Floating Time with QEMU at the Denver Art Museum

Overview

Emulation and virtualization provide an excellent method of preserving attributes of a software-based artwork for both exhibition and long term conservation. When done ethically and accurately, emulation can realize visual, aural, and interactive aspects of a software-based work of art long made non-functional as the technology to render it obsolete. Of course, making creating faithful versions of the original vision of the artist with the processing power and rendering capabilities of contemporary computers and displays is quite difficult.

The creation of an emulated version of Tatsuo Miyajima's 2000 software-based installation, *Floating Time -Marine Blue* served as an excellent case for exploring the command-line based emulation software QEMU. After many attempts with various other easier to use software emulation programs, the open source, command-line software QEMU created an emulation with the highest fidelity to the original artwork. This report can be seen as an introduction to QEMU and, in a broad sense, software emulation generally.

Harnessing the Flightless Bird

QEMU is commonly understood an acronym for Quick EMUlator. A graphic illustration of an emu in front of the letter "Q" is its application avatar and the open source software is is described as "a generic and open source machine emulator and virtualizer" on the QEMU main page. Now available via the command line in Linux,

Mac, and Windows, the application was originally programmed in 2003 by Fabrice Bellard in Linux. Among other open source tools, Bellard created the audiovisual transcoding, compression, and manipulation software FFmpeg, a powerful tool used by many in the gallery, library, archive, and museum (GLAM) community. Similar to FFmpeg, QEMU offers versatile, high-quality results and provides a great payoff for anyone seeking to use its bash script commands to accomplish tasks. Unlike FFmpeg, QEMU has parameters for a wide variety of computer components and operating systems. Not to diminish digital audiovisual transcoding and manipulation, but the fundamentals behind these tasks are relatively established compared to the myriad of variables underlying computer system emulation and virtualization.

The versatility of QEMU can make it very daunting to begin working with. In writing about my initial forays into using it, I can only capture my stumbles, frustrations, and ultimate results in trying to accomplish an extremely narrow emulation task; when compared to the multitude of possibilities with the software. After trying out other emulation solutions and learning to work with QEMU, it provided the best rendering of speed for an emulation "off the shelf" without very much tweaking. It became well worth the struggle of getting it to work. I hope my experience can help guide someone else taking their first steps with the software (for core the QEMU, step-by-step, skip to the section "Installing QEMU"). But first, why emulate and how does that differentiate itself from virtualization?

It is easy to confuse the terms "emulation" and "virtualization." Emulation describes the process of reproducing and attempting to imitate the behavior of a

computing environment or hardware. Virtualization describes simulating isolated computing environment(s) or hardware simulated from a single physical computer. For a more detailed description and diagrams visualizing the difference between emulation and virtualization, view the QSAN blog on the subject. The goal with QEMU in this situation, as explained in the following section, is to recreate a computing environment to allow for a software-based artwork to exhibit its original behavior as accurately as possible. So the process will be best described as emulation, even though the tools of virtualization will be employed.

Software emulation has its roots with video gaming enthusiasts seeking to replicate the time and technological constraints or an era or version of a beloved game or gaming system. It also has become very practical when working with servers to create virtual machines with different operating systems for testing and other tasks. It allows for running an operating system (OS) in another system in order to virtually mimic the functions or features of the former OS. The GLAM community has adopted emulation of operating systems and software in order to replicate or recreate a faithful engagement with interactive programs built at a specific technological time and place. For time-based media art, emulation is an excellent way of conserving or exhibiting an artwork without intervening or altering the original acquired elements.

