Note: Please buy Zizi's book and cite the published version of this paper. Please also note there may be differences between this author's version and the published chapter.

Stevenson, Michael, and Robert W. Gehl. 2018. "The Afterlife of Software." In *A Networked Self and Birth, Life, Death*, edited by Zizi Papacharissi, 190–208. New York: Routledge. [1]

Death on the Internet is not limited to human death. The business model of planned obsolescence, the technical work of preserving old websites, systems and applications, as well as a cultural emphasis on the new and immediate all combine to make the Internet a place where many software technologies have gone to die. From the earliest web servers and browsers to relatively recent platforms like Vine, web software has disappeared due to fashion, neglect, or entropy.

Networked modes of living engender networked modes of loss, and a key question is how our connection to the past is reconfigured when software dies. As scholars in the discipline of software studies have routinely shown, running software is lively and agential, and understanding how it shapes the contours of networked life (past or present) arguably requires its execution. In order to better understand digital history, then, it stands to reason that we need to bring history to life -- specifically, historical software to life. This chapter explores the techniques, pitfalls, and promises of resurrecting old software for historical appreciation and analysis.

Today, discussions of software preservation are largely contained within communities of professional and amateur archivists, heritage institutions, and academics from computer science and archival studies (Nguyen & Kay, 2015; Rosenthal, 2015; Ross & Gow, 1999; Stuckey & Swalwell, 2014; Swalwell, 2009). Although these discussions detail the specific material, legal and cultural challenges to software preservation, so far little has been written about the *kind* of history made possible within these specific constraints and affordances. The question is brought into particular focus through recent projects that reconstruct obsolete or forgotten software in a new, "lively" context, such

[1] Research for this chapter was supported by the Dutch National Science Foundation (NWO) in connection with the Veni research project 'The web that was.'

as an Amsterdam-based effort to replicate the 1990s online community website De Digitale Stad ("The Digital City," abbreviated hereafter as DDS) (Alberts, Went, & Jansma, 2017), or the KDE "restoration project" in which the first version of KDE's Unix desktop environment was rewritten to work on modern machines and operating systems. Such projects blur the line between preserved object and contemporary context in a way that seems to go against the spirit of institutional preservation and serious historical research, but arguably does more to recreate user experience than traditional preservation and static portrayals. More importantly, their time-shifting effects should perhaps be seen less as an exceptional approach within digital preservation, and more as an extension of the fundamental "dynamic" nature of its objects (Alberts et al., 2017).

In the following, we outline an argument for understanding such software resurrection as *reenactment*[2] (c.f. Lowood, 2016). Materially, software preservation depends largely on the precise reconstruction of an underlying computing environment, a process that may require finding and restoring period hardware and equipment, yet increasingly depends on emulation, or a virtual reconstruction of the original computing environment on a new platform. In the first section we discuss how the work of emulating software conflicts with the widespread (yet erroneous) notion of software as context-free source code, instead revealing the complex set of dependencies – computational and otherwise – needed for old software to run. In the second and third sections, we we turn to the contemporary landscape of software preservation – from ambitious calls for universal solutions to the activities of open-source communities building console and PC emulators – to show how software reenactment is necessarily situated within a diverse and entangled set of historical, commercial and

---

2 The authors are indebted to Gerard Alberts for suggesting the connection between software emulation and historical reenactment during the *404: Internet History Not Found* workshop organized by Camille Paloques-Berges and Kevin Driscoll at the Association of Internet Researchers 2016 conference in Berlin. We also would like to recognize Henry Lowood's (2016) discussion of the relationship between software preservation and historical reenactment. Where Lowood is interested in how preservation efforts may enable the kind of commitment to lived experience that is seen in historical reenactment (and the extent to which this is feasible), our purpose is to discuss how reenactment offers a frame for understanding software preservation both as material and cultural practice.

aesthetic values and motivations. In the fourth and final section, we ask how software reenactment resembles existing reenactment and revival practices, and thus what we can learn from scholarship on these topics. In this way, software reenactment – in particular such popular initiatives as the Internet Archive's Internet Arcade https://archive.org/details/internetarcade) and Console Living Room (https://archive.org/details/consolelivingroom), as well as such playful reconstructions as the DDS and KDE restoration projects – may be considered in terms of what Vanessa Agnew calls a broader "affective turn" in how we relate to the past, one that collapses temporalities and emphasizes affect, experimentation, embodiment and play over traditional practices of historical contemplation (Agnew, 2007).

The ideas presented in this chapter stem in part from a project on emulating the Everything Development Engine, an ambitious and innovative web framework written in Perl in 1999 (Stevenson, 2016). Throughout this essay we have inserted accounts of this first-person experience of emulating "dead" software. These italicized sections serve to illustrate some of the theoretical points and show how resurrecting the digital dead is at once pleasurable, nostalgic, frustrating and boring – in other words, how it is lively.

Where software ends and begins again

As Wendy Chun (2011) argues, our contemporary understanding of software relies on a fetishization of source code. Although originally used as a verb, "program" in the context of computing has taken on the meaning of a concrete object, one conflated with the storage medium on which software is recorded and abstracted from the computing environment required for its execution. To understand source code as something automatically executable, one must disregard how it can only be executed within a

particular computational environment (e.g. a specific compiler or interpreter, operating system, hardware, etc.) (Chun, 2011). Source code, Paul Dourish notes, is commonly perceived as a series of precise instructions carried out by a dutiful processor, yet is also radically underspecified: issues of processing speed, memory management, and other details of the "instructions" necessary are typically not found in source code, delegated as they are to other "layers" in the computer's architecture. This means there is always some "gap between what is denoted and what is expressed, or between the specification and the execution" (Dourish, 2016, p. 33). This gap becomes particularly noticeable in attempts to bring outmoded software back to life.

