression inside a for-loop, sends the evaluation to the audio output, and increments *t.* It continually re-evaluates the expression and sends it to the audio output with each new value of *t*, usually around 8,000 times per second for an acceptable audio sampling rate.

For example, the following expression creates a brief looping passage known as the “Forty-Two Melody” (origin unknown). It includes multiplication (*), a bitwise AND operator (&), and a bitwise right-shift (>>):

```
t * (42 & t >> 10)
```

One popular bytebeat interpreter, which we in the class came to call by the handle Greggman, as it was created by Gregg Tavares, runs in a web browser [4]. Since many bytebeat interpreters allow users to change the input expression in real time, they can be used for live coding performances. The dense level of mathematical abstraction is what keeps the artist in an exploratory (rather than deterministic) programming mindset, especially during a live coding performance, even if a random number generator is not employed.

This practice emerged from the *demoscene*, an art- and skill-focused outgrowth of the early software *cracking* community (which means to bypass software copy protection mechanisms). Whereas early software crackers would add animations to cracked software in order to sign their work and further demonstrate their skills and style, the demoscene leaves cracking aside and focuses solely on the programmer’s ability to achieve the most creative and sophisticated results from the most compact code.

Since bytebeat code is so constrained and compact, it does not require fast or sophisticated computer power. It would have been possible (as a non-real-time practice) as early as the mid-1950s, sometime after Alan Turing (UK) and Geoff Hill (Australia; both working separately) first used computers to play melodies by varying the speed of the computer’s alert buzzer in 1951. Artist-programmers could have pursued bytebeat even before Max Mathews created the first music and sound programming language, called MUSIC, at Bell Labs in 1957. So, bytebeat is an anachronism, a branch of computer music history that was made possible in the 1950s but lay unexplored for almost 60 years, while history instead built upon Mathews’ MUSIC language and its

orchestra–score structure that became pervasive in almost every computer music development after it.

2.2 The Technology Tree

First appearing in the 1980 *Civilization* board game [5] and having expanded in the long-running *Civilization* video game series since 1991 [6], the *technology tree* (or *succession table*, as the original game called it) is a branching path of options for players to advance their abilities within a game incrementally, building upon previous choices. For example, since *Civilization*-style games mimic the historical evolution of human societies, when players have opportunities to advance their respective societies in the game, they might choose to "research" incremental options in the agriculture path, in order to develop animal husbandry abilities, which allows them to develop horseback riding skills in future turns, which in turn makes cavalries possible. Alternatively, players might choose to advance along a path that starts with mining, which leads to metalworking, which can enable better construction, weapons, or other technologies. The technology tree throttles the evolution of gameplayer abilities by forcing players to prioritise certain paths over others or to prioritise breadth over depth. It balances power while promoting diverse abilities and strategies among players, keeping the game fair and engaging. Tuur Ghys [7] gives a comparative analysis of technology trees in this type of video game, within a Game Studies context.

For applying this structure more broadly, the hybrid term *succession tree* might be more neutral and appropriate. Even *Civilization*-style games include paths to develop aspects of culture and government, but the term *technology tree* is still used the most, and it is unproblematic to use in this paper, given its subject matter.

2.3 The Tier List

Compared to the technology tree, the *tier list* deals with similar factors, but it considers them from the opposite perspective: it is a synchronic, ranked taxonomy of game characters or tools in their completed states or their current states of development, for analyzing a game in terms of balance or for a player to strategically choose game characters or tools that would perform favorably against a given challenge. Because of this application, tier lists often emerge from players' analyses of games, periodicals that review video games, and organizations that host gaming competitions. Tier lists inform *balance*—a fair fight—as weight classes were meant to do in boxing, and good balance is considered essential for satisfying gameplay.

Tier lists may be compiled by comparing individual attributes (if they are quantified and disclosed, e.g., speed, intelligence, or hit points), by considering the results of past matchups, by public opinion polls, or simply by intuition. They commonly result in grouping characters that are approximately balanced with each other, each having different particular strengths and weaknesses in relation to the others. Because video games are popular media and tier lists are heavily discussed among gamers, this concept has made its way into Internet memes reflecting on other aspects of life and culture [8], and it has proven to be a useful framework in popular media outside of gaming. This is in part because, as a tool designed to assess balance, it exposes areas that lack balance, in which one element is *overpowered* (or *OP*) in relation to another.