The Artwork History, Conservation, and Exhibition at the DAM

The software-based installation, *Floating Time - Marine Blue* (2000) by Tatsuo Miyajima, was originally exhibited and acquired by the Denver Art Museum in 2003. Miyajima, born 1957, is a Japanese installation, sculpture, and performance artist whose

work is influenced by Buddhist philosophy and modern physics. As he explains in this 1997 interview on his website, "Zero Does Not Count," when working with LEDs in the late 1980s, Miyajima was excited by the technological and conceptual aspect of digital counters sequencing through numbers from '0' to '9.' He saw that "digital numbers have all ten numbers contained in one." This helped him form what he calls his core concepts: "keep changing, connect with everything, continue for ever." Many of his numerically based works, including *Floating Time*, randomly or sequentially cycle through the numbers '1' through '9' infinitely. As he explains in the "Zero Does Not Count" interview, the number zero, which represents the void or the pause in the cycle of life, gets removed from the sequence of numbers in order to draw attention to it. The Denver Art Museum commissioned the installation of an LED artwork of Miyajima's, *ENGI*, which is a series of 80 LED counters embedded in the walls of the atrium in their Hamilton building when it was constructed in 2006.

Floating Time - Marine Blue is a room size, software-based installation comprised of a video projection of animating numbers that sequence from "1" to "9" onto the floor of a gallery. Visitors experiencing the artwork enter the room where the different color integers scale in size and move horizontally, diagonally, and vertically across the floor of the room, and onto anyone walking through it. *Floating Time* stands out from other Miyajima installations as it does not employ LED technology, but rather a consumer-grade computer renders the animation and provides the projected image.

The animation itself was built with Java applets. These plug-ins now are obsolete. I arranged a conference call with Jonathan Farbowitz, Fellow in the Conservation

of Computer-Based Art at the Solomon R. Guggenheim Museum, for he, Kate Momaw, and I to discuss Java applets; something the Guggenheim has had a lot of experience with. Nearly all my understanding of Java applets came out of that conversation. Many artists gravitated toward using Java applets for their interactive and computer-based art in the late 1990s and 2000s. They are programs that work within a 'sandbox' run in a web browser upon installing Java from Oracle (originally Sun Microsystems). This allowed content to be easily render-able and accessible cross platform and on whatever browser the Java Virtual Machine (JVM) was installed. Their flexibility across platforms and browsers also made Java applets vulnerable to being exploited by malicious actors on the web. Due to this, they are no longer supported and will not function in modern browsers.

The original exhibition computer hard drive, as well as CD-ROMs containing both the artwork and supporting operating system (OS) files were disk imaged in 2018 by Eddy Colloton, Assistant Conservator for Electronic Media at the DAM. These were the first steps in the conservation of *Floating Time*. All of the acquired elements, from the Dell computer, to the installation software, to paper instructions describing how to launch the animation during exhibition were given unique IDs and catalogued thoroughly upon *Floating Time's* acquisition in 2003. It was one of nearly 80 software-based works that were part of a two-year IMLS funded electronic media conservation grant, along with hundreds of other time-based media artwork (see Eddy's excellent blog post for details about the entire project). I was brought on by Kate Moomaw, Associate Conservator for Modern and Contemporary Art and the Director of

Conservation at the DAM, Sarah Melching, to overlap with Eddy before he moved on to a new position at the Hirshhorn Museum and Sculpture Garden. My tasks were to finish the grant period and then work on complex time-based artworks scheduled for exhibition in 2019. One of these works was *Floating Time*. Once I added the preserved digital assets of the work into the museum's digital repository, I was able to view the animation on the original machine. After video recording the playback of the software, I then needed to see whether a successful emulation of it was possible.

My initial emulation attempts, while unsatisfactory, proved that emulation of the nearly 20-year-old piece was possible. I tried many different options for emulation. I used VMWare and VirtualBox on both a Mac tower running OS 10.11.6 as well as a Dell laptop running Windows 10. Working with the Technology staff at the Denver Art Museum, specifically Derek Sava and David Karr, I used Windows Virtual PC and VirtualBox on a computer with Windows 7, using the Microsoft created emulator XP Mode, which was part of initial releases of Windows 7 (and can be downloaded if you have a Windows 7 license key). Even the best case scenario emulation, which was Windows XP Mode via VirtualBox, required dumbing down the virtual machine settings to a level that the VirtualBox software itself deemed unsafe. It was neither an ideal nor a stable emulation.