The challenge of software preservation is not so much restoration as it is careful reenactment. Whereas restoring hardware to working order may require repairing vacuum tubes and transistors, the work of reviving old software often lies in reconstructing the underlying computing environment on a new platform. Original software can only run unmodified on the hardware and software it was designed for, or in environments that reproduce the underlying architecture all the way down. Such reconstructions on new platforms are called *emulation*, and can be distinguished from *simulation*, or creating a new object that imitates the external features of the entire stack – for example, a PC program that imitates both the visual and functional characteristics of an old arcade game (Ross & Gow, 1999). In terms of digital preservation strategies, emulation may also be distinguished from migration, or periodically moving data and software to new environments, "rewriting" them as required (Holdsworth & Wheatley, n.d.).

Once a program's underlying hardware and operating system have been emulated – a process that can have significant hurdles (Dietrich, Kim, McKeehan, & Rhonemus, 2016) – running original software may face further obstacles in the form of the dependency graph.[3] A program typically has a range of dependencies from a specific operating system and various programs to different software

3 The term dependency graph is most often used in connection with automated software installers that "walk" the graph to discover missing packages.

libraries and packages, some of which may not be standard on a given system. For example, programming languages like Python and Perl may be extended with libraries of pre-written code, and each of these will in turn have their own set of dependencies and so on. Taken together with underlying hardware and software, one can think of all software as embedded within a network of requirements among programs, systems, packages, drivers, peripherals and so on – whatever dependencies are not emulated will have to be found and installed. Although the problem of the dependency graph obviously applies not just to historical software, it is exacerbated by time: an old program's dependencies will likely include specific (contemporaneous) versions of other software, which the archivist or historian will need to recover or simulate.

Running old software on an emulated system turns the concerns of software production on their head. Best practice demands that developers spend significant amounts of production time testing software for interoperability, adapting their software to meet the requirements, specifications and quirks of different machines, operating systemss, browsers and so on. The software archaeologist is faced with the opposite task of running code on different systems or configurations until a suitable environment is found; that is, trying different configurations of an emulated environment until there are no bugs or the only errors produced are those that were found in the original.

Software does not end with source code, nor with electronic pulses producing material changes in underlying hardware and storage media (Kittler, 1995). Time reveals how the dependency graph consists not only of computational objects, but also "tacit knowledge" formed through experience and expressed through embodied practice and human memory (MacKenzie & Spinardi, 1995). Extending this metaphor further we might think of how reviving old software aligns with the genealogical aspects of "critical reverse engineering" (Gehl, 2016), a process by which a technical artifact is closely examined with an eye towards moving from its specificity back to the more abstract design principles, goals, and values that were held by its creators, and in turn the social, cultural and economic histories

that shaped them. We may only have access to the artifact itself, but we are interested in seeing the "way of life" that helped produce the artifact.

Even in the case of software from the recent past, running old code on emulated machines may easily be obstructed by the contingencies and frustrations of rebuilding the dependency graph (Dietrich et al., 2016). The Everything Development Engine, described in 1999 by creator Nate Oostendorp as an open-source, multi-user information management system – what today might be termed a web framework for building social media sites – seems a relatively safe bet for preservation. The engine was written at the height of celebrations of all things open-source, by a member of the original Slashdot *crew no less, and was clearly "cool" software (Liu, 2004) in that it was a highly sophisticated yet somewhat subversive information management product. Beyond the then-controversial idea of allowing users to publish and edit website content, the software proudly broke several programming conventions, for example by storing scripts in the database, allowing scripts to be edited from the web interface and taking a hypertext-inspired design principle ("Everything's a node") to various clever and complicated extremes. The dot.com crash had forced its developers to abandon the project in 2001, but what it lacked in longevity and widespread use it made up for in pioneering features and symbolic capital, and thus seems likely to receive sustained historical interest. Perhaps most importantly, as an open-source product from this period, it is still available on SourceForge along with documentation. As part of my research on the engine as "autonomous" new media culture (Stevenson, 2016), I asked two friends with extensive web development experience, Joost Rohde and Marijn de Vries Hoogerwerff. if they could help me get it running. Joost glanced at it quickly and replied I should drop by their office someday after work; it would take an hour or two at most.*

Several Friday evenings later, surrounded by cold take-out food and warm beer, we sat deflated, the terminal window displaying error after fatal error. The software appeared to be dead. We were

close to giving up, and I remember thinking how abstract the code would remain. There was no way to meaningfully study the source code, a bunch of folders and files with names like "nodeballs," without actually running it. Although a few websites like Perlmonks.org are still run on highly modified versions of the engine and much of the original's functionality can be gleaned from them, the only real access I had was archived documentation and the memories of the developers I would interview. I was bummed.

Our project had gotten off to an auspicious start. The DEBIAN_SETUP text file described the environment Nate Oostendorp was using to run Everything: the Debian 2.1 operating system, the Perl programming language and the MySQL and Apache programs included with that Debian release, and several dependencies in the form of additional Perl libraries such as a database interface. We located an image file for Debian 2.1r2 (a revision that included millennium bug fixes) and installed it onto a virtual machine, and smiled when we realized it would take a half-hour to install. Our good mood vanished, however, when upon installation it became clear we couldn't connect to the internet, which we would need to do to install additional software packages. Joost recalled how early versions of Debian often had compatibility issues with various network cards – a problem that carried over to the virtual network card conjured up in Qemu, the x86 emulation software we were using to construct our virtual machine. Marijn started to look into alternative emulation software, while I searched in vain for old online discussions of the problem. Meanwhile, Joost drew on memory and his general knowledge of network cards and drivers to play with the settings until the network connection finally worked. Between installation and the network connection, we'd already spent a few hours on the project and decided to call it a night. Tired yet also accustomed to drawn-out software problems like this, Joost quipped "At least we've recreated the experience of doing open-source stuff in the 1990s."

Universal machines and cuneiform tablets

How do material constraints on archiving software – obsolete data formats, deteriorating storage media and the dependency graph – animate current preservation efforts? The stakes of this problem are what internet pioneer and Google vice president Vint Cerf calls the "digital Dark Age" (Ghosh, 2015). Cerf warns that our digital information is poorly suited to long-term preservation, emphasizing that "bitrot" is not just about storage but also preserving our ability to interpret stored data in the future (Cerf, 2011). Without solutions to the formatting and software problems of digital preservation, the Dark Age that could result would be one where "even if we accumulate vast archives of digital content, we may not actually know what it is" (quoted in Ghosh, 2015).