2.4 The Overpowered and Its Opposite

Applying a tier list framework to phenomena outside of gaming can lead to novel and valuable observations. For example, a Tier Farm video on evolutionary biology [9] states: “Sloths are the worst-ranked build in the entire game [i.e., current reality].” It goes on to describe ancient sloths, in contrast, as overpowered, and surprisingly, and it argues that the overpowered Ice Age sloths died out because they were overpowered, leaving only modern sloths surviving. It concludes that “sometimes it pays to be low-tier.”

Beyond the notion of winning, however it is defined, early game designer and theorist Chris Crawford emphasised the “illusion of winnability” [10]. Besides the obvious goal to win a game, this statement has two equally crucial and opposing components: if a game is impossible to win, a player will be unmotivated to play it; however, motivation also wanes when the game is finally won. Therefore, Crawford stipulates that a player must feel like a game is winnable and that this must remain only a feeling. Winning must remain elusive, or else the game will end. Of course, some games do end; on the other hand, there is more than one way to play some games.

Overpowered elements are considered poor game design, and using them is considered to be dishonourable, because it usually leads to predictable, uninteresting gameplay. Conversely, and beyond simply avoiding overpowered elements, it is considered especially honourable to take on special challenges in gameplay, such as using the weakest character or by completing a game with the lowest score possible (in a game that expects players to pursue high scores) [11]. These may be self-imposed, stipulated by competitive organisations, or offered in the game as special *challenge modes*. This is a kind of honour similar to that found in demoscene and bytebeat communities.

2.5 The Metagame

The point of a game, within the world of the game, is simply to win. These gaming concepts, the technology tree, tier list, the overpowered, and special challenges, all consider the game from outside the world of the game. They serve strategic gameplay and the analysis of games, and they unlock other forms of honour, beyond the simple high score. Because they are about the game but outside it, and because they also may be game-like in themselves, these concepts come together under the term *metagaming*, or the game of playing games.

3. Applying Gaming Concepts to Bytebeat

3.1 Pedagogy

A technology tree emerged in our class knowledge base as I sought to introduce new techniques incrementally and break down examples so they could be understood in terms of more fundamental principles working together. The following demonstrates possible paths of learning and applying skills by navigating a technology tree. It is not necessary to understand the technical terms introduced here, only how they build upon and work with each other.

One branch of development might start with a *noise generator*. A relatively compact and satisfying (although not entirely pure) noise generator in bytebeat is as follows. Variables, spaces, and line breaks are used to facilitate readability and discussion, but they are not necessary.

```
a = t * t % ((t % 10) +
256)
```

Having achieved a *noise generator*, one could pursue multiple development paths. For example, to make a rhythmic burst of noise, create a *sloped ramp*:

```
b = t / 1000
```

Then use the *sloped ramp* as an *amplitude envelope* (fading it out over time, restarting each time you restart the interpreter's clock):

```
a / b
```

The whole resulting program would be:

```
a = t * t % ((t % 10) +
256),

b = t / 1000,

a / b
```

Or, in its most compact form:

```
(t*t%((t%10)+256))/(t/1000
)
```

While this is not a very interesting result in itself, it is on a path toward creating something like the following code. To use terms from our technology tree, it *encapsulates* the *sloped ramp* (by applying `% 256` to it), to make a *recurring amplitude envelope*, and it replaces the slope of that *sloped ramp* with another *sloped ramp* so that the rate of the *recurring amplitude envelope* changes over time.

```
(t * t % ((t % 10) + 256))
/
((t / (( t / 1000) % 50))
% 256)
```

In an alternative path of development, one could take the original *noise generator* and combine it with a *sample-and-hold* function to create a *random number generator* with an arbitrary rate of output. Having created this *random number generator*, one might add its latest output to its previous output to create a *drunk walk.* Or, one could apply a *Boolean* test (e.g., `> 128`) to the *random number generator* and multiply the result by some other sound-generating code (a technique called a *gate*), which would create a *sieve* (which would turn the sound on and off, randomly). Explaining all of these terms is beyond the scope of this paper, but I have taken care to use the most standard and clear terms for each of these techniques as they lie along their respective paths in the technology tree.