For *Floating Time*, the nature of how the numbers generate to create an experience for the viewer of the artwork is also critical to the identity of the work. However, without drastically reducing the performance of the virtual machine created to play back the software-based animation, everything animated ridiculously fast. Like a

hair raising, final level of a video game. The speed of the work was key to the experience of the installation. It likely goes without saying that computers have become exponentially faster and more powerful than the single processor, Dell Dimension L600cx with 256MB of RAM used to exhibit *Floating Time* in 2003.

So I asked for some help and spoke with Jonathan Farbowitz as well as Mark Hellar, owner of Hellar Studios and technology consultant for SFMOMA among others. Mark along with Martina Haidvogel, Associate Media Conservator, and Layna White, Head of Collections Information and Access for SFMOMA were in Denver for the Museums and Computer Network conference to speak about their use of the SFMOMA Media Wiki for documentation of media artworks. Based on what I described to Jonathan and Mark, they both independently recommended using QEMU on Linux. In a follow up meeting with the Technology department at the DAM, Kate and I proposed the need for QEMU on a Linux machine to Technology department of the DAM. Because they preferred to exhibit with Dell machines, and after some research, Technology provided me with a Dell OptiPlex installed with Ubuntu and QEMU to work with.

Installation of QEMU

Like most command-line interface (CLI) software, the easiest way to install QEMU is through packages available via Linux distributions or through package managers like Homebrew on the Mac.[1] The Download page on the QEMU website (https://www.qemu.org/download) provides the latest available packages, as well as the commands for the Terminal application to install them for Linux and Mac. QEMU for

[1] I didn't work with the Mac/Homebrew version of QEMU until late in my process of trying to accomplish the emulation, but it became invaluable for testing and working with newer versions of QEMU.

Windows and the executable installers for Windows 32 and 64 bit are created and maintained by Stefan Weil, who runs the site https://qemu.weilnetz.de. He offers this disclaimer on the download page "QEMU for Windows is experimental software and might contain even serious bugs, so use the binaries at your own risk." I wanted to stick to working in Linux based on how QEMU was recommended to me.

Once it is installed, you can check what version of QEMU you are running by adding the flag `--version` after the system you want to work with in bash script, i.e.:

- `qemu-system-i386 --version`

There are various systems or virtual computers you can work with in QEMU for an emulation. I will go into QEMU systems in a moment but suffice it to say, when working with the software, you need to tell QEMU the type of computer system you would like to invoke. But before any computer system can be booted up, a disk image needs to be created and an operating system software needs to be installed into it.

I'll be using the term 'virtual machine' or VM to mean the entity comprised of the computer hardware, operating system software, applications, plugins, and settings installed on it. The artwork, or any digital object you would like to emulate, resides along with any supporting files contained within an installed OS. As with a physical computer, all software gets installed, downloaded, or copied onto a hard drive. In the case of QEMU, the volume of this virtual hard drive is a disk image. Most commonly, disk images are a software copy of a physical volume, i.e. an optical disk or a connected external drive (https://en.wikipedia.org/wiki/Disk_image). In this case, the volume is created first, and then files necessary to create a virtual hard drive are loaded into it.

That disk image is then 'booted up' into a virtual computer system available in QEMU and further parameters can be added to this virtual machine as you would install graphics cards or expand RAM on a physical computer.