Cerf's vision of a digital Dark Age puts some of the more ambitious proposals for "solving" digital preservation into perspective. Recognizing that straightforward preservation of original hardware and software is not viable in the long-term (if even the short-term), a combination of heritage institutions and computer scientists have proposed several large-scale solutions. Proposed in the early 2000s and seemingly stalled in development since 2007, the Universal Virtual Computer (UVC) (Lorie & van Diessen, 2005) combines emulation and migration strategies in a machine that would decode and reconstruct preserved digital objects. UVC follows "the insight that durable access to digital objects need flexible, robust and durable standards and standardization of technical subjects" (van der Hoeven, Van Diessen, & van der MEER, 2005, p. 197), and relies on a UVC format decoder (written before a format becomes obsolete) and a pre-constituted "language" for representing the original objects, for example a set of attributes and defined values for representing various object "types" such as images or spreadsheets. While it is geared more toward these various content types, one could imagine the UVC model applying to software as well, migrating programs on demand by decoding source code from known programming languages and translating instructions into a pre-defined set of UVC operations. However the feasibility of such an undertaking could easily be called into question, given the added

complexity of dependency graphs and the diversity of different (versions of) programming languages, as well as the impact this would have on the overall usability of the UVC itself.

Beyond the issue of complexity, there is the question of what is preserved. The future historian would view preserved content by running the platform-independent UVC, which automatically migrates the object “and reconstructs a specific representation of the original object’s meaning” on the universal platform (van der Hoeven et al., 2005: 198). This vague language about “the object’s meaning” relates to a universal truth about archives as constructed objects – every emulation or migration effort requires some sort of conscious “abstraction” in which the “significant properties” of the object must be selected for preservation (Holdsworth & Wheatley, n.d.). In this case the model is geared toward individual digital objects, privileging not only content over context but also product over process. For example, a UVC approach may recreate the functionality and appearance of, say, an old word processor, however this would likely not reproduce the experience of installing the software or of its interaction with the original platform and operating system.

In contrast to the one-size-fits-all approach of UVC, Long Tien Nguyen and Alan Kay (Nguyen & Kay, 2015) propose a software preservation system that seeks to retain more of the original character of software and its underlying environment, while also lowering the threshold for future historians to engage with forgotten software, systems and architectures. In their imagined technology of "cuneiform tablets" for the digital age, the archivist would package a program as a bitstream that includes an emulator for the original platform. This pre-cooked emulator would be designed to run on a simple virtual machine that a future programmer could build during “an afternoon’s hack” (ibid). By bundling a development environment along with the program itself, the tablet would allow for further exploration and remixing, thereby preserving not just bits but a particular technological ethic of transparency and play (Coleman, 2012).

Both approaches, in addition to assuming a long-term storage solution will become available, conceptualize preservation as primarily a technical problem rather than a cultural or economic one. Computer scientist and digital preservation scholar David S.H. Rosenthal (Rosenthal, 2016) points out that even if technically feasible, the economic costs they imply far exceed the willingness of political and corporate actors to implement such schemes. Because of this, Rosenthal argues, such attempts to devise "a one-time solution [are] a distraction from the real, continuing task of preserving our digital heritage" (ibid). Instead, preservation involves repetition, redundancy and continuous auditing, as this is the best way to guard against multiple material and cultural threats, including those that are not yet known. Elsewhere, Rosenthal and colleagues argue that this continuing preservation process should be "bottom-up" and "transparent" (Rosenthal, Robertson, Lipkis, Reich, & Morabito, 2005), with individual preservation efforts conforming less around technical standards, and more around how they define their threat models and disclose their efforts to combat them.

The solitary review on the Everything Development Engine's SourceForge page was posted by user n_oostendorp on July 20, 2009, ten years after the Engine was written. Oostendorp had started working at SourceForge in 2004, a few years after shutting down the Everything Development Company and starting his search for a"real job." All these years later, perhaps aware of the anniversary, but more likely simply testing SourceForge's new system of user ratings and comments, Oostendorp gave Everything 5 stars and wrote: "What other web development framework even begins to approach it's [sic] utter magnificence?" ("Everything Development Engine," n.d.)

A similar mix of pride and irony is found throughout Everything's history. Because of their work on Slashdot, Oostendorp and his friends - Rob "Cmdr Taco" Malda, Jeff "Hemos" Bates and several others – had fast become the equivalent of rock stars in the open source community. And while they enjoyed their celebrity and the virtual riches they'd acquired through selling Slashdot during the

open-source bubble of 1999, they retained their roots in a hacker culture in which code was supposed to do the talking, and outward expressions of pride were muted. "The art of programming," wrote Dijkstra (1970), "is the art of organizing complexity, of mastering multitude and avoiding its bastard chaos as effectively as possible." Such familiarity with chaos tends to rein in inflated egos. And so Everything's actual technical ambition was displayed only through layers of irony, like a programmer's version of Dave Eggers's novel A Heartbreaking Work of Staggering Genius *(released a year after Oostendorp started his company). In addition to the ironic ambition suggested by its name, there was Everything's playful use of Perl, as well as evident joy and requisite self-mockery in its documentation. Everything's "cool" was in turn situated in a particular set of financial and symbolic economies that afforded a "have-it-both-ways-ism" that permeated this corner of tech culture in the 1990s. Everything was both fun and productive, subversive and corporate:*

> *Note to reader: This corporate philosophy (like most) has no practical value. The objective of a business is to make money. The objective of a geek is to do cool stuff. Sometimes we try and marry the two. ("Corporate Philosophy," n.d.)*

The idea for Everything came to Oostendorp from the successes of IMDB and the All Music Guide, which represented the kind of rich, flexible databases that Tim O'Reilly had dubbed "infoware" (O'Reilly, 1998). *Combined with Slashdot's army of content-generating users, a new generation of infoware would comprise themed websites where users could read and write about their favorite topics while a sophisticated backend served up related article after related article. Inspired by Slashdot's "karma"-based comment moderation system and a lifetime of playing Dungeons & Dragons, Oostendorp also knew that a key to encouraging participation would be to have users compete for a quantified testament to their prestige among their peers. Although the similarities with later social media are obvious, there were key differences in Oostendorp's vision, from the relatively low emphasis on images and personal profiles (although both were included) to the idea that there would be*

numerous sites for different themes rather than a single platform that would serve multiple communities. Rather, the Everything Development Company would maintain a platform for building this future army of websites. The company would release its software under an open-source license, and earn revenue through consulting and customized solutions.