3.2 Taxonomy

In the class, while conducting controlled demonstrations to isolate and teach about certain specific aspects of bytebeat programming, we discovered several undocumented differences between bytebeat interpreter programs. The task of an interpreter seems straightforward, all interpreters appear to be roughly equivalent regarding the basic task, and these differences may seem inconsequential. However, they resulted in significant differences in musical results and even made it impossible to reach some areas of the technology tree using a given interpreter.

For example, most bytebeat interpreters allow users to assign values to variables that can be used elsewhere in the code. However, some interpreters initialise these variables outside the for-loop. This

means they are *external variables* in relation to the bytebeat code. This allows a user's code to recall the last state of a variable so that information can persist across iterations of the for-loop. Without external variables, recursive variable assignments are impossible (e.g., `x = x + 1`), and this is necessary for many common and rewarding structures, such as the *sample-and-hold* and *drunk walk* mentioned above, as well as counters and Euclidean rhythms. BitWiz [12] is an interpreter that uses external variables. Greggman does not, although we discovered an exploit that would allow us to achieve this functionality in some cases.

So, interpreters that allow external variables lie in a different category of sophistication. On the one hand, one might argue that such *stateful* code is more impressive because it has a larger technology tree to master and coordinate; on the other hand, one might say stateless code is more respectable because of its greater limitations or because it is more pure or elegant. Either way, we realised that it is worth segmenting technology trees into tiers like these and that the knowledge of what tier a programmer is using can affect our impression of the performance.

Another tier includes interpreters that do not limit themselves to the 8-bit (range of 256) output values that are traditionally used. Greater bit depth yields higher audio quality and smoother control curves, yielding a less glitchy, noisy, and retrospective sound. Indeed, since most interpreters run on computers that can handle floating points, the tradition of dealing only with integers is nostalgic but not necessary. Further, restricting outputs to low-bit integers is not necessarily authentic to historic processors. Through accidental discoveries and controlled follow-up explorations, we discovered that some interpreters preserve floating-point values until the final output, which a computer without a floating-point processor would not be able to do. Differences in when and how values are integerised (e.g., by rounding down, up, or to the nearest integer) can significantly affect the resulting sound and the user's capability. For example, although "`/ 1024`" is considered equivalent to "`>> 10`," the following two expressions yield drastically different results in Greggman (a bass arpeggio versus a smooth full-range sweep), whereas both versions sound identical in BitWiz (bass arpeggio):

```
t * ((t >> 10) % 4)
```

and:

```
t * ((t / 1024) % 4)
```

Further, code that uses trigonometric functions stands apart from others. A sine function allows users to achieve common computer music techniques like additive synthesis and frequency modulation synthesis easily. Trig-tier code allows greater sophistication, but it is less native to the notion of bytebeat, and it is more like other platforms that would be easier to use instead.

Other classifications include (a) infix notation (as used above) versus postfix notation (as in Reverse Polish Notation), which makes certain coding techniques easier and others more difficult, especially during live coding; (b) external control inputs, such as accelerometers, cursor position, MIDI or OSC input, or even audio input; and (c) video capability, allowing the same code to create sound and animations, as with Heikkilä's *IBNIZ* [13].

Taxonomies emerged pragmatically in our class knowledge base, to segment our technology tree into levels of difficulty and to articulate the strengths and limitations of a given interpreter. However, they also

allowed us to begin to reflect on other thoughts to be had about a performance, once its tier or class is known.

3.3 Aesthetics

The technology tree concept is parallel to the pedagogical concept that made it necessary for my teaching (and for my learning): the zone of proximal development. Early twentieth-century psychologist Lev Vygotsky depicted the education process by articulating the set of things a student has mastered and the set of things the student has not mastered. The *zone of proximal development* is the liminal area, where learning objectives lie that are outside the student's area of mastery but which the student could master, with assistance (by an instructor) [14]. This process bears a resemblance to Crawford's "illusion of winning" in game design. The technology tree makes learning incremental and makes the zone of proximal development apparent, while it remains impossible to master all the possible combinations of techniques and creative ways to use them.

This concept of balance between the possible and the impossible can also be rewarding to audiences, even if they are not trained in performance. For example, watching a live, acoustic performance of Rimsky-Korsakov's "Flight of the Bumblebee" [15] is exciting, in part because of the violinist's obvious training effort and dexterity. Using a piano roll MIDI sequencer to play the same music would not achieve the same excitement, even though it could play much faster than the violinist. This is because the piano roll is overpowered in relation to the violin, in this case.