The operating system for *Floating Time* was Windows 2000. From the original acquisition, the Denver Art Museum had original Equipment Manufacturer (OEM) installation disc of Windows 2000. Using this install disc for the emulation had two issues: first, it was a Japanese language version of the OS; and second, and more importantly, the OEM installation package would not initiate the installation process unless it recognized it was being installed in a Dell machine. I sought out others options for Windows 2000 installer and downloaded one from WinWorld,

To create a virtual machine that works in QEMU, you will need to create a disk image using the software. The web-based manual for QEMU has a "Quick start for disk image creation" with an example of the command to invoke. From the steps outlined in it, I used the following command to create the disk image:

- `qemu-img create win2k.img 20G`

The `qemu-img` command creates the image file with the name you give it at the given file size. The `win2k` before the .img file type is just the name I gave my image file. The final part of the command, `20G`, sets the size of the volume to 20 gigabytes. As the Quick start for disk image creation in the QEMU manual explains, the size is defined in kilobytes, "can add an M suffix to give the size in megabytes and a G suffix for gigabytes." I could have chosen a smaller size, but a volume too small did cause initial

problems with the Windows 2000 installation[2]. This quick start bash command line string creates a 'raw' disk image, which is the default format created if none other is specified. There are many disk image formats you can create to work with, including native QEMU qcow formats (https://en.wikibooks.org/wiki/QEMU/Images), but I stuck to the raw default.

Now with my destination disk image created, I used the following command to install the software:

- `qemu-system-xi386 -hda win2k.img -boot d -cdrom /[USER]/EN_WIN2000_PRO.ISO`

This command launches QEMU, then the installer .iso disk image I downloaded from WinWorld. When everything runs correctly, I simply go through the same Windows 2000 installation process I would if my VM was a physical hard drive. Once that process was complete, I can simply launch my version VM with this command:

- `qemu-system-xi386 -hda win2k.img`

When successfully installed, the emulated operating system will retain all the files loaded into it as well as any OS specific settings into your disk image. A final note on the above bash script assumes the disk image file with your installed hard drive resides at the user level you are logged into. Without specifying a different file path, the root user level is the default location to execute and save files with bash scripts. Additional files

[2] This may have had something to do with the Windows 2000 disk full problem cited in the QEMU manual. I should note, I came to the QEMU manual quick start instructions late in my process of getting running with QEMU. Instead I worked with instructions on this older, but aptly titled post "How To Install And Configure QEMU In Ubuntu," which provides a 20 gigabyte disk image in its example code.

you want to load into the virtual machine need to reside at this level unless you specify where they are located.

The machine running QEMU is considered the host. Its own hardware characteristics are inherent in the specifications of the installed processor, random access memory (RAM) installed, and the computer's graphics card, which sort of acts like the processor for the computer's display port. The software for the guest machine is the installed OS and any files contained within that system. The parameters for the guest machine are defined by both the hardware you are virtualizing and the operating system you want your VM to emulate. QEMU supports a variety of hardware for emulation and virtualization. It can be used to either virtualize hardware or emulate an operating system. Available platforms for virtualization and emulation can be found on the Documentations/Platforms page on the QEMU Wiki (https://wiki.qemu.org/Documentation/Platforms). When you install QEMU, all the supported platforms, or computer systems, are available. This diversity of options both make it a difficult software to fully grasp and is what makes it such a powerful and popular emulator. I would only be working in a system made for Windows computers from the late 1990s to early 2000s as described in the QEMU Wiki Documentation pages as i386/x86: (https://wiki.qemu.org/Documentation/Platforms/PC). The QEMU command to call up this system in QEMU is either:

$ `qemu-system-i386` OR $ `qemu-system-x86_64`

The choice of what system to use depends on the time period when the processor you are seeking to emulate was manufactured. The i386, also known as 80386 was a 32

bit microprocessor released in 1985 and stopped being used by Intel in 2006 (https://en.wikipedia.org/wiki/Intel_80386. The x86, also known as 8086, was a family of processors introduced in 1985 as 32 bit, and 2003 as 64 bit (https://en.wikipedia.org/wiki/X86). Because Windows 2000 is 32 bit, the only version I worked with was qemu-system-i386. As far as 32 vs 64 bit goes, the higher number reflects the exponentially higher amount of data a computer's processor can handle. Software needs to be written to take advantage of a processor's capacity, which includes operating systems, applications, and the files created by them. A 64 bit system can run 32 bit software, but it can't go the other way around. A more detailed, byte-laden explanation can be found on the Geeks for Geeks website.