On SourceForge, little evidence of Everything's rich and lively history is on display. There are categories and other metadata. There is a list of contributors, and there are the various folders with version numbers, including the one we chose for our emulation (called "Everything 0.8.3 [old]"). There is the about section and assorted readme files, all of them written in a colder, more technical tone when compared to the company website, an artifact now accessible only via the Internet Archive's Wayback Machine. And, at the bottom of the page, there is the five-star review.

Consoles and the cloud

If bottom-up, continuous preservation is the way forward, then software's afterlife will depend not just on the work of a few heritage institutions. Today, software preservation already benefits from the ad-hoc efforts of communities of amateur computing enthusiasts, especially gaming communities. And while the technology industry is rightly criticized by preservation advocates for putting commercial and proprietary interests ahead of cultural heritage, the relationship between industry and digital preservation is more complicated: technologies for software development and "virtualization" (a technique for maximizing the efficient use of computing power, especially in cloud-computing) also provide infrastructures for preserving archived software and making it accessible to users. However, neither amateur participation nor industry practices align perfectly with preservation aims – instead, what is clear is that the contributions made by these "outsiders" to software preservation are subject to different, at times conflicting hierarchies of symbolic and economic value.

Several paradoxes characterize the preservation of digital games. Games are crucial to the history of computers and their widespread adoption in the 1980s, yet they are also "low" culture and thus are often overlooked by mainstream cultural heritage institutions (Swalwell, 2009). Like all software, games must be run in order to be experienced, however Intellectual Property (IP) restrictions keep the few institutions that do collect games from making emulated versions available to the public (Barwick, Dearnley, & Muir, 2011).[4] Meanwhile, despite these institutional roadblocks and the fact that console emulation is especially difficult (since both hardware and software tend to be proprietary), digital games preservation is arguably the most successful form of software preservation today, due to the efforts of emulator developers. Emulators for video games and arcade games began to appear in the mid- to late-1990s (Wen, 1999), and these "amateur" efforts have resulted in a great number of individual emulators as well as large-scale projects like the Multiple Arcade Machine Emulator (MAME) and the Multi Emulator Super System (MESS). MESS was incorporated in MAME in 2015, and MAME now emulates thousands of arcade machines and consoles (with the exception of modern consoles that are “likely to remain beyond the state of the art” of emulation [Rosenthal, 2015]).

Given the IP issues surrounding emulation, the MAME project makes a point of stating that its:

> purpose is to be a refefence [sic] to the inner workings of the emulated machines. This is done both for educational purposes and for preservation purposes, in order to prevent historical software from disappearing forever once the hardware it runs on stops working. Of course, in order to preserve the software and demonstrate that the emulated behavior matches the original, you must also be able to actually use the software. This is considered a nice side effect, and is not MAME's primary focus (“About MAME,” n.d.).

Playing original games is framed as necessary for validating technical knowledge of the platform (Murphy, 2013), and in this way preservation is not only a goal but helps to legitimize technology that

4 A notable exception to this is the Internet Archive, which has made emulated games available through the *Internet Arcade* (https://archive.org/details/internetarcade) and the *Console Living Room* (https://archive.org/details/consolelivingroom).

routinely comes under fire for copyright violations. As Murphy (Murphy, 2013) argues, MAME may be understood as a "ludic technology" (Consalvo, 2007) through which users wrest control from game console companies, thus continuing a tradition of participatory engagement with digital games seen in practices like modding and machinima. The overlaps with hacker culture are also clear: where early MIT hackers created games as a diversion from their computer science research, MAME constitutes serious emulation research done by game players (Murphy, 2013).

The particular legal and cultural context of emulation gives rise not only to the rhetorical moves through which these projects distance themselves from piracy, but also a value-system that departs in some ways from that of traditional preservation. This is what Murphy calls "code-based authenticity," where the "authentic" archival object is not considered to be physical objects like the arcade machine or console, but rather the original code that ensures the game's sound and images are reproduced exactly, regardless of underlying hardware. Murphy also argues that "by defining authenticity on a code level, [MAME] is specifically designed so it will not compete with commercial emulation" (Murphy, 2013), which historically has focused on usability and (improved) performance (Rhodes, 2007). MAME's emphasis on accuracy, by contrast, results in some "games that run so slowly on current hardware systems that they are virtually unplayable" (Murphy, 2013).

Other factors impacting the work done by these communities are the developers' sense of prestige as well as the commercial aims of the growing retro-gaming industry. Building emulators for arcade machines and consoles typically requires a mix of archival research and reverse engineering, and emulating marginal or "forgotten" machines thus brings a greater sense of accomplishment as well as more esteem from peers (Altice, 2012). At the same time, emulation-as-preservation must also be seen within its commercial context, as the retro-gaming industry is estimated to be worth $200 million per year (Geigner, 2015). This industry is largely associated with high-priced vintage machines and games, however it also includes a range of emulated and simulated products that thrive despite the

widespread availability of pirated works (ibid). MAME itself must also be seen in this commercial context: its "code-based authenticity" may separate its work from commercial vendors, as Murphy contends, but the project also explicitly makes cooperation with the industry possible. While MAME as a whole is licensed under the GNU General Public License, much of its code is available under the more flexible 3-Clause BSD license, which provides fewer hurdles for creating derivative commercial products.

The tangled relationship between business and preservation is even more pronounced in the area of PC emulation and virtualization. PC emulators were first created in the 1990s as software and hardware development tools, which would make it cheaper and less time-consuming to test new products in different environments. Importantly, this requires a dedication to "code-based authenticity," even if the goal of debugging new products is seemingly more instrumental than that of reviving forgotten video games. As the developers of Bochs (the first open-source PC emulation project) note in their documentation, emulation as development tool only works through a commitment to detailed accuracy: “Because Bochs simulates the whole PC environment, the software running in the simulation "believes" it is running on a real machine. This approach allows Bochs to run a wide variety of software with no modification” (“Introduction to Bochs,” n.d.).