Next, consider this version of the same music:

```
t*((p=(q=t>>11)%4<3)*18+(
        (r=1-p)*2-
1)*(t>>9)%4+r*13+(q&1))
```

This version of the opening motive is titled "Byte of the Bumblebeat." (An attentive listener will notice that this is slightly different from Rimsky-Korsakov's version; this is discussed in section 4.1.)

Even though it is like the piano roll version in that it is fully automated and only requires a human to press the Play button, the bytebeat version might impress more audiences than the piano roll version because of the difficulty of the challenge. Here, speed and accuracy are not the challenges; mathematical complexity and elegance are.

It is traditional to include a visual element in laptop music performance. The Dallas-based Laptop Deathmatch series (now defunct) scored stage presence along with creativity and technique. It emphasised giving the audience something to look at, at least by using an external control interface [16]. Projecting the performer's computer display during a live coding performance has become a common solution. Even for non-programmers in the audience, seeing the code change and hearing the sound change at the same time makes the music seem accessible or graspable, even more so if raising or lowering a value results in a noticeable increase or decrease in some aspect of the sound. Even this barest understanding of the performer's technique can make a performance more engaging, when the next incremental level of sophistication appears graspable while the full range of creative possibilities feels infinite.

This might lead one to conclude that visual elements or background knowledge are necessary in order to make rewarding musical experiences; however, this is not true. Such *extramusical* factors (i.e., outside of the music) are often effective in

making performances more engaging, and they are often unavoidable — acoustic performances always require the performer to move, and those movements betray information about effort and skill. Because such elements are both effective and unavoidable, it is easy for an audience to rely upon them instead of focusing solely on the musical content of a performance. The path to purely musical enjoyment involves the zone of proximal development and a balanced "illusion of winning" as well, suggesting that some form of honour might come to an adventurous listener. However, that is beyond the scope of this paper.

Still, among practitioners, honour in gaming, e.g., by embracing challenges and not overpowered elements, is parallel to honour in demoscene and bytebeat programming, e.g., creative and elegant results despite constraints, and taxonomies help articulate those properties. One element of risk in bytebeat that is similar to the violinist's constant risk of missing a note might be to use an interpreter that does not implement an error checking process to prevent a typed syntax error ruining a live coding performance.

Beyond considering the capabilities of the bytebeat interpreter software, taxonomies could also classify various limitations on the coding techniques used. For example, it may be considered more honourable to code without using commas, which arguably break the "single line of code" definition. In class, we discovered and developed a number of techniques that lie outside the code, including clever uses of undo, redo, cut, and paste functions, line breaks, comment characters, and even physical, dextrous typing techniques we came to call *backspace flams* (replacing single characters almost instantaneously by pressing Delete or Backspace and then typing a new character, all in a quick, two-stroke gesture) and *padding* and *trimming* (quickly jumping to larger or smaller orders of magnitude by placing the cursor somewhere in a number and inserting or deleting any digit, any number of times).

3.4 Nativeness

We used the following guidelines for the purpose of grading in the class: (a) performed the full work in a bytebeat interpreter from beginning to end, limiting any post-processing to minimal cleanup; (b) only use basic arithmetic and logic operators (e.g., not the sine function); and (c) keep it "native," i.e., do not use bytebeat to achieve something that would be more appropriate to do in another platform, e.g., additive synthesis, sample playback, or sequencing; external controllers and data inputs were allowed, as long as bytebeat was not used as in a static way, as a synthesizer.

Further, while it was not prohibited, we sought to avoid falling into Mathews' familiar and pervasive orchestra–score paradigm, which divides code into signal-rate sound generators (as musical instruments) and symbolic, control-rate instructions for the orchestra to play (like sheet music). This was another guideline in pursuit of discovering and reflecting on bytebeat's native idiosyncrasies. Although it is familiar and sensible, the orchestra–score paradigm adopts the model of musics that are structured in other ways and probably would be unnative and unnecessarily awkward to realise in bytebeat. In contrast, bytebeat deals more naturally with code in which the sound-producing and sound-controlling elements are inextricable. For example, changing one character in "Byte of the Bumblebeat" can dramatically change pitch, rhythm, loudness, and timbre, all at once, whereas a single change in the score for "Flight of the Bumblebee" might well go unnoticed.