Guides for Code and Rabbit Holes

QEMU has a menu in the command line, which has thankfully been copied and formatted into a website I previously noted (https://qemu.weilnetz.de/doc/qemu-doc.html). The web-based manual is as close to an official manual as it gets outside of the command line. It is linked to from the official QEMU Wiki and is updated regularly with new releases of the software. The QEMU web-based manual does a good job of explaining the various tools and components for each system. However, as describes features for every flavor, system, and option of QEMU, it is very easy to get lost in. Also, the manual itself doesn't always let you know that a feature is no longer supported. For example, when choosing the QXL graphics card for my VM in the command line, the command line provides an error message letting you know it is no longer supported, but the web-based manual doesn't.

Something like that isn't a huge deal, especially when there are a variety of other options to choose from, but it indicates the trial and error nature of working with the program.

A great aspect of working with QEMU is that it is that there are many people using who post questions and solutions online. These third party resources can cause frustration, however, as they may contain legacy examples of code or present options that are no longer the most efficient way to perform a task. The Ubuntu forums, Stack Overflow, and other sites have many users inquiring about errors they are encountering. Sometimes these explanations for the errors led me down a rabbit hole of trying to fix the specific error listed in the message provided in the command line by QEMU.

To help illustrate the feedback process when trying to execute a QEMU bash command string, I'm going to use the "Suggested command line code" listed on the Windows 2000 page in the QEMU Wiki. First, when if I copy and paste the suggested command line into Terminal:

- `qemu-system-i386 -hda` [your disk image with OS installed]`.img -m 384 -boot c -vga cirrus -net nic,model=rtl8139 -net user -usb -soundhw sb16 -localtime`

When I hit enter to execute the code, it will prompt this error "`-localtime: invalid option`" and it will not execute the code and not open the program. When you remove the `-localtime` flag, the code executes, and then QEMU opens. That's a straightforward example of the command line providing feedback. However, while removing the `-localtime` flag allowed the program to open, a new error appears in Terminal:

- `WARNING: Image format was not specified for 'win2k.img' and probing guessed raw. Automatically detecting the format is dangerous for raw images. Specify the 'raw' format explicitly to remove the restrictions.`

Not wanting to have QEMU run a VM with what it is referring to as a potentially dangerous error, I copied and pasted the error in a search engine. Sorting through the resulting scenarios with users seeking a solution to the error, I found that adding a comma and then `format=raw` just after my disk image name defines the disk as being in the raw format. To be honest, I'm not sure why the format I thought I created as 'raw' by default is not being read as raw. This may be due to the code evolving since the sample Windows 2000 code page was edited in 2017.

In the process of searching through forum posts with other parts of the sample code, whenever a user would post code for the hard drive using the `-hda` flag, other more experienced users pointed out that `-drive` is a more accurate and up to date way to flag the VM image to launch as the hard drive. With those three new changes, here is what became my adjusted version of the suggested Windows 2000 code:

- `qemu-system-i386 -drive win2k.img,format=raw -m 384 -boot c -vga cirrus -net nic,model=rtl8139 -net user -usb -soundhw sb16`

To explain some of the flags for components in the above code, here's a quick summary:

- `-drive` followed by the file name of the disk image you created and installed the guest operating system onto, tells QEMU this is the hard drive to boot
- `-m` followed by a number specifies the installed RAM for your VM;

- -boot allows you to add a volume, i.e. a floppy disk drive or CD ROM. [*In this case the flag is booting the graphics card vga, which is no longer needed to specify a graphics card].*
- `-vga` to specify a graphics card *[options like 'cirrus' can be found in the manual and depend on the computer system you are trying to emulate up as well as or extensions you may choose to add]*
- `-net` is the flag for network; `nic` is short for network interface card `,model=` is where you can specify the model card *[options can be found in the manual, [search for "-net nic"] The network interface card also depends on the system you are trying to emulate]*
- `-usb` enables the usb driver *[if the guest OS is not doing that by default]*
- `-soundhw` specifies sound hardware or sound card *[options can be found in the manual and depend on the system you are trying to emulate]*

The explanations I provided came from a mix of reading the manual, performing web searches for answers, and responses from more experienced QEMU users.