Although emulation has an important to role to play in preserving legacy systems that are often "mission-critical" to various business and organizations (especially military) (Harris, 2013), innovation in the area of emulation is now largely driven by the need to make more efficient use of processing power in "cloud" infrastructures. Virtualization refers to running multiple virtual machines concurrently on one system. The technique actually has a long history in computing and was originally developed to enhance time-sharing in the mainframe era, beginning with IBM’s CP/CMS system released in 1968 (“History of Virtualization,” 2011). As Nicholas Carr (Carr, 2008) argues in *The Big Switch*, the present era of cloud-computing represents somewhat of a return to the centralization of

processing power that characterized mainframes, and it is therefore no surprise that the innovations of the previous era have been given new life. Because of the relatively long life-span of computing architectures – the x86 family of processors dates back to 1978 – the ongoing development of open virtualization products must be taken seriously for their potential as preservation tools (Rosenthal, 2015).

There is no question that preservation efforts are enhanced by the work being done on PC virtualization, however it is also apparent that commercial motivations will help determine which older software can be revived. For example there are known issues with running old versions of Windows on QEMU (now the major open-source PC virtualization software), but the community does not seem likely to address these anytime soon. Rather, QEMU largely "evolves to meet current demands" (Rosenthal, 2015).

We had Debian 2.1 running. From the command line, we started MySQL and the Apache web server. We used Qemu's feature for sharing an internet connection between host and virtual machine, and smiled as our present-day browser displayed Apache's message from 2001: "Welcome to your new home in Cyberspace!" Our LAMP (Linux-Apache-MySQL-Perl) stack was complete and we ran Everything's install file, only to see the terminal window overflow with long lists of errors. The problem was the remaining outstanding nodes in this particular dependency graph. Although the Comprehensive Perl Archive Network (CPAN) largely lives up to its name, there were specific versions of specific Perl modules that we could not find online, a problem that snowballed as some of the modules and packages that we were able to find would not compile. No matter what we tried in terms of mixing older and newer versions of the various modules, or how many lines of Everything code we "commented out" in the naïve hope that bugs would be circumvented, the program failed to run. For the second time, we called it a night.

Meanwhile, I had committed to presenting a workshop paper titled "CPR for a CMS: on bringing the Everything Engine back to life." I wasn't worried about the conference paper, since there was plenty to discuss even if our efforts failed, but I also figured that if it didn't happen now I'd move on to other projects. Although Marijn was away, I managed to convince Joost to try one more time, namely an install of Everything on a newer version of Debian. This could bring about a different set of dependency-related bugs, and either way would feel less authentic, but we agreed to try nonetheless.

We found images of the install cds quickly (they are hosted on archive.org, among other places), and as Joost went to work on a new virtual machine I compared the requirements listed in Everything's readme files to the version numbers of various software included in Debian 2.2. I expected differences that would need to be resolved, but the lists matched perfectly. Even better, we soon discovered that all of the packages we needed – the correct versions, even – were included on the second install CD. This may have been a fluke where Oostendorp's setup essentially turned Debian 2.1 into its successor, or possibly a kind of archival miscommunication: I suspect that between Oostendorp's frequent updates to Everything in 1999-2002 there was a mixup in which an older readme file was included with a later version of Everything. Regardless, it worked.

In the terminal window, we saw a few errors, but these were delivered to us from the program itself. In the browser we saw "The Default Node," the landing page for the engine that carries Oostendorp's ironic declaration: "this is the clean everything. this is the master everything. New Everythings are merely a shadow of this." I had been in contact with Oostendorp, and immediately emailed him. For no real reason, I sent a photo rather than a screenshot, and the strange choice

reminds me of how excited we were to get it working.