Beyond the practical need for a clear and fair grading policy, and beyond the notions

of challenge and honour (which are, after all, extramusical factors), the purely musical interest in considering classes and tiers of bytebeat tools and techniques lies in the fact that they each sound different: they yield different subspecies of bytebeat music. Since the goal of this pursuit is to understand bytebeat's idiosyncratic nature (rather than to turn bytebeat into other things), recognizing and analyzing taxonomies — including both their potential and the side-effects they introduce — facilitates understanding by helping to define purism, or different types of purism, and various deviations from it, in relation to their impacts on creative processes and products.

4. Discussion

4.1 More about "Byte of the Bumblebeat"

My approximation of Rimsky-Korsakov's "Flight of the Bumblebee" in section 3.3 differs from the original motive, in that the original version uses groups of 5–3–4–4 notes, respectively, but my version only uses constant groupings of 4–4–4–4. This is in part because of difficulty but also to facilitate discussion. Because of time constraints, I would only be able to realise the 5–3–4–4 grouping pattern by using a certain technique that would simply be overpowered for this task.

Ken Downey found a way to use a bit-shifting operator on a single, large hexadecimal number, to create a step sequencer [17]. While this is a very clever achievement, it would be similar to using a piano roll MIDI editor to play "Flight of the Bumblebee," and I remain confident there is another approach that is more appropriate to the scale of this task and of the rest of my code. Although I stopped my work on it in order to discuss this choice, in my next step, I would try to exploit the rounding artifacts of integerising the quotient of a ramp, wrapped and scaled to a range of 3–5, and offset so that it begins on 5 before wrapping around to 3. I haven't achieved this yet, so I could be wrong here, but it would be a more honourable approach to this task.

Honour aside, Downey's single-number step sequencer technique opens a set of possibilities so wide that it is worth considering as a separate tier of technique. For my goal of replicating "Flight of the Bumblebee," that tier seems unnecessary, my finished product would be a very poor representation of music native to that tier, and inasmuch, I would be missing my greater goal of understanding bytebeat's musical nature. Here, extramusical honour and purely musical lessons about bytebeat are intertwined: honour has a stronger connection to my intuition than my abstract research subject matter does, intuition informs my path of inquiry, and the framework presented here helps me articulate musical reasons for that gut response.

4.2 The Metagame of Music

This discussion brings to light some notions that deserve further exploration in future work. Powerful tools are valued when the point is to complete a task. The dishonour of the overpowered reminds us that the point of gaming is to play rather than to finish, although success is also valued, in balance. This is also how extramusical elements can enhance a performance experience — not what is done but how it is done, the metagame of performance. Extramusical elements are not necessary for a rewarding musical listening experience, but purely musical enjoyment relies more heavily on the mindset of the listener — a metagame of listening musically.

5. Conclusions

While a full technology tree and taxonomy are still in progress, this account shares unexpected lessons from the early stages of developing and analysing bytebeat techniques and aesthetics, toward realising a full technology tree and an optimal taxonomy of bytebeat techniques, which will facilitate further analysis and spark further creative exploration. To summarise the relationships among these concepts, the technology tree elucidates how different abilities are interrelated, and it informs choices regarding development paths. Tier lists group feature sets by approximate levels of sophistication or power, and they expose the use of techniques that are overpowered in relation to the established context and the task at hand. Special challenges lie in contrast to overpowered approaches. While they may be seen as more honourable, this is in part because they avoid invoking a higher tier of tools or techniques than is necessary, allowing the resulting achievement to more fully explore and manifest the essence of the tier it is primarily exploring. These concepts are all parts of the metagame, which, when applied as a lens upon musical aesthetics, articulates the influence of extramusical elements on audience experience, as well as the power and importance of the listener's mindset in listening musically.

Bytebeat purists are few if there are any. Most treat bytebeat as a mere curiosity and give the nod to demoscene-style honour but readily embrace more advanced taxonomies, perhaps because it feels challenging enough to work under the "single line" rule. I am not a purist, either, although I find it useful to see purism and deviations from it clearly defined. I have pursued this research path, most practically, for the pedagogy, including advancing my own skills in bytebeat performance. So, one could make the self-similar reflection (a metagame of research) that in pursuing my personal path of development as an artist-scholar, I chose to research bytebeat pedagogy, which led me to begin articulating a technology tree and led to analysis and theory. The technology tree put techniques at my disposal in performance, in a conceptual framework to navigate among them more facilely in performance, but it also allowed me to discover and articulate taxonomies of bytebeat interpreters, which allowed me to begin reflecting on the aesthetics of bytebeat performance in the ways described here.