Variations and Emulation for Conservation

As I worked on the code to emulate *Floating Time*, I found that I didn't need any sound, any internet access, or to specify a USB connection. By adding specifications for the graphics card as well as for the installed RAM, this was the resulting bash string:

- `qemu-system-i386 -drive file=win2k.img,format=raw -m 256 -vga cirrus`

The original machine had 256 MB or RAM installed onto it. The Cirrus graphics card is the default for QEMU, but I want to specify it to launch the VM that will be part of the conservation treatment. This specificity helps for future treatments. Once I set the display resolution within the OS, a very similar experience to what I saw on the original exhibition computer emerged. Setting the display parameters simply comes by changing the settings within Windows 2000 as I would on a physical computer. Under Control Panel, I then chose Display and then clicked on the Settings tab. There, I chose 1024 x 768 aspect ratio with the highest color resolution available, which was 24 bit.

There is no way to load files into the VM similar to enabling "Guest Additions" in VirtualBox and dragging the files into the guest OS. Adding the artwork and its supporting files to the VM meant loading them from a booted volume, such as a virtual CD-ROM, by downloading them from the web, or copying them into the guest OS via a shared network connection. Getting the files into the VM ended up being the most difficult part of building a workable emulation for me. I had difficulty with getting the network interface card working consistently in order to connect to a shared network folder with the files. When I created disk images of the files used to boot into *Floating Time*, the Windows 2000 guest OS wouldn't recognize them. After a lot of trial and error, I ended up using a disk image created as a preservation file from a physical CD-ROM originally acquired with *Floating Time*. It booted into the VM normally when I using the flag `-cdrom`[3]. From there I copied files into the same locations and the

[3] I also placed the disk image with the assets in the same folder as my VM disk image.

original Windows 2000 computer. Once the files were in place, *Floating Time* launched without a problem.

The speed of the QEMU emulation was nearly identical to the original. This result was a very noticable difference every other method of emulation I tried previously. The rendering of the animation, however, was not quite on par with the original. Further work was needed to make it match. The biggest change ended up being the version of QEMU itself. When I first started with QEMU in November 2018 was version I was working with was 2.0.0. This was what available in the package from the Ubuntu 14.04 OS installed on the desktop (it turns out this is the same for the 18.04 installer I used later). While I was working on the emulation in December 2018, I noticed version 3.1 was made available. I thought I could install the latest version by the command `apt-get upgrade qemu`, but if the new version of the software wasn't loaded onto the package installer, I couldn't get the upgrade via that method.

As it turned out, if I wanted a newer version of the program on Ubuntu, I needed to manually install it. This involves manually removing previous versions of QEMU and running the `configure` and `make` commands in order to first compile an installer and then run it. This process can take over an hour to do. Luckily, you probably won't have to do that, but I used this Ubuntu forum post as a guide. One thing to note is that the Homebrew package for QEMU did have a much more recent version of the program. I discovered this when version 4.0 came out in March of 2019. With version 3.0, I noticed a much smoother rendering of the animation of *Floating Time*. Finally, when running QEMU 4.0, and tweaking the virtual RAM settings within the Windows 2000 guest, I

was able to achieve an emulation very close to the video documentation of the *Floating Time* on the original machine I was using as a reference. This worked in both in Mac OS and Linux using QEMU 4.0.

The final emulation has ended up serving as a potential reserve exhibition copy of the artwork, but also is now a record of its conservation treatment. The process of creating an emulation brought out elements of the software-based artwork that were key to make it function. Only through working with the digitally preserved assets could the artwork's Java Applet distinctiveness be discovered. On top of that the conservation and and digital preservation process has created digital elements that can be plugged into future exhibitions of the artwork; and potentially more effective releases of QEMU or another emulation solution. After performing the conservation treatment for *Floating Time*, both the Denver Art Museum and I have a much better understanding of how to implement emulation solutions effectively, while keeping the integrity and identity of the work intact. This will likely inform other software-based work in the DAM's collection and help provide a methodology for emulation solutions in the future.

QEMUprovisor?[4]

As noted, many guides and helpful forum posts are out there for problem solving with QEMU. But there is nothing like ffmprovisor for it. For those unfamiliar, ffmprovisor is a GitHub webpage with a list of command line FFmpeg bash script

[4] Since I originally wrote this, Ethan Gates, Software Preservation Analyst for Yale University Laboratory, unbeknown to me launched QEMU QED. This site utilizes the same code as ffmprovisor and is hosted on the Github page of Yale's Emulation As A Service (Eaasi). Launched for a hackathon project during the 2019 iPres conference, it is a resource for QEMU bash script recipes, such as type of collaboration I discuss in this section.

commands that act a sort of recipe to accomplish a particular video and audio processing or conversion task. It was built by audiovisual archivists with preservation and archiving tasks in mind. It was started during an event held every year at the Association of Moving Image Archivists (AMIA) conference called 'AMIA HackDay.' This event brings those both savvy and interested in coding together to collaboratively create something for the archival community. This day is modeled on events held by open source software designers, users, and programming enthusiasts. The resulting page is hosted within the AMIA Open Source Committee GitHub repository.

I'm not sure a similar type resource would be possible with QEMU. The main reason for my thinking this lies with the variety of operating systems that could be emulated and the number of guest computer systems available within the application. Specifically creating guides for `qemu-system-i386` and `qemu-system-x86-64` for Windows emulation, or `powerpc` (for pre-Intel Mac) would be a worthy endeavor. Just to be realistic, creating an emulation is also a more nuanced and specific task than relatively established preservation tasks like video transcoding or manipulation. Creating a bash script 'recipe' for one situation likely will not work in another. Additionally, emulation of software for exhibition or conservation happens much, much less frequently than exhibition or conservation of video or audio. How many bash scripts could be 'hacked' by seasoned conservators, archivists, and emulation enthusiasts if something similar to a Hack Day event was held? How many scripts created for an index of bash script recipes would stay relevant as QEMU continually updates and new

efficiencies are developed? Moreover, how often would a resource like this be used for real world tasks within a galleries, museums, archives, or libraries?

When creating preservation, access, or exhibition video files, I continually refer to ffmprovisr. So many solutions for what I cam seeking to do exist there. Also, the ability to link out to specific URLs of a bash script recipe becomes excellent way to source steps taken for documentation within a treatment. These URLs also offers a breadcrumb trail to verify those bash scripts in the future when the page is managed by the audiovisual archival community. As I waded my way into the murky text-based waters of QEMU, I would definitely have appreciated some clearly marked signposts like ffmprovisor. Something similar to ffmprovisor could be developed for QEMU. It would take time and a sustained interest to make it happen. To make a high-quality, but difficult to begin to use, emulation software like QEMU viable as an option in software-based art conservation, a resource like it might be necessary. Of course it would all depend on interest and usage of QEMU itself as a choice for emulation. Hopefully, this offering of my first experience working with QEMU to emulate *Floating Time - Marine Blue* can serve as a sort of orientation for those interested in trying out emulation in the command line. It's worth a try to see the results yourself.

Thanks to everyone mentioned above. Additional thanks to Jonathan Farbowitz for emphasizing the need to clarify the terms emulation versus virtualization. This post was updated in September 10, 2019 to reflect that change.

A big, belated thanks to Ethan Gates for starting QEMU QED during the 2019 iPres conference and putting a link to this paper on the main page of that resource during the conference.. Ethan was also assisted me when I was seeking out a Windows 2000 installer for my VMs at the DAM. This post was updated September 24, to acknowledge Ethan and QEMU QED.