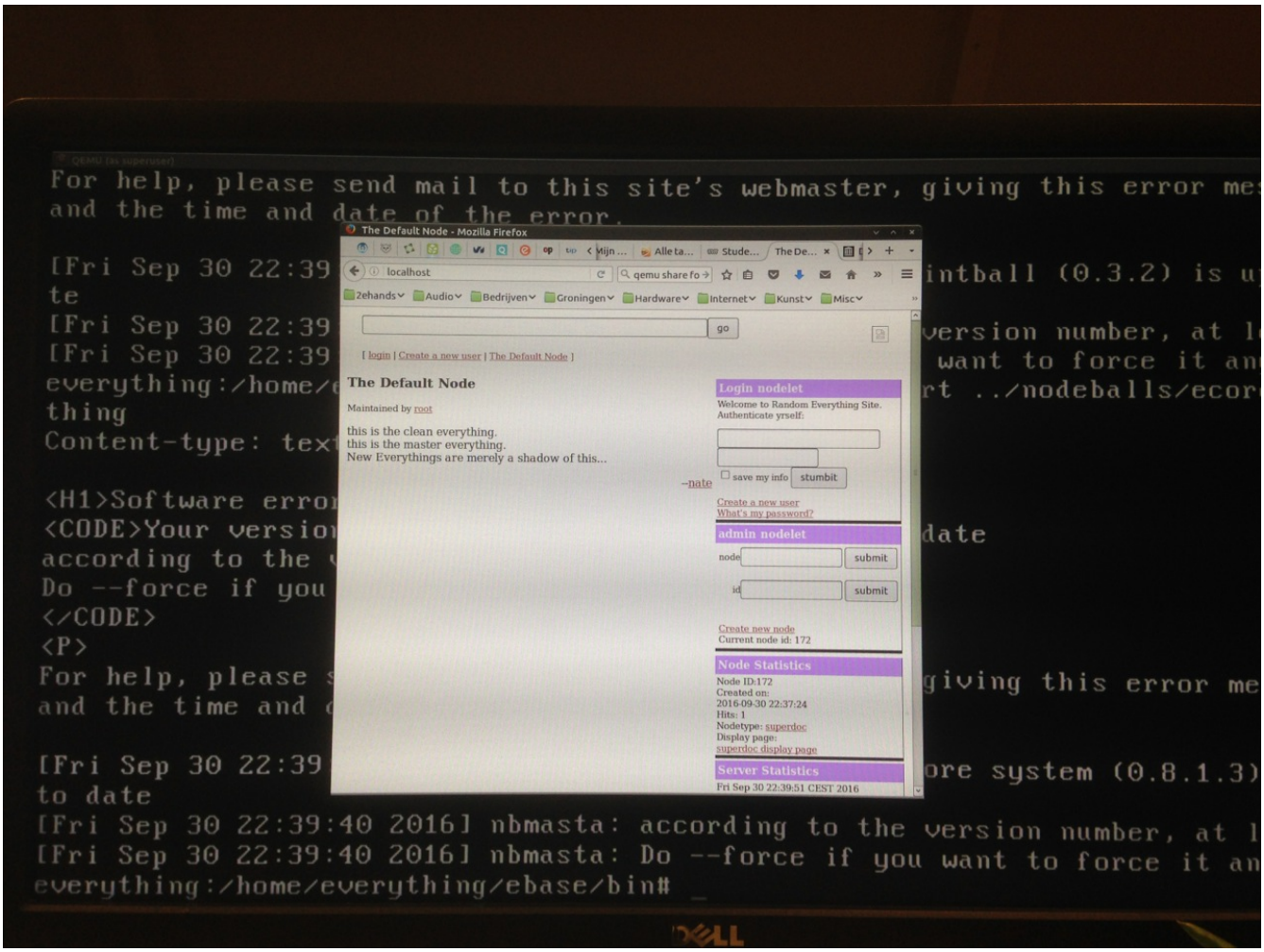

Figure 1: screenshot of the Everything Development Engine running on a virtual machine (courtesy of Nate Oostendorp)

---

4. Theorizing software reenactment

The diverse material challenges of software preservation as well as the varied social, cultural and economic contexts that govern preservation practices call into question not only the feasibility of software preservation, but also the status of the *form* of history such preservation affords. Here we would like to propose that understanding how old software may be brought back to life – both as material challenge and cultural practice – may be informed by scholarship on historical reenactment.

Historical reenactment is, of course, the practice of staging historical events and acting them out, with actors dressed in period-authentic costumes, using period-authentic technologies, all with the goal of, as it is often said, "bringing the past to life." As Vannessa Agnew, Jonathan Lamb, and Iain McCalman note in their introduction to the book series *Reenactment History*, historical reenactment "has begun to make its way into historiography as a new concept in the understanding of the past" (2010, p. ii). For them, reenactment is a "practical engagement with the past in which the empirical outcome is determined not by what is known in advance, but by the experience of making it" (2010, p. ii). As they and other scholars of historical reenactment repeatedly note, reenactment has had a somewhat dubious reputation among "serious" historians, who dismiss it as dress-up games and at odds with serious scholarship. However, in their contribution to the *Reenactment History* series, McCalman and Paul Pickering argue that "taking reenactment seriously as a methodology is worth the risk and that its potential is best explored through an interdisciplinary lens" (McCalman & Pickering, 2010, p. 13). Following their lead, we suggest that reenactment of past software can be a viable tool to re-experience the use of important software artifacts from the past.

A specific category of historical reenactment, music revival, is a good model for software reenactment. Kate Bowan draws parallels between historical reenactment and early music revival, a trend in musical performance in which older scores are played on period-specific instruments with historically accurate techniques, often in authentic settings. Revivalists speak of "bringing the music to life" (Bowan, 2010, p. 134). Bowan cites historian R.G. Collingwood, who argues, "We may therefore boldly say that the *sine qua non* of writing the history of past music is to have this music *re-enacted in the present*" (qtd. in Bowan, 2010, p. 137). She notes that many of the practices and debates in early music revival anticipate current practices and debates among historical reenactors.

In her theoretical model of music revival, Livingston argues that earlier musical styles are revived based on a host of factors:

- the feasibility of revival
- the existence of source material to draw on
- the interests of the revivalists

This is a complex "music system" (Livingston, 1999, p. 66) in which materials (period instruments), practices (period playing techniques, embodied in musicians) and discourses (authenticity, historical fidelity, social relations) are all associated. Indeed, as Livingston notes, music revival is often much more than a hobby; it is an attempt to critique dominant cultural practices (for example, a hypervalorization of the new or the modern), educate others about the past, and recover lost cultural values.

Moreover, just as in the larger practices of historical reenactment, music revival is marked by temporal collapses. On a surface level, music revival is marked by discourses of "timelessness," where the revived musical style is seen as always relevant, no matter its age (Livingston, 1999, p. 69). However, scholarship on music revivalists notes that the specificity of time is important to some revivalists: the music – and the musician – must *emphasize*, rather than universalize, the specific time period she is reviving (Bowan, 2010). Between these extremes of timelessness and temporal specificity, ethnographic work on revivalists have noted the constant, pragmatic trade-offs between older practices and modern ones; for example, wearing modern shoes when standing on one's feet all day performing (Decker, 2010; Shelemay, 2001). Music revival is always a mix of temporalities, from the use of antique instruments to modern recording technologies, but with the overall goal of learning more about past musical styles.

For our purposes, one of the analogies used to describe software is that it is like music. In music, there is the score, akin to lines of code in a software program, and there is the performance, akin to running software. While one can learn a great deal through study of the former, it is the latter,

performance, where the lines are executed on hardware, whether it be on musical instruments or on computer hardware. As a specific form of historical reenactment, music revival – bringing old music "back to life" – can inform our project of bringing old software back to life.

In addition, music revival relies upon deeply embodied – i.e., tacit – knowledge of the performers, who not only play their instruments, but do so with period-specific techniques gleaned through historical research and repeated practice. Our interest in software reenactment takes up this performative aspect to remind us that software relies on the tacit knowledge of programmers and users: keyboard shortcuts, arrangements on screens, body-computer articulations, debugging procedures, even one's own experience listening to the computer's fan and harddrive all come into play in the "performance" of software.

Moreover, the cross-cutting temporalities of reenactment/revival echo the odd temporalities and pragmatic trade-offs of software, where a software package is dependent upon other software packages and hardware, and where older software can be run on modern machines with the right mix of period-specific packages and custom code, and where the original programmers – some of whom are still alive and active – gather together with younger programmers to revive dead software.

In taking up reenactment and revival, we sound notes of caution: our interest in historical reenactment broadly and music revival specifically does not extend to the emphasis – even obsession – of the reenactors with "authenticity," or with exact replication of past materials or practices. Indeed, historians and scholars writing about reenactment lament the "authenticity fascism" that can come with the field (Gapps, 2010, p. 53). The goal of hyper-authenticity-minded reenactors is to bring the contemporary audience into a precise replication of a past event, down to exact smells, sounds, sights, and spatial experiences. However, as McCalman and Pickering warn us, it is deeply flawed to think that "somatic experience and feeling are timeless" (2010, p. 9). Despite analogies to "time machines," historical reenactments cannot truly transport contemporary audiences to the past; to claim otherwise is

to deny the social construction and evolution of cultural and affective experiences. Rather than a "mastery of the past" – the implied goal of the hyper-authenticity-minded – the goal ought to be to have a contemporary audience engage in a critical confrontation with the past (Agnew, 2007, p. 302). As Kate Bowan argues, this will have a paradoxical effect of making the past seem distant and strange – and thus open to critique. Feeling "close" to the past is to misconstrue it. To understand it is to sense its remoteness (Bowan, 2010, p. 151).

We also note the critiques historians level at reenactment: that it can very often add an entertaining and guilt-relieving gloss over deeply disturbing historical practices, such as colonialism or slavery (Agnew, 2007). Writing about the popular genre of reality television shows that reenact the past, Vanessa Agnew notes the common approach is to "elegize certain aspects of the past and elide what remains uncomfortable and troubling. Whether these reenactments advance new historical knowledge thus seems doubtful" (Agnew, 2007, p. 302). The common practice of reenacting battles can overwhelm institutional practices that are harder to reenact and yet are far more consequential (say, closed-door decisions made by powerful people), and the result is a substitution of military history for history itself (Gapps, 2010).

However, the lessons of historical reenactment, especially those of music revival, for our proposed *software reenactment* are important. The complex temporal shifts of dependency graphs and changing versions, the performative aspects of software, the association of lines of code with hardware platforms, the attention to detail required – all of these can be informed by analogies from reenactment. Beyond this material connection, actual software reenactments require the kind of emotional investment and volunteer efforts typical of historical reenactment and music revival. By taking software reenactment seriously, we hope it does for software studies what reenactment can do for historians: "reenactments open up possibilities that allow history to be, as is its want, unfinished business. They are also quite useful in getting to the heart of those events from the past that appear to

be left outstanding in the present" (Gapps, 2010, p. 61). As software lives and then dies in the churn of technological development, software reenactment can help contemporary people wrestle with the social and cultural implications of historical software-as-events. As Bowan puts it, "It may be impossible to have an 'authentic' affective relationship with the past, however the act of trying to form such an emotional connection can reveal much about that particular instance both past and present" (Bowan, 2010, p. 150). Software reenactment should have the same goals. It should remind us that the past is the past, that software in the past worked in a radically different way, that our contemporary software systems could have been different, and that there are many paths to take as one implements a social vision into software technology. Software reenactment is a mode of remembering and engaging with history that is iterative and networked. Ideally it reveals the past as such as well.

References

About MAME. (n.d.). Retrieved from http://mamedev.org/about.html

Agnew, V. (2007). History's affective turn: Historical reenactment and its work in the present. *Rethinking History*, *11*(3), 299–312.

Agnew, V., Lamb, J., & McCalman, I. (2010). Reenactment History. In I. McCalman & P. Pickering (Eds.), *Historical reenactment: from realism to the affective turn* (p. ii). London: Palgrave Macmillan.

Alberts, G., Went, M., & Jansma, R. (2017). Archaeology of the Amsterdam digital city; why digital data are dynamic and should be treated accordingly. *Internet Histories*, *0*(0), 1–14. https://doi.org/10.1080/24701475.2017.1309852

Altice, N. (2012, April 6). Interview: Paul Robson, programmer of the NESA emulator. Retrieved March 1, 2017, from http://metopal.com/2012/04/06/interview-paul-robson-programmer-of-the-nesa-emulator/

Barwick, J., Dearnley, J., & Muir, A. (2011). Playing games with cultural heritage: A comparative case study analysis of the current status of digital game preservation. *Games and Culture*, *6*(4), 373–390.

Bowan, K. (2010). R. G. Collingwood, Historical Reenactment and the Early Music Revival. In I. McCalman & P. Pickering (Eds.), *Historical reenactment: from realism to the affective turn* (pp. 134–158). London: Palgrave Macmillan.

Carr, N. (2008). *The Big Switch: Rewiring the World, from Edison to Google* (Edition Unstated edition). New York: W. W. Norton & Company.

Cerf, V. G. (2011). Avoiding" Bit Rot": Long-Term Preservation of Digital Information [Point of View]. *Proceedings of the IEEE*, *99*(6), 915–916.

Chun, W. H. K. (2011). *Programmed Visions: Software and Memory*. Cambridge, MA: MIT Press.

Coleman, E. G. (2012). *Coding Freedom: The Ethics and Aesthetics of Hacking*. Princeton: Princeton University Press.

Consalvo, M. (2007). *Cheating: Gaining advantage in videogames*. Cambridge, MA: Mit Press.

Corporate Philosophy. (n.d.). Retrieved May 1, 2017, from https://web.archive.org/web/20060507034746/http://everydevel.com/index.pl?node=Corporate%20Philosophy

Decker, S. K. (2010). Being Period: An Examination of Bridging Discourse in a Historical Reenactment Group. *Journal of Contemporary Ethnography*, *39*(3), 273–296. https://doi.org/10.1177/0891241609341541

Dietrich, D., Kim, J., McKeehan, M., & Rhonemus, A. (2016). How to Party Like it's 1999: Emulation for Everyone. *The Code4Lib Journal*, (32). Retrieved from http://journal.code4lib.org/articles/11386

Dourish, P. (2016). Rematerializing the platform: emulation and the digital-material. In S. Pink, E. Ardèvol, & D. Lanzeni (Eds.), *Digital Materialities: Design and Anthropology* (pp. 29–44). London: Bloomsbury.

Everything Development Engine. (n.d.). Retrieved May 1, 2017, from https://sourceforge.net/projects/everydevel/

Gapps, S. (2010). On Being a Mobile Monument: Historical Reenactments and Commemorations. In I. McCalman & P. Pickering (Eds.), *Historical reenactment: from realism to the affective turn* (pp. 50–62). London: Palgrave Macmillan.

Gehl, R. W. (2016). (Critical) Reverse Engineering and Genealogy. *Foucaultblog*. https://doi.org/10.13095/uzh.fsw.fb.153

Geigner, T. (2015, August 17). Retro Games Industry Booming Despite Pirate-Options Being Super Available. Retrieved April 13, 2017, from https://www.techdirt.com/articles/20150817/09563431981/retro-games-industry-booming-despite-pirate-options-being-super-available.shtml

Ghosh, P. (2015, February 13). Google's Vint Cerf warns of "digital Dark Age." *BBC News*. Retrieved from http://www.bbc.com/news/science-environment-31450389

Harris, D. (2013, December 13). Legacy Systems: Tried and True Systems Whose Time Has Come. Retrieved April 13, 2017, from http://www.datacenterknowledge.com/archives/2013/12/13/legacy-systems-tried-true-systems-whose-time-come/

History of Virtualization. (2011, August 1). Retrieved from http://www.everythingvm.com/content/history-virtualization

Holdsworth, D., & Wheatley, P. (n.d.). Emulation, Preservation and Abstraction. Retrieved from http://sw.ccs.bcs.org/CAMiLEON/dh/ep5.html

Introduction to Bochs. (n.d.). Retrieved from http://bochs.sourceforge.net/doc/docbook/user/introduction.html

Kittler, F. (1995). There is no software. *Ctheory*, *10*(18). Retrieved from http://www.ctheory.net/articles.aspx?id=74

Liu, A. (2004). *The Laws of Cool: Knowledge Work and the Culture of Information*. Chicago: University of Chicago Press.

Livingston, T. E. (1999). Music revivals: Towards a general theory. *Ethnomusicology*, *43*(1), 66–85.

Lorie, R. A., & van Diessen, R. J. (2005). UVC: A universal virtual computer for long-term preservation of digital information. *IBM Res. Rep. RJ*, *10338*.

Lowood, H. (2016). It Is What It is, Not What It Was. *Refractory: A Journal of Entertainment Media*. Retrieved from http://refractory.unimelb.edu.au/2016/08/30/henry-lowood/

MacKenzie, D., & Spinardi, G. (1995). Tacit knowledge, weapons design, and the uninvention of nuclear weapons. *American Journal of Sociology*, 44–99.

McCalman, I., & Pickering, P. (2010). *Historical reenactment: from realism to the affective turn*. London: Palgrave Macmillan.

Murphy, D. (2013). Hacking public memory: understanding the multiple arcade machine emulator. *Games and Culture*, *8*(1), 43–53.

Nguyen, L. T., & Kay, A. (2015). The cuneiform tablets of 2015. In *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* (pp. 297–307). ACM. Retrieved from http://dl.acm.org/citation.cfm?id=2814250

O'Reilly, T. (1998, November). The open-source revolution. *Release 1.0*, 3–26.

Rhodes, T. (2007, October 2). Best Little Emulator Ever Made! *The Escapist*. Retrieved from http://www.escapistmagazine.com/articles/view/video-games/issues/issue_117/2295-Best-Little-Emulator-Ever-Made

Rosenthal, D. S. H. (2015). *Emulation & Virtualization as Preservation Strategies* (report). The Andrew W. Mellon Foundation. Retrieved from https://mellon.org/resources/news/articles/emulation-virtualization-preservation-strategies/

Rosenthal, D. S. H. (2016, March 24). Long Tien Nguyen & Alan Kay's "Cuneiform" System. Retrieved April 3, 2017, from http://blog.dshr.org/2016/03/long-tien-nguyen-alan-kays-cuneiform.html

Rosenthal, D. S. H., Robertson, T. S., Lipkis, T., Reich, V., & Morabito, S. (2005). Requirements for Digital Preservation Systems: A Bottom-Up Approach. *D-Lib Magazine*, *11*(11). Retrieved from http://arxiv.org/abs/cs/0509018

Ross, S., & Gow, A. (1999). Digital Archaeology: Rescuing Neglected and Damaged Data Resources. A JISC/NPO Study within the Electronic Libraries (eLib) Programme on the Preservation of Electronic Materials [Research Reports or Papers]. Retrieved February 28, 2017, from http://eprints.gla.ac.uk/100304/

Shelemay, K. K. (2001). Toward an ethnomusicology of the early music movement: Thoughts on bridging disciplines and musical worlds. *Ethnomusicology*, *45*(1), 1–29.

Stevenson, M. (2016). The cybercultural moment and the new media field. *New Media & Society*, 1461444816643789. https://doi.org/10.1177/1461444816643789

Stuckey, H., & Swalwell, M. (2014). Retro-Computing Community Sites and the Museum. In M. C. Angelides & H. Agius (Eds.), *Handbook of Digital Games* (pp. 523–547). Hoboken, NJ: Wiley-IEEE Press.

Swalwell, M. (2009). Towards the preservation of local computer game software: Challenges, strategies, reflections. *Convergence*, *15*(3), 263–279.

van der Hoeven, J. R., Van Diessen, R., & van der MEER, K. (2005). Development of a Universal Virtual Computer (UVC) for long-term preservation of digital objects. *Journal of Information Science*, *31*(3), 196–208.

Wen, H. (1999, June 4). Why emulators make video-game makers quake. Retrieved April 12, 2017, from http://www.salon.com/1999/06/04/emulators/