Thanks to Peter McCulloch for many enlightening discussions along this path of inquiry, and thanks to the students of PERF 318 Electronic Composition in spring 2019 at Texas A&M University for pursuing this inquiry with me. A playlist of performances and tutorials resulting from this class is available at: https://www.youtube.com/playlist?list=PLe4ojWnlX92OOrDhM8_yGIP9LNqSE_gR2

6. References

[1] V.-M. Heikkilä, "Discovering Novel Computer Music Techniques by Exploring the Space of Short Computer Programs," arXiv, Dec. 6, 2011. https://arxiv.org/abs/1112.1368

[2] Vinazza, Gabriel. Rampcode (Version 4ff8343, 2019). Buenos Aires, 2018. [Program Code]. https://github.com/gabochi/rampcode/

[3] N. Montfort, "Sound," in *Exploratory Programming for the Arts and Humanities,* Cambridge, MA: MIT Press, 2016, pp. 249–256.

[4] G. Tavares, *HTML5 Bytebeat* (Version 35874c1, 2019). Tokyo, 2012. [Online]. https://greggman.com/downloads/examples/html5bytebeat/html5bytebeat.html (Accessed: November 8, 2019).

[5] F. Tresham, *Civilization.* United Kingdom: Hartland Trefoil, 1980. [Board Game].

[6] S. Meier, Sid Meier's Civilization: Build an Empire to Stand the Test of Time. Hunt Valley, MD: MicroProse, 1991. [Video Game].

[7] T. Ghys, "Technology Trees: Freedom and Determinism in Historical Strategy Games," *Game Studies: The International Journal of Computer Game Research*, vol. 12, no. 1, Sept. 2012. http://gamestudies.org/1201/articles/tuur_ghys

[8] C_Mill24, "Tier Lists," *Know Your Meme*, 2016. https://knowyourmeme.com/memes/tier-lists (Accessed Nov. 8, 2019).

[9] Tier Farm, "Sloths Used to Be Overpowered" (Nov. 10, 2018). [Online Video]. https://www.youtube.com/watch?v=gn6sQt_wwBQ

[10] C. Crawford, "Design Techniques and Ideals for Computer Games," *Byte: The Smail Systems Journal*, vol. 7, no. 12, pp. 96–98, 1982. https://archive.org/stream/byte-magazine-1982-12/1982_12_BYTE_07-12_Game_Plan_1982#page/n97/mode/2up

[11] L. Sullivan, "13 Hardcore Challenges Invented by Players," *GamesRadar+*, 2015. https://www.gamesradar.com/13-hardcore-challenges-invented-players/

[12], J. Liljedahl, *Bitwiz Audio Synth: ByteBeat Machine* (2.3.6, 2017). Stockholm: Kymatica, 2012. [Mobile Application]. http://kymatica.com/apps/bitwiz

[13] V.-M. Heikkilä, *IBNIZ*. Oulu, Finland: 2012. [Computer Program]. http://pelulamu.net/ibniz/

[14] L. Vygotsky, *Mind in Society: The Development of Higher Psychological Processes*. Michael Cole, Ed., Cambridge, MA: Harvard University Press, 1978.

[15] N. Rimsky-Korsakov, "Flight of the Bumblebee," in *The Tale of Tsar Saltan*, 1900. [Musical Composition].

[16] *Laptop Deathmatch* (Nov. 24, 2005). [Archived Website]. https://web.archive.org/web/20051124055942/http://www.laptopdeathmatch.com/

[17] K. Downey, "Mary Had a Little Lamb" (Aug. 15, 2016) [Program Code], in a comment on V.-M. Heikkilä, "Algorithmic symphonies from one line of code -- how and why?" (Oct. 2, 2011). http://countercomplex.blogspot.com/2011/10/algorithmic-symphonies-from-one-line-of.html?showComment=1471248354487#c6209059819789654981 (Accessed Nov. 8, 2